NATIONAL TECHNICAL UNIVERSITY OF ATHENS

SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING
DIVISION OF COMPUTER SCIENCE

From Circuits to SoC Processors: Arithmetic Approximation Techniques & Embedded Computing Methodologies for DSP Acceleration

Ph.D. Dissertation

Vasileios K. Leon

Athens
October 2022

From Circuits to System-on-Chip Processors: Arithmetic Approximation Techniques & Embedded Computing Methodologies for Digital Signal Processing Acceleration

Vasileios Leon

Examination Committee
Prof. Kiamal Pekmestzi (Supervisor), *National Technical University of Athens*
Prof. Dimitrios Soudris, *National Technical University of Athens*
Associate Prof. Georgios Goumas, *National Technical University of Athens*
Prof. Dionysios Reisis, *National and Kapodistrian University of Athens*
Prof. Apostolos Dollas, *Technical University of Crete*
Prof. Dimitris Gizopoulos, *National and Kapodistrian University of Athens*
Prof. Antonis Paschalis, *National and Kapodistrian University of Athens*

National Technical University of Athens
School of Electrical & Computer Engineering
Division of Computer Science
Microprocessors and Digital Systems Laboratory

From Circuits to SoC Processors:
Arithmetic Approximation Techniques
& Embedded Computing Methodologies
for DSP Acceleration

Ph.D. Dissertation
of
Vasileios Leon

Supervising Committee: Kiamal Pekmestzi
Dimitrios Soudris
Georgios Goumas

Approved by the Examination Committee on October 10, 2022.

....
Kiamal Pekmestzi
Professor NTUA

....
Dimitrios Soudris
Professor NTUA

....
Georgios Goumas
Assoc. Professor NTUA

....
Dionysios Reisis
Professor NKUA

....
Apostolos Dollas
Professor TUC

....
Dimitris Gizopoulos
Professor NKUA

....
Antonis Paschalis
Professor NKUA

Vasileios K. Leon

PhD, Electrical & Computer Engineering, National Technical University of Athens
Diploma, Computer Engineering & Informatics, University of Patras

Βασίλειος Κ. Λέων
Διδάκτωρ Ηλεκτρολόγος Μηχανικός & Μηχανικός Υπολογιστών Ε.Μ.Π.
Διπλωματούχος Μηχανικός Η/Υ & Πληροφορικής Πανεπιστημίου Πατρών

Στους γονείς μου

Abstract

The recent end of Dennard's Scaling and the declining Moore's Law have signified a new era for the computing systems. Power efficiency has now become a critical factor for both cloud and edge computing. Concurrently, the rapid growth of compute-intensive applications from the Digital Signal Processing (DSP) and Artificial Intelligence (AI) domains challenges the resources of computing systems. As a result, the computing industry is forced to find alternative design approaches and computing platforms to sustain increased power efficiency, while providing sufficient performance. Among the examined solutions, *Approximate Computing*, *Hardware Acceleration*, and *Heterogeneous Computing* have gained great momentum. Approximate Computing is a novel design paradigm that exploits the inherent error resilience of DSP/AI applications to deliver gains in power, area, and/or performance by reducing the quality of the results. Hardware Acceleration refers to the execution of demanding computational tasks on specialized hardware, such as the Application-Specific Integrated Circuits (ASICs) and the Field-Programmable Gate Arrays (FPGAs), rather than general-purpose processors. Finally, Heterogeneous Computing refers to versatile processing architectures, such as the Vision Processing Units (VPUs), which integrate more than one type of processor and various memory technologies.

In this Dissertation, we introduce design solutions and methodologies, built on top of the preceding computing paradigms, for the development of energy-efficient DSP and AI accelerators. In particular, we adopt the promising paradigm of Approximate Computing and apply new approximation techniques in the design of arithmetic circuits. Based on our methodology, these arithmetic approximation techniques are then combined with hardware design techniques to implement approximate ASIC- and FPGA-based DSP and AI accelerators. Moreover, we propose methodologies for the efficient mapping of DSP/AI kernels on distinctive embedded devices, such as the new space-grade FPGAs and the heterogeneous VPUs. On the one hand, we cope with the decreased flexibility of the space-grade technology and the technical challenges that arise in new FPGA tools and devices. On the other hand, we unlock the full potential of heterogeneity by surpassing the increased hardware complexity and exploiting all the diverse processors and memories.

In more detail, the proposed arithmetic approximation techniques involve bit-level optimizations, inexact operand encodings, and skipping of computations, while they are applied in both fixed- and floating-point arithmetic. To increase the design space and extract the most efficient solutions, we also conduct an extensive exploration on

combinations among the approximation techniques. Furthermore, we propose a low-overhead scheme for seamlessly adjusting the approximation degree of our circuits at runtime. In comparison with state-of-the-art designs, the proposed arithmetic circuits feature a very large approximation space, i.e., a wide range of approximation configurations, which enable to maximize the resource gains for a given error constraint. Our techniques induce a mean relative error of up to ~2%, i.e., typical error values for approximate circuits. The most prominent approximate circuits of the Dissertation form a high-resolution Pareto front in a comparative evaluation involving state-of-the-art designs of the literature, and they deliver up to 63% better energy consumption. Finally, our runtime-configurable circuits exhibit a small area overhead of ~3% compared to the accurate design, and they provide ~1.5× less energy gains than their respective design-time counterparts with fixed approximation. Nevertheless, they can dynamically change the approximation degree, namely, the accuracy of the calculations, while they still attain remarkable energy gains versus the accurate circuit and state-of-the-art approximate circuits. At the accelerator level, we develop a plethora of approximate kernels for 1D/2D signal processing and Convolutional Neural Networks (CNNs). The experimental results show that we achieve small relative errors for classic DSP calculations and 0%–5% accuracy loss in CNNs for various arithmetic formats, while providing up to 70% area and energy savings.

Regarding the DSP acceleration on new space-grade FPGAs, we apply our methodology to efficiently map computer vision algorithms onto the radiation-hardened NanoXplore's FPGAs. In the end, we achieve balanced resource utilization, which is comparable to that of well-established FPGA vendors. Moreover, the throughput is sufficient (e.g., up to 10 FPS for feature detection on MPixel images), considering the performance requirements of vision-based space applications. In terms of Heterogeneous Computing, we accelerate custom DSP kernels, a sophisticated computer vision pipeline, and a demanding CNN with ResNet-50 backbone on Intel's Myriad VPUs. The proposed methodology and embedded design techniques provide speedups up to 20× for classic DSP on Myriad 2, while the power lies around 1W. The CNN is accelerated on Myriad X with 2W, achieving ~8.5× and ~1.7× better performance-per-Watt than the ARM CPU and the Jetson Nano GPU, respectively.

Keywords: Approximate Computing, Approximation Techniques, Arithmetic Circuits, Computer Arithmetic, Hardware Design, Hardware Accelerators, ASIC, FPGA, VPU, SoC, Heterogeneous Computing, Embedded Systems, Space-Grade, Digital Signal Processing, Computer Vision, Convolutional Neural Networks.

Περίληψη

Το πρόσφατο τέλος της Κλιμάκωσης του Dennard και η φθίνουσα πορεία του Νόμου του Moore έχουν σηματοδοτήσει μια νέα εποχή για τα υπολογιστικά συστήματα. Η κατανάλωση ισχύος αποτελεί πλέον έναν κρίσιμο παράγοντα, τόσο για το υπολογιστικό νέφος όσο και για υπολογισμούς στην άκρη του δικτύου. Ταυτόχρονα, η ταχεία ανάπτυξη απαιτητικών εφαρμογών από τους τομείς της Ψηφιακής Επεξεργασίας Σήματος (DSP) και της Τεχνητής Νοημοσύνης (ΑΙ) δημιουργεί προκλήσεις στους πόρους των υπολογιστικών συστημάτων. Ως αποτέλεσμα, η βιομηχανία των υπολογιστών υιοθετεί εναλλακτικές μεθόδους σχεδίασης κυκλωμάτων και συστημάτων, ώστε να διατηρήσει χαμηλή κατανάλωση ισχύος, παρέχοντας όμως και επαρκή ταχύτητα. Ανάμεσα στις λύσεις που εξετάζονται, ο *Προσεγγιστικός Υπολογισμός* εκμεταλλεύεται την εγγενή ανθεκτικότητα σε σφάλματα των DSP/ΑΙ εφαρμογών ώστε να προσφέρει κέρδη σε πόρους μειώνοντας την ποιότητα των αποτελεσμάτων. Η *Επιτάχυνση Υλικού* αναφέρεται στην εκτέλεση απαιτητικών υπολογιστικών εργασιών σε εξειδικευμένο υλικό, όπως τα Ολοκληρωμένα Κυκλώματα Ειδικής Εφαρμογής (ASICs) και οι Συστοιχίες Επιτόπια Προγραμματιζόμενων Πυλών (FPGAs). Τέλος, ο *Ετερογενής Υπολογισμός* αναφέρεται σε ευέλικτες αρχιτεκτονικές επεξεργασίας με πολλαπλούς τύπους επεξεργαστή και μνήμης, όπως οι Μονάδες Επεξεργασίας Όρασης (VPUs).

Στην παρούσα Διατριβή, εισάγουμε σχεδιαστικές λύσεις και μεθοδολογίες βασισμένες στα προαναφερθέντα πρότυπα σχεδίασης, με στόχο την ανάπτυξη ενεργειακά αποδοτικών επιταχυντών υλικού. Σχετικά με τον Προσεγγιστικό Υπολογισμό, εφαρμόζουμε νέες τεχνικές προσέγγισης στη σχεδίαση αριθμητικών κυκλωμάτων. Οι τεχνικές αυτές συνδυάζονται με βάση τη μεθοδολογία μας με κλασσικές τεχνικές σχεδίασης, ώστε να υλοποιήσουμε προσεγγιστικούς DSP και ΑΙ επιταχυντές σε ASIC και FPGA. Επιπλέον, προτείνουμε μεθοδολογίες για την αποτελεσματική αποτύπωση DSP/ΑΙ πυρήνων πάνω σε ιδιόμορφες ενσωματωμένες συσκευές, όπως τα νέα FPGAs διαστημικού βαθμού και οι ετερογενείς VPUs. Όσον αφορά τα FPGAs, αντιμετωπίζουμε τις τεχνικές προκλήσεις που προκύπτουν κατά τη χρήση νέων εργαλείων, ενώ για τις VPUs, ξεκλειδώνουμε όλες τις δυνατότητες της ετερογένειας, ξεπερνώντας την αυξημένη πολυπλοκότητα υλικού και αξιοποιώντας όλους τους διαφορετικούς πόρους.

Οι προτεινόμενες τεχνικές αριθμητικής προσέγγισης περιλαμβάνουν βελτιστοποιήσεις σε επίπεδο δυαδικού ψηφίου, μη ακριβείς κωδικοποιήσεις τελεστών, και παράλειψη υπολογισμών, ενώ εφαρμόζονται σε αριθμητική τόσο σταθερής όσο και κινητής υποδιαστολής. Για να αυξηθεί ο χώρος σχεδίασης και να εξάγουμε τις πιο αποτελεσματικές

λύσεις, πραγματοποιούμε επίσης μια εκτενή εξερεύνηση πάνω στους συνδυασμούς των τεχνικών. Επιπλέον, προτείνουμε ένα σχήμα χαμηλής επιβάρυνσης για την απρόσκοπτη ρύθμιση του βαθμού προσέγγισης των κυκλωμάτων κατά το χρόνο εκτέλεσης. Σε σύγκριση με σημαντικά κυκλώματα της βιβλιογραφίας, οι προτεινόμενες λύσεις διαθέτουν πολύ μεγαλύτερο χώρο προσέγγισης (ευρύτερο φάσμα προσεγγίσεων), επιτρέποντας τη μεγιστοποίηση των κερδών σε πόρους για έναν δεδομένο περιορισμό σφάλματος. Οι τεχνικές μας προκαλούν ένα μέσο σχετικό σφάλμα έως και ~2%, δηλαδή τυπικές τιμές σφάλματος προσεγγιστικών κυκλωμάτων. Τα πιο εξέχοντα προσεγγιστικά κυκλώματα της Διατριβής σχηματίζουν ένα σύνορο Pareto υψηλής ανάλυσης στη συγκριτική αξιολόγηση με σημαντικές εργασίες της βιβλιογραφίας, προσφέροντας έως και 63% καλύτερη κατανάλωση ενέργειας. Τέλος, τα κυκλώματα που μπορούν να ρυθμίσουν δυναμικά την προσέγγιση, έχουν αυξημένη επιφάνεια κατά ~3% σε σύγκριση με το ακριβές κύκλωμα, και παρέχουν ~1.5× λιγότερα κέρδη ενέργειας από τα αντίστοιχα κυκλώματα με σταθερή προσέγγιση. Όμως, έχουν τη δυνατότητα να αλλάζουν την ακρίβεια των υπολογισμών, ενώ εξακολουθούν να προσφέρουν αξιοσημείωτα ενεργειακά κέρδη έναντι του ακριβούς κυκλώματος και κυκλωμάτων της βιβλιογραφίας. Σε επίπεδο επιταχυντή, αναπτύσσουμε μια πληθώρα από προσεγγιστικούς πυρήνες για επεξεργασία σημάτων/εικόνων και Συνελικτικά Νευρωνικά Δίκτυα (CNNs). Με βάση την πειραματική ανάλυση, τα σφάλματα είναι μικρά σε κλασικούς DSP υπολογισμούς και η απώλεια ακρίβειας κυμαίνεται ως 5% στα νευρωνικά δίκτυα, ενώ επιτυγχάνεται έως και 70% εξοικονόμηση επιφάνειας και ενέργειας.

Σχετικά με τα νέα FPGAs διαστημικού βαθμού, εφαρμόζουμε τη μεθοδολογία μας για την αποτελεσματική απεικόνιση αλγορίθμων υπολογιστικής όρασης στα ανθεκτικά-σε-ακτινοβολία FPGAs της NanoXplore. Στο τέλος, επιτυγχάνουμε ισορροπημένη χρήση πόρων, η οποία είναι συγκρίσιμη με αυτή των καθιερωμένων προμηθευτών FPGAs. Επιπλέον, η ταχύτητα είναι επαρκής (π.χ., έως και 10 FPS για την ανίχνευση χαρακτηριστικών σε MPixel εικόνες), λαμβάνοντας υπόψη τις απαιτήσεις απόδοσης των διαστημικών εφαρμογών. Σχετικά με τον Ετερογενή Υπολογισμό, επιταχύνουμε DSP πυρήνες, μια ακολουθία αλγορίθμων υπολογιστικής όρασης, και ένα απαιτητικό CNN στις Myriad VPUs της Intel. Οι προτεινόμενες μεθοδολογίες και τεχνικές ενσωματωμένης σχεδίασης παρέχουν επιτάχυνση έως και 20× σε κλασικούς DSP υπολογισμούς στη Myriad 2 με κατανάλωση ισχύος 1W. Το CNN επιταχύνεται στη Myriad X με 2W, προσφέροντας ~8.5× και ~1.7× καλύτερη απόδοση-ανά-Watt από τον επεξεργαστή γενικού-σκοπού ARM και τον επεξεργαστή γραφικών Jetson Nano, αντίστοιχα.

Λέξεις Κλειδιά: Προσεγγιστικός Υπολογισμός, Τεχνικές Προσέγγισης, Αριθμητικά Κυκλώματα, Αριθμητική Υπολογιστών, Σχεδίαση Υλικού, Επιταχυντές Υλικού, Ετερογενής Υπολογισμός, Ενσωματωμένα Συστήματα, Τεχνολογία Διαστήματος, Ψηφιακή Επεξεργασία Σήματος, Υπολογιστική Όραση, Συνελικτικά Νευρωνικά Δίκτυα.

Contents

List of Figures

6. Cooperative Approximation: Combination of Arithmetic Encodings

7. Approximate DSP & AI Hardware Accelerators

8. DSP Acceleration with New Space-Grade FPGA Devices & Tools

9. DSP & AI Acceleration on Heterogeneous Multi-Core SoCs

List of Tables

List of Abbreviations

AI	Artificial Intelligence
ALU	Arithmetic Logic Unit
ANN	Artificial Neural Network
ASIC	Application-Specific Integrated Circuit
BER	Bit Error Rate
BRAVE	Big Re-programmable Array for Versatile Environments
CER	Correct Edge Ratio
CFA	Circuit Functional Approximation
CIF	Camera Interface
CKG	Clock Generator
CMB	Conventional Modified Booth
CMX	Connection Matrix
CNN	Convolutional Neural Network
CORDIC	Coordinate Rotation Digital Computer
COTS	Commercial-Off-The-Shelf
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
CV	Computer Vision
CY	Carry Unit
DDR	Double Data Rate
DFF	Delay Flip-Flop
DLSB	Double Least Significant Bit
DMA	Direct Memory Access
DNN	Deep Neural Network
DRAM	Dynamic Random Access Memory
DSE	Design Space Exploration
DSP	Digital Signal Processing
ECR	Erroneous Classification Ratio
EDAC	Error Detection And Correction
EO	Earth Observation

ESA	European Space Agency
FE	Functional Element
FIFO	First-In First-Out
FIR	Finite Impulse Response
FPGA	Field-Programmable Gate Array
FPS	Frames Per Second
FPU	Floating-Point Unit
GPU	Graphics Processing Unit
HDL	Hardware Description Language
IoT	Internet of Things
LCD	Liquid Crystal Display
LEO	Low Earth Orbit
LLR	Log-Likelihood-Ratio
LNS	Logarithmic Number System
LOCE	Location Error
LSB	Least Significant Bit
LUT	Look-Up Table
MB	Modified Booth
MDK	Myriad Development Kit
ML	Machine Learning
MPDS	Mega Pixel Disparities per Second
MRED	Mean Relative Error Distance
MSB	Most Significant Bit
NaN	Not-a-Number
NCE	Neural Computer Engine
NCS	Neural Computer Stick
OC	Over-Clocking
ORIE	Orientation Error
PDF	Probability Density Function
PE	Processing Element
PLL	Phase-Locked Loop
PON	Possibility of Overflow-Normal
PP	Partial Product
PRED	Possibility of Relative Error Distance
PSNR	Peak Signal-to-Noise Ratio
PUN	Possibility of Underflow-Normal

QAM	Quadrature Amplitude Modulation
QoS	Quality of Service
RAMB	Random Access Memory Block
RED	Relative Error Distance
RF	Register File
RH	Radiation Hardened
RNS	Residue Number System
ROM	Read-Only Memory
RT	Radiation Tolerant
RTL	Register-Transfer Level
SHAVE	Streaming Hybrid Architecture Vector Engine
SIMD	Single Instruction Multiple Data
SIPP	Streaming Image Processing Pipeline
SNR	Signal-to-Noise Ratio
SoC	System-on-Chip
SoM	System-on-Module
SRAM	Static Random Access Memory
STA	Static Timing Analysis
SSIM	Structural Similarity Index
SWaP	Size, Weight and Power
TDP	Thermal Design Power
TMR	Triple Modular Redundancy
TOPS	Tera Operations Per Second
TPU	Tensor Processing Unit
UART	Universal Asynchronous Receiver-Transmitter
VBN	Vision-Based Navigation
VLIW	Very Long Instruction Word
VOS	Voltage Over-Scaling
VPU	Vision Processing Unit
WFG	Waveform Generator

Chapter 1

Introduction

1.1. The Landscape of Embedded Systems

The rapid technological advancements in processing, communication, storage, and sensing have transformed the landscape of embedded systems. With the emergence of Internet of Things (IoT) [1], there is a huge increment in the amount of data that are generated, which imposes technical challenges in the typical resource-constrained devices. On the other hand, the transmission of all these data to cloud infrastructures and data centers for processing creates communication bottlenecks, does not guarantee real-time response, while it is often avoided due to safety and privacy issues. Due to the ever-growing number of IoT connections, which is expected to be 27 billion in 2025 (as shown in Figure 1.1), the original cloud-centric system is already stressed to meet the runtime requirements. As a result, there is a tendency to process the data at the edge of the network, namely, upon they are generated. This new computing paradigm is established by the name Edge Computing [2] and has gained significant momentum over the last years.

Concurrently, the massive growth of demanding applications and powerful algorithms from domains such as Digital Signal Processing (DSP), Computer Vision (CV), Artificial Intelligence (AI), and Machine Learning (ML), marks a new era for the computing systems at the edge. Conventional embedded processors, such as Central Processing Units (CPUs) and microcontrollers, do not have the computational power to handle these compute-intensive workloads, and thus, they are unable to meet the performance requirements [3–5]. Another limitation is the restricted memory capacity at the edge, which slows down the processing, especially in applications with a large amount of I/O data.

It is evident that the proliferation of data along with the emergence of applications with increased complexity requires alternative design solutions to sustain sufficient performance at the edge. For this reason, the embedded systems have evolved over the years and constitute multi-purpose systems that sense, act, and communicate
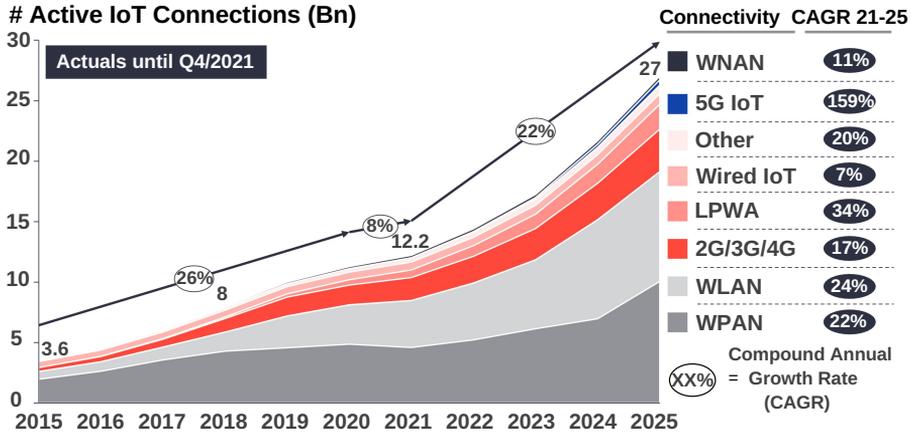
Figure 1.1: Number of connected IoT devices from 2015 to 2025. Source: IoT Analytics, https://iot-analytics.com/number-connected-iot-devices/.

with their environment. In earlier years, the embedded systems consisted of simple microcontrollers and memories. In contrast, contemporary embedded systems are build on complex System-on-Modules (SoMs) and System-on-Chips (SoCs), i.e., they are placed in single boards and single chips, respectively. These systems integrate CPUs, novel specialized processors, memory blocks, high-bandwidth peripherals, and communication interfaces. Nevertheless, besides space limitations, there is a vital factor that does not allow to seamlessly increase the computation resources, e.g., by adding more devices and/or processing units: the power consumption. Typical embedded systems consume some Watts [6], while ultra-low-power embedded systems (e.g., wearable) have a power budget of a few milliWatts [7]. Consequently, there is a trade-off between performance and power: from the one side, the applications demand speed and real-time response, while from the other side, there are tight power constrains at the edge.

1.2. The Evolution of Integrated Circuits

The integrated circuits constitute the cornerstone of computing systems, as they inherently impact their performance, power consumption, and area utilization. Figure 1.2 illustrates the trends for the microprocessors over the last 50 years. Historically, the semiconductor technology was driven for more than 40 years by two fundamental principles: Moore's Law [8] and Dennard's Law [9]. According to Gordon Moore [8], the number of transistors in a dense integrated circuit doubles approximately every two years. As shown in Figure 1.2, the transistor scaling is linear until today, even
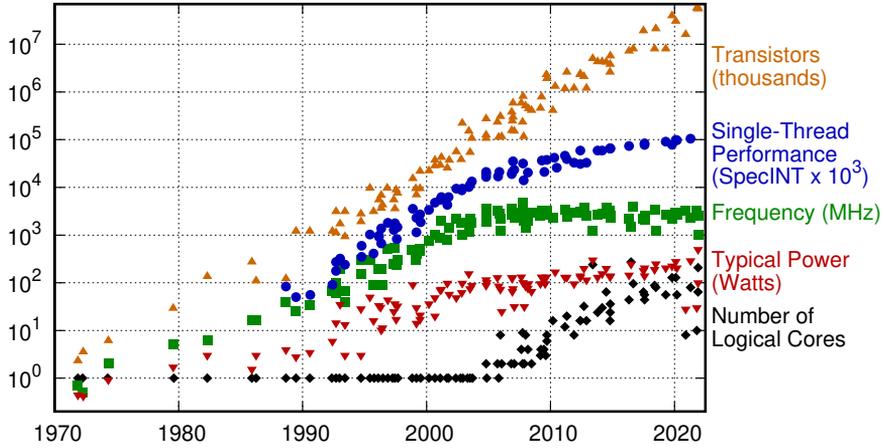
Figure 1.2: 50-year trends in microprocessors. Source: Karl Rupp, https://github.com/karlrupp/microprocessor-trend-data.

though several researchers forecast that Moore's Law will end soon [10]. Moore's Law is often quoted together with the prediction of David House (executive of Intel), who then said that the overall computing performance would double approximately every 18 months. On the other hand, Dennard's Law, which expired in the mid-2000s, is the force behind Moore's Law. According to Robert Dennard [9], the power density (power per silicon area) remains stable as the transistors get smaller, so that the power use stays in proportion with the area, i.e., both voltage and current scale (downward) with the length. Practically, for a given area size, the power consumption of the chip remained the same in each transition to a new generation of process technology (technological node). Therefore, each new process technology doubled the number of transistors in a chip without increasing the power consumption. The combination of the two laws allowed to scale the supply voltage and the threshold voltage, resulting in lower power per transistor, and thus, almost stable power density.

Nevertheless, the scaling law of Dennard did not consider the impact of the transistor sub-threshold leakage on the total chip power [11]. More specifically, in technological nodes of some nanometers, the decrease of the threshold voltage results in an exponential increase of the leakage power. This did not happen in 1970s, because the sub-threshold leakage was small and had negligible impact on the total chip power. As a result, the threshold voltage can no longer be reduced, and thus, the scaling of the supply voltage stopped (further scaling could affect the performance).

In summary, even though the number of transistors integrated per area is increasing, the supply voltage is not scaled accordingly, and thus, the power density is increased. The failure of Dennard's scaling in conjunction with other factors such as the cooling technology and the natural limits of silicon has led us to the "Dark Silicon" era [12–14]. In this era, the entire circuitry of an integrated circuit cannot be powered-on at the nominal operating voltage for a given Thermal Design Power (TDP) constraint. All the things considered, the power efficiency and management is nowadays a critical issue for computing systems, either they are placed at the edge (embedded systems) or on the cloud (data centers). Therefore, the industry of computing systems is forced to find new design approaches and computing platforms, which will improve the power efficiency while providing the desired performance.

Towards power-efficient, real-time, and high-yielding computing systems, ***Approximate Computing*** [15–19], ***Hardware Acceleration*** [3, 4, 20–22], and ***Heterogeneous Computing*** [23–27] have attracted much interest from the research community. The Approximate Computing paradigm exploits the error resilience of applications from the DSP/AI domains to reduce the quality of the results and deliver in exchange gains in power, area, and/or performance. Hardware Acceleration refers to the process of offloading compute-intensive tasks onto specialized hardware, such as the Application-Specific Integrated Circuits (ASICs) and the Field-Programmable Gate Arrays (FPGAs), rather than executing them on general-purpose CPUs. Finally, Heterogeneous Computing refers to systems integrating more than one type of processor, such as the Graphics Processing Units (GPUs) and the Vision Processing Units (VPUs). These design approaches are inherently linked and overlapping each other: (i) practically, heterogeneous hardware architectures (i.e., GPUs and VPUs) belong in the wider range of hardware accelerators, and (ii) approximations can be applied in implementations of all the hardware accelerators.

1.3. Approximate Computing

Approximate computations have been applied since the 1960s. For example, in one of the first works of the field, Mitchell proposed the logarithmic-based multiplication/-division [28]). Nevertheless, the first systematic efforts to define the Approximate Computing paradigm started in the late 2000s. Since then, various terms have been used to describe the process of generating approximate architectures, programs, and circuits. Approximate Computing is synonymous or overlaps with them. The most well-known terms of the literature are listed as follows:

- Chakradhar *et al.* [29] define "Best-Effort Computing" as *"the approach of designing software/hardware computing systems with reduced workload, improved*

parallelization and/or approximate components towards enhanced efficiency and scalability".

- Carbin *et al.* [30] introduce the term "Relaxed Programming" to express *"the transformation of programs with approximation methods and relaxed semantics to enable greater flexibility in their execution"*.

- Chippa *et al.* [31] use the term "Scalable Effort Design" for *"the systematic approach that embodies the notion of scalable effort into the design process at different levels of abstraction, involving mechanisms to vary the computational effort and control knobs to achieve the best possible trade-off between energy efficiency and quality of results"*.

According to Mittal [16], *"Approximate Computing exploits the gap between the accuracy required by the applications/users and that provided by the computing system to achieve diverse optimizations"*. Han and Orshansky [15] distinguish Approximate Computing from Probabilistic/Stochastic Computing, stating that *"it does not involve assumptions on the stochastic nature of the underlying processes implementing the system and employs deterministic designs for producing inaccurate results"*. Another interesting point of view is expressed by Sampson [32], who claims that Approximate Computing is based on *"the idea that we are hindering the efficiency of the computer systems by demanding too much accuracy from them"*. In this Dissertation, we attribute the following definition:

◇ Approximate Computing: *A progressive paradigm shift for the development of systems, circuits & programs, build on top of the error-resilient nature of application domains, and based on disciplined methods to intentionally induce errors that will provide valuable resource gains in exchange for tunable accuracy loss.*

The world of computing systems is full of workloads with intrinsic error tolerance [29]. These workloads are significant pillars of domains such as DSP (e.g, signal/image/video/audio processing), AI/ML (e.g., artificial neural networks, clustering), and Data Analytics (e.g., web search, data mining). Approximate Computing exploits this error resilience across the entire computing stack, i.e., at both software and hardware levels. The factors that allow Approximate Computing to decrease the quality of the results in exchange for resource gains, originate from [29]:

- the user's intention to accept results of lower quality.

- the limited human perception, e.g., in multimedia applications.

- the lack of perfect/golden results for validation, e.g., in data mining applications.

- the lack of a unique answer/solution, e.g., in machine learning applications.

- the application's self-healing property, i.e., the inherent capability to absorb/-compensate errors.

- the application's inherent approximate nature, e.g., in probabilistic calculations, iterative algorithms, and learning systems.

- the application's analog/noisy real-world input data, e.g., in multimedia/signal processing.

There are several challenges in the design of approximate systems. First of all, the accuracy is still of the utmost importance. Namely, even though errors are tolerated, the accuracy needs to be retained within the acceptable limits of the application. This requires the analysis of the application with realistic datasets and the development of accurate models that emulate the approximations. Moreover, modern approximate systems need to adapt their accuracy at runtime depending on the application's/user's constraints. Hence, the approximation methods should enable runtime approximation tuning instead of providing a single static approximation configuration. Another challenge in Approximate Computing is the development of systems with cross-layer approximation, i.e., the synergistic application of approximation techniques from different layers of the computing stack. This approach has shown promising results, however, sophisticated methodologies for the automatic configuration of the approximations in all system's modules are still missing.

In this <u>Dissertation</u>, regarding "Approximate Computing", we report an extensive literature review of the state-of-the-art software and hardware approximation techniques and then, we focus on hardware-level Approximate Computing. In particular, we propose new approximation techniques for the design of power-efficient arithmetic circuits, and employ them along with other design techniques to develop approximate hardware accelerators from the DSP and AI domains.

1.4. Hardware Acceleration

The conventional CPU-based computing platforms do not provide sufficient performance for compute-intensive DSP/AI workloads [3, 4, 20–22]. As already discussed, the computing industry employs novel processors based on ASICs, FPGAs, GPUs, and VPUs to cope with the increased demands of modern DSP/AI workloads. The execution of high-complexity computing tasks on customized hardware, such as the preceding specialized processors, is widely known as Hardware Acceleration. In terms of development, Hardware Acceleration requires more programming effort and time than the respective development on general-purpose CPU-based processors. Nevertheless, the performance results are very impressive, providing speedups of orders of magnitude. We note that besides the aforementioned accelerators, the market offers

additional hardware platforms, such Google's Tensor Processing Unit (TPU) [33], which is an AI accelerator. Below, we introduce in brief the most common hardware accelerators:

- <u>ASIC</u>: an integrated circuit that is customized for a specific application/function. It cannot be reprogrammed or modified after its production. ASICs are used for the efficient implementation of DSP/AI functions and general-purpose tasks of computing systems.

- <u>FPGA</u>: an integrated circuit that is manufactured with configurable logic blocks and programmable interconnects. It can be reprogrammed numerous times after its production. FPGAs are mainly used for accelerating heavy DSP/AI functions, as well as for interfacing and prototyping purposes.

- <u>GPU</u>: a processor integrating multiple specialized small cores. GPUs are used for accelerating graphics rendering, calculations involving massive amount of data, scientific calculations, and CV/AI workloads.

- <u>VPU</u>: a processor integrating vector cores, image processing filters and AI engines. VPUs excel in imaging/vision tasks and are used for accelerating DSP/AI workloads.

In the remainder of this Dissertation, we report a plethora of related works and discussions involving the aforementioned hardware accelerators. Indicatively, in Figure 1.3, we depict the well-known Eyeriss ASIC for Convolutional Neural Networks (CNNs) [34] and Xilinx's Zynq-7000 SoC FPGA [35]. Eyeriss is designed for accelerating CNNs with many layers and varying shapes, and it is based on a spatial array of 168 Processing Elements (PEs) and a global on-chip buffer. The data movement is optimized by exploiting data reuse and inter-PE communication, while data gating and compression are used to reduce the power consumption. On the other hand, the Zynq SoC FPGAs have been established in the market as state-of-the-art devices for Hardware Acceleration. These devices offer the software programmability of an ARM-based processing system (2 ARM Cortex-A9, on-chip memory, caches, memory controllers, and peripherals) and the hardware programmability of a traditional FPGA (programmable resources for logic functions, arithmetic operations, registers, and memories). All these components are integrated in a single chip to provide a fully scalable SoC platform for high-performance DSP/AI processing.

In this <u>Dissertation</u>, regarding "Hardware Acceleration", we implement approximate arithmetic circuits using standard-cell libraries for ASIC. Our arithmetic circuits are also integrated in hardware DSP/AI functions, which are accelerated on standard-cell ASIC and Xilinx's FPGAs. Moreover, we implement demanding CV kernels on

(a)

(b)
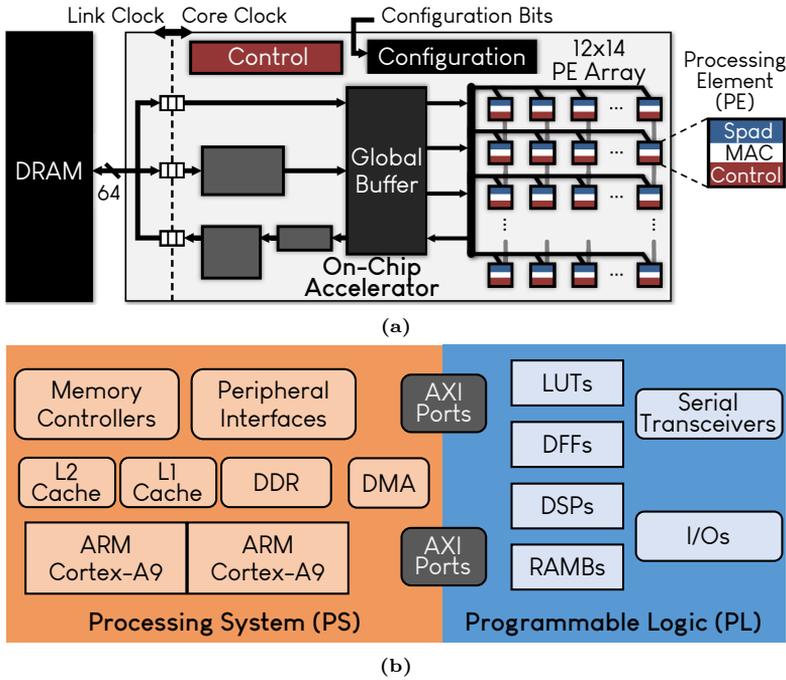
Figure 1.3: High-level architecture of **(a)** the Eyeriss ASIC [34] and **(b)** the Xilinx Zynq-7000 SoC FPGA [35].

the new European space-grade FPGAs. Finally, we accelerate custom DSP kernels, a sophisticated CV pipeline, and CNNs on Intel's VPUs.

1.5. Heterogeneous Computing

The increased diversity of modern DSP/AI workloads in I/O, computational, and memory requirements has marginalized the use of homogeneous CPU-based computing platforms. To cope with all these requirements and provide efficient design solutions, the computing industry has turned to Heterogeneous Computing architectures [23–27]. These architectures integrate more than one type of processor and potentially different memory technologies. In particular, contemporary heterogeneous architectures offer both general-purpose processors and specialized acceleration cores/engines. In terms of storage, the heterogeneity usually offers global memory, caches and scratchpad (working) memory. Figure 1.4 shows the evolution of heterogeneous architectures according to Shalf [10]. Computing architectures with two similar CPUs or a multi-core CPU (Figure 1.4a) are now considered homo-
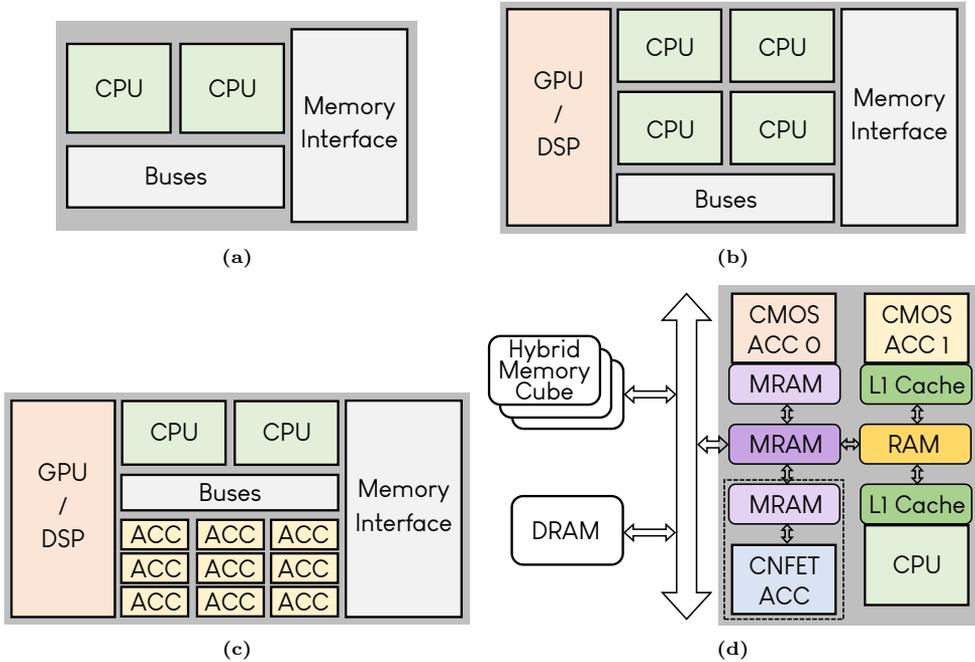
Figure 1.4: Evolution of heterogeneous computing architectures [10]: **(a)** homogeneous CPU (past), **(b)** CPU + GPU/DSP (present), **(c)** CPU + GPU/DSP + accelerators (present), and **(d)** extreme heterogeneity in processors and memories (future).

geneous systems. Currently, there are two prevailing kinds of computing architecture:

(i) the heterogeneous architecture of Figure 1.4b, which combines the general-purpose CPU with a GPU/DSP accelerator [25].

(ii) the very heterogeneous architecture of Figure 1.4c, which additionally includes small accelerators (e.g., the VPU SoCs [36]).

The heterogeneity is expected to increase in the future (Figure 1.4d), integrating accelerators and memories of different technologies. The very heterogeneous architecture of Figure 1.4c is met in Intel's Myriad VPUs [36], which constitute embedded SoCs for Edge Computing. These VPUs have recently emerged as an attractive solution for accelerating imaging applications with only 1–2W. Compared to the CPU–GPU architectures, the VPU SoCs are more heterogeneous, as they offer general-purpose CPUs, vector cores, hardware filters for image processing, and a dedicated AI accelerator in the case of Myriad X.

Heterogeneous Computing imposes several challenges. From the developer side, the programming model involves parallel computing and mapping to specialized hard-

ware, and thus, it is more complex than the respective one of conventional comput-
ing. The workloads need to be efficiently distributed and parallelized among the cores
and accelerators to deliver improved performance and power efficiency. Towards this
direction, the developer has to make decisions regarding the application's decomposi-
tion into parallel computing tasks, the selection of the most suitable processor for each
task, and the identification of the parts that offer limited parallelization opportunities
or do not require acceleration. In the case of the Myriad VPUs, there are more tech-
nical challenges, given that these SoCs are mainly build for power efficiency rather
than high performance. Therefore, the developer needs to exploit every piece of the
VPU heterogeneity to provide sufficient performance.

In this Dissertation, regarding "Heterogeneous Computing", we develop methodolo-
gies for the efficient mapping and scheduling of DSP/CV kernels and CNNs on the
Myriad VPUs. We introduce several high- and low-level implementation techniques
and evaluate the suitability of the VPUs as edge processors.

1.6. Scope and Contribution of Dissertation

The **scope** of the Dissertation is the design of arithmetic circuits and DSP & AI
accelerators. In this context, we propose design solutions and methodologies for im-
proving the efficiency of the implementations on ASIC/FPGA and multi-core SoCs.
At circuit level, we adopt the promising design paradigm of Approximate Comput-
ing and propose new arithmetic approximation techniques, which are then used to
design various approximate hardware accelerators on ASIC/FPGA technology. At
platform level, we aim to unlock the full potential of new embedded devices, such
as the space-grade FPGAs and the multi-core VPUs, by surpassing the bottlenecks
of the tools and exploiting the heterogeneity of the SoCs, respectively, to accelerate
high-performance DSP/AI workloads.

The main **differentiation** of the Dissertation compared to prior art is summarized
as follows:

- At design technique level, we propose approximation methods that provide a
  larger approximation space, i.e., multiple approximation configurations, en-
  abling to maximize the resource gains under a specified error constraint.

- At circuit level, we propose energy-efficient approximate circuits that can *seam-
  lessly* adjust their approximation configuration at runtime.

- At hardware accelerator level, we perform an *extensive* design space exploration
  on approximation techniques, arithmetic formats, algorithms, and hardware
  design techniques to generate approximate ASIC/FPGA-based accelerators.

- At computing platform level, we *systematically* examine the capabilities of the programming tools and exploit the underlying hardware architectures to accelerate high-performance DSP/AI workloads.

The ***contribution*** of the Dissertation is summarized as follows:

(i) In Chapter 2, we report a comprehensive up-to-date *literature survey* for Approximate Computing, where we report the basic terminology, and then, classify and analyze the state-of-the-art software and hardware approximation techniques.

(ii) In Chapter 3, we highlight the significance of the underlying arithmetic in circuits, and show that novel numerical formats and *sophisticated bit-level optimizations* can provide valuable resource gains in hardware.

(iii) In Chapter 4, we address the circuit overheads of the classic high-radix encodings and propose a new approximate hybrid high-radix encoding, which is parametric in terms of approximation degree. This encoding is used to design the *RAD* family of approximate multipliers.

(iv) In Chapter 5, we introduce a low-overhead dynamic configuration scheme for adjusting the approximation degree of multipliers at runtime. This technique is applied in fixed- and floating-point arithmetic, generating the *DyFXU* and *DyFPU* families of runtime-configurable approximate multipliers.

(v) In Chapter 6, we highlight the efficiency of integrating more than one approximation technique in the design of approximate circuits. In this context, we combine various state-of-the-art techniques and provide a very large design space for approximate multiplication. This extensive exploration results in the *ROUP* family of approximate multipliers, which form the state-of-the-art Pareto-front.

(vi) In Chapter 7, we introduce a methodology for designing approximate DSP/AI hardware accelerators. Based on this methodology, we fuse approximation techniques with various arithmetic formats, DSP/AI algorithms, and hardware design techniques, to generate energy-efficient *hardware accelerators* for 1D/2D signal processing and neural networks.

(vii) In Chapter 8, we introduce a methodology for efficiently mapping and accelerating high-performance DSP algorithms on the *new European space-grade FPGAs*. Based on this methodology, we apply our tool-level exploration to surpass issues that arise because the FPGA vendor is new and the space-grade FPGAs exhibit decreased flexibility compared to the commercial ones.

(viii) In Chapter 9, we introduce a methodology for partitioning, scheduling and optimizing demanding DSP and AI workloads on the *heterogeneous multi-core*

VPUs. Based on this methodology, we exploit the full potential of the VPU SoCs to provide sufficient DSP and AI acceleration with limited power consumption.

1.7. Structure of Dissertation

The remainder of the Dissertation is organized as follows. Chapter 2 introduces the Approximate Computing paradigm, reviewing the terminology and state-of-the-art software & hardware approximation techniques. Chapters 3–9 report the main work of the Dissertation. Finally, Chapter 10 concludes the Dissertation by summarizing the contributions and discussing future extensions. The structure of the main work is presented in Figure 1.5 and is divided in two parts:

Part I: *"Arithmetic Approximation Techniques for Circuit Design"*

Part II: *"Design Methodologies for Embedded Computing"*

Part I includes Chapters 3–7 and focuses on arithmetic approximation techniques and the design of approximate DSP/AI hardware accelerators. Part II includes Chapters 8–9 and focuses on new embedded computing platforms (space-grade FPGAs and multi-core VPU SoCs) and the efficient acceleration of DSP/AI kernels. Both parts have a ***common goal***: the development of energy-efficient DSP/AI accelerators. Part I reaches this goal from a lower design abstraction layer, while Part II reaches it from higher design abstraction layers. Figure 1.6 depicts how we achieve this goal and what issues we have to surpass: (i) by using arithmetic approximations, while we have to care about the errors and application's accuracy, (ii) via extensive and systematic tooling on the new space-grade FPGAs, where we have to surpass the limitations/issues of newly released tools/devices, (iii) by exploiting the heterogeneity of low-power multi-core VPUs, where we have to cope with resource-constrained edge devices and the increased complexity of the SoCs. Next, we discuss the content of each chapter included in Part I and Part II.

Chapter 3: This chapter acts as introductory to low-level logic optimizations, targeting to highlight the significance of studying the arithmetic of circuits/accelerators. For this purpose, it focuses on the Double Least Significant Bit (DLSB) numerical format, in which the numbers have an extra least significant bit, and it proposes sophisticated low-level optimizations. These bit manipulations are also used in the next chapters, and specifically in the design of approximate circuits. More explicitly, in this chapter, we improve the DLSB multiplication, resulting in decreased overheads versus the straightforward design approach. Moreover, as case study, we demonstrate how the proposed optimized circuit can be used as building block in the implementation of large-size multiplications.
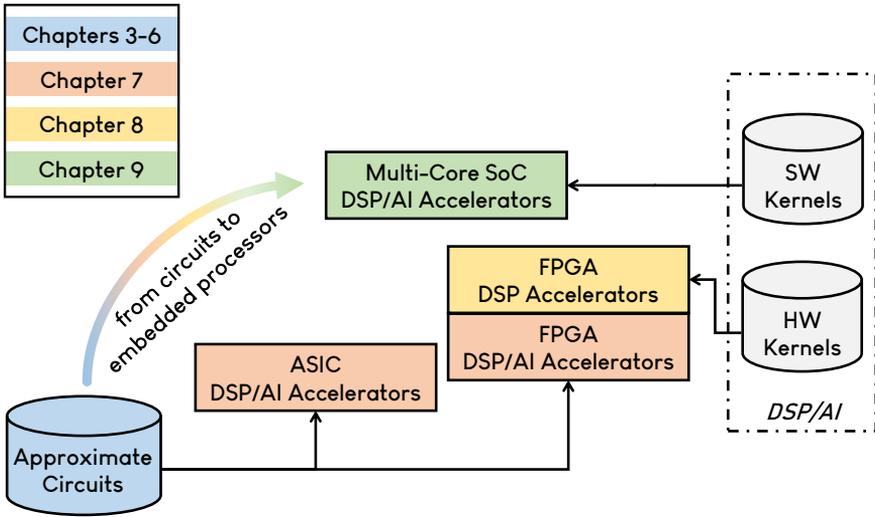
Figure 1.5: The structure of the Ph.D. Dissertation.
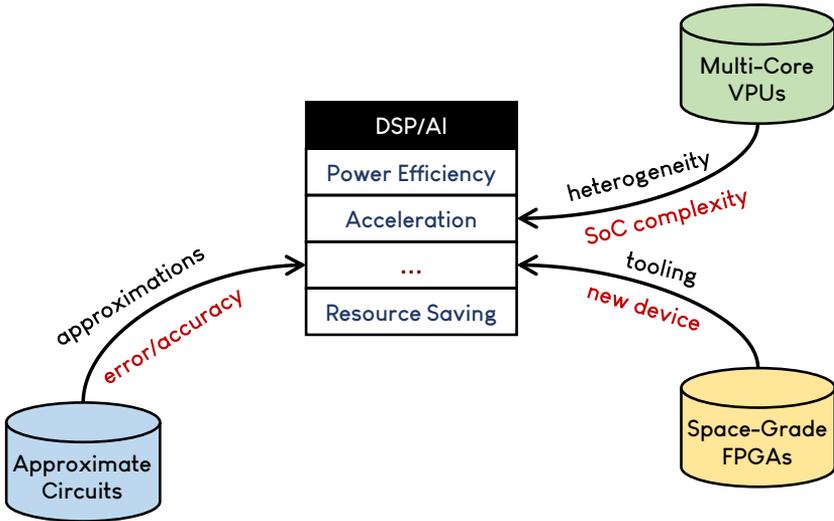
Figure 1.6: The goal of the Ph.D. Dissertation achieved through different design layers.

Chapter 4: This chapter proposes an approximate hybrid high-radix encoding for generating the energy-efficient RAD multipliers. The proposed encoding scheme approximately encodes one of the operands, using the accurate radix-4 encoding for its most significant part and an approximate high-radix encoding for its least significant part. The approximation is inserted by mapping all the high-radix values to a set of values including only the 4 largest powers of two. The proposed RAD family of approximate multipliers is configurable, and can be tuned to achieve the desired

energy–accuracy trade-off.

Chapter 5: This chapter proposes runtime-configurable approximate multipliers for fixed- and floating-point arithmetic. The approximation is inserted by two orthogonal techniques, i.e., partial product perforation and partial product rounding, which allow to integrate a low-overhead scheme for tuning the approximation at runtime. The runtime circuit variants DyFXU and DyFPU deliver negligible overhead versus their design-time counterparts (AxFXU and AxFPU). However, they still provide energy efficiency and benefit from their capability of selecting a different approximation configuration (among numerous ones) at runtime.

Chapter 6: This chapter proposes the concept of cooperative approximation, namely, the application of more than one arithmetic approximation technique in the design of a circuit. The goal is twofold: (i) to create a very large approximation space that serves various design scenarios and can handle different error constraints or power budgets, and (ii) to identify the most efficient approximation solutions in terms of both accuracy and resources. Our extensive design space exploration results in 5 new families of approximate multipliers, from which, ROUP is the most prominent, as it forms the state-of-the-art Pareto front with increased resolution.

Chapter 7: This chapter introduces a methodology for the systematic development of approximate DSP/AI hardware accelerators. The proposed methodology consists of two stages, i.e., software-level exploration and hardware development. The main feature of the methodology is the combination of approximation techniques with different arithmetic formats (e.g., fixed/floating-point, quantized integer), alternative algorithms for the same application, and classic hardware design techniques. The goal of this chapter is twofold: (i) to assess the Dissertation's approximate designs in real-world DSP/AI applications, and (ii) to evaluate the error resilience and quantify the resource gains of approximate DSP/AI accelerators.

Chapter 8: This chapter proposes a design methodology for porting demanding DSP algorithms on the new European space-grade FPGAs. The methodology is divided with respect to the stages of the typical FPGA design flow. Our systematic design approach aids us to surpass issues that arise when using new tools or porting designs developed in other FPGA vendors, as well as confront the decreased flexibility and lower performance of space-grade FPGAs compared to their commercial counterparts. The evaluation is performed with hardware kernels for feature detection and stereo vision, and it includes comparisons to other FPGAs.

Chapter 9: This chapter proposes a design methodology for the efficient mapping and acceleration of compute-intensive algorithms on the heterogeneous multi-core VPUs. The methodology aims to highlight the most efficient partitioning and scheduling schemes in such complex and very heterogeneous SoCs. In this context, we intro-

duce several high- and low-level implementation techniques. Given that the VPUs are designed to provide low power consumption, our methodology and design choices allow us to deliver sufficient performance in custom DSP kernels and a sophisticated CV pipeline. Moreover, we deploy a demanding CNN. The evaluation includes comparison results with other state-of-the-art embedded devices.

Finally, even though all chapters are related to each other, they constitute standalone structures of text. Namely, each chapter has its own abstract, introduction, list of contributions, proposed techniques/designs/methodologies, experimental evaluation and conclusion.

1.8. Overview of Technologies, Tools and Devices

In this section, we report all the industrial-strength programming/development tools, devices/platforms and technologies that are used in the Dissertation. Table 1.1 summarizes all the relevant details. In Chapters 3–6, all the circuits are synthesized on

Table 1.1: Overview of Dissertation's tools and devices.

Tool	Usage	Reference
Synopsys Design Compiler	standard-cell synthesis	Chapters 3–7
Synopsys PrimeTime	standard-cell power measurement	Chapters 3–7
Siemens QuestaSim	simulation (validation & power)	Chapters 3–8
Xilinx Vivado	FPGA implementation	Chapters 7–8
Intel Quartus	FPGA implementation	Chapter 8
Microsemi Libero	FPGA implementation	Chapter 8
NanoXplore NXmap	FPGA implementation	Chapter 8
Intel MDK	VPU implementation	Chapter 9
Intel OpenVINO	VPU implementation	Chapter 9
Google TensorFlow	CNN development	Chapters 7,9

Device/Technology		Reference
TSMC Standard Cells (65-nm, 45-nm)		Chapters 3–7
Xilinx FPGAs (ZCU106, Zynq-7020)		Chapter 7
Xilinx FPGA (Virtex-5QV)		Chapter 8
Intel FPGA (Cyclone III)		Chapter 8
Microsemi FPGA (RTG4)		Chapter 8
NanoXplore FPGAs (NG-Medium, NG-Large)		Chapter 8
Intel VPUs (Myriad 2, Myriad X, NCS2)		Chapter 9
Nvidia GPU (Jetson Nano)		Chapter 9

TSMC standard-cell libraries (65-nm and 45-nm) with Synopsys' Design Compiler tool. All the simulations are performed with Siemens' (Mentor Graphics) QuestaSim, while for the power measurements, we use Synopsys' PrimeTime. The same libraries and tools are used for the synthesis of the ASIC-based accelerators in Chapter 7. Moreover, this chapter includes implementation results for accelerators on Xilinx's ZCU106 and Zynq-7020 FPGAs, while the associated tool is Vivado. In Chapter 8, we report numerous results from the implementation of DSP kernels on various FPGAs (Xilinx Virtex-5QV, Intel Cyclone III, and Microsemi RTG4). For these implementations, we use the corresponding software tool of each FPGA vendor. Chapter 8 also includes results for NanoXplore's new space-grade FPGAs (NG-Medium and NG-Large), which are generated by the NXmap tool. Chapter 9 focuses on Intel's VPUs (Myriad 2 and Myriad X), and all the implementations are performed with the associated tools, i.e., the MDK design suite for custom development and the OpenVINO framework for the deployment of neural networks. This chapter also reports results for Nvidia's Jetson Nano GPU.

Chapter 2

The Approximate Computing Paradigm

The emergence of complex applications in domains applying multimedia processing and machine learning tasks has transformed the computing paradigm in embedded systems and data centers. These applications involve massive data and/or high computational complexity. Consequently, there is an emerging need for computational resources, which results in increasing more and more the power consumption of the computing systems. In the past, the technology scaling has played significant role towards surpassing these challenges, however, its declining efficiency pushes us to examine new computing paradigms. Approximate Computing is such an alternative paradigm, which trades-off accuracy loss and decreased quality of results for resource gains (e.g., in power/energy, area, latency, throughput). This computing paradigm is applied to error-resilient applications, such as those involving multimedia and machine learning, and it induces errors based on a disciplined approach to improve the efficiency of the systems/circuits and deliver the desired resource gains. Approximation opportunities abound in every layer of the typical computing stack, i.e., from transistors and circuits to compilers and programming languages. Therefore, there is a great variety of approximation techniques at each design abstraction layer, which study the errors and relax the accuracy from different perspective. In this chapter, we review state-of-the-art research works in Approximate Computing from all the layers of the computing stack. At first, we discuss the terminology of Approximate Computing and then, we classify and analyze the state-of-the-art approximation techniques with respect to their application layer (software or hardware). Our taxonomy is fine-grained, namely we study in-depth the approximation techniques of each layer and classify them based on their design approach.
*This chapter is based on our **publication** in [37].*

2.1. Introduction

Approximate Computing covers the entire computing stack, i.e., approximation techniques are applied at all design abstraction layers. Significant research has been conducted in the field of software approximation techniques, which involve approximate programming languages, approximate compilers, approximation frameworks, quality-aware runtime systems, as well as approximations via precision scaling, task skipping and memoization. On the other hand, the most common hardware techniques target the modification of the circuits and hardware architectures, i.e., they generate a lossy circuit/architecture from the nominal accurate one. There is also wide research on the tunable scaling of the circuit's voltage or frequency.

This chapter provides background in Approximate Computing. In particular, we introduce the terminology of Approximate Computing based on our literature search, and we report an extensive review of the state-of-the-art software and hardware approximation techniques. Our literature review includes newer works and more works compared to previous well-established surveys in Approximate Computing, i.e., from Mittal *et al.* (2016) [16], Xu *et al.* (2016) [17], and Shafique *et al.* (2016) [18]. Furthermore, it differentiates from the survey of [19], as it focuses on implementation details and analyzes the approximations of each reviewed technique.

The remainder of this chapter is organized as follows. Section 2.2 reports the terminology. Section 2.3 classifies and analyzes the software approximation techniques, while Section 2.4 classifies and analyzes the hardware approximation techniques.

2.2. The Terminology of Approximate Computing

Table 2.1 describes the most frequently used terms in Approximate Computing. The term *error* is used to indicate that the output result is different from the accurate result (produced with conventional computing). Error is distinguished from *fault*, which refers to an unexpected condition (e.g., stuck-at-logic in circuits, bit-flips in memories, faults in operating systems) that causes the system to unintentionally output erroneous results. Another significant term is *accuracy*, which is defined as the distance between the approximate and the (nominal) accurate result and is expressed with various error metrics (e.g., error rate, mean relative error, mean squared error, classification accuracy). Accuracy is distinguished from *precision*, which expresses the differentiation between nearby discrete values and does not refer to errors of Approximate Computing but to quantization noise (inserted by the real-to-digital value mapping).

Table 2.1: Basic terminology of Approximate Computing.

Term	Description
Error-Resilient Application	The application that tolerates errors and accepts results of lower quality.
Quality of Service	The quality of the results in terms of errors and accuracy.
Error Constraints	The quality/accuracy requirements that the results should satisfy.
Error Threshold	The maximum error allowed in the results.
Golden Result	The result that is obtained from the accurate computations.
Acceptable Result	The result that satisfies the application's error constraints.
Variable Accuracy	The capability of providing different levels of accuracy.
Non-Critical Task	The task/computation that can be safely approximated due to its small impact on the quality of the output results.
Error Analysis	The study involving metrics, mathematics and simulations to examine the range, frequency, scaling, and propagation of errors.
Approximation Technique	The systematic and disciplined approach/method to insert computation errors in exchange for resource gains.
Approximation Degree	The strength of the approximation technique in terms of computations approximated and errors induced.
Approximation Configuration	An instance of the parameters/settings of the approximation technique.
Frozen Approximation	The approximation degree is fixed and cannot be re-configured.
Dynamic Approximation Tuning	The capability of adjusting the approximation degree at runtime to satisfy the desired error constraints.
Cross-Layer Approximation	The approximation that is applied at multiple design abstraction layers (software, hardware, architecture).
Heterogeneous Approximation	The approximation that concurrently applies multiple configurations of different degree within the same system.
Approximation Space Exploration	The study involving error analysis and gain quantification to examine trade-offs and select the suitable approximations.
Approximation Localization	The systematic approach to locate the computations and regions that are offered for approximation.
Error Modeling	The process of emulating the errors of the approximations.
Error Prediction	The process of predicting errors before computing the final result.
Error Detection	The process of identifying an error occurrence.
Error Compensation	The process of modifying the erroneous result to reduce the error.
Error Correction	The process of correcting the erroneous result.

More specifically, in computer arithmetic, the more bits are used for the decimal number part, the higher the precision, i.e., there are more bits for the representation and

the numbers are closer to their real value. Moreover, in Approximate Computing, the term *Quality-of-Service (QoS)* is used to describe the overall quality of the results (in terms of accuracy and errors), considering the expected/accurate results as baseline and the application's quality constraints.

2.3. Classification of Software Approximation Techniques

In this section, we classify and introduce approximation techniques that are applied at software level, i.e., the higher level of the design abstraction hierarchy. The goal of software Approximate Computing is to improve the execution time of the program and/or the energy consumption of the system. The techniques of the literature, illustrated in Figure 2.1, can be categorized into six classes: (i) *Selective Task Skipping*, (ii) *Approximate Memoization*, (iii) *Relaxed Synchronization*, (iv) *Precision Scaling*, (v) *Data Sampling*, and (vi) *Approximate Programming Languages*. Typical techniques include some of the following features: approximation libraries/frameworks, compiler extensions, accuracy tuning tools, runtime systems, and language annotations. Moreover, there are numerous techniques allowing the programmer to specify QoS constraints, provide approximate code variants, and mark the program's regions/tasks for approximation.

The remainder of this section reports representative state-of-the-art works for software Approximate Computing. Besides classifying state-of-the-art techniques, we
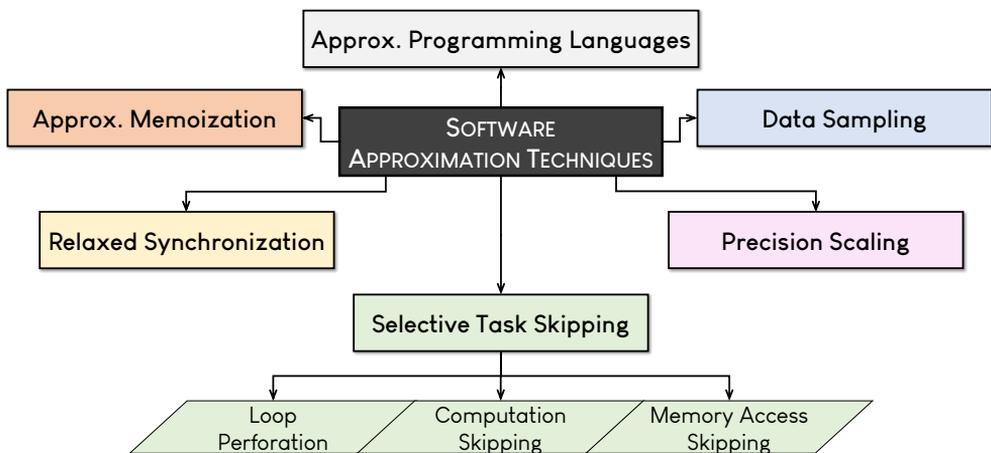
Figure 2.1: Classification of state-of-the-art software approximation techniques.

Table 2.2: Classification of software approximation techniques.

SW Approximation Class	References
Loop Perforation	[40–48]
Computation Skipping	[49–57]
Memory Access Skipping	[58–63]
Approximate Memoization	[64–73]
Relaxed Synchronization	[74–81]
Precision Scaling	[82–97]
Data Sampling	[98–109]
Approximate Program. Languages	[30, 110–127]

discuss their approximations and how they are applied. Table 2.2 reports the references of all the reviewed research works. We note that the literature also includes entire approximation frameworks, such as ACCEPT [38] and OPPROX [39], which integrate multiple of these state-of-the-art techniques. Their goal is to perform an extensive exploration using differing approximation approaches, identify the best approximation opportunities, and hence, maximize the resource gains while satisfying the application's/user's constraints.

2.3.1. Selective Task Skipping

Loop Perforation

The loop perforation technique skips some of the loop iterations in a software program to provide performance/energy gains in exchange for QoS loss. Subsequently, we present several relevant works [40–48] involving design space exploration on loop perforation with programming frameworks and profiling tools.

Starting with one of the first state-of-the-art works, the SpeedPress compiler [40] supports a wide range of loop perforation types, i.e., modulo, truncation, and randomized. It takes as input the original source code, a set of representative inputs, as well as a programmer-defined QoS acceptability model, and it outputs a loop perforated binary. In the same context, Misailovic *et al.* [41] propose a QoS profiler to identify computations that can be approximated via loop perforation. The proposed profiling tool searches the space of loop perforation and generates results for multiple perforation configurations. In [42], the same authors propose a methodology to exclude critical loops, i.e., whose skipping results in unacceptable QoS, and they perform exhaustive and greedy design space explorations to find the Pareto-optimal perforation configurations for a given QoS constraint.

In [43], the authors propose an architecture that employs a profiler to identify non-critical loops towards their perforation. To protect code segments that can be affected by the perforated loops, the architecture is equipped with HaRE, i.e., a hardware resilience mechanism. Another interesting work is GraphTune [44], which is an input-aware loop perforation scheme for graph algorithms. This approach analyzes the input dependence of graphs to build a predictive model that finds near-optimal perforation configurations for a given accuracy constraint. Li *et al.* [45] propose a compiling & profiling system, called Sculptor, to improve the conventional loop perforation, which skips a static subset of iterations. More specifically, Sculptor dynamically skips a subset of the loop instructions (and not entire iterations) that do not affect the output accuracy. More recently, the authors of [46] develop LEXACT, which is a tool for identifying non-critical code segments and monitoring the QoS of the program. LEXACT searches the loop perforation space, trying to find perforation configurations that satisfy pre-defined metrics.

The loop perforation technique has also been used in approximation frameworks for heterogeneous multi-core systems combining various approximation mechanisms. Tan *et al.* [47] propose a task scheduling algorithm, which employs multiple approximate versions of the tasks with loops perforated. Kanduri *et al.* [48] target applications in which the main computations are continuously repeated, and they tune the loop perforation at runtime.

Computation Skipping

This technique omits the execution of blocks of codes with respect to the acceptable QoS loss, programmer-defined constraints, and/or runtime predictions regarding the output accuracy [49–57]. Compared to loop perforation, these techniques do not focus only on skipping loop iterations, but also skip higher-level computations/tasks e.g., an entire convolution operation. Most of the state-of-the-art works perform application-specific computation skipping.

Meng *et al.* [49] introduce a parallel template to develop approximate programs for iterative-convergence recognition & mining algorithms. The proposed programming template provides several strategies (implemented as libraries) for task dropping, such as convergence-based computation pruning, computation grouping in stages, and early termination of iterations. Another interesting work involving application-specific computation skipping is presented in [50]. The authors of this work study the error tolerance of the supervised semantic indexing algorithm to make approximation decisions. Regarding their task dropping approach, they choose to omit the processing of common words (e.g., "the", "and") after the initial iterations, as these computations have negligible impact on accuracy.

The authors of [51] propose two techniques to find computations with low impact on the QoS of the Reduce-and-Rank computation pattern, targeting to approximate or skip them completely. To identify these computations, the first technique uses intermediate reduction results and ranks, while the second one is based on the spatial or temporal correlation of the input data (e.g., adjacent image pixels or successive video frames). Similar to the other state-of-the-art works, Vassiliadis *et al.* [52,53] propose a programming environment that skips (or approximates) computations with respect to programmer-defined QoS constraints. More specifically, the programmer expresses the significance of the tasks using pragmas directives, optionally provides approximate variants of tasks, and specifies the task percentage to be executed accurately. Based on these constraints, the proposed system makes decisions at runtime regarding the approximation/skipping of the less significant tasks.

Rinard [54] builds probabilistic distortion models based on linear regression to study the impact of computation skipping on accuracy. The programmer partitions the computations into tasks, which are then marked as "critical" or "skippable" through random skip executions. The probabilistic models estimate the output distortion as function of the skip rates of the skippable tasks. This approach is also applied in parallel programs [55], where probabilistic distortion models are employed to tune the early phase termination at barrier synchronization points, targeting to keep all the parallel cores busy.

Significant research has also been conducted on skipping the computations of Convolutional Neural Networks (CNNs). Lin *et al.* [56] introduce PredictiveNet to predict the sparse outputs of the nonlinear layers and skip a large subset of convolutions at runtime. The proposed technique, which does not require any modification in the original CNN structure, examines the most-significant part of the convolution to predict if the nonlinear layer output is zero, and then it decides whether to skip the remaining least-significant part computations or not. In the same context, Akhlaghi *et al.* [57] propose SnaPEA, exploiting the convolution–activation algorithmic chain in CNNs (activation inputs the convolution result and outputs zero if it is negative). This technique early predicts negative convolution results, based on static re-ordering of the weights and monitoring of the partial sums' sign bit, in order to skip the rest computations.

Memory Access Skipping

Another approach to improve the execution time and energy consumption at software level is the memory access skipping. Such techniques [58–63] aim to avoid high-latency memory operations, while as a result, they also reduce the number of computations.

Miguel *et al.* [58] exploit the approximate data locality to skip the required memory accesses due to L1 cache miss. In particular, they employ a load value approximator, which learns value patterns using a global history buffer and an approximator table, to estimate the memory data values. RFVP [59] uses value prediction instead of memory accessing. When selected load operations miss in the cache memory, RFVP predicts the requested vales without checking for misprediction or recovering the values. As a result, timing overheads from pipeline flushes and re-executions are avoided. Furthermore, a tunable rate of cache misses is dropped after the value prediction to eliminate long memory stalls. The authors of [60] propose a framework that skips costly last-level cache misses according to a programmer-defined error constraint and an heuristic predicting skipped data.

To improve the performance of CUDA kernels on GPUs, Samadi *et al.* [61] propose a runtime approximation framework, called SAGE, which focuses on optimizing the memory operations among other functionalities. The approximations lie in skipping selective atomic operations (used by kernels to write shared variables) to avoid conflicts leading to performance decrease. Furthermore, SAGE reduces the number of memory accesses by packing the read-only input arrays, and thus, allowing to access more data with fewer requests. Karakoy *et al.* [62] propose a slicing-based approach to identify data (memory) accesses that can be skipped to deliver energy/performance gains within an acceptable error bound. The proposed method applies backward and forward code slicing to estimate the gains from skipping each output data. Moreover, the '0' value is used for each data access that is not performed. The ApproxANN framework [63], apart from performing approximate computations, skips memory accesses on neural networks according to the neuron criticality. More specifically, a theoretical analysis is adopted to study the impact of neurons on the output accuracy and characterize their criticality. The neuron approximation under a given QoS constraint is tuned by an iterative algorithm, which applies the approximations and updates the criticality of each neuron (it may change due to approximations in other neurons).

2.3.2. Approximate Memoization

The memoization technique stores results of previous calculations or pre-computed values in memory to use them instead of performing calculations. Namely, this memory functions as a look-up table, which maps a set of data identifiers to a set of stored data. Subsequently, we focus on approximate memoization techniques [64–69] relying on software frameworks, compilers and programmer's decisions. Nevertheless, we note that there are also approaches [70–73] requiring hardware modification to support memoization.

Chadhuri *et al.* [64] propose an approximate memoization for computations in loops. Prior executing an expensive function within a loop, this technique checks a look-up table to find if this computation was previously performed for similar input data. In this case, the cached result is used, otherwise, the function is executed and the new computation is stored in the look-up table. Paraprox [65] is a software framework for identifying common patterns in data-parallel programs and applying tailored approximations. For the Map & Scatter/Gather patterns, Paraprox uses memoization rather than performing computations. In particular, it fills a look-up table with pre-computed data, which are obtained from the execution of the Map & Scatter/Gather function for some representative inputs, and it performs runtime look-up table queries instead of the conventional computations.

iACT [66] is another approximation framework that applies runtime memoization among other functionalities. The programmer uses pragmas to declare the functions for memoization and specify the error tolerance percentage. For each function call-site, the framework creates a global table to store pairs of function arguments and output results. In case the function arguments are already stored in the table (within an error bound), the corresponding output results are returned. Otherwise, the function is accurately executed and the new input–output pairs are stored in the table. The ATM approach [67] performs runtime task memoization, relying on hashing functions to store the task inputs and an adaptive algorithm to automatically decide whether to use memoization or execute the task. The programmer needs to use pragmas to specify the tasks that are suitable for memoization. The authors of [68] introduce an approximate memoization mechanism for GPU fragment shading operations, which reduces the precision of the input parameters and performs partial matches. To identify approximate memoization opportunities, they characterize various fragment shader instructions in terms of memoization hits and output accuracy. Moreover, runtime policies are proposed to tune the precision according to the errors introduced.

Contrary to the aforementioned techniques, TAF-Memo [69] is an output-based function memoization technique, i.e., it memoizes function calls based on their output history. TAF-Memo checks for temporal locality by calculating the relative arithmetic difference of two consecutive output values from the same function call-site. In case this difference is below the acceptable error constraint, memoization is applied by returning the last computed output for the following function calls.

2.3.3. Relaxed Synchronization

The execution of parallel applications on multi/many-core systems requires time-consuming synchronization to either access shared data or satisfy data dependencies.

The literature includes various techniques [74–81] that relax the conventional synchronization requirements guaranteeing error-free execution, to improve the performance.

The authors of [74] propose the four-step RaC methodology to systematically relax synchronization, while always satisfying a programmer-defined QoS constraint. Initially, the programmer specifies the parallel code segments, and then applies the RaC methodology. This methodology identifies criteria for quantifying the acceptable QoS, selects the relaxation points, modifies the code to enable the execution of both the original and relaxed versions, and selects the suitable relaxation degree (i.e., which instances to relax for each synchronization point). Misailovic *et al.* [75] propose the Dubstep system, which relaxes the synchronization of parallelized programs based on a "find-transform-navigate" approach. Dubstep performs a profiling-based analysis of the original program to find possible optimizations, inserts opportunistic synchronization and barriers, and finally, performs an exploration including accuracy, performance and safety analysis.

QuickStep [76] is a system for approximately parallelizing sequential programs without preserving their semantics within statistical accuracy bounds. Among other transformations, QuickStep replicates shared objects to eliminate the bottlenecks of synchronized operations on them. HELIX-UP [77] is another parallelizing compiler that selectively relaxes strict adherence to program semantics to tackle runtime performance bottlenecks, involving profiling and user interaction to tune QoS. The compiler also offers a synchronization-relaxing knob to decrease the inter-core communication overhead by synchronizing sequential segments with prior iterations. More recently, the authors of [78] introduce PANDORA, which is an approximate parallelizing framework based on symbolic regression machine learning and sampled outputs of the original function. To avoid timing bottlenecks, such as data movement and synchronization, and improve parallelism, PANDORA eliminates loop-carried dependencies using fitness functions and constraints regarding error and performance. In [79], the authors exploit the concept of approximate shared value locality to reduce synchronization conflicts in programs using optimistic synchronization. The reduction of conflicts on approximately local variables, detected for a given similarity constraint, is achieved through an arbitration mechanism that imprecisely shares the values between threads. The authors of [80] apply aggressive coarse-grained parallelism on recognition & mining algorithms by relaxing or even ignoring data dependencies between different iterations. As a result, the timing overheads are reduced in comparison with the conventional parallel implementation, which also applies parallelization only within the iteration (iterations are executed serially). Rinard [81] introduces synchronization-free updates to shared data structures by eliminating the conventional use of mutual exclusion and dropping array elements at the worst scenario. Moreover, the same work applies relaxed barrier synchronization, allowing the threads to pass the barrier

without stalling to wait for the other threads.

2.3.4. Precision Scaling

Precision scaling (tuning) refers to the discipline reduction of the nominal numerical precision, resulting in improved calculation speed and/or memory bandwidth [128]. The state-of-the-art software-level works [82–97] address several challenges, such as the scaling degree, scaling automation, mixed precision, and dynamic scaling.

Starting with works based on formal methods to reduce the precision and examine the errors, the Gappa tool [82] automates the study of rounding errors in elementary functions and floating-point calculations using interval arithmetic. An extended version of this tool is Gappa++ [83], which provides automated analysis of numerical errors in a wide range of computations, i.e., fixed-point, floating-point, linear and non-linear. This tool integrates several features, such as operation rewriting to facilitate the isolation of rounding errors, and affine arithmetic to accurately bound linear calculations with correlated errors. FPTuner [84] is a tool that performs formal error analysis based on symbolic Taylor expansions and quadratically constrained quadratic programming. It searches for precision allocations that satisfy constraints such as the number of operators at a given precision and the number of type casts. Rosa [85] is a source-to-source compiler that combines satisfiability modulo theories with interval arithmetic to bound the round-off errors of the fixed- and floating-point formats.

Several works employ heuristics and automated search to scale the precision of floating-point programs. Precimonious [86] searches all the program variables in their order of declaration using the delta-debugging algorithm, and it lowers their precision according to an error constraint specified by the programmer. In the same context, HiFPTuner [87] firstly groups dependent variables that may require the same precision, and then it performs a customized hierarchical search. Lam *et al.* [88] introduce a framework that employs the breadth-first search algorithm to identify code regions that can tolerate lower precision. Similar to this technique, CRAFT [89] performs binary searches to initially determine the required program precision, and then truncate the results of some of the floating-point instructions. Towards the detection of large floating-point errors, the authors of [90] propose S^3FP. This tool is based on an heuristic-guided search to find the inputs causing the largest errors. The Blame Analysis [91] combines concrete and shadow execution to generate a blame set for the program instructions, which contains the minimum precision requirements under a giver error constraint. This approach can be also used in cooperation with the previous search-based works, and specifically, as pre-processing to reduce the search space.

Schkufza *et al.* [129] treat the scaling of floating-point precision as a stochastic search problem. In particular, they repeatedly apply random program transformations and use a robust search to guarantee the maximum errors.

The concept of dynamic precision scaling, i.e., the precision tuning at runtime with respect to the input data and error tolerance, has been studied in [92]. The dynamic scaling framework of this work integrates an offline application profiler, a runtime monitor to track workload changes, and an accuracy controller to adjust the precision accordingly. ApproxMA [93] dynamically scales the precision of data memory accesses in algorithms such as mixture model-based clustering. This framework integrates a runtime precision controller, which generates custom bit-widths according to the QoS constraints, and a memory access controller, which loads the scaled data from memory. The custom bit-widths are generated by analyzing a subset of data and intermediate results and calculating metrics regarding the error appearance and the number of tolerable errors.

Mixed floating-point precision has also been studied in high-performance computing workloads. ADAPT [94] uses algorithmic differentiation, i.e., a technique for numerically evaluating the derivative of a function corresponding to the program, to estimate the output error of high-performance computing workloads. It provides a precision sensitivity profile to guide the development of mixed-precision programs. The authors of [95] propose an instruction-based search that explores information about the dynamic program behaviour and the temporal locality.

To enable mixed floating-point precision in GPUs, the authors of [96] propose the GPUMixer tool, which relies on static analysis to find code regions where precision scaling improves the performance. Next, GPUmixer performs a dynamic analysis involving shadow computations to examine if the scaled program configurations satisfy the accuracy constraints. In the same context, PreScaler [97] is an automatic framework that generates precision-scaled OpenCL programs, considering both kernel execution and data transfer. Initially, it employs a system inspector to collect information about precision scaling on the target platform, and an application profiler to identify memory objects with floating-point elements for potential scaling. This information is exploited by a decision maker, which finds the best scaling configuration using decision tree search on a minimized space.

2.3.5. Data Sampling

Approximate Computing is also exploited in big data analysis, in an effort to reduce the increased number of computations due to the large amount of data. The key idea is to perform computations on a representative data sample instead of the entire dataset. In this context, data sampling methods [98–109] provide real-time processing

with error bounds in applications involving stream analytics, database search, and model training.

EARL [98] is an extension of Hadoop (i.e., a software framework that provides distributed storage and big data processing on clusters), which delivers early results with reliable error bounds. It applies statistics-based uniform sampling from distributed files. Goiri *et al.* [99] propose the ApproxHadoop framework to generate approximate MapReduce programs based on task dropping and multi-stage input sampling. They also bound the errors using statistical theories. The programmer tunes the approximation by specifying either the ratio of task dropping and/or input sampling or the desired error bound. Similarly, ApproxSpark [100] performs sampling at multiple arbitrary points of long chains of transformations to facilitate the aggregation of huge amounts of data. This framework models the clustering information of transformations as a data provenance tree, and then it computes the approximate aggregate values as well as error thresholds. Moreover, the sampling rates are dynamically selected according to programmer-specified error thresholds.

Sampling methods have also been examined in stream analytics. StreamApprox [101] is an approximate stream analytics system that supports batched and pipelined stream processing. It employs two sampling techniques (stratified and reservoir sampling) to approximate the outputs with rigorous error bounds. IncApprox [102] combines approximate and incremental computations to provide stream analytics with bounded error. This system executes a stratified sampling algorithm that selects data for which the results have been memoized from previous runs, and it adjusts the computations to produce an incrementally updated output. On the other hand, PrivApprox [103] combines sampling and randomized response to provide approximate computations and privacy guarantee. This system integrates a query execution interface that systematically explores the trade-off between accuracy and query budget. ApproxIoT [104] facilitates approximate stream analytics at the edge by combining stratified reservoir sampling and hierarchical processing.

A variety of sampling methods have been employed in approximate query processing systems for databases. BlinkDB [105] performs approximate distributed query processing, supporting SQL-based aggregation queries with time and error constraints. It creates stratified samples based on past queries and uses an heuristic-based profiler to dynamically select the sample that meets the query's constraints. Another system applying approximate big-data queries is Quickr [106], which integrates operators sampling multiple join inputs into a query optimizer, and then it searches for an appropriate sampled query plan. Sapprox [107] is a distribution-aware system that employs the occurrences of sub-datasets to drive the online sampling. In particular, the exponential number of sub-datasets is reduced to a linear one using a probabilistic map, and then, cluster sampling with unequal probability theory is applied for sub-

dataset sampling. Sapprox also determines the optimal sampling unit size in relation with approximation costs and accuracy.

Numerous works use data sampling to decrease the increased computational cost of model training for machine learning applications. Zombie [108] is a two-stage system that trains approximate models based on clustering and active learning. The first stage applies offline indexing to organize the dataset into index groups of similar elements. Afterwards, the stage of online querying uses the index groups that are likely to output useful features to create the training subset of data. BlinkML [109] approximately trains a model on a small sample, while providing accuracy guarantees. The sample is obtained through uniform random sampling, however, in case of very large datasets, a memory-efficient algorithm is employed.

2.3.6. Approximate Programming Languages

The high-level approximation of software programs has been examined through approximate programming languages, i.e., language extensions that allow the programmer to systematically declare approximate code regions, variables, loops, and functions, insert randomness in the program, and/or specify error constraints. The literature involves numerous works [30, 110–127] that enable approximate procedural, object-oriented, and probabilistic programming.

Ansel *et al.* [110] introduce a set of PetaBricks language extensions that allow the programmer to write code of variable accuracy. These extensions expose the performance–accuracy trade-off to the compiler, which automatically searches the algorithmic space to tune the program according to the programmer's accuracy constraints. Eon [111] is a programming language that allows the programmer to annotate program flows (paths) with different energy states. The Eon runtime system predicts the workload and energy of the system, and then it adjusts the execution of flows according to the programmer's declarations and the energy constraints. In the same context, Baek and Chilimbi [112] propose Green, which is a two-phase programming framework providing language extensions to approximate expensive functions and loops. The programmer uses pragma-like annotations to specify approximate variants of functions. In the calibration phase, Green builds a model to quantify the QoS loss and the performance/energy gains. This model is then used in the operational phase to generate an approximate program satisfying the programmer's QoS constraint. DECAF [113] is a type-based approximate programming language that allows the programmer to specify the correctness probability for some of the program variables. The DECAF type system also integrates solver-aided type inference to automatically tune the type of the rest variables, code specialization, and dynamic typing.

Flikker [116] provides language annotations to mark the program variables and partition the data into critical and non-critical regions (stored in unreliable memories). Topaz [117] is a task-based language that maps tasks onto approximate hardware and uses an outlier detector to find and re-execute the computations producing unacceptable results.

In [30], the authors introduce language constructs for generating approximate programs and proof rules for verifying the acceptability properties. Rely [114] is an imperative language that allows the programmer to introduce quantitative reliability specifications for generating programs with data stored in approximate memory and inexact arithmetic/logical operations. Chisel [115] automates the selection of Rely's approximations while satisfying the programmer-defined reliability and accuracy constraints. To solve this optimization problem, Chisel employs an integer programming solver. All these works include safety analysis and program verification for sequential programs. In contrast, Parallely [126] is a programming language for approximating parallel programs through canonical sequentialization, i.e., a verification method that generates sequential programs capturing the semantics of parallel programs.

Targeting approximations in Java programs, the authors of [118] propose EnerJ, i.e., a language extension providing type qualifiers to specify data that can be approximately stored or computed. EnerJ guarantees isolation of the approximate computations. FlexJava [119] offers another set of language extensions to annotate approximate programs. Using an approximation safety analysis, FlexJava automates the approximation of data and operations while ensuring safety guarantees. ExpAX [120] allows the programmer to explicitly specify error expectations for a subset of Java. Based on an approximation safety analysis, it identifies operations that are candidate for approximation, and then, an heuristic-based framework approximates those that statistically satisfy the error expectations.

Significant research has also been conducted on probabilistic programming languages. Church [121] is a probabilistic language that inserts randomness on a deterministic function subset using stochastic functions. The Church semantics are defined in terms of evaluation histories and conditional distributions on the latter. Similarly, Venture [122] is another language that enables the specification of probabilistic models and inference problems. The Anglican [123] language and runtime system provides probabilistic evaluation model and functional representations, e.g., distributions and sequences of random variables.

Uncertain<T> [124] is a language abstraction that manipulates data as probability distributions. Random variables are declared as "uncertain" and a Bayesian network for representing computations is build, where nodes correspond to the variables and edges correspond to conditional variable dependencies. The Uncertain<T> runtime

system performs hypothesis tests and sampling to evaluate the network. Similarly, Sampson *et al.* [125] use probabilistic assertions on random variables. Their tool, called MayHap, performs probabilistic evaluation by statically building a Bayesian representation network based on the input distribution and dynamically interpreting it via sampling. In the same context, AxProf [127] is a profiling-based framework for analyzing randomized approximate programs. The programmer specifies probabilistic predicates for the output, i.e., regarding the expectation of the output value and/or the probability that the output satisfies a condition, and AxProf generates approximate programs based on statistical tests.

2.4. Classification of Hardware Approximation Techniques

In this section, we classify and introduce the hardware approximation techniques, which are applied at the lower level of the design abstraction hierarchy. These techniques aim to improve the area, power consumption, and performance of the circuits i.e., the basic building blocks of accelerators, processors, and computing platforms. The hardware approximation techniques can be categorized into three classes: (i) *Circuit Functional Approximation (CFA)*, (ii) *Voltage Over-Scaling (VOS)*, and (iii) *Over-Clocking (OC)*. In approximate hardware, we distinguish two types of errors: the *functional errors* (produced by CFA) and the *timing errors* (produced by VOS and OC). Figure 2.2 illustrates the hardware approximation techniques, including a
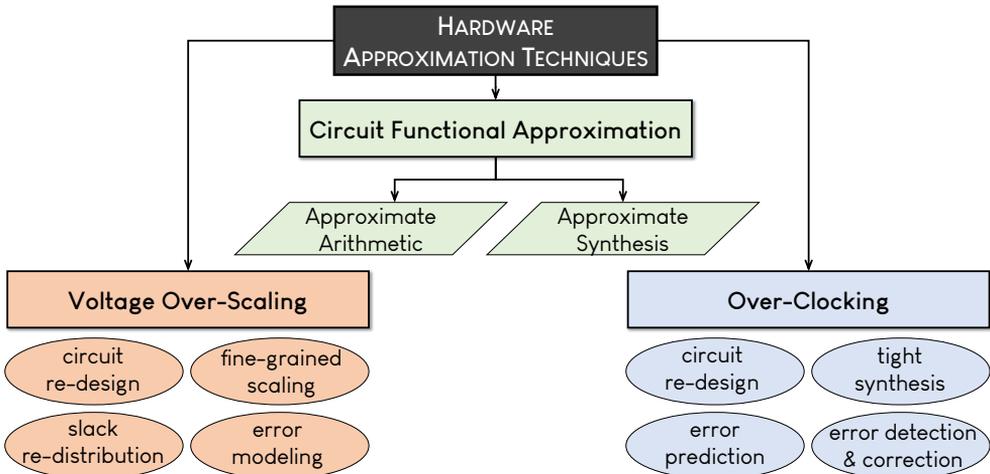
Figure 2.2: Classification of state-of-the-art hardware approximation techniques.

Table 2.3: Classification of hardware approximation techniques.

HW Approximation Class	Technique/Approach
Adder Approximation	use of approximate full adder cells [130–133]
	segmentation and carry prediction [134–141]
Multiplier Approximation	truncation and rounding [142–148]
	approximate radix encodings [149–153]
	use of approximate compressors [154–158]
	logarithmic approximation [159–162]
Divider Approximation	bit-width scaling [163, 164]
	use of approximate adder/subtractor cells [165–168]
	simplification of computations [169–173]
Approximate Synthesis	structural netlist transformation [174–178]
	Boolean rewriting [179–182]
	high-level approximate description [183–187]
	evolutionary synthesis [188–192]
Voltage Over-Scaling	slack re-distribution [193]
	circuit re-design and architecture modification [194–196]
	fine-grained scaling [197–199]
	error modeling [200–204]
Over-Clocking	tight synthesis [205]
	circuit re-design and architecture modification [206–208]
	error detection & correction [209–211]
	error prediction [212–215]

further taxonomy to sub-classes.

In the remainder of this section, we present state-of-the-art works, organized according to the proposed classification. Table 2.3 summarizes the state-of-the-art hardware approximation techniques. We also note that some works belong to more than one sub-class, however, we opt to assign them to the sub-class of their prevalent technique and highlight the relevant features.

2.4.1. Circuit Functional Approximation

Circuit functional approximation modifies the original accurate design by reducing its circuit complexity at logic level. Typical CFA approaches include: (i) the modification of the circuit's truth table, (ii) the use of an approximate version of the initial hardware algorithm, (iii) the use of small inexact components as building blocks, and

(iv) approximate circuit synthesis. The main target of CFA is the arithmetic circuits [216], as they constitute the key processing units of processors and accelerators, and thus, they inherently affect the power efficiency and performance of the entire system. Interestingly, the literature provides several open-source libraries of approximate arithmetic circuits, such as ApproxAdderLib [138], EvoApprox8b [190] and SMApproxLib [217]. In this literature review, we focus on approximate adders, multipliers, and dividers. However, we note that numerous works design and evaluate other approximate arithmetic operations, such as circuits for multiply-accumulate [218, 219], square root [164], squaring [220], square-accumulate [221], and Coordinate Rotation Digital Computer (CORDIC) [222]. Moreover, the literature includes automated methods for generating approximate circuits, which are presented in the context of approximate logic synthesis.

Approximate Adders

Significant research has been conducted on the design of approximate area- and power-efficient adders. The approximation techniques for inexact adders involve: (i) *use of approximate full adder cells* [130–133] and (ii) *segmentation and carry prediction* [134–141]. In the following, we present representative state-of-the-art works with approximate adders.

The IMPACT adders are based on inexact full adder cells, which are approximated at the transistor level to deliver up to 45% area reduction [130]. Another transistor-level cell approximation is proposed in [131], where the AXA 4-transistor XOR/XNOR-based adders are implemented, delivering up to 31% gain in dynamic power consumption. Moreover, in [132], approximate reverse carry-propagate full adders are used to build the hybrid RCPA adders. Targeting higher level approximations, the OLOCA adder splits the addition into accurate and approximate segments [133], and for the latter, it employs OR gates for the most-significant bit additions and outputs constant '1' for the least-significant ones. To reduce the worst-case carry propagation delay, Kim *et al.* [134] propose a carry prediction scheme leveraging the less-significant input bits, which is 2.4× faster than the conventional ripple-carry adder. Similarly, Hu *et al.* [135] introduce a carry speculating method to segment the carry chain in their design, which also performs error and sign correction. Compared to the accurate adder, the proposed design is 4.3× faster and saves 47% power.

The quality constraint of applications may vary during runtime, thus, research efforts have also focused on designing dynamically configurable adders, which can tune their accuracy. In [136], the authors propose an accuracy-configurable adder, called ACA, which consists of several sub-adders and an error detection & correction module. The

design controls the accuracy at runtime and can operate in accurate mode. Another dynamically configurable adder, called GDA, is proposed in [137], where multiplexers select the carry input either from the previous sub-adder or the carry prediction unit, providing a more graceful degradation of the accuracy. In the same direction, the GeAr adder employs multiple sub-adders of equal length to variable approximation modes [138]. This architecture also supports accurate mode via a configurable error correction unit.

More recently, Akbari *et al.* [139] introduce the RAP-CLA adder, which splits the conventional carry look-ahead scheme into two segments, i.e., the approximate part and the augmenting part, supporting approximate and accurate mode. When operating at the approximate mode, the augmenting part is power-gated to reduce power consumption. Another carry-prediction-based approach supporting both modes is the SARA design [140]. This adder uses carry ripple sub-adders, and the carry prediction does not require a dedicated circuitry. Finally, the BSCA adder, which is based on a block-based carry speculative approach [141], integrates an error recovery unit and non-overlapped blocks consisting of a sub-adder, a carry prediction unit, and a selection unit.

Approximate Multipliers

The multiplication circuits have attracted significant interest from the research community. The literature includes a plethora of inexact multipliers, which can be categorized according to the prevailing approximation techniques: (i) *truncation and rounding* [142–148], (ii) *approximate radix encodings* [149–153], (iii) *use of approximate compressors* [154–158], and (iv) *logarithmic approximation* [159–162]. Subsequently, we introduce the state-of-the-art works from each category.

Starting with the rounding and truncation techniques, the DRUM multiplier [142] dynamically reduces the input bit-width, based on the leading '1' bits, to achieve 60% power gain in exchange for mean relative error of 1.47%. Zendegani *et al.* [143] propose the RoBa multiplier, which rounds the operands to the nearest exponent-of-two and performs a shift-based multiplication in segments. In [144], the PR approximate multiplier perforates partial products and applies rounding to the remaining ones, delivering up to 69% energy gains. The same approximation technique is integrated in the mantissa multiplication of floating-point numbers to create the AFMU multiplier [145]. Vahdat *et al.* [146] propose the TOSAM multiplier, which truncates the input operands according to their leading '1' bit. To decrease the error, the truncated values are rounded to the nearest odd number. In [147], different rounding, perforation and encoding schemes are combined to extract the most energy-efficient designs. Finally, Frustaci *et al.* [148] implement an alternative dynamic truncation

with correction, along with an efficient mapping for the remaining partial product bits.

Next, we present multipliers generating their partial products based on approximate radix encodings. Liu *et al.* [149] modify the Karnaugh map of the radix-4 encoding to create approximate encoders for generating the least-significant partial product bits. A similar approach is followed in [150], where approximate radix-4 partial product generators are designed. Jiang *et al.* [151] use an approximate adder to generate the $\pm 3\times$ multiplicand term in the radix-8 multiplier. In [152], the authors propose a hybrid low-radix encoding, which encodes the most-significant bits with the accurate radix-4 encoding and the least-significant bits with the proposed approximate radix-8 encoding. Correspondingly, the hybrid high-radix encoding, i.e., accurate radix-4 and approximate radix-2^k, is examined in [153].

Several works employ approximate compressors for the partial product accumulation. Momeni *et al.* [154] modify the truth table of the accurate 4:2 compressor to create two simplified designs and use them in the Dadda multiplier. The authors in [155] design 4:2 compressors, again for Dadda multipliers, which can switch between accurate and approximate mode at runtime, providing 68% lower power consumption. In [156], an approximate 4:2 compressor is implemented in FinFET based on a three-input majority gate, and then it is used in the Dadda architecture along truncation. Esposito *et al.* [157] introduce a new family of approximate compressors and assign them to each column of the partial product matrix according to their allocation algorithm. Another interesting work is the design of approximate compressors for multipliers using the concept of stacking circuit [158].

Regarding the approximate logarithmic multipliers, Liu *et al.* [159] employ a truncated binary-logarithm converter and inexact adders for the mantissa addition to design the ALM family of multipliers. The logarithmic-based REALM multiplier [160] partitions the power-of-two intervals of the input operands into segments, and determines an error compensation factor for each one. The ILM multiplier [161] differentiates from the conventional design, as it rounds the input operands to their nearest power-of-two using a nearest '1' bit detector. Pilipovic *et al.* [162] propose a two-stage trimming logarithmic multiplier, which reduces at first the bit-width of the input operands, and then the bit-width of the mantissas.

Approximate Dividers

The division circuits have received less attention than the other arithmetic circuits (adders and multipliers), even though they feature complex calculations. Nevertheless, the literature provides numerous works aiming to reduce the large critical paths of the conventional dividers. The approximation techniques for the di-

vision circuits can be categorized as follows: (i) *bit-width scaling* [163, 164], (ii) *use of approximate adder/subtractor cells* [165–168], and (iii) *simplification of computations* [169–173].

The first class of approximation techniques uses exact dividers with reduced bit-width. The approximate divider of [163] dynamically selects the most relevant bits from the input operands and performs accurate division at lower bit-width, providing up to 70% power gains in exchange for 3% average error. The design makes use of leading '1' bit detectors, priority encoders, multiplexers, subtractor and barrel shifter. Similarly, the AAXD divider of [164] detects the leading '1' bits and uses a pruning scheme to extract the bits that will be given as input to the divider. Additionally, the design integrates an error correction unit to form the final output.

Regarding the second class of approximation techniques, Chen *et al.* [165] perform the subtraction of the non-restoring array divider with inexact subtractor circuits employing pass transistor logic. For the proposed divider, called AXDnr, the authors examine different schemes with regard to which subtractions of the division array to approximate. Similarly, in the AXDr divider of [166], some of the subtractions of the restoring array divider are performed with inexact subtractor circuits. The use of inexact cells has also been examined in the high radix SRT divider [167]. In this divider, called HR-AXD, the inexact cell is a signed-digit adder, which is employed based on different replacement schemes and along with cell truncation and error compensation. More recently, Adams *et al.* [168] introduce two approximate division architectures, called AXRD-M1 and AXRD-M2, which deliver up to 46% area and 57% power gains, respectively, compared to the exact restoring divider. The first design replaces some of the restoring divider cells with inexact ones of simplified logic, while the second one involves the elimination of some rows of the divider.

Targeting to perform the division with simplified computations, the SEERAD divider [169] rounds the divisor to a specific form based on the leading '1' bit position, and thus, the division is transformed to shift-&-add multiplication. Similarly, Vahdat *et al.* [170] propose the TruncApp divider, which replaces the division with the multiplication of the truncated dividend by the approximate inverse divisor. In the same context, the CADE divider of [171] performs the floating-point division by subtracting the input mantissas. To compensate a large error, which is estimated by analyzing the most-significant input bits, a pre-computed value is retrieved from memory. In [172], the proposed AXHD divider approximates the least-significant computations of the division using a non-iterative logarithmic approach that is based on leading '1' bit detection and subtraction of the logarithmic mantissas. Finally, Saadat *et al.* [173] propose approximate integer and floating-point dividers with near-zero error bias, called INZeD and FaNZeD, respectively, by combining an error correction method

with the classical approximate logarithmic divider.

Approximate Synthesis

An automated approach to generate inexact circuits is the approximate logic synthesis. This method provides increased approximation diversity, i.e., multiple approximate variants of circuits, without relying on the manual approximation inserted by the designer, such as in the case of the aforementioned arithmetic approximations. Another benefit of approximate synthesis is that several techniques generate the approximate variant that leads to the maximum hardware gains for a given approximation/error constraint. The state-of-the-art techniques can be categorized as follows [223]: (i) *structural netlist transformation* [174–178], (ii) *Boolean rewriting* [179–182], (iii) *high-level approximate description* [183–187], and (iv) *evolutionary synthesis* [188–192].

Several works of the literature employ a direct acyclic graph to represent the circuit netlist, where each node corresponds to a gate. In this context, the GLP technique [174] prunes nodes with an iterative greedy approach according to their impact on the final output and their toggle activity. In contrast, the CC framework [175] performs an exhaustive exploration of all possible node subsets that can be pruned without surpassing the error constraint. Venkataramani *et al.* [176] propose SASIMI, which is based on a greedy heuristic to find signal pairs assuming the same value and substitute one with the other. This automatic synthesis framework guarantees that the user-defined quality constraint is satisfied and generates accuracy configurable circuits. To apply stochastic netlist transformation, the SCALS framework [177] maps an initial gate-level network to the targeted technology (standard cell or FPGA), and then it iteratively extracts sets of sub-netlists and inserts random approximations in them. These sub-netlists are evaluated using statistical hypothesis testing. More recently, Castro-Codinez *et al.* [178] propose the AxLS framework, which converts the Verilog netlist to XML format and then applies typical transformation techniques, e.g., gate pruning, according to an error threshold.

The second category includes techniques that apply approximations in a formal Boolean representation of the circuit before it is synthesized. The SALSA approach [179] encodes the error constraints into a quality logic function, which compares the outputs of the accurate and approximate circuits. Towards logic simplification, SALSA computes the "observability don't cares" for each output of the approximate circuit, i.e., the set of input values for which the output is insensitive. In the same direction, but for sequential circuits, Ranjan *et al.* [180] introduce ASLAN. This framework generates several approximate variants of the combinational blocks, and then it identifies the best approximations for the entire sequential circuit based on

a gradient-descent approach. Miao *et al.* [181] use a two-phase Boolean minimization algorithm to address the problem of approximate synthesis. The first phase solves the problem under a given constraint for the error magnitude, and the second phase iteratively finds a solution that also satisfies the error frequency constraint. In an iterative fashion, the BLASYS methodology [182] partitions the circuit into smaller circuits, and for each one, it generates an approximate truth table based on Boolean matrix factorization. The approximate sub-circuits are synthesized and the trade-off between error and power/area efficiency for the entire circuit is evaluated.

Regarding approximations introduced at the hardware description level, Yazdanbakhsh *et al.* [183] propose the Axilog language annotations, which provide syntax and semantics for approximate design and reuse in Verilog. Axilog allows the designer to partition the design into accurate and approximate segments. ABACUS [184] is another interesting work, which parses the behavioral/RTL Verilog description of the design to create its abstract syntax tree. Next, a set of diverse transformations is applied to the tree to create approximate variants, which are then written in Verilog. An expanded version of ABACUS is introduced in [186], where sorting-based evolutionary algorithms are employed for design space exploration. Moreover, the new ABACUS version focuses on approximations in critical paths to facilitate the reduction of the supply voltage. Lee *et al.* [185] generate approximate designs in Verilog from C accurate descriptions. The proposed framework computes data statistics and mobility information for the given design and employs an heuristic solver for optimizing the energy–quality trade-off. Targeting to high-level synthesis, the AxHLS approach [187] performs a design space exploration, based on analytical models, to identify the best arithmetic approximations for a given error constraint. Starting from a C description, AxHLS adopts scheduling and binding operations to apply the approximations provided by the exploration and generate the Verilog code.

The fourth class of techniques for automated synthesis of approximate circuits is based on evolutionary algorithms, i.e., heuristic-based search algorithms that treat circuit approximation as multi-objective optimization problem and generate a set of solutions. In this context, Sekanina *et al.* [188] use Cartesian genetic programming to minimize the error in adders considering the number of logic gates as constraint. This approach is extended in [189], where approximate multipliers and median filters are evolved through randomly seeded Cartesian genetic programming. Based on the same utilities, the authors of [190] propose the EvoApprox8b library of approximate adders and multipliers. This library is generated by examining various trade-offs between accuracy and hardware efficiency, and it offers different approximation variants and circuit architectures. In [191], a search-based technique for evolutionary circuit synthesis for FPGAs is proposed. In particular, this approach

represents the circuit as a directed acyclic graph, and re-synthesizes approximate configurations based on Cartesian genetic programming. Vasicek *et al.* [192] adjust the approximation degree according to the significance of the inputs. To do so, they adopt a weighted error metric to determine the significance of each input vector and use Cartesian genetic programming to minimize the circuit's area while satisfying a threshold.

2.4.2. Voltage Over-Scaling

Voltage over-scaling reduces the circuit's supply voltage below its nominal value, while keeping the clock frequency constant. The circuit operation at a lower voltage value produces timing errors due to the failure of the critical paths to meet the delay constraints. Nevertheless, considering that power consumption depends on the voltage value, VOS techniques are continuously examined in the literature. An exploration and quantification of the benefits and overheads of VOS is presented in [224]. Research involving VOS can be classified in the following categories: (i) *slack re-distribution* [193], (ii) *circuit re-design and architecture modification* [194–196], (iii) *fine-grained scaling* [197–199], and (iv) *error modeling* [200–204].

Kahng *et al.* [193] shift the timing slack of the frequently executed near-critical paths through slack redistribution, and thus, they reduce the minimum voltage at which the error rate remains acceptable. The proposed technique is based on post-layout cell resizing to deliver the switching activity-aware slack redistribution. More specifically, a heuristic finds the voltage satisfying the desired error rate, and then it increases the transistor width of the cells to optimize the frequently executed paths.

In [194], the authors optimize building blocks for more graceful degradation under VOS, using two techniques, i.e., dynamic segmentation & error compensation and delay budgeting of chained datapath. The first technique bit-slices the datapath of the adder and employs a multi-cycle error correction circuitry that tracks the carries. The second technique adds transparent latches between chained arithmetic units to distribute the clock period. To facilitate VOS, Chen *et al.* [195] build their designs on the residue number system, which provides shorter critical paths than conventional arithmetic. They also employ the reduced precision redundancy scheme to eliminate the timing errors. Another interesting work is Thundervolt [196], which provides error recovery in the Multiply-And-Accumulate (MAC) units of systolic Deep Neural Network (DNN) accelerators. To detect timing errors, Thundervolt employs Razor shadow flip-flops. In case an error occurs in a MAC, a multiplexer forwards the previous MAC's accurate partial sum (stored in the Razor flip-flop) to the next MAC.

Targeting fine-grained VOS solutions, i.e., different voltages across the same circuit architecture, Pandey *et al.* propose GreenTPU [197]. This design integrates a timing error control unit in each MAC row of the systolic array, which stores input sequences producing timing errors. As a result, when such an input sequence pattern is identified, the voltage of the MAC is scaled accordingly to prevent timing errors. In the same context, the authors of [198] propose NN-APP. This framework analyzes the error propagation in neural networks to model the impact of VOS on accuracy. Based on this analysis, as well as an error resilience study for the neurons, NN-APP uses a voltage clustering method to assign the same voltage to neurons with similar error resilience. Another fine-grained VOS approach is proposed in [199]. This framework provides voltage heterogeneity by using a greedy algorithm to solve the optimization problem of grouping and assigning the voltage of arithmetic units to different islands.

The analysis of errors in circuits under VOS is considered a key factor, as it guides the aggressiveness of voltage scaling towards the acceptable error margins. In [200], an analytical method to study the errors in voltage over-scaled arithmetic circuits is proposed. Similarly, the authors of [201] introduce a probabilistic approach to model the errors of the critical paths. In the same category, we include works relying on simulations to analyze the errors of VOS. Ragavan *et al.* [202] characterize arithmetic circuits in terms of energy efficiency and errors using transistor-level SPICE simulation for various voltages. Based on this characterization, they propose a statistical model to simulate the behavior of arithmetic operations in VOS systems. Exploiting the machine learning methods, Jiao *et al.* [203] propose LEVAX to model voltage over-scaled functional units. This input-aware model is trained on data from gate-level simulations to predict the timing error rate for each output bit. To provide accurate VOS-aware gate-level simulation, Zervakis *et al.* propose VOSsim [204]. This framework performs an offline characterization of the flip-flop for timing violations and calculates the cell delays for the targeted voltage, enabling gate-level simulation under VOS.

2.4.3. Over-Clocking

Over-clocking (or frequency over-scaling) configures the circuit/system at higher clock frequencies than those that respect the critical paths. As a result, timing errors are induced in exchange for increased performance. A trade-off analysis between accuracy and performance when over-clocking FPGA-based designs is presented in [225]. In the same work, the authors show that OC outperforms the traditional bit truncation for the same error constraint. For our analysis, we consider that the state-of-the-art works of the domain focus on the following directions: (i) *tight synthesis* [205], (ii) *circuit*

re-design and architecture modification [206–208], (iii) *error detection & correction* [209–211], and (iv) *error prediction* [212–215].

The first approach aims to reduce the timing errors of OC by optimizing the critical paths of the design. In this context, the SlackHammer framework [205] synthesizes circuits with tight delay constraints to reduce the number of near-critical paths, and thus, decrease the probability of timing errors when frequency is over-scaled. At first, Slack-Hammer isolates the paths and identifies potential delay optimizations. Based on the isolated path analysis, the framework performs an iterative synthesis with tighter constraints for the primary outputs with negative slack.

The second class of techniques modifies the conventional circuit architecture to facilitate frequency OC and increase the resilience to timing errors. The retiming technique [206] re-defines the boundaries of combinational logic by moving the flip-flops backward or forward between the stages. Based on this circuit optimization, the synthesis is relaxed by ignoring the paths that are bottleneck to minimum period retiming. Targeting different circuit architectures, Shi *et al.* [207] adopt an alternative arithmetic, called online, and show that online-based circuits are more resilient to the timing errors of OC than circuits with traditional arithmetic. The modification of the initial neural network model to provide resilience in timing errors has also attracted research interest. In this direction, Wang *et al.* [208] propose an iterative reclocking-and-retraining framework for operating neural network circuits at higher frequencies under a given accuracy constraint. The clock frequency is gradually increased and the network's weights are updated through back-propagation training until to find the maximum frequency for which the timing errors are mitigated and the accuracy constraint is satisfied.

Several works propose circuits for timing error detection & correction, enabling the use of over-clocking. These techniques either improve the frequency value of the first failure, i.e., the first timing error, or reduce the probability of timing errors. TIM-BER [209] masks timing errors by borrowing time from successive pipeline stages. According to this approach, the use of discrete time-borrowing flip-flops and continuous time-borrowing latches slows down the appearance of timing errors with respect to the frequency scaling. Ragavan *et al.* [210] detect and correct timing errors by employing a dynamic speculation window on the double-sampling scheme. This technique adds an extra register (called shadow and clocked by a second "delayed" clock) at the end of the pipelined path to sample the output data at two different time instances. The proposed approach also uses an online slack measurement to adaptively over-clock the design. The TEAI approach [211] is based on the locality of the timing errors in software-level instructions, i.e., the tendency of specific instructions to produce timing errors. TEAI identifies these instructions at runtime, and sends error alarms to hardware, which is equipped with error detection & correction

circuits.

There is also wide research on the prediction of the timing errors in advance, which allows to over-scale the frequency according to the acceptable error margins. In [212], the authors introduce an instruction-level error prediction system for pipelined micro-processors, which stalls the pipeline when critical instructions are detected. Their method is based on gate-level simulations to find the critical paths that are sensitized during the program execution. Similarly, Constantin *et al.* [213] obtain the maximum delays for each arithmetic instruction through gate-level simulations, and they dynamically exploit timing margins to apply frequency over-scaling.

Besides instruction-level prediction models, there are numerous works that build models based on machine learning and simulations of functional units. A representative work of this approach is WILD [214], which builds a workload-dependent prediction model using logistic regression. In the same direction, SLoT [215] is a supervised learning model that predicts timing errors based on the inputs and the clock frequency. At first, SLoT performs gate-level simulation to extract timing class labels, i.e., "timing error" or "no timing error", for different inputs and frequencies. These classes are then used, along with features extracted from random data pre-processing, to train the error prediction model.

Part I.

Arithmetic Approximation Techniques for Circuit Design

Prologue

Dissertation's Part I focuses on the arithmetic of circuits and accelerators. At first, it proposes logic-level arithmetic approximation techniques, and then, it presents the development of approximate hardware accelerators. Chapters 3–7 examine different aspects of computer arithmetic and approximate circuit design: alternative numerical formats in Chapter 3, hybrid radix encodings in Chapter 4, dynamic/runtime approximation configuration in Chapter 5, cooperative approximation in Chapter 6, and systematic design of approximate accelerators in Chapter 7.

Chapter 3 highlights the benefits of applying sophisticated bit-level optimizations. Chapter 4 proposes a hybrid high-radix encoding that is used to design the RAD family of approximate multipliers. Chapter 5 proposes runtime-configurable approximate multipliers for fixed-point (DyFXU design family) and floating-point arithmetic (DyFPU design family). Chapter 6 examines the combination of arithmetic approximation techniques, resulting in the state-of-the-art ROUP family of approximate multipliers. Finally, Chapter 7 integrates all the proposed approximate circuits in the design of ASIC/FPGA accelerators for DSP and AI applications.

Chapter 3

Arithmetic Optimization:
Double-LSB Encoding

Computer arithmetic has received significant attention, as it inherently affects the efficiency and performance of the hardware accelerators. In this context, novel numerical formats are examined, targeting to provide increased resource gains compared to the conventional binary formats in various application domains, e.g., in Digital Signal Processing (DSP). In this chapter, we adopt the Double Least Significant Bit (DLSB) format, where the numbers have an extra least significant bit, and we apply optimizations in the multiplication, i.e., one of the most resource-hungry DLSB operations. The DLSB arithmetic delivers several benefits, such as the symmetric representation range, the number negation performed only by bitwise inversion, the improvement of the residue number circuits, and the improvement of the rounding process in floating-point calculations. However, all these advantageous features come with some penalties in the design of the arithmetic DLSB units. Towards reducing the DLSB overheads in the multiplication, we propose an energy-efficient scheme for multiplying 2's-complement binary numbers, which is based on sophisticated bit-level manipulations. The overhead of the proposed DLSB design is negligible compared to the conventional design for ordinary 2's-complement numbers, i.e., ~3 % area/energy overhead on average for different multiplier sizes. Moreover, our design outperforms the straightforward design approach by providing 4×–5× less resource overhead. Finally, as case study, we demonstrate how the DLSB multiplier can be effectively used as building block for the implementation of larger multiplications, delivering 31 % area and 43 % energy gains.
*This chapter is based on our **publication** in [226].*

3.1. Introduction

The integration of novel numerical representation formats in the design of circuits appears as an effective solution for reducing the power/energy consumption, area and delay [227]. In this context, several alternative arithmetic formats have been proposed in the literature. The Residue Number System (RNS) [228] and the Logarithmic Number System (LNS) [229] are typical examples of non-standard representation formats that aim to improve the hardware efficiency of the arithmetic units. Another interesting format is the carry-save representation [230], which is mainly used to build fast adders with multiple inputs and accumulation trees for multipliers.

In this chapter, we focus on the novel *Double Least Significant Bit (DLSB)* arithmetic representation, which is proposed by Parhami [231]. In particular, the DLSB arithmetic adopts the conventional arithmetic and considers an additional Least Significant Bit (LSB), i.e., an extra bit that has the same weight as the original LSB. With this format, the number representation range becomes fully symmetric, i.e., for 2's-complement arithmetic, it is $[-2^{n-1}, 2^{n-1}]$ instead of $[-2^{n-1}, 2^{n-1} - 1]$. Other benefits of the DLSB arithmetic are the simpler number inversion, the simpler rounding process in floating-point calculations, and the increased efficiency in residue number operations.

The symmetry in the representation range allows to perform negation with only bitwise inversion, hence, the addition of '1' (performed in the conventional 2's-complement arithmetic) is eliminated. The latter is an important feature, especially in algorithms where the sign change is not followed by another addition [231]. Moreover, the symmetry of DLSB guarantees that all the numbers can be complemented without resulting in overflow.

In floating-point calculations, if the result has more bits than those supported by the format, then the extra bits are discarded and the remaining ones are adjusted with rounding. This process requires addition involving carry propagation, which is a task that can add significant delay to the floating-point arithmetic units. In the case of DLSB format, the rounding algorithm needs only to determine the value of the extra LSB in the rounded result [231]. Specifically, instead of performing the required addition of the regular rounding, the extra LSB of the result is set to '1' for rounding up, avoiding in this way the carry propagation.

The DLSB format has also been used in RNS operations to improve the design efficiency. In this direction, the DLSB encoding of the $2^n + 1$ residues is employed to design the modulo $2^n + 1$ adder [232] and multiplier [233]. The representation range of the unsigned numbers is $[0, 2^n]$, and thus, the end-around carry is stored and

there is no need for a post-increment operation. Furthermore, the authors of [234] propose reverse converters for two 4-moduli sets that are based on DLSB encodings. The derived results show that both the area and delay of the DLSB-based converters are comparable to that of the converters using the conventional binary encoding.

These advantageous features of the DLSB arithmetic come with some delay and area overheads, considering the corresponding conventional circuits as baseline. In [231], the delay and area theoretical penalties are reported for a variety of DLSB circuits (e.g., adders, different types of multipliers, dividers), showing that complex arithmetic units such as the multipliers exhibit the largest overheads. Furthermore, these theoretical overheads have not been actually evaluated with industrial synthesis toolflows and technology libraries. In this chapter, motivated by the benefits of the DLSB format, and targeting to eliminate the penalties that arise, we propose an improved algorithm for the multiplication of two 2's-complement DLSB numbers. We provide both theoretical and robust experimental evaluations on industrial tools, showing that sophisticated bit-level manipulations provide significant gains and decrease the penalties.

We further motivate the applicability and effectiveness of the proposed DLSB technique by examining a realistic case study design scenario. In particular, we show that large-size multiplications can be efficiently implemented using small DLSB multipliers as building blocks. The multiplication of large operands is a key component in various applications, e.g., cryptographic schemes and scientific calculations. To improve the hardware efficiency, significant research has been conducted on the implementation of large-size multiplications [235, 236]. The mapping of large-size multiplications in Field-Programmable Gate Arrays (FPGAs) is also examined in the literature [237]. In FPGAs, such multiplications are implemented by segmenting the input operands based on the bit-width of the hardwired multipliers that are integrated in the DSP blocks.

The **contribution** of this chapter is summarized as follows:

(i) We highlight the significance of computer arithmetic and show that novel numerical formats can provide valuable gains in hardware.

(ii) We propose an improved algorithm for the multiplication of two 2's-complement DLSB numbers, which is strictly defined by a rigorous theoretical analysis involving sophisticated bit-level manipulations.

(iii) We show that the proposed DLSB circuit outperforms its unoptimized counterpart, and also, it can be effectively used as building block for improving the implementation of large-size multiplications.

The remainder of this chapter is organized as follows. Section 3.2 introduces the DLSB arithmetic format, Section 3.3 presents the proposed DLSB multiplier, and Section 3.4 includes the theoretical and experimental evaluation. Finally, Section 3.5 draws the conclusions.

3.2. The Double Least Significant Bit Format

This section includes an introduction in the 2's-complement DLSB format [231]. Let $X = \langle x_{n-1} x_{n-2} \cdots x_0 \rangle_{2\text{'s}}$ be a n-bit 2's-complement number. X is converted to a 2's-complement DLSB number by attaching an extra LSB (x_{0+}) next to the original LSB, as shown in Eq. (3.1).

$$X^+ = X + x_{0+} = \langle x_{n-1} x_{n-2} \cdots x_0 \rangle_{2\text{'s}} + x_{0+} \tag{3.1}$$

For the rest of the discussion, we assume that X^+ is a n-bit 2's-complement DLSB number (the extra LSB is not calculated in the bit-width). Moreover, the following notations are used: $\langle \ldots \rangle_{2\text{'s}}$ denotes a 2's-complement number, and $\langle \ldots \rangle_{\text{u}}$ denotes an unsigned number. Subsequently, we present some numerical examples.

Example 1. The DLSB number $\langle 0111 \rangle_{2\text{'s}} + 1$ is the number 8_{10} in the decimal system, while $\langle 1010 \rangle_{2\text{'s}} + 0$ is -6_{10}.

Example 2. The DLSB number $\langle 0111 \rangle_{2\text{'s}} + 1 = 8_{10}$ is complemented by inverting all its bits: $\langle 1000 \rangle_{2\text{'s}} + 0 = -8_{10}$.

The addition of two n-bit DLSB numbers is performed using a conventional n-bit adder [231]. The extra LSB of one of the operands is given to the carry input of the adder, while the extra LSB of the other is attached to the sum as its extra LSB. The operation of the addition is illustrated in Figure 3.1. The subtraction is performed in the same way, however, the subtrahend is firstly complemented, namely all its bits (including the extra LSB) are logically inverted.

The multiplication of a DLSB number by a power of two, i.e., 2^e, is performed like in the conventional arithmetic, i.e., via e left shifts. Regarding the multiplication of two DLSB numbers, well-established architectures and algorithms can be employed with some extra overheads, as discussed in [231]. This arithmetic operation is exhaustively examined in the rest of the chapter.
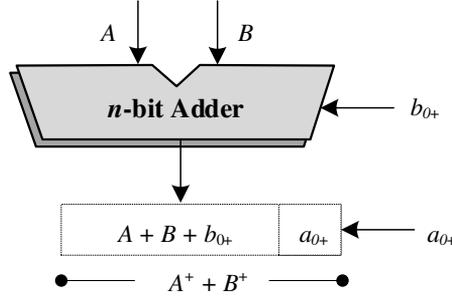
Figure 3.1: The addition of two n-bit DLSB numbers is performed using a conventional adder. The subtraction is performed with the same architecture, but with the inverted bits of B^+.

3.3. Design of DLSB Multiplication Circuits

In this section, we design the DLSB multiplier in two flavours, i.e., based on the straightforward approach and a more sophisticated improved approach. As baseline multiplication algorithm, we adopt the Modified Booth (MB) method [227], which facilitates low-level optimizations and also outperforms several well-established multiplication methods [226].

3.3.1. Straightforward DLSB Design

Let $A^+ = \langle a_{n-1}a_{n-2}\cdots a_0\rangle_{\text{2's}} + a_{0+}$ and $B^+ = \langle b_{n-1}b_{n-2}\cdots b_0\rangle_{\text{2's}} + b_{0+}$ be two n-bit 2's-complement DLSB numbers. We form their product as shown in Eq. (3.2).

$$A^+ \times B^+ = \left(\langle a_{n-1}a_{n-2}\cdots a_0\rangle_{\text{2's}} + a_{0+}\right) \cdot B^+ \tag{3.2}$$

Following the Modified Booth multiplication algorithm, B^+ is encoded with its extra LSB b_{0+} included in the least significant Modified Booth digit, i.e., $b_{-1} = b_{0+}$ instead of $b_{-1} = 0$ (conventional encoding), as shown in Eq. (3.3)–(3.4).

$$B^+ = -2^{n-1}b_{n-1} + \sum_{i=0}^{n-2}2^i b_i + b_{0+} = \sum_{j=0}^{n/2-1}4^j b_j^{MB} \tag{3.3}$$

$$\text{where } \quad b_j^{MB} = -2b_{2j+1} + b_{2j} + b_{2j-1} \quad \implies \quad b_j^{MB} \in \{0, \pm1, \pm2\} \tag{3.4}$$

The Modified Booth digits $b_j^{MB} \in \{0, \pm1, \pm2\}$ are functions of three consecutive bits of B^+ ($b_{2j+1}, b_{2j}, b_{2j-1}$), and they are formed according Table 3.1. Each digit is

Table 3.1: Modified Booth encoding.

Input			MB Digit	Output		
b_{2j+1}	b_{2j}	b_{2j-1}	b_j^{MB}	s_j	two_j	one_j
0	0	0	0	0	0	0
0	0	1	1	0	0	1
0	1	0	1	0	0	1
0	1	1	2	0	1	0
1	0	0	−2	1	1	0
1	0	1	−1	1	0	1
1	1	0	−1	1	0	1
1	1	1	0	1	0	0

represented by three 1-bit signals that define its sign (s_j) and absolute value (two_j, one_j). Using these encoding signals, the Modified Booth digits are calculated by Eq. (3.5).

$$b_j^{MB} = (-1)^{s_j} \cdot (2two_j + one_j), \;\; j = 0, 1, \cdots n/2 - 1 \qquad (3.5)$$

Using Eq. (3.2) and (3.3), the multiplication of the DLSB numbers A^+ and B^+ is formed as shown in Eq. (3.6).

$$A^+ \times B^+ = \langle a_{n-1}a_{n-2}\cdots a_0\rangle_{2\text{'s}} \cdot B^+ + a_{0+} \cdot B^+ =$$

$$= \langle a_{n-1}a_{n-2}\cdots a_0\rangle_{2\text{'s}} \cdot \sum_{j=0}^{n/2-1} 4^j b_j^{MB} + a_{0+} \cdot \left(\langle b_{n-1}b_{n-2}\cdots b_0\rangle_{2\text{'s}} + b_{0+} \right) \quad (3.6)$$

The implementation of the DLSB multiplier based on Eq. (3.6) requires a conventional Modified Booth multiplier to calculate the first term of the final expression, as well as considerable overhead to calculate the extra term $a_{0+} \cdot \left(\langle b_{n-1}b_{n-2}\cdots b_0\rangle_{2\text{'s}} + b_{0+} \right)$.

3.3.2. Sophisticated DLSB Design

To reduce the overhead derived by the calculation of the extra term in the straightforward approach, we consider an alternative representation for the operand A^+. Firstly, we examine how A^+ is formed with respect to the value of a_{0+}:

- if $a_{0+} = 0$, we get:

$$A^+ = \langle a_{n-1}a_{n-2}\cdots a_0 \rangle_{2\text{'s}} + 0 = \langle a_{n-1}a_{n-2}\cdots a_0 \rangle_{2\text{'s}} \qquad (3.7)$$

- if $a_{0+} = 1$, and using the expression $a_i = -\bar{a}_i + 1$ ($\bar{a}_i$: inverted a_i), we get:

$$
\begin{aligned}
A^+ &= \langle a_{n-1}a_{n-2}\cdots a_0 \rangle_{2\text{'s}} + 1 = -2^{n-1}a_{n-1} + \sum_{i=0}^{n-2} 2^i a_i + 1 = \\
&= -2^{n-1}(-\bar{a}_{n-1} + 1) + \sum_{i=0}^{n-2} 2^i(-\bar{a}_i + 1) + 1 = \\
&= -(-2^{n-1}\bar{a}_{n-1} + 2^{n-1}) - \sum_{i=0}^{n-2} 2^i\bar{a}_i + \sum_{i=0}^{n-2} 2^i + 1 = \\
&= -(-2^{n-1}\bar{a}_{n-1} + \sum_{i=0}^{n-2} 2^i\bar{a}_i) - 2^{n-1} + \sum_{i=0}^{n-2} 2^i + 1 = \\
&= -(-2^{n-1}\bar{a}_{n-1} + \sum_{i=0}^{n-2} 2^i\bar{a}_i) = -\langle \bar{a}_{n-1}\bar{a}_{n-2}\cdots\bar{a}_0 \rangle_{2\text{'s}} \qquad (3.8)
\end{aligned}
$$

Next, we encode A^+ using the expression $a'_i = a_i \oplus a_{0+}$. Namely, each bit of $A = \langle a_{n-1}a_{n-2}\cdots a_0 \rangle_{2\text{'s}}$ is driven to a XOR gate along with the extra LSB a_{0+} to form $A' = \langle a'_{n-1}a'_{n-2}\cdots a'_0 \rangle_{2\text{'s}}$. Hence, A^+ is encoded as shown in Eq. (3.9).

$$A^+ = (-1)^{a_{0+}} \cdot (-2^{n-1}a'_{n-1} + \sum_{i=0}^{n-2} 2^i a'_i) = (-1)^{a_{0+}} \cdot A', \qquad a'_i = a_i \oplus a_{0+} \qquad (3.9)$$

This encoding is equivalent to the initial representation of A^+, as verified by assigning either 0 or 1 to a_{0+} in Eq. (3.9):

- for $a_{0+} = 0$, A^+ is equal to the expression of Eq. (3.7).

- for $a_{0+} = 1$, A^+ is equal to the expression of Eq. (3.8).

To perform the DLSB multiplication, we use Eq. (3.3) for B^+ and Eq. (3.9) for A^+. Their product is defined as shown in Eq. (3.10).

$$A^+ \times B^+ = (-1)^{a_{0+}} \cdot A' \cdot \sum_{j=0}^{n/2-1} 4^j b_j^{MB} = A' \cdot \sum_{j=0}^{n/2-1} (-1)^{a_{0+}} \cdot 4^j b_j^{MB} \qquad (3.10)$$

The next step is to integrate the term $(-1)^{a_{0+}}$ in the calculation of b_j^{MB}, i.e., the Modified Booth digits, which are calculated by Eq. (3.5). Considering that this term

affects the sign (depending on the value of a_{0+}), we drive s_j, i.e., the sign of the Modified Booth digits, to a XOR gate along with a_{0+}. As a result, the absolute value of the new Modified Booth digits, labeled as b_j^{MB+}, is the same with that of b_j^{MB}, however, their sign depends on s_j and a_{0+}. In particular, the new expression for the sign is given by Eq. (3.11).

$$s_j' = s_j \oplus a_{0+} \tag{3.11}$$

Based on the above analysis, Eq. (3.10) is transformed to Eq. (3.12), and the calculation of the new Modified Booth digits is performed as shown in Eq. (3.13).

$$A^+ \times B^+ = A' \cdot \sum_{j=0}^{n/2-1} 4^j b_j^{MB+} \tag{3.12}$$

$$\text{where} \quad b_j^{MB+} = (-1)^{s_j'} \cdot (2two_j + one_j) \tag{3.13}$$

The logic equations of the encoding signals s_j, one_j, and two_j are derived from Table 3.1. The proposed encoding circuit is illustrated in Figure 3.2a. Compared to the conventional Modified Booth encoder that is illustrated in Figure 3.2b, the proposed encoder has an extra XOR-2 gate for calculating s_j', which includes the extra LSB of A^+.

For the calculation of the product defined in (3.12), the generation of $n/2$ partial products PP_j is required, as shown in Eq. (3.14) and (3.15).

$$A^+ \times B^+ = A' \cdot \sum_{j=0}^{n/2-1} 4^j b_j^{MB+} = \sum_{j=0}^{n/2-1} 4^j PP_j \tag{3.14}$$

$$\text{where} \quad PP_j = A' \cdot b_j^{MB+} = 2^n \overline{p}_{j,n} + \sum_{i=0}^{n-1} 2^i p_{j,i}, \quad j = 0, 1, \cdots n/2 - 1 \tag{3.15}$$

The generation of the i-th bit of the partial product PP_j is illustrated at gate level in Figure 3.2c. The circuit of the DLSB partial product generator is the same with that employed in the conventional Modified Booth algorithm. To calculate the Most Significant Bit (MSB) of each partial product, we consider $a_n' = a_{n-1}'$, and thus, $a_n = a_{n-1}$. Their LSB is calculated considering $a_{-1}' = 0$, and thus, $a_{-1} = a_{0+}$. In comparison with the conventional Modified Booth multiplier, the generation of the partial product bits is intact, except for the LSBs, where $s_j' = s_j \oplus a_{0+}$ is used instead of s_j.
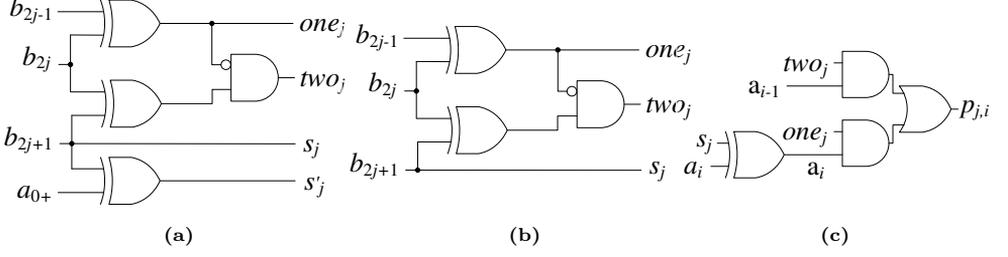
Figure 3.2: Circuits used in the DLSB multiplication: **(a)** DLSB Modified Booth encoder (used in the sophisticated approach), **(b)** conventional Modified Booth encoder (used in the straightforward approach), **(c)** conventional partial product generator (a_i: i-bit of A, $\mathrm{a}_i = s_j \oplus a_i$, used in both approaches).

The generated partial products are accumulated, properly weighted, using a Wallace tree [238] along with the constant terms ('1's) and the correction terms (c'_j). Finally, the carry-save output of the Wallace tree is driven to a fast prefix adder [238] to form the final result. Compared to the conventional Modified Booth multiplier, again the only difference is the use of s'_j instead of s_j in the calculation of the correction terms, as shown in its logic function in Eq. (3.16).

$$c'_j = s'_j \cdot (one_j + two_j) = (s_j \oplus a_{0+}) \cdot (one_j + two_j) \tag{3.16}$$

Overall, the proposed DLSB multiplier incorporates the typical stages of the Modified Booth multiplication circuit: (i) encoding, (ii) partial product generation, (iii) partial product accumulation, and (iv) final addition. As shown, with careful design and bit-level manipulations we reduce the circuit overheads of the straightforward approach: the product $a_{0+} \cdot B^+$ of Eq. (3.6) is eliminated and the only overhead regards the encoding stage, where an extra signal (s'_j) is generated.

For comparison purposes, Figure 3.3 illustrates the partial product matrices of the 16-bit examined multipliers, i.e., the conventional Modified Booth multiplier (Figure 3.3a), the straightforward DLSB Modified Booth multiplier (Figure 3.3b), and the sophisticated DLSB Modified Booth multiplier (Figure 3.3c). All the necessary bits (partial product bits, constant terms, and correction terms) that are required for the partial product accumulation are included. As shown, the straightforward DLSB multiplier implements the conventional multiplier plus extra logic for the product $a_{0+} \cdot B^+$. In contrast, the sophisticated design does not implement this logic, it has the same depth of accumulation tree with the conventional multiplier, and its only difference compared to the latter is the calculation of s'_j. This signal is used instead of s_j for generating the LSB of the partial products ($p'_{j,0}$) and the correction terms (c'_j).
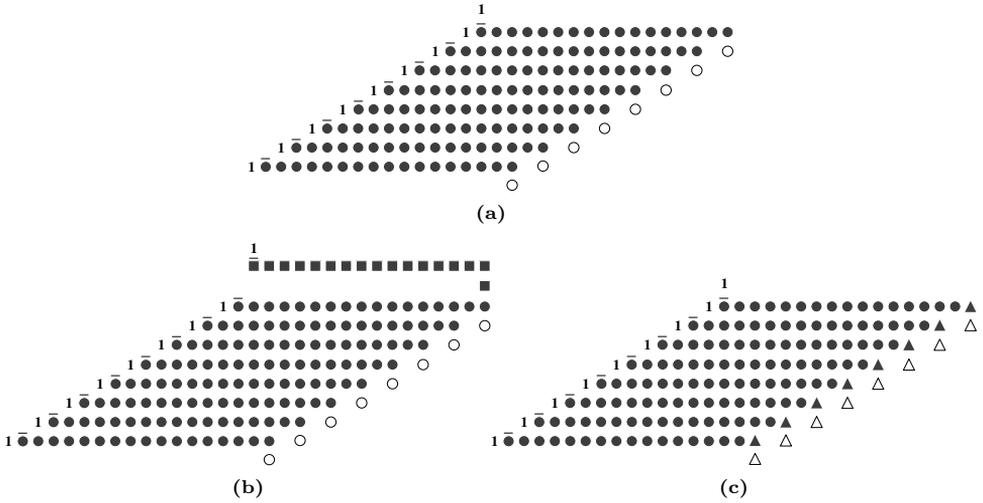
Figure 3.3: Partial product matrices of 16-bit multipliers: **(a)** conventional Modified Booth multiplier (ordinary 2's-complement numbers), **(b)** straightforward DLSB Modified Booth multiplier of Eq. (3.6), **(c)** sophisticated DLSB Modified Booth multiplier of Eq. (3.14).
Symbols: ●: $p_{j,i}$ ○: c_j ■: $a_{0+} \cdot b_i$ ▲: $p'_{j,0}$ △: c'_j

3.4. Evaluation

In this section, firstly, we evaluate the DLSB multipliers, and then, we demonstrate how they can be used to improve the hardware efficiency of large-size multiplications. We conduct both theoretical and experimental analysis. We consider the following notations for the examined multipliers:

- CMB: conventional Modified Booth multiplier for ordinary 2's-complement numbers (Figure 3.3a).

- DLSB1: straightforward Modified Booth multiplier for 2's-complement DLSB numbers (Figure 3.3b).

- DLSB2: sophisticated Modified Booth multiplier for 2's-complement DLSB numbers (Figure 3.3c).

3.4.1. Theoretical Analysis

As already discussed in the previous section, DLSB1 implements CMB and requires additional logic ($n + 1$ AND-2 gates) for calculating the extra product, as well as

considerable overhead for accumulating it with the rest partial products. On the other hand, DLSB2 implements CMB with a small overhead of $n/2$ XOR-2 gates for calculating s'_j. Namely, it uses the encoder of Figure 3.2a instead of that of Figure 3.2b. Subsequently, we evaluate the area complexity of the DLSB multipliers based on the unit gate model used in [239]. According to this model, the primitive gates AND-2 and OR-2 are equal to 1 unit gate, the NOT gate counts as 0.5 unit gates, and the XOR-2 gate is equal to 2 unit gates. Moreover, the area of a full adder and a half adder is 7 and 3 unit gates, respectively. The gate equivalence of the main components used in the partial product generation and accumulation of the multipliers is summarized in Table 3.2. In the next paragraphs, we examine separately the stages of the partial product generation and accumulation, and then we quantify the resources of the entire multiplication architectures.

Regarding the partial product generation, the n-bit CMB multiplier generates $n/2$ $(n+1)$-bit partial products, Therefore, it requires $n/2$ MB encoders of Figure 3.2b, $n/2$ $\times (n + 1)$ 1-bit partial product generators of Figure 3.2c, and $n/2$ correction terms generators (signals c_j). Additionally, $n/2$ NOT gates are needed for inverting the MSB of each partial product. DLSB1 requires the same resources, and also employs $n + 1$ AND-2 and 1 NOT gates for calculating the product $a_{0+} \cdot B^+$. Compared to CMB, DLSB2 uses only a different encoder, i.e., the DLSB encoder of Figure 3.2a.

Next, we analyze the resources needed for the partial product accumulation. CMB and DLSB2 accumulate $n/2 + 1$ n-bit vectors, i.e., $n/2$ partial products and 1 vector containing the constant and correction terms. This multi-operand addition is performed in carry-save form [238], thus, $(n/2 - 1) \times n$ full adders are required in total. In DLSB1, one more number must be added, and thus, $n/2 \times n$ full adders are required. Finally, in all multipliers, the $2n$-bit carry-save output of the Wallace tree is driven to a fast adder [238], which consists of $2n$ half adders, $n \log_2 2n$ propagate group circuits (each one is 3 unit gates), and $2n$ XOR-2 gates.

Based on our theoretical analysis about the resources of each design, Table 3.3 reports the overhead in unit gates of the DLSB multipliers for different bit-widths, i.e., $n = 8, 16, 32$. This overhead is calculated by comparing the unit gates of the DLSB multipliers with those of CMB. As expected, DLSB1, which is based on the straightforward design approach, exhibits larger overhead compared to DLSB2. On the other hand, the overhead of DLSB2 is negligible, lying in the range 0.5%–1.4%. The unit gate model is a simplified model, but it gives a rough estimation about the circuit area. In the next section, we provide experimental analysis based on industrial-strength synthesis of the circuits, in which it is shown that theoretical outcomes and trends follow in a fine manner the experimental results.

Table 3.2: Unit gates per component and components of the n-bit multipliers.

Component	Unit Gates	# in CMB	# in DLSB1	# in DLSB2
MB Encoder	5.5	$n/2$	$n/2$	$-$
DLSB MB Encoder	7.5	$-$	$-$	$n/2$
MB PP Generator	5	$n/2 \times (n+1)$	$n/2 \times (n+1)$	$n/2 \times (n+1)$
AND PP Generator	1	$-$	$n+1$	$-$
Corr. Term Generator	2	$n/2$	$n/2$	$n/2$
Full Adder	7	$(n/2-1) \times n$	$n/2 \times n$	$(n/2-1) \times n$
Half Adder	3	$2n$	$2n$	$2n$
Propagate Group	3	$n \log_2 2n$	$n \log_2 2n$	$n \log_2 2n$
XOR-2 Gate	2	$2n$	$2n$	$2n$

Table 3.3: Unit gate overhead of the n-bit DLSB multipliers.

Multiplier	$n = 8$	$n = 16$	$n = 32$
DLSB1	11.8%	6.7%	3.7%
DLSB2	1.4%	0.8%	0.5%

3.4.2. Experimental Results

This section evaluates the multiplication circuits in terms of delay, area, and energy consumption, using industrial-strength tools. All the multipliers are implemented in Verilog and synthesized with the Synopsys Design Compiler and the TSMC 45-nm standard-cell library. The simulations are performed with Mentor Graphics QuestaSim. Both synthesis and simulation are performed at 1V, i.e., the nominal supply voltage. Moreover, the designs are configured to operate at their critical path delay, i.e., at maximum frequency. The critical path delay and the area are reported by Synopsys Design Compiler, while the power consumption is measured with Synopsys PrimeTime. The energy consumption is also calculated by the product of delay and power.

Table 3.4 presents the experimental results from the synthesis of the multipliers for various sizes, i.e., $n = 8, 16, 32$. Furthermore, the energy and area overheads compared to CMB are reported. In terms of critical paths, as expected, the DLSB multipliers deliver almost the same delays with CMB. DLSB2 exhibits similar delay even for the smallest bit-width, i.e., 0.38ns versus 0.37ns. This is justified by the fact that the depth of DLSB2's partial product tree has not increased. In comparison with DLSB1, the sophisticated DLSB2 design achieves higher efficiency for all the resources. On average, DLSB2 imposes small area and energy overheads (i.e., 3.1% and 3.3%, respectively), while the corresponding overheads for DLSB1 are 11.9% and

Table 3.4: Synthesis results of the *n*-bit DLSB multipliers on TSMC 45-nm standard-cell.

Design		Delay (ns)	Area (μm^2)	Area $(\%)^1$	Energy ($\mu W \cdot ns$)	Energy $(\%)^1$
$n = 8$	CMB	0.37	485	−	384	−
	DLSB1	0.40	572	+18	480	+25
	DLSB2	0.38	510	+5.2	406	+5.7
$n = 16$	CMB	0.51	1519	−	1186	−
	DLSB1	0.52	1701	+12	1344	+13.3
	DLSB2	0.51	1562	+2.8	1217	+2.6
$n = 32$	CMB	0.67	5189	−	4340	−
	DLSB1	0.67	5491	+5.8	4685	+7.9
	DLSB2	0.67	5256	+1.3	4412	+1.7

[1] Refers to % area/energy overhead (relative increase) in comparison with CMB.

15.4%. Regarding the bit-width scaling, the derived results show that as the multiplier's size increases, the impact of the extra LSB is becoming smaller in both area and energy. This observation is normal, as by increasing the bit-width, the rate of the overheads induced by the DLSB increases slower than the rate of the area added due to the bit-width scaling.

3.4.3. Case Study: DLSB for Large-Size Multiplication

In this section, we demonstrate how the sophisticated DLSB multiplier can be effectively used as building block for implementing multiplications with large input operands. For the rest of the analysis, we consider 2's-complement arithmetic and small fixed-size multiplication circuits, i.e., there is the constraint of hardwired arithmetic units with specific small bit-width, like in embedded devices [237].

We follow the well-established approach of decomposing the large bit-width operands to smaller sizes [237]. The reduction of the operand bit-width to match the desired bit-width (imposed by the constraint) is a recursive procedure. Nevertheless, to ease the description and without loss of generality, we focus our analysis on the first recursion, assuming that the bit-width constraint exposed by the available arithmetic unit is satisfied. Next, we discuss how the large-size multiplications are partitioned in the conventional arithmetic, and then, we present a more efficient DLSB-based partition.

Conventional 2's-Complement Arithmetic: Let A and B be two k-bit 2's-complement numbers to multiply. Without loss of generality, we consider $k = 2n$,

where n is an integer, and we split A and B in two n-bit words, as shown in Eq. (3.17). The product $A \times B$ is formed as shown in Eq. (3.18).

$$A = \langle a_{2n-1}a_{2n-2}\cdots a_n\rangle_{2\text{'s}} \cdot 2^n + \langle a_{n-1}a_{n-2}\cdots a_0\rangle_{\text{u}} = A_s \cdot 2^n + A_u$$

$$B = \langle b_{2n-1}b_{2n-2}\cdots b_n\rangle_{2\text{'s}} \cdot 2^n + \langle b_{n-1}b_{n-2}\cdots b_0\rangle_{\text{u}} = B_s \cdot 2^n + B_u \qquad (3.17)$$

$$A \times B = (A_s \cdot 2^n + A_u) \cdot (B_s \cdot 2^n + B_u) =$$
$$= A_s \cdot B_s \cdot 2^{2n} + (A_s \cdot B_u + A_u \cdot B_s) \cdot 2^n + A_u \cdot B_u \qquad (3.18)$$

The above product requires the execution of 4 multiplications: (i) 2's-complement $\times$ 2's-complement ($A_s \cdot B_s$), (ii) unsigned $\times$ unsigned ($A_u \cdot B_u$), (iii) unsigned $\times$ 2's-complement ($A_u \cdot B_s$), (iv) 2's-complement $\times$ unsigned ($A_s \cdot B_u$). We note that the multiplications involving unsigned numbers must employ an extra bit for the sign extension. As a result, to cover all the bit-width cases and calculate the 4 sub-products with the same module, the $(n + 1)$-bit CMB multiplier needs to be used.

2's-Complement DLSB Arithmetic: We propose an alternative operand partition that aims to create DLSB-like bit vectors. As shown in Eq. (3.19)–(3.20), we divide the operands in two 2's-complement segments, and attach the MSB of their least significant part, i.e., a_{n-1} and b_{n-1}, as an extra LSB in their most significant part. Assuming an extra LSB equal to 0 for the least significant parts, the k-bit operands are now formed as functions of two n-bit 2's-complement DLSB numbers.

$$A = \big(\langle a_{2n-1}a_{2n-2}\cdots a_n\rangle_{2\text{'s}} + a_{n-1}\big) \cdot 2^n + \big(\langle a_{n-1}a_{n-2}\cdots a_0\rangle_{2\text{'s}} + 0\big) =$$
$$= A_1^+ \cdot 2^n + A_2^+ \qquad (3.19)$$

$$B = \big(\langle b_{2n-1}b_{2n-2}\cdots b_n\rangle_{2\text{'s}} + b_{n-1}\big) \cdot 2^n + \big(\langle b_{n-1}b_{n-2}\cdots b_0\rangle_{2\text{'s}} + 0\big) =$$
$$= B_1^+ \cdot 2^n + B_2^+ \qquad (3.20)$$

Considering the proposed DLSB-based partition, the 4 sub-products, derived by applying again the distributive property, can be calculated by the n-bit 2's-complement DLSB multiplier, which can effectively replace the $(n+1)$-bit CMB as building block. To evaluate the efficiency of the proposed scheme, we compare the n-bit DLSB2 with the $(n + 1)$-bit CMB for various bit-widths. The experimental setup is the same as described in the previous section. Figure 3.4 illustrates the energy and area gains achieved by the proposed DLSB2 compared to CMB. In particular, DLSB2 provides significant gains, especially for small bit-width (i.e., 30.6% and 43.3% area
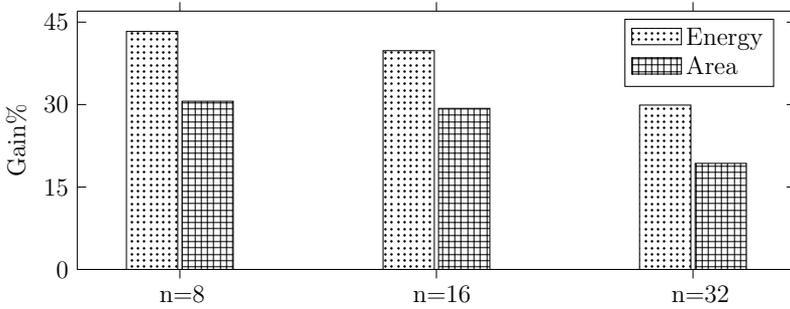
Figure 3.4: Energy and area gains provided by *n*-bit DLSB2 compared to $(n+1)$-bit CMB (targeting the use of DLSB2 instead of CMB as building block in large-size multiplications).

and energy gains, respectively, for $n = 8$). The inefficiency of CMB is justified by the extra bit in the multiplier's bit-width, which results in the generation of one more partial product and the addition of an extra bit in each partial product.

3.5. Conclusion

The goal of this chapter was to highlight the benefits of using novel numerical formats and applying low-level arithmetic optimizations and sophisticated bit-level manipulations. In particular, we adopted the DLSB arithmetic, which assumes an extra LSB in the number representation and provides advantageous features, such as the representation symmetry and the simplification of rounding in floating-point operations. Targeting to decrease the overheads induced by the extra LSBs, we introduced a new and effective algorithm for multiplying two 2's-complement DLSB numbers. Through an alternative representation, we optimized the Modified Booth multiplication scheme to decrease the overheads of the straightforward DLSB multiplication. According to our experimental evaluation for different multiplication bit-widths, our DLSB design provides average overheads of ~3% in area and energy. In contrast, the DLSB design that does not adopt our sophisticated low-level optimizations, suffers from average overheads of ~12% and ~15% in area and energy, respectively. Finally, we demonstrated how the proposed DLSB multiplier can be efficiently used as building block in large-size multiplications, replacing the conventional multiplier. In this case, both the area and energy consumption are significantly improved, considering that the DLSB multiplier outperforms the conventional design that is needed to implement the sub-products, by up to 30.6% and 43.3% in area and energy, respectively.

Chapter 4

Arithmetic Approximation: Hybrid High-Radix Encoding

Approximate Computing forms a design alternative that exploits the intrinsic error resilience of various applications and produces energy-efficient circuits with small accuracy loss. In this chapter, we examine the impact of applying low-level optimizations and disciplined approximations in the design of arithmetic circuits. Towards this direction, we propose an approximate hybrid high-radix encoding for generating the energy-efficient RAD multipliers. The proposed encoding scheme approximately encodes one of the operands, using the accurate radix-4 encoding for its most significant part and an approximate higher radix encoding for its least significant part. In the high-radix encoding, the approximations are applied by mapping all the high-radix values to a set of values that includes only the 4 largest powers of two. The proposed hybrid encoding is configurable and can be tuned to achieve the desired energy–accuracy trade-off. Another important feature of the proposed design is that the error induced by the approximations is bounded by a Gaussian distribution with near-zero average. In addition, the mean relative error of the multiplication depends only on the approximately encoded operand, and thus, the calculation of the corresponding error metrics is accelerated, eliminating the need for exhaustive hardware simulations. In terms of resources, the proposed multipliers deliver up to 56 % energy and 55 % area gains compared to the accurate radix-4 multiplier, when operating at the same frequency. Moreover, our designs outperform state-of-the-art multipliers by up to 40 % in energy consumption for similar error values. Finally, we examine the scalability of our technique, showing that as the multiplier's size increases, our designs achieve larger gains in energy consumption and critical path delay, i.e., up to 64 % and 22 %, respectively, while the error remains constant.
*This chapter is based on our **publication** in [153].*

4.1. Introduction

In modern embedded systems and data centers, power efficiency and performance are critical design concerns. Considering that various application domains exhibit an intrinsic error resilience (e.g., digital signal processing, data analytics, and data mining [29, 240]), Approximate Computing [16, 17, 19] appears as an effective solution to provide remarkable power gains and/or speed improvements. In Approximate Computing, error is viewed as a commodity that can be traded for significant gains in resources (e.g., area, power, energy, latency, or throughput) [241], and thus, it composes a promising design paradigm to generate power-efficient systems and circuits. In particular, Approximate Computing exploits the innate error tolerance of the applications and deliberately relaxes the correctness of some computations to decrease their power consumption and/or accelerate their execution.

To take advantage of the benefits provided by Approximate Computing, massive research has been conducted in the field of approximate hardware and circuits. In this field, automated synthesis techniques [242] and hardware description language annotations [183] have been proposed to facilitate the design of approximate hardware. Significant research has also been reported in approximate processors that employ quality programmable vectors [243], neural networks [244] and approximate custom instructions [245]. Circuit-level approximations involve voltage over-scaling [202, 224] and over-clocking [213, 225] techniques. Voltage over-scaling lowers the supply voltage of the circuit below the nominal value, and thus, the power consumption is decreased in exchange for erroneous outputs due to the critical paths' failure to meet the timing constraints [224]. Over-clocking inserts timing violations, reducing expensive timing guard-bands and providing performance improvement [225]. Another approach is to apply logic-level approximations, i.e., modify the truth table, employ inexact components, or prune nodes of the circuit.

The main target of logic-level Approximate Computing is the arithmetic circuits [216], which are key computation units in general-purpose processors and custom hardware accelerators. Extensive research is reported in approximate adders [130, 246–249], which provide significant gains in critical path delays and power consumption. On the other hand, research activities on the approximate multipliers [142, 143, 149, 151, 154, 250–256] are less comprehensive compared to the respective on approximate adders. In the multiplication circuit, approximations can be applied in the partial product generation [149, 151, 255] and the partial product accumulation [154, 250, 251, 253]. Both types of approximation are synergistic and can be applied in collaboration to achieve higher power reduction [149, 151]. Although significant research regards the partial product accumulation, approximation techniques in the partial product generation have received less attention. Another limitation of the existing approximate

multipliers is that the majority of them (e.g., [154, 250, 252, 256]) does not examine signed multiplication.

In this chapter, motivated by the resource gains provided by arithmetic circuits, we present a novel *hybrid high-radix encoding* for approximate multipliers. Our technique addresses some of the issues of prior designs, such as the signed multiplication and the reduction of the critical paths, and it also provides better results than state-of-the-art works. The proposed encoding is hybrid, i.e., it splits the number and encodes it with two different schemes (accurate and approximate). More specifically, the Most Significant Bits (MSBs) are encoded using the accurate radix-4 encoding, whereas the k Least Significant Bits (LSBs) are encoded using the approximate high-radix-2^k (where $k \geq 4$). To simplify the increased complexity induced by the conventional high-radix encodings, we alter their truth tables and generate their approximate variants. Using this approximate hybrid encoding, the number of the partial products is decreased, and simpler tree architectures are used for the partial product accumulation.

The **contribution** of this chapter is summarized as follows:

(i) We highlight the efficiency of applying disciplined low-level approximations, which deliver valuable resource gains while keeping the accuracy in acceptable levels.

(ii) We address the circuit overheads of the classic high-radix encodings, e.g., radix-64, radix-256, radix-1024, which constitute them inefficient for use in multiplication circuits.

(iii) We propose a new hybrid encoding for approximate partial product generation, which is parametric in terms of approximation degree and can be combined with other techniques, e.g., with approximate partial product accumulation.

(iv) We show that the proposed technique outperforms related state-of-the-art approximation techniques, providing remarkable area and energy gains for comparable error values.

The remainder of this chapter is organized as follows. Section 4.2 introduces the proposed approximate hybrid high-radix encoding and its application in the design of approximate radix-based multipliers. Section 4.3 performs the evaluation of the proposed design, including theoretical resource analysis, error analysis, and comparative experimental results. Finally, Section 4.4 draws the conclusions.

4.2. Design of Approximate High-Radix Encodings and Multipliers

Conventional high-radix operand encodings reduce the total number of partial products in multipliers, and thus, the partial product accumulation is performed with simpler adder tree architectures. However, the high-radix encoders, as well as the high-radix-based partial product generators, are characterized by increased logic complexity, negating thus the benefits of the partial product reduction. Therefore, the prevailing radix encoding for multiplication circuits is radix-4, outperforming its high-radix counterparts.

In this section, we propose an approximate hybrid encoding, which applies simultaneously the conventional (accurate) radix-4 encoding and a new (approximate) high-radix-2^k encoding. In more detail, we present the functionality of the hybrid encoding, we introduce the proposed approximations that simplify the complexity of the conventional high-radix encoding, and finally, we design inexact multipliers based on the proposed approximate encoding.

4.2.1. Approximate Operand Encoding

Let B be a n-bit 2's-complement number and k be an even number belonging in the interval $[4, n - 2]$, i.e., $k = 2m$: $m \in \mathbb{Z}$ and $2 \leq m \leq (n - 2)/2$. B is divided into two segments with respect to k: the most significant part containing its $n - k + 1$ MSBs and the least significant part containing its k LSBs (there is a shared bit between the two segments). The $n - k + 1$ MSBs are encoded with the radix-4 encoding, while the k LSBs are encoded with the radix-2^k encoding, as shown in Eq. (4.1)–(4.3).

$$B = \langle b_{n-1}b_{n-2}\cdots b_0\rangle_{2's} = -2^{n-1}b_{n-1} + \sum_{i=0}^{n-2}2^i b_i = \sum_{j=k/2}^{n/2-1}4^j y_j^{R4} + y_0^{R2^k} \tag{4.1}$$

$$y_j^{R4} = -2b_{2j+1} + b_{2j} + b_{2j-1} \implies y_j^{R4} \in \{0, \pm1, \pm2\} \tag{4.2}$$

$$y_0^{R2^k} = -2^{k-1}b_{k-1}+2^{k-2}b_{k-2}+\cdots+b_0 \implies y_0^{R2^k} \in \{0, \pm1, \ldots, \pm(2^{k-1}-1), -2^{k-1}\} \tag{4.3}$$

Eq. (4.2) refers to the radix-4 encoded digits, while Eq. (4.3) refers to the radix-2^k encoded digit. In more detail, the radix-4 encoding creates $(n - k)/2$ digits $y_j^{R4} \in \{0, \pm1, \pm2\}$, while the radix-$2^k$ encoding creates the digit $y_0^{R2^k} \in \{0, \pm1, \pm2,$

± 3, $\ldots$, $\pm(2^{k-1} - 1)$, $-2^{k-1}\}$. In total, B is encoded with $(n - k)/2 + 1$ digits.

The encoding circuit for the above accurate hybrid encoding features increased logic due to the large number of $y_0^{R2^k}$ values and the values that are not power of two. Therefore, we propose an alternative approximate variant of the high-radix encoding, while we keep the low-complexity radix-4 encoding of the MSBs to avoid huge errors. To approximate $y_0^{R2^k}$, we map all the values that are not power of two, as well as the $k - 4$ smallest powers of two, to their nearest of the 4 largest powers of two or 0. In this way, the approximate digit $\hat{y}_0^{R2^k}$ takes values from a smaller set that includes only 4 absolute values plus 0. In addition, the sum of these values is 0, like in the set of values of y_j^{R4}. We choose to keep only the 4 largest powers of two, so that the approximate radix-2^k encoder requires about double the logic of the $(n - k)/2$ accurate radix-4 encoders.

Table 4.1 presents the accurate radix-4 encoding. The signal $sign_j$ indicates the sign of y_j^{R4}, i.e., it is activated when the digit is negative. The signals $\times 1_j$ and $\times 2_j$ regard the magnitude of y_j^{R4}, i.e., they are activated when its magnitude is 1 and 2, respectively. The logic functions of these signals are provided in Eq. (4.4)–(4.6).

$$sign_j = b_{2j+1} \tag{4.4}$$

$$\times 1_j = b_{2j-1} \oplus b_{2j} \tag{4.5}$$

$$\times 2_j = (b_{2j+1} \oplus b_{2j}) \cdot \overline{(b_{2j-1} \oplus b_{2j})} \tag{4.6}$$

Table 4.2 presents the approximate radix-2^k encoding. The first two columns show the proposed mapping to create the approximate set of values ($y_0^{R2^k} \rightarrow \hat{y}_0^{R2^k}$). For example, if the value of $y_0^{R2^k}$, as originally calculated by Eq. (4.3), belongs in $[2^{k-5}, 2^{k-4} + 2^{k-5})$, it is mapped to 2^{k-4}. Similar to the radix-4 encoding, the encoding signals are activated to indicate the value of $\hat{y}_0^{R2^k}$. The logic functions of the approximate radix-2^k encoding signals are provided in Eq. (4.7)–(4.11). We note that for $k = 4$, we consider an extra bit $b_{-1} = 0$.

$$sign = b_{k-1} \tag{4.7}$$

$$\times 2^{k-4} = (\overline{b}_{k-2} \cdot \overline{b}_{k-3} \cdot \overline{b}_{k-4} + b_{k-2} \cdot b_{k-3} \cdot b_{k-4}) \cdot (b_{k-4} \oplus b_{k-5}) \tag{4.8}$$

$$\times 2^{k-3} = \overline{b}_{k-1} \cdot \overline{b}_{k-2} \cdot (\overline{b}_{k-3} \cdot b_{k-4} \cdot b_{k-5} + b_{k-3} \cdot \overline{b}_{k-4}) +$$
$$+ b_{k-1} \cdot b_{k-2} \cdot (b_{k-3} \cdot \overline{b}_{k-4} \cdot \overline{b}_{k-5} + \overline{b}_{k-3} \cdot b_{k-4}) \tag{4.9}$$

$$\times 2^{k-2} = \overline{b}_{k-2} \cdot b_{k-3} \cdot (b_{k-1} + b_{k-4}) + b_{k-2} \cdot \overline{b}_{k-3} \cdot (\overline{b}_{k-1} + \overline{b}_{k-4}) \tag{4.10}$$

$$\times 2^{k-1} = \overline{b}_{k-1} \cdot b_{k-2} \cdot b_{k-3} + b_{k-1} \cdot \overline{b}_{k-2} \cdot \overline{b}_{k-3} \tag{4.11}$$

Table 4.1: Accurate radix-4 encoding.

Operand Bits			R4 Digit	Encoding Signals		
b_{2j+1}	b_{2j}	b_{2j-1}	y_j^{R4}	$sign_j$	$\times 2_j$	$\times 1_j$
0	0	0	0	0	0	0
0	0	1	1	0	0	1
0	1	0	1	0	0	1
0	1	1	2	0	1	0
1	0	0	-2	1	1	0
1	0	1	-1	1	0	1
1	1	0	-1	1	0	1
1	1	1	0	1	0	0

Table 4.2: Approximate high-radix-2^k encoding.

Accurate Digit	Approx. Digit	Encoding Signals				
$y_0^{R2^k}$	$\hat{y}_0^{R2^k}$	$sign$	$\times 2^{k-1}$	$\times 2^{k-2}$	$\times 2^{k-3}$	$\times 2^{k-4}$
$[0, 2^{k-5})$	0	0	0	0	0	0
$[2^{k-5}, 2^{k-4} + 2^{k-5})$	2^{k-4}	0	0	0	0	1
$[2^{k-4} + 2^{k-5}, 2^{k-3} + 2^{k-4})$	2^{k-3}	0	0	0	1	0
$[2^{k-3} + 2^{k-4}, 2^{k-2} + 2^{k-3})$	2^{k-2}	0	0	1	0	0
$[2^{k-2} + 2^{k-3}, 2^{k-1})$	2^{k-1}	0	1	0	0	0
$[-2^{k-1}, -2^{k-2} - 2^{k-3})$	-2^{k-1}	1	1	0	0	0
$[-2^{k-2} - 2^{k-3}, -2^{k-3} - 2^{k-4})$	-2^{k-2}	1	0	1	0	0
$[-2^{k-3} - 2^{k-4}, -2^{k-4} - 2^{k-5})$	-2^{k-3}	1	0	0	1	0
$[-2^{k-4} - 2^{k-5}, -2^{k-5})$	-2^{k-4}	1	0	0	0	1
$[-2^{k-5}, 0)$	0	1	0	0	0	0

Overall, starting from the accurate hybrid encoding of Eq. (4.1)–(4.3), we approximate the high-radix encoding of the LSBs through the mapping of Table 4.2, and thus, B is approximately encoded to $\tilde{B}$ as shown in Eq. (4.12)–(4.14).

$$\tilde{B} = \sum_{j=k/2}^{n/2-1} 4^j y_j^{R4} + \hat{y}_0^{R2^k} \tag{4.12}$$

$$\text{where } y_j^{R4} \in \{0, \pm 1, \pm 2\} \tag{4.13}$$

$$\text{and } \hat{y}_0^{R2^k} \in \{0, \pm 2^{k-4}, \pm 2^{k-3}, \pm 2^{k-2}, \pm 2^{k-1}\} \tag{4.14}$$

Next, we present some examples to show how the proposed hybrid encoding is ap-

plied. We consider $n = 16$, i.e., 16-bit numbers and $k = 6, 10$, i.e., the LSBs are encoded with radix-64 and radix-1024, respectively. In the hybrid radix-64 encoding, the bits of B are grouped as shown in Eq. (4.15). Regarding the approximation of the radix-64 encoding, the 4 largest powers of two are ± 4, ± 8, ± 16 and ± 32. Thus, we map all the rest values of the original digit y_0^{R64} to their nearest value from the set $\{0, \pm 4, \pm 8, \pm 16, \pm 32\}$ according to Table 4.2 (for $k = 6$).

$$
\underbrace{b_{15}b_{14}\overbrace{b_{13}b_{12}b_{11}}^{y_6^{R4}}b_{10}\overbrace{b_9b_8b_7}^{y_4^{R4}}b_6\overbrace{b_5b_4b_3}^{y_0^{R64}}b_2b_1b_0}_{}
\tag{4.15}
$$

In the hybrid radix-1024 encoding, the bits of B are grouped as shown in Eq. (4.16). Similarly, the values of the digit y_0^{R1024} are mapped to $\{0, \pm 64, \pm 128, \pm 256, \pm 512\}$ according to Table 4.2 (for $k = 10$).

$$
\underbrace{b_{15}b_{14}\overbrace{b_{13}b_{12}b_{11}}^{y_6^{R4}}b_{10}\overbrace{b_9b_8b_7b_6b_5b_4b_3}^{y_0^{R1024}}b_2b_1b_0}_{}
\tag{4.16}
$$

For the implementation of the hybrid high-radix encoding, the designer needs only to use the logic functions of Eq. (4.4)–(4.6) to generate the $(n - k)/2$ accurate radix-4 encoders, as well as the logic functions of Eq. (4.7)–(4.11) to generate the single approximate radix-2^k encoder. We note that an important feature of this encoding is that the logic resources of the approximate radix-2^k encoder are fixed and independent of k.

4.2.2. RAD: Approximate High-Radix Multiplier

In this section, we present how the proposed hybrid encoding can be used to design approximate high-radix multipliers. We consider the notation RAD2^k for the multiplier that implements the hybrid high-radix-2^k encoding. Let $A \cdot B$ be the accurate multiplication of two n-bit 2's-complement numbers. We encode the multiplicand B with the proposed approximate high-radix encoding and perform the approximate multiplication $A \cdot \tilde{B}$.

The approximate architecture of the RAD2^k multiplier is illustrated in Figure 4.1. It consists of three stages: operand encoding, partial product generation and partial product accumulation. In the first stage, B is encoded as discussed in the previous section. The second stage inputs A and the encoding signals of $\tilde{B}$, and it generates $(n - k)/2$ accurate partial products based on the radix-4 encoding (digits y_j^{R4}) and
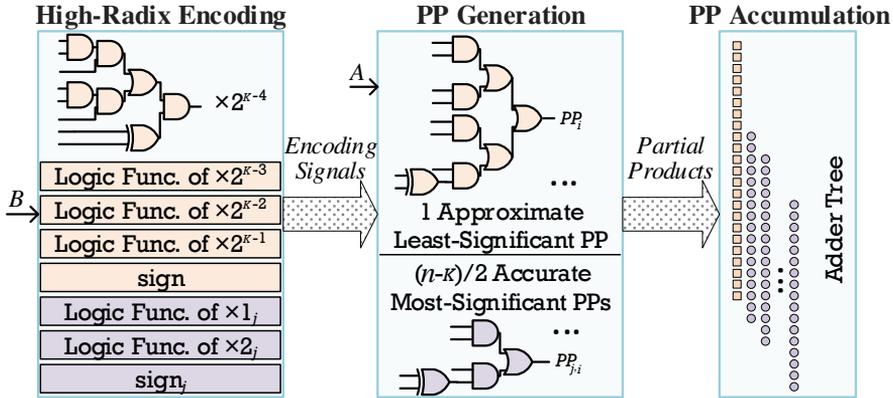
Figure 4.1: The architecture of the RAD2^k multiplier that is based on the approximate hybrid high-radix encoding.

Table 4.3: Partial products generated by the radix encodings ($k = 6, 8, 10$).

Encoding	Partial Products
Radix-4	$0, \pm A, \pm 2A$
Radix-64	$0, \pm 4A, \pm 8A, \pm 16A, \pm 32A$
Radix-256	$0, \pm 16A, \pm 32A, \pm 64A, \pm 128A$
Radix-1024	$0, \pm 64A, \pm 128A, \pm 256A, \pm 512A$

1 approximate partial product based on the radix-2^k encoding (digit $\hat{y}_0^{R2^k}$). The approximate partial product practically substitutes the $k/2$ least significant partial products of the accurate radix-4 multiplier. Namely, we achieve a reduction of $k/2-1$ partial products, as the accurate radix-4 multiplier generates $n/2$ partial products, while the proposed one generates $(n-k)/2 + 1$. Table 4.3 summarizes the partial products that can be generated by the accurate radix-4 encoding and the approximate radix-2^k encodings ($k = 6, 8, 10$). All the possible partial products express the multiplication of A with the encoded digits of B, which take the values discussed in the previous section. In the last stage, the partial products are accumulated to form the final approximate product. We note that the accumulation stage is orthogonal to the approximate encoding and partial product generation. Therefore, the designer can choose any method for the accumulation, either accurate or approximate.

The circuit of the i-bit partial product generator based on radix-4 encoding is the one discussed in Chapter 3 and illustrated in Figure 3.2c. It inputs the signals $sign_j$, $\times 1_j$, $\times 2_j$ and the bits of A, and it calculates the partial product bits. In Figure 4.2, we illustrate the i-bit partial product generators based on the approximate radix-64,
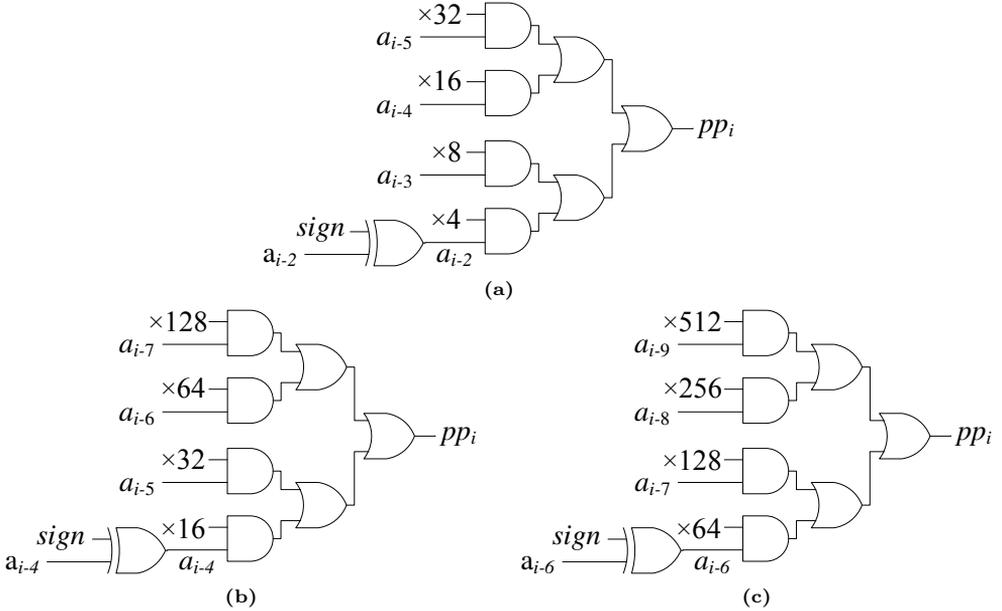
Figure 4.2: Approximate i-bit partial product generators based on the high-radix encodings: **(a)** radix-64 ($k = 6$), **(b)** radix-256 ($k = 8$), **(c)** radix-1024 ($k = 10$).
Notations: a_i: i-bit of A, $a_i = s_j \oplus a_i$

radix-256, and radix-1024 encodings. As shown, similar to the encoders, the area of the partial product generator for 1 bit is fixed and independent of the encoding (i.e., the value of k).

Figure 4.3 illustrates the partial product matrices of the 16-bit approximate multipliers for $k = 6, 8, 10$. The matrices also include constant and corrections terms, as those discussed in Section 3.3. As shown, each increment of the k parameter omits an accurate radix-4 partial product, which is encoded in the larger approximate radix-2^k partial product. The bit-width of each accurate partial product is $n + 1$, i.e., there are n generated partial product bits (black circles) plus the inverted MSB of the partial product (gray circle). Correspondingly, the bit-width of the approximate partial product is $n + k - 1$.

4.3. Evaluation

In this section, we evaluate our approximate high-radix multipliers. Firstly, we report a theoretical analysis on the resources of our approximate designs, then we examine

(a)

(b)   **(c)**
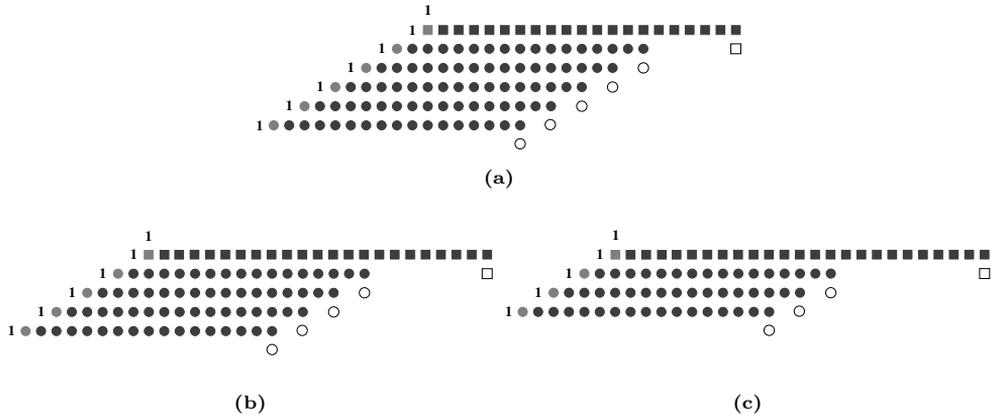
Figure 4.3: Partial product matrices of the 16-bit approximate multipliers using the accurate radix-4 and an approximate high-radix-2^k encoding: **(a)** radix-64 ($k = 6$), **(b)** radix-256 ($k = 8$), **(c)** radix-1024 ($k = 10$).

Symbols: ●: partial product bits from radix-4   ■: partial product bits from radix-2^k ●, ■: inverted partial product MSBs   ○, □: correction terms

the error introduced by the approximations, and finally, we attach experimental results including comparisons with state-of-the-art multipliers.

4.3.1. Theoretical Analysis

The advantage of the RAD2^k multipliers is their decreased logic compared to the accurate radix-4 multiplier (labeled as ACCR4), which results in faster operation and area/power gains. RAD2^k generates a larger least significant partial product, however, this product substitutes $k/2$ radix-4 partial products, and also, the corresponding circuits for its generation do not impose significant overheads. To provide a theoretical evaluation, we employ the unit gate model used in [239]: the AND-2/OR-2 gate is equal to 1 unit gate, the NOT gate is equal to 0.5 unit gate, the XOR-2 gate counts as 2 unit gates, and the unit gates of the full adder and half adder are 7 and 3, respectively. Based on this model, Table 4.4 reports the unit gates of each component used in the RAD2^k multipliers.

For the partial product generation, ACCR4 uses $n/2$ radix-4 encoders and $(n/2) \times n$ radix-4 partial product generators to produce the $n/2$ n-bit partial products. Correspondingly, RAD2^k employs $(n - k)/2$ radix-4 encoders, 1 radix-2^k encoder, $((n - k)/2) \times n$ radix-4 partial product generators, and $n + k - 2$ radix-2^k partial product generators. For the partial product accumulation, we choose the Wallace tree and a final fast adder, like in the design of Chapter 3. ACCR4 accumulates $n/2 + 1$

Table 4.4: Unit gates per component of RAD-2^k multipliers.

Component	Reference	Unit Gates
Radix-4 Encoder	Eq. (4.4)–(4.6)	5.5
Radix-2^k Encoder	Eq. (4.7)–(4.11)	41.5
Radix-4 PP Generator	Fig. 3.2c	5
Radix-2^k PP Generator	Fig. 4.2	9

Table 4.5: Unit gates of 16-bit RAD2^k multipliers ($k = 6, 8, 10$).

Multiplier Stage	ACCR4	RAD64	RAD256	RAD1024
Radix-4 Encoding	44	25	20	15
Radix-2^k Encoding	–	41.5	41.5	41.5
Radix-4 PP Generation	640	400	320	240
Radix-2^k PP Generation	–	180	198	216
PP Accumulation	784	560	448	336
Final Addition	400	400	400	400
Total Unit Gates	1868	1606.5	1427.5	1248.5
Reduction	–	14%	24%	33%

operands, i.e., $n/2$ partial products and 1 operand with the correction and constant terms, while RAD2^k accumulates $(n - k)/2 + 2$ operands. Considering the logic required for the multi-operand accumulation of n-bit numbers in carry-save form [238], as well as our analysis in Section 3.4, ACCR4 requires $7n(n - 2)/2$ unit gates, while RAD2^k requires $7n(n - k)/2$ unit gates. Finally, regarding the final $2n$-bit fast addition, which is performed in both ACCR4 and RAD2^k, it requires $2n$ half adders, $n \log_2 2n$ propagate group circuits (each one is 3 unit gates), and $2n$ XOR-2 gates.

Based on the above analysis, Table 4.5 includes the per-stage and total number of unit gates for ACCR4 and RAD2^k ($k = 6, 8, 10$). As shown, even though the entire encoding component of the RAD2^k multipliers has ~1.5× more unit gates than the encoder of ACCR, their partial product generation and accumulation requires less resources than that of ACCR. More specifically, the RAD2^k partial product generation and accumulation are up to 29% and 57% better, respectively. In total, the logic reduction achieved by RAD2^k is up to 33%. We note that the unit gate model is a simplified model (e.g., it does not take into account the interconnections complexity), and it only gives a rough estimation for the area reduction. The exact resource gains are presented in Section 4.3.3.

4.3.2. Error Analysis

The quality of the results calculated by approximate circuits is of utmost importance, thus, the analysis of the errors induced by the approximations constitutes a separate object of study. In this context, we evaluate the accuracy of the RAD2^k multipliers based on the execution of their software models emulating the logic-level approximations. Because our approximations can be emulated at software level, we can provide fast and accurate error analysis. This is an important feature of our approximate designs, eliminating the need for time-consuming hardware simulations to obtain the approximate results. In our analysis, we employ error metrics that are widely used in the field of Approximate Computing [149, 151, 257]. These metrics aim to evaluate the significance and the frequency of the errors in approximate arithmetic circuits.

To evaluate the mean error of our designs, we calculate the Mean Relative Error Distance (MRED), which is the average of the Relative Error Distance (RED) values for a set of inputs. RED refers to the arithmetic difference between the accurate and the approximate result, divided by the accurate result. In our case, let $A \cdot B$ be the accurate multiplication and $A \cdot \tilde{B}$ be the approximate multiplication (calculated by the RAD2^k multipliers). B is accurately encoded as in Eq. (4.1) and $\tilde{B}$ is approximately encoded as in Eq. (4.12). Considering these encodings, the RED of the multiplication is calculated by Eq. (4.17).

$$
\text{RED}_{AB} = \frac{|A \cdot B - A \cdot \tilde{B}|}{|A \cdot B|} = \frac{|B - \tilde{B}|}{|B|} = \frac{|y_0^{R2^k} - \hat{y}_0^{R2^k}|}{\left| \sum_{j=k/2}^{n/2-1} 4^j y_j^{R4} + y_0^{R2^k} \right|} = \text{RED}_B \qquad (4.17)
$$

Hence, the RED of the RAD2^k multipliers depends only on B, i.e., $\text{RED}_{AB} = \text{RED}_B$. Let p_A and p_B be the Probability Density Functions (PDFs) of A and B, respectively. In this case, MRED is calculated by Eq. (4.18).

$$
\text{MRED} = \sum_{\forall A,B} p_A(A) \cdot p_B(B) \cdot \text{RED}_{AB} = \sum_{\forall A} p_A(A) \cdot \sum_{\forall B} p_B(B) \cdot \text{RED}_B =
$$
$$
= \sum_{\forall B} p_B(B) \cdot \text{RED}_B = \sum_{\forall B} p_B(B) \cdot \frac{|y_0^{R2^k} - \hat{y}_0^{R2^k}|}{\left| \sum_{j=k/2}^{n/2-1} 4^j y_j^{R4} + y_0^{R2^k} \right|} \qquad (4.18)
$$

Therefore, the MRED of the RAD2^k multipliers for a set of input pairs $\{A, B\}$ can be calculated using only the values of B and their PDF values (e.g., for uniform input distribution: $p_B(B) = 1/2^n$).

The possibility of RED $> M\%$ (PRED$_M$) is another widely used error metric [149, 151]. More specifically, this metric is defined as shown in Eq. (4.19).

$$\text{PRED}_M = p(\text{RED}_{AB} \geq M\%) = \frac{\#A \cdot B|_{\text{RED}_{AB} \geq M\%}}{\#A \cdot B} \tag{4.19}$$

In our error analysis, we consider 16-bit multiplication. In Figure 4.4a, we illustrate the PDFs of RED for the RAD64 multiplier. RAD256 and RAD1024 deliver a similar curve, proving that the probability of having small RED is high. In Figures 4.4b–4.4d, we plot the RED distribution for RAD2^k ($k = 6, 8, 10$) with respect to the encoded operand B. As shown, the error follows a Gaussian distribution with bounds for maximum error, i.e., the error is large only for a small near-zero interval. For the rest values of B (either positive or negative), RED $\in [0\%, 1\%]$. In particular, RAD64 exhibits RED larger than 10% only if $B \in [-100, 100]$. The respective intervals for RAD256 and RAD1024 are $[-400, 400]$ and $[-1500, 1500]$. These intervals represent a very small fraction of all the possible values of B for 16-bit arithmetic. As a result, PRED$_{10}$ is 0.001%, 0.004%, and 0.02% for RAD64, RAD256, and RAD1024, respectively.

Overall, the proposed 16-bit RAD2^k multipliers introduce small errors for the majority of the multiplications. Even RAD1024, which applies the most aggressive approximations, features a MRED of 0.93%, while its PRED$_2$ and PRED$_{10}$ are 6.74% and 0.02%, respectively. We note that the accuracy loss of approximate designs should be examined along with the respective gains in resources. Thus, our analysis in the next section examines the trade-off between accuracy and resources.

An advantage of the RAD2^k multipliers compared to other state-of-the-art designs is that their error, as shown in Eq. (4.17), depends only on the operand that is encoded (B). This feature enables the fast calculation of the RED-based error metrics. In case RED was a function of both A and B, its calculation would require to use a larger input dataset (to cover different pair combinations) and also execute the approximate multiplication.

4.3.3. Experimental Results

This section includes the evaluation of the RAD2^k multipliers in terms of resources (delay, area, power and energy). It also examines the accuracy of the approximate
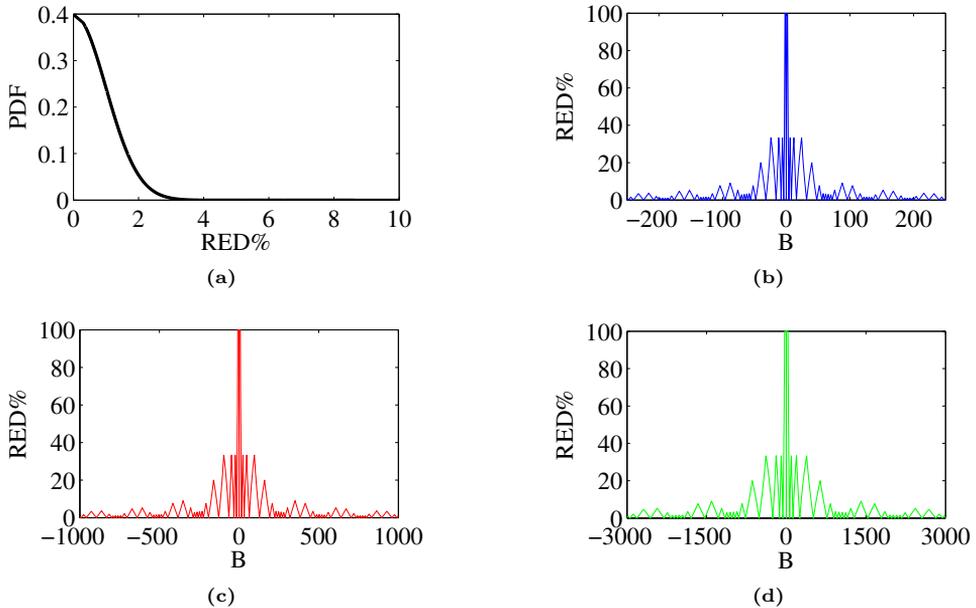
Figure 4.4: (a) Probability density function of RED for RAD64. The RED distribution with respect to the values of B for **(b)** RAD64, **(c)** RAD256, and **(c)** RAD1024.

designs in parallel with the provided resource gains. The evaluation consists of two stages, i.e., the comparison of RAD2^k with state-of-the-art approximate multipliers and the exploration of RAD2^k's bit-width scaling.

The designs are implemented in Verilog, synthesized with Synopsys Design Compiler and the TSMC 65-nm standard-cell library, and simulated with Mentor Graphics QuestaSim. Both synthesis and simulation are performed at 1V, i.e., the nominal supply voltage. The critical path delay and the area of the circuits are reported by Synopsys Design Compiler, while the power consumption is measured with Synopsys PrimeTime. Energy is defined as the product of power and delay. Moreover, we define the gain of the approximate design as the relative resource reduction from the accurate design.

Comparative State-of-the-Art Evaluation

For comparison, we implement the accurate radix-4 (ACCR4) and radix-8 (ACCR8) multipliers, as well as relevant state-of-the-art approximate multipliers [142, 149, 151].

R8ABM1 and R8ABM2-15 [151] employ the radix-8 encoding and calculate the product $3A$ with an adder operating approximately for the 8 LSBs. R8ABM2-15 extends the approximation by truncating the 15 LSBs of the partial products. R4ABM1-14, R4ABM1-16, R4ABM2-14, and R4ABM2-16 [149] use the accurate radix-4 encoder for producing the MSBs of the partial product matrix and an approximate radix-4 encoder for the 14/16 LSBs. The difference between R4ABM1 and R4ABM2 is that the latter design performs more aggressive approximations. For the radix multipliers of [151] and [149], the partial product accumulation is performed accurately with a Wallace tree and a fast adder, like in RAD2^k. Finally, we implement DRUM6 [142], which selects a 6-bit segment, starting from the leading non-zero bit of the input operands, and sets the LSB of the truncated values to '1'.

Table 4.6 presents the results from the synthesis of the multipliers for 16-bit input operands. The circuits are configured to operate at their critical path delay, i.e., at maximum frequency. We note that Table 4.6 also reports error metrics (MRED and PRED$_2$), which have been obtained by performing simulations over all the possible input combinations. At first, we notice the inefficiency of ACCR8 compared to ACCR4, which proves that the conventional accurate high-radix encodings impose significant overheads. Next, we compare the accuracy and the hardware efficiency of the examined multipliers.

RAD64 gives the best MRED among all the designs, as approximations are performed only in the 6 LSBs of B. In addition, the approximated values have small absolute distance from the accurate ones. In R4ABM1-14, a very small error is introduced, considering that only four entries of the radix-4 encoder's K-Map are modified [149], while R8ABM1 has a small MRED because the approximate adder that calculates $3A$ involves both accurate and approximate parts. However, the errors of R8ABM1 are of greater significance, namely there are large absolute distances from the accurate results. In R8ABM2-15, the error does not increase significantly, although the 15 LSBs of the partial products are truncated. This is due to the correction '1' added to the 17-th bit that compensates the error generated by the truncated lower part [151]. Finally, DRUM6 delivers the worst MRED among all the multipliers with 1.47%, although its error distribution is bounded. Our RAD2^k multipliers deliver MRED smaller than 1% as standalone circuits, meaning that they can be used in real-world applications that tolerate mean errors of 5% or 10% [258].

DRUM6 has the worst critical path delay, as it initially calculates the absolute value of the products [142], while our designs RAD64, RAD256, and RAD1024 are the fastest circuits, with delays of 0.72ns, 0.69ns, and 0.65ns, respectively. Regarding energy, R8ABM1 delivers the worst consumption among all the multipliers, as it may use an approximate adder to calculate $3A$, but it contains a 7-bit precise adder to avoid very large errors [151]. The energy is improved in R8ABM2-15 due to the

Table 4.6: Experimental results of the 16-bit approximate multipliers on TSMC 65-nm standard-cell.

Design	Delay (ns)	Power (µW)	Area (µm^2)	Energy (µW·ns)	MRED (%)	PRED$_2$ (%)
ACCR4	0.75	4998	4153	3749	–	–
ACCR8	0.80	5343	4639	4274	–	–
RAD64	0.72	4497	3489	3238	0.08	0.42
RAD256	0.69	3493	2769	2410	0.28	1.69
RAD1024	0.65	3422	2624	2224	0.93	6.74
R8ABM1 [151]	0.77	5058	4210	3895	0.15	1.05
R8ABM2-15 [151]	0.74	3377	2926	2499	0.61	2.70
R4ABM1-14 [149]	0.74	4676	3958	3460	0.12	0.41
R4ABM1-16 [149]	0.73	4447	3725	3246	0.49	1.49
R4ABM2-14 [149]	0.73	4648	3732	3393	0.24	0.68
R4ABM2-16 [149]	0.72	4307	3467	3101	1.18	2.50
DRUM6 [142]	1.07	2148	3993	2298	1.47	28.85

truncated partial product bits. However, as our multipliers feature small MRED, the truncation technique can also be applied to them to provide even better resource gains. The radix-4 multipliers of [149] exhibit similar results, with R4ABM2-16 delivering the best area and energy gains from their design family. RAD1024 features the best energy consumption, with DRUM6 being second. However, the latter delivers significantly larger MRED than all the other multipliers. The same applies to its PRED$_2$, which is increased (28.85%), while the other multipliers have PRED$_2$ smaller than 3% (apart from RAD1024 that has PRED$_2$ equal to 6.74%).

Figure 4.5 illustrates the scatter plot of the examined multipliers for MRED and energy consumption. The purpose of this plot is to highlight the most prominent designs when considering both error and energy consumption. As shown, the Pareto Front is formed exclusively by RAD2^k multipliers. Namely, RAD2^k constitute the most efficient approximate multiplier alternatives compared to all the examined state-of-the-art approximate multipliers, as they exhibit the best energy–MRED trade-off. RAD64 attains the smallest MRED value and should be preferred when error is of high importance, while RAD1024 delivers the highest energy reduction. Finally, RAD256 features significant energy reduction for a very small error value.

Next, we evaluate the RAD2^k multipliers by exploring the energy and area gains compared to the accurate radix-4 multiplier. The main target of Approximate Computing is to trade accuracy for energy gains, i.e., generate good enough results at comparable performance and lower energy consumption. Thus, in this evaluation, we leverage the delay slack between the RAD2^k multipliers and the accurate one,
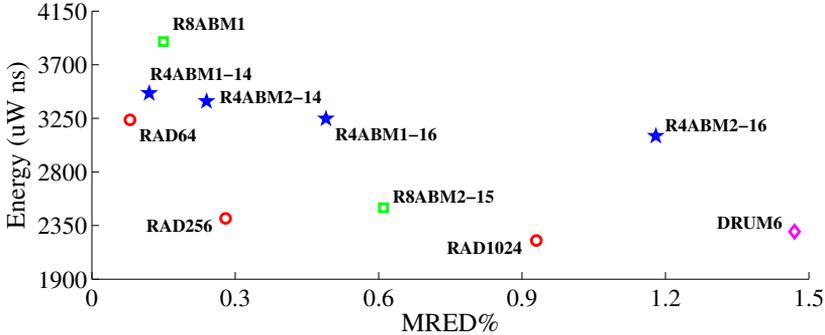
Figure 4.5: Comparative Pareto analysis for the approximate multipliers considering MRED and energy.
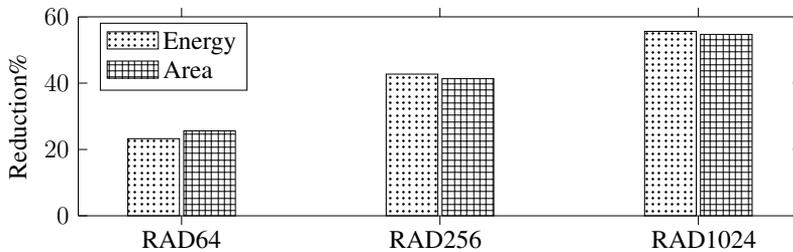
Figure 4.6: Area and energy gains of RAD2^k compared to the accurate radix-4 multiplier.

and targeting energy efficiency, we synthesize and simulate the RAD2^k designs at the critical path delay of ACCR4. Figure 4.6 reports the delivered gains of RAD2^k multipliers compared to ACCR4, when they operate at the same frequency. Remarkable reduction in area and energy is attained by the RAD2^k multipliers, and it is shown that the gains become larger when using higher radix encoding, because less partial products are generated. In these designs, the number of the partial products is reduced from 8 to 6, 5 and 4, and thus, the area and the depth of the accumulation tree is reduced by up to 50%. As expected, RAD1024 delivers the largest gains (56% in energy consumption and 55% in area).

The presented analysis regards fixed-point arithmetic, but it can be also extended for floating-point multiplication. In the latter case, the proposed RAD2^k multipliers can replace the fixed-point multiplication of the mantissas, which is executed in floating-point multiplication. Considering that the error of RAD2^k is very small, it will affect only the mantissa's accuracy, whereas the exponent value (calculated by an accurate adder) will remain the same as in the accurate floating-point multiplication. Therefore, similar error values are expected in RAD2^k-based floating-point multiplication. In terms of energy gains, they depend on the floating-point repre-
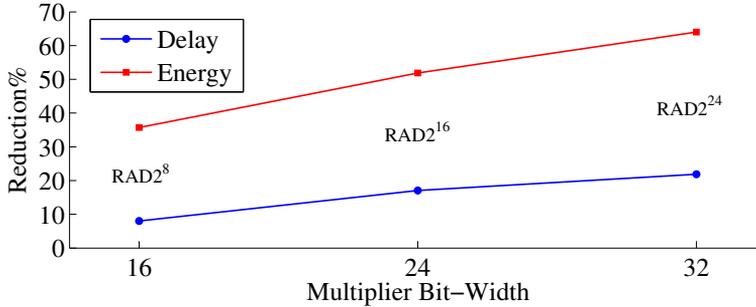
Figure 4.7: Energy and delay gains of RAD2^k for scaled bit-width and MRED = 0.28% (error constraint).

sentation. For higher floating-point precision, similar or even larger energy gains are expected.

Evaluation of Bit-width Scaling

Finally, we examine the efficiency of RAD2^k when increasing the multiplier's bit-width, i.e., to 24 and 32 bits. We consider ACCR4 as the baseline design and an MRED of 0.28% as a quality constraint that should be satisfied. We select 0.28% as error constraint because it is the MRED of the 16-bit RAD256 multiplier, which is the most efficient design when taking into account both the provided energy gains and the error values. Namely, as shown in Figure 4.5, RAD256 attains a very efficient energy-error trade-off, i.e., significant error reduction for very small error.

Figure 4.7 presents the scaling of the gains in delay and energy with respect to the multiplier's size. For the 16-bit arithmetic, we implement the high-radix-2^8 multiplier (RAD256), while for 24-bit and 32-bit arithmetic, we implement the high-radix-2^{16} and high-radix-2^{24} multipliers, respectively, in order to deliver the same MRED of 0.28%. The reductions in energy consumption and delay scale up to 64% and 22%, respectively, while the quality constraint is satisfied. Thus, the results show that as the multiplier's size increases, the proposed hybrid high-radix encoding achieves larger gains in critical path delay and energy consumption for the same error. The scaling behavior is theoretically confirmed, as the RAD2^k designs of Figure 4.7 have an MRED of 0.28%, while generating 37%, 58%, and 60% less partial products for 16, 24, and 32 bits, respectively. More specifically, each one generates 5 partial products in total (1 approximate and 4 accurate). In contrast, ACCR4 generates 8, 12, and 16 partial products, respectively.

4.4. Conclusion

In this chapter, we introduced an approximate hybrid high-radix encoding for generating the partial products of a signed multiplier. The most significant bits of the multiplicand are encoded with the accurate radix-4 encoding, while its k least significant bits are encoded with an approximate high-radix-2^k encoding. The parameter k determines the approximate high-radix encoding, and thus the approximation degree, and it can be tuned to provide the desired trade-off between accuracy and resources. Our approximation approach maps all the high-radix values to a set including only the 4 largest powers of two. Therefore, it surpasses the bottlenecks of the conventional high-radix encodings, which provide partial product reduction, however, they suffer from increased encoding logic. The error of the RAD2^k multipliers follows a Gaussian distribution with near-zero average. The mean relative error of our designs lies in the range 0.08%–0.93%, and also, it depends only on the approximately encoded operand, allowing the fast calculation of the error metrics and eliminating the need for exhaustive circuit simulations. In terms of resource savings, our designs achieve up to 56% energy and 55% area gains compared to the accurate radix-4 multiplier, when operating at the same frequency. Compared to state-of-the-art approximate multipliers, RAD2^k constitute better approximate design alternative, as they form the Pareto front in our analysis involving energy and error. Finally, our approximation technique is scalable, delivering larger resource gains as the multiplier's size increases, while keeping the error constant. More specifically, for 32-bit multiplication, the gains reach to 64% and 22% in energy and delay, respectively, while the mean relative error is retained at 0.28%.

Chapter 5

Dynamic Approximation: Runtime-Configurable Arithmetic Circuits

The challenging deployment of Digital Signal Processing (DSP) and Artificial Intelligence (AI) algorithms pushes the community to examine alternative design approaches, such as Approximate Computing. This novel design paradigm provides valuable resource gains by exploiting the error tolerance of the DSP/AI applications. Nevertheless, approximate designs with fixed approximation configuration provide limited flexibility and cannot accommodate different workloads or tune the quality of the results with respect to the given accuracy and energy constraints. As a result, there is a growing need for approximate circuits and systems that support multiple approximation configurations and can seamlessly change among them at runtime. In this context, we design runtime-configurable approximate multipliers for integer/fixed-point and floating-point arithmetic. We employ two orthogonal approximation techniques, i.e., partial product perforation and partial product rounding, to enlarge the approximation space, and we provide a low-overhead configuration scheme for tuning the approximation at runtime. The evaluation is performed for both the design-time (static) and runtime (dynamic) approximate variants, and it involves an in-depth error analysis and diverse experimental results. The error analysis shows that our designs feature slow error scaling with multiple values, which allows them to satisfy various accuracy constraints. According to the experimental results, the design-time variants outperform all the examined state-of-the-art multipliers, considering either error or resource gains as target. The runtime variants provide negligible area overhead (e.g., 4 % and 2 % for half and single floating-point precision, respectively) and smaller energy gains, however, they still deliver remarkable gains versus the accurate multiplier and other state-of-the-art design-time multipliers. In more detail, the fixed-point runtime variant consumes only up to $1.2\times$ and $1.7\times$ more energy than its design-time variant for low-strength and more aggressive approximation, respectively. Correspondingly, the floating-point runtime variant provides $\sim1.4\times$ and $\sim1.6\times$ less energy gains in half and single precision, respectively.

*This chapter is based on our **publications** in [144,145].*

5.1. Introduction

The proliferation of compute-intensive workloads from domains such as Digital Signal Processing (DSP) and Artificial Intelligence (AI) is changing the landscape in both cloud and embedded computing. The efficient deployment of DSP/AI applications is a first-class concern due to their increased computational and memory demands, as well as the resource constraints imposed by the computing systems (e.g., specific energy budget or limited number of available processing units). For this reason, the research community explores new design alternatives towards power-efficient and high-performance computing.

One of the most attractive and well-established solutions is the emerging paradigm of Approximate Computing [16, 17, 19], which exploits the inherent approximate nature and error resilience of the DSP/AI applications [29, 240]. This design approach trades accuracy loss for resource gains, e.g., in power, area, or throughput. In particular, errors are inserted in the computations based on a systematic and disciplined approach, which aims to provide resource gains while retaining the quality of the results at acceptable levels. Approximation techniques are applied at all layers of the computing pyramid, i.e., from algorithms to software and down to circuits and transistors.

The increased diversity of the real-world error-resilient applications demands flexible approximate designs that offer various approximation configurations. More explicitly, the approximate system/circuit/architecture needs to be capable of adjusting the approximation degree in order to satisfy the end-to-end application-specific accuracy, while providing the desired performance within the constrained power envelope. Even for the same application, different input distributions may require a new approximation configuration to provide the acceptable quality of results (e.g., consider an approximate video processor that inputs frames with different content). As a result, there is a growing need for approximate designs that can: (i) provide multiple approximation configurations (namely different levels of accuracy), and (ii) dynamically configure their approximation degree, either offline (e.g., before starting the execution of a new application) or at runtime, with respect to the given accuracy/power constraints.

In this chapter, motivated by the demand for dynamic approximation configuration, we target the arithmetic circuits and design approximate multipliers that can configure their *approximation degree at runtime.* Contrary to the RAD2^k circuits of Chapter 3, which are configured at design-time to implement a fixed (frozen) approximation, the designs of this chapter provide a larger approximation space and seamless dynamic configuration. The proposed family of runtime-configurable approximate multipliers can be integrated in processors and custom hardware accelerators that need to tune

the approximation degree. We note that we neither examine which approximation configuration should be selected nor we apply automatic approximation tuning. Our goal is to implement circuits that can change approximation on an efficient way (without significant area overhead and timing penalties).

Additionally, compared to Chapter 3 and Chapter 4, we examine floating-point arithmetic, i.e., we design both runtime-configurable fixed- and floating-point multipliers. Arithmetic computations impose a trade-off in range, precision, and hardware resources. Range is the capability of representing small/large numbers, while precision is the differentiation between nearby values. Fixed-point arithmetic delivers hardware-friendly designs, but it sacrifices range and offers limited precision. On the other hand, floating-point arithmetic provides a larger range of values and higher precision for the same word-length, but it suffers from increased hardware cost. Furthermore, the floating-point format offers a simplified programming model, contrary to fixed-point, which needs to compensate for the quantization noise. Below, we discuss the significance of the floating-point arithmetic and analyze what pushed us to design floating-point multipliers.

Numerous compute-intensive algorithms use a wide range of values and require high precision. Therefore, floating-point arithmetic is favored in applications from domains such as DSP, computer graphics, scientific computing, and speech recognition, which handle real numbers and produce results with unpredictable range. However, the increased hardware cost of floating-point calculations results in using fixed-point arithmetic or alternative data formats that are more hardware-efficient, but they do not provide the benefits of floating-point. For this reason, there is also limited integration of Floating-Point Units (FPUs) in embedded devices, e.g., the commercial Field-Programmable Gate Arrays (FPGAs) do not have hardwired FPU blocks [259]. The power inefficiency of FPUs is also proven by a recent study on Graphics Processing Units (GPUs) [260]. This study revealed that the portion of the power consumed for arithmetic operations in compute-intensive benchmarks reaches more than the 70% of the total power, with the FPU being the most power-hungry unit. Specifically for floating-point multiplication, it is widely used in operations such as convolution, matrix multiplication, and Fourier transform, and thus, its efficiency inherently affects the entire application. Recent studies on general OpenCL applications from the AMD APP SDK showed that over 85% of the floating-point arithmetic involved multiplication [261]. Although floating-point multipliers are more expensive in power and area, they have received less research attention than their fixed-point counterparts for disciplined approximations [142, 143, 146, 149, 150, 154, 155, 250, 253, 254, 262].

Regarding the technical details of our work, we employ two orthogonal approximation techniques, i.e., *partial product perforation* and *partial product rounding*. The first

technique perforates entire partial products, while the second technique rounds them to a smaller bit-width. These two techniques are applied in collaboration to generate a large approximation space, satisfying the requirement for multiple approximation configurations and varying levels of accuracy. More specifically, at first, we discard partial products starting from the least significant, and then, we apply rounding to the remaining ones. Besides offering a large approximation space, we select these two techniques because their configuration is associated with the input operands, and thus, we can easily change it at runtime with negligible area overhead. This overhead is $2n$ AND gates and two n-bit control signals (one per operand), where n is the operand bit-width. The approximation configuration is determined by simply setting the bits of the control signals to either '1' or '0'.

The **contribution** of this chapter is summarized as follows:

   (i) We highlight the significance of dynamic approximation configuration and integrate this attractive feature in the design of approximate multipliers.

  (ii) We extend our approximation techniques to floating-point arithmetic, which imposes increased hardware cost compared to integer/fixed-point arithmetic.

 (iii) We combine two orthogonal approximation techniques to generate a large approximation space, which can accommodate numerous accuracy constraints and explore the accuracy–energy trade-off to provide the most efficient solution.

 (iv) We introduce a low-overhead dynamic configuration scheme for adjusting the approximation degree at runtime.

  (v) We show that the proposed solution outperforms related state-of-the-art designs in both fixed- and floating-point arithmetic, providing remarkable area and energy gains for comparable error values and slow error scaling.

The remainder of this chapter is organized as follows. Section 5.2 introduces the proposed approximation techniques and the design-time and runtime variants of our approximate fixed- and floating-point multipliers. Section 5.3 includes the evaluation of the proposed designs, including error analysis and comparative experimental results. Finally, Section 5.4 draws the conclusions.

5.2. Design of Runtime-Configurable Approximate Multipliers

To facilitate the dynamic configuration of the approximation degree, we target to approximate partial product generation, i.e., the multiplication stage that is as-

sociated with the input operands. Moreover, to provide multiple approximation configurations, we employ two orthogonal approximation techniques, namely, the partial product perforation and the partial product rounding. The approximation degree of each technique is tuned independently and can be easily tailored to satisfy various design constraints, e.g., maximum error bound or specific power budget. Regarding the multiplication scheme, we select the radix-4 encoding for generating the partial products, as it outperforms other well-established algorithms [226]. Firstly, we present our approximate design for integer/fixed-point arithmetic, which is also the main approximate component of our floating-point unit. Subsequently, we present the extension of our design to support dynamic approximation configuration.

5.2.1. AxFXU: Approximate Fixed-Point Multiplier

Let $A = \langle a_{n-1}a_{n-2}\cdots a_0\rangle_{2\text{'s}}$ and $B = \langle b_{n-1}b_{n-2}\cdots b_0\rangle_{2\text{'s}}$ be two n-bit 2's-complement numbers. The accurate radix-4 multiplication $A \times B$ is performed by generating and accumulating $n/2$ partial products. The application of partial product perforation omits the generation of P successive partial products, starting from the least significant ones. Afterwards, partial product rounding is applied to round the remaining partial products to a smaller bit-width, i.e., it truncates their $R - 1$ Least Significant Bits (LSBs) and add their $(R - 1)$-th bit to the most significant part. We name this family of approximate circuits as AxFXU$|_{P,R}$, which denotes approximate fixed-point multiplication with P and R configuration of perforation and rounding, respectively.

Using the accurate radix-4 encoding for B, as presented in Chapter 4, the approximate multiplication in AxFXU is performed as shown in Eq. (5.1).

$$A \times B|_{P,R} = \sum_{j=P}^{n/2-1} 4^j \tilde{P}P_j = \sum_{j=P}^{n/2-1} 4^j A_R \cdot y_j^{R4} \qquad (5.1)$$

$$\text{where} \quad P \in [0, n/2 - 1), \quad R \in [0, n - 1) \qquad (5.2)$$

$$y_j^{R4} = -2b_{2j+1} + b_{2j} + b_{2j-1} \implies y_j^{R4} \in \{0, \pm 1, \pm 2\} \qquad (5.3)$$

$$A_R = \langle a_{n-1}a_{n-2}\cdots a_R\rangle_{2\text{'s}} + a_{R-1} \qquad (5.4)$$

Therefore, the final approximate product of AxFXU is calculated by accumulating the $n/2 - P$ most significant rounded partial products $\tilde{P}P_j$. The application of the two approximation techniques is independent, as illustrated in Figure 5.1. As shown,
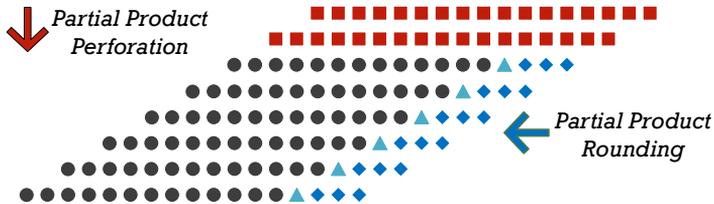
Figure 5.1: Approximate partial product matrix that is generated using partial product perforation ($P = 2$) and rounding ($R = 4$).
Symbols: ■: perforated bits   ◆: truncated bits   ▲: bits added to the remaining words for rounding
●: accurately generated (remaining) bits

the partial product matrix is reduced vertically by perforation and horizontally by rounding.

The partial product generation technique removes P radix-4 encoders and $P \times n$ 1-bit partial product generators from the accurate multiplier (see Section 4.2 of Chapter 4). These components are required to implement the perforated partial product bits (red rectangles in Figure 5.1). To improve the efficiency of rounding and avoid possible penalties due to the addition of a_{R-1} (blue rhombuses in Figure 5.1), we adopt our bit-level manipulations for DLSB arithmetic (see Section 3.4 of Chapter 3), assuming $a_{0+} = a_{R-1}$. As a result, the remaining partial product bits are generated like in the accurate multiplier, and only one XOR gate is added in the calculation of each correction term, i.e., $sign_j \oplus a_{R-1}$ is used instead of $sign_j$.

5.2.2. AxFPU: Approximate Floating-Point Multiplier

Next, we introduce the family of our approximate floating-point multipliers, which are named AxFPU$|_{P,R}$. This design makes use of AxFXU for multiplying the mantissas, it employs the logic of the conventional floating-point multiplier for the rest operations (e.g., exponent addition), and also, it integrates some extra functionalities to handle the approximations. Before introducing AxFPU, we make a brief introduction in floating-point arithmetic and the representation of the floating-point data. This information is required to understand the design of the floating-point multiplier, but mainly, we include it as background to the error analysis of AxFPU.

The Floating-Point Arithmetic

Floating-point arithmetic performs a systematic approximate mapping of the real arithmetic. More specifically, the floating-point format is an established encoding

for representing a finite subset of the continuum of real numbers. A floating-point datum can be a finite non-zero number, a signed zero (± 0), a signed infinity ($\pm \infty$), or a Not-a-Number (NaN).

In base (radix) $b \geq 2$, the finite non-zero numbers and the signed zero are represented approximately with a fixed number of significant digits (called mantissa), and they are scaled by raising the base to an integer (called exponent). In addition, they include a sign that indicates if they are positive or negative. These three parameters are defined in more detail as follows:

- the sign is '0' (positive) or '1' (negative).
- the exponent is any integer that belongs in the interval $[e_{min}, e_{max}]$, where $e_{min} = 1 - e_{max}$.
- the mantissa is any number that belongs in the interval $[0, b)$, and it is represented as $\langle d_0 \cdot d_1 d_2 \ldots d_{m-1} \rangle$, where $0 \leq d_i < b$.

The smallest normal floating-point magnitude is $b^{e_{min}}$. All the non-zero floating-point numbers with magnitude less than $b^{e_{min}}$ are called subnormal, because they lie between zero and the smallest normal magnitude.

The IEEE-754 standard [263] defines various encodings for the binary floating-point format ($b = 2$). The most widely used binary formats are presented in the left side of Table 5.1. The Most Significant Bit (MSB) of a floating-point datum is the sign (S). The exponent is encoded using an offset, referred as exponent bias in the IEEE-754 standard, which is equal to e_{max}. Therefore, the biased exponent E is equal to $e + e_{max}$, where e is the true exponent, and it is stored as a w-bit unsigned integer. Regarding the mantissa, its leading bit (d_0) is implicitly encoded with the biased exponent. Thus, only $m - 1$ bits of mantissa are stored, i.e., $M = \langle d_1 d_2 \ldots d_{m-1} \rangle$, even though the total precision is m bits.

In case the biased exponent E belongs in the interval $[1, 2^w - 2]$, the floating-point datum represents the normal value of Eq. (5.5).

$$(-1)^S \cdot 2^{E - e_{max}} \cdot \left(1 + \sum_{i=1}^{m-1} 2^{-i} d_i \right) \tag{5.5}$$

In case the biased exponent E is zero and the mantissa M is non-zero, the floating-point datum is a subnormal value. Additionally, special numbers (± 0, $\pm \infty$, NaN) are defined depending on the values of exponent and mantissa. All the possible floating-point data types are summarized in the right side of Table 5.1.

Table 5.1: The IEEE floating-point formats and data types [263].

Format Properties	Precision			Datum Type	Case	
	Half	Single	Double		Exponent	Mantissa
Total Bits	16	32	64	Normal	$\in [1, 2^w - 2]$	don't care
- Sign Bit	1	1	1	Subnormal	$= 0$	$\neq 0$
- Exponent Bit (w)	5	8	11	± 0	$= 0$	$= 0$
- Mantissa Bits ($m-1$)	10	23	52	NaN	$= 2^w - 1$	$\neq 0$
Exponent Bias (e_{max})	15	127	1023	$\pm \infty$	$= 2^w - 1$	$= 0$

Design of Accurate Architecture

Before introducing AxFPU, we present the baseline accurate floating-point multiplication architecture. Let A_f and B_f be two n-bit floating-point normal numbers. The product $A_f \times B_f$ is calculated by Eq. (5.6).

$$A_f \times B_f = (-1)^{(S_A+S_B)} \cdot 2^{(E_A+E_B-2e_{max})} \cdot \underbrace{\left(1 + \sum_{i=1}^{m-1} 2^{-i} a_i\right)}_{A=1.M_A} \cdot \underbrace{\left(1 + \sum_{i=1}^{m-1} 2^{-i} b_i\right)}_{B=1.M_B} \quad (5.6)$$

The above multiplication is performed in the following basic steps: (i) calculation of the result's sign, (ii) addition of the exponents, (iii) multiplication of the mantissas, (iv) normalization of the mantissa product, (v) update of the result's exponent, (vi) rounding of the mantissa product, and (vii) handling of special cases. Below, we discuss the technical details of each step.

The sign of the result is calculated by the XOR of the input signs, i.e., $S_R = S_A \oplus S_B$. Regarding the result's exponent, the input exponents (E_A, E_B) are biased, thus, the bias (e_{max}) is removed before their addition. Afterwards, the bias is added to the result's exponent, i.e., in total $E_R = E_A + E_B - e_{max}$. For the mantissa multiplication, all the m mantissa bits are employed. The MSB of each mantissa string is '1', as we assume normal floating-point numbers. Therefore, the multiplication of $1.M_A = \langle 1a_1a_2 \ldots a_{m-1} \rangle$ and $1.M_B = \langle 1b_1b_2 \ldots b_{m-1} \rangle$ is performed. Due to multiplying normal floating-point numbers, the result can be in one of the following forms: (a) $01.xx \ldots x$, (b) $10.xx \ldots x$, or (c) $11.xx \ldots x$. The next step is to normalize the product in case it is in the form (b) or (c), so that there is only one leading '1' before the radix point. To do so, the radix point is moved one place to the left and the intermediate exponent E_R is increased by 1. Otherwise, the exponent and the already-normalized mantissa remain intact. Finally, only the $m-1$ MSBs that are

placed after the radix point can be stored in the mantissa field, thus, bit rounding is applied.

Our architecture also handles special cases that may arise due to the exponent value. Specifically, if the exponent is too small/large to be represented, then underflow/overflow occurs. To consider these special cases, an underflow/overflow detector is employed after the component that performs the exponent update in case of normalization. This detector checks the value of the exponent and decides if the result is normal number, underflow or overflow. If the exponent lies in the interval $[1, 2^w - 2]$, then the result is a normal number. In contrast, if the result is smaller than 1, it is marked as underflow, while if it is bigger than $2^w - 2$, it is marked as overflow. We note that if underflow occurs, the exponent is stored as 0 and the product $A_f \times B_f$ is either a subnormal number or ± 0, depending on the value of the mantissa product (see Table 5.1). Similarly, if overflow occurs, the exponent is stored as $2^w - 1$ and the product $A_f \times B_f$ is either NaN or $\pm \infty$.

Design of Approximate Architecture

The most costly component of the floating-point unit is the mantissa multiplier [264], as the rest of the circuits are comparators, small adders, and multiplexers. Thus, to improve its efficiency, we use AxFXU to multiply the mantissas.

Considering the accurate floating-point multiplication of Eq. (5.6) and the approximate AxFXU multiplication of Eq. (5.1), the multiplication in AxFPU$|_{P,R}$ is calculated by Eq. (5.7).

$$A_f \times B_f|_{P,R} = (-1)^{(S_A + S_B)} \cdot 2^{(E_A + E_B - 2e_{max})} \cdot \underbrace{A_R \cdot \sum_{j=P}^{m/2-1} 4^j y_j^{R4}}_{A \times B|_{P,R}} \tag{5.7}$$

Figure 5.2 illustrates the block diagram of AxFPU. In comparison with the baseline accurate design, an additional detection component is implemented, because due to the approximations, the mantissa multiplication of very large operands may produce a result in the form $00.xx \ldots x$. We remind that the result of a conventional floating-point multiplication is always in the form $01.xx \ldots x$, $10.xx \ldots x$, or $11.xx \ldots x$. In AxFPU, this may happen due to discarding negative partial products. In such a case, the absence of these products results in not decreasing the value of the final product (as in the accurate multiplication), and thus, the floating-point result appears in the aforementioned form. This case is handled as overflow, however, we note that according to our experimental results, the possibility of having either the conventional or this special overflow is around 0.35% on average for half and single floating-point
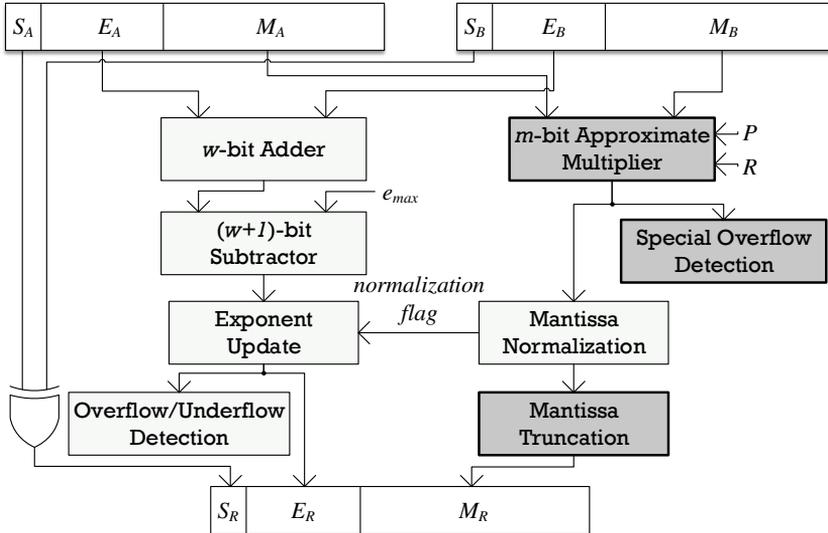
Figure 5.2: The approximate floating-point multiplication architecture of AxFPU$|_{P,R}$. The configuration parameters P and R adjust the approximation degree of perforation and rounding, respectively. The highlighted blocks are modified/added compared to the conventional accurate design.

precision. Another differentiation of AxFPU compared to the accurate design regards the rounding process. Considering that the mantissa product is now approximate, its rounding to the $m-1$ MSBs, which is typically applied after normalization, offers negligible accuracy gain. Therefore, we eliminate the rounding unit, i.e., the second most power-consuming component of the floating-point multiplier [264], and we simply truncate the leftover LSBs. We note that this design choice is optional and the designer can apply rounding instead of truncation.

5.2.3. Dynamic Configuration of the Approximation Degree

The AxFXU$|_{P,R}$ and AxFPU$|_{P,R}$ circuits apply static approximation, namely, the approximation parameters P and R are configured at the design time and cannot change after the implementation. In this section, we introduce their runtime variants (DyFXU and DyFPU), which can dynamically configure their approximations degree, i.e., adjust the parameters P and R at runtime. These designs employ the accurate radix-4 multiplier, thus, they also support operation in fully-accurate mode. The two approximation techniques that are applied (partial product perforation and partial product rounding) in conjunction with the selected radix-4 multiplication algorithm facilitate the design of a low-overhead scheme for dynamic configuration because:

(i) the $2P - 1$ LSBs of B are used only for the generation of the P least significant partial products.

(ii) the $R - 1$ LSBs of A are used only for the generation of the $R - 1$ LSBs of each partial product.

From the first ascertainment, we conclude that the truncation of the $2P - 1$ LSBs of B is equivalent to configuring perforation to P. From the second ascertainment, we conclude that the truncation of the $R - 1$ LSBs of A, along with the addition of a_{R-1} to the most significant part of A, is equivalent to configuring rounding to R. Therefore, to dynamically configure the parameters P and R, we drive the i-th bit of each operand to an AND gate along with an input signal s_i $(i = 1, 2, \ldots n)$. The value of s_i determines whether the i-th bit of the operand will be "virtually" truncated $(s_i = 0)$ or not $(s_i = 1)$.

Figure 5.3 illustrates the dynamic configuration of the approximation degree to $P = 2$ and $R = 4$. The control signal s_i is set to '0' in the AND gates driven by:

(i) the 3 LSBs of B, and thus, the 2 least significant partial products are not calculated (they are 0) $\rightarrow$ perforation is configured to $P = 2$.

(ii) the 3 LSBs of A, and thus, the 3 LSBs of each partial product are not calculated (they are 0) $\rightarrow$ rounding is configured to $R = 4$.

In the rest AND gates, s_i is set to '1', and the corresponding partial products bits are accurately generated.

It is obvious that the runtime variants do not provide area gains, as they implement the entire n-bit multiplier plus $2n$ AND gates and $2n$ 1-bit input signals. However, they still deliver significant energy gains while offering accurate operation mode and multiple levels of accuracy, which can be configured at runtime. In terms of performance, the critical paths are not reduced, and thus, the clock frequencies are set at their nominal value, i.e., that of the accurate designs, to satisfy the accurate operation mode. To provide performance gains, the frequencies can be dynamically scaled to satisfy the critical paths of the respective design-time multipliers, e.g., operate DyFXU$|_{2,4}$ at the maximum frequency of AxFXU$|_{2,4}$.

The control signals that tune the approximation degree are directly exposed to the system. Namely, they can be seamlessly set to 0/1 at runtime to increase/decrease the approximation degree. The tuning can be performed by a high-level policy (e.g., in an approximate custom processor/accelerator) that sets the control signals with respect to pre-stored error values and an error constraint defined by the application/user. For instance, the mean error for all the combinations of the approximation parameters P and R can be calculated offline (e.g., for an application-specific input distribution or for all the possible inputs), in order to be stored and accessed by the high-level policy.
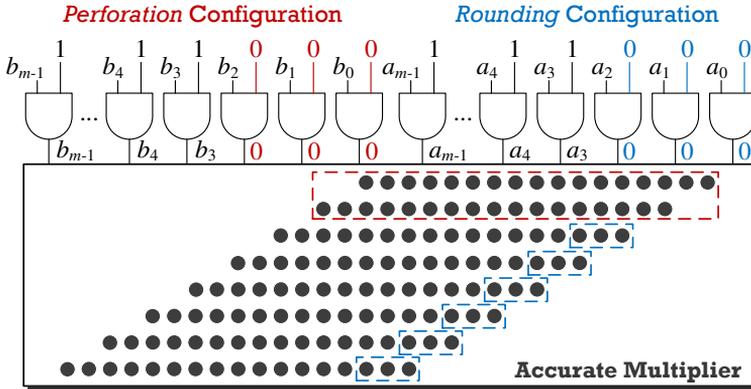
Figure 5.3: Dynamic configuration of the approximation degree by adjusting perforation and rounding via control signals. The "virtual" truncation of 3 LSBs in each operand sets the highlighted partial product bits to 0, activating $P = 2$ (perforation configuration) and $R = 4$ (rounding configuration).

In this scenario, given an error constraint (that may change at runtime), the high-level policy selects the most resource-efficient approximation configuration that satisfies it. A similar approach is proposed in [265], where the authors use runtime-configurable circuits in an approximate RISC-based processor.

5.3. Evaluation

In this section, we evaluate the design-time and runtime variants of our fixed-point and floating-point multipliers. The evaluation begins with the error analysis, which examines the accuracy of our designs, and continues with the experimental results from the synthesis of the circuits, involving comparisons with state-of-the-art approximate designs.

5.3.1. Error Analysis

In the design of approximate circuits, the error inserted in the computations is considered a critical issue, and thus, its impact on the final result is studied using either circuit simulations or rigorous error expressions and models. To evaluate our approximate designs in terms of accuracy, we execute the respective software models emulating the logic-level approximations. Our analysis is based on the calculation of well-established error metrics, as well as new error metrics that are tailored to the designs, such as in the case of floating-point approximations.

Study of AxFXU Accuracy

Firstly, we study the error of AxFXU, namely, our approximate fixed-point multiplier. We consider the error metrics used in the error analysis of Chapter 4. In brief, these metrics are defined as follows:

- RED_{AB}: the relative error distance between the approximate multiplication and the accurate multiplication for a given operand pair A and B.

- MRED: the average of all relative error distances for a given set of operand pairs.

- $PRED_2$: the possibility of having a relative error distance larger than 2%.

We note that the approximation space is large, i.e., there are numerous approximation configurations (combinations of the P and R parameters). Therefore, for each examined configuration, we consider 200K different input pairs, which are uniformly distributed over the multiplication bit-width. The uniform input distribution forms a stressed scenario for AxFXU, because more narrow distributions would lead to biased MRED values, which could be efficiently handled with a limited set of approximation configurations. Nevertheless, we note that AxFXU tunes the approximation degree with two independent parameters, and as a result, it provides the flexibility to handle different input distributions.

Figure 5.4 presents how MRED is affected by the approximation configuration for multiplication bit-width $n = 16, 24, 32$. As shown, the MRED values increase linearly with the approximation degree, i.e., as more approximations are applied. Perforation introduces larger error than rounding, due to the significance of the bits that are pruned (entire partial products versus partial product bits). Moreover, as the bit-width increases, MRED is less affected by the approximations. Specifically for $n = 24, 32$, MRED is up to 0.02% and 0.00005%, respectively. This feature is an advantage of our designs, because it provides the flexibility to perform more aggressive approximations in large-sized multipliers, resulting in increased energy and area gains. Correspondingly, as the multiplier's size increases, a smaller error is introduced to retain the same energy budget and area.

Regarding the significance of the errors, which is examined with the metric $PRED_2$, the large approximation space of AxFXU generates designs with varying $PRED_2$ values. Namely, there are designs with zero or near-zero $PRED_2$, as well as designs with $PRED_2$ of 5% or 10%, which, however, deliver remarkable resource gains. Concluding, the set of approximations of AxFXU can support various design scenarios involving different error and resource constraints.

(a)

(b)

(c)
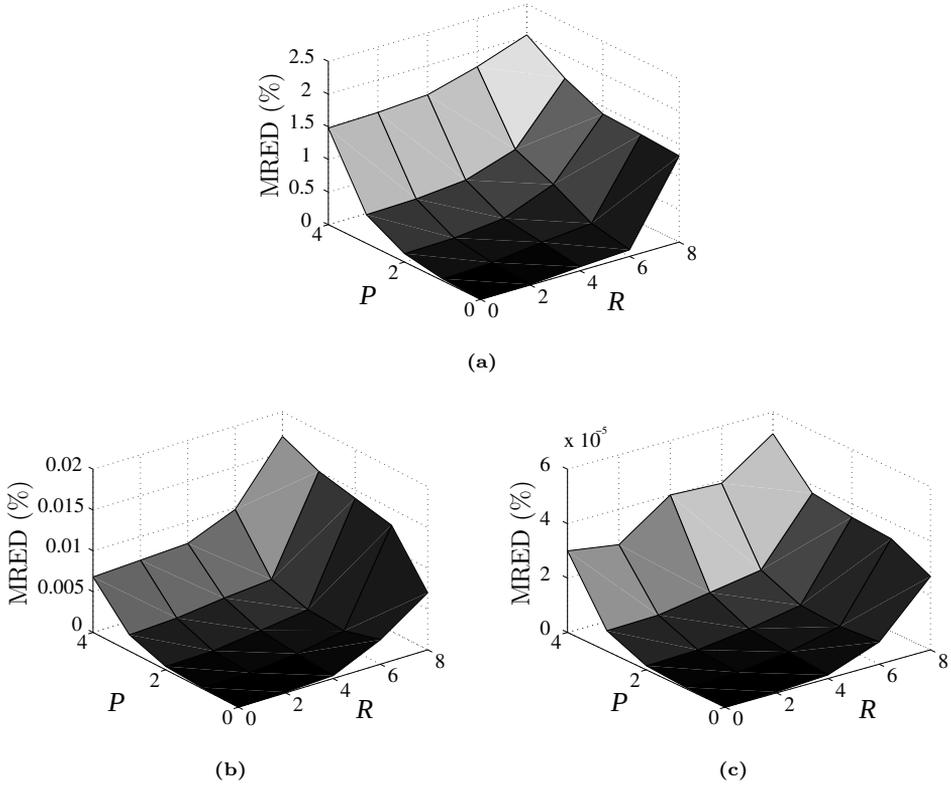
Figure 5.4: MRED variation of AxFXU with respect to the approximation configuration P (perforation) and R (rounding) for multiplication bit-width: **(a)** $n = 16$, **(b)** $n = 24$, and **(c)** $n = 32$.

Study of AxFPU Accuracy

Next, we study the error of AxFPU, namely, our approximate floating-point multiplier. We consider the same error metrics with AxFXU, however, we also employ new error metrics to evaluate the special cases of floating-point arithmetic.

Let $A_f \times B_f$ be the accurate floating-point multiplication of Eq. (5.6) and $A_f \times B_f|_{P,R}$ be the approximate floating-point multiplication of Eq. (5.7). The RED of AxFPU is formed as shown in Eq. (5.8).

$$\text{RED}_{A_f B_f} = \frac{\left| A_f \cdot B_f - A_f \cdot B_f|_{P,R} \right|}{\left| A_f \cdot B_f \right|} = \frac{\left| A \cdot B - A \cdot B|_{P,R} \right|}{\left| A \cdot B \right|} = \text{RED}_{AB} \qquad (5.8)$$

Hence, the RED of AxFPU is equal to the RED of the approximate mantissa multiplier (implemented with AxFXU), i.e., $\text{RED}_{A_f B_f} = \text{RED}_{AB}$. It is important to mention that this formula assumes that possible normalization in the accurate mantissa multiplication, which results in increasing the exponent, also appears in the approximate multiplication. Additionally, we assume that AxFPU does not produce erroneous need for normalization. Regarding MRED and PRED$_2$, they are calculated as defined in Chapter 4.

Considering that in floating-point multiplication overflow or underflow may occur, RED is involved in the calculation of MRED and PRED$_2$ only in three cases, depending on the floating-point data type of the approximate and the respective accurate product:

(i) if both products are normal numbers, where RED is calculated by Eq. (5.8).

(ii) if both products are overflow, where RED is considered 0.

(iii) if both products are underflow, where RED is considered 0.

To evaluate the possibility of having one of the rest combinations, e.g., normal accurate product and underflow approximate product, two more error metrics are introduced:

- PON: the possibility of overflow approximate product and normal accurate product, or vice versa.

- PUN: the possibility of underflow approximate product and normal accurate product, or vice versa.

We note that the appearance of unexpected floating-point data type in the approximate result, which occurs due to the applied approximations, is not examined in prior works of approximate floating-point multipliers. The PON and PUN metrics are calculated by Eq. (5.9)–(5.10).

$$\text{PON} = p(A_f \times B_f|_{P,R} : \text{overflow} \ \& \ A_f \times B_f : \text{normal} \ |$$
$$A_f \times B_f|_{P,R} : \text{normal} \ \& \ A_f \times B_f : \text{overflow}) \tag{5.9}$$

$$\text{PUN} = p(A_f \times B_f|_{P,R} : \text{underflow} \ \& \ A_f \times B_f : \text{normal} \ |$$
$$A_f \times B_f|_{P,R} : \text{normal} \ \& \ A_f \times B_f : \text{underflow}) \tag{5.10}$$

Table 5.2 summarizes all the possible accurate and approximate products and reports which error metric is employed for each combination. According to the analysis of our approximations as well as exhaustive simulations, there are two impossible

Table 5.2: Error metrics for approximate floating-point multipliers.

Acc. Product	Appr. Product	Possible	Error Metrics
normal	normal	✓	MRED, PRED$_2$: w/ RED of Eq. (5.8)
normal	underflow	✓	PUN
normal	overflow	✓	PON
underflow	normal	✓	PUN
underflow	underflow	✓	MRED, PRED$_2$: w/ RED= 0
underflow	overflow	✗	–
overflow	normal	✓	PON
overflow	underflow	✗	–
overflow	overflow	✓	MRED, PRED$_2$: w/ RED= 0

combinations. Specifically, AxFPU cannot produce overflow/underflow in case the accurate result is underflow/overflow.

For the accuracy evaluation, we use half and single floating-point precision, i.e., we employ the 16-bit and 32-bit AxFPU multipliers, labeled as AxFPU16 and AxFPU32, respectively. Similar to the error analysis of AxFXU, we consider a uniform distribution over the normal floating-point numbers (to cover all the data range) and employ 200K different input pairs. Again, narrower input distributions, e.g., only small floating-point numbers, would lead to more biased error values, which could be handled with a limited set of approximation configurations. Figure 5.5 and Figure 5.6 present the variation of the error metrics for AxFPU16 and AxFPU32, respectively. To explore the error scalability, we combine different values for P and R, tailored to the floating-point bit-width, i.e., for the single-precision AxFPU32, we set larger P and R values.

The derived results show that the range of MRED is 0.05%–3.33% for AxFPU16 (see Figure 5.5a), and 0.01%–2.20% for AxFPU32 (see Figure 5.6a), which are typical mean error values for approximate arithmetic units. By examining the impact of rounding for fixed perforation values, we notice that the MRED of AxFPU16 grows sharply when we set $R = 6$, i.e., when the bit-width of each partial product is halved. Specifically, considering perforation values $P < 3$, there is an average ~4× increase of MRED when moving from $R = 4$ to $R = 6$. This threshold is bigger for AxFPU32 (i.e., $R = 20$), considering that the mantissa bit-width is 23 rather than 10. Moreover, as expected, the error is highly affected by perforation, which omits entire partial products. For instance, the MRED of AxFPU16 explodes from 0.81% to 3.25%, when discarding $P = 4$ rather than $P = 3$ partial products. Multipliers with large bit-widths are favored by more aggressive perforation, and as a result, AxFPU32 delivers relatively small error, even though several partial products are perforated,
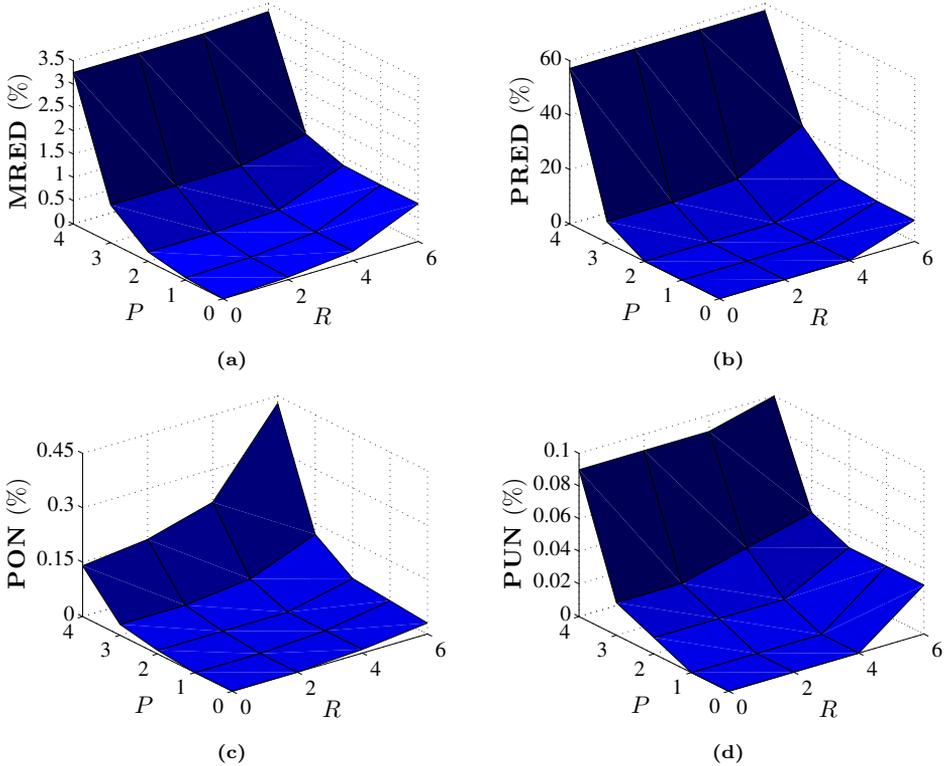
Figure 5.5: Variation of error metrics for half-precision AxFPU with respect to the approximation configuration P (perforation) and R (rounding): **(a)** MRED, **(b)** PRED$_2$, **(c)** PON, **(d)** PUN.

e.g., for $P = 10$ the MRED is 1.63% and raises up to 2.20%, depending on the rounding configuration.

Regarding the PRED$_2$ metric, presented in Figure 5.5b and Figure 5.6b, both multipliers exhibit near-zero values for the smallest examined pairs of configurations, which, however, offer remarkable resource savings. For example, the PRED$_2$ of AxFPU16 is 0.02% in exchange for $P = 2$ perforated partial products and $R = 4$ rounding value. Similarly, the PRED$_2$ of AxFPU32 is 0.01% for $P = 8$ and $R = 18$, i.e., values that deliver significant area reduction considering the number of the remaining partial products and their bit-width. On the other hand, more aggressive approximations produce larger PRED$_2$ values. We note that these values are decreased for input distributions involving bigger floating-point numbers, which are less sensitive to the applied approximations.

Finally, the PON values of our approximate designs lie below 0.43% and 0.26% for AxFPU16 (see Figure 5.5c) and AxFPU32 (see Figure 5.6c), respectively. The
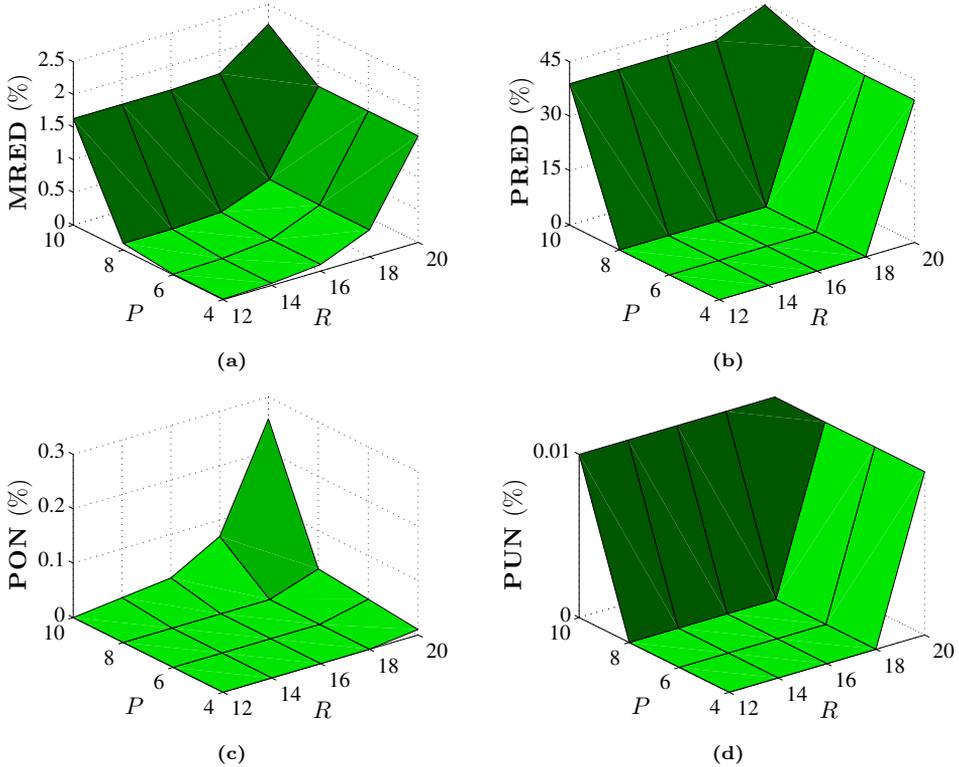
Figure 5.6: Variation of error metrics for single-precision AxFPU with respect to the approximation configuration P (perforation) and R (rounding): **(a)** MRED, **(b)** PRED$_2$, **(c)** PON, **(d)** PUN.

corresponding values for PUN are 0.1% (see Figure 5.5d) and 0.01% (see Figure 5.6d), namely, near-zero possibilities. Moreover, the low-strength approximation configurations completely avoid erroneous type of result (unexpected overflow or underflow) delivering zero PON and PUN values. Overall, only a negligible number of inputs trigger such wrong results, which otherwise could render AxFPU inefficient.

5.3.2. Experimental Results

This section reports experimental results for our approximate fixed- and floating-point multipliers. We employ relevant state-of-the-art designs for comparison, analyze the trade-off between accuracy and resources, and also discuss the benefits of the design-time and runtime variants.

All the designs are implemented in Verilog and synthesized with the Synopsys Design Compiler tool and the TSMC 65-nm standard-cell library. The simulations for the functional verification and the power measurements are performed with Mentor Graphics QuestaSim. The nominal supply voltage (1V) is used in both synthesis and simulation. The critical path delay and the area of the circuits are reported by Synopsys Design Compiler, while the power consumption is measured with Synopsys PrimeTime after performing gate-level simulation. We also evaluate the energy consumption, which is defined as the product of power and delay. For our analysis, we define the gain of the approximate design as the relative resource reduction from the respective accurate design.

Comparative State-of-the-Art Evaluation of Fixed-Point Designs

We implement several configurations of our design-time $(\text{AxFXU}|_{P,R})$ and runtime $(\text{DyFXU}|_{P,R})$ multipliers, as well as the accurate radix-4 multiplier (ACCR4) and the approximate multipliers of $[142, 143, 149, 151, 153, 255, 266]$. In total, our evaluation is performed considering two design scenarios regarding the clock constraint, and involves: (i) comparison among state-of-the-art designs, (ii) comparison between our design-time and runtime designs, and (iii) Pareto analysis considering the error–energy trade-off. Below, we explain the approximation techniques of the literature's designs, and then we present the experimental results.

The perforation of k partial products [255] is labeled as PERFk. We note that the approximation configurations of PERFk are covered by our designs with $P = k$ and $R = 0$, i.e., when applying only perforation and not rounding. R8ABM1 [151] uses the radix-8 encoding to generate the partial products, while calculating an approximate $3A$ product. R8ABM1-15 [151] is the same design, but it also truncates 15 bits of the partial products. The R4ABM1-k and R4ABM2-k multipliers [149] employ an approximate radix-4 encoding for generating the k LSBs of the partial product matrix. R4ABM1-k applies less approximations than R4ABM2-k [149]. RAD2^k [153], which is our design presented in Chapter 4, generates the partial products based on the accurate radix-4 encoding and an approximate high-radix-2^k encoding. TMCk [266] truncates the k LSBs of the partial product matrix and adds correction terms for reducing the error. DRUM6 [142] selects a segment of 6 bits from the multiplication operands, starting from the leading '1' and sets the LSB of the truncated values to '1'. Finally, RoBA [143] rounds the operands to the nearest exponent of two and performs the multiplication in segments using the shift operation.

Table 5.3 presents the experimental results for all the multipliers in 16-bit arithmetic. We note that the experiments are presented in two flavours with respect to the clock frequency of the circuits:

Table 5.3: Experimental results of 16-bit approximate fixed-point multipliers on TSMC 65-nm standard-cell.

Design	Delay (ns)	MIN-Delay		ISO-Delay		Accuracy	
		Area (μm^2)	Energy ($\mu W \cdot ns$)	Area (μm^2)	Energy ($\mu W \cdot ns$)	MRED (%)	PRED$_2$ (%)
ACCR4	0.75	4153	3749	2925	1407	−	−
AxFXU$|_{1,2}$	0.73	3209	2793	2341	1116	0.06	0.32
AxFXU$|_{2,4}$	0.69	2672	2274	1800	919	0.23	1.21
AxFXU$|_{3,4}$	0.65	2320	1830	1529	816	0.53	3.01
AxFXU$|_{3,6}$	0.63	2017	1487	1264	741	0.78	4.81
AxFXU$|_{4,4}$	0.60	1888	1453	1253	716	1.55	10.38
AxFXU$|_{4,6}$	0.59	1513	1178	1025	642	1.76	12.17
DyFXU$|_{1,2}$	0.75	4259	2880	3065	1278	0.06	0.32
DyFXU$|_{2,4}$	0.75	4259	2382	3065	1105	0.23	1.21
DyFXU$|_{3,4}$	0.75	4259	2206	3065	1002	0.53	3.01
DyFXU$|_{3,6}$	0.75	4259	2148	3065	981	0.78	4.81
DyFXU$|_{4,4}$	0.75	4259	2024	3065	896	1.55	10.38
DyFXU$|_{4,6}$	0.75	4259	1953	3065	802	1.76	12.17
PERF1 [255]	0.75	3355	2880	2547	1202	0.03	0.17
PERF2 [255]	0.72	2927	2473	2214	1083	0.13	0.60
PERF3 [255]	0.66	2892	2342	1874	965	0.44	2.38
PERF4 [255]	0.65	2285	1859	1564	858	1.48	9.75
R8ABM1 [151]	0.77	4210	3895	2694	1268	0.15	1.05
R8ABM1-15 [151]	0.74	2926	2499	1686	890	0.61	2.70
R4ABM1-14 [149]	0.74	3958	3460	2634	1257	0.12	0.41
R4ABM1-16 [149]	0.73	3725	3246	2577	1108	0.49	1.49
R4ABM2-14 [149]	0.73	3732	3393	2609	1128	0.24	0.68
R4ABM2-16 [149]	0.72	3467	3101	2547	1079	1.18	2.50
RAD64 [153]	0.72	3489	3238	2257	1057	0.08	0.42
RAD256 [153]	0.69	2769	2410	1911	926	0.28	1.69
RAD1024 [153]	0.65	2624	2224	1663	884	0.93	6.74
TMC8 [266]	0.77	3823	3075	2685	1195	0.11	0.59
TMC15 [266]	0.71	2867	2242	1829	928	1.19	2.75
DRUM6 [142]	1.07	3993	2298	3167	1404	1.47	28.85
RoBA [143]	0.94	4040	2345	2999	1216	2.66	50.07

(i) <u>MIN-Delay</u>: the clock constraint of each circuit is set to its critical path delay (high-performance mode).

(ii) <u>ISO-Delay</u>: the clock constraint of all the circuits is set to the same relaxed

value (low-power mode).

The advantage of AxFXU is its large approximation space, which provides the flexibility to target multiple error levels, while delivering remarkable resource gains. This is justified by the experimental results showing that our approximation technique delivers the best exploitation of the error imposed. Namely, for similar error values, AxFXU outperforms the rest designs in terms of resources (delay, area, energy). Indicatively, we mention that $AxFXU|_{4,4}$ inserts an MRED of 1.55%, while PERF4 and DRUM6 have an MRED of 1.48% and 1.47%, respectively. However, in the MIN-Delay scenario, $AxFXU|_{4,4}$ delivers 17% area and 22% energy gains versus PERF4, and 53% area and 37% energy gains versus DRUM6. Similarly, in the ISO-Delay scenario, the corresponding gains are 20% and 17% versus PERF4, and 60% and 49% versus DRUM6. We notice that in the ISO-Delay scenario, the gains of $AxFXU|_{4,4}$ versus DRUM6 increase due to its large critical path delay (i.e., 1.07ns), which is close to the relaxed clock constraint used.

The radix-4 [149] and radix-8 [151] multipliers exhibit small error values, however, compared to AxFXU, they suffer from increased energy consumption and delay. Moreover, for every $RAD2^k$ multiplier [153], there is an $AxFXU|_{P,R}$ circuit that outperforms it in energy consumption, while delivering a slightly smaller MRED. For instance, in MIN-Delay, $AxFXU|_{2,4}$ offers 6% better energy than RAD256, while it attains smaller error (0.23% versus 0.28%). In general, the $RAD2^k$ family of multipliers features a limited set of approximation configurations, and thus, its error scaling is abrupt, which does not allow to efficiently serve various error constraints. Furthermore, AxFXU is more efficacious than the partial product perforation [255], as the application of rounding delivers significant energy savings, whereas the error slightly increases. Finally, DRUM6 [142] and RoBA [143] deliver good energy savings, however, they exhibit significant errors (considered large for standalone arithmetic circuits). In particular, their large MRED values (1.47% and 2.66%, respectively) are also translated to large $PRED_2$ values (28.85% and 50.07%, respectively). Moreover, the critical paths of the DRUM6 and RoBA circuits are larger compared to the encoding-based circuits, because they implement a different multiplication algorithm.

Next, we evaluate the runtime-configurable variant. We remind that, contrary to AxFXU, where each approximation configuration is implemented as a new circuit, there is only one DyFXU circuit. This circuit implements ACCR4 plus some AND gates to tune the approximation degree at runtime. Namely, $DyFXU|_{P,R}$ denotes the DyFXU circuit that is configured via the control signals to execute the multiplication with perforation P and rounding R. The results for various $DyFXU|_{P,R}$ that are presented in Table 5.3, are reasonable. As expected, DyFXU delivers the same critical path delay and slightly increased area compared to ACCR4. Regarding energy
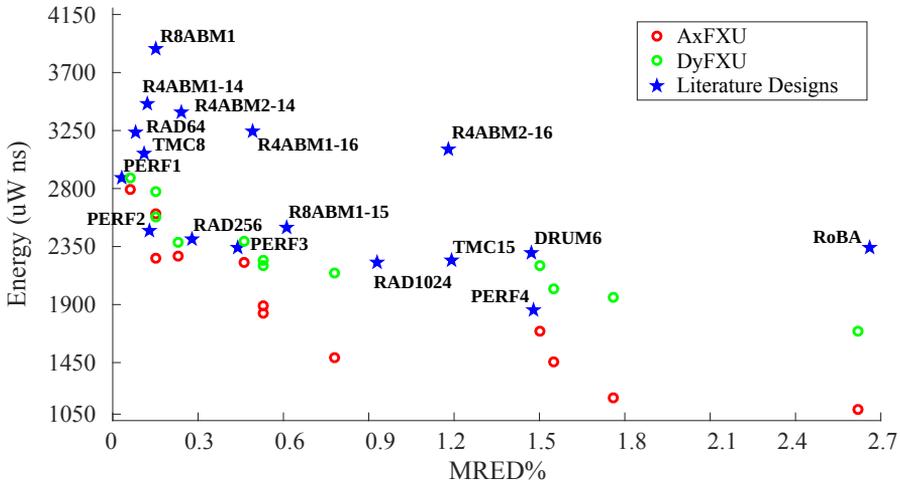
Figure 5.7: Comparative Pareto analysis for approximate fixed-point multipliers considering MRED and energy.

consumption, which has been measured when DyFXU is operating for its P and R configuration, it is improved compared to ACCR4 due to the logic paths that are not activated. Finally, it is interesting to compare $DyFXU|_{P,R}$ with its design-time counterpart, i.e., $AxFXU|_{P,R}$. The energy consumption of $DyFXU|_{P,R}$ is increased, which is reasonable, considering the leakage power of the inactive circuit nodes and the power consumed by the extra AND gates, as well as the control signals that are given as input to the circuit. Indicatively, $DyFXU|_{1,2}$ consumes $1.03\times$ and $1.2\times$ more energy than $AxFXU|_{1,2}$, in the MIN-Delay and ISO-Delay scenarios, respectively. When considering more aggressive approximations, $DyFXU|_{4,6}$ consumes $1.7\times$ and $1.3\times$ more energy than $AxFXU|_{4,6}$, in the MIN-Delay and ISO-Delay scenarios, respectively. Summarizing, in terms of area, DyFXU does not provide gains, however, the overhead is negligible compared to ACCR4. In terms of energy, it provides gains and outperforms several design-time multipliers of the literature, even though these gains are reduced compared to AxFXU. In any case, the negligible area overhead and the reduced, but also significant, energy gains are achieved while supporting dynamic configuration of the approximation.

A comprehensive comparison of all the examined designs is presented in Figure 5.7, where we consider both the MIN-Delay energy and MRED in a scatter plot. We note that this plot includes additional AxFXU and DyFXU designs. As shown, the Pareto front is formed exclusively by different configurations of AxFXU, i.e., our design exhibits the best error–energy trade-off in this comparison involving sev-

eral state-of-the-art designs. $\text{AxFXU}|_{2,2}$ provides significant energy reduction in exchange for a very small error. Similarly, $\text{AxFXU}|_{2,6}$, $\text{AxFXU}|_{3,4}$, and $\text{AxFXU}|_{3,6}$ deliver remarkable energy gains in exchange for MRED values up to 0.78%. Regarding DyFXU, even though it has worse energy consumption than AxFXU, it retains, in almost all cases, the Pareto front with regard to the rest approximate multipliers.

Comparative State-of-the-Art Evaluation of Floating-Point Designs

The evaluation of our approximate floating-point designs, namely, $\text{AxFPU}|_{P,R}$ and $\text{DyFPU}|_{P,R}$, is performed in the following stages: (i) Pareto analysis of the error–resources trade-off, (ii) comparison to state-of-the-art approximate designs, and (iii) efficiency analysis of the runtime variant.

As already discussed, the approximation space of our designs is defined by two independent approximation techniques, and thus, it is large. It is also obvious that it increases as we move on to larger floating-point bit-widths. Therefore, at first, we perform a Pareto analysis involving the resources (delay, area, energy) and MRED, targeting to extract the best approximation configurations and also study the impact of the two approximation techniques in floating-point arithmetic. Again, we consider half- and single-precision floating-point multiplier, i.e., AxFPU16 and AxFPU32. Our Pareto analysis considers approximation configurations that do not produce very large errors. Furthermore, we stress the tool to perform synthesis at the critical path delay of each design, and we also configure the clock period of the gate-level simulation to be equal to the critical path delay.

The scatter plots for the Pareto analysis of AxFPU16 and AxFPU32 are presented in Figure 5.8 and Figure 5.9, respectively. The range of AxFPU16's delay is 0.75ns–0.51ns, while the delay range for AxFPU32 is 0.9ns–0.54ns. Our exploration shows that there is a wide range of area and energy values, along with typical MRED, which constitutes AxFPU as a sustainable solution for scenarios with diverse area/energy constraints. According to the results, rounding has remarkable impact on the accuracy for small perforation configurations, while for bigger P values, MRED almost remains intact. For instance, the MRED of AxFPU16 is stable at ~3.3% for $P = 4$ and different values of R, however, as the rounding aggressiveness increases, significant gains are achieved, e.g., 25% energy reduction for $R = 6$ versus $R = 2$. Regarding perforation, the single increment of the P parameter, considering stable rounding configuration, delivers 5%–12% and 3%–6% energy reduction on average for AxFPU16 and AxFPU32, respectively. Furthermore, as expected, the slower scaling of AxFPU32's error due to its larger mantissa bit-width, allows more aggressive approximations, and as a result, its delay, area and energy are similar to those of
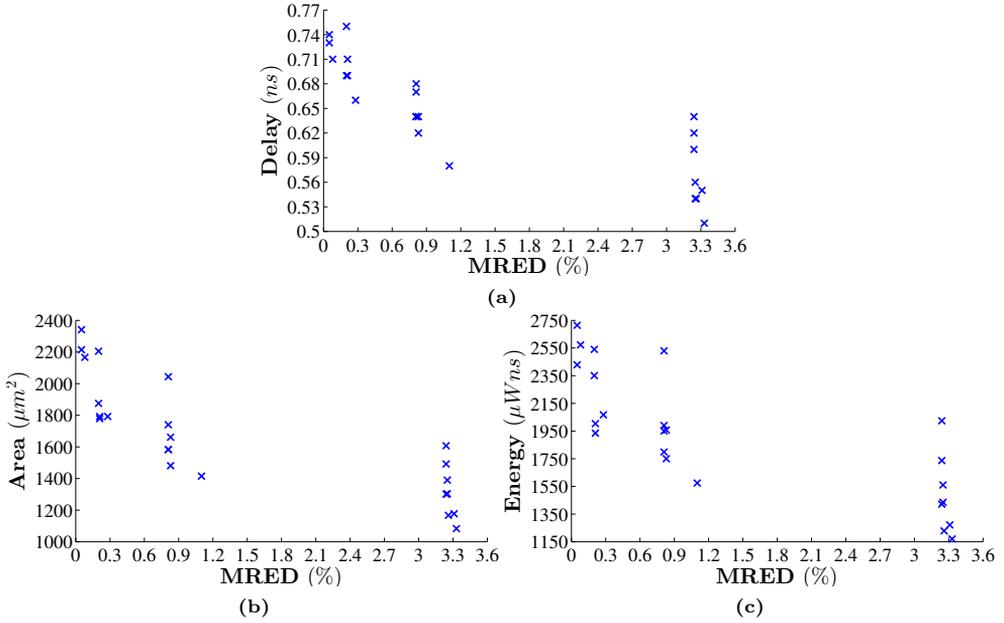
Figure 5.8: Pareto analysis for the half-precision AxFPU|$_{P,R}$ considering: **(a)** Delay and MRED, **(b)** Area and MRED, and **(c)** Energy and MRED.
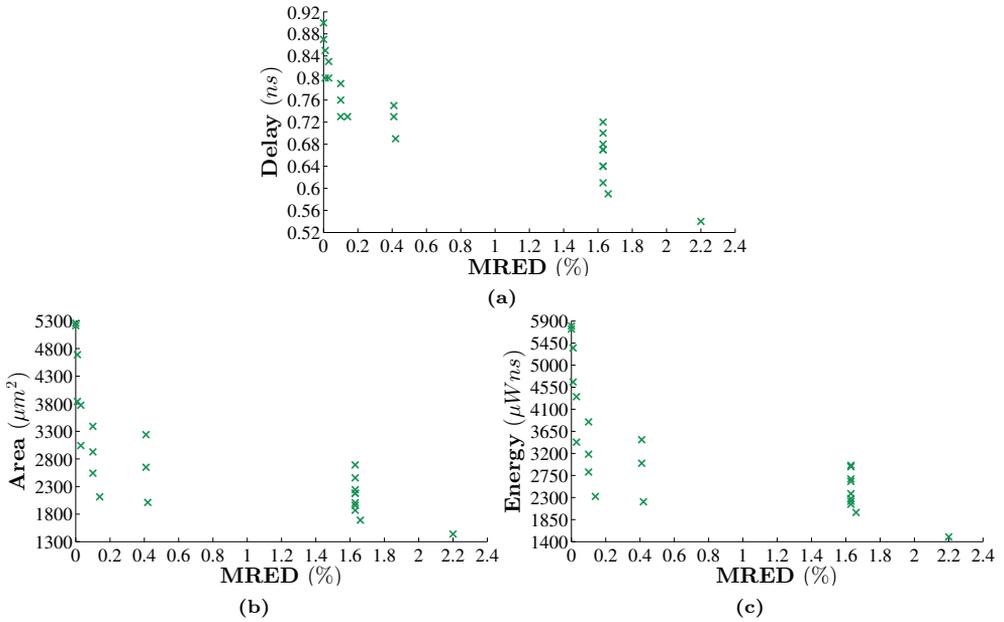
Figure 5.9: Pareto analysis for the single-precision AxFPU|$_{P,R}$ considering: **(a)** Delay and MRED, **(b)** Area and MRED, and **(c)** Energy and MRED.
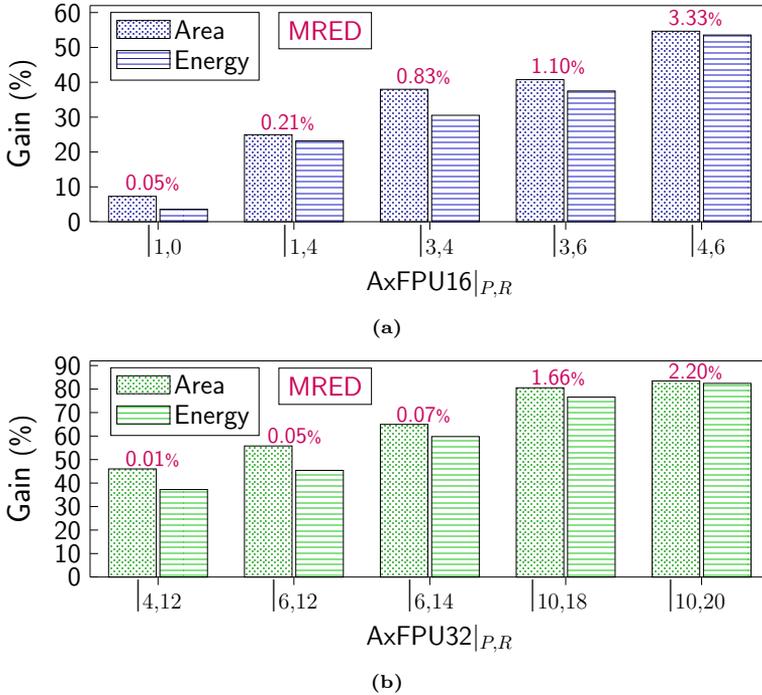
Figure 5.10: Area and energy gains of AxFPU$|_{P,R}$ compared to the accurate floating-point multiplier for **(a)** half precision and **(b)** single precision.

AxFPU16. Namely, our approximate half- and single-precision multipliers impose comparable resource demands.

Following our Pareto analysis, we examine the total resource gains of the Pareto-front AxFPU designs. In Figure 5.10, we present the area/energy gains of AxFPU in comparison with the accurate floating-point multiplier, as well as their MRED values. We select designs with varying MRED values, i.e., from 0.05% to 3.33% for AxFPU16 and from 0.01% to 2.20% for AxFPU32. The half-precision AxFPU family delivers area gains in the range 7.3%–54.6% and energy gains in the range 3.6%–53.5%. The corresponding ranges for the single-precision AxFPU family are 46.1%–83.4% and 37.2%–82.4%. For both precisions, remarkable gains are achieved even for small MRED, e.g., 31% and 59% energy reduction for AxFPU16 and AxFPU32, respectively, in exchange for 0.83% and 0.07% error values. Moreover, in terms of performance, our approximations decrease the critical paths, delivering up to 32.9% and 46% delay gain in AxFPU16 and AxFPU32, respectively. According to the results, AxFPU32 provides increased gains compared to AxFPU16, even for smaller error values. This is justified by its larger bit-width, which offers more room from approximations with-

out significantly affecting the accuracy of the calculations. We note that larger gains can be achieved for both floating-point precisions by increasing the approximation degree. However, more aggressive approximations should be accompanied by a careful error analysis involving the calculation of the metrics presented in the previous section.

Next, we compare AxFPU with the approximate floating-point multipliers of [267] and [268], as well as approximate floating-point multipliers that employ the designs of [149] and [151] to calculate the mantissa product. CFPU [267] does not perform the mantissa multiplication and uses directly one of the two input mantissas as output. On the other hand, RMAC [268] replaces the mantissa multiplication with the addition of the input mantissas. R4ABM [149] and R8ABM [151] are based on the approximate radix-4 and radix-8 encoding, respectively, and they are also included in the evaluation of our fixed-point design. For this comparison, we implement the best approximation configuration. To make a fair comparison, we implement the approximation of these designs in the mantissa multiplication of the accurate floating-point multiplier described in Section 5.2 (the rest components remain intact). The synthesis of AxFPU, CFPU [267], RMAC [268], R4ABM [149] and R8ABM [151] is performed using the same relaxed clock constraint, i.e., the critical path delay of the accurate floating-point multiplier, targeting to study their effectiveness, but also their sustainability under relaxed design margins.

In Table 5.4, we present the resource gains and the accuracy results of the examined floating-point multipliers. We note that we select AxFPU configurations with error values that are comparable to those of the other designs. In terms of accuracy, CFPU [267] is the most inefficient design, as it delivers around $4\times$–$6\times$ larger MRED and PRED_2 of 70%. It is important to mention that both PON and PUN, i.e., the metrics examining unexpected output concerning overflow and underflow, is below 0.5% for all the designs. In case of half precision, RMAC [268] has slightly better accuracy results than AxFPU, however, our design provides increased energy gains. Regarding single precision, AxFPU delivers better MRED and PRED_2 values than RMAC as well as higher energy gains. Overall, the proposed AxFPU design provides larger energy savings than RMAC [268] for similar errors, as well as better energy efficiency than CFPU [267], while exhibiting significantly better accuracy. In terms of critical path delay, CFPU [267] and RMAC [268] deliver larger gains, as they model the mantissa multiplication with less complex operations. Finally, the gains of R4ABM [149] and R8ABM [151] are considered small compared to the other designs. The advantage of these designs is their small error values. Nevertheless, in case there is a constraint of small error values, AxFPU with less aggressive approximation can provide larger resource gains. Indicatively, we mention that the MRED of AxFPU16$|_{3,6}$ is 1.10% and that of AxFPU32$|_{10,18}$ is 1.66%.

Table 5.4: Experimental results of approximate half- and single-precision floating-point multipliers on TSMC 65-nm standard-cell.

	Design	**Delay** (%)[1]	**Area** (%)[1]	**Energy** (%)[1]	**MRED** (%)	**PRED₂** (%)	**PON** (%)	**PUN** (%)
n = 16	AxFPU16$|_{4,6}$	32.9	70.8	67.1	3.33	57.40	0.43	0.10
	CFPU [267]	44.7	71.6	64.6	12.96	70.68	0.46	0.11
	RMAC [268]	42.2	71.1	63.2	3.16	49.42	0.20	0
	R4ABM [149]	7.8	43.3	39.1	2.12	20.61	0.09	0.01
	R8ABM [151]	10.1	47.6	41.3	1.56	7.14	0	0
n = 32	AxFPU32$|_{10,20}$	46.0	89.9	87.4	2.20	44.86	0.26	0.01
	CFPU [267]	53.9	86.5	79.7	12.80	70.63	0.05	0.02
	RMAC [268]	50.6	83.9	78.3	2.92	49.73	0.02	0
	R4ABM [149]	9.1	53.3	50.7	1.44	16.32	0.02	0
	R8ABM [151]	14.3	59.5	51.1	0.99	5.46	0	0

[1] Refers to % delay/area/energy gains (relative reduction) in comparison with the accurate design.

Finally, we assess the hardware efficiency of DyFPU, which is the runtime-configurable variant of AxFPU. We remind that DyFPU$|_{P,R}$ denotes the single DyFPU circuit that is configured via the control signals to execute the multiplication with perforation P and rounding R. In contrast, AxFPU$|_{P,R}$ is configured at design-time, i.e., a new circuit is implemented for each approximation configuration. Table 5.5 presents the comparison of the two variants by examining the gains compared the accurate design. Both DyFPU and AxFPU are synthesized and simulated under tight clock constraints, i.e., their critical path delays. The outcomes from this comparison are similar to those derived from the comparison of the fixed-point designs (AxFXU and DyFXU). More specifically, as expected, DyFPU imposes an area increase of 4.3% and 2.1% for half and single precision, respectively. Regarding energy consumption, DyFPU is worse than its design-time counterpart, delivering ~1.4× and ~1.6× less gain for remarkable approximation in half and single precision, respectively. However, it is still more energy-efficient than the accurate design, providing energy gains up to 49.7%, while also allowing to dynamically tune the approximation degree by seamlessly changing the values of the P and R parameters.

In comparison with the runtime-configurable variants of the CFPU [267] and RMAC [268] designs, DyFPU is not designed to automatically configure its operation to either accurate or approximate mode based on the mantissa inputs or the approximate mantissa product. However, it prevails over these two methods in the following factors: (i) it supports multiple approximation configurations instead of one fixed configuration, (ii) it does not insert extra delays (e.g., to check the inputs/outputs or even re-calculate the result using the accurate circuit), (iii) it exposes the tuning

Table 5.5: Comparison of the design-time and runtime approximate floating-point multipliers.

Design-Time Config.		Area (%)[1]	Energy (%)[1]	Runtime Config.		Area (%)[1]	Energy (%)[1]
n=16	AxFPU16$\|_{1,0}$	7.3	3.6	*n=16*	DyFPU16$\|_{1,0}$	−4.3	1.1
	AxFPU16$\|_{3,4}$	38.0	30.5		DyFPU16$\|_{3,4}$	−4.3	22.5
	AxFPU16$\|_{4,6}$	54.6	53.5		DyFPU16$\|_{4,6}$	−4.3	37.2
n=32	AxFPU32$\|_{4,12}$	46.0	37.2	*n=32*	DyFPU32$\|_{4,12}$	−2.1	23.1
	AxFPU32$\|_{6,14}$	64.9	59.8		DyFPU32$\|_{6,14}$	−2.1	34.1
	AxFPU32$\|_{10,18}$	80.5	76.6		DyFPU32$\|_{10,18}$	−2.1	49.7

[1] Refers to % area/energy gains (relative reduction) in comparison with the accurate design.

logic (AND gates) to the system, allowing a high-level policy or framework to configure the approximation degree with respect to the application's energy and accuracy constraints.

5.4. Conclusion

In this chapter, we examined an attractive aspect of Approximate Computing, i.e., the dynamic configuration of the approximation degree. This feature is extremely important for modern embedded systems and circuits, which need to adapt the accuracy of the calculations at runtime, depending on the type of the application, the input dataset of the application, and the given energy constraints. Towards this direction, we targeted the runtime-configurable arithmetic circuits and designed energy-efficient multipliers that can change approximation configuration at runtime. In particular, we employed two orthogonal approximation techniques for calculating the partial products and generated a large approximation space that serves different scenarios in terms of accuracy and energy budget. The technique of partial product perforation omits the generation of least significant products, while partial product rounding efficiently decreases the bit-width of the remaining ones. These two approximation techniques facilitate dynamic configurability when combined with the radix-4 operand encoding, as their application is associated with the input operand bits. Therefore, we added negligible logic overhead in the input of the accurate design, i.e., $2n$ AND gates, where n is the operand bit-width, and enabled dynamic configuration via input signals. We designed both integer/fixed-point and floating-point circuits integrating our approximation techniques and runtime functionalities. In terms of accuracy, we evaluated our designs by performing an in-depth exploration involving diverse error metrics for various approximation configurations, as well as new error metrics that are tailored to floating-point arithmetic. The results show that the proposed solution exhibits dense

error scaling, i.e., it provides multiple configurations with mean error values across the entire range that is considered acceptable (up to 2%). In terms of circuit resources, our design-time variants outperform all the examined state-of-the-art multipliers in both fixed- and floating-point arithmetic. For similar error values, the fixed-point AxFXU designs deliver gains up to 60% and 49% in area and energy, respectively, when operating at the same clock frequency. Similarly, the floating-point AxFPU designs exhibit either better accuracy for the same resource gains or larger resource gains for comparable error values. The respective runtime variants provide negligible area overhead and smaller energy gains, however, they still deliver remarkable gains versus the accurate multiplier and other state-of-the-art design-time multipliers, while allowing to seamlessly change approximation configuration at the runtime. In fixed-point arithmetic, DyFXU consumes only up to $1.2\times$ and $1.7\times$ more energy than AxFXU for low-strength and more aggressive approximation, respectively. Similar behaviour is observed in our floating-point designs, where DyFPU provides ~$1.4\times$ and ~$1.6\times$ less energy gains in half and single precision, respectively.

Chapter 6

Cooperative Approximation:
Combination of Arithmetic Encodings

The rapid growth of error-resilient applications with different accuracy constraints and computational demands creates the need for approximate circuits and systems that can support multiple approximations. Towards this direction, the designers tend to select approximation techniques that do not generate a single approximate design, but they can provide various design variants with different approximation configuration, and thus, different accuracy and resource gains. In this context, we combine arithmetic approximation techniques to create a very large approximation space for the design of multiplication circuits. Besides providing numerous approximation configurations, we also target to identify the most-efficient design solution among the well-established arithmetic approximation techniques. Our pool of techniques consists of high-radix encoding, partial product perforation and partial product rounding. The feasible combinations of these techniques generate 5 new design families of approximate multipliers. Our extensive design space exploration shows that the combination of approximation techniques, called as "cooperative approximation", results in slow error scaling with numerous configurations in the typical acceptable range 0%–2%. This feature creates increased design flexibility, allowing to efficiently handle workloads and applications with different constraints. The experimental evaluation is based on a comparative state-of-the-art Pareto analysis involving the Pareto-front designs RAD and AxFXU, namely all the approximate designs presented in the Dissertation, as well as other designs of the literature. The results show that the Pareto front is formed exclusively by designs with cooperative approximation. In particular, the ROUP family of multipliers, which applies perforation and a new type of rounding, constitutes the most energy-efficient design alternative, improving the Pareto front by up to 1.5×–2×. Furthermore, the new Pareto front has increased resolution (i.e., more design configurations) due to the large approximation space. Finally, in comparison with state-of-the-art designs, the ROUP family provides energy gains up to 63% for the same error constraint.
*This chapter is based on our **publication** in [147].*

6.1. Introduction

One of the goals of Approximate Computing is to provide design solutions that can handle various accuracy and resource constraints. This feature is extremely important, considering that the error-tolerant applications [29, 240] impose non-identical demands regarding the quality of results and exhibit different error propagation within their calculations. Approximation techniques are applied at software, architecture and hardware layers [16, 17, 19]. To expand the approximation space, the designers are examining the simultaneous application of more than one approximation technique, either from different or the same layer. Although vertical cross-layer approximation techniques have recently emerged [18], the full potential of horizontal approximation, i.e., within the same layer of design abstraction, still remains an open issue for further exploration. In this chapter, we explore, for the first time, the efficiency of combining arithmetic approximation techniques in the design of energy-efficient multiplication circuits. We focus on combining approximation techniques at the arithmetic level, as it inherently affects both the dynamic and static power consumption of the underlying circuits. Moreover, the implementation delivered at this level can be straightforwardly adopted in a vertical cross-layer design approach.

The work presented in this chapter is inspired from the promising results of the AxFXU design (see Chapter 5), which combines two orthogonal approximation techniques. The goal of AxFXU is twofold: (i) to provide multiple approximation configurations and enrich the available accuracy options at runtime, and (ii) to apply an approximation scheme that facilitates the integration of dynamic configuration capabilities. In this chapter, we perform an extensive design space exploration on combining arithmetic approximation techniques. Contrary to Chapter 5, the goal of this chapter is the expansion of the approximation space with diverse sets of approximation configurations and the in-depth exploration of the design space to identify the most efficient solution. We use the term *cooperative approximation* to state that two techniques are applied in the design of the approximate circuit. As a result of our exploration on cooperative approximation, we propose 5 families of approximate multipliers, which combine some of the techniques that have been already examined, i.e., high-radix encoding, partial product perforation, and partial product rounding.

Figure 6.1 summarizes the motivation behind our work by demonstrating the benefits of combining approximation techniques rather than using a single one. This plot illustrates the scatter points of partial product perforation, partial product rounding, and their combination. The red points label the single application of perforation and rounding, while the blue points label the cooperative perforation & rounding. We note that for the cooperative approach, we employ two perforation configurations
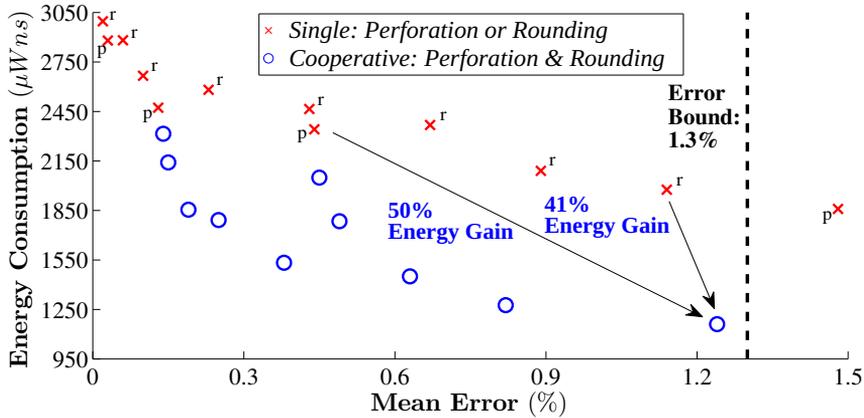
Figure 6.1: Motivation plot for cooperative arithmetic approximation.

and combine each one with five rounding configurations. As shown, the cooperative approximation approach provides three advantages versus the single approximation approach (either perforation or rounding): (i) it forms the Pareto front, thus, it is considered a better design solution, (ii) it provides increased Pareto front resolution due to the larger design/approximation space, and (iii) it delivers a more energy-efficient circuit for a given error constraint, e.g., for mean error of 1.3%, it gains 50% and 41% in energy versus perforation and rounding, respectively.

The **contribution** of this chapter is summarized as follows:

(i) We highlight the efficiency of integrating more than one approximation technique in the design of approximate circuits.

(ii) We propose 5 new families of approximate multipliers that feature a very large design space with differing approximation configurations that can efficiently handle diverse error constraints.

(iii) We reform the state-of-the-art energy/area–error Pareto front by improving it and also increasing its resolution.

(iv) We provide a more energy-efficient design solution than state-of-the-art designs for a given error constraint.

The remainder of this chapter is organized as follows. Section 6.2 classifies the arithmetic approximation techniques and reports representative state-of-the-art works. Section 6.3 introduces our approximation techniques and presents how they can be combined in the design of multipliers with cooperative approximation. In Section 6.4, we assess the cooperative approximation techniques in terms of accuracy and circuit efficiency by reporting a comparative state-of-the-art Pareto evaluation, which

includes the previous Pareto-front designs. Finally, Section 6.5 draws the conclusions.

6.2. Classification of Arithmetic Approximation Techniques

Arithmetic approximations in circuits have been extensively studied in the past, as they deliver significant energy gains at the application/system level. Interestingly, we focus on techniques that apply approximations based on the input operands and the bit significance. i.e., they can be included in the wider range of encoding-based techniques, and we attempt to categorize them into the following classes: (i) pruning, (ii) radix encoding, (iii) rounding, and (iv) dynamic scaling. More specifically, we provide a closer look at each class by describing their basic approximation concept and presenting representative state-of-the-art works for multiplication circuits.

Pruning: This class of techniques aims to reduce the logic by discarding either bits, terms, or nodes of any arithmetic circuit. A well-established pruning technique for approximate multipliers is the partial product perforation [255]. In this technique, partial products are omitted, and thus, simpler partial product matrices are generated. One downside of this technique is that the error is exponentially increased as more partial products are excluded. The literature also provides more general pruning approaches. In [174], the authors propose a methodology and tool to automatically trade accuracy for area, power and delay gains. Their method applies gate-level pruning to any combinational circuit, and it is quite effective especially on arithmetic circuits that have a notion of bit significance. In the same context, probabilistic pruning is proposed in [269], where a greedy approach is used to generate approximate circuits.

Radix Encoding: The techniques that are based on radix encodings, approximately encode the input operands to reduce the complexity of the calculations. Specifically for multipliers, approximate radix encodings result in less partial product bits or even reductions in the total number of partial products. Liu *et al.* [149] designed approximate radix-4 encoders by transforming the Karnaugh map of the accurate encoding. Moving to radix-8 multipliers, Jiang *et al.* [151] employed an approximate adder for producing the partial product $\pm 3A$. In Chapter 4, we presented our approximate high-radix encodings [153]. In particular, we issue the increased complexity of the conventional accurate high-radix encodings by mapping all the radix values to their nearest of the 4 largest powers of two. This approximation provides simpler operand

encoders and partial product generators. In the same context, there are various radix-based approximate multipliers in the literature, e.g., using radix-4 [270], radix-8 [152] and radix-256 [271] encodings.

Rounding: Both truncation and rounding techniques have been examined in arithmetic circuits. Truncation simply discards least significant bits to reduce the bit-width. In contrast, rounding performs mapping to a smaller bit-width, while trying to compensate for the error, e.g., by inserting correction terms. These techniques are applied to either the input operands, intermediate results or the final result. Several approximate multipliers are designed by applying rounding/truncation along with error compensation techniques. A representative work is the truncated multiplier proposed in [266], where the least significant bits of the partial products are vertically discarded and a constant correction term is added to reduce the total error. In [272], the authors propose an error compensation method for the radix-4 multiplier that outputs products with the same bit-width as the input operands. Zhang *et al.* [273] design an approximate multiplier by dividing the partial product matrix into two segments: the main segment, which is accurately accumulated, and the truncated segment, which is further partitioned into two parts. The least significant part of the truncated segment is calculated through a probabilistic approach. In Chapter 5, we introduced partial product rounding, which rounds the partial products to a smaller bit-width based on low-level optimizations [144]. As we showed, the rounding of the partial products is equivalent to rounding one of the input operands.

Dynamic Scaling: In this class, the approximation degree is determined with respect to the input operands. In [254], the authors statically capture and multiply bit segments, either starting from the most significant bit or ending at the least significant bit. A limitation of this technique is the difficulty in scaling to higher bit-widths, and thus, its benefits are reduced as the input bit-width grows. Based on the varying bit significance, Hashemi *et al.* [142] proposed a more fine-grained input segmentation, using leading one detector circuits to locate the most significant '1' in each operand. However, this dynamic segmentation implies extra circuits for the signed multiplications, increasing the total circuit area.

6.3. Design of Multipliers with Cooperative Approximation

In this section, we present the proposed combinations of arithmetic approximation techniques. Firstly, we make a brief introduction in the examined techniques, targeting to explore the feasibility of all the possible combinations, and then, we discuss the technical details of each combination.

6.3.1. The Pool of Arithmetic Approximation Techniques

We consider the three techniques used in Chapter 4 and Chapter 5, i.e., high-radix encoding, partial product perforation, and partial product rounding. As baseline, we consider the accurate radix-4 multiplier for the n-bit operands A and B. The partial product matrix of the 16-bit radix-4 multiplier is illustrated in Figure 6.2a.

High-Radix Encoding: This technique is configured with the parameter k, which is an even number belonging in the interval $[4, n-2]$. The multiplicand B is approximately encoded with respect to k: the $n-k+1$ Most Significant Bits (MSBs) are encoded with the accurate radix-4 encoding, while the k Least Significant Bits (LSBs) are encoded with the approximate high-radix-2^k encoding. This hybrid encoding is then used to generate $(n-k)/2$ accurate and 1 approximate partial product. Figure 6.2b illustrates the partial product matrix of the 16-bit multiplier that is based on high-radix encoding with $k = 6$. The least significant partial product (black triangles) is approximately generated from the radix-64 encoding, substituting the three least significant partial products of the accurate design (see Figure 6.2a).

Partial Product Perforation: This technique is configured with the parameter P, which is an integer number belonging in the interval $[0, n/2 - 1)$. It is applied by not generating the P least significant partial products. In practice, this technique is equivalent to encoding B to $B_P = \langle b_{n-1}b_{n-2}\cdots b_{2P-1}\rangle$. Figure 6.2c illustrates the partial product matrix of the 16-bit multiplier that is based on partial product perforation with $P = 2$. As shown, the matrix has two less partial products than the accurate design (see Figure 6.2a).

Partial Product Rounding: This technique is configured with the parameter R, which is an integer number belonging in the interval $[0, n-1)$. It is applied by rounding the partial products to their R-th bit. In practice, rounding is equivalent to reducing the bit-width of A, e.g., the rounding technique of Chapter 5 is equivalent to encoding A to $A_R = \langle a_{n-1}a_{n-2}\cdots a_R\rangle + a_{R-1}$. We note that in this chapter, we consider additional methods for rounding, which reduce the partial products to a tailored bit-width based on their significance. In contrast, the rounding of Chapter 5 rounds all the partial products to the same bit-width. We call this variant "asymmetric" rounding and use the label "symmetric" for the one presented in Chapter 5. Figure 6.2d illustrates the partial product matrix of the 16-bit multiplier that is based on asymmetric rounding. As shown, the four least significant partial products are rounded to a different bit-width. The red squares denote the correction terms that are added for error compensation.

The next step is to explore all the possible combinations of our approximation techniques. Table 6.1 summarizes the combinations when considering one encoding per
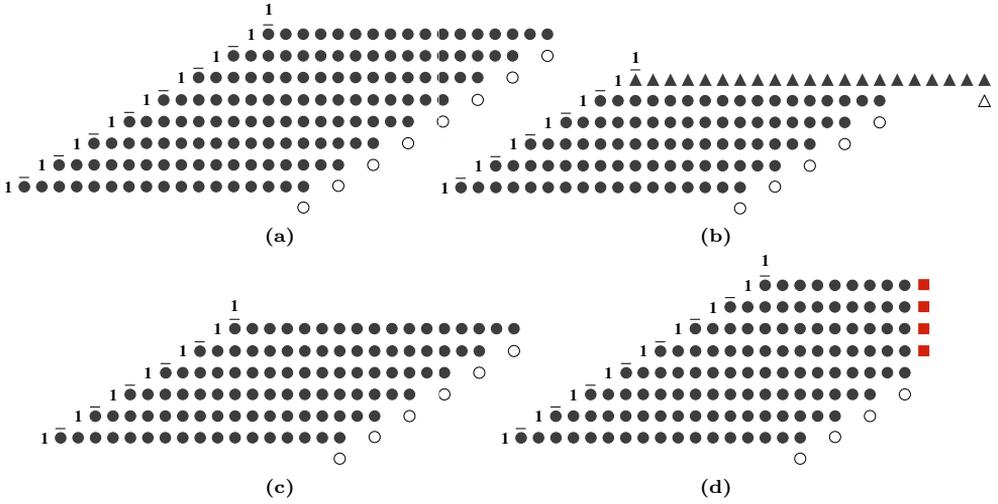
Figure 6.2: Partial product matrices of 16-bit multipliers based on: **(a)** accurate radix-4 encoding (baseline), **(b)** high-radix encoding, **(c)** partial product perforation, **(d)** partial product rounding.

Table 6.1: Combinations of arithmetic approximation techniques.

1st Operand	2nd Operand	Feasibility	Combination Label
high-radix	high-radix	✓	DRAD
high-radix	perforation	✓	equivalent to perforation
high-radix	rounding	✓	RADR
rounding	rounding	✗	–
rounding	perforation	✓	ROUP
perforation	perforation	✗	–

Extra Combinations	Feasibility	Combination Label
high-radix & high-radix + perforation	✓	DRADP
high-radix & rounding + perforation	✓	equivalent to ROUP
rounding & perforation + perforation	✓	equivalent to ROUP

operand. As shown, there are two combinations, i.e., rounding & rounding and perforation & perforation, which are not feasible. The combination high-radix & perforation is equivalent to single perforation, as the high-radix approximate partial product is perforated, and thus, it can be safely excluded for further examination. Additionally, we examine the application of perforation to all the feasible combinations (bottom part of the table). The only extra meaningful combination is perforation with the double high-radix encoding, as the rest ones are equivalent to rounding & perforation.

6.3.2. Combining Arithmetic Approximation Techniques

Next, we analyze the selected combinations of arithmetic approximations. More details about the high-radix encoding and the perforation/rounding are provided in Chapter 4 and Chapter 5, respectively.

DRAD: High-Radix & High-Radix

Let m and k be the configuration parameters for encoding A and B, respectively. The two operands are transformed using the high-radix-2^m and high-radix-2^k encodings as shown in Eq. (6.1)–(6.6). We note that B is encoded exactly as shown in Chapter 4, i.e., with the hybrid radix-4–radix-2^k encoding, while for A, we divide it into two words and encode only the least significant with radix-2^m. To create the high-radix digit in A, we add and subtract the term $2^{m-1}a_{m-1}$ in its representation.

$$A = -2^{n-1}a_{n-1} + \sum_{i=0}^{n-2} 2^i a_i = A_1 + x_0^{R2^m} \tag{6.1}$$

$$\text{where} \quad A_1 = -2^{n-1}a_{n-1} + \sum_{i=m-1}^{n-2} 2^i a_i + 2^{m-1}a_{m-1} \tag{6.2}$$

$$x_0^{R2^m} = -2^{m-1}a_{m-1} + 2^{m-2}a_{m-2} + \cdots + a_0 \tag{6.3}$$

$$B = -2^{n-1}b_{n-1} + \sum_{i=0}^{n-2} 2^i b_i = B_1 + y_0^{R2^k} \tag{6.4}$$

$$\text{where} \quad B_1 = \sum_{j=k/2}^{n/2-1} 4^j y_j^{R4}, \quad y_j^{R4} = -2b_{2j+1} + b_{2j} + b_{2j-1} \tag{6.5}$$

$$y_0^{R2^k} = -2^{k-1}b_{k-1} + 2^{k-2}b_{k-2} + \cdots + b_0 \tag{6.6}$$

Following the approximation approach of the high-radix encoding, we approximate the high-radix digits of A and B, i.e., we use $\hat{x}_0^{R2^m} \in \{0, \pm2^{m-4}, \pm2^{m-3}, \pm2^{m-2}, \pm2^{m-1}\}$ and $\hat{y}_0^{R2^k} \in \{0, \pm2^{k-4}, \pm2^{k-3}, \pm2^{k-2}, \pm2^{k-1}\}$, respectively. Based on these operand encodings, the multiplication of $\text{DRAD}|_{k,m}$ is calculated by Eq. (6.7).

$$\text{DRAD}|_{k,m} = A_1 \cdot B_1 + B_1 \cdot \hat{x}_0^{R2^m} + A \cdot \hat{y}_0^{R2^k} \tag{6.7}$$

DRADP: High-Radix & High-Radix + Perforation

When combining the double high-radix encoding with perforation, we create the $\text{DRADP}|_{k,m}$ multiplier, which perforates the least significant partial product, as shown in Eq. (6.8).

$$\text{DRADP}|_{k,m} = A_1 \cdot B_1 + B_1 \cdot \hat{x}_0^{R2^m} \tag{6.8}$$

RADR: High-Radix & Rounding

For this combination, we consider the approximate high-radix encoding of B, i.e., $B_1 + \hat{y}_0^{R2^k}$, while A is implicitly encoded via the rounding of the partial products. To apply asymmetric rounding, we truncate the R least significant columns of the partial product sub-matrix that is generated by $A \cdot B_1$. As a result, the multiplication of $\text{RADR}|_{k,R}$ is calculated by Eq. (6.9).

$$\text{RADR}|_{k,R} = A \cdot B_1|_R + A \cdot \hat{y}_0^{R2^k} \tag{6.9}$$

To compensate for the truncation error, we insert a correction term in the place of the truncated MSB of each partial product. This term is different for each partial product and it is affected by the radix-4 encoding, i.e., it is equal to $one_j + two_j$. We remind that the signals one_j and two_j are activated if the j-th partial product is equal to $\pm 1A$ or $\pm 2A$, respectively (see the analysis of Chapter 4). To further improve our rounding, we attach a constant '1' in the position that the sub-matrix of $A \cdot B_1$ is truncated.

ROUP: Rounding & Perforation

The symmetric rounding & perforation combination has been already examined in Chapter 5, where $\text{AxFXU}|_{P,R}$ is presented. Therefore, in this chapter, we combine asymmetric rounding and perforation, and more specifically, we employ two different variants of asymmetric rounding.

The first combination is labeled $\text{ROUP1}|_{P,R}$ and uses the asymmetric rounding of the RADR design. In particular, we perforate the P least significant products, and then we apply rounding by truncating the R least significant columns of the partial product matrix and inserting the correction terms. The multiplication of ROUP1 is calculated by Eq. (6.10).

$$\text{ROUP1}|_{P,R} = \left.\sum_{j=P}^{n/2-1} 4^j \tilde{P}P_j \right|_R = \left.\sum_{j=P}^{n/2-1} 4^j A \cdot y_j^{R4} \right|_R \tag{6.10}$$

The second combination is labeled ROUP2$|_{P,R}$ and converts the rounding of AxFXU$|_{P,R}$ to asymmetric. The multiplication of ROUP2 is performed by accumulating the non-perforated, rounded partial products as shown in Eq. (6.11)–(6.12).

$$\text{ROUP2}|_{P,R} = \sum_{j=P}^{n/2-1} 4^j \tilde{P}P_j = \sum_{j=P}^{n/2-1} 4^j A_{R_j} \cdot y_j^{R4} \tag{6.11}$$

$$\text{where} \quad A_{R_j} = \langle a_{n-1}a_{n-2}\cdots a_{R_j}\rangle_{\text{2's}} + a_{R_j-1} \tag{6.12}$$

Compared to the respective expressions of AxFXU, which are given in Eq. (5.1)–(5.4), the only difference is that there are j A_R words (one per partial product) instead of one. The rest rounding optimizations are the same, resulting in integrating the a_{R_j-1} bits with a XOR gate in the correction terms, i.e., $(sign_j \oplus a_{R_j-1}) \cdot (one_j + two_j)$. We note that the notation ROUP2$|_{P,R}$ considers $R = R_P$, i.e., it shows the rounding configuration of the first non-perforated partial product.

6.3.3. Overview of Cooperative Approximation Techniques

Figure 6.3 illustrates the 16-bit partial product matrix of each combination with specific configuration. As shown, depending on the combination, we achieve a different structure compared to the accurate matrix of Figure 6.2a. Even though it is possible to achieve comparable horizontal and vertical matrix reduction with all combinations, we note that each combination: (i) requires different logic for implementing the operand encoding and/or partial product generation, (i) imposes different overheads in the partial product accumulation, which depends on the bit arrangement within the matrix (e.g., for carry propagation penalties), and (iii) has different impact on the accuracy of the results. For example, ROUP1$|_{3,8}$ (see Figure 6.3d) and ROUP2$|_{3,10}$ (see Figure 6.3e) may deliver similar matrices, however, as we show in the evaluation in the next section, their results and total efficiency are different.

In the matrix of DRAD, the triangles denote the product $A \cdot \hat{y}_0^{R2^k}$ and the rectangles denote the product $B_1 \cdot \hat{x}_0^{R2^m}$. The circles are the products from $A_1 \cdot B_1$. The difference of DRADP is that it perforates the product $A \cdot \hat{y}_0^{R2^k}$. Regarding RADR, the triangles denote the product $A \cdot \hat{y}_0^{R2^k}$, while the circles are the products from $A \cdot B_1|_R$. The red rectangles are the correction terms that are inserted for error compensation. The circles in the ROUP1 matrix are the non-perforated, rounded partial products. Like in RADR, the red rectangles are the rounding correction terms. Finally, the matrix of ROUP2 is similar to that of ROUP1, however, in this combination, the correction terms (gray rectangles) are different and the matrix is not truncated vertically.
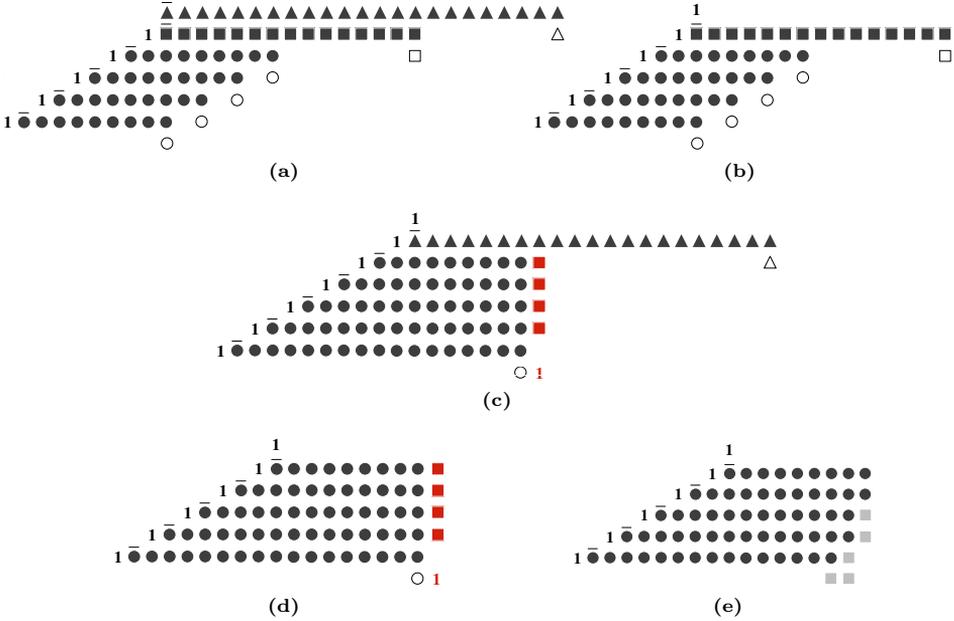
Figure 6.3: Partial product matrices of 16-bit multipliers based on cooperative approximation: **(a)** high-radix & high-radix (DRAD$|_{8,8}$), **(b)** high-radix & high-radix + perforation (DRADP$|_{8,8}$), **(c)** high-radix & asymmetric rounding v.1 (RADR$|_{6,8}$), **(d)** perforation & asymmetric rounding v.1 (ROUP1$|_{3,8}$), and **(e)** perforation & asymmetric rounding v.2 (ROUP2$|_{3,10}$).

6.4. Evaluation

In this section, we evaluate the application of cooperative approximation in multiplication circuits. Like in Chapter 4 and Chapter 5, we begin with the error analysis of each combination, and then we report comparative experimental results involving state-of-the-art works, as well as all the approximate designs proposed in the Dissertation.

6.4.1. Error Analysis

For the error analysis, we rely again on the MRED metric (see the respective analysis of Chapter 4 and Chapter 5 for more details). In brief, MRED is the average of the relative errors for a given set of operand pairs. All the proposed multipliers feature a very large design space, as they are configured by two independent parameters. Therefore,
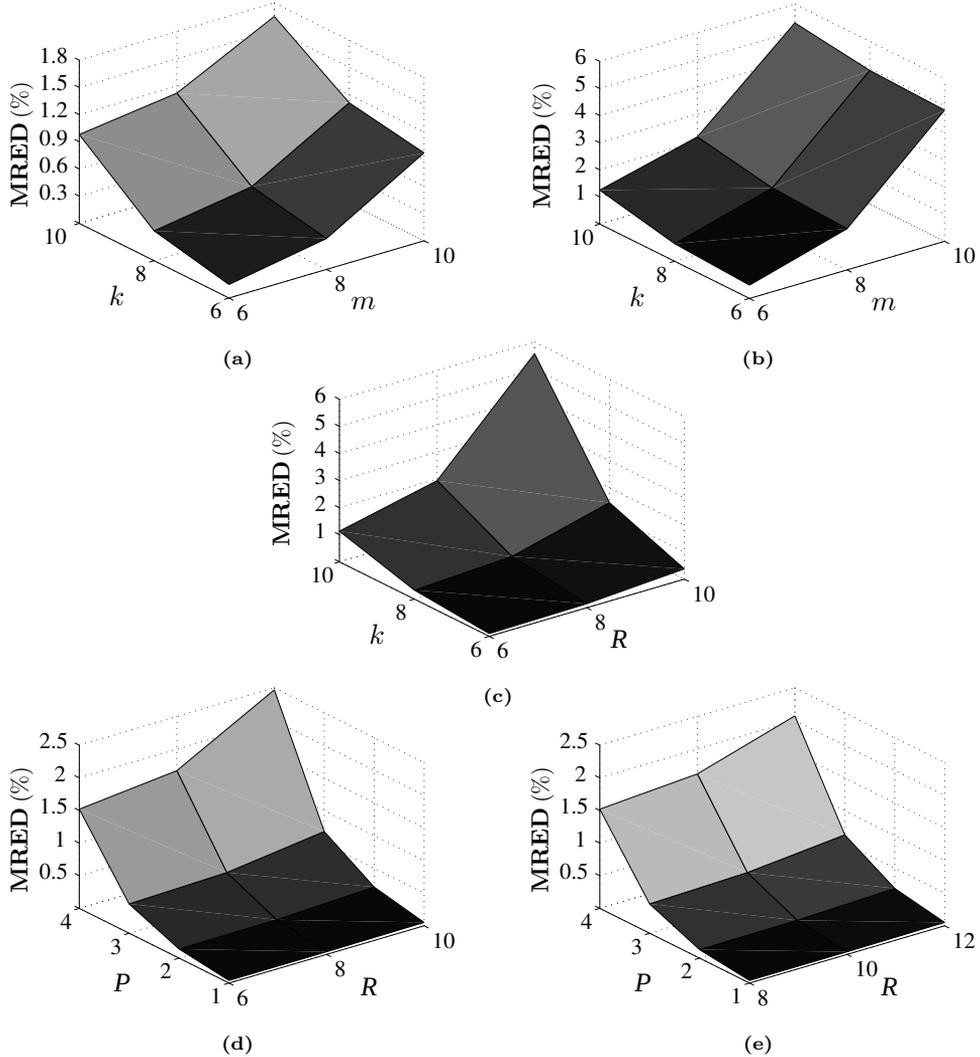
Figure 6.4: MRED variation of 16-bit multipliers based on cooperative approximation: **(a)** high-radix & high-radix (DRAD$|_{k,m}$), **(b)** high-radix & high-radix + perforation (DRADP$|_{k,m}$), **(c)** high-radix & asymmetric rounding (RADR$|_{k,R}$), **(d)** asymmetric rounding & perforation v.1 (ROUP1$|_{P,R}$), and **(e)** asymmetric rounding & perforation v.2 (ROUP2$|_{P,R}$).

for each approximation configuration we calculate MRED for 200K pairs of operands that are uniformly distributed over the 16-bit range.

Figure 6.4 presents the variation of MRED with respect to the approximation configuration. Regarding DRAD, even though the error range is small, i.e., [0.15%,

1.65%], it grows rapidly creating blank error segments. However, the error scaling provides increased density compared to the RAD design of Chapter 4, which has only three configurations ($k = 6, 8, 10$) with MRED values 0.08%, 0.28% and 0.93%. As expected, for the same configurations, DRADP exhibits larger error values than DRAD, starting from 0.49%, while the rapid error scaling is again observed. The advantage of RADR is that it smooths the rapid error scaling of RAD by adding multiple error values between two consecutive RAD configurations, especially for the small k values. The error range and scaling of ROUP1 and ROUP2 are similar to those of the AxFXU design of Chapter 5. ROUP1 features dense error scaling from 0.04% ($P = 1$) to 2.47% ($P = 4$) with several intermediate values. The MRED values of ROUP2 are similar, however, its maximum error is smaller (2.07%) compared to that of ROUP1 (2.47%).

6.4.2. State-of-the-Art Comparison: Pareto Efficiency Analysis

This section includes the experimental evaluation of the cooperative approximation techniques. To provide an overall resource–accuracy Pareto analysis and extract the most efficient designs proposed in the Dissertation, we also employ the approximate designs of Chapter 4 and Chapter 5. We remind that these designs have already formed the Pareto fronts in comparative evaluations including several state-of-the-art designs [142, 143, 149, 151, 255, 266]. Table 6.2 summarizes all our approximate multipliers.

All the designs are implemented in Verilog for $n = 16$ multiplication bit-width. The synthesis is performed with the Synopsys Design Compiler tool and the TSMC 65-nm standard-cell library. The simulations for the functional verification and the power measurements are performed with Mentor Graphics QuestaSim. The nominal supply voltage (1V) is used in both synthesis and simulation. The critical path delay and the area of the circuits are reported by Synopsys Design Compiler, while the

Table 6.2: Overview of Dissertation's approximate arithmetic circuits.

Design	Approximation Techniques	Reference
RAD2^k	high-radix-2^k	Chapter 4 [153]
AxFXU$\|_{P,R}$	perforation P, symmetric rounding R	Chapter 5 [144]
DRAD$\|_{m,k}$	high-radix-2^m, high-radix-2^k	Chapter 6 [147]
DRADP$\|_{m,k}$	high-radix-2^m, high-radix-2^k, perforation $P = 1$	Chapter 6 [147]
RADR$\|_{k,R}$	high-radix-2^k, asymmetric rounding R v.1	Chapter 6 [147]
ROUP1$\|_{P,R}$	perforation P, asymmetric rounding R v.1	Chapter 6 [147]
ROUP2$\|_{P,R}$	perforation P, asymmetric rounding R v.2	Chapter 6 [147]

power consumption is measured with Synopsys PrimeTime after performing gate-level simulation. We also evaluate the energy consumption, which is defined as the product of power and delay. We note that, like in Chapter 5, we synthesize and simulate the circuits under two different design scenarios:

(i) <u>MIN-Delay</u>: the clock constraint of each circuit is set to its critical path delay (high-performance mode).

(ii) <u>ISO-Delay</u>: the clock constraint of all the circuits is set to the same relaxed value (low-power mode).

In Figure 6.5, we present the MRED–area and MRED–energy scatter plots with all Dissertation's proposed designs. Regarding the cooperative high-radix multipliers, i.e., DRAD, DRADP and RADR, they increase the resolution of the RAD front, and even improve it with DRADP, which constitutes a better design in most cases. We note that we present only the RAD multipliers for $k = 6, 8, 10$, because large error values are produced for $k \geq 12$. This limitation of the single high-radix technique is surpassed with the cooperative approximation techniques. Regarding the multipliers implementing cooperative perforation & rounding, the asymmetric rounding techniques of ROUP1 and ROUP2 outperform the symmetric rounding of AxFXU, and thus, they constitute better design alternatives. We conclude that more fine-grained bit-level optimizations, such as the tailored rounding per partial product of ROUP2, provide better results than generic coarse-grained solutions, such as the global partial product rounding of AxFXU. Additionally, for small error values, AxFXU is not considered the most energy-efficient design, as several configurations of the cooperative approximation techniques provide improved energy consumption. Nevertheless, as shown in Chapter 5, the advantage of AxFXU is that it facilitates dynamic approximation configuration, which would be more difficult in ROUP1/ROUP2 due to the different rounding per partial product.

Overall, as shown in both design scenarios, the Pareto front is formed exclusively by the ROUP2 multipliers. The ROUP2 design family provides the best exploitation of the energy–error trade-off and further improves the state-of-the-art Pareto front (previously held by RAD [153] and AxFXU [144]) by up to $1.5\times$–$2\times$. Moreover, it is important to mention that it increases the resolution of the front, namely, it expands the already-large approximation space even more, providing a great variety of design options.

As a final stage of evaluation, we compare the new Pareto front with other state-of-the-art designs [142, 149, 151, 266]. In particular, we evaluate the energy consumption of the circuits for various MRED constraints in the range 0.15%–1.47%. Figure 6.6 reports the energy gains of the ROUP2 multipliers compared to R8ABM [151],
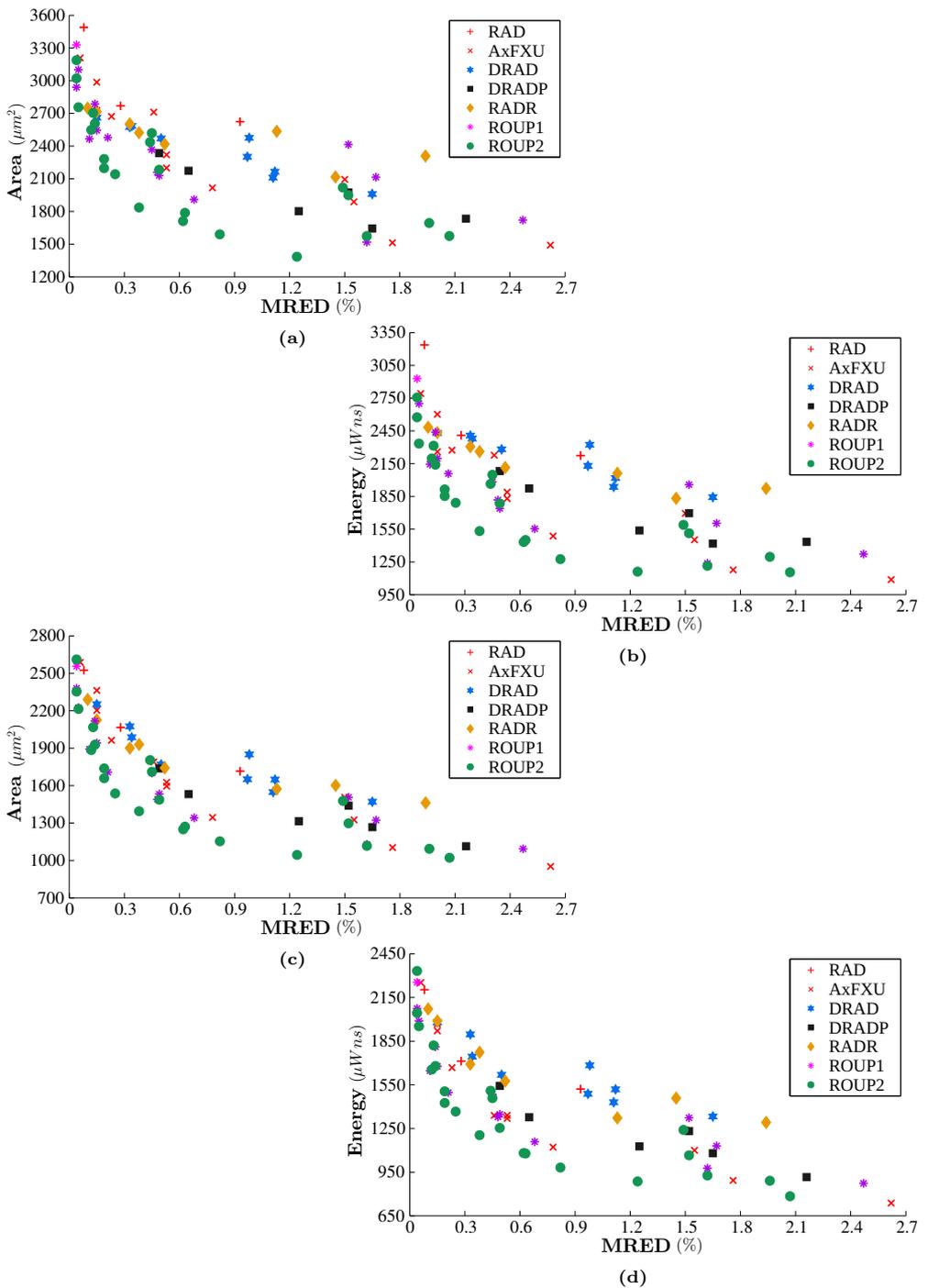
Figure 6.5: Pareto analysis for Dissertation's approximate multipliers considering MRED and area/energy under two design scenarios: **(a)**, **(b)** MIN-Delay and **(c)**, **(d)** ISO-Delay.
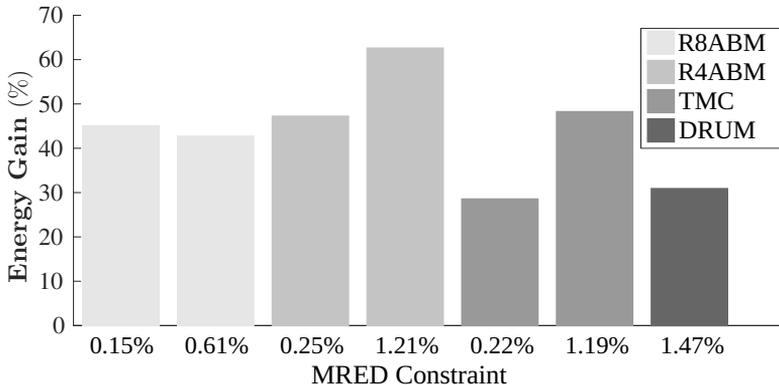
Figure 6.6: Energy gains of Dissertation's most efficient designs (ROUP2) versus state-of-the-art multipliers (R8ABM [151], R4ABM [149], TMC [266], DRUM [142]) for the same error constraint.

R4ABM [149], TMC [266], and DRUM [142]. We consider the relative energy reduction from the other design as the energy gain of ROUP2. The large approximation space and dense error segments of ROUP2 provides increased flexibility, and thus, we select configurations with MRED equal or slightly smaller than the MRED constraint. This is extremely important, because ROUP2 can maximize its gains for a given error constraint. As expected, the derived results show that ROUP2 provides significant gains ranging from 29% to 63%.

6.5. Conclusion

In this chapter, we examined the combination of arithmetic approximation techniques, targeting to expand the approximation space and identify the most efficient design solution in a comparative state-of-the-art Pareto evaluation. We defined the approximation space by combining the approximate encodings presented in Chapter 4 and Chapter 5 and as a result, we propose 5 new families of approximate multipliers. Each design family integrates two approximation techniques that can be configured independently, and thus, it provides numerous approximation configurations with different accuracy, namely, dense error scaling in the range 0%–2%. The experimental evaluation, which is performed under two design scenarios (MIN-Delay and ISO-Delay), reveals that the Pareto front is formed exclusively by designs applying cooperative approximation techniques, and more specifically, by the ROUP2 family. The ROUP2 family applies partial product perforation and asymmetric partial product rounding, and it improves the state-of-the-art Pareto front that was held by RAD [153] and AxFXU [144] by up to 1.5×–2×. Moreover, besides forming an improved Pareto

front, the cooperative approximations increase its resolution with their large sets of available configurations. Compared to other designs of the literature, the ROUP2 family can efficiently handle all the error bounds and provides energy gains up to 63% for the same error constraint.

Chapter 7

Approximate DSP & AI Hardware Accelerators

The worldwide demand for faster applications with less power consumption challenges the design of Digital Signal Processing (DSP) and Artificial Intelligence (AI) hardware accelerators. As an alternative design strategy, there is a tendency to exploit the error tolerance of DSP/AI workloads and produce approximate accelerators, which, however, improve the power efficiency and/or performance. Our work aims at the design and evaluation of approximate DSP and AI accelerators that integrate our approximate circuits, i.e., RAD, AxFXU/AxFPU, and ROUP. To provide a great diversity in terms of DSP/AI workloads, we develop several 1D/2D signal processors and Convolutional Neural Networks (CNNs). Both the development and evaluation are directed by our design methodology, which includes design space exploration at both software and hardware level. The design is not limited only to the use of our approximation techniques, but it also involves various arithmetic formats, different algorithms and state-of-the-art parallelization techniques. The exploration of all these design parameters targets to provide the most efficient approximate variants of the targeted DSP/AI hardware accelerator. Specifically for CNNs, besides fusing parallel design techniques with arithmetic approximations, we apply fine-grained approximation without retraining via our MAx-DNN framework. Moreover, we examine networks that are built on floating-point, fixed-point, and quantized integer arithmetic. The evaluation is performed with industrial-strength tools, i.e., Synopsys Design Compiler and Xilinx Vivado. In terms of accuracy, our approximate accelerators attain mean relative error up to 3 % in applications based on multiply-accumulate operations, typical quality of output image in image processing, and up to 5 % accuracy loss in CNNs. We note that the large design space offered by our approximation techniques can provide, if necessary, near-zero accuracy loss. In terms of resource gains, depending on the implementation, we deliver energy and area gains up to 70 %.
*This chapter is based on our **publications** in **[144, 145, 147, 153, 274–277]**.*

7.1. Introduction

In recent years, significant research has been conducted on the development and optimization of applications from the Digital Signal Processing (DSP) and Artificial Intelligence (AI) domains. Such applications impose strict performance and energy constraints, which constitute the selection of the computing platforms/devices for their deployment a major open issue [3]. The powerful and compute-intensive DSP/AI algorithms make the general-purpose Central Processing Units (CPUs) and the low-end Graphics Processing Units (GPUs) unfit for satisfying the application constraints [4,278]. As a result, alternative computing platforms, such as the Field-Programmable Gate Arrays (FPGAs) and the Application-Specific Integrated Circuits (ASICs), are examined for accelerating DSP/AI workloads. Both solutions provide high parallelization capabilities and increased design flexibility. The FPGAs provide reconfigurability and attractive throughput-per-power ratio [3]. The ASICs offer high computational efficiency along with low power consumption [279], while allowing more custom implementations. In all cases, the design of optimized circuits for DSP/AI workloads is a key goal for the research community.

AI brings forth various demanding workloads and algorithms, including the Convolutional Neural Networks (CNNs), which are a class of Artificial Neural Networks (ANNs) based on deep learning. CNNs are considered a state-of-the-art AI approach to provide high accuracy in computer vision tasks such as object recognition [280] and image classification [281]. FPGA-based accelerators [282–286] have started to take their place as viable and promising solutions, thus, investing in their efficient implementation forms an emerging highly valuable design paradigm. ASIC implementations also provide significant gains in the performance of CNNs [287–290], even though they lack of re-configurability. On the other hand, classic DSP algorithms are employed for the implementation of functions for image/video/signal processing. In this context, various FPGA implementations are proposed in the literature, e.g., for image processing [291–293] and telecommunication functions [294,295]. Correspondingly, there is significant research on ASIC-based accelerators for computer vision [296,297] and telecommunications [298,299].

One of the main advantages of FPGA/ASIC design is the ability to perform arbitrary bit-level manipulations, i.e., tune the bit-width of the datapath and optimize the arithmetic optimizations. Namely, the designer can select the desired arithmetic representation, including alternative formats [300,301] (also see Chapter 3), as well as define n-bit words without restriction and use custom arithmetic operators. All these operations are seamlessly applied on FPGA/ASIC technology, contrary to the general-purpose GPU/CPU processors. The design paradigm of Approximate Computing [16, 17, 19] exploits this flexibility of the FPGA/ASIC design and leverages

the error tolerance of DSP/AI applications [29, 240] to provide gains in resources (power, energy, area) and/or performance. The approximation techniques that are applied in DSP hardware accelerators involve the use of approximate arithmetic [302], voltage over-scaling [224], and over-clocking [225]. Obviously, in FPGA/ASIC design, the optimization of the datapath's bit-width is a typical task, non-related to Approximate Computing, which may result in accuracy loss to provide hardware-friendly designs (e.g., due to the use of fixed-point arithmetic or the truncation of the results). On the other hand, the design of approximate AI accelerators involves application-specific techniques: low-bit numerical formats [301], approximate arithmetic operators [303], weight quantization [304], neuron connection pruning [305], and voltage over-scaling [196].

In this chapter, we focus on *approximate DSP and AI accelerators* that are developed with Hardware Description Language (HDL) and can be deployed either on FPGA or standard-cell ASIC technology. Our main approximation approach lies in using the Dissertation's approximate multipliers in DSP and AI hardware architectures. The arithmetic components are key processing units in hardware accelerators, as they inherently affect the energy efficiency and the performance of the entire application. Their impact becomes even greater, considering that modern accelerators implement parallel architectures consisting of multiple processing units. The integration of our approximate circuits in accelerators is based on a design methodology involving both software- and hardware-level Design Space Exploration (DSE). Our goal is to assess the Dissertation's approximate designs in real-world DSP/AI applications, and in reverse, explore the error resilience of the applications and quantify the resource gains of approximate accelerators. Moreover, our approximation approach allows to seamlessly decrease the resources or improve the throughput of any given DSP/CNN without modifying the initial hardware architecture. Namely, we improve the efficiency of DSP accelerators without tuning their underlying arithmetic (a typical task of the hardware development flow), and specifically for CNNs, we provide approximate accelerators without re-training (a task that may be required in order to reduce the accuracy loss caused by the approximations). The elimination of additional training/exploration is considered an advantage, as proprietary datasets/models may not be available, and also, the training time time is usually increased due the emulation of the hardware approximations and the use of custom approximate arithmetic operators.

The **contribution** of this chapter is summarized as follows:

(i) We highlight the benefits of studying and optimizing the arithmetic of DSP and AI hardware accelerators, while proving the inherent error resilience of their workloads.

(ii) We propose a methodology for developing approximate DSP and AI hardware accelerators, which is based on extensive design space exploration involving arithmetic formats, approximation techniques, and algorithms.

(iii) We explore and quantify the resource gains and accuracy of DSP and AI hardware accelerators with approximate arithmetic.

The remainder of this chapter is organized as follows. Section 7.2 introduces our design methodology and summarizes all the approximate designs, applications and accelerators presented in this chapter. Sections 7.3–7.5 evaluate our approximate designs when used in DSP/AI accelerators. Finally, Section 7.6 draws the conclusions.

7.2. Design Methodology

The proposed methodology for designing approximate DSP and AI accelerators is illustrated in Figure 7.1. It consists of two stages: the exploration at the software level and the development of the accelerator at the hardware level. We note that our methodology follows the typical steps of hardware design. However, it includes additional functionalities (i.e., the use of approximate arithmetic units), error analysis (due to the approximations), and multi-level DSE (involving approximation techniques/configurations, arithmetic formats, and algorithms). The multi-level DSE of our design methodology is very important, because each combination of approximations, arithmetic formats, and algorithms has a different impact on the accuracy and the hardware efficiency of the accelerator.

Before analyzing the steps of our methodology, we make the following clarifications:

- Approximate Design Library: it includes the software models and the hardware descriptions of the approximate arithmetic units (e.g., RAD [153], AxFXU [144], AxFPU [145], and ROUP [147]), along with their error analysis and experimental results.

- Arithmetic Pool: it includes various arithmetic formats, e.g., the conventional fixed- and floating-point, block floating-point [306], Flexpoint [301], and quantized fixed-point [307].

- Algorithmic Pool: it includes algorithms for the entire application (e.g., the ResNet model [308] for CNN or the Log-Likelihood-Ratio (LLR) method [309] for signal demodulation) and key operations (e.g., the Winograd method [310] for convolution).
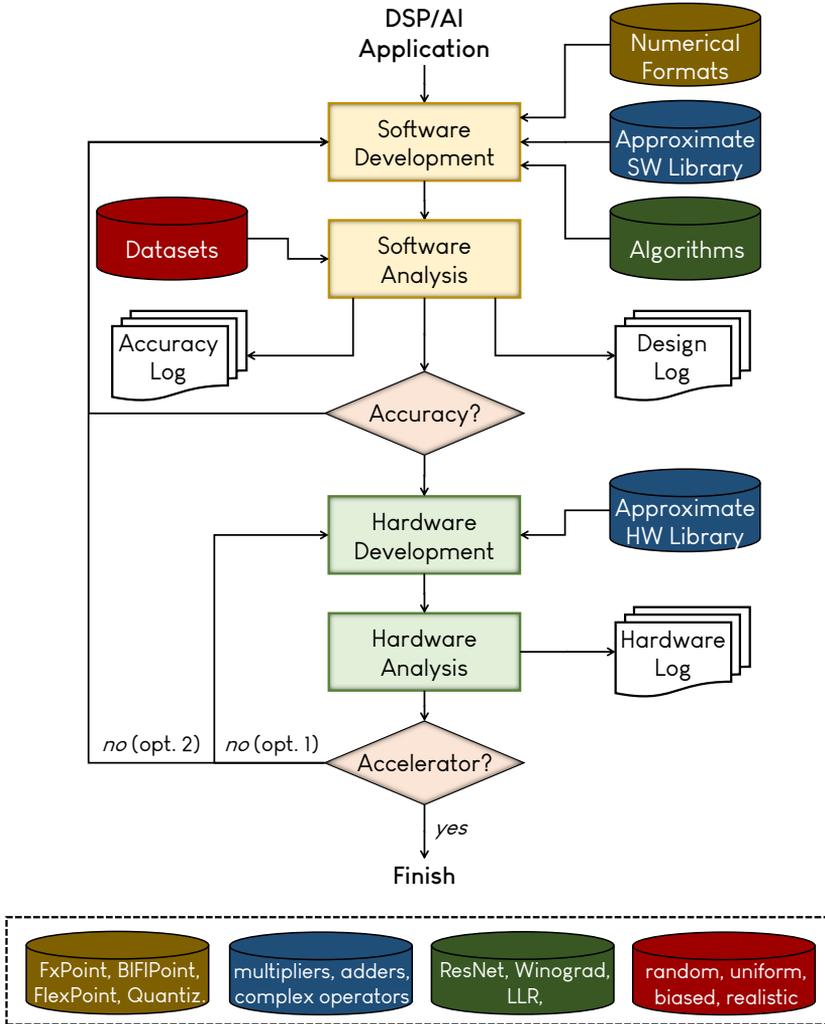
Figure 7.1: Design methodology for approximate DSP and AI hardware accelerators.

- <u>Dataset Pool</u>: it includes various application-specific test datasets (e.g., random, Gaussian/uniformly distributed, and realistic data) for evaluating the quality of the results at software level.

Firstly, we develop the application in software using various arithmetic formats and algorithms. Our software models are bit-accurate, which allows us to study the accuracy when using approximate arithmetic formats and/or approximate operators. To obtain the groundtruth (accurate) results, we run simulations on the datasets. Afterwards, we start to integrate approximate arithmetic units and create approximate

model variants of the targeted accelerator. For each variant, we run simulations to obtain the approximate results and evaluate its accuracy with application-specific metrics. Indicatively, we use metrics such as the Structural Similarity Index (SSIM) and Peak Signal-to-Noise Ratio (PSNR) for image processing, the Bit Error Rate (BER) for telecommunications, and the classification accuracy for CNNs. Furthermore, for each variant, we report possible bottlenecks in hardware, e.g., regarding parallelization or the implementation of complex operations. The next step is to examine which approximate variants satisfy the constrained quality of results, and then select the most efficient for implementation on FPGA/ASIC technology. In case the accuracy constraints are not met, we design new approximate variants by combining different arithmetic, algorithms, and approximate units.

When the software-level DSE is over, we continue with the typical HDL development. We implement the selected approximate variants integrating our design choices for arithmetic, algorithms, and approximation techniques. Finally, in the evaluation phase we examine if the accelerator satisfies the constraints for performance and resource utilization. In case the results are not the expected ones, we examine alternative design approaches (e.g., parallelization scheme) or even return to the initial software-level DSE to create new approximate variants.

In the remainder of the chapter, we present experimental results from DSP/AI hardware accelerators employing our approximate multipliers. More specifically, based on our methodology, we design parallel architectures with approximate arithmetic for 1D/2D signal processing and CNNs. Each section is dedicated to one of our approximate designs, i.e., RAD [153] (presented in Chapter 4), AxFXU [144] & AxFPU [145] (presented in Chapter 5), and ROUP [147] (presented in Chapter 6). Table 7.1 summarizes the details for all the applications and accelerators that are presented in the

Table 7.1: Overview of Dissertation's approximate DSP & AI hardware accelerators.

Design	Domain	Application	Accelerator[1]
RAD	Digital Signal Processing	Sobel, FIR, MatMul	ASIC
RAD	Digital Signal Processing	QAM Demodulation	FPGA
RAD	Artificial Neural Networks	CNN (Ship Detection)	FPGA
AxFXU	Digital Signal Processing	Sobel, FIR, MatMul	ASIC
AxFPU	Digital Signal Processing	Gaussian Blurring	ASIC
AxFPU	Artificial Neural Networks	CNNs (MNIST, CIFAR-10)	ASIC
AxFPU	Machine Learning	K-Means Clustering	–
AxFPU	Linear Algebra	LU Decomposition	–
ROUP	Artificial Neural Networks	ResNet-8 (CIFAR-10)	ASIC

[1] Implementation details: with HDL, for ASIC (standard-cell) or FPGA (programmable logic).

following sections.

7.3. Design and Evaluation of Applications with RAD

In this section, we employ our approximate high-radix RAD multipliers [153] in real-world applications. From the DSP domain, we implement the Sobel edge detector, a Finite Impulse Response (FIR) filter, a matrix multiplication unit, and a Quadrature-Amplitude-Modulation (QAM) demodulation filter. From the AI domain, we implement a custom CNN for ship detection on satellite images.

7.3.1. Approximate DSP Accelerators

A well-known filter for finding the object boundaries in an image is the discrete Sobel operator [311]. The Sobel operator consists of two 3×3 kernels that are applied linearly to the image to compute an approximation of the gradient of the intensity function. The two convolutions compute the changes in brightness in the horizontal and vertical orientation. For the hardware implementation of convolution, we adopt the ordinary approach [312], which includes a serial-to-parallel converter along with the structure of multipliers and adders, as illustrated in Figure 7.2 for generic $r \times r$ kernel. The convolution engine inputs one pixel per clock cycle in raster-scan order and forwards it to the serial-to-parallel converter that outputs $r \times r$ pixels per cycle. The converter consists of $r \times r$ DFFs connected to r FIFOs in a linear array topology. In practice, the $r \times r$ DFFs slide over the input image in raster-scan order, while each FIFO temporarily stores a row of pixels. In a pipeline fashion, the $r \times r$ pixels along with the $r \times r$ kernel weights are forwarded to r^2 multipliers, and then, the multiplication results are added to produce the new pixel. Similar architectures are designed for the FIR filter and the matrix multiplication unit. Regarding FIR, we employ a 32-tap low-pass filter with cut-off frequency equal to 20KHz and 16-bit coefficients, while for matrix multiplication we select 3×3 tiling.

Besides the aforementioned hardware architectures, we design circuits for QAM demodulation, i.e., a key digital function in the baseband processing chain (telecommunications). We consider 64-QAM keying signals and their corresponding gray-coded constellation map, where each constellation point is represented by a 6-bit vector. We employ the approximate LLR method [309] for predicting the i-th bit of the received symbol. This method avoids the expensive logarithmic and exponent calculations of the exact LLR method by using only the nearest constellation points with i-th bit equal to 0 and 1. The generic hardware architecture for M-QAM demodulation

(a)

(b)
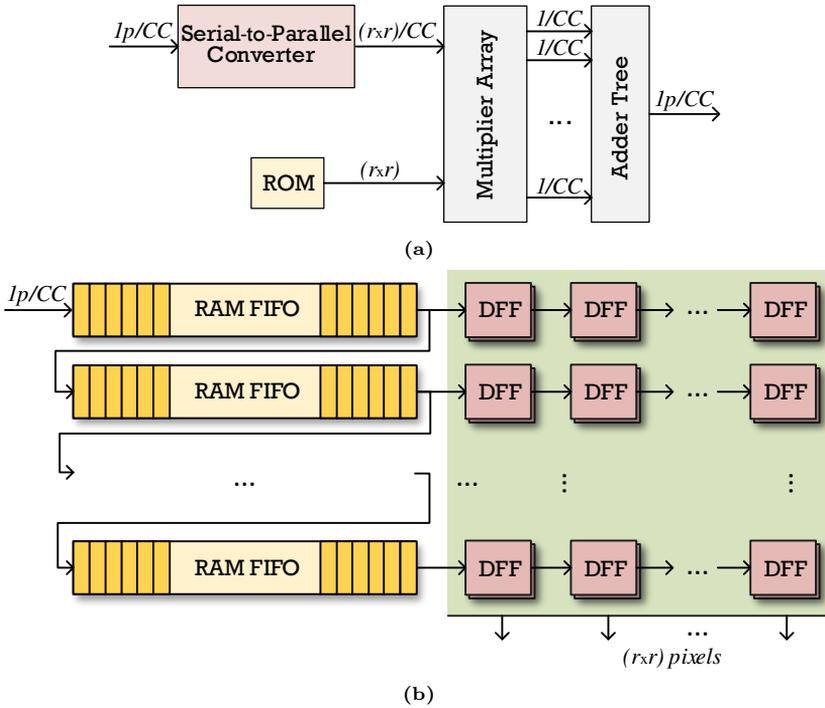
Figure 7.2: Hardware architecture of $r \times r$ convolution: **(a)** convolution engine and **(b)** serial-to-parallel converter.
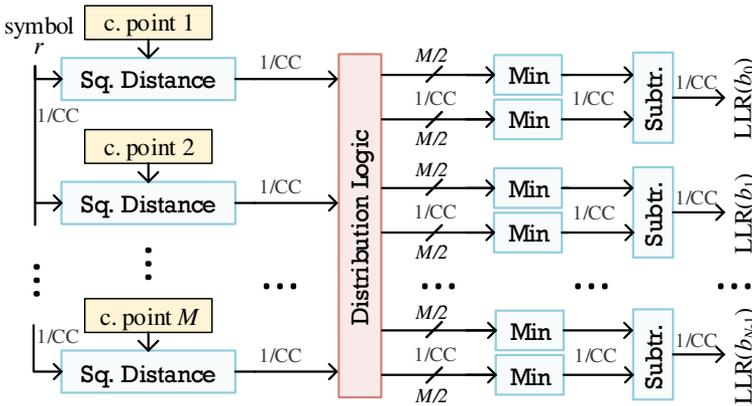
Figure 7.3: Hardware architecture of M-QAM demodulation.

is presented in Figure 7.3. It processes in parallel the input symbol per constellation point and performs on-the-fly computations (e.g., square distances, minimum, subtractions).

Experimental Evaluation

We implement the Sobel edge detector, the FIR filter, and the matrix multiplication in Verilog and synthesize them with Synopsys Design Compiler and the TSMC 65-nm standard-cell library, targeting to ASIC-based approximate DSP accelerators. The resource gains and accuracy results of the accelerators with our RAD multipliers [153] are reported in Table 7.2. The Table also includes results for the multipliers [142, 149, 151] that are employed in the evaluation of RAD in Chapter 4. Regarding accuracy, for Sobel, we use the Correct Edge Ratio (CER), which measures the number of correct edges detected per total number of edges, while for the other two accelerators, we use MRED (as defined in Chapter 4). For input data, we use the Cameraman benchmark image for Sobel (illustrated in Figure 7.4 along with the various output images) and 200K random generated inputs for the FIR filter and the matrix multiplication.

Starting with the Sobel edge detector, the RAD multipliers achieve the highest accuracy by detecting almost all the edges (more than 99.87% edges are correctly detected). Moreover, they deliver remarkable resource gains, i.e., up to 54.8% in energy, 46.% in area, and 8.4% in critical path delay. DRUM6 exhibits similar energy gain (55.3%), however, it imposes a delay overhead of 7.8% and detects 1% less edges compared to RAD1024. Furthermore, it is worth to note that although R8ABM1 and R8ABM2-15 feature small MRED values (see Chapter 4), they decrease the quality of the results in Sobel, as they exhibit a CER of 58.90% and 54.41%, respectively.

The RAD multipliers also outperform the other designs in the FIR filter. More specifically, they achieve up to 34.1% energy gain in exchange for small a MRED of 3.60%. Similar to Sobel, R8ABM1 and R8ABM2-15 introduce large errors, i.e., MREDs of 13.51% and 33.98%, respectively. In this application, DRUM6 achieves significant energy reduction (21.4%) for considerable error (9.18%). However, these values are worse than those of RAD1024, which provides 1.6× larger energy gains and 2.6× smaller MRED.

Regarding the matrix multiplication, all the examined multipliers perform very well in terms of accuracy. When considering similar error values, our RAD multipliers achieve larger energy savings than the other designs. In particular, RAD1024 delivers 37.7% energy reduction while exhibiting an MRED of 0.57%. R4AMB1-14 and RAD64 feature the smallest MRED (0.07%), however, RAD64 delivers 1.4× larger energy gains. Similarly, R4AMB2-14 and RAD256 exhibit similar MRED, however, the latter delivers 3.6× larger energy gains.

Finally, we implement the 64-QAM demodulation on the Zynq UltraScale+ ZCU106 FPGA using VHDL and the Xilinx Vivado tool. Based on our methodology, we em-

Table 7.2: Experimental results of approximate RAD-based DSP applications on TSMC 65-nm standard-cell.

	Design	Delay (%)[1]	Energy (%)[1]	Area (%)[1]	CER (%)
Sobel	RAD64	1.8	22	20.8	99.98
	RAD256	6	37.3	33.9	99.96
	RAD1024	8.4	54.8	46	99.87
	R8ABM1 [151]	1.1	1.3	1.6	58.90
	R8ABM2-15 [151]	5.4	9.6	8.4	54.41
	R4ABM1-14 [149]	0.6	3.4	3.8	99.80
	R4ABM1-16 [149]	0.6	6.9	4.7	99.36
	R4ABM2-14 [149]	1.8	7	5	99.11
	R4ABM2-16 [149]	3	7.1	5.1	98.27
	DRUM6 [142]	−7.8	55.3	46.4	98.87

	Design	Delay (%)[1]	Energy (%)[1]	Area (%)[1]	MRED (%)
FIR	RAD64	3	6	9.8	0.41
	RAD256	7.1	17	18.3	0.96
	RAD1024	11.1	34.1	34.7	3.60
	R8ABM1 [151]	0	−0.2	−0.1	13.51
	R8ABM2-15 [151]	6.1	5.3	9.2	33.98
	R4ABM1-14 [149]	0	1.5	2.2	5.28
	R4ABM1-16 [149]	1	3	3.1	23.38
	R4ABM2-14 [149]	1	3.9	4.8	6.10
	R4ABM2-16 [149]	3	4.1	5.5	30.43
	DRUM6 [142]	−32.3	21.4	21.6	9.18

	Design	Delay (%)[1]	Energy (%)[1]	Area (%)[1]	MRED (%)
MatMul	RAD64	6.2	7.8	14	0.07
	RAD256	8	25.8	26.9	0.17
	RAD1024	11.5	37.7	38.7	0.57
	R8ABM1 [151]	0	−1.7	0	0.10
	R8ABM2-15 [151]	5.3	7.1	12.8	0.38
	R4ABM1-14 [149]	0.9	5.8	6.4	0.07
	R4ABM1-16 [149]	0.9	6.6	7.3	0.29
	R4ABM2-14 [149]	3.5	7.1	7.7	0.15
	R4ABM2-16 [149]	3.5	8.9	9.5	0.75
	DRUM6 [142]	−30.1	32.8	34.9	1.60

[1] Refers to % resource gains (relative reduction) in comparison with the accurate design.
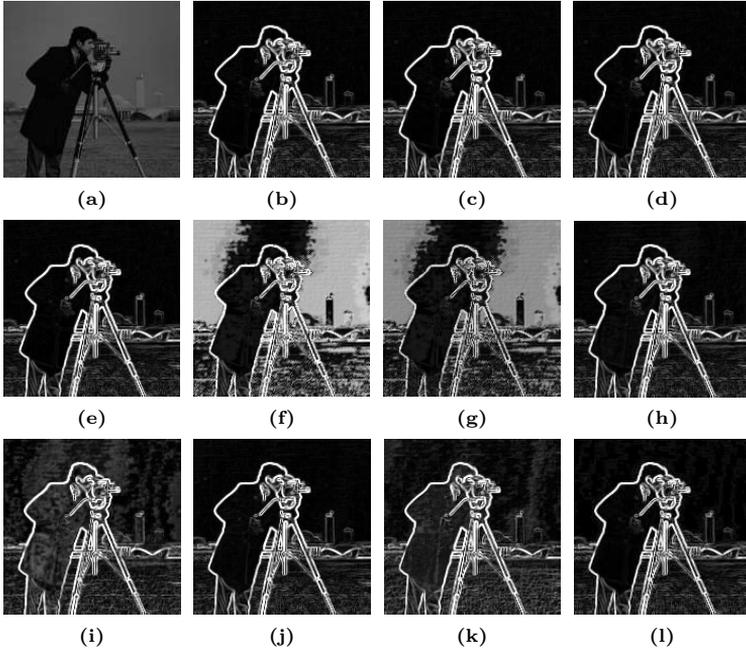
Figure 7.4: I/O images of approximate Sobel edge detector: **(a)** input image and output image with **(b)** accurate multiplier, **(c)** RAD64, **(d)** RAD256, **(e)** RAD1024, **(f)** R8ABM1 [151], **(g)** R8ABM2-15 [151], **(h)** R4ABM1-14 [149], **(i)** R4ABM1-16 [149], **(j)** R4ABM2-14 [149], **(k)** R4ABM2-16 [149] and **(l)** DRUM6 [142].

Table 7.3: Experimental results of approximate RAD-based 64-QAM demodulation on Zynq ZCU106 FPGA.

Design	LUT (%)[1]	Clock (MHz)	Throughput (MSamples/s)	BER[1]
Accurate (Fl. Point)	46	286	286	$10^{-1} - 10^{-4}$
Accurate (Fx. Point)	24	312	312	$10^{-1} - 10^{-4}$
RAD64 (Fx. Point)	16	321	321	$10^{-1} - 10^{-4}$

[1] Refers to % resource utilization of ZCU106 (230400 LUTs).
[2] Measured for SNR/symbol values 0–14dB.

ploy both 32-bit floating-point and 16-bit fixed-point arithmetic. As accuracy metric, we use BER. Table 7.3 summarizes the results for the most efficient 64-QAM circuits. Fixed-point provides similar BER values with the "golden" floating-point model, i.e., from 10^{-1} to 10^{-4} for Signal-to-Noise Ratio (SNR) per transmitted symbol in the range 0–14dB. RAD64 retains BER at the same orders of magnitude (the maximum

relative error is 1% compared to floating-point). In terms of resources, the design using RAD64 provides 65% LUT reduction and 1.12× speedup versus the floating-point variant, and small but still important gains versus the fixed-point variant. Finally, according to our design methodology, the algorithmic pool for QAM demodulation includes other LLR methods, such as the exact and the piecewise LLR. However, our software/hardware DSE highlights the approximate LLR method as the most efficient one. For example, the approximate LLR method almost matches the BER scaling of the exact method (for small SNR values, it is the same), while it provides significantly smaller resource utilization, i.e., ~3× less LUTs.

7.3.2. Approximate CNN Accelerators

Following the evaluation in DSP applications, we employ RAD in CNNs. In particular, we develop a parallel CNN architecture for detecting ships on satellite images. Our CNN model consists of 4 convolutional and 2 fully connected layers. In total, the convolutional layers have 144 filters (32+16+64+32) with 33K weights, while each filter includes a ReLU activation function and 4-to-1 max pooling. The fully connected layers consist of 50 neurons (48+2) with 98K weights. The CNN is developed in TensorFlow and trained with 16-bit floating-point arithmetic on 128×128 RGB images [313], providing a classification accuracy of 96.8%.

For our design, we consider the typical CNN: the network consists of convolutional layers, which include filters, which perform multiple convolutions. Namely:

- a convolutional layer inputs M image channels, processes them with N filters, and outputs N feature maps (one by each filter).

- each one of the N filters inputs M image channels, perform M convolutions (one per channel), and outputs 1 feature map.

We apply parallelization at two distinct levels: (i) at network level, where we deploy multiple parallel convolution engines per filter, and (ii) at convolution level, where we parallelize the number of arithmetic operations per $r \times r$ convolution. The network-level parallelization is supported by a parallel multi-bank memory organization [312], which allows multiple channels to be accessed concurrently. On the other hand, the convolution-level parallelization is implemented as in Sobel (see Section 7.3.1 and Figure 7.2). Regarding the arithmetic, we consider both the conventional fixed- and floating-point formats, and we also adopt the block floating-point format [306]. Finally, for the implementation of the convolutional engine, besides the ordinary approach used in Sobel, we employ the Winograd algorithm [310]. Next, we present the details for each one of our design choices.
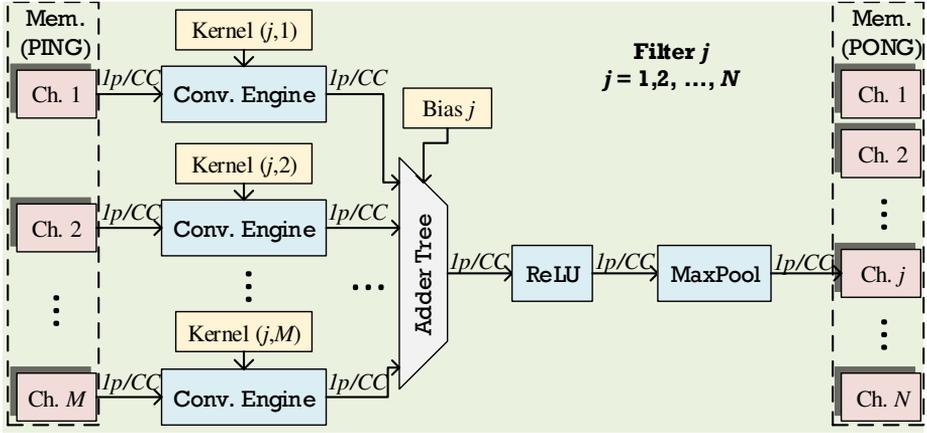
Figure 7.5: Hardware architecture of filter in M-input convolutional layers.

<u>Network-Level Parallelization</u>: The convolution layers are executed serially by reusing E convolutional engines in parallel. Figure 7.5 shows a representative setup with a fully parallelized filter processing all M input channels with $E = M$ engines. Each engine computes a 2D convolution with a throughput of 1 pixel per clock cycle, as shown in Figure 7.2. In a pipelined fashion, the E output pixels per cycle are accumulated in an adder tree along with the filter's bias, and then, they are forwarded through ReLU and MaxPool to the memory storing the output channels. When $E > M$, we compute multiple filters in parallel. The data memories (left and right in Figure 7.5) are designed with 1 bank per channel (being reused during CNN steps) in a ping-pong setup to interchange between successive convolutional layers. For each convolution engine, the corresponding kernel weights are stored in ROM and are loaded according to the running filter. The architecture operates in burst mode with a convolution's cycle budget being almost equal to the size of the input channel.

<u>Block Floating-Point</u>: In our floating-point convolution engine, instead of processing each number as an individual exponent-mantissa pair, we group an entire block of pixels by assuming a common exponent [306]. As a result, the main operations reduce to fixed-point arithmetic, i.e., simple mantissa multiplication/addition. At the start/end of the block processing, we transform all data (pixels and weights) from/to the standard floating-point format to/from block floating-point based on their maximum exponent. For the transformation to block floating-point (before starting the processing), the mantissas of pixels and weights are shifted according to their max exponent, which is either detected or given as input. The transformed mantissas are forwarded to the convolutional engine (see Figure 7.2). In parallel, the block's common exponents (for pixels and weights) are added and shifted to be included in the mantissa

accumulation that is performed in the convolutional engine. For the transformation to floating-point (after finishing the processing), the new max exponent is detected and is used to normalize the mantissas. We note that the internal multiplications can be performed by accurate or approximate multipliers, and also, another convolution engine can be used (e.g., the Winograd-based one).

Winograd Convolution: The Winograd 2D convolution algorithm [310] with an $r \times r$ kernel computes an $m \times m$ output using $(m + r - 1) \times (m + r - 1)$ image tiles with stride $r - 1$. It requires $(m + r - 1) \times (m + r - 1)$ multiplications to compute the $m \times m$ output, while the ordinary convolution needs $(m \times m) \cdot (r \times r)$. The $\{m = 2, r = 3\}$ configuration is widely used, because it provides resource efficiency and numerical stability. Compared to the ordinary convolution, this Winograd configuration requires $2.25\times$ less element-wise multiplications. In brief, it applies the following steps [310]:

  i) split the input image into 4×4 tiles d_i with stride 2.
 ii) transform the 3×3 kernel g into the 4×4 kernel $G = AgA^T$, where A is a 4×3 matrix with elements $0, -1/2, +1/2$.
iii) transform the image tiles d_i into 4×4 tiles $D_i = B^T d_i B$, where B is a 4×4 matrix with elements $0, -1, +1$.
 iv) compute the intermediate outputs F_i via the element-wise multiplication $F_i = D_i \odot G$.
  v) transform the intermediate outputs F_i into 2×2 matrices $Y_i = C^T F_i C$, where C is a 4×2 matrix with elements $0, -1, +1$.

Figure 7.6 depicts the architecture of our Winograd convolution engine, which replaces 4 ordinary convolution engines. This design exploits the gap created by the aforementioned stride-2 steps during continuous single-pixel raster-scan streaming, and it employs one common core for processing 4 distinct channels via time multiplexing. In the input, we use 4 serial-to-parallel converters to slide a distinct 4×4 window per channel. Each 4×4 window is multiplexed towards the 3 common processing units to perform the D_i, F_i, Y_i calculations via pipelined multiplications/additions. At each clock cycle, the corresponding kernel is selected from an initialized DFF array and transformed to G. We use fixed input delays and a control unit to synchronize the 4 streams and achieve a constant output rate of 4 pixels per clock cycle. We note that the arithmetic operations can be approximated either using the block floating-point format or approximate arithmetic units.

Experimental Evaluation

We develop various CNN variants in parametric VHDL based on the DSE of our methodology, and we use Xilinx Vivado to implement them on the Zynq-7020 FPGA.
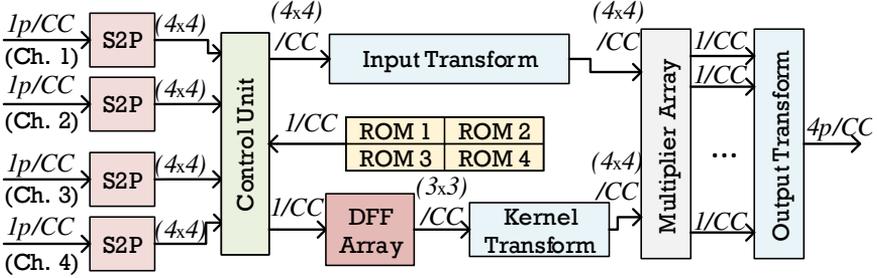
Figure 7.6: Hardware architecture of Winograd convolution processing 4 channels in parallel.

Towards exploiting every specialized resource of the FPGA, we implement the entire network with a mixture of accurate and approximate convolution engines. The key idea is, on one hand, to utilize the DSP blocks by doing 'default' multiplications, while on the other hand, to decrease the LUT utilization via approximations (e.g., when bound to revert to LUT usage due to lack of DSPs). The 'default':'approximate' engine ratio depends on the capacity of the FPGA. For Zynq-7020, we select the 16:16 ratio, i.e., 16 typical and 16 approximate engines, for a total of 32× parallel engines (which also leads to the full utilization of the available RAMBs). For the approximations, we employ our RAD multipliers [153], as well as all the algorithms and arithmetic representations discussed in this section, i.e., conventional fixed- and floating-point, block floating-point, and Winograd convolution.

Table 7.4 reports representative results from our DSE. The accurate floating-point CNN achieves a maximum parallelization of only 8× due to increased cost and constraints of Xilinx's IP for floating-point calculations, which relies on DSPs. By applying the schemes derived by our methodology, we achieve 4× higher throughput with a negligible accuracy loss of 0.1%. More specifically, the block floating-point format delivers 32× parallelization, whether with Winograd or not, delivering up to 730 Frames Per Second (FPS). In terms of resource utilization, as expected, the increased parallelization requires more RAMBs, however, the use of the RAD256 multiplier results in not over-utilizing the LUT resources. Moreover, the use of the Winograd algorithm instead of the ordinary convolution in the RAD-based block floating-point design reduces the DSP utilization by 41%, while maintaining almost the same throughput and LUT utilization. For the accurate fixed-point CNN, the FPGA resources are automatically balanced, to some extent, by Vivado. The gains provided by our approximate variants are summarized in the LUT resources (there is also a small increase in throughput). In particular, the mixture of engines decreases the LUTs by 7% when using only RAD256 and by 38% when also using Winograd. Again, the accuracy is not affected by either the fixed-point arithmetic or our RAD256 multiplier, as there

Table 7.4: Experimental results of approximate RAD-based Ship-Detection CNN (128×128×3, 132K param.) on Zynq-7020 FPGA.

Design	Paral.	LUT (%)[1]	DSP (%)[1]	RAMB (%)[1]	Clock (MHz)	Throughput (FPS)
Accurate (Fl. Point)	8	37	91	78	125	182
RAD256 (BFl. Point)	32	65	100	95	125	730
RAD256 (Win. BFl. Point)	8 × 4	69	59	95	124	724
Accurate (Fx. Point)	32	69	58	95	118	689
RAD256 (Fx. Point)	32	60	54	95	124	724
RAD256 (Win. Fx. Point)	8 × 4	43	58	95	112	654

[1] Refers to % resource utilization of Zynq-7020 (53200 LUTs, 220 DSPs, 140 RAMBs).
[*] The accuracy loss is 0.1%–0.2% among the designs.

is only loss up to 0.2%.

7.4. Design and Evaluation of Applications with AxFXU/AxFPU

In this section, we evaluate our approximate AxFXU [144] and AxFPU [145] multipliers in real-world applications. We remind that AxFXU is the fixed-point variant and AxFPU is the floating-point variant, as well as that both variants combine the perforation and rounding approximation techniques. To evaluate AxFXU, we employ again our hardware architectures for the Sobel edge detector, the FIR filter, and matrix multiplication. Additionally, we implement a Gaussian blurring filter on floating-point arithmetic to evaluate AxFPU. From the AI domain, we develop two floating-point CNNs on the MNIST and CIFAR-10 datasets. Finally, we evaluate the AxFPU design in two functions that employ floating-point numbers, i.e., the K-means clustering and the LU decomposition.

7.4.1. Approximate DSP Accelerators

Starting with AxFXU, we employ again the fixed-point hardware architectures for Sobel, FIR, and matrix multiplication. The Sobel edge detector uses two 3×3 convolution kernels, the FIR filter is low-pass with cut-off frequency of 20KHz and 16-bit 32 coefficients, while for matrix multiplication we implement 3×3 tiling.

Regarding AxFPU, we select the Gaussian blurring filter, which smooths the image to remove details and noise. This convolution filter is widely used in image processing,

e.g., before edge detection to reduce the levels of noise in the image. We implement the 3×3 Gaussian blurring with our convolution engine illustrated in Figure 7.2. We note that hardware filters such as Gaussian blurring, do not require floating-point arithmetic to provide acceptable accuracy results. However, considering that the optimization of the arithmetic/bit-width in DSP applications is not a negligible task, and it is mainly performed by low-level hardware engineers, we evaluate the scenario where the default floating-point arithmetic is selected. Moreover, considering that our AxFPU multipliers can be integrated in an embedded processor, our evaluation aims to prove their power efficiency in case the embedded developer does not use integer coefficients for Gaussian blurring.

Experimental Evaluation

We develop the DSP kernels in Verilog and synthesize them with Synopsys Design Compiler and the TSMC 65-nm standard-cell library, targeting to ASIC-based accelerators. Like in the evaluation of RAD, for Sobel we use the CER error metric, which measures the number of correct edges detected per total number of edges, while we use MRED for FIR and the matrix multiplication. For the Gaussian blurring, we use two well-established image processing metrics, i.e., PSNR and SSIM. As benchmark images, we use Cameraman for Sobel and both Cameraman and Lena for Gaussian blurring. Moreover, the FIR and matrix multiplication kernels take 200K random generated inputs.

For evaluating AxFXU, we employ three configurations with different approximation degree (AxFXU$|_{2,2}$, AxFXU$|_{3,6}$, and AxFXU$|_{4,6}$). Figure 7.7 shows the delivered reductions in resources along with the application's accuracy metric. Regarding Sobel, as shown in 7.7a, CER is more than 98.5% in all configurations, like in the RAD-based Sobel accelerators (see Table 7.2). However, the use of AxFXU provides increased resource gains, i.e., up to 54% in area, 58% in energy, and 18% in delay. Especially for delay, AxFXU provides $2.3\times$ smaller critical path versus RAD. Similar behavior is observed for the other two applications. The important outcome from this exploration is that, again, the use of our approximate multipliers does not damage the accuracy of the application. Only FIR outputs larger error, however, this is normal, as every output depends on the previous ones, and the error is propagated to the next calculations. Specifically, the MRED is 3.66%–6.65%, but it can be handled using less aggressive approximation configurations.

Regarding the AxFPU-based Gaussian blurring, we employ Pareto-front configurations with diverse approximation degree and error sensitivity. Our goal is to assess the smoothing's accuracy over different approximation levels. Table 7.5 reports the resource gains and the accuracy metrics for the benchmark images. The results

(a)

(b)

(c)
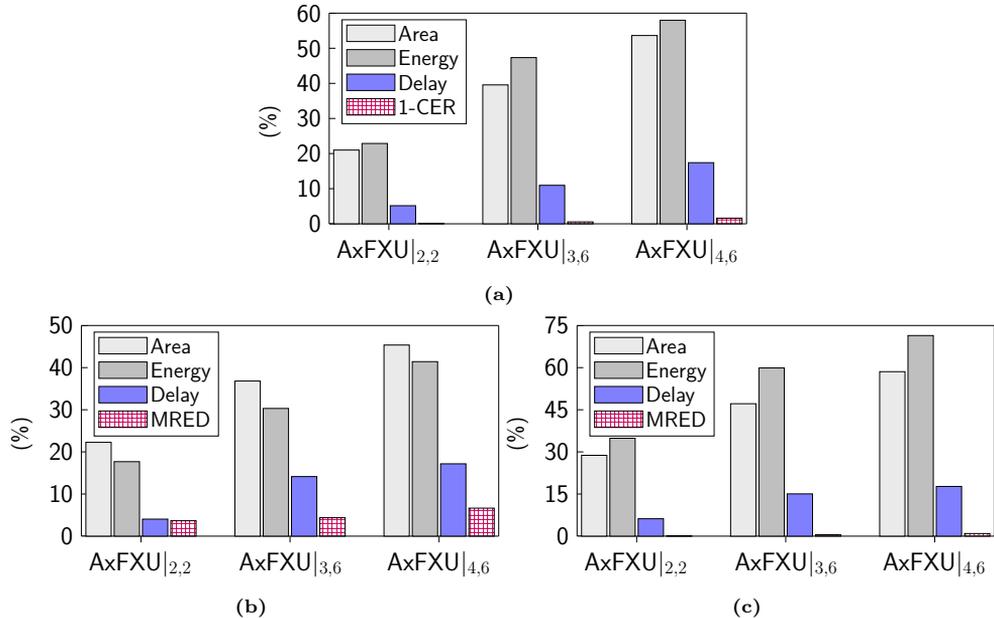
Figure 7.7: Experimental results of approximate AxFXU-based DSP applications on TSMC 65-nm standard-cell: resource gains and accuracy results of **(a)** Sobel, **(b)** FIR, and **(c)** MatMul.

Table 7.5: Experimental results of approximate AxFPU-based Gaussian Blurring on TSMC 65-nm standard-cell.

Design	Area (%)[1]	Power (%)[1]	PSNR[2] (dB)	SSIM[2] −	PSNR[3] (dB)	SSIM[3] −
AxFPU16$\|_{1,0}$	5.4	2.6	∞	1	∞	1
AxFPU16$\|_{3,4}$	28.4	22.9	59.22	0.99	55.17	0.99
AxFPU16$\|_{4,6}$	41.9	40.3	50.86	0.98	48.20	0.87
AxFPU32$\|_{4,12}$	34.5	27.9	∞	1	∞	1
AxFPU32$\|_{6,14}$	48.7	44.8	∞	1	∞	1
AxFPU32$\|_{10,18}$	60.4	57.4	54.46	0.99	52.99	0.95

[1] Refers to % resource gains (relative reduction) in comparison with the accurate design.
[2,3] Refers to Lena and Cameraman benchmark images, respectively.

show that the quality of the calculations is barely affected by the approximations, which, at the same time, deliver significant resource gains in the convolution engine. More specifically, the designs exhibit more than 50dB PSNR on average, i.e., acceptable PSNR values for lossy image processing, and near 1 SSIM values, while the less approximation-aggressive designs produce the "golden" output. For instance,

Figure 7.8: I/O images of approximate Gaussian blurring: **(a)**,**(f)** input images, and output image with **(b)**,**(g)** accurate multiplier, **(c)** AxFPU16$|_{1,0}$, **(d)** AxFPU16$|_{3,4}$, **(e)** AxFPU16$|_{4,6}$, **(h)** AxFPU32$|_{4,12}$, **(i)** AxFPU32$|_{6,14}$, **(j)** AxFPU32$|_{10,18}$.

AxFPU32, which has already proven to be less sensitive to approximations (see Chapter 5), produces the "golden" output for both images, while delivering power gains up to 45%. The gains reach 58% in exchange for a PSNR of 53.7dB and a SSIM of 0.97 on average. Finally, Figure 7.8 presents a visual inspection of the images produced using the approximate AxFPU multipliers, indicating a very small difference compared to the image produced with accurate computations.

7.4.2. Approximate CNN Accelerators

Next, we assess the resource efficiency and accuracy of CNNs that replace the multiplications in their the convolutional layers with AxFPU multipliers. The first CNN, which is trained and evaluated on the MNIST dataset [314], consists of one convolutional layer with 12 filters, followed by another one with 24 filters. The second CNN, which is trained and evaluated on the CIFAR-10 dataset [315], consists of two convolutional layers with 32 filters and two convolutional layers with 64 filters. Both CNNs use ReLU as activation function and rely on two fully-connected layers to generate the 10-value output vectors. The CNN models are generated with TensorFlow in two flavors, i.e., with 16- and 32-bit floating-point arithmetic, in order to evaluate our half- and single-precision AxFPU multipliers. The training is performed on 80% of the datasets, generating 32-bit floating-point weights and biases. The 16-bit floating-point models are created by applying post-training quantization on the 32-bit models. Finally, we remind that we do not aim to generate the most efficient CNN networks

(e.g., for the selected models/datasets, fixed-point or quantized integer arithmetic may be sufficient). Our goal is to evaluate our approximations in floating-point CNN workloads, provide hardware efficiency without structural modifications, and quantify the actual resource gains.

Experimental Evaluation

To estimate the power consumption of the approximate CNN accelerators, we adopt the model used in [303]. Namely, we calculate the power of each layer based on the number of its multiplications and the power of the multiplication circuits, which is experimentally measured. To estimate the total power consumption, we accumulate the individual powers of all the layers. Moreover, we perform similar calculations to estimate the area of the CNN accelerators. Targeting to standard-cell ASIC technology, the AxFPU multipliers are implemented on the TSMC 65-nm library using industrial-strength tools, i.e., Synopsys Design Compiler for synthesis and Synopsys PrimeTime for measuring power.

Table 7.6 reports the results for the two CNNs using the AxFPU designs. In terms of accuracy, the small configurations of AxFPU do not affect at all the classification results, especially for the MNIST dataset, while for CIFAR-10, the accuracy loss starts growing faster. For more aggressive approximations, i.e., $P = 4$, $R = 6$ in AxFPU16 and $P = 11$, $R = 19$ in AxFPU32, the accuracy loss for MNIST is 0%, while it is small for CIFAR-10 (in the range 1.3%–5.4%). This accuracy loss is traded for significant gains in area, ranging in 33%–67.1% and 30.9%–63.4% for 16- and 32-bit floating-point, respectively, and similar gains in power (up to ~64%). Finally, according to our exploration, more aggressive approximation configurations are not recommended, as the accuracy loss explodes, while the extra gains are negligible.

7.4.3. Approximate Clustering and Linear Algebra

In this section, we evaluate the use of AxFPU in Rodinia 3.1 benchmarks [316]. Targeting to explore different application domains than the classic DSP/AI ones, we employ two benchmarks from machine learning and linear algebra, i.e., domains that require floating-point arithmetic and high precision, and we examine the accuracy results when using AxFPU. The first benchmark is K-means clustering [317], which is an unsupervised learning algorithm used extensively in the data mining domain to classify unlabeled data into clusters. Moreover, we use AxFPU in the LU decomposition [318], which is a typical task of the numerical analysis and linear algebra domains. For these two benchmarks, we do not provide hardware implementations and results.

Table 7.6: Experimental results of approximate AxFPU-based CNNs on TSMC 65-nm standard-cell.

Design	2× ConvLayers on MNIST			4× ConvLayers on CIFAR-10		
	Area (%)[1]	Power (%)[1]	Accuracy (%)[2]	Area (%)[1]	Power (%)[1]	Accuracy (%)[2]
AxFPU16$\|_{1,0}$	5.8	2.8	0	5.4	2.5	0
AxFPU16$\|_{3,4}$	30.3	24.3	0	28.4	22.1	0
AxFPU16$\|_{3,8}$	33.0	26.1	0	30.9	23.5	1.8
AxFPU16$\|_{4,6}$	43.7	42.8	0	40.9	38.8	2.5
AxFPU16$\|_{5,4}$	53.5	49.9	2.7	50.2	45.2	19.8
AxFPU32$\|_{4,12}$	36.8	29.8	0	34.5	26.9	0
AxFPU32$\|_{8,16}$	59.3	54.7	0	55.6	49.6	0
AxFPU32$\|_{10,18}$	64.4	61.2	0	60.4	55.5	1.3
AxFPU32$\|_{11,19}$	67.1	63.8	0	63.4	57.6	5.4
AxFPU32$\|_{12,4}$	69.6	65.5	5.5	65.1	61.2	12.5

[1] Refers to % resource gains (relative reduction) in comparison with the accurate design.

[2] Refers to % accuracy loss.

Nevertheless, the accuracy evaluation is extremely important, considering that such benchmarks are usually implemented in software and they can be executed in processors integrating AxFPU in their floating-point units. For our experiments, we develop a generic C/C++ function for AxFPU that takes as input the approximation configuration parameters. This function is called in the Rodinia programs instead of the default accurate floating-point multiplication.

For K-means clustering, we consider the accurate 32-bit floating-point model as baseline, and examine the Erroneous Classification Ratio (ECR), i.e., the number of wrong classifications per the total number of classifications. As shown in Figure 7.9a, AxFPU achieves the classification results of the "golden" model, even when applying approximations that deliver significant resource gains (e.g., AxFPU32$\|_{8,12}$ provides 68% area gains versus the accurate multiplier). According to the derived results, for large perforation values, i.e., $P \geq 8$, the impact of rounding increases for $R \geq 14$, which results in very high ECR. Nevertheless, the resource gains and the classification accuracy of the "golden" model, which are provided by either AxFPU32$\|_{8,12}$ or AxFPU32$\|_{9,12}$, establish AxFPU as an efficient solution for such floating-point calculations. For the LU decomposition, we consider MRED as accuracy metric and the accurate 32-bit floating-point model as baseline. Figure 7.9b presents the experimental results for a large matrix size (2048×2048). Again, the proposed AxFPU achieves good accuracy results, as MRED is smaller than 0.1% for several configurations that deliver significant hardware gains.

(a)

(b)
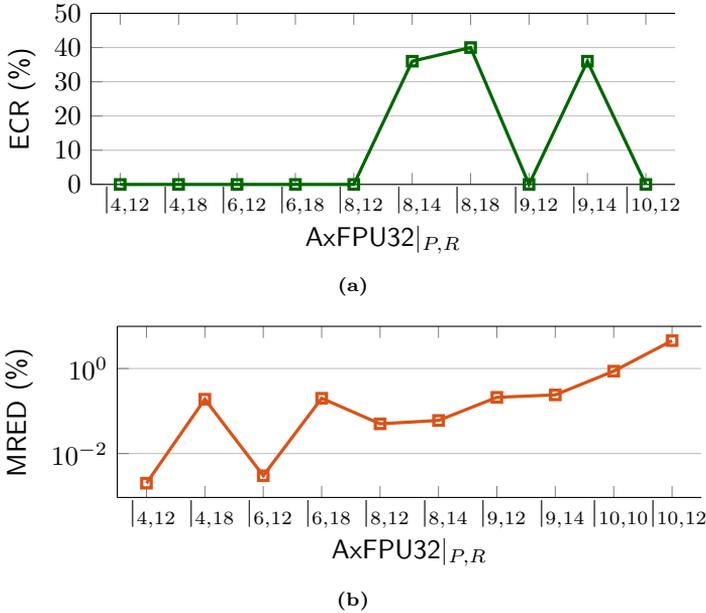
Figure 7.9: Accuracy variation of approximate AxFPU-based **(a)** K-means clustering and **(b)** LU decomposition.

7.5. Design and Evaluation of Applications with ROUP

In this section, we employ our state-of-the-art ROUP multipliers [153] in CNNs. In particular, we examine the interplay of fine-grained error resilience of CNNs in collaboration with arithmetic approximation, targeting to achieve higher energy efficiency. The large approximation space offered by the ROUP multipliers allows us to systematically explore their fine-grained distribution across the network according to different approaches. For evaluating our approximations, we employ the ResNet-8 CNN on the CIFAR-10 dataset.

7.5.1. Fine-Grained Approximate CNN Accelerators

The approximate CNNs presented in Sections 7.3-7.4 execute all their multiplications with the same approximate multiplier. Namely, each approximate CNN variant employs a single multiplier. In contrast, the current section examines the use of different approximate multipliers within the same network. More explicitly, the approximate multipliers are distributed across the network based on three approaches corresponding to the abstraction levels of the CNN architecture (convolutional layers $\rightarrow$ filters $\rightarrow$

convolution engines). Moreover, compared to our previous work on approximate CNN kernels, which are built on 16/32-bit fixed- and floating-point arithmetic, we adopt quantized network models (8-bit unsigned integer arithmetic).

To perform an extensive DSE regarding the distribution of approximate multipliers within the CNN, we employ ALWANN [303], which is a state-of-the-art framework for generating approximate CNN hardware accelerators without retraining. Towards larger design/approximation space, we extend it with our approximation approaches and the ROUP multiplication library. The proposed framework is called *MAx-DNN* and assigns different ROUP multipliers either in each convolutional layer, filter, or convolution engine (kernel). On the other hand, ALWANN inserts the same multipliers in each convolutional layer (multipliers may differ among layers). In the next paragraphs, we make a brief introduction to ALWANN, and then we introduce our approximation approaches.

ALWANN [303]: The framework takes as input the trained (frozen) network model in protobuf format, a library of approximate multipliers, and architecture constraints for the hardware accelerator (e.g., pipelined or power-gated mode, number of approximate units). It implements accurate addition and approximate multiplication, as well as one approximation type per convolutional layer. Moreover, to improve the accuracy without re-training, the network weights are tuned/updated according to the multipliers' properties. The approximate networks, labeled as AxNNs, are modified versions of the initial model and satisfy the user constraints for the architecture of the accelerator.

To enable the ALWANN functionalities, the TensorFlow framework is extended to support approximate quantized layers. In particular, a new operator is created that replaces the default `QuantizedConv2D` layers with `AxConv2D` layers. This operator allows to specify which approximate multipliers to employ via the `AxMult(str)` parameter (a C/C++ model of the multipliers is necessary), and optionally, to use the weight tuning feature via `AxTune(bool)`. The frozen model is processed by the TensorFlow's tool for graph transform, which inserts the `AxConv2D` layers, and then, the Pareto-optimal AxNNs are extracted by the NSGA-II algorithm.

In our work, we use ALWANN's infrastructure and introduce new approximation approaches regarding the distribution of the approximate multiplications across the network. The toolflow and architecture of MAx-DNN is illustrated in Figure 7.10. Compared to ALWANN, we modify the `AxConv2D` TensorFlow operator to support our approximation approaches at different CNN abstraction levels, i.e., layer, filter, convolution engine, and we also employ the state-of-the-art ROUP library of approximate multipliers. Below, we discuss our approximation approaches, targeting to explore the approximation space of the CNNs and identify the best approximation opportunities within the network.
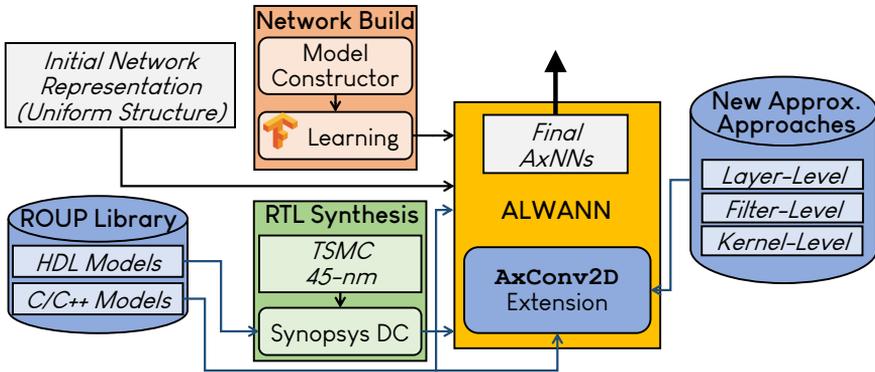
Figure 7.10: The toolflow and architecture of the MAx-DNN framework (extension of ALWANN [303]) that applies fine-grained CNN approximation. Max-DNN distributes the state-of-the-art ROUP multipliers across the network based on differing approximation approaches.

Layer-Level Approximate Multiplication: The first approach, illustrated in Figure 7.11a, aims to create approximate convolutional layers, in which all the multiplication operations are executed with the same approximate multiplier. This approach is the straightforward one, where the approximations are distributed across the network by assigning an approximate multiplier, either the same (uniform distribution) or different (non-uniform distribution), to each convolutional layer. In MAx-DNN, we create various layer-level approximate variants by using different ROUP configurations among the convolutional layers.

Filter-Level Approximate Multiplication: Our second approximation approach, illustrated in Figure 7.11b, creates different approximate filters within each convolutional layer of the CNN. Namely, we use different ROUP multipliers in each filter of the layer, contrary to the first approach, where all the filters of a layer have the same ROUP multiplier. MAx-DNN implements this approach by creating groups of filters and assigning them ROUP multipliers.

Kernel-Level Approximate Multiplication: In the third approximation approach, we proceed deeper in the CNN architecture and examine separately the multiplications of each convolution engine. As a result, the convolutions of each filter are performed with different ROUP multipliers. Figure 7.11c illustrates the three proposed flavors of this approach: the channel flavor, where all the multiplications of the convolution kernel are performed with the same multiplier, and the row-/column flavor, where different multipliers are employed for each kernel's row/column.
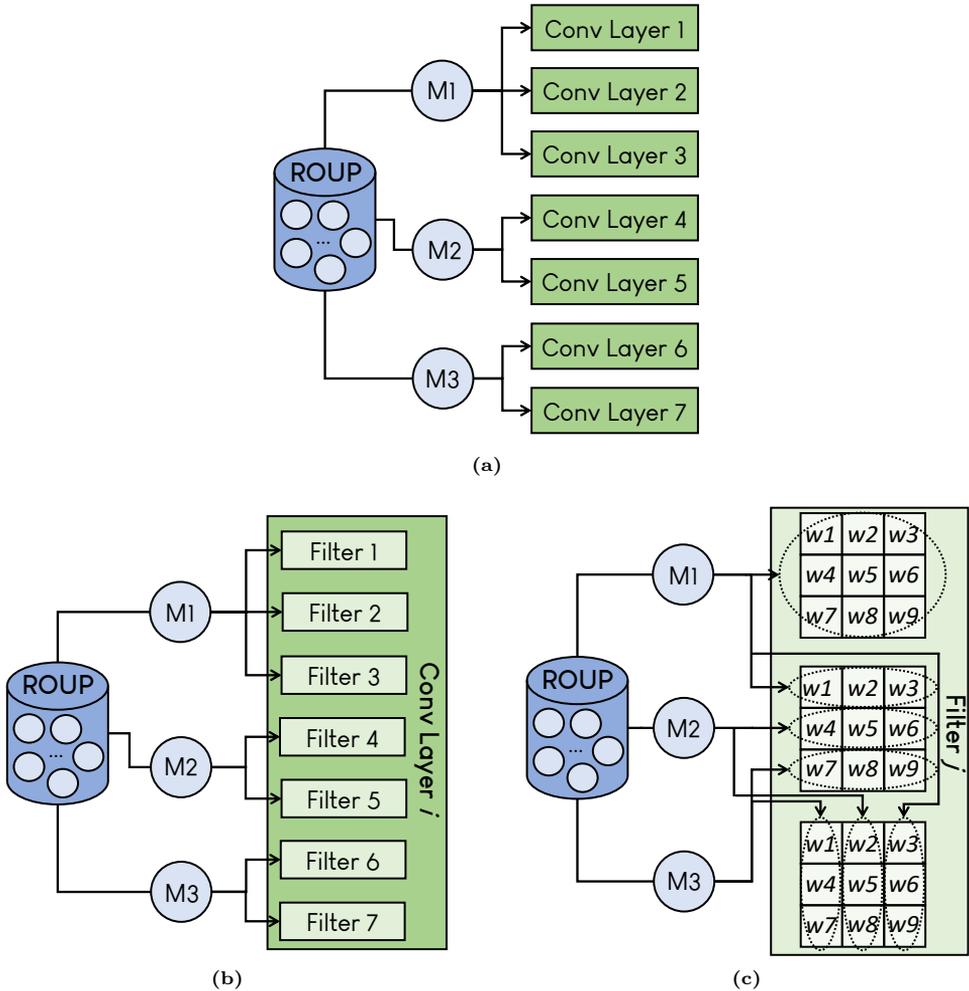
Figure 7.11: Fine-grained non-uniform CNN approximation of the MAx-DNN framework at **(a)** layer level, **(b)** filter level, and **(c)** kernel level.

Experimental Evaluation

For evaluating the ROUP-based MAx-DNN framework, we employ the ResNet-8 CNN [308] and the open-source ALWANN framework [303]. ResNet-8 is trained with quantization on the CIFAR-10 dataset [315], achieving 83% classification accuracy. The energy consumption of the approximate CNN accelerators is estimated with the model used in Section 7.4.2, which is also used in ALWANN. According to this model, the energy consumption is estimated based on the energy of the multipliers. Targeting

to standard-cell ASIC technology, we implement the ROUP multipliers on the TSMC 45-nm library using industrial-strength tools, i.e., Synopsys Design Compiler for synthesis and Synopsys PrimeTime for measuring power. Next, we calculate the energy of each convolutional layer based on the number of multiplications and the energy of the multiplication circuits. To estimate the energy of the entire CNN, we accumulate the individual energies of the layers. Our evaluation is performed in the following three stages: (i) study of CNN layer's error resilience, (ii) exploration of the CNN's approximation space, and (iii) comparison to the original ALWANN approximation framework and its approximate multipliers.

At first, we examine the error sensitivity of the convolutional layers, in an effort to understand which layers are offered for approximations. For this experiment, we pick three ROUP multipliers with different approximation strength, i.e., "low", "medium", and "high", labeled as $ROUP_L$, $ROUP_M$, and $ROUP_H$, respectively. Figure 7.12a illustrates the accuracy scaling when using these multipliers only in the m-th convolutional layer ($m = 1, 2, \ldots 7$). Regardless of the approximation strength, it is shown that approximating one of the first layers results in remarkable accuracy loss ($m = 0$ shows the baseline model with 83% accuracy). This is more evident in the $ROUP_H$ configurations, where significant computation errors are inserted. In this case, when approximating one of the layers 4–7, the accuracy loss is decreased and stabilized around 8%. Figure 7.12b, depicts the accuracy scaling when approximating the first m layers. As expected, the accuracy loss is increased with respect to the number of approximate layers, however, we again notice the error resilience of the last layers. Specifically, the accuracy loss is slowing down when extending the approximation after the 4-th layer. Another important outcome from this exploration is the negligible accuracy loss of the $ROUP_L$ and $ROUP_M$ configurations, regardless of which and how many layers are approximated. In exchange for such small accuracy loss, these multipliers provide increased energy gains compared to their accurate design (around 10% and 20%, respectively).

Subsequently, we perform an extensive design space exploration on the proposed approaches, involving various ROUP multipliers, to extract their most prominent configurations in terms of accuracy and energy. For each approach, several multiplication replacements and combinations are examined. In Figure 7.13, we present all the configurations that deliver at least 75% classification accuracy, i.e., up to 8% accuracy loss compared to the baseline quantized model. As shown in the upper left segment of the plot, multiple configurations deliver negligible accuracy loss compared to the quantized model, which ranges from 0.02% to 1%, while improving the energy efficiency due to using the ROUP multipliers. Therefore, our approaches can satisfy near-zero accuracy loss with more energy-efficient computing. Regarding the efficiency of each approach, the Pareto front is formed almost exclusively from the kernel- and filter-level configurations. For the same accuracy loss, these configura-
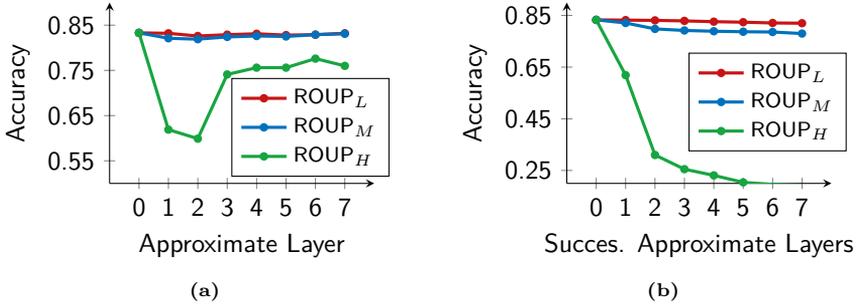
Figure 7.12: The scaling of ResNet-8 accuracy with respect to the layers approximated: **(a)** only the m-th layer (e.g., only the 5th), and **(b)** the first m layers (e.g., layers 1 to 5). "Layer=0" denotes the baseline quantized model.
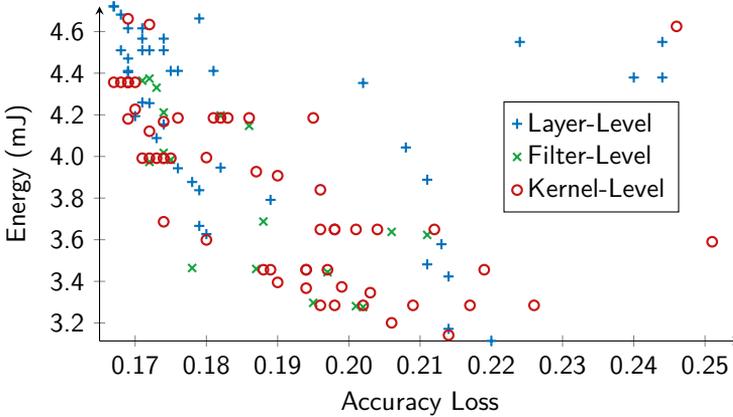
Figure 7.13: Pareto analysis for the approximate MAx-DNN-based ResNet-8 CNNs (with ROUP multipliers) considering energy consumption and accuracy loss.

tions provide better energy than the straightforward layer-level approach, which has to sacrifice a large amount of accuracy, i.e., more than 40%, to deliver this energy efficiency.

Table 7.7 compares the Pareto-front configurations with ALWANN networks employing the EvoApprox8b multipliers [190]. The proposed designs deliver better accuracy, as the average loss of the EvoApprox8b configurations is ~23%, while in terms of energy, they provide 2× gains. This comparison highlights the advantage of studying the approximations at a lower CNN abstraction level, i.e., filter or kernel. Namely, the non-uniform fine-grained use of multipliers with different approximation degree

Table 7.7: Experimental results of approximate ROUP-based ResNet-8 CNNs on TSMC 45-nm standard-cell.

Approximation Approach & Configuration		Energy (%)[1]	Accuracy (%)[2]
MAx-DNN/ROUP	FLAM-3clas._2_1_1	49	18
	FLAM-3clas._2_2_1	52	20
	KLAM-chan._1_0_1	46	17
	KLAM-chan._2_0_2	53	21
	KLAM-chan._1_1_2	50	19
	KLAM-chan._2_1_2	54	21
	KLAM-row_2_1_1	50	19
	KLAM-row_2_1_2	52	20
ALWANN [190, 303]	Evo_mul8_2AC	23	20
	Evo_mul8u_2HH	23	23
	Evo_mul8u_NGR	32	23
	Evo_mul8u_ZFB	39	23
	Evo_mul8u_7C1	20	24

[1] Refers to % energy gain (relative reduction) in comparison with the accurate design.

[2] Refers to total accuracy loss (the baseline quantized model already has 17% loss).

outperforms the straightforward approximation, which either selects the same approximate multiplier for the entire network or assigns an approximate multiplier per layer. At the same time, the use of numerous approximate multipliers based on different approximation approaches significantly expands the design space. As a result, there is increased design flexibility, which allows to efficiently handle various accuracy constraints.

7.6. Conclusion

In this chapter, we developed various approximate DSP and AI ASIC/FPGA-based accelerators that integrate the Dissertation's approximate circuits. Our goal was twofold: (i) to evaluate our approximate multipliers in real-world error-resilient applications, and (ii) to quantify the resource gains and examine the accuracy of approximate DSP and AI hardware accelerators. To facilitate the evaluation, as well as examine various design scenarios and combinations, we proposed a software/hardware design methodology, which is based on design space exploration involving arithmetic, algorithms and approximation techniques/configurations. Based on our methodology, we designed several approximate variants of accelerators for 1D/2D signal process-

Table 7.8: Summarized results of Dissertation's approximate DSP & AI hardware accelerators.

Application	Approximation	Accelerator	Gain	Accuracy
Sobel Edge Detector	RAD, AxFXU	65-nm ASIC	58% energy	CER $> 98.5\%$
FIR Filter	RAD, AxFXU	65-nm ASIC	34% energy	MRED $= 3.6\%$
Matrix Multiplication	RAD, AxFXU	65-nm ASIC	71% energy	MRED $= 0.9\%$
QAM Demodulation	RAD	ZCU106 FPGA	65% LUTs	BER $\in [10^{-1}, 10^{-4}]$
ShipDetect CNN	RAD	Zynq-7020 FPGA	38% LUTs	0.2% loss
Gaussian Blurring	AxFPU16	65-nm ASIC	40% power	PSNR $= 51$dB
Gaussian Blurring	AxFPU32	65-nm ASIC	57% power	PSNR $= 55$dB
CNN/MNIST	AxFPU16	65-nm ASIC	49% power	2.7% loss
CNN/MNIST	AxFPU32	65-nm ASIC	66% power	5.5% loss
CNN/CIFAR-10	AxFPU16	65-nm ASIC	39% power	2.5% loss
CNN/CIFAR-10	AxFPU32	65-nm ASIC	58% power	5.4% loss
K-Means Clustering	AxFPU32	–	–	ECR $= 0\%$
LU Decomposition	AxFPU32	–	–	MRED $\in [0.1, 0.8]\%$
ResNet-8/CIFAR-10	ROUP	45-nm ASIC	54% energy	4% loss

ing and CNNs. Regarding the HDL development of the accelerators, we employed various state-of-the-art parallelization techniques that were combined with our arithmetic approximations. The most important experimental results are summarized in Table 7.8. In terms of quality of results, the approximations attain small accuracy loss, which is tunable due to the various approximation configurations that are offered by our design families. More explicitly, we achieve typical values for the quality metrics of image processing (convolutions) and small errors in applications involving multiply-accumulate operations (signal filtering, matrix multiplication and LU decomposition). Our approximations also provide promising accuracy results in applications/algorithms from other domains, such as telecommunications (QAM demodulation) and machine learning (K-means algorithm). Moreover, our approximate CNNs deliver small accuracy loss in various arithmetic formats (16-bit fixed-point, 32/16-bit floating-point, 8-bit quantized integer), which can be tuned even to provide the baseline accuracy. At the accelerator level, we achieve remarkable resource gains on both ASIC and FPGA. Depending the application, we achieve power/energy gains in the range 30%–70% on ASIC technology, as well as valuable logic reduction, i.e., up to 65% in FPGA's LUTs and up to 60%-70% in ASIC's logic-cell area.

Part II.

Design Methodologies for Embedded Computing

Prologue

Dissertation's Part II focuses on higher design abstraction layers and proposes design methodologies for the efficient mapping and acceleration of DSP and AI algorithms on space-grade FPGAs and embedded heterogeneous VPUs. Chapter 8 regards the new European space-grade FPGAs, which insert several bottlenecks towards efficient implementation, mainly because the tool is new and the performance is lower than that of commercial FPGAs. We show that the systematic development and exploration of all the tool settings can provide sufficient acceleration of high-performance DSP algorithms. Chapter 9 regards the multi-core VPUs, which are characterized by increased SoC complexity and decreased performance due to their limited power. We show that the systematic development along with high- and low-level embedded design techniques can provide sufficient acceleration of demanding DSP and AI algorithms.

The work presented in Part II was mainly performed in the context of research activities of the European Space Agency (ESA). Namely, it is product of groups of people that took part in the respective activities and publications. The contribution of the Dissertation's author is as follows. For Chapter 8, he contributed to the methodology, FPGA benchmarking, tool exploration, hardware/software development for FPGA execution, and experimental analysis/evaluation. For Chapter 9, he contributed to the methodology, VPU development/coding, and experimental evaluation/analysis.

Chapter 8

DSP Acceleration with New Space-Grade FPGA Devices & Tools

*The advent of space applications with increased computational requirements pushes the space industry to consider specialized processors for high-performance on-board data processing. The excellent performance-per-Watt ratio of modern Field-Programmable Gate Arrays (FPGAs) establishes them as a promising device solution, which outperforms the general-purpose Central Processing Units (CPUs). Especially for demanding Digital Signal Processing (DSP) algorithms, the FPGAs offer increased parallelization (to provide high processing throughput) and interfacing capabilities (to handle various sensors and provide high I/O rate). In a relatively limited market of space-grade FPGAs, the new European FPGA family of NanoXplore offers such novel radiation-hardened solutions. Nevertheless, the full exploitation of these new space-grade FPGAs, considering also that they lack in performance compared to their commercial counterparts, requires a systematic design approach. Towards the efficient mapping and acceleration of high-performance DSP algorithms on new space-grade FPGAs, this chapter devises and applies a methodology to thoroughly examine the implementation. We focus on NG-Large, i.e., the largest 65nm radiation-hardened SRAM FPGA of NanoXplore, and we accelerate demanding computer vision kernels for feature detection and depth extraction. Our design approach comprises a number of customized steps that perform exhaustive exploration of all the tool settings and generate a variety of results, which are directly compared to results from well-established FPGA vendors. The experimental evaluation shows that NG-Large provides sufficient performance, e.g., 5–10 FPS for feature detection on MPixel images, while the resource utilization is balanced and comparable to that of the other FPGA vendors. This chapter is based on our **publication** in [**319**].*

8.1. Introduction

In the last decade, the Field-Programmable Gate Arrays (FPGAs) have been established as state-of-the-art devices in the semiconductor market. Their attractive performance-per-Watt ratio marks a new era, in which they are not used only for prototyping or interfacing, but also for accelerating demanding workloads [282–286, 291, 294, 295]. Modern FPGAs still do not match the power consumption of Application-Specific Integrated Circuits (ASICs) [279], however, their re-configurability and high-performance constitute them as first-class devices for acceleration in data centers [20] and embedded systems [3]. The throughput rate of FPGAs is significantly better than that of the Central Processing Units (CPUs), while in terms of power consumption, they outperform the Graphics Processing Units (GPUs) [3, 4, 278, 320]. The FPGAs are used in various fields (such as telecommunications, robotics, space) for accelerating Digital Signal Processing (DSP) functions [291–295, 321], e.g., for image/video processing and signal filtering, and more recently, for deploying Artificial Intelligence (AI) [282–286], e.g., neural networks and machine learning algorithms.

The space industry is one of the communities seeking for high-performance embedded platforms to handle the increased computational demands and fast data transfers of modern space applications. The enhanced performance of FPGAs facilitates on-board/embedded computing in space, e.g., for Earth Observation (EO) and Vision-Based Navigation (VBN) tasks, decreasing the need for downlink transmission of sensor data to the ground stations. As a result, space-grade and Commercial-Off-The-Shelf (COTS) FPGAs are constantly being evaluated [322–329] as on-board accelerators and framing processors. When radiation, thermal and vibration resilience are of utmost importance, space-grade FPGAs are used instead of their COTS counterparts to achieve increased reliability. The literature also includes numerous works with space avionics co-processing architectures that include FPGAs [330–332]. In terms of algorithms, besides accelerating DSP, the FPGAs are used for implementing data transcoding for instruments/sensors (e.g., via SpaceWire/SpaceFibre [333]) and data compression [334, 335].

The space-grade FPGAs are either Radiation Hardened (RH) or Radiation Tolerant (RT). Their main difference is that RH FPGAs are radiation hardened by design, i.e., they are fabricated on special technology to endure in radiation environments, while RT FPGAs are fabricated to operate under lower radiation values (usually along with fault-tolerant techniques to ensure reliable operation). Nevertheless, the space-grade FPGAs come with some penalties. Besides their increased cost, they are slower and offer decreased design flexibility (e.g., in terms of tool options, vendor IPs) than their COTS counterparts [336, 337]. In any case, they still outper-

form RH CPUs such as the PowerPC-based RAD750 (12W@200MHz) [338] and the LEON2-based AT697F (1W@100MHz) [339]. Currently, the market offers few space-grade FPGAs, from which the most well-established are categorized per vendor as follows:

- Xilinx [340]: Virtex-4QV (SRAM, 90nm), Virtex-5QV (SRAM, 65nm), RT Kintex UltraScale (SRAM, 20nm).

- Microsemi [341]: RTSX-SU (anti-fuse, 250nm), RT ProASIC3 (flash, 130nm), RTAX-S/SL (anti-fuse, 150nm). RTG4 (flash, 65nm), RT PolarFire (SONOS, 28nm).

- Atmel [342]: AT40K (SRAM, 350nm), ATF280 (SRAM, 180nm).

All the above FPGAs vary with respect to fabrication technology, capacity, performance capabilities and radiation resilience. Very recently, there are new promising additions in this limited pool of space-grade FPGAs. NanoXplore [343] provides the first European RH high-density FPGAs [344, 345], also known as *BRAVE*, i.e., Big Re-programmable Array for Versatile Environments. The BRAVE FPGAs have attracted the interest of the space community and are expected to play a key role in upcoming space missions, especially in Europe. This new space-grade FPGA family includes low-end and high-end RH SRAM-based chips, i.e., NG-Medium (65nm), NG-Large (65nm) and NG-Ultra (28nm), as well as software tools for end-to-end development and seamless re-configuration. NG-Medium is the first FPGA of the BRAVE series and was introduced in 2016 along with the initial tool versions. The BRAVE FPGAs integrate all the traditional FPGA programmable logic resources, i.e., Look-Up-Tables (LUTs), D Flip-Flops (DFFs), Carry Units (CYs), Digital Signal Processors (DSPs), RAM Blocks (RAMBs), and Register Files (RFs). Additionally, NG-Large and NG-Ultra integrate the single-core ARM Cortex-R5 and quad-core ARM Cortex-R52 processors, respectively, with the latter implementing the DAHLIA System-on-Chip (SoC). Furthermore, the BRAVE FPGAs include features that are essential for on-board embedded computing in space, such as the SpaceWire interface for fast I/O data transfers and memory scrubbing to ensure continuous error-free functionality. The BRAVE FPGAs can be configured via multiple interfaces (JTAG, SPI/flash, SpaceWire).

The efficient utilization of a new FPGA family such as BRAVE, as well as the full exploitation of the associated software tools, require a systematic and disciplined approach. Moreover, the developer needs to surpass the bottlenecks and limitations of new tools to provide sufficient resource utilization and/or performance for compute-intensive DSP. The development becomes even more challenging due to the special features of the space-grade FPGAs compared to the commercial ones (e.g., lower performance). For these reasons, the European Space Agency (ESA) is supporting a set

of research activities[1], which involve hardware benchmarking on the BRAVE chips/boards and testing of the BRAVE software tools. These activities aim to improve the new space-grade devices and tools and evaluate the suitability of the BRAVE solution as on-board data processor. In the context of these activities, we develop a methodology to support the design on the BRAVE FPGAs and evaluate the devices/tools compared to solutions provided by well-established FPGA vendors. We note that even though our methodology is applied in space-grade FPGAs, it can be adopted to aid the design on any FPGA platform or perform benchmarking/evaluation among different FPGAs, either space-grade or COTS. Moreover, our goal is not to design new parallel DSP architectures. In contrast, we employ DSP hardware kernels mainly from the Computer Vision (CV) field, which were developed in past ESA activities by Lentaris *et al.* [326–329][2], and we accelerate them on the BRAVE technology. More explicitly, in a systematic manner, we deploy these high-performance kernels on NanoXplore's FPGAs and other FPGAs of the market with similar architecture (mentioned as "3rd-party"), and we assess the software tools and the underlying hardware.

The **contribution** of this chapter is summarized as follows:

(i) We highlight the bottlenecks that arise when immigrating to new FPGA technologies, as well as the significance of the systematic exploration of the tool settings and device capabilities towards improved resource utilization and performance.

(ii) We propose a methodology for the efficient deployment of high-performance DSP kernels (developed on well-established technologies) on the programmable logic of new FPGAs.

(iii) We propose a methodology for the assessment and testing of FPGA tools and devices.

(iv) We evaluate NanoXplore's new space-grade FPGAs as candidate on-board accelerators for future space missions by testing all the relevant features and comparing them to well-established competitor devices.

The remainder of this chapter is organized as follows. Section 8.2 overviews the market's space-grade FPGAs and presents the new BRAVE devices and tools. Section 8.3 introduces our design and assessment methodology. Section 8.4 presents the CV kernels and reports issues and tool bugs that arose during their implementation with the early tool versions. Section 8.5 conducts the experimental evaluation, which

---

[1]**ESA QUEENS1/QUEENS2/QUEENS3:**
  4000119331/17/NL/PS, 4000128041/19/NL/AR/va, 4000134874/21/NL/AR/va
[2]Special thanks to Dr. G. Lentaris *et al.* from NTUA for providing the DSP hardware kernels.

includes various FPGA results and comparisons. Section 8.6 demonstrates the execution of the CV kernels on the BRAVE hardware. Finally, Section 8.7 draws the conclusions.

8.2. Background

8.2.1. The Landscape of Space-Grade FPGAs

Table 8.1 presents an overview of the most prominent space-grade FPGAs, including the BRAVE ones. The Atmel FPGAs are omitted here due to their limited on-chip resources. We note that Xilinx has recently announced a new space-grade FPGA, named XQR Versal [346], which includes additional features such as ARM processors and AI engines. The examined FPGAs can be categorized in three classes, with each class including chips with similar resources but from different vendors, e.g., {Virtex-5QV, RTG4, NG-Large}. BRAVE is the only FPGA family that offers hard embedded processors, facilitating hardware/software co-design. In terms of radiation resilience, similar to Virtex-5QV, the advantage of BRAVE is their RH SRAM-based chips. More specifically, they are fabricated with 12-transistor (12-T) configuration memory cells outperforming the simple SRAM cells and competing with anti-fuse and flash technologies. This is extremely important, considering that, for example, Virtex-4QV would require full Triple Modular Redundancy (TMR) and configuration memory scrubbing to achieve reliable operation.

Several of these FPGAs have been used in space missions [347–350]. More details about their use in past, present and future missions can be found in our publication in [319]. At research level, the space-grade FPGAs are being evaluated for the implementation of compute-intensive functions and novel space applications, while they are also examined as part of co-processing architectures [330–332]. Next, we present some representative research works involving space-grade FPGAs.

In [351], the CCSDS 1.2.3 standard for compressing hyperspectral images is implemented on Virtex-4QV, achieving real-time compression for sensors such as AVIRIS ($680\times512\times224$ image size), while utilizing $1/3$ of the chip resources and consuming limited power. Similarly, the Virtex-5QV and RTG4 FPGAs are used for the implementation of the SHyLoC 2.0 CCSDS 121 and 123 lossless compression standards [334], providing up to 138 and 81 MSamples/s, respectively, for the AVIRIS sensor. In [352], the authors implement a single-chip payload data processing unit on the Virtex-5QV FPGA, which integrates both the instrument system supervisor and data processing functions. The proposed architecture supports self-configuration management and mitigation techniques to provide fault-tolerance. In [353], the new SCCC-X

Table 8.1: Overview of market's space-grade FPGAs.

Vendor	FPGA	Rad. Resilience	Technology	TID / SEL[1]	Processor
Xilinx	Virtex-4QV	Tolerant	SRAM, 90nm	300 / 125	–
	Virtex-5QV	Hard	SRAM, 65nm	1000 / 125	–
	RT Kintex US	Tolerant	SRAM, 20nm	100 / 80	–
Microsemi	RTAX-S/SL	Tolerant	anti-fuse, 150nm	300 / 117	–
	RTG4	Tolerant	flash, 65nm	100 / 103	–
	RT PolarFire	Tolerant	SONOS, 28nm	100 / 80	–
NanoXplore	NG-Medium	Hard	SRAM, 65nm	100 / 60	–
	NG-Large	Hard	SRAM, 65nm	100 / 60	Cortex-R5
	NG-Ultra	Hard	SRAM, 28nm	50 / 60	Cortex-R52

Vendor	FPGA	Logic Cells	Total RAM	DSPs	User I/Os	SERDES[2]
Xilinx	Virtex-4QV	55–200K	4.1–9.9Mb	32–192	640–960	–
	Virtex-5QV	131K	12.3Mb	320	836	18@4.3Gbps
	RT Kintex US	726K	38Mb	2.7K	620	32@12.5Gbps
Microsemi	RTAX-S/SL	2–40K	0.05–0.5Mb	0–120	198–840	–
	RTG4	151K	5Mb	462	720	24@3.1Gbps
	RT PolarFire	481K	33Mb	1.4K	584	24@10.3Gbps
NanoXplore	NG-Medium	34K	2.9Mb	112	566	–
	NG-Large	137K	10.1Mb	384	684	24@6.3Gbps
	NG-Ultra	536K	34Mb	1.3K	744	32@12.5Gbps

[1] Total ionizing dose in Krad (Si) and single-event latch-up immunity to linear energy transfer in MeV-cm^2/mg.

[2] High-performance serialization/deserialization transceivers.

telemetry transmitter, which is an extension of the CCSDS 131.2-B-1 standard, is implemented and evaluated on RT Kintex UltraScale and RTG4, delivering more than 450 MSym/s and 250 MSym/s, respectively. Very recently, radiation-tolerant FPGA-based platforms for AI applications have gained momentum. In [354], the authors propose a deep learning architecture for RT Kintex UltraScale, which is based on Xilinx's deep learning processing unit and the TMR MicroBlaze subsystem. The authors of [355] use the VectorBlox software development kit to deploy AI models on a matrix processor implemented on the RT PolarFire FPGA.

The literature includes several works with the BRAVE FPGAs. In [356], the re-configuration capabilities of NG-Medium via the SpaceWire interface are evaluated, while in [325], benchmarking results for NG-Medium are reported, and re-configuration scenarios with different algorithms are examined. Moreover, NG-Medium is used for the implementation of dense stereo vision algorithms [329]. Regarding NG-Large, pre-

liminary benchmarking results are provided in [357]. In [334], both NG-Medium and NG-Large FPGAs are evaluated for hyperspectral image compression.

8.2.2. The NanoXplore Space-Grade FPGAs and Tools

The NG-Large FPGA

In this Dissertation, we focus on NanoXplore's NG-Large FPGA, i.e., the second chip of the BRAVE series, thus, we analyze its architectural details and most important features. NG-Large is a radiation-hardened-by-design SRAM-based FPGA that is manufactured on the 65nm STM C65-Space process technology [344]. Below, we analyze its fabric architecture, which is illustrated in Figure 8.1.

The NG-Large die features 7 rows of 48 Tiles, with a single Tile consisting of 384 4-input LUTs, 384 DFFs, 96 1-bit CYs, 24 X-LUTs, and 2 64×16-bit RFs. Each Tile includes 384 Functional Elements (FEs), with a single FE integrating 1 LUT–DFF pair and additional logic for CYs, X-LUTs and RFs. The CYs of NG-Large combine 1 LUT with carry propagation logic to support up to 96-bit carry chain. To allow the implementation of up to 16-input logic functions, 4 LUTs drive the inputs of another LUT (called X-LUT) without routing through the interconnect network. The RF of NG-Large is a synchronous dual-port SRAM with read-only and write-only ports and optional output register. Furthermore, NG-Large features 4 rows of 48 48-Kbit RAMBs and 4 rows of 96 DSPs. Each RAMB is a true dual-port SRAM with optional output register and supports multiple memory configurations, i.e., 48K×1-bit, 24K×2-bit, 12K×4-bit, 6K×8-bit, 4K×12-bit and 2K×24-bit, as well as Error Detection and Correction (EDAC) configuration. A single DSP includes a 19×24 multiplier, a 56-bit Arithmetic Logic Unit (ALU), an 18-bit pre-adder, as well as pipeline registers. The DSPs operate either in signed or unsigned mode, and can cascade up to 96 blocks. Finally, NG-Large includes 4 Clock Generators (CKGs), one at each die corner. The CKG block includes 1 Phase-Locked Loop (PLL) and 10 Waveform Generators (WFGs), i.e., frequency dividers. Moreover, it has 4 High Speed Serial Links (HSSLs) of 6 lanes, providing up to 6.25 Gbps data rate.

Overall, the total resources of NG-Large are summarized as: 137088 LUTs, 129024 DFFs, 32256 CYs, 384 DSPs, 192 RAMBs, 672 RFs, 4 PLLs. Compared to NG-Medium, NG-Large is ~4× larger (NG-Medium has 34272 LUTs, 112 DSPs, 56 RAMBs) [344]. Correspondingly, NG-Large is ~4× smaller than NG-Ultra.
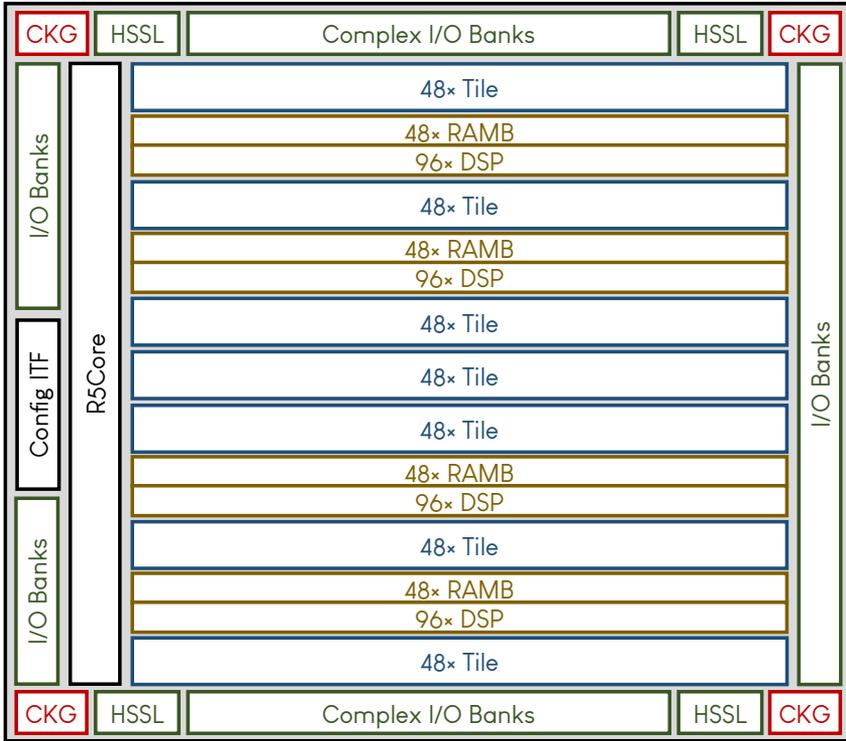
Figure 8.1: The fabric architecture of NanoXplore's space-grade NG-Large FPGA [343].

The NXmap Software Tool

NXmap is the design suite provided by NanoXplore to support all classical FPGA development stages except for simulation, which is currently performed with 3rd-party tools (ModelSim/QuestaSim). The tool supports both Verilog and VHDL hardware description languages and includes functions for timing analysis and power estimation. It is divided in two components: (i) the graphical interface, which allows the user to compile existing projects, view the floorplan and inspect the implemented design, (ii) the Python wrapper, which allows the user to build projects by compiling Python scripts with the desired tool settings and functionalities.

The Python wrapper of NXmap supports the Python syntax and structures, and it provides a plethora of NanoXplore routines/modules for each stage of the FPGA development flow. These Python routines can be categorized as follows:

- Project-related: e.g., `createProject`, `addFiles`
- Tool-related: e.g., `setOptions`

- Mapping-related: e.g., `createRegion`, `addRAMLocation`
- Stage-related: e.g., `synthesize`, `generateBitstream`
- Board-related: e.g., `addPads`, `addBanks`
- Timing-related: e.g., `createClock`, `addFalsePath`

The NXmap routines take numerous arguments as input, providing a wide range of functionalities. NXmap also offers routines for monitoring the design flow and defining the verbosity of the reports. A Python code snippet demonstrating some basic NXmap functionalities is attached in Code 8.1.

```python
# project creation
project = createProject(my_directory)
project.setVariantName('NG-LARGE')
project.setTopCellName('top_module')
project.addFiles(['top_module.vhd', 'design1.vhd', 'design2.vhd'])

# general tool settings
project.setOptions(['MappingEffort': 'Medium',
                    'RoutingEffort': 'High',
                    'DisableDSPRegisters': 'Yes',
                    'DefaultROMMapping': 'LUT',
                    'ManageUnconnectedOutputs': 'Ground'])

# custom mapping targets
project.addMappingDirective('getModels(.*mult.*)', 'MUL', 'DSP')
project.addMappingDirective('getModels(add_9u_9u)', 'ADD', 'CY')

# custom mapping locations
project.addDSPLocation('*mult_L267*', 'CGB[1x8]:L')
project.addRAMLocation('*RAM_INST0*', 'CGB[48x20]')

# clock constraint
project.createClock('getClockNet(clk)', 'clk', 40000)

# I/O signals and pads pairing
project.addPad('clk', {'location':'IO_B18D02P',
                       'standard':'LVCMOS', 'drive':'2mA'})
# project save
project.save('project.nym')

# synthesis
project.synthesize()
project.save('synthesis_netlist.vhd')

# place
project.place()
project.save('place_netlist.vhd')

# route
project.route()
```

```
project.save('route_netlist.vhd')

# reports
project.reportPorts()
project.reportInstances()

# static timing analysis
analyzer = project.createAnalyzer()
analyzer.launch()

# bitstream generation
project.generateBitstream('bitstream.nxb')
```

Code 8.1: Example of Python script for building an FPGA project in NanoXplore's NXmap tool.

8.3. Design & Assessment Methodology

In this section, we introduce a methodology for deploying DSP kernels on BRAVE FPGAs, as well as for assessing the capabilities of the tools and chips. Our methodology regards the three main stages of the typical FPGA design flow, i.e., synthesis, place & route (implementation), and bitstream generation & configuration. Currently, the methodology is executed manually by the developer, however, some segments, such as the exploration of the tool settings via Python scripting, could be performed in an automatic fashion. Moreover, this methodology is generic, namely it can be adopted to test/evaluate other FPGAs or facilitate the development with new devices and tools.

8.3.1. Synthesis of the Design

From the tool assessment perspective, our synthesis-related methodology aims to:

(i) Test the correct functionality of all the tool settings, attributes, and strategies by examining if they apply the expected functionality and if the output results of the post-synthesis simulation remain correct.

(ii) Evaluate the quality of results for different tool configurations by comparing the resource utilization of the respective synthesis netlists.

(iii) Evaluate the capability of the tool's synthesizer to efficiently map the input design on the FPGA blocks by examining the resource utilization compared to the expected (theoretical) one.

(iv) Rate the resource utilization via systematic comparisons to 3rd-party devices/-tools that are already established in the market.

From the efficient DSP deployment perspective, given a kernel developed in Hardware Description Language (HDL), our methodology aims to:

(i) Quantify the expected resource utilization of the kernel based on well-established state-of-the-art 3rd-party devices/tools.

(ii) Extract the tool configurations that result in the most efficient synthesis netlists.

(iii) Resolve the issues that arise from the tool's "immaturity" (recently released tool) or because the kernel was initially developed in other vendor's technology.

The part of our methodology targeting to synthesis is illustrated in Figure 8.2. It is divided into three main phases: the parametric configuration of the DSP kernels, the tool-level exploration, and the HDL-level exploration. Initially, we adapt the algorithmic parameters of the DSP kernel with respect to the features of the targeted BRAVE FPGA (e.g., resources, architecture of FPGA blocks), and we run syntheses on 3rd-party tools. To be more specific, we configure parameters such as the size of the input image, the size of the convolution masks, the data bit-width, the accuracy of the calculations, and the parallelization factor. For example, the bigger BRAVE devices can handle larger input images and/or increased parallelization. The next two phases aim to generate an efficient error-free synthesis netlist.

In the phase of the tool-level exploration, we perform a preliminary synthesis with the default tool settings to retrieve the "default" reports and detect potential issues. This step is also considered as test for the synthesizer's capability to automatically balance the resource utilization and generate an error-free netlist. Next, we explore all the available synthesis-related settings and assess their capability to produce a synthesis netlist according to the developer's choices and preferences. Indicatively, we test tool settings regarding the mapping effort of the synthesizer, the mapping target of the arithmetic/memory components, the DSP utilization ratio, the register duplication, and the style of the finite-state machine encodings. The settings are applied in both standalone and combinatorial fashion. In case of error or unexpected tool behaviour, we report the issue (to be resolved, if possible, via an alternative tool configuration or HDL coding). The resource utilization of the most prominent synthesis netlists is compared with that of the 3rd-party tools. In case spikes are observed, we apply different tool settings or we move on to the next phase of our methodology.

In the phase of the HDL-level exploration, we recursively decompose the kernel to smaller building blocks and test them individually. This step is very important in our methodology, as it provides an in-depth investigation of various optimization issues and/or errors, which, otherwise, would be very difficult to be detected in the entire kernel. This low-level exploration uses standard template-based coding and
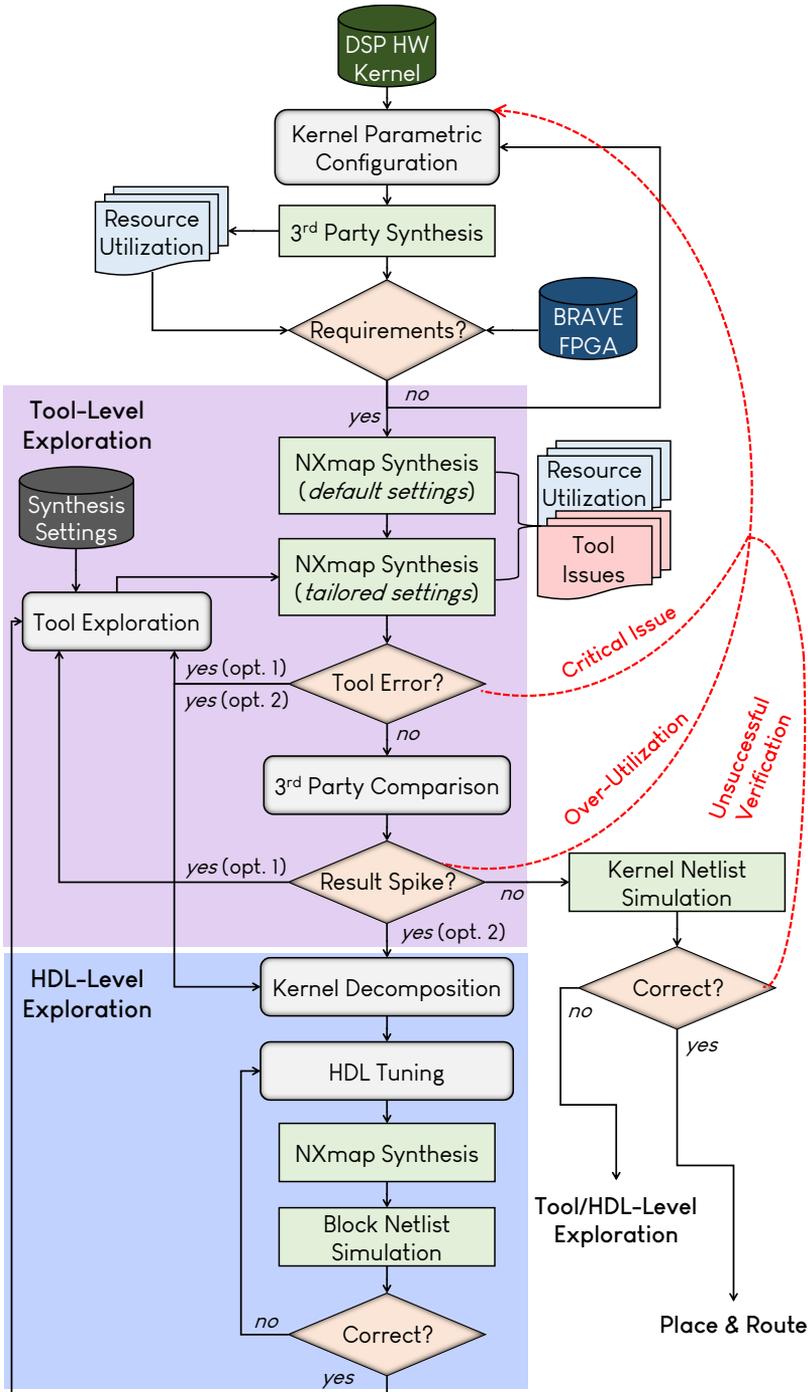
Figure 8.2: Methodology for the synthesis of DSP kernels on the new BRAVE FPGAs.

attributes/directives to express memories, finite-state machines, and arithmetic components, as well as vendor-specific HDL templates (i.e., from NanoXplore in the present case). In this phase, for every new or modified building block, we perform synthesis and functional verification via post-synthesis simulation. If we identify a type of HDL coding that generates improved results or resolves one of the arisen issues, we adopt it and return to the tool-level exploration to examine the entire kernel (also in comparison with the 3rd-party tools).

In case neither tool-level nor HDL-level exploration can provide a solution in issues such as critical tool error, resource over-utilization, and unsuccessful verification, our methodology includes feedback loops (red dashed lines in Figure 8.2). The developer can use them to return to the first phase to re-customize the algorithmic parameters, and then perform the tool-level and HDL-level exploration with a new kernel configuration.

8.3.2. Placement & Routing of the Design

A similar methodology is developed for the place & route (implementation) stage. This methodology, illustrated in Figure 8.3, inputs the error-free synthesis netlist of the DSP kernel and performs a performance-wise tool exploration, targeting to provide the best possible clock frequency. Like in synthesis, we perform a preliminary run with the default settings, and then start the exploration with tailored tool configurations. In our exploration, we employ all the settings related to the implementation, i.e., placement/routing and physical constraints.

In this phase, at first, we perform location-specific placements by specifying mapping regions on the FPGA floorplan, either at fine-grained (LUTs, DFFs) or coarse-grained (Tiles/FEs, DSPs, RAMBs) level. In parallel, we examine the efficiency of the more general placement settings, e.g., the placement effort. Regarding the routing constraints, we stress the implementation towards performance and increased routing congestion. Namely, we employ all the available timing constraints (e.g., clock constraint, timing driven option, options for setting false path or max delay) and routing settings (e.g., router effort, router mode). We note that we combine these timing/routing settings with different placement constraints. For every combination of settings, our performance-wise tool exploration examines the Static Timing Analysis (STA) reports to identify the most efficacious tool settings.

Our methodology involves systematic comparisons with the 3rd-party tools and functional and timing verification via post-place and post-route netlist simulations, as well as floorplan inspection for verification purposes. Moreover, we include feedback loops (red dashed lines in Figure 8.3) to return to synthesis if this stage cannot provide solution to issues such as critical tool error or unsuccessful verification. As synthesis and
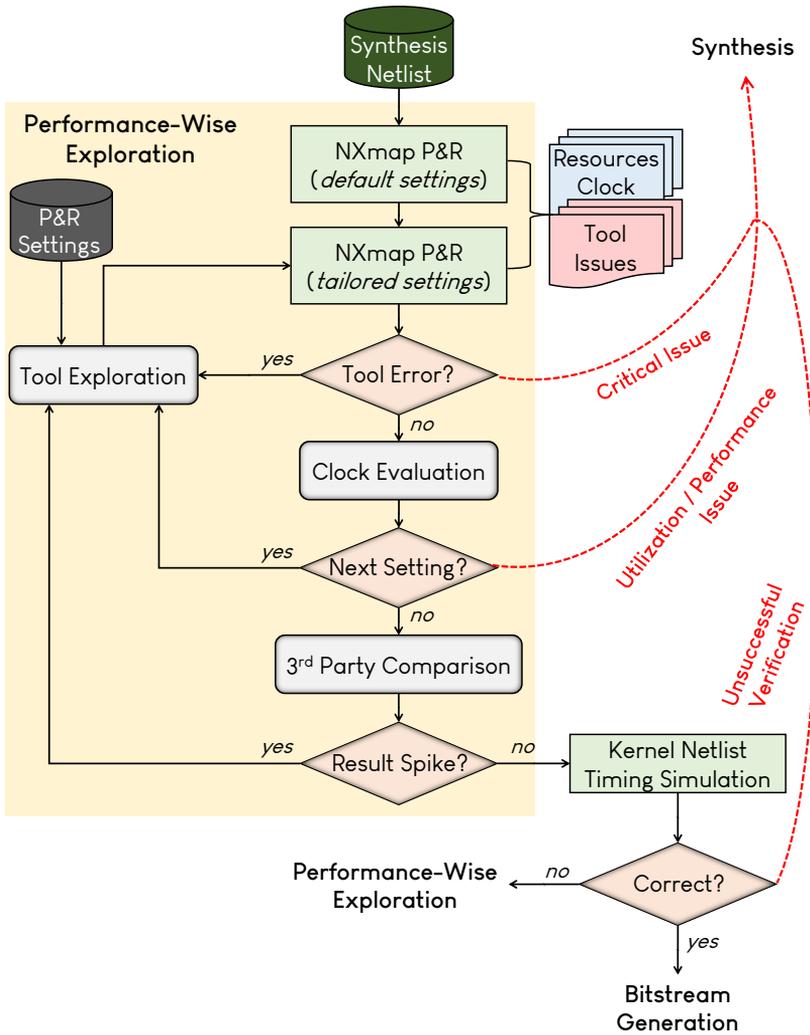
Figure 8.3: Methodology for placing & routing DSP kernels on the new BRAVE FPGAs.

place & route are tightly coupled, i.e., different synthesis netlists may lead to different place & route results, we can also return to synthesis and generate a new netlist in case of issues with performance or resource utilization.

8.3.3. Bitstream Generation and Hardware Execution

The final step of our methodology is to examine the bitstream generation, the FPGA configuration/programming, and the actual hardware execution. For these purposes,
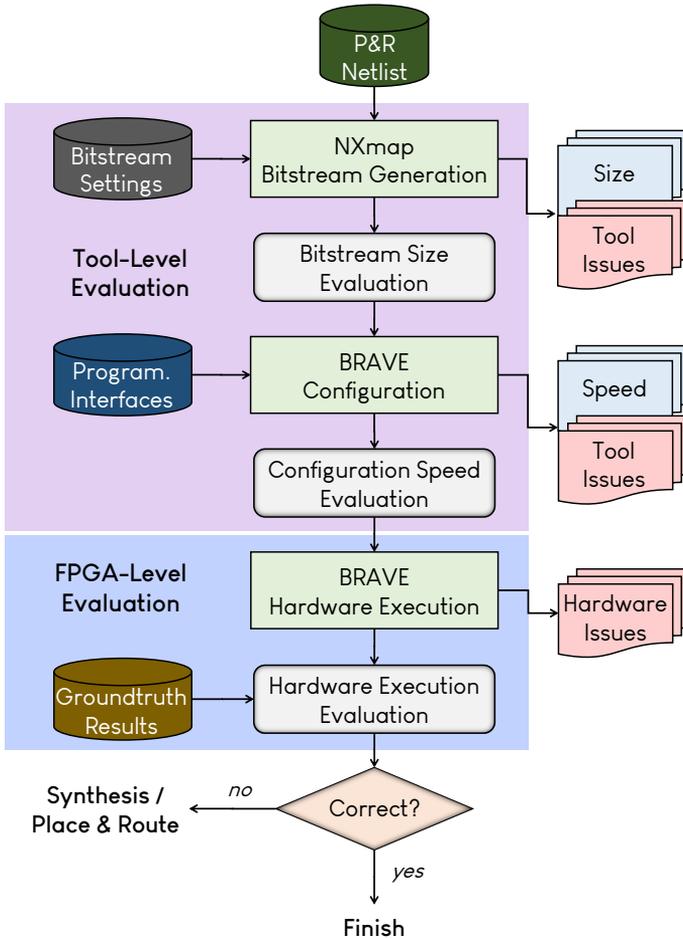
Figure 8.4: Methodology for the bitstream generation and hardware execution of DSP kernels on the new BRAVE FPGAs.

we employ the methodology illustrated in Figure 8.4. More specifically, for different relevant tool settings, we examine the correct functionality of the bitstream, the bitstream size, and the programming speeds via the available configuration interfaces (e.g., JTAG, SpaceWire, EPROM). The examination of the bitstream size and the programming speed is essential, considering that an application may require multiple bitstreams, e.g., a space mission storing on-board different bitstreams/algorithms and interchanging between them. Regarding the chip configuration, we also examine the correct functionality of the FPGA after multiple successive re-configurations, as in real-world scenarios (like in space missions), it may be required to reprogram the FPGA several times, even in a very short period of time. Finally, we perform hardware executions to verify the correct implementation of the DSP kernels on such

new chips like the BRAVE FPGAs. The validation of the results is performed by comparing the FPGA outputs with ground-truth data obtained from behavioral or post-place-&-route netlist simulations.

8.4. Porting of Computer Vision Kernels on the NG-Large FPGA

8.4.1. CV Kernels for Feature Detection and Depth Extraction

Based on our methodology, we accelerate CV hardware kernels from past ESA activities [326–329] on the space-grade NG-Large FPGA. In particular, we employ HDL kernels for feature extraction (image edges and corners), i.e., Canny Edge Detector [326] and Harris Corner Detector [327], as well as stereo matching (depth extraction in 3D scene reconstruction), i.e., GAD-Disparity [328] and Spacesweep [329]. These kernels impose increased requirements in calculations and memory resources, stressing the new tools and devices towards efficient and high-performance deployment. Their development was performed in parametric VHDL code, and thus, at compile time, we can change various parameters, e.g., the input image size, the datapath bit-width, or certain parallelization factors, as required by our methodology (see Figure 8.2). Next, we present in brief one kernel from each CV class:

- Harris Corner Detector [327]: The kernel inputs a grayscale image and outputs a set of "corners", i.e., the coordinates of the most salient features depicted in the image. The image is divided in horizontal stripes, which are processed successively in the FPGA by resource reusing. Functionally, in a loop over all pixels, the kernel employs Gaussian-smoothed products of image derivatives to define an auto-correlation matrix, whose eigenvalues capture the principal intensity changes in the examined point's neighborhood. Corners are detected on pixels whose "cornerness" is sufficiently high and exceeds that of its neighboring pixels in a 3×3 region (via non-maximum suppression). All these operations are implemented via a succession of deep, fine-grained, pixel-based pipelines connecting memories that store intermediate results.

- GAD-Disparity [328]: The kernel inputs a pair of stereo images and outputs a two-dimensional disparity map, which can be transformed to depth map via simple calculations involving the focal length and baseline of the camera. The images are divided in horizontal stripes, which are processed in the FPGA by resource reusing. In an iterative fashion, by implementing an outer loop over all examined disparities and an inner loop over all pixels, the kernel matches all 7×7 image blocks between images by minimizing a Gauss-aggregated sum of absolute differences. The on-chip

memory is mostly used to store the pixels and their corresponding intermediate aggregated values that are continuously compared/updated. On the other hand, the on-chip logic is mostly used to calculate the Gauss-aggregated values.

8.4.2. Implementation Details and Issues

The initial porting of the CV kernels on NG-Large generated many tool issues and unoptimized results. More specifically, functional errors appeared due to the HDL description of the kernels, which was based on other FPGA technologies. Moreover, the tool reported unbalanced or increased resource utilization and low clock frequency due to its "immaturity" (newly released tool). Next, we report some representative issues that appeared during the implementation. However, we note that most of them have been resolved either by newer tool versions or via custom tool configuration and alternative HDL coding.

One of our first issues was with the HDL description of the dual-port memory banks. The memory core of NG-Large's RAMB and RF is not read-first, thus, we had to add registers to pipeline the write data/address, and also perform the write operation on the falling edge of the clock. This issue was not reported by the tool, however, after investigation based on our methodology, we managed to isolate the corresponding components and detect the memory problem via post-synthesis simulation. Another incidence occurred regarding the recognition of the ROM memories. The tool always mapped our ROM arrays (declared as `constant` in VHDL) onto the LUTs, even when specifying a different mapping target, i.e., RAMB/RF. Therefore, we used the VHDL `signal` in the description of our ROMs. This issue was not functional, however, it did not allow us to deliver the desired resource optimizations, while it was difficult to detect. Moreover, we note that the tool could not implement very large multiplications onto the DSPs via cascading, thus, in certain cases, we had to modify the HDL code and split the multipliers into smaller ones (or alternatively map them onto CYs).

Regarding the mapping of the CV kernels on the underlying hardware, we observed several inefficient tool choices. Indicatively, we mention that the tool did not employ the internal registers of the DSPs and RAMBs and used the DFFs of the FEs (e.g., to implement the registers before a multiplication or after the memory output). In addition, the tool did not use all the available memory configurations to efficiently map our RAM components onto the RAMBs, resulting in large fragmentation. Finally, we note that the tool did not exploit the CY at all (e.g., for additions), resulting in DSP over-utilization and decreased performance.

Nevertheless, we note again that all the functional issues have been resolved and the CV kernels are successfully implemented, while in terms of mapping efficiency, the

tool is continuously improving. In any case, we report indicative issues that arose during the implementation to highlight the difficulty of using new tools for FPGA design, as well as to indicate the significance of accompanying the development with a methodology.

8.5. Evaluation

8.5.1. Experimental Setup

The experimental evaluation is conducted at two levels: (i) at tool level, where we evaluate the NXmap settings and examine the resource utilization of the CV kernels, and (ii) at hardware level, where we evaluate the chip's maximum frequency, the throughput of the CV kernels, and the configuration speeds/sizes of the bitstreams. Overall, we report various experimental results for the acceleration of the CV kernels on NG-Large, and we also directly compare them to the results obtained by well-established FPGA tools and devices (mentioned as "3rd-party").

From NanoXplore, besides NG-Large, which is our targeted FPGA device, we also employ its predecessor NG-Medium for comparison purposes. Regarding the 3rd-party FPGAs, we employ Xilinx Virtex-5QV XQR5VFX130 and Microsemi RTG4 RT4G150 (see Table 8.1), but we also use Cyclone III EP3CLS150 of Intel [358]. Even though the latter is not a space-grade chip, it provides similar characteristics with NG-Large (i.e., 65nm SRAM-based, 150K LUT4s, 150K DFFs, 666 9-Kbit RAMBs, 320 DSPs) and Intel is considered a well-established vendor in the FPGA market. In terms of tools, we use NanoXplore NXmap v3.5.0.4 and the 3rd-party development suites (Xilinx Vivado, Intel Quartus, Microsemi Libero).

The functional kernel verification is performed via post-synthesis simulations, post-place-&-route simulations, and actual hardware execution on realistic datasets. We use images depicting rocky Martian terrains or satellites for Harris and Canny, and synthetic stereo images depicting a rover's view on Martian terrain for GAD-Disparity and Spacesweep. As discussed in our methodology, we compare the outputs with the groundtruth results of the behavioral simulations.

8.5.2. Design Space Exploration on Market's FPGA Vendors

Before porting the CV kernels on NG-Large, we perform a design space exploration on 3rd-party tools/devices, as indicated by our methodology (see Figure 8.2). The goal is to determine the most suitable kernel configuration according to NG-Large's capacity, but also, to extract baseline results for evaluation purposes.
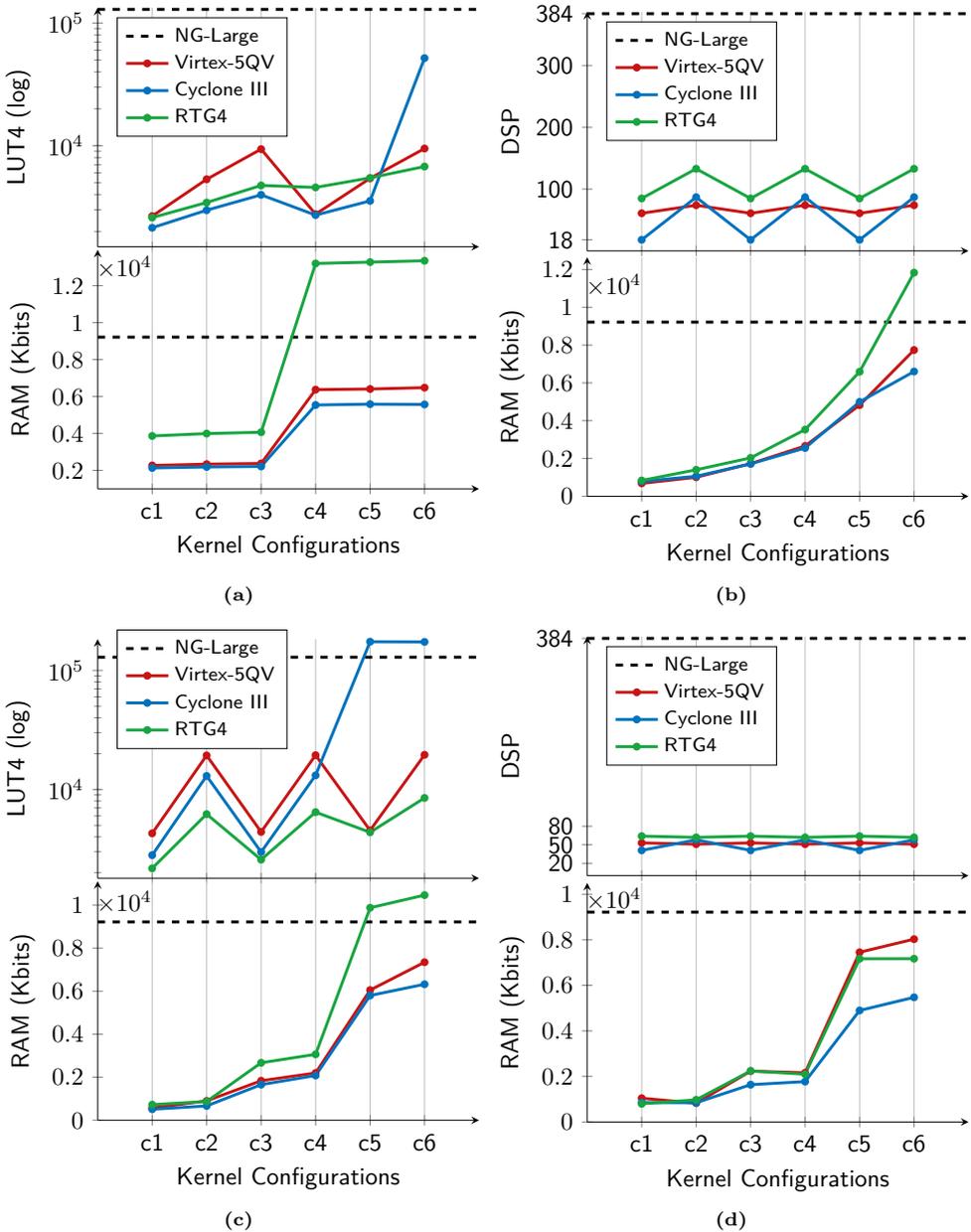
Figure 8.5: Design space exploration on FPGA vendors: synthesis results for **(a)** Canny Edge Detector [326], **(b)** Harris Corner Detector [327], **(c)** GAD-Disparity [328], **(d)** Spacesweep [329].

Figure 8.5 illustrates the resource utilization from the synthesis of 6 different kernel configurations with the 3rd-party tools. These diverse configurations impose different

constraints in calculations, datapath and input sizes, in an effort to stress the tools in both logic and memory. We note that Virtex-5QV has 6-input LUT while the other FPGAs have 4-input LUTs. To make a fair comparison, we rely on our experiments and statistically use LUT4 = 1.5×LUT6 for Virtex-5QV. Moreover, as each FPGA has different RAMB size, we present the total RAM in Kbits. As explained, even though our goal is not to compare the 3rd-party tools, we perform similar scaling in all FPGA vendors. This is quite evident in the scaling of the RAM resources of all CV kernels. The LUT and DSP utilization are tightly coupled, and they also depend on the strategy of each vendor regarding the mapping of the arithmetic/logic operations. Nevertheless, we observe some spikes, e.g., in the LUT utilization of Cyclone III for Canny (Figure 8.5a) and Harris (Figure 8.5b), and the RAM of RTGA for Canny (Figure 8.5a) and Disparity (Figure 8.5c).

Following our 3rd-party exploration, we configure the CV kernels as shown in Table 8.2 to port them on NG-Large. For example, Harris inputs a 1024×1024 8-bit image, partitioned in 1024×32 pixel stripes (image divided into 32 such blocks), performs convolutions with 7×7 14-bit masks, and outputs 32-bit corners.

8.5.3. Experimental Results

Evaluation of NXmap Tool

In this section, we evaluate the implementation of the CV kernels on NG-Large and present results from our exploration in the NXmap synthesis and place & route stages. We remind that for the default runs we do not modify any tool settings, while for the tailored runs we apply custom tool settings. Moreover, we note that our tailored tool configurations were more important in earlier versions of NXmap, which provided unoptimized results (see Section 8.4.2). In the examined tool version, the default results have been greatly improved compared to previous versions.

The upper/lower part of Table 8.3 presents the NG-Large utilization of synthesis with the default/tailored NXmap settings. Towards more balanced resource utilization, as indicated by our methodology (see Figure 8.2), we share the arithmetic operations between DSPs and CYs using the corresponding NXmap routines. For example, the majority of Harris multiplications are mapped onto CYs instead of DSPs in the default synthesis. As a result, in the tailored synthesis, we achieve a balanced utilization i.e., from 43% CY and 8% DSP to 23% CY and 22% DSP. For Disparity, the default settings deliver the same resources with our custom settings that balance the DSP and CY utilization. For Spacesweep, the tool maps by default the small memories onto RFs, thus, it is capable of making decisions with respect to the memory size. However,

Table 8.2: Final configuration of computer vision kernel's algorithmic parameters.

Kernel	Data			Mask	
	Img Size	Img Partition	I/O Bits	Size	Bits
Canny Edge Detector [326]	1024×1024	−	8/4	3×3	8×3
Harris Corner Detector [327]	1024×1024	1024×32	8/32	7×7	8×14
GAD-Disparity Stereo Vision [328]	1024×1024	1024×32	8/10	7×7	8×7
Spacesweep Stereo Vision [329]	1024×1024	1024×16	8/32	13×13	8×8

Table 8.3: Synthesis' resource utilization of computer vision kernels on NG-Large FPGA.

Kernel	Default Tool Settings[1]					
	LUT	DFF	CY	DSP	RF	RAMB
Canny	1845 (2%)	2348 (2%)	1167 (4%)	2 (1%)	0 (0%)	177 (93%)
Harris	6210 (5%)	16398 (13%)	13794 (43%)	27 (8%)	0 (0%)	69 (36%)
Disparity	1000 (1%)	3628 (3%)	4548 (15%)	4 (2%)	0 (0%)	85 (45%)
Spacesweep	5500 (5%)	10222 (8%)	6277 (20%)	50 (14%)	8 (2%)	74 (39%)

Kernel	Tailored Tool Settings[2]					
	LUT	DFF	CY	DSP	RF	RAMB
Canny	1845 (2%)	2299 (2%)	1086 (4%)	4 (2%)	0 (0%)	177 (93%)
Harris	6110 (5%)	15304 (12%)	7112 (23%)	81 (22%)	0 (0%)	69 (36%)
Disparity	1000 (1%)	3628 (3%)	4548 (15%)	4 (2%)	0 (0%)	85 (45%)
Spacesweep	5499 (5%)	10222 (8%)	6277 (20%)	50 (14%)	0 (0%)	79 (42%)

* %: utilization of NG-Large (137K LUTs, 129K DFFs, 32K CYs, 384 DSPs, 672 RFs, 192 RAMBs).
[1] Previous NXmap tool versions did not utilize CYs by default, and we had to share the arithmetic operations between DSPs and CYs to balance the utilization for our tailored synthesis.
[2] Tailored tool settings selected towards the best possible clock frequency (see Table 8.4).

we force the tool to use RAMBs even for these small memories, because according to our experiments, it results in better clock frequency.

Before proceeding to place & route, we examine how our mapping choices affect the clock frequency. Canny and Harris achieve better clock frequency when using the DSP for their multiplications. Therefore, as in Harris, we map the few multipliers of Canny onto the DSPs. On the other hand, Disparity and Spacesweep achieve better timing when the default mapping is used, as the custom mapping decreases the clock frequency by 5%. As a result, we do not adopt the custom mapping in these kernels and tune only the settings of place & route.

Next, we proceed with the place & route stage, where we apply our performance-

Table 8.4: Implementation's resource utilization of computer vision kernels on NG-Large FPGA.

Kernel	Default Tool Settings					
	LUT	DFF	CY	DSP	RAMB	MHz
Canny	1844 (2%)	2412 (2%)	1167 (4%)	2 (1%)	177 (93%)	35
Harris	6205 (5%)	16516 (13%)	13794 (43%)	27 (8%)	69 (36%)	31
Disparity	994 (1%)	3664 (3%)	4548 (15%)	4 (2%)	85 (45%)	47
Spacesweep	5493 (5%)	10280 (8%)	6277 (20%)	50 (14%)	74 (39%)	51

Kernel	Tailored Tool Settings					
	LUT	DFF	CY	DSP	RAMB	MHz
Canny	1843 (2%)	2349 (2%)	1086 (4%)	4 (2%)	177 (93%)	38
Harris	6105 (5%)	15413 (13%)	7112 (23%)	81 (22%)	69 (36%)	40
Disparity	998 (1%)	3672 (3%)	4548 (15%)	4 (2%)	85 (45%)	50
Spacesweep	5458 (5%)	10276 (8%)	6277 (20%)	50 (14%)	79 (42%)	52

[*] %: utilization of NG-Large (137K LUTs, 129K DFFs, 32K CYs, 384 DSPs, 192 RAMBs).

wise tool exploration (see Figure 8.3). The upper/lower part of Table 8.4 presents the NG-Large utilization of place & route with the default/tailored NXmap settings. Firstly, we observe that the memory and arithmetic resources, i.e., RFs/RAMBs and CYs/DSPs, remain equal to those reported by synthesis. Moreover, the variations in resources for tuning the place & route settings (`DensityEffort`, `CongestionEffort`, `PolishingEffort`, `RoutingEffort`, `BypassingEffort`) are negligible for all the kernels, i.e., ±10 LUTs. In terms of performance, all the kernels achieve better clock frequency by modifying some of the default settings. We combine different options for the aforementioned tool settings and report those that provide the maximum frequency according to NXmap's STA. For Canny, the `PolishingEffort` setting is set to "high" rather than "medium", providing an increase of 2.7MHz. For Harris, the `PolishingEffort` setting is set to "low" rather than "medium", giving an increase of 9.5MHz. For Disparity, the `PolishingEffort` setting is set to "low" rather than "medium" and the `DensityEffort` to "medium" rather than "low", delivering an increase of 2.6MHz. For Spacesweep, the `CongestionEffort` setting is set to "medium" rather than "high", delivering an increase of 0.7MHz. These gains in the clock frequency may be considered limited, but as we show in the next section, they are still important towards improving the throughput of the CV kernels. Moreover, we note again that our tool exploration provided significant gains in the initial tool versions, i.e., up to 3× frequency improvement.

Overall, with respect to the reported resource utilization, we conclude that the NXmap tool maps the CV kernels as expected. Considering that these kernels im-

pose diverse memory requirements, the tool correctly employs several of the available RAMB configurations, e.g., 24K×2 and 12K×4. As a result, reasonable RAMB utilization is derived for 1024-pixel-wide images, ranging from 36% (Harris) to 93% (Canny). Canny utilizes almost all the on-chip RAM resources, as it receives the entire image and operates in burst mode, contrary to the other kernels that input image stripes. Regarding the arithmetic and logic components of the kernels, NXmap successfully recognizes, reports, and maps all the arithmetic operators, finite-state machines, and logic functions.

Evaluation of NG-Large's Performance, Power and Configuration

Table 8.5 reports the performance results for the kernels (clock frequency, latency for one input frame and throughput). The throughput metric excludes I/O and refers only to processing, i.e., the single execution of the kernel. For the feature detection kernels, we measure the Frames Per Second (FPS), while for the depth extraction kernels, we employ the MPixel Disparities per Second (MPDS), which is a metric combining both resolution and performance. The results show that NG-Large provides sufficient performance for 1-MPixel image, taking into account the requirements of the corresponding VBN space applications. The time required for a complete reconstruction using Disparity and Spacesweep could improve the conventional depth extraction of Mars rovers by 1 order of magnitude (in terms of resolution and speed). We note that in the tested configuration, Spacesweep examines 3× more depth levels than Disparity (300 versus 100), and thus, it provides much higher accuracy. Furthermore, given that most VBN pipelines require 1–10 FPS, we conclude that the throughput of Canny and Harris, which is 5.3 FPS and 10 FPS, respectively, leaves enough room for the complementary components of an algorithmic chain to finish on time. Regarding the power consumption of NG-Large, we report a detailed analysis in [319]. In brief, the static power (when the FPGA is not programmed) is 2W, while the dynamic power varies, as it depends on the resource utilization (e.g., it is ~200mW when utilizing 10K FEs or 200 DSPs, and ~40mW when utilizing 150 RAMBs). Based on our power measurements and analysis in [319], we report a power estimation in Table 8.5, which, however, regards only the resource utilization and does not take into account the I/Os, interconnections, and switching activity. When considering these parameters, the power consumption of NG-Large lies around the typical FPGA values for such high-performance DSP workloads.

In Table 8.6, we report the bitstream size of each kernel and the time required for NG-Large to be programmed via JTAG operating at 8MHz. According to the results, the configuration time of NG-Large is almost proportional to the bitstream size. In particular, the JTAG interface programs the chip with a rate of 452 KB/s for Canny, 389 KB/s for Harris, 381 KB/s for Disparity and 399 KB/s for Spacesweep. The

Table 8.5: Performance and power of computer vision kernels on NG-Large FPGA.

Kernel	Clock (MHz)	Latency (s/frame)	Throughput[1] (FPS) | (MPDS)	Power[2] (W)
Canny	38	0.10	10	2.3
Harris	40	0.19	5.3	2.5
Disparity	50	6.7	18	2.3
Spacesweep	52	10.8	29	2.4

[1] Excluding I/O: FPS for Harris and Canny, MPDS for Disparity and Spacesweep.
[2] Estimation based on the static FPGA power and the dynamic resource power (without taking into account the I/Os, interconnections and switching activity).

Table 8.6: Results from the configuration of the computer vision kernels on NG-Large FPGA.

Kernel	Bitstream Size (KB)	Configuration Time[1] (s)	Configuration Rate (KB/s)
Canny	2669	5.9	452
Harris	1751	4.5	389
Disparity	1563	4.1	381
Spacesweep	1719	4.3	399

[1] FPGA configured via JTAG using Intel Core i7-4500U@1.80GHz×4, 8GB RAM.

bitstream of Canny is around 1MB larger than that of the other kernels, which is due to its 93% RAMB utilization, and thus, it requires more time to be configured. The configuration times can be greatly improved by using the SpaceWire interface, which can operate up to 400 Mbps. Furthermore, we observe that the size of the NG-Large bitstream is not fixed, like in other 3rd-party FPGAs, where the bitstream size is pre-determined regardless of the design size and complexity. This is an advantage of NG-Large, considering that in space missions various bitstreams are stored on-board. These bitstreams implement either different kernels (in case of algorithmic pipelines) or multiple variants of the same kernel (e.g., performance-wise and accuracy-wise variants). In this context, our work in [325] discusses adaptive scenarios in space applications that require multiple bitstreams to be configured. The same work also evaluates the reconfiguration capabilities of NanoXplore's NG-Medium based on the expected reconfiguration rates of the space applications.

8.5.4. Comparative FPGA Evaluation

In this section, we compare the implementations of the CV algorithms on NG-Large with those on other FPGAs (Virtex-5QV, Cyclone III, RTG4, NG-Medium). Our goal

is to evaluate the BRAVE tools and devices by comparing them to well-established and more mature solutions in the space domain that exist in the market for much longer. Moreover, we compare NG-Large with NG-Medium in order to evaluate the evolution of the space-grade BRAVE FPGAs.

Comparison to 3rd-Party FPGAs

For Canny, NXmap provides LUT utilization that is comparable to that of the 3rd-party tools, i.e., LUTs are increased by only 6%. However, we note that if we also consider the route-thru LUTs that are consumed due to CY utilization, the LUTs are increased by 48%. The DFF resources are increased by almost 50% in NG-Large. This huge increment is due to not utilizing the internal registers of RAMBs (given that the RAMB utilization is 93% and our memories include registers). In terms of DSPs, NXmap utilizes the same resources with the 3rd-party tools, while it is more efficient in the utilization of the memory resources (considering the RAMB size difference between devices). Overall, we consider the Canny resources as reasonable, taking into account that it almost fills up all the RAMBs, and thus, the tool is stressed to provide efficient mapping for the rest resources. For Harris, NXmap provides a good LUT utilization, i.e., $3.2\times$ less LUT versus the average 3rd-party value. When considering the pass-thru LUTs for the CY utilization, the total number of LUTs increases, but it is still comparable with the 3rd-party value. Moreover, the LUT utilization should be examined along with the number of employed DSPs, for which NXmap utilizes $1.5\times$ less. Regarding the memory resources, NXmap delivers 56% less RAMBs, while the total RAMB Kbits consumed are less than the average 3rd-party value. Similar results are observed for the other two CV kernels. For Disparity, NXmap provides promising LUT utilization, as it employs a small number, even when considering the CY resources. Regarding RAM resources, NXmap is below the 3rd-party value in both blocks and Kbits. For Spacesweep, NXmap provides small LUT utilization, which is better by 52% compared to the other tools, and almost the same when considering the CYs. The DFF utilization is also better by 5%, while the DSP and RAMB utilization is excellent, as NXmap outperforms the average 3rd-party values by 20% and 30%, respectively.

Comparison to the NG-Medium FPGA

Next, we compare NG-Large against its predecessor, i.e., NG-Medium, to evaluate the progress of BRAVE devices and examine if NG-Large provides significant advantage due to being $4\times$ bigger in resources. We implement the same kernels on NG-Medium, initially configured as shown in Table 8.2, and we apply the final tailored tool settings that provided the best frequency on NG-Large. We note that we

do not change the algorithm of the kernels, namely, the same HDL sources are implemented in both devices. Only in case of resource over-utilization or unexpected tool issue in NG-Medium, we modify either the kernel parameters (e.g., use smaller input image) or the tool settings (e.g., apply different mapping). Table 8.7 reports the resource utilization in both BRAVE FPGAs. As expected, the memory resources of NG-Medium force us to modify the parameters in all kernels. More specifically, in Harris and Disparity, we decrease the height of the input image stripe by $4\times$ and $2\times$, respectively, while in Canny and Spacesweep, we decrease the size of the entire image by $4\times$ (from 1024×1024 to 512×512). The derived results show that, even though the designs of NG-Large regard inputs with larger size, its resource utilization percentage is significantly better, leaving room for implementing complementary HDL components, increasing the parallelization, or serving even bigger input images.

In Figure 8.6, we report the latency improvement in NG-Large compared to NG-Medium. Harris in NG-Large is better by $4.6\times$, as the clock frequency is increased by ~$4\times$, and NG-Medium has to process $4\times$ more (smaller though) image stripes. Canny achieves better clock frequency and execution time in NG-Medium, but for the 1/4 of NG-Large's input image. To serve an 1024×1024 image, it will have to process each 1/4 of the image, send the partitioned edge map back to the CPU, and at the end, bring all the partitioned edge maps back to the FPGA. These extra steps will add an overhead of ~60ms in the good scenario, i.e., when using the SpaceWire interface at 100 Mbps for data transmission. Thus, the total performance improvement in NG-Large is $1.4\times$. Regarding Disparity, both devices achieve almost the same clock frequency, i.e., around 50MHz, however, NG-Medium has to process $2\times$ more (smaller though) image stripes. This differentiation in the partition of the input image is translated to $1.2\times$ performance improvement in NG-Large. For Spacesweep, considering that NG-Medium has to process smaller input image to avoid resource over-utilization, and also with a clock frequency decreased by $1.7\times$, NG-Large delivers around $1.7\times$ better performance. In any case, we note again that we implement the same algorithms on both devices, and the performance in NG-Large can be greatly improved by exploiting its bigger capacity and increasing the parallelization, e.g., implement parallel arithmetic operators or process stripes in parallel.

Concluding, in comparison with the 3rd-party FPGAs, NG-Large exhibits comparable, or even better in several cases, resource utilization (depending on the kernel and the FPGA block). Considering that NXmap is a newly released tool and that we have seen continuous improvement in every new tool version, the results are expected to improve in the future. Compared to NG-Medium, NG-Large provides increased flexibility due to its bigger capacity, while the former is stressed (or is unable) to implement CV kernels for typical image sizes, e.g., 1 MPixel. NG-Large delivers bet-

Table 8.7: Comparison of NanoXplore's FPGAs for the implementation of computer vision kernels.

Kernel[1]	FPGA	NG-Med. vs. NG-Lar.[2]	LUT	DFF	CY	DSP	RAMB	MHz
Canny	NG-Large	4× smaller input image	2%	2%	4%	1%	93%	38
	NG-Medium	(0.25MP vs. 1MP)	7%	8%	13%	5%	90%	50
Harris	NG-Large	4× smaller img. partition	5%	13%	23%	22%	36%	40
	NG-Medium	(1K×8 vs. 1K×32)	19%	38%	73%	100%	63%	11
Disparity	NG-Large	2× smaller img. partition	1%	3%	15%	2%	45%	50
	NG-Medium	(1K×16 vs. 1K×32)	4%	10%	18%	41%	86%	52
Ssweep	NG-Large	4× smaller input image	5%	8%	20%	14%	42%	52
	NG-Medium	(0.25MP vs. 1MP)	22%	32%	78%	45%	95%	30

[1] The same algorithms are implemented in both FPGAs (no HDL modification).

[2] NG-Large is 4× bigger than NG-Medium in resources.
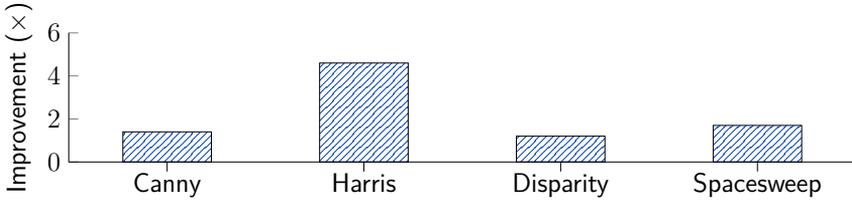
Figure 8.6: Latency improvement in computer vision kernels by NG-Large compared to NG-Medium for the same algorithm (no HDL modification, e.g., for increased parallelization).

ter performance that can be further improved by exploiting its resources to increase the parallelization. Furthermore, NG-Large leaves a significant amount of resources that can be used for implementing other complementary HDL components in the case of algorithmic pipelines, as shown in the system-level evaluation of our publication in [319].

8.6. Demonstration of the Computer Vision Kernels

To evaluate and demonstrate the actual hardware execution of the CV kernels on NG-Large, we develop a hardware/software architecture that is based on the serial UART communication. Our goal is to transmit and receive I/O data to/from NG-Large and validate the results, and not to provide the optimal I/O and system throughput (which can be achieved via the high-performance SpaceWire interface). For this purpose, we develop software in our host-PC for I/O data handling and demonstration, as well as hardware in NG-Large for I/O data handling and kernel control.
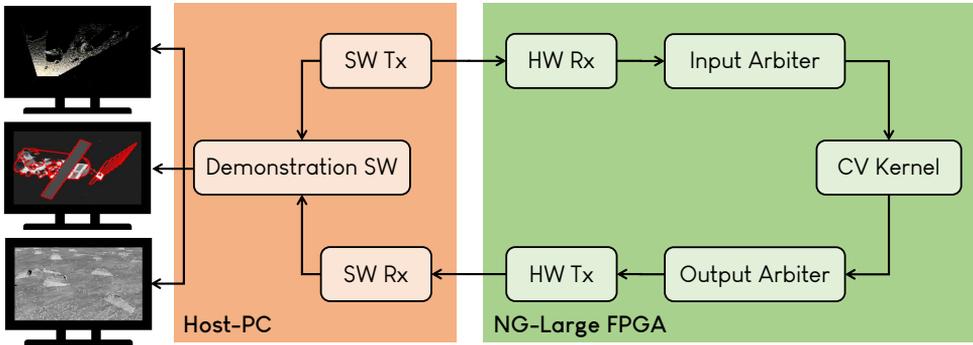
Figure 8.7: Hardware/Software UART-based architecture for the execution of the computer vision kernels on NG-Large (testing and demonstration).

8.6.1. Development of CPU–FPGA Communication

To establish a communication for data transfers between the host-PC and NG-Large, we adopt the infrastructure of Figure 8.7. In the host-PC side, we develop C functions using the GCC compiler, which are tailored to the data sizes and bit-widths of each kernel. In brief, the host-PC functions perform the following operations:

- open the USB port for the serial communication.

- specify the settings for the communication, e.g., baud rate, timeouts, blocking or non-blocking mode.

- read the input data (stored in the host-PC) and send them to NG-Large by applying the appropriate bit manipulations and writing the USB port.

- receive the output data from the NG-Large execution by reading the USB port.

- write the output data to files for comparison with the groundtruth data and demonstration.

In the FPGA side, we implement a hardware architecture for receiving/transmitting data from/to the host-PC, as well as for applying the required data encoding/decoding and controlling the kernel's I/Os. The development is performed in pure VHDL, and the architecture is parametric in terms of baud rate and global clock frequency. The UART modules are implemented based on the protocol specifications: the Rx/Tx component serially receives/transmits the data, while the tick generator synchronizes the UART logic with respect to the baud rate. The arbiters for controlling the kernel I/Os, i.e., the glue logic between UART and the kernel, are based on the kernel's architecture (e.g., processing in image stripes or in burst mode) and I/O signals.
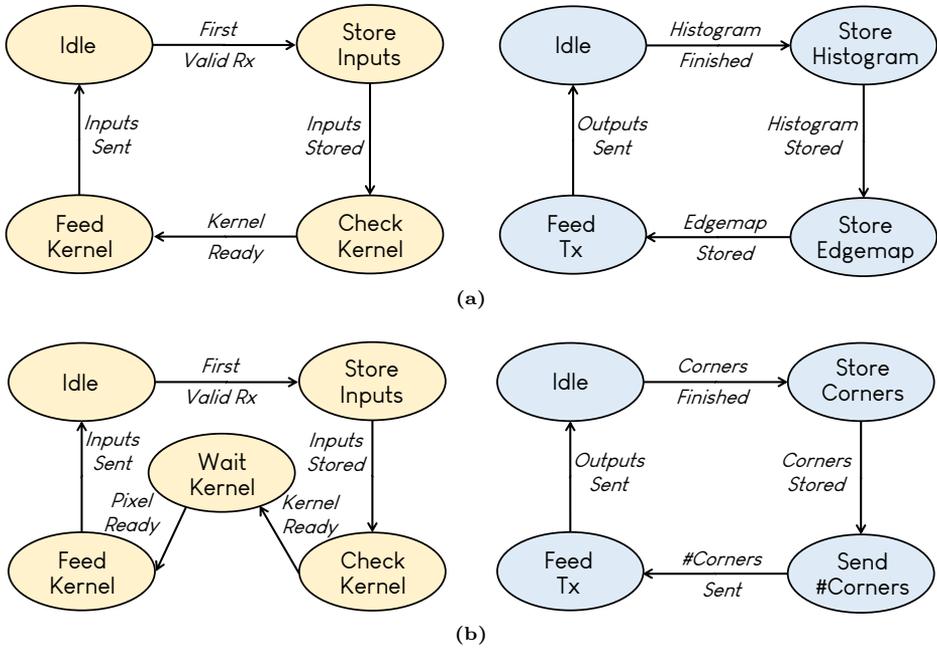
Figure 8.8: Input and output arbiter for handling the I/O data of **(a)** Canny and **(b)** Harris. The corresponding arbiters for Disparity and Spacesweep are similar to those of Harris.

In general, the input arbiter collects the data from the Rx component, packs them accordingly, and feeds the kernel when it is ready to start the processing. Correspondingly, the output arbiter retrieves the kernel's outputs, decouples them to packets, and forwards them to the Tx component when the latter is ready to transmit them to the host-PC.

Figure 8.8 illustrates the main finite-state machines of Canny's and Harris' arbiters. The arbiters of Disparity and Spacesweep are similar to those of Harris. At first, Canny receives the two thresholds required for the hysteresis thresholding task, and then it receives the entire image. The input arbiter remains in the "Idle" state until the flag signal for valid data reception is activated for the first time by the Rx component, which means that the first threshold is successfully received. This threshold is stored in a register and the arbiter moves to the state "Store Inputs". It remains there until the second threshold and all the image pixels are received and stored in register and memory bank, respectively. Next, in the state "Check Kernel", the arbiter examines if Canny is ready to receive the input data. When that happens, the controller moves to the "Feed Kernel" state, where it sends the thresholds and pixels to Canny and activates all the signals required for starting the processing of a new

image. Upon finishing this process, it returns to the "Idle" state, waiting for the next image. Correspondingly, the output arbiter remains in the "Idle" state until Canny activates its output signals indicating that both the histogram and the edge map have been calculated. When that happens, the arbiter starts to store the histogram and edgemap in memory banks, before sending them to the Tx component. This data storing is required, as the Tx component is not always ready to transmit the outputs to the host-PC. Therefore, the output memory banks are read when Tx is ready for transmission. When all the outputs are sent to Tx, the arbiter returns to its "Idle" state to wait for the next Canny outputs. In the same context, we design the arbiters of Harris, which, however, transfer and process the image in stripes. Moreover, before starting the processing of a new image, we feed Harris with a new threshold, which is updated in the software based on the previous frame's corners. Another difference is that Harris produces 32-bit output data, while the UART communication is designed for 8-bit data. Therefore, we transmit the outputs of Harris in bytes, and perform the required bit manipulations in the software receiving the data from the FPGA.

8.6.2. Real-Time Processing and Visualization

We assume the real-world scenario of space missions, where NG-Large is equipped on-board. The camera captures random images during the traversal of the rover, and NG-Large detects features on them using Canny and Harris, i.e., edges and corners, respectively. These features are useful in VBN pipelines for pose tracking [326], rover localization [327], and other space applications. On the other hand, Disparity and Spacesweep are used to extract the depth of the scene when the camera captures stereo images (e.g., every 20cm) [328, 329].

For demonstration purposes, we visualize the output results of the CV kernels. For the feature detection kernels (Canny and Harris), we use both real-time camera input and stored images, and we plot the edges and corners on them. For the depth extraction kernels (Disparity and Spacesweep), we use 20 synthetic stereo images depicting the Martian terrain and visualize the data in 2D and 3D. Figure 8.9 illustrates output images from the execution of the kernels on NG-Large. Our demonstration video is also available online[3]. As explained in the previous section, we aim only to demonstrate the correct hardware execution and not evaluate the system throughput. The serial UART communication does not provide increased I/O rate, and thus, the total system throughput is low. Considering the processing throughput of the CV kernels (see Table 8.5), the use of the high-performance SpaceWire interface (e.g., at 100 Mbps) would provide sufficient system throughput.

---

[3]Demo available on YouTube: **https://youtu.be/q8NKV4rpcY4**.

(a)

(b)
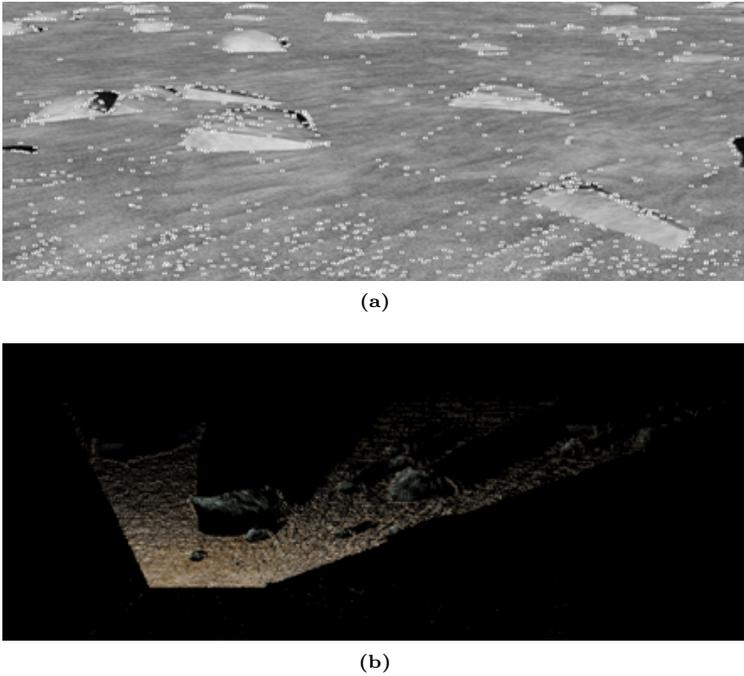
Figure 8.9: Visualization of results from the execution of computer vision kernels on NG-Large: **(a)** Harris Corner Detector [327] and **(b)** GAD-Disparity [328].

8.7. Conclusion

In this chapter, we examined the acceleration capabilities of the new European space-grade BRAVE FPGAs. In particular, we employed high-performance CV kernels that were developed in other FPGA devices/tools, and we systematically ported them on NanoXplore's NG-Large FPGA. Towards exploiting all the available software tools settings, as well as for surpassing the issues and inefficient results of such new devices/tools, we proposed a three-stage development & assessment methodology. The methodology regards the typical stages of the FPGA design flow, i.e., synthesis, place & route (implementation), and bitstream generation & hardware execution, and it can be used for testing and development on new FPGAs. In our work, we used it to efficiently port the high-performance CV kernels on NG-Large and evaluate this new space-grade chip as on-board processor for space missions. Our experimental evaluation examined the resource utilization, performance, power consumption, bitstream size, and configuration time, which are all taken into account in space mission scenarios. It also included comparisons with well-established FPGA devices/tools. Besides measuring the flexibility and efficiency of the new NXmap tool, which is continuously

evolving and improving, we showed that NG-Large could achieve feature extraction with a throughput of up to 10 FPS and depth extraction with a latency of around 10s. Overall, NG-Large could implement high-performance and complicated algorithms with sufficient hardware metrics, i.e., resource utilization, throughput, and power consumption, which were all shown to be competitive/comparable with the 3rd-party FPGA designs.

Chapter 9

DSP & AI Acceleration on Heterogeneous Multi-Core SoCs

The increased computational demands of modern applications from domains such as Artificial Intelligence (AI) and Digital Signal Processing (DSP) challenges their deployment at the edge. At the same time, embedded systems are presented with tight energy constraints, which limit their processing capabilities compared to the high-performance computers of the cloud. Heterogeneous System-on-Chip (SoC) processors emerge as an attractive hardware solution, however, they still require sophisticated development to provide efficient implementations. In this chapter, we exploit the heterogeneity of the multi-core Vision Processing Units (VPUs), which are low-power SoCs with increased diversity in processors and memories, to accelerate demanding DSP and AI algorithms. Towards the efficient utilization of such heterogeneous and complex SoCs, we employ a design methodology and various high- and low-level techniques. Our implementations include custom DSP kernels, as well as a Computer Vision (CV) pipeline and a demanding Deep Neural Network (DNN), which both perform pose estimation in space. The individual kernels are accelerated by $10\times$–$20\times$ on the Myriad 2 VPU, while for the CV pipeline, we provide a speedup of $8.5\times$–$12\times$. The throughput of this pipeline for MPixel frames is up to 5 FPS, while the power consumption lies between 0.8 W and 1.1 W. The deployment of the DNN on Myriad X for resampled MPixel frames provides a throughput of 2.7 FPS and a maximum power consumption of 2 W. Moreover, we directly compare the VPUs to other embedded devices. According to our analysis, the Myriad VPUs exhibit significantly better power efficiency, i.e., $5\times$ versus the Jetson Nano GPU and $4\times$ versus the Zynq FPGA. When examining the performance-per-Watt ratio of the devices, the VPUs provide comparable results, and in some cases, even better: e.g., for the CV pipeline, Myriad 2 trades a $3\times$ loss in speed for a $4\times$ gain in mean power consumption.
*This chapter is based on our **publications** in [359–362].*

9.1. Introduction

The last decade is characterized by a rapid growth of powerful Artificial Intelligence (AI) and complex Digital Signal Processing (DSP) algorithms. This algorithmic evolution along with the large increment in sensor data have significantly affected computing systems at the edge. The worldwide demand for speed challenges the integration of AI/DSP functionalities in novel applications, especially at the power-constrained embedded systems. Heterogeneous System-on-Chip (SoC) processors emerge as a promising solution [25], offering increased programming flexibility and diversity in terms of processors and storage. Besides the well-established SoCs integrating Central Processing Units (CPUs) and Graphics Processing Units (GPUs), a novel class of heterogeneous SoCs has recently appeared, namely the Vision Processing Unit (VPU) [36, 363]. Compared to the CPU–GPU SoCs, the VPUs provide better power efficiency. For example, Nvidia's Jetson Nano GPU consumes 10W/5W, while Intel's Myriad VPUs consume 1W–2W. Compared to the Field-Programmable Gate Arrays (FPGAs), the VPUs offer improved programmability, smaller development time/effort, and lower power consumption. Moreover, the VPUs can handle classic DSP workloads, contrary to AI-specific processors, such as Google's Tensor Processing Units (TPUs). In general, the VPUs integrate a variety of processors, e.g., general-purpose cores, hardware filters, vector cores, and neural network accelerators. The VPU SoCs excel in low-power imaging applications, covering domains such as robotics, automotive, and space.

Intel manufactures the Myriad VPUs [364], i.e., Myriad 2 (28nm) and Myriad X (16nm), which are prominent and well-established processors for embedded DSP & AI applications. By offering numerous heterogeneous processing cores, these VPUs enable the efficient parallelization of demanding algorithms. A single VPU chip integrates multiple hardware I/O peripherals, a variety of hardcoded low-power imaging filters, general-purpose processors, acceleration vector cores, and AI acceleration engine(s). A similar heterogeneity is provided in storage, for which the VPU SoCs offer a memory hierarchy consisting of DRAM, scratchpad, and cache memories. It is evident that the full utilization and exploitation of such complex heterogeneous SoCs requires a meticulous and systematic development approach. Furthermore, considering that these SoCs are build towards more "low-power" than "high-performance", the need for efficient mapping and deployment becomes even more crucial.

As explained in Chapter 8, space is one of the communities searching for alternative processing platforms to comply with the tight constraints of on-board processing. Besides the FPGAs, the enhanced performance of modern low-power edge devices can efficiently serve tasks of Earth Observation (EO) and Vision-Based Navigation

(VBN). The heterogeneity of platforms such as the VPUs, allows for improved adaptability to various mission scenarios and seamless in-flight re-programmability. To further improve the performance and Size, Weight and Power (SWaP), as well as additional costs (e.g., development effort), the space industry is studying mixed-criticality architectures [365–368], i.e., the integration of both space-grade and Commercial-Off-The-Shelf (COTS) components. The use of COTS components in Low Earth Orbit (LEO) missions and CubeSats relies on the partial shielding provided by Earth's magnetosphere and/or the short mission lifetime, which limit the damage or unavailability of electronics due to radiation. In this context, FPGAs [322–324, 326], GPUs [330, 331, 369–371] and VPUs [372–376] are evaluated as accelerators, while they are also subjected to radiation tests, such as the Myriad 2 VPU [377]. A second challenge for the space industry is the wider adoption of AI, which is currently limited to offline/ground data processing and not on-board processing, mostly due to insufficient computational power and increased memory footprint, as well as qualification issues when deployed in orbit [377].

The promising features of *Myriad VPUs* has attracted the interest of the European Space Agency (ESA), which is thoroughly involved in the safari of COTS embedded devices. Towards the use of VPUs in space, ESA is supporting research activities[1] that evaluate the overall performance of the Myriad SoCs, including their integration in high-performance compute boards and mixed-criticality architectures for space avionics. These activities aim to assess the suitability of the VPUs as COTS parts of the on-board computer, mainly for low-power DSP/AI acceleration. In the context of these activities, we propose a methodology to support the development on the Myriad VPUs and accelerate demanding DSP/AI workloads. Our goal is to surpass the bottlenecks of the resource-constrained embedded computing, unlock the full potential of VPU's heterogeneity, and thus, efficiently deploy complex algorithms. Our application domain is space, however, both our methodology and development techniques are generic, i.e., they can be used as design paradigm to provide DSP/AI acceleration on the heterogeneous multi-core VPUs. Regarding development, we apply various high-level parallelization and partitioning techniques, while at lower level, we optimize the memories and the acceleration cores. In terms of algorithms, at first, we implement custom DSP and AI kernels on Myriad 2, targeting to demonstrate the VPU's capabilities and evaluate embedded design techniques for parallelization and optimization. Afterwards, we accelerate a sophisticated 5-stage Computer Vision (CV) pipeline (developed by Lourakis and Zabulis [378][2]) on Myriad 2. Finally, we accelerate a compute-intensive Deep Neural Network (DNN) [379] (that was not developed for embedded systems) on the AI acceleration engine of Myriad

---

[1]**ESA LEOTOME:** 4000126083/18/NL/FE
 ESA HPCB: 4000126129/18/NL/AF
[2]Special thanks to Dr. M. Lourakis & Dr. X. Zabulis from FORTH for providing the initial CV SW.

X.

The **contribution** of this chapter is summarized as follows:

(i) We highlight the great diversity of heterogeneous SoCs in terms of processors and storage, as well as the significance of the meticulous exploitation of the heterogeneity towards improved performance.

(ii) We propose a methodology for the optimized mapping and acceleration of demanding DSP and AI algorithms on heterogeneous multi-core VPUs, while we demonstrate several high- and low-level implementation techniques for embedded systems.

(iii) We report comparative experimental results for embedded CPUs, VPUs, GPUs and FPGAs, and we discuss the trade-offs of each processor.

(iv) We test and evaluate Intel's Myriad VPUs as candidate on-board COTS accelerators for future space missions.

The remainder of this chapter is organized as follows. Section 9.2 overviews the market's embedded platforms and presents the Myriad VPUs. Section 9.3 introduces our design methodology. Sections 9.4–9.6 discuss the implementation details of the DSP and AI algorithms. Section 9.7 reports various experimental results. Finally, Section 9.8 draws the conclusions.

9.2. Background

9.2.1. The Landscape of Embedded Devices

Table 9.1 summarizes well-established embedded devices of the market, including the Myriad VPUs. We report platforms with power consumption lying within the same order of magnitude (e.g., there are other Nvidia Jetson devices consuming more power). As shown, Intel's VPUs integrate LEON general-purpose processors along with the RTEMS real-time operating system. On the other hand, Nvidia's GPUs and Google's TPUs feature ARM processors and are equipped with Linux-based operating systems. In terms of accelerators, the VPUs have more heterogeneity than the other devices, offering multiple Streaming Hybrid Architecture Vector Engine (SHAVE) cores and hardware filters, while Myriad X also integrates a neural engine. The GPUs are based on the Compute Unified Device Architecture (CUDA) cores for acceleration, whereas the key processing unit of TPUs is a systolic array with multipliers and accumulators. The AI performance of these devices varies and shows that the TPUs are the winners, however, these metrics are theoretical and should also be examined along the

Table 9.1: Overview of market's embedded devices.

Vendor	Device	CPU	Accelerators
Intel	Myriad 2	LEON4 ($\times$2)	12-Core SHAVE, HW Filters
	Myriad X	LEON4 ($\times$2)	16-Core SHAVE, AI Engine, HW Filters
Nvidia	Jetson Nano	Cortex-A57	128-Core Maxwell GPU
	Jetson TX2	Cortex-A57, Denver 2	256-Core Pascal GPU
Google	Coral Mini	Cortex-A35	64$\times$64-Array[1] Edge TPU
	Coral	Cortex-A53, Cortex-M4F	64$\times$64-Array[1] Edge TPU

Vendor	Device	OS	Max Clock	AI Performance[2]	Power[3]
Intel	Myriad 2	RTEMS	600MHz	100 GFLOPS (fp16)	1W
	Myriad X	RTEMS	700MHz	1 TFLOPS (fp16)	2W
Nvidia	Jetson Nano	Linux4Tegra	922MHz	472 GFLOPS (fp16)	10W / 5W
	Jetson TX2	Linux4Tegra	1122MHz	1.3 TFLOPS (fp16)	15W / 7.5W
Google	Coral Mini	Mendel Linux	500MHz	4 TOPS (int8)	5W
	Coral	Mendel Linux	500MHz	4 TOPS (int8)	5W

[1] Estimation based on AI performance and testing (the array dimensions have not been revealed).
[2] Reported officially by the vendors.
[3] According to our in-house measurements.

AI accuracy and the actual power consumption. Moreover, a disadvantage of TPUs is that they are AI-specific accelerators, and thus, they do not provide acceleration cores for classic DSP and CV workloads. Regarding power, the VPUs are the most efficient solution, as they consume up to 2W. The TPU chip consumes 0.5W per Tera Operation Per Second (TOPS), however, the power of the entire system is ~5W. Especially for AI, the VPUs and the GPUs support various development frameworks, e.g., TensorFlow and PyTorch, while the TPUs rely only on TensorFlow Lite. Finally, both the VPUs and TPUs are produced as USB accelerators that can be hosted on PC or single-board computer (e.g., Raspberry Pi).

These devices are used for accelerating compute-intensive DSP and AI workloads at the edge. Because of their increased programming complexity and performance issues compared to their desktop and data center counterparts, they have received significant research attention. More specifically, the literature includes several methodologies and techniques for the development on these devices and provides a plethora of benchmarking/evaluation results. There are also relevant works that propose new frameworks/libraries and co-processing embedded architectures. As our application domain is space, we report related work on the aforementioned devices.

The Myriad VPUs are systematically evaluated by the space community. Furano *et al.* [377] report results from the radiation tests on Myriad 2, involving various functional

tests and characterization of all SoC's memories. Their results show that Myriad 2 remains fully functional after being exposed to a total ionizing dose of 49 krad(Si). Agarwal *et al.* [374] use Myriad 2 to implement a star identification neural network. Their experimental evaluation shows that Myriad 2 provides sufficient performance, while it consumes around 1W and retains 99% accuracy. Myriad 2 is also integrated in custom boards and co-processing architectures. This VPU is equipped on-board in the Φ-Sat-1 CubeSat mission of ESA as DNN demonstrator for EO [372]. Furthermore, it is integrated in Ubotica's CogniSat platform [375], which is an AI inference and CV engine that exposes Myriad 2 to the payload developer. In the same context, Myriad 2 is the main accelerator of the HPCB platform [373], which is a payload data processor board provided by Gobham Gaisler. Interestingly, the HPCB includes three Myriad 2 SoCs to provide fault tolerance and increased performance. For the successor of Myriad 2, i.e., Myriad X, the authors of [376] report results from the deployment of neural networks classifiers. The networks are trained on Mars imagery from the Reconnaissance Orbiter and Curiosity rover, and the average inference time is 16-20ms, while the power consumption lies around 1.9W.

Regarding the use of embedded GPUs in the space domain, Kosmidis *et al.* [369] examine their applicability from both the software and hardware perspectives. In particular, they analyze the algorithms and workloads of space applications to identify their suitability for GPUs, and they also perform benchmarking on GPUs. In [370], the authors evaluate two graphics-based computing methodologies (OpenGL 2.0 and Brook Auto) for safety-critical systems. Their main benchmark is an application modeling a VBN scenario where the aircraft performs rendez-vouz with an object. Moreover, the literature includes various FPGA–GPU co-processing architectures. The hybrid FPGA-GPU architecture of [330] employs Nvidia's TX2/TX2i GPU as main accelerator. The heterogeneous architecture of [331] integrates the AMD SoC (CPU–GPU) for acceleration, and optionally, a VPU for AI deployment. The authors of [371] use Nvidia's Jetson Nano GPU to accelerate neural networks for object detection along with image compression techniques. Finally, the edge TPUs are used in several terrestrial applications [380, 381], however, they are still not adopted in the space domain (ESA is working towards this direction [382][3]). Very recently, a CubeSat-sized co-processor with three TPUs was introduced [383], supporting various operation modes (high-performance, fault-tolerant, low-power).

9.2.2. The Intel VPUs and Tools

The Myriad family of VPUs offers heterogeneous multi-core SoCs for mobile/embedded applications. Besides the space domain, the Myriad VPUs are used for implement-

---

[3]**ESA CAIRS21:** 4000135491/21/NL/GLC/ov

ing DNNs [384–387], machine learning algorithms (e.g., SVM classifiers [388]), and CV functions (e.g., stereo vision [389]). The fabric architectures of Myriad 2 and Myriad X are illustrated in Figure 9.1 and Figure 9.2, respectively. Next, we discuss the SoC details and present the associated software tools.

The Myriad SoCs integrate multiple I/O peripherals and different types of processors. All these components are connected to a multi-ported high-bandwidth shared memory hierarchy. Regarding general-purpose CPUs, the SoCs include two LEON4 processors that implement the 32-bit RISC SPARCv8 architecture. The first processor is called LEON OS (LOS) and primarily handles the external communication, i.e., it controls peripherals such as UART, SPI, ETH, and USB3. Moreover, LOS runs the RTEMS real-time operating system. The second core is called LEON RT (LRT) and manages the media devices such as camera sensors and HDMI, while it also controls all the imaging interfaces (MIPI, LCD, CIF). The key processing units are the SHAVE cores, which are controlled at high level by the LEON processors. These cores are 128-bit Very-Long-Instruction-Word (VLIW) processors that are suitable for executing the bulk of compute-intensive tasks, offering not only core parallelization, but also room for low-level custom optimization. Each SHAVE supports Single-Instruction-Multiple-Data (SIMD) instructions on various data types, i.e., 16/32-bit floating-point and 8/16/32-bit integer. The Myriad VPUs are also equipped with hardware imaging accelerators, which are called Streaming Image Processing Pipeline (SIPP) filters. These specialized accelerators are configurable up to a certain degree and provide low-power implementation for numerous image processing kernels.

Regarding memory hierarchy, the VPUs provide on-chip DDR DRAM (global memory), which is accessed by the processors through a single DDR controller. Furthermore, the SoCs have a small SRAM memory, called Connection Matrix (CMX), which is primarily used by the acceleration cores as scratchpad memory. Each SHAVE core has preferential ports into a 128KB slice of the CMX memory, whereas the remaining storage can be exploited for other purposes. For the communication between DDR and CMX, the hardcoded DMA engine is used, providing high-bandwidth data transfers in either direction. The processors come along with their cache memories. More specifically, each LEON processor has both L2 and L1 caches, while the SHAVEs share a common L2 and have a dedicated L1.

The fabrication process of Myriad 2 is 28nm HPC+/HPC/HPM. This VPU offers 12 SHAVEs and its clock frequency can be configured up to 600MHz. In terms of memories, it has on-chip a 512MB DDR and a 2MB CMX. LOS has 64KB L1 and 256KB L2 caches, whereas LRT has 8KB L1 and 32KB L2 caches. SHAVEs share a common 256KB L2 cache, in addition to a 3K L1 per core (1KB for data and 2KB for instructions). The fabrication process of Myriad X is 16nm FFC. This next-generation
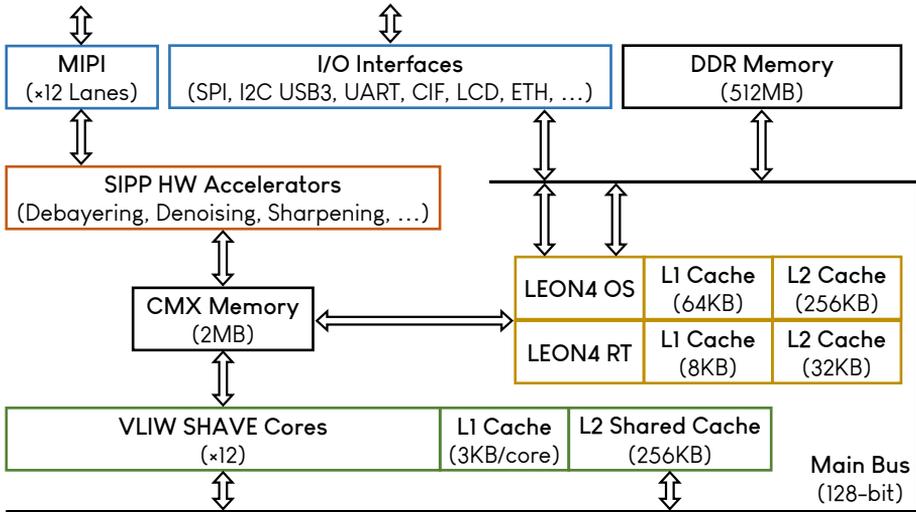
Figure 9.1: The fabric architecture of Intel's Myriad 2 VPU [364].
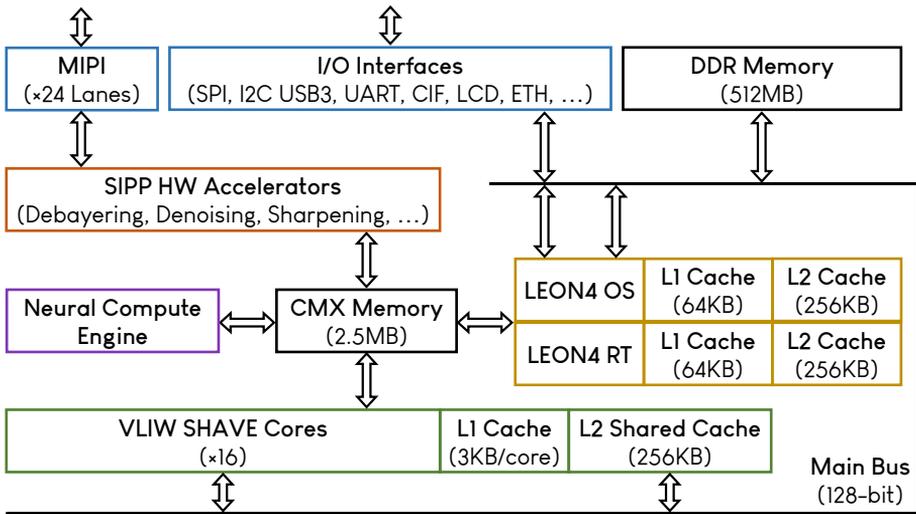
Figure 9.2: The fabric architecture of Intel's Myriad X VPU [364].

VPU features a dedicated on-chip accelerator, called Neural Computer Engine (NCE), for inferencing DNNs. Besides the AI accelerator, the main differences compared to Myriad 2 are the addition of 4 more SHAVE cores, the increment of the CMX capacity by 0.5MB, and the increment of the clock frequency to 700MHz. There are also more MIPI lanes and slightly bigger caches for LEONs.

To implement custom CV and DNN applications on the Myriad VPUs, the developer

uses the Myriad Development Kit (MDK). This programming suite integrates GCC toolchain for the LEON processors and a C/C++ compiler with extensive intrinsic support for SHAVEs, while offering numerous C/C++ libraries (e.g., for DMA transactions, SHAVE initialization, I/O data handling, power measurement). Moreover, MDK provides debugger, simulator, trace profiler, and libraries with CV kernels. In essence, the Myriad VPUs are programmed via an LLVM-based vectorizing C/C++ compiler, allowing the generation of assembly code, which is on a par with hand-optimized assembly. Towards improved assembly code, the developer can also write Myriad-friendly C/C++ code, e.g., explicitly apply the vectorization to leverage the automatic SIMD calculations.

To deploy DNN models of well-known frameworks (e.g., TensorFlow) on Myriad X, the developer uses Intel's OpenVINO toolkit [390]. Figure 9.3 shows the deployment procedure via OpenVINO with respect to the targeted platform: (i) the USB accelerator integrating Myriad X, which is called Neural Computer Stick 2 (NCS2) and (ii) the Myriad X SoC. In the first case (Figure 9.3a), OpenVINO inputs the frozen graph of the network and generates its intermediate representation with the model optimizer. The intermediate representation consists of an XML file for the network topology and a binary file for the weights and biases. These files are deployed on NCS2 using the OpenVINO C/C++ or Python API. In the second case (Figure 9.3b), OpenVINO converts the intermediate representation to a binary programming file, which is loaded on NCE via the mvNCI API. The same API is also used to feed NCE with input data and receive its outputs in the form of tensors.

9.3. Design Methodology

The efficient utilization of very heterogeneous SoCs, such as Myriad 2 and Myriad X, requires a methodical design approach. To exploit the full potential of heterogeneity, the diverse algorithmic functions of the application should be mapped to the most suitable processing units, while the development should be customized to the underlying hardware. Towards this direction, we propose a design methodology for efficiently partitioning, scheduling and mapping any DSP/AI application that consists of multiple algorithmic functions.

We divide our methodology in two branches. The first branch regards the implementation of an algorithm from the DSP domain, e.g., the CV pipeline of [378]. The second branch regards the deployment of an AI network, e.g., the DNN of [379]. The methodology for the AI application refers only to the Myriad X SoC and the NCS2 USB, which include NCE, i.e., the hardcoded AI accelerator. In case the targeted platform is Myriad 2 or more custom implementations are desired, the developer can adopt the methodology for the DSP application.
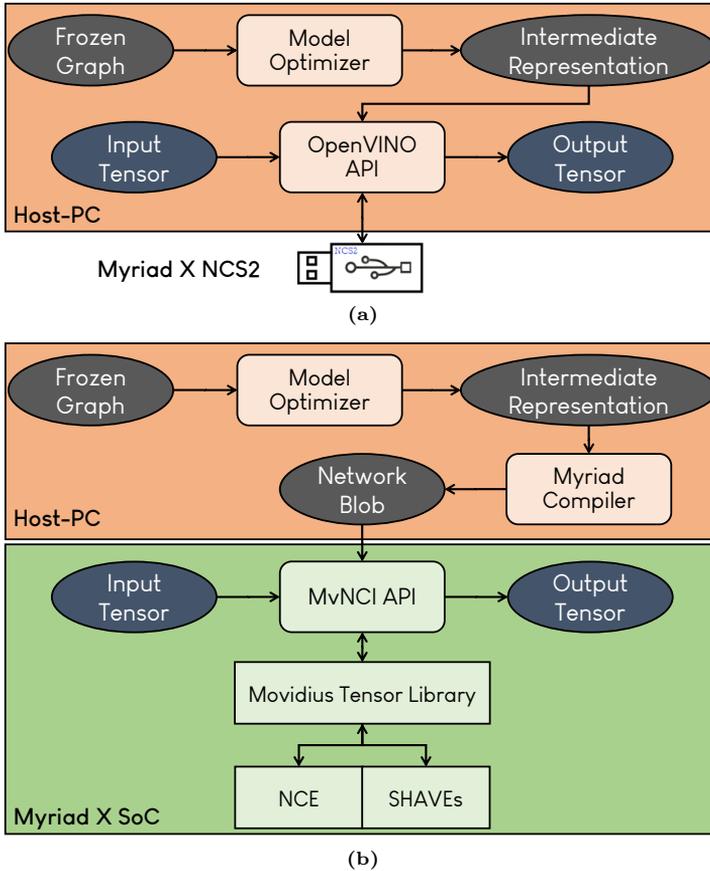
Figure 9.3: Deployment of DNNs on Myriad X via OpenVINO on **(a)** Neural Computer Stick 2 and **(b)** Myriad X SoC.

Regarding the DSP application, we begin by developing all the algorithmic functions or porting their pure C/C++ code (developed in other platforms) on the general-purpose LEON CPU. Upon all the functions are successfully compiled and executed, we profile and analyze them to derive the execution time, the requirements in memory, I/O and arithmetic, and even the programming complexity. Based on this analysis, and according to our design goal, e.g., more "low-power" or more "high-performance", we partition the entire algorithm on the SoC, i.e., we determine the mapping target for each algorithmic function. Moreover, based on our analysis as well as the SoC's micro-architecture and the libraries/frameworks provided by the vendor, we develop the utility software, e.g., for handling the I/Os, managing the memory allocation, and parallelizing the tasks of each function. Afterwards, we begin the implementation of each function at both high and low level. At high level, we apply the most efficient

parallelization and mapping scheme to the SoC cores and organize our software in terms of time and memory allocation. At low level, we accelerate each function by rearranging its operations to facilitate pipelining and maximize the memory reuse, we apply word-length optimization to further minimize buffering, and we perform parallelization via vectorization and/or data decomposition techniques. The next step is to integrate all the functions to the system and explore all coding parameters to fine-tune the implementation for the given problem/dataset. In case the design constraints are not satisfied, we use our feedback loop to return either to the initial partitioning and scheduling of the entire algorithm or the high- and low-level implementation. All these methodology steps are summarized as follows.

Methodology for DSP Application:

1. Development/Porting of the application/algorithm on the LEON core.
2. Profiling and analysis of each algorithmic function with realistic dataset.
3. Partitioning and scheduling of the entire algorithm on the Myriad SoC.
4. Development of utility software for I/O handling, memory management, and low-level optimizations.
5. High-level parallelization of each algorithmic function to the SHAVE cores.
6. Low-level implementation and optimization in the SHAVE cores.
7. System integration of each algorithmic function.
8. Testing and tuning with application-specific datasets.
9. Return to step 3 or 5 until the constraints are met and the system is optimized.

Regarding the AI application, we follow steps similar to those proposed by Open-VINO [390]. The training and optimization of the network with the AI framework (e.g., TensorFlow, PyTorch) are out of the Dissertation's context. OpenVINO supports tuning and optimization on the network frozen graph via the model optimizer. One of the provided methods is the fusing of the linear operations, e.g., the multiplications and additions can be merged into a single multiply-add instance or fused to convolutional and fully-connected layers. The tool also offers other methods, such as grouped convolution fusing and network pruning. Moreover, if required, the developer can create custom operations in OpenCL. All these methodology steps for the AI deployment on Myriad X are summarized as follows.

Methodology for AI Application:

1. Generation of the DNN model in the AI development framework.
2. Optimization and development of custom network operations in OpenVINO.
3. Creation of the network programming file in OpenVINO.
4. Inference run on Myriad X and analysis of the results.

5. Return to step 1 or 2 until the constraints are met and the inference is optimized.

9.4. Implementation of DSP & AI Applications on the Myriad 2 VPU

In this section, we present the implementation of three custom kernels on Myriad 2. In particular, we develop in C/C++ an image binning kernel, floating-point convolutions, and a Convolutional Neural Network (CNN)[4] for detecting ships on satellite images [313]. This section acts as introductory in the VPU development and aims to introduce the Myriad computing paradigm and the SoC's functionalities, as well as demonstrate our programming approach.

9.4.1. Development of Custom DSP and CNN Kernels

In image processing, binning is the procedure of extracting a single pixel from an image region (cluster of pixels). This task, although results in loss of information, reduces the amount of data to be transferred and processed (e.g., a 4-MPixel sensor image is transformed to 1-MPixel). For our kernel, we adopt 2×2 Averaging Binning, namely, we assume 2×2 regions with stride 2. The new pixels are calculated as the mean values of the regions' pixels. For our convolution kernels, we employ single-precision floating-point masks of different sizes (3×3, 7×7, 13×13), targeting to stress the VPU with this fundamental DSP operation.

In the implementation of Averaging Binning and Floating-Point Convolution, the LEON processor initializes the SHAVEs cores and performs all the necessary high-level tasks for the core parallelization. Our design choices are the same for both kernels. The input and output images are stored in DDR (global memory), while we perform DMA transactions to transfer image slices in the CMX (scratchpad/working memory), which is accessed by the SHAVE acceleration cores. The high-level parallelization procedure is illustrated in Figure 9.4. Regarding general implementation details in SHAVEs, we do not use additional working buffers, i.e., the processing is performed in-place (in the input buffer), and we enable the caches. For Averaging Binning, we employ 2048×2048 8-bit input images, which are transformed to 1024×1024 8-bit output images. We divide the input image into 36 stripes, i.e., 35 stripes of size 2048×58 and 1 smaller stripe of size 2038×18, and we assign 3 stripes to each SHAVE to process them successively. For the Floating-Point Convolution, we consider 1024×1024 8-bit input images and zero padding. Our design is parametric in

---

[4]Special thanks to E. Petrongonas for coding on the CNN and Myriad 2.
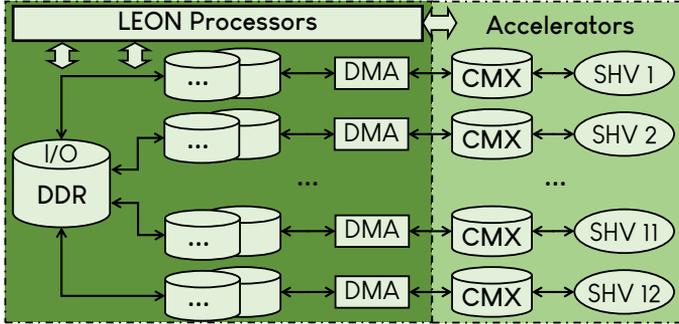
Figure 9.4: High-level parallelization of image processing workload in the Myriad 2 VPU.

terms of the mask size, allowing to change masks at compile time. The masks are stored in CMX to be accessed directly by SHAVEs. Regarding parallelization, we divide the image into 24 stripes, i.e., 22 of size 1024×43 and 2 of size 1024×39, and we assign 2 stripes to each SHAVE. The SHAVE processing uses vectorized operations as indicated by the generated assembly code.

Our CNN model is trained in TensorFlow with 128×128×3 images and 32-bit floating-point weights and biases. The accuracy is 96.8% for binary classification (ship detected or not). The network consists of four convolutional layers and two fully-connected layers, while the total number of weights is 131K. For our custom implementation in Myriad 2, the 32-bit floating-point weights and biases are converted to 16-bit floating-point using the corresponding routine of the MDK suite. The implementation is based on a custom parametric inference engine for $128 \times 128 \times 3$ input tensors, which is mapped onto SHAVEs and supports all the layers of our Ship Detection CNN. In our case, we develop the CNN accelerator for 1024×1024×3 16-bit input images. Considering that the inference engine is built for smaller tensors, we employ a function running on LEON that divides the input image into 64 $128 \times 128 \times 3$ patches. Subsequently, it successively stores them in the engine's input buffer and orders the SHAVEs to start the patch processing. In terms of memory utilization, we store the images, weights, and working buffers in DDR.

9.5. Porting of Computer Vision Pipeline on the Myriad 2 VPU

Based on our design methodology, we accelerate on Myriad 2 a vision-based pose tracking algorithm [378], which is representative of spacecraft proximity operations. In particular, it inputs a sequence of high-definition images and continuously performs

rendering, feature detection, feature matching, and data fitting via robust regression. The output is the 6D pose of the recorded satellite (Envisat) during a hypothetical maneuver in an Active Debris Removal mission [326]. This sophisticated 5-stage CV pipeline exhibits increased diversity in terms of computations and memory, thus, its porting on a low-power SoC is challenging. Next, we present the functions of the CV algorithm and discuss details about their implementation.

9.5.1. The CV Algorithm for Satellite Pose Tracking

The pose tracking algorithm of [378] assumes small motion between successive frames and uses a model-based approach to estimate the object's pose relative to the camera. Frame after frame, it evolves an initial pose by continuously rendering a depth map from the object's mesh model, detecting edges on it, and then matching them to the edges detected on the input image. These matches are used to refine the current pose i.e., to perform data fitting via least median of squares regression followed by iteratively reweighted least squares.

The key functions and dataflow of the algorithm are presented in Figure 9.5. The algorithmic functions are summarized as follows: (i) edge detection on the input/intensity image, (ii) depth map rendering, (iii) edge detection on the rendered/depth image, (iv) perpendicular edge matching, and (v) pose refinement. Regarding I/O, the algorithm inputs 1024×1024 8-bit grayscale images and outputs a 6×1 floating-point vector corresponding to the pose $\langle x, y, z, pitch, roll, yaw \rangle$. Moreover, it employs the object's mesh model (the Envisat satellite in our case), which consists of 20K vertices and 35K triangles.

For Edge Detection, the algorithm uses the well-known method of Canny [391] for both the intensity and depth images. This detector performs the following tasks: (i) Sobel convolution to compute image gradients (magnitude and direction), (ii) non-maximum suppression to remove spurious edges, (iii) hysteresis thresholding on the magnitudes to identify strong edges and suppress the weak ones. The hysteresis thresholding task initially retains all the mid-strength edges, but then it traces recursively their spatial connections to keep only those related to strong edges. This procedure is based on two thresholds, which are calculated based on the median in the histogram of gradients.

For Depth Rendering, the algorithm uses a triangle 3D mesh model and the current 6D pose to generate an image, whose pixels encode the distance between the camera and the nearest point on the model's surface. The projection of the triangles on the image is performed via rasterization, i.e., by projecting their vertices, then using bounding box traversal to determine the pixels residing inside the projected triangles and, finally, calculating the distance of the model's triangles from each pixel. When
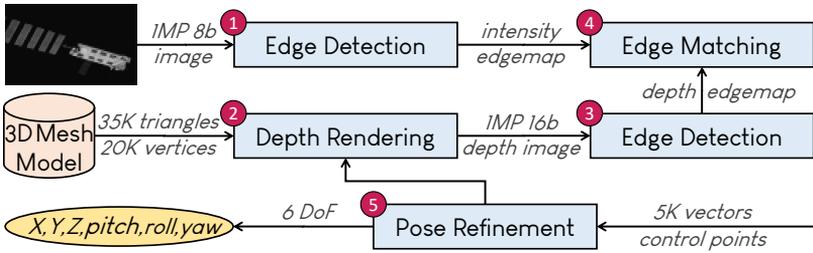
Figure 9.5: The 5-stage computer vision pipeline for satellite pose tracking [378].

multiple triangles project on the same pixel, the algorithm retains the projection that is closest to the camera.

Perpendicular Edge Matching finds the correspondences between intensity and depth edges. For each edge of the depth map, it searches along the gradient direction in the intensity map until an intensity edge is found or a maximum distance is covered. Finally, Pose Refinement utilizes the set of control points (matches plus spatial information) to determine the change of 6D pose between the two edge maps representing the previous and current frames. The change is determined in a robust regression framework that mostly involves linear algebra operations, e.g., SVD and QR decompositions. In an incremental fashion, the change is used to update the pose estimation.

9.5.2. Partitioning and Scheduling

The most central design choice when partitioning a CV algorithm in Myriad 2 is whether to accelerate each function in the SHAVE subsystem, execute it on LEON, or assign it to a hardware filter. The main criteria to select the mapping targets are: (a) performance and power constraints, (b) library dependencies, (c) parallelization amenability, and (d) memory access patterns. Regarding criterion (a), the use of SHAVEs provides better performance than the hardware filters, however, the latter attain better power efficiency. Criterion (b) regards the software complexity. For instance, it is preferable for the functions depending on software libraries (e.g., for linear algebra) to run on the general-purpose LEON processor in case of decreased support and availability in SHAVEs. Criterion (c) relates to the efficiency of the parallelization. For example, it may be preferable to execute an inherently sequential algorithm on LEON rather than parallelize it to SHAVEs. Finally, criterion (d) refers to the selection and the configuration of the available memories according to the access patterns (global or scratchpad, use of cache or not).
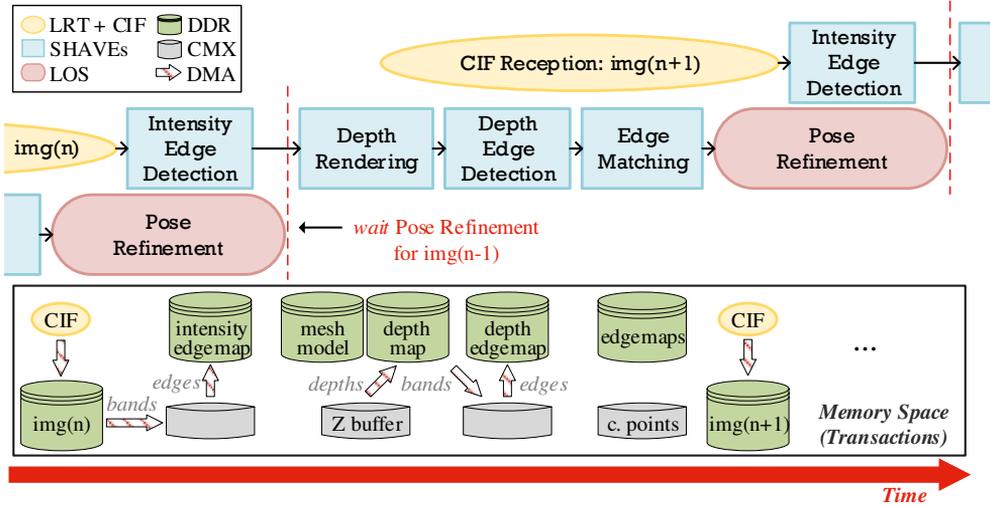
Figure 9.6: Partitioning, scheduling and memory transactions of the computer vision pipeline in Myriad 2.

The profiling and analysis of the CV pipeline based on our methodology and the above criteria results in the partitioning and scheduling shown in Figure 9.6. First, we avoid using the SoC's hardware filters, because we target maximum performance gain. Second, the profiling on LEON shows the very slow execution of the compute-intensive functions, i.e., Edge Detection and Depth Rendering, thus, we identify all the parallelization opportunities in their algorithmic nature and accelerate them on SHAVEs. Even though the Edge Matching function does not include demanding computations (it applies only image scanning and comparisons), we also map it onto SHAVEs. Otherwise, LEON would have to scan a 1024×1024 image, resulting in increased execution time. We note that we sequentially assign each function to all 12 SHAVEs instead of splitting them among the three functions. The latter would achieve questionable performance gains at the cost of significantly increased development effort for handling split memory and function synchronization. Regarding Pose Refinement, we assign it to LEON due to its dependencies on the BLAS/LAPACK libraries. For high-level scheduling, we follow the dependencies shown in Figure 9.5, while we maximize the function-level parallelization between SHAVEs and LEON. More specifically, we start detecting edges on a new input image even while LEON is still processing the previous frame to output its pose. When both are finished, we start the execution of Depth Rendering for the current frame. Finally, we consider that Myriad 2 receives the input frames from its Camera Interface (CIF). In our scheduling, the reception of each new frame is performed in parallel to the main processing by the hardware CIF peripheral of the SoC.

9.5.3. Development of Utility Software

The purpose of developing a utility software is to increase the productivity and enable sophisticated parallel programming over MDK, i.e., the VPU development tool, which demands low-level coding when targeting custom implementations. For example, extra development is required when multiple diverse functions must be assigned to SHAVEs for non-embarrassingly parallel execution. Our utility software creates an abstraction layer for handling the I/O peripherals, memory management, task scheduling, and inter-process communication. It includes a set of lightweight, standalone, and transparent C/C++ libraries, which, like the other MDK components, are included at compile-time and alleviate the coding/testing effort from the main development. Without this software, the implementation of the 5-function CV pipeline would not be efficient (or even feasible at all). Overall, our software modules extend MDK by introducing new mechanisms, improving the performance of the existing ones, and providing automatic/transparent device configuration. More details about our utility software for the Myriad VPUs can be found in our publications in [359, 363].

Figure 9.7 presents the basic mechanisms of our utility software. The first mechanism, shown in Figure 9.7a, provides inter-process communication using hardware mutexes and a variable size buffer per SHAVE, which is placed in the shared CMX memory slice. For data exchange, the SHAVEs read and write the buffers. The second mechanism, shown in Figure 9.7b, pre-loads the code of each function in DDR and allocates the requested CMX memory at runtime, i.e., during the execution of the function, based on a pointer technique. The third mechanism, shown in Figure 9.7c, aims to reduce the idle core time that is caused by the static task assignment. For embarrassingly parallel workloads, the developer creates a task pool by splitting the workload into small (independent) tasks, which are inserted in a FIFO struct and assigned to SHAVEs at runtime. Namely, the first task of the FIFO is assigned to the first available SHAVE, and hence, each SHAVE is immediately assigned a new task upon completing its previous one.

9.5.4. Parallelization and Low-Level Optimization

This section reports the implementation details on SHAVEs in two interdependent stages: (i) core parallelization, (ii) low-level mapping and optimization within each core. All the functions employ our memory allocation mechanism in order to use the common scratchpad CMX memory. More specifically, before the execution of each function, all the necessary working buffers are allocated, and correspondingly, after the execution they are freed.
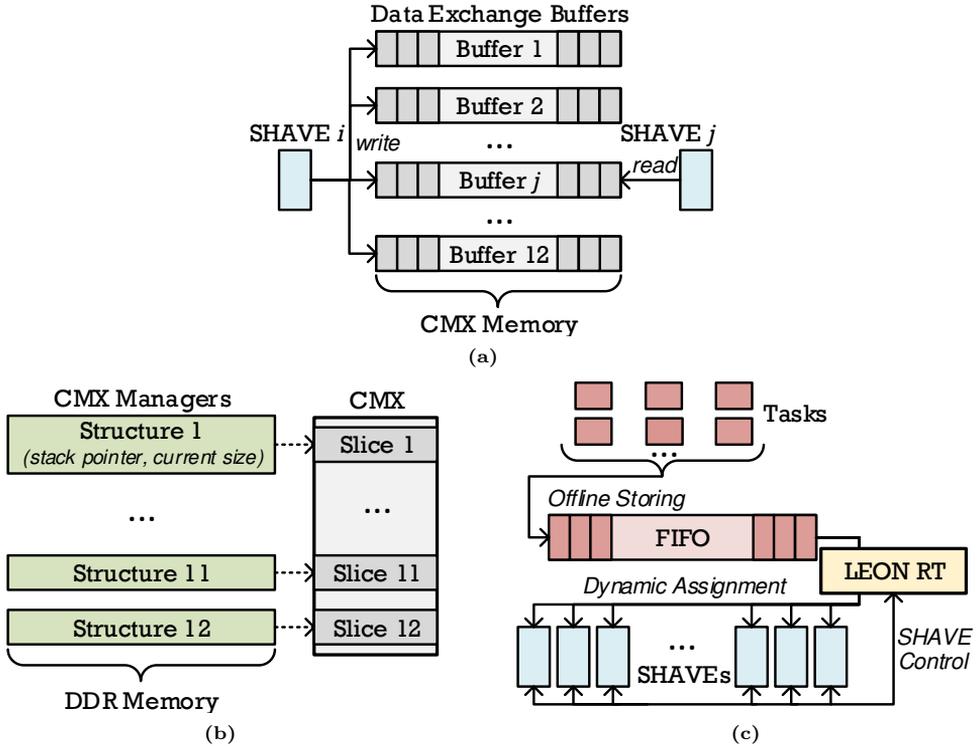
Figure 9.7: Custom utility mechanisms for development support in Myriad 2: **(a)** inter-process communication, **(b)** memory management, and **(c)** dynamic task assignment.

Canny Edge Detection

To provide improved workload balancing among the 12 SHAVEs, we split the input image in half, divide each half-image into 12 slightly overlapping stripes, and assign each one to a distinct SHAVE. In total, each SHAVE processes two 1024×43-pixel stripes from remote parts of the image, which have decreased correlation in terms of content.

The initial porting of Canny Edge Detection utilized a large memory, e.g., at least 4 buffers for the images and the derivatives, as well as extra 32-bit integer buffers for the histogram calculation and the execution of hysteresis thresholding. In our custom embedded implementation, we adopt in-place processing with reduced buffering to best fit in the 2MB CMX memory. That is, we interleave the loops for calculating the gradients and performing non-maximum suppression, such that with extensive memory reuse, we rely only on the input image buffer and two small buffers of size 1024×3 to slide all 3×3 kernels. Furthermore, we calculate the histogram of each

stripe inside this convolution loop (while generating the gradients). Another low-level knob regards the optimization of the buffers via word-length tuning and data type adaptation to work with the smallest required data size. Besides CMX, this customization exploits the capability of the VPU to process various data types (e.g., 16-bit integers instead of 32-bit). Upon the loop completion, the image buffer stores all local maxima and the hysteresis thresholding task is executed. This recursive procedure re-labels the weak edges neighboring any strong edge. The tracing continues until all connecting paths are followed. In our implementation, each SHAVE executes a local hysteresis in its own stripe and afterwards, it exchanges the strong edges of the stripe borders with its neighboring SHAVEs using our inter-process communication mechanism. After updating their border edges, SHAVEs execute another iteration of hysteresis specifically for those pixels. The last (low-complexity) task is executed on LEON and calculates the entire histogram of the gradients by accumulating the 24 stripe histograms (returned to DDR from CMX). This process also determines the hysteresis thresholds for the next frame.

Depth Rendering

For this function, we divide the image into horizontal R-row stripes, which are assigned to distinct SHAVEs to be rendered almost independently from each other. That is, we create a pool of $B = 1024/R$ individual tasks, where B determines the CMX memory resources per SHAVE and the DMA transactions (each rendered stripe is sent to DDR). The full utilization of CMX requires $B = 18$ stripes, however, our exploration shows that $B = 32$ improves the execution. Namely, smaller stripes adapt better to image content and distribute the workload more fairly to SHAVEs, even though larger B increases the repetitions of model reading. To reduce the idle time per core, we employ our mechanism for dynamic task assignment: each SHAVE is assigned a new task (i.e., stripe to render) from the pool at runtime, immediately upon finishing its previous stripe.

After exploration, we store the static Envisat's mesh model in the DDR global memory instead of CMX. The scratchpad is small and is preferred only for storing working buffers. Each SHAVE gets direct access to the full model in DDR, and hence, multiple cores read the same set of memory locations repetitively. However, due to their shared L2 cache, we measure negligible time penalties with this approach (compared to storing the model in CMX). Regarding the low-level optimization in SHAVEs, we reduce the number of buffers and customize the data types. In particular, we do not employ the working buffers used in typical CPU implementations, and ultimately, we utilize only the Z-buffer containing the rendered image stripe. Furthermore, we successfully enable SIMD operations to accelerate amenable functions (almost half of total computations), such as projection, bounding, and depth calculation. In detail, we arrange

the data using the Clang Intrinsic vectors, e.g., `float4` for defining a vector for 4 float numbers, and we call the corresponding MDK routines to calculate the dot/cross products (`mvuDot`, `mvuCross`) and find the min/max numbers (`mvuMax`, `mvuMin`). Moreover, we transform extra multiplications and additions to dot and cross products to expand our parallel/vectorized operations.

Perpendicular Edge Matching

This function searches for matches between the intensity and depth edgemaps. We divide the image into 12 independent stripes and assign each stripe to a distinct SHAVE. After exploration, we opt to store both the depth and intensity edge maps in DDR, while we enable the L1 (including data) and L2 caches of SHAVEs. Considering that Edge Matching relies on simple data comparisons (fast computations), the processing time on SHAVEs cannot mask the overhead of transferring the edges to CMX via the DMA engine. We store the output matches in the uncached CMX, which is directly accessed by LEON (to start the execution of the next function, i.e., Pose Refinement), in order to avoid disturbing the caching mechanism fetching the input edges. The race conditions to the shared output buffer are resolved by fine grained locking using the SoC's hardware mutexes.

9.6. Inference of Deep Neural Network on the Myriad X VPU

In this section, we present the inference of a demanding DNN on Myriad X, and more specifically, on the NCS2 USB accelerator via OpenVINO[5]. In particular, we accelerate a DNN of ResNet backbone, namely UrsoNet [379], which estimates the satellite's pose. To surpass the limited embedded computation power (compared to high-performance computers and host machines) and comply with the real-time constraints, we deploy a mobile version of UrsoNet. Moreover, to improve the DNN inference, we decrease the image resolution, using various resampling algorithms. More details about the resampling can be found in our publication in [362].

9.6.1. Deployment of DNN for Satellite Pose Estimation

The architecture of the UrsoNet DNN [379] is illustrated in Figure 9.8. In comparison with the original ResNet architecture, the global average pooling layer and the last

---

[5]Special thanks to P. Minaidis for coding on the DNN and NCS2.
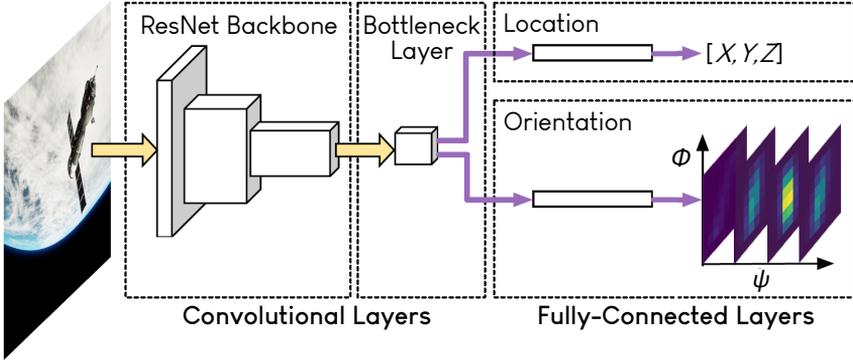
Figure 9.8: The UrsoNet DNN for satellite pose estimation [379].

Table 9.2: Configuration of the UrsoNet DNN for deployment on Myriad X VPU (NCS2).

Network		Training	
Parameter	**Value**	**Parameter**	**Value**
Backbone	ResNet-50	Pre-Trained Weights	ImageNet
Bottleneck Width	32	Dataset	"soyuz_hard"
Input Image	1024×1024×3	Arithmetic	fp32
Resampling	Bilinear/Bicubic/Lanczos	Epochs	100
Inference Image	512×512×3	Augmentation	No
Ori./Loc. Resolution	16/16	Optimizer	SGD

fully-connected layer are removed. In their place, the designers of UrsoNet insert a bottleneck layer that consists of a 3×3 convolution with stride 2, as well as two fully-connected layers for calculating the satellite's location and orientation. To generate a mobile network for deployment on Myriad X, we adopt the UrsoNet configuration presented in Table 9.2. We use the dataset for the Soyuz spacecraft ("soyuz_hard") and follow the training process of [379], but we start from a pre-trained ImageNet model, perform training for 100 epoques, and do not apply data augmentation. The image resolution is decreased using different resampling algorithms. We consider an 1024×1024×3 input image, which is scaled to 512×512×3 for efficient deployment and inference at the edge.

To generate the binary network file and deploy it on Myriad X (NCS2 accelerator), we follow the OpenVINO toolflow discussed in Section 9.2.2. Figure 9.9 illustrates the mapping of the basic network blocks of UrsoNet on Myriad X. Compared to the original ResNet architecture, the batch normalization blocks are replaced by add operations, which are accelerated on SHAVEs. Smaller operations, e.g., permutations before the fully-connected layers, are also executed on SHAVEs. The convolutions and
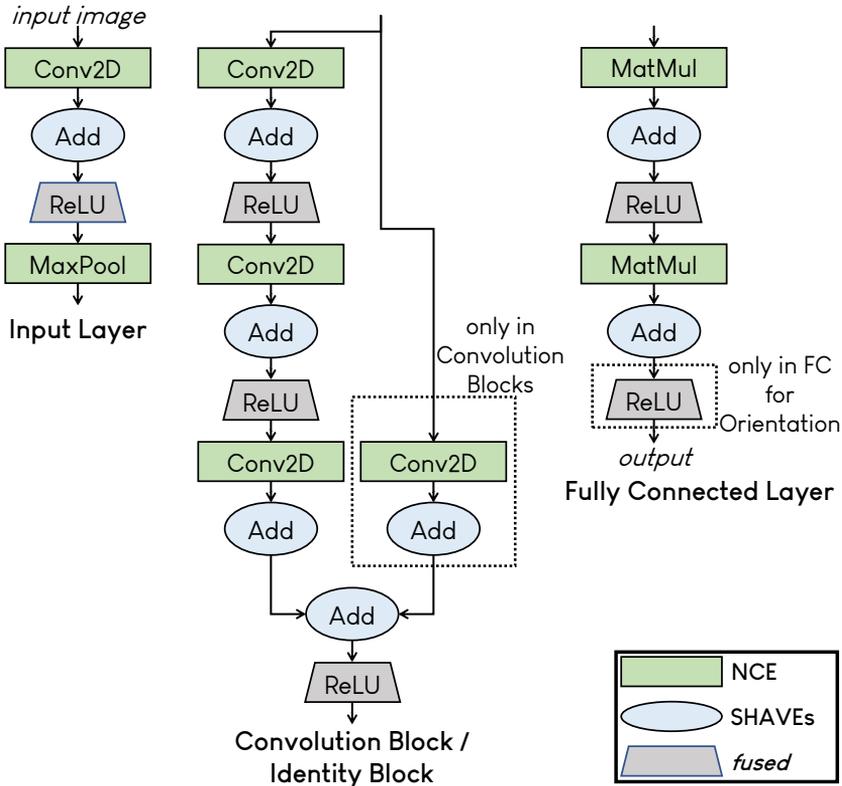
Figure 9.9: Mapping of the UrsoNet DNN in Myriad X.

matrix multiplications are mapped onto NCE, while almost all the ReLU activation functions are optimized out by OpenVINO, i.e., fused with other operations during the graph transformation stage.

9.7. Evaluation

This section conducts the evaluation of our development on the VPUs. Section 9.7.1 reports results from the implementation of the DSP and CNN kernels on Myriad 2 (presented in Section 9.4). Section 9.7.2 regards the implementation of the CV algorithm for pose tracking on Myriad 2 (presented in Section 9.5). All these algorithms are developed with the MDK tool. For their evaluation, we consider an FPGA–VPU co-processing architecture, in which the FPGA sends/receives the I/O data to/from the VPU. We also employ a host-PC for validating and demonstrating the results. Finally, Section 9.7.3 evaluates the inference of the DNN for pose estimation (pre-

Table 9.3: Overview of Dissertation's DSP & AI VPU accelerators.

Design	Input Data	Output Data	VPU	Reference
Averaging Binning	4-MPixel, 8b	1-MPixel, 8b	Myriad 2	Sec. 9.4, 9.7.1
Fl. Point Convolution	1-MPixel, 8b	1-MPixel, 8b	Myriad 2	Sec. 9.4, 9.7.1
ShipDetect CNN	1-MPixel×3, 16b	64×1, 32b	Myriad 2	Sec. 9.4, 9.7.1
Canny Edge Detection	1-MPixel, 8/16b	1K–5K, 5b	Myriad 2	Sec. 9.5, 9.7.2
Depth Rendering	6×1, 32b	1-MPixel, 16b	Myriad 2	Sec. 9.5, 9.7.2
Edge Matching	1-MPixel×2, 8&16b	1K–5K, 8b	Myriad 2	Sec. 9.5, 9.7.2
UrsoNet DNN	1-MPixel×3, 8b	4099×1, 32b	Myriad X	Sec. 9.6, 9.7.3
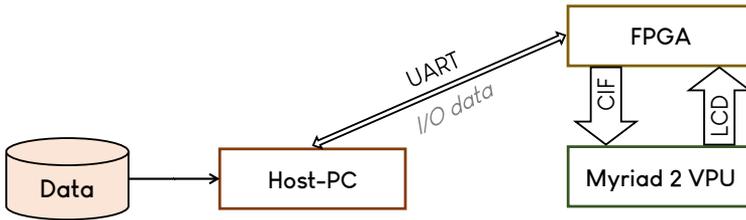
Figure 9.10: FPGA–VPU co-processing architecture for the acceleration of DSP/CNN kernels on Myriad 2.

sented in Section 9.6). For this experimentation, we use the OpenVINO tool and the Myriad X NCS2 accelerator. Table 9.3 summarizes all the VPU implementations of the Dissertation along with their I/O data.

9.7.1. Experimental Results of Custom DSP and CNN Kernels

Figure 9.10 illustrates our testing setup for the evaluation of the custom DSP and CNN kernels. The host-PC is connected with an FPGA, which transmits/receives the I/O data to/from Myriad 2. Myriad 2 receives the input data via CIF, as in our CV pipeline (see Section 9.5.2) and sends the output to the FPGA via the Liquid Crystal Display (LCD) interface[6]. More details about our FPGA–VPU architecture, the components/functions of each device that enable the communication, and the I/O transfers are reported in our publication in [360].

Acceleration of Kernels

Firstly, we assess the acceleration of our kernels using the general-purpose LEON processor as baseline. For Averaging Binning, we achieve 14× speedup, which mainly

---

[6]Special thanks to Prof. D. Reisis *et al.* from NKUA for coding on the CIF/LCD interface.
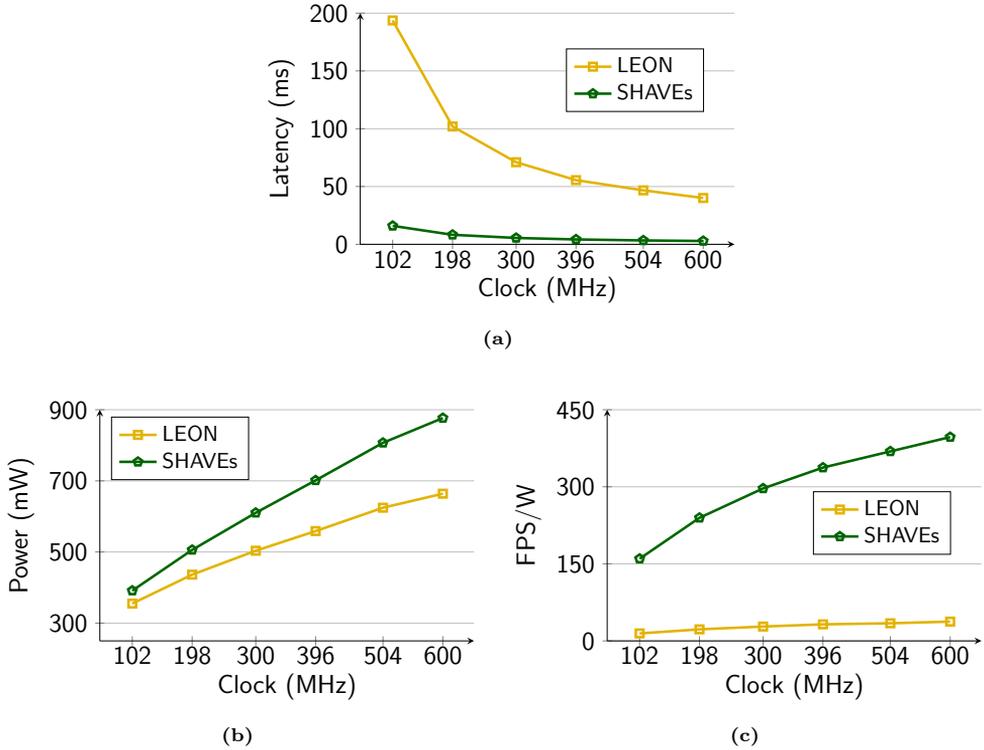
Figure 9.11: Experimental results from the implementation of Averaging Binning on Myriad 2 with respect to the clock frequency: **(a)** latency, **(b)** power consumption, and **(c)** FPS-per-Watt.

comes from the parallelization to 12 SHAVEs (LEON has to scan the entire 4-MPixel image, resulting in significant delay). Depending on the size of the convolution mask, we achieve up to 75× speedup for Floating-Point Convolution. Compared to Averaging Binning, the speedup is larger, because the convolutions require more demanding computations. Finally, the Ship Detection CNN is implemented only on SHAVEs, i.e., where the custom inference engine is mapped, however, considering the performance of convolutions, and given that LEON does not support 16-bit floating-point (thus, it must execute the 32-bit model), the speedup is expected to be more than 2 orders of magnitude.

Figure 9.11 illustrates the scaling of latency, power consumption, and performance-per-Watt of Averaging Binning for different clock frequencies. Regarding latency, it is almost linear to the clock frequency whether Averaging Binning is executed on LEON or the 12 SHAVEs. The SHAVE implementation is ~13× faster than the respective LEON implementation in all experiments. In terms of power, as expected, LEON

provides $1.1\times$–$1.3\times$ smaller consumption, however, SHAVEs still deliver low power, i.e., up to 900mW. Moreover, the power consumption of the SHAVE implementation increases faster than that of LEON. When considering both Frames Per Second (FPS) and power, LEON attains negligible scaling, while the FPS-per-Watt of SHAVEs increase almost linearly. Similar results are derived for the Floating-Point Convolution, i.e., up to 900mW and $58\times$ speedup.

System Evaluation

We also evaluate the system performance involving both I/O and processing based on the FPGA–VPU co-processing architecture of Figure 9.10, where the data transfers are performed via the CIF and LCD interfaces. In this analysis, the clock frequency of CIF and LCD is configured at 50MHz, which guarantees error-free data transfers according to our experiments presented in [360]. The evaluation regards two distinct scenarios concerning the execution order of the I/O handling and processing tasks:

1) Unmasked I/O: assuming serial I/O–processing, the VPU receives the input frame from the FPGA, performs the processing, and transmits the output data to the FPGA.

2) Masked I/O: assuming pipelined I/O–processing and streaming input, the VPU performs in parallel two processes: (i) buffering of output frame $n-1$, CIF reception and buffering of input frame $n+1$, LCD transmission of output frame $n-1$, and (ii) processing of frame n. In that case, the one LEON processor executes the first process, i.e., the I/O handling, and the other takes over the second process, i.e., it manages the processing performed by SHAVEs.

The performance results are presented in Table 9.4. Both I/O interfaces are operating at 50MHz and as expected, they transmit an 1-MPixel image in ~21ms. In the Unmasked I/O mode, the total throughput ranges between 9–20 FPS for kernels with small processing time. To implement the Masked I/O mode, the I/O data are buffered to an allocated DDR space for data integrity reasons (copying an 1-MPixel image requires ~42ms). As a result, the latency of a single frame increases considerably. Even so, the kernels featuring excessive processing time can benefit from this masking technique and improve their throughput by $1.1\times$–$1.3\times$. This effect is shown in the Floating-Point Convolution with a 13×13 mask and the Ship Detection CNN. In contrast, kernels with small processing time suffer a throughput decrease when applying masking, and the developer must be cautious with respect to the selected mode of operation. This is evident in Averaging Binning, which has only 3ms computation latency and the buffering of its 4-MPixel input image adds considerable timing overhead.

Table 9.4: Experimental results of custom DSP & CNN kernels on Myriad 2 VPU.

Kernel	Function Latency[1]			Sys-Unmasked I/O[2]		Sys-Masked I/O[3]	
	CIF (ms)	VPU (ms)	LCD (ms)	Latency (ms)	Throughput (FPS)	Latency (ms)	Throughput (FPS)
Avg Binning	85	3	21	109	9.1	906	3.2
3×3 Conv.	21	8	21	50	20	336	8
7×7 Conv.	21	29	21	71	14.1	336	8
13×13 Conv.	21	114	21	156	6.4	336	8
ShipDetect	63	658	~0	721	1.4	1505	1.5

[1] Refers to the input reception via CIF, the processing on VPU, and the output transmission via LCD.

[2] Regards serial I/O–processing → **Throughput** = 1/(CIF_Time + VPU_Time + LCD_Time).

[3] Regards pipelined I/O–processing → **Throughput** = 1/(max{VPU_Time, LCD_Buffering_Time + CIF_Time + CIF_Buffering Time + LCD_Time}).

Comparison to Embedded Devices

Finally, we compare our VPU accelerators with other embedded devices. For the same CNN model, Myriad 2 provides ~2.5× less FPS-per-Watt than the Zynq-7020 FPGA [276], however, the latter utilizes almost all the chip resources and exhibits 4× larger power consumption. Compared to the Jetson Nano GPU [276], the VPU delivers ~4× better FPS-per-Watt for the CNN. For Averaging Binning, we achieve ~3× better throughput than a typical Zynq FPGA implementation with one binning pipeline in programmable logic (one input pixel per clock cycle), also due to the slower DMA engines of Zynq.

9.7.2. Experimental Results of CV Pipeline

Figure 9.12 illustrates our testing setup for the evaluation of the CV pipeline. In this setup, we employ EGSE [392], which comprises an interface unit for the real-time simulation of the high-bandwidth SpaceWire link (i.e., a spacecraft communication network for data transmission). The host-PC configures EGSE, which feeds the input data to the FPGA. The FPGA transmits the data to Myriad 2 via CIF operating at 5MHz. For validation and demonstration[4] purposes, we transfer the outputs of the CV pipeline (6D poses) back to the host-PC. More details about the FPGA–VPU architecture, the configuration of SpaceWire, and the I/O transfers are reported in our publication in [359].

---

[4]Demo available on YouTube: **https://youtu.be/9wDLm56zsss**.
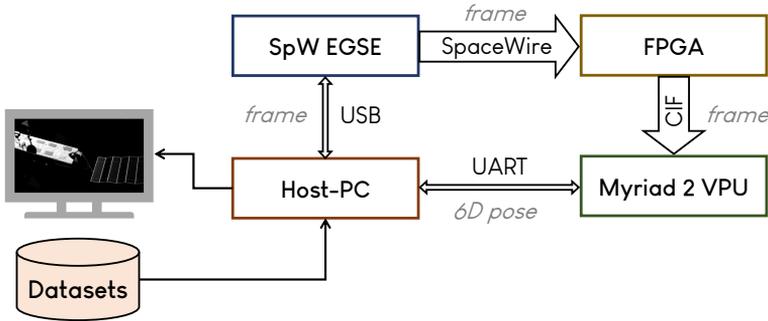
Figure 9.12: FPGA–VPU co-processing architecture for the acceleration of the computer vision pipeline on Myriad 2.
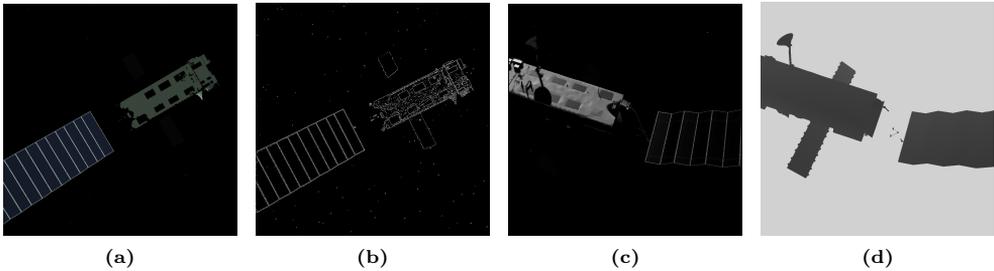
Figure 9.13: I/O data of main functions in the computer vision pipeline (30m–20m Envisat sequence): **(a)**,**(c)** input, **(b)** output of Edge Detection, and **(d)** output of Depth Rendering.

We test our system with a synthetic dataset[5] that includes two sequences of 1000 images (1024×1024 8-bit grayscale pixels) realistically simulating the motion of Envisat. The nature of the data is similar to that of actual rendezvous images and suffices to evaluate the system performance. In particular, the first frame sequence depicts a satellite rotation out of plane at ∼50m away from the camera, while the second one depicts a satellite approach from 30m to 20m while tumbling. Figure 9.13 illustrates frames from the 30m–20m dataset, together with outputs from the basic functions of the CV pipeline.

Acceleration of Functions

At first, we evaluate the implementation of each function of the CV pipeline. In particular, targeting to assess our development methodology and embedded implementation techniques, we gradually study the speedup factor. Figure 9.14 presents

---

[5]Special thanks to D. Gonzalez-Arjona *et al.* from GMV for providing the test dataset of Envisat.
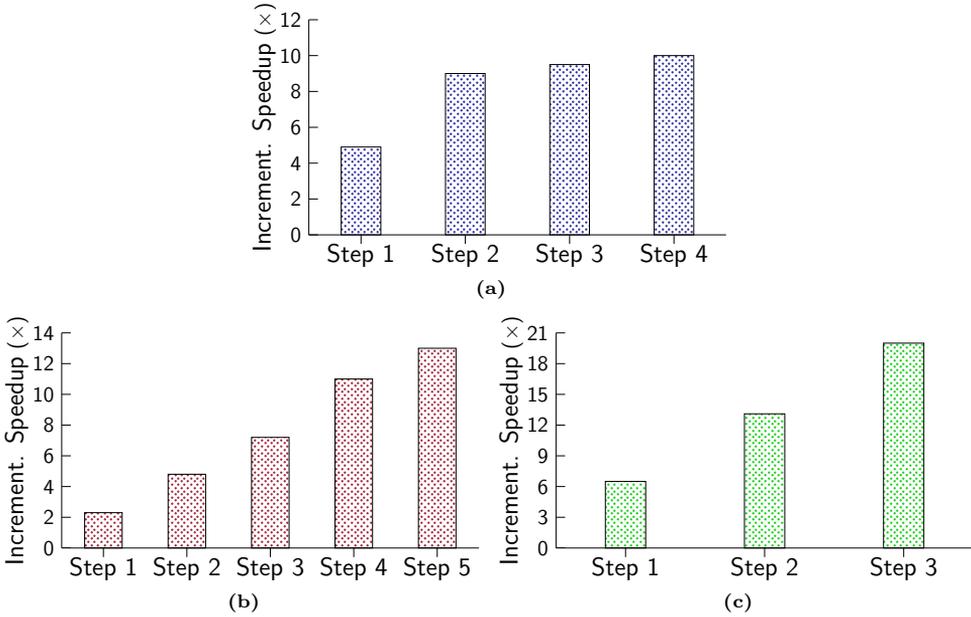
Figure 9.14: Incremental acceleration of the computer vision functions with respect to their major implementation steps on Myriad 2: **(a)** Edge Detection[1], **(b)** Depth Rendering[2], and **(c)** Edge Matching[3].

[1] Step 1: SHAVE porting, Step 2: improved buffering and in-place computations, Step 3: loop merging, Step 4: improved task partition.

[2] Step 1: SHAVE porting, Step 2: dynamic task assignment, Step 3: improved task partition, Step 4: SIMD computations, Step 5: optimized cache configuration.

[3] Step 1: SHAVE porting, Step 2: shared L2 and instructions at L1, Step 3: shared L2 per 2 cores and instructions & data at L1.

how each major implementation step contributes in the speedup for the 30m–20m Envisat dataset. Namely, the speedup of the last step is the final speedup achieved. Below, we analyze the implementation steps and the speedup of the CV functions accelerated on SHAVEs.

The parallelization of Edge Detection to 12 SHAVEs delivers only 4.9× speedup, regardless of the pixel bit-width. The re-design of the function using improved buffering and in-place computations almost doubles the speedup to 9×. By performing the thresholding before hysteresis and by merging loops (histogram calculation during thresholding), the speedup is increased to 9.5×. Dividing the image to 24 stripes rather than 12 adds negligible overhead due to extra DMA transactions, however, it improves the workload balancing among SHAVEs during the content-dependent hysteresis task and results in the final speedup of 10×.

The parallelization of Depth Rendering to 12 SHAVEs delivers only 2.3× speedup

when using static task assignment. We note that the performance of this function varies among frames, as it is highly dependent on the image content. By employing dynamic task assignment, the idle time per SHAVE core significantly reduces, and thus, the speedup is increased to 4.8×. Moreover, the image partition into more stripes results in more fine-grained workload balancing and provides an additional speedup boost to 7.2×. Subsequently, the exploitation of the SIMD capabilities of SHAVEs enlarges the speedup by 9×–12×. Finally, the read-only nature of the data allows cache optimizations that decrease the execution time even further, to about 119ms–212ms. In total, depending on Envisat's orientation in the image, the Depth Rendering functions attains a speedup of 10×–16×.

Out of all functions, the configuration of the memory hierarchy has the highest impact on the performance of Edge Matching. Its parallelization to 12 SHAVEs provides a remarkable speedup of 6.5×, however, our cache configurations result in even larger speedup. In particular, the employment of the shared L2 and L1 cache for instructions increases it to 13.1×. By enabling a shared L2 cache of 16KB per two SHAVEs and using the dedicated L1 also for data, the speedup increases to 20×.

System Evaluation

Table 9.5 reports the main experimental results from the implementation of the CV pipeline in Myriad 2. It includes results from the profiling on LEON and the final SHAVE implementation for the 30m–20m Envisat dataset. Similar results are derived for the 50m Envisat dataset. Nevertheless, this dataset imposes slightly less computation demands (the workload is content-dependent), as the distance of the satellite from the camera is larger, and thus, its size is smaller.

The execution times in Table 9.5 vary during the test sequence, because the algorithmic workload is content-dependent, i.e., it is affected by the apparent size of Envisat in the image. In addition, the efficiency of the task parallelization depends on the orientation of Envisat due to the mapping of rendering areas to SHAVEs. We note that the image reception via CIF operating at 5MHz requires ~210ms, however, in our final implementation it is entirely masked by the processing of the previous frame (see Figure 9.6). Hence, the system speedup increases to 8.5×–12× (Table 9.5 does not take into account the I/Os). Overall, the achieved throughput is 2.6–3.8 FPS for the 20m–30m sequence (325ms per frame on average), while for the 50m sequence, we achieve a throughput of 3.8–4.9 FPS (235ms per frame on average).

Regarding power consumption, the MDK routines report 0.7W–1W (see Table 9.5) for the execution of each individual function on SHAVEs, with Depth Rendering be-

Table 9.5: Experimental results of the computer vision pipeline on Myriad 2 VPU.

Function	LEON		SHAVEs				Gains	
	Latency (ms)	Memory (MB)	Latency (ms)	Memory (MB)	Power (mW)	Accuracy (%)	Latency (×)	Memory (%)
I. Edge Det.	370–375	6.1	36–37	2.3	771	99	10	62.9
D. Edge Det.	375–380	7.1	39–40	3.8	771	99	10	47.3
Depth Rend.	1900–2100	6.4	119–212	3.3	980	100	10–16	48.6
Edge Match.	100–120	3	5–6	3	810	100	20	0
Pose Refin.	100–130	6.3	100–130[1]	6.3	644	Fig. 9.15	1	0
Pipeline	2845–3105	–	263–388[2]	–	898	Fig. 9.15	8–11[3]	–

[1] Pose Refinement was not accelerated on SHAVEs.
[2] Intensity Edge Detection is not added in the total latency (masked by Pose Refinement).
[3] Without I/O. With image reception via CIF (∼210ms), the speedup increases to 8.5×–12× (CIF is masked).

ing the most power-hungry. In comparison with the porting on LEON, these values are slightly increased by 1.2×–1.4×, as LEON consumes 600mW–700mW depending on the function. On average, when the entire CV pipeline operates, we get 0.9W. For the execution of the full system, i.e., when including the image reception via CIF, the power consumption lies between 0.8W and 1.1W. We note that our board measurements via an external multimeter report a power consumption of 1.2W.

In terms of memory, the integration of the entire CV pipeline along with the CIF module results in small utilization. The total memory (data and instructions) is around 20MB, i.e., smaller than the 10% of the available DDR resources. Furthermore, by storing the instruction code in DDR, there are no timing penalties for the execution on SHAVEs, and CMX is used only for data and internal structures. Additionally, due to the applied optimizations, i.e., improved buffering and in-place processing, we achieve significant memory gains: 55% on average for the two Edge Detection functions and 49% for Depth Rendering (see Table 9.5).

In terms of individual accuracy, the functions generate the same sets of results as the initial software. A small exception is in Edge Detection, which misses 1% of the edges when executing slightly fewer hysteresis recursions at the stripe borders (compile-time configurable). We also note that for Pose Refinement, we port on LEON a relatively old version of BLAS/LAPACK (v3.2.1, without guaranteed compatibility to the initial software). Figure 9.15 shows the alignment error of Envisat's pose, as it is defined in [326]. The error versus Envisat's distance is in the area of 1% and tracking is lost only in few specific frames. In general, the behavior of pose tracking on Myriad 2 is similar to that of the CPU and FPGA implementations [326].
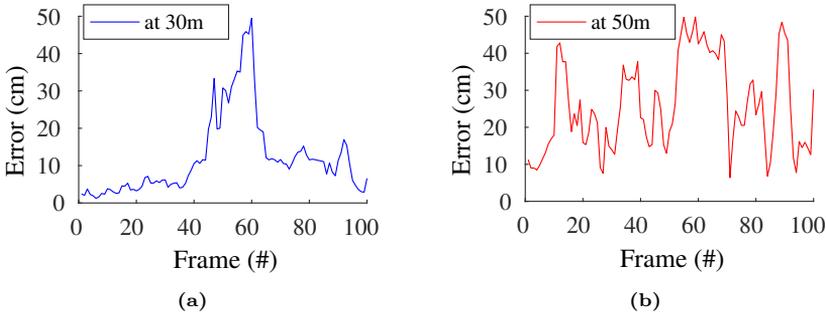
Figure 9.15: Alignment error of Envisat's pose computed by the computer vision pipeline in Myriad 2 at **(a)** 30m and **(b)** 50m camera distance.

Table 9.6: Data volume reduction in the implementation of the computer vision pipeline in Myriad 2.

VPU Input		VPU Output	
Data Type	1024×1024 8-bit frames	Data Type	6×1 32-bit vectors
Data Volume	8MB per frame	Data Volume	192b per vector
Reception Time	~210ms	Processing Time	~325ms | ~235ms
Reception Rate	~38 Mbps	Processing Rate	~590 bps | ~817 bps

Finally, in Table 9.6, we evaluate the efficacy of the proposed architecture as an edge processor by examining the data volume reduction. Interestingly, we achieve the very important goal of data reduction at the edge, which relieves several potential bottlenecks concerning the storage, the network's bandwidth and energy consumption, as well as the I/O throughput. In particular, considering the input and output data, i.e., 1024×1024 8-bit images and 6×1 32-bit vectors, respectively, as well as the latency of the input reception and processing, we achieve a data reduction factor of 10^5 bps.

Comparison to Embedded Devices

Finally, we provide an overall evaluation of the Myriad 2 VPU, including comparisons to other embedded devices. When comparing individually the implementation of the functions to other ARM-based or GPU-based mobile devices, we verify the power efficiency of Myriad 2. For Canny Edge Detection, we achieve 1 order of magnitude faster execution than ARM Cortex-A9 (~326ms per MPixel image, single-threaded at 667MHz) [326], and approximately 3× better performance-per-Watt than Nvidia's Jetson TK1 GPU (~12ms per MPixel image, 192 CUDA cores, but with 10W) [393].

For Depth Rendering, we achieve ~7× faster execution than ARM Cortex-A9 (~869ms per 30m-distant-image, single-threaded at 667MHz) [326] and 5×–11× better power consumption than high-end mobile GPUs (given their nominal values, see Table 9.1). Regarding system performance, our efficient SoC utilization provides tenfold acceleration versus LEON4@600MHz. The achieved FPS approach 5 and can be further improved via customization at algorithmic level, e.g., smaller rendering, which is generally considered as sufficient for VBN in space [3]. When comparing to Zynq-7000 implementing the same algorithm [326], the FPGA processes approximately 3× more FPS than Myriad 2, in total. For specific functions, i.e., Edge Detection or Depth Rendering, Zynq also achieves 6×–14× faster execution.

Regarding the power consumption, the 0.8W–1.1W of Myriad 2 is significantly lower than that of other high-performance processors, with only few CPUs consuming ~1W by operating at significantly lower clock frequency. Based on our methodology, we exploit all the capabilities of the tools and SoC to deliver a very high performance-per-Watt ratio in Myriad 2, which matches the notable FPGA results measured in [3,326] for such tasks. Compared to Zynq [326], Myriad 2 achieves the same, or slightly better, performance-per-Watt by trading approximately 3× speed for 4× mean power. Additionally, it has a more stable consumption than the 2W–9W variation of the FPGA, allowing for simpler electronics design.

Another advantage of Myriad 2 over the FPGA is that it utilizes less than 10% of the available memory space, allowing to implement additional functions or alternative algorithms for the current functions. In contrast, the same FPGA implementation utilizes 28%–77% of the chip [326], thus, it would require dynamic reconfiguration to change functionality. In terms of accuracy, the Myriad 2 processor can meet exactly the same arithmetic requirements as an ordinary CPU during the entire algorithm execution. This can be viewed as an advantage in cases where the original algorithm, which is usually developed on CPU, must remain 100% intact with respect to its intermediate and output values.

9.7.3. Experimental Results of DNN Kernel

Acceleration of UrsoNet

In this section, we report the results from the deployment of the UrsoNet DNN on Myriad X. The latency of the pre-processing stage, which involves image resampling to transform the 1024×1024×3 input image to 512×512×3, is 1ms–20ms, depending on the algorithm. A single DNN inference (synchronous execution) on the 512×512×3 image requires around 373ms, which is 5× faster than inferencing on the initial image. At the same time, the mean Location Error (LOCE) is 1.68m–1.89m and the mean

Orientation Error (ORIE) is 25.08°–27.89°, namely both remain in the range of the original DNN without resampling, where LOCE is 1.78m and ORIE is 27.83°. In all cases, as shown in [379], more fine-grained training, which is not the purpose of our work, can provide better accuracy results for the same performance. We remind that we apply different training settings (e.g., another pre-trained model, less epochs) and architectural modifications (e.g., decreased resolution for location and orientation, and smaller bottleneck layer). Overall, the pre-processing stage for image resampling, which is parallelized to 16 SHAVEs, adds very small latency, however, it facilitates the inference stage. As a result, the entire synchronous execution sustains a throughput of ~2.7 FPS. The throughput is expected to increase for asynchronous inferencing, while further speedup can be achieved by inferencing on smaller tensor, e.g., $192{\times}256{\times}3$. In this case, Myriad X delivers 15 FPS, whereas the accuracy loss in not significant, as LOCE lies in the range 1.9m–2m.

Comparison to Embedded Devices

Following the analysis of the experimental results on Myriad X, we provide a comparison to other embedded devices that are also considered for space avionics. Table 9.7 presents the results for inferencing on VPU, CPU, and GPU. For this experiment, we use input images that are scaled to $512{\times}640{\times}3$. The CPU is the ARM Cortex-A57 of Nvidia's Jetson Nano board, while the GPU is the 128-core Maxwell of the same board. For Jetson Nano, we consider its two power modes: the "low-power" at 5W and the "high-performance" at 10W. To make a fair comparison, we consider the NCS2 VPU hosted on a single-board computer (i.e., a Raspberry Pi 3), and thus, the total power consumption for inferencing is 5W. In terms of latency, the 4-core Cortex-A57 operating at 1.4GHz is relatively slow, thus, NCS2 delivers a $6{\times}$ speedup. Compared to the GPU, NCS2 achieves a slight improvement of $1.3{\times}$, but with half of the GPU's power consumption. When considering both throughput and power, NCS2 provides ~$1.7{\times}$ more FPS-per-Watt than GPU and ~$8.5{\times}$ more FPS-per-Watt than CPU. Finally, we report comparison results for the original ResNet-50 network ($224{\times}224{\times}3$ image), which is the backbone of UrsoNet. For ResNet-50, the Jetson Nano GPU provides increased throughput versus Pi 3 + NCS2, i.e., $1.3{\times}$–$1.9{\times}$ more FPS, however, when also considering FPS-per-Watt, the VPU inference outperforms the GPU by $1.1{\times}$–$1.5{\times}$.

9.8. Conclusion

In this chapter, we accelerated various DSP and AI algorithms on the multi-core Myriad VPUs. These SoCs are characterized by extremely low power consumption

Table 9.7: Experimental results of the UrsoNet DNN on embedded devices (512×640×3 input tensor).

Device	Latency (ms)	Throughput (FPS)	Power (W)	FPS-per-Watt
Intel Myriad X VPU (NCS2 + Pi 3)[1]	588	1.7	5	0.34
ARM Cortex-A57 CPU (Jetson Nano)[2]	2830	0.4	10	0.04
	7519	0.1	5	0.02
Nvidia Maxwell GPU (Jetson Nano)[3]	761	1.3	10	0.13
	958	1	5	0.2

[1] NCE & 16-core SHAVE @700MHz. NCS2 (2W) hosted on a Raspberry Pi 3 (3W).
[2] 10W mode: 4-core @1.4GHz. 5W mode: 2-core @918MHz.
[3] 10W mode: 128-core @921MHz. 5W mode: 128-core @614MHz.

(i.e., ~1W–2W) and increased heterogeneity in terms of processors and memories. As shown in practice, the efficient utilization of such an heterogeneous SoC architecture requires a design methodology involving algorithmic analysis, multi-level parallelization, and extensive low-level optimization and tuning, especially when having to deploy multiple diverse software kernels. Based on our methodology, at first, we implemented custom kernels, i.e., averaging image binning, floating-point convolutions, and a CNN for detecting ships on satellite images. Afterwards, we accelerated a sophisticated 5-stage CV pipeline for tracking the pose of the Envisat satellite, which includes kernels such as Canny edge detection, depth rendering and perpendicular edge matching. Moreover, we deployed the demanding UrsoNet DNN of ResNet-50 backbone for estimating the pose of the Soyuz spacecraft. For the implementations, we successfully applied various high- and low-level techniques such as dynamic task scheduling, SIMD operations, improved buffering, variable tuning, and optimized memory configurations. Our experimental evaluation on Myriad 2 shows that for individual kernels, we provided 10×–20× speedup compared to the general-purpose LEON4 CPU. At system-level, the entire CV pipeline for pose tracking on MPixel images provided a speedup of 8.5×–12×, while sustaining a throughput of 2.6–4.9 FPS with only 0.8W–1.1W. Regarding Myriad X, the inference of UrsoNet on resampled MPixel images delivered a throughput of 2.7 FPS within the power envelope of 2W. Finally, compared to other embedded devices, the Myriad VPUs provide significantly better power efficiency, i.e., 5× versus the Jetson Nano GPU and 4× versus the Zynq FPGA. In terms of performance, they are outperformed by the Zynq FPGA, however, when considering the performance-per-Watt ratio, they provide the same or even better results. Indicatively, for the CV pipeline, Myriad 2 trades a 3× loss in speed for a 4× gain in mean power consumption.

Chapter 10

Conclusion

10.1. Summary of Main Contributions

The goal of the current Ph.D. Dissertation was to design and evaluate DSP and AI accelerators that fulfill the requirements of modern computing systems. Towards reaching this goal, we adopted various design approaches from different layers of the computing stack. Starting from the bottom to the top design layer, our work involved arithmetic circuits, hardware accelerators, FPGA implementations and implementations on embedded SoCs.

At circuit level, we exploited the promising design paradigm of Approximate Computing to propose new approximation techniques for energy-efficient multipliers, which are key processing units in DSP/AI hardware accelerators. In this context, we examined several aspects of Approximate Computing, including low-level optimizations, hybrid encodings, runtime configurability, and cooperative approximation. Our approximate multiplication circuits provide a very large approximation space and slow error scaling in the typical acceptable error segment, while they outperform several state-of-the-art designs. All the proposed circuits were efficiently integrated in bigger DSP/AI hardware accelerators based on our design methodology.

Next, we provided acceleration and efficient mapping of DSP algorithms on the new European space-grade FPGAs, which require special treatment due to several factors (e.g., new tools, lower performance than commercial FPGAs). Our development was accompanied by a design methodology targeting to highlight all the acceleration and mapping opportunities, and also aid us to surpass issues that arose either by the new tool or due to HDL porting on a different FPGA vendor. The final resource utilization of high-performance algorithms for feature detection and stereo vision was comparable to that of well-established FPGAs, while the performance was sufficient for space applications.

Finally, we provided acceleration and efficient mapping of DSP/AI algorithms on the multi-core VPUs. These SoCs are very heterogeneous in terms of processors

and memories, however, several challenges need to be addressed to deliver increased acceleration. To offer very low power consumption, the VPUs have sacrificed computational power, while the SoC's complexity requires systematic study to efficiently map and schedule compute-intensive algorithms. Therefore, we proposed a design methodology and several high- and low-level implementation techniques to accelerate classic DSP kernels, as well as a sophisticated CV pipeline and a DNN algorithm (both for satellite pose estimation).

Overall, the **key contributions** of the Ph.D. Dissertation are summarized as follows:

- Extensive and up-to-date survey in the field of Approximate Computing, which reviews and classifies software and hardware approximation techniques (**Chapter 2**).

- Low-level optimizations in the alternative DLSB numerical format (**Chapter 3**).

- New arithmetic approximation techniques, which generate the RAD, AxFXU/ AxFPU and ROUP families of approximate multipliers (**Chapters 4–6**).

- Improvement (3 times) of the state-of-the-art energy-error Pareto front of approximate multipliers (**Chapters 4–6**).

- Seamless runtime configuration of the approximation in the DyFXU/DyFPU approximate multipliers (**Chapter 5**).

- The design approach of "cooperative approximation", which combines various orthogonal approximation techniques to provide a very large design (approximation) space (**Chapter 6**).

- Methodology for the development of approximate DSP and AI hardware accelerators, either for ASIC or FPGA, which is based on extensive design space exploration with differing approximations, algorithms, arithmetic formats, and hardware design techniques (**Chapter 7**).

- Experimental results for various approximate DSP and AI hardware accelerators, including kernels for 1D/2D signal processing and CNNs (**Chapter 7**).

- Methodology for the mapping and acceleration of high-performance DSP kernels on the new space-grade BRAVE FPGAs and tools (**Chapter 8**).

- Experimental results from the implementation of various DSP kernels (feature detection and stereo vision) on space-grade FPGAs of the market (**Chapter 8**).

- Methodology for the mapping and acceleration of high-performance DSP and AI algorithms on the heterogeneous multi-core Myriad VPUs (**Chapter 9**).

- High-level and low-level techniques for the partitioning, scheduling, and mapping of demanding algorithms on the heterogeneous multi-core Myriad VPUs (**Chapter 9**).

- Experimental results from the implementation of various DSP and AI kernels (convolution, image classification, pose tracking) on the heterogeneous multi-core Myriad VPUs (**Chapter 9**).

- Evaluation of NanoXplore's space-grade BRAVE FPGAs and Intel's COTS Myriad VPUs as candidate on-board processors for space missions (**Chapters 8–9**).

10.2. Future Work

Regarding the work of **Part I**, Approximate Computing is applied in all the layers of the typical computing stack. As a result, research is conducted at the transistor layer, circuits, hardware accelerators, micro-architecture, runtime systems, compilers, and programming languages. In this Ph.D. Dissertation, the proposed approximation techniques are applied at the logic level and the design of arithmetic circuits, which are then integrated in hardware accelerators. However, to exploit the full potential of Approximate Computing, our arithmetic approximation techniques can be combined with approximations inserted in other layers, namely, apply cross-layer approximation. Other future extensions include the integration of our approximate circuits in processors (e.g., in open-source GPUs or RISC-V CPUs), the system-level application of our runtime approximation configuration (e.g., to provide tuning with respect to the desired accuracy constraints), and the automatic selection of the best approximation configurations for a given application (which is currently performed manually).

Regarding the work of **Part II**, segments of our design methodologies can be performed in an automatic fashion, e.g., the exploration of the tool settings in the FPGA development. Our methodology about the programmable logic of the space-grade FPGAs can be extended to take into account the ARM processor of the SoCs, while an open issue is the successful integration of high-performance I/O links for data transmission. Our work in the VPUs can be extended towards the synergistic application of both classic CV and AI algorithms, in order to increase the system robustness, e.g., execute both the CV pipeline and the UrsoNet DNN for pose estimation/tracking. Moreover, we are already developing similar methodologies and performing benchmarking on other embedded platforms such as the TPUs. Finally, as our research activities with the FPGAs and VPUs target on-board computing in space, we are also working on equipping COTS devices (e.g., Zynq and Myriad) with fault-tolerant

mitigation techniques, in an effort to increase their reliability without affecting the DSP/AI performance.

Εκτεταμένη Περίληψη

1. Εισαγωγή

Οι ραγδαίες τεχνολογικές εξελίξεις στην αίσθηση, την επεξεργασία και την αποθήκευ-ση δεδομένων έχουν μεταμορφώσει το τοπίο των ενσωματωμένων συστημάτων. Με την εμφάνιση του Διαδικτύου των Πραγμάτων (IoT) [1], υπάρχει μια τεράστια αύξηση στον όγκο των δεδομένων που παράγονται, η οποία επιβάλλει τεχνικές προκλήσεις στις περιορισμένων-πόρων συσκευές επεξεργασίας. Επιπλέον, η μετάδοση όλων αυτών των δεδομένων σε υποδομές σύννεφου και κέντρα δεδομένων για επεξεργασία δημιουργεί προβλήματα στην επικοινωνία, δεν εγγυάται απόκριση σε πραγματικό χρόνο, ενώ συχνά αποφεύγεται λόγω ζητημάτων ασφάλειας και ιδιωτικότητας. Λαμβάνοντας υπόψη τον ολοένα και αυξανόμενο αριθμό IoT συνδέσεων, το αρχικό συννεφο-κεντρικό σύστη-μα πιέζεται για να καλύψει τις απαιτήσεις σε απόδοση, και ως αποτέλεσμα, έχει αρχίσει να εγκαταλείπεται. Για αυτόν τον λόγο, υπάρχει μια τάση να επεξεργάζονται τα δεδομένα στην άκρη του δικτύου, δηλαδή, μόλις αυτά δημιουργούνται [2].

Ταυτόχρονα, η μαζική ανάπτυξη απαιτητικών εφαρμογών και ισχυρών αλγορίθμων από τομείς όπως η Ψηφιακή Επεξεργασία Σήματος (DSP), η Υπολογιστική Όραση (CV), η Τεχνητή Νοημοσύνη (AI), και η Μηχανική Μάθηση (ML), σηματοδοτεί μια νέα εποχή για τα υπολογιστικά συστήματα στην άκρη του δικτύου. Συμβατικοί ενσωματωμένοι επεξεργαστές, όπως Κεντρικές Μονάδες Επεξεργασίας (CPUs) και μικροελεγκτές, δεν έχουν την υπολογιστική ισχύ για να χειριστούν τους φόρτους εργασίας αυτών των ε-φαρμογών και αλγορίθμων, και έτσι, δεν μπορούν να ανταποκριθούν στις απαιτήσεις απόδοσης [3–5]. Ένας άλλος περιορισμός είναι η μικρή χωρητικότητα μνήμης στην άκρη του δικτύου, η οποία επιβραδύνει την επεξεργασία, ειδικά σε εφαρμογές με μεγάλο όγκο δεδομένων εισόδου/εξόδου.

Είναι προφανές ότι ο πολλαπλασιασμός των δεδομένων μαζί με την εμφάνιση εφαρμο-γών αυξημένης πολυπλοκότητας απαιτούν εναλλακτικές σχεδιαστικές λύσεις ώστε να διατηρηθεί επαρκής απόδοση στην άκρη του δικτύου. Ως αποτέλεσμα, τα ενσωματω-μένα συστήματα έχουν εξελιχθεί με τα χρόνια, και αποτελούν πολύπλοκα συστήματα που περιλαμβάνουν CPUs, εξειδικευμένους επεξεργαστές/επιταχυντές, μπλοκ μνήμης, περιφερειακά υψηλού εύρους ζώνης, και διεπαφές επικοινωνίας. Όμως, η ανάγκη για χα-μηλή κατανάλωση ισχύος [6,7] δεν επιτρέπει την απρόσκοπτη αύξηση των υπολογιστικών πόρων. Συνεπώς, το σχεδιαστικό πρόβλημα ανάγεται ως εξής: από τη μια πλευρά, οι

εφαρμογές απαιτούν ταχύτητα και απόκριση σε πραγματικό χρόνο, ενώ από την άλλη πλευρά, υπάρχουν στενοί περιορισμοί ισχύος.

Τα ολοκληρωμένα κυκλώματα αποτελούν τον ακρογωνιαίο λίθο των υπολογιστικών συστημάτων, καθώς επηρεάζουν εγγενώς την απόδοσή τους, την κατανάλωση ισχύος και την επιφάνεια τους. Ιστορικά, η τεχνολογία ημιαγωγών βασίστηκε για περισσότερα από 40 χρόνια σε δύο θεμελιώδεις αρχές: τον Νόμο του Moore [8] και τον Νόμο του Dennard [9]. Όμως, το τέλος της κλιμάκωσης του Dennard σε συνδυασμό με άλλους παράγοντες, όπως η τεχνολογία ψύξης και τα φυσικά όρια του πυριτίου, μας οδήγησε στην εποχή του "Dark Silicon" [12–14]. Γίνεται λοιπόν αντιληπτό ότι η κατανάλωση ισχύος αποτελεί ένα κρίσιμο ζήτημα στη σχεδίαση κυκλωμάτων και συστημάτων. Σε αυτό το πλαίσιο, η βιομηχανία των υπολογιστικών συστημάτων ψάχνει νέες σχεδιαστικές προσεγγίσεις και υπολογιστικές πλατφόρμες, οι οποίες βελτιώνουν την κατανάλωση ισχύος παρέχοντας παράλληλα την επιθυμητή απόδοση.

Στοχεύοντας σε υπολογιστικά συστήματα χαμηλής κατανάλωσης και υψηλής απόδοσης, διάφορες προσεγγίσεις σχεδίασης υιοθετούνται, όπως ο ***Προσεγγιστικός Υπολογισμός*** [15–19], η ***Επιτάχυνση Υλικού*** [3, 4, 20–22], και ο ***Ετερογενής Υπολογισμός*** [23–27]. Ο Προσεγγιστικός Υπολογισμός εκμεταλλεύεται την ανθεκτικότητα σε λάθη των DSP/AI εφαρμογών και χειροτερεύει της ποιότητα των αποτελεσμάτων με αντάλλαγμα κέρδη σε ισχύ, επιφάνεια και/ή απόδοση. Η Επιτάχυνση Υλικού αναφέρεται στη διαδικασία εκτέλεσης διαδικασιών υψηλής πολυπλοκότητας σε εξειδικευμένο υλικό, όπως τα Ολοκληρωμένα Κυκλώματα Ειδικής Εφαρμογής (ASICs) και οι Συστοιχίες Επιτόπια Προγραμματιζόμενων Πυλών (FPGAs). Τέλος, ο Ετερογενής Υπολογισμός αναφέρεται σε συστήματα που ενσωματώνουν πολλαπλούς τύπους επεξεργαστή και διαφορετικές τεχνολογίες μνημών, όπως οι Μονάδες Επεξεργασίας Όρασης (VPUs). Η παρούσα Διατριβή στοχεύει στα τρία αυτά πρότυπα υπολογισμού, προτείνοντας σχεδιαστικές λύσεις και μεθοδολογίες για τη βελτίωση των επιταχυντών υλικού.

2. Συμβολή Διδακτορικής Διατριβής

Το ***αντικείμενο*** της Διατριβής είναι η σχεδίαση αριθμητικών κυκλωμάτων και DSP & AI επιταχυντών. Σε αυτό το πλαίσιο, προτείνουμε λύσεις και μεθοδολογίες με στόχο τη βελτιωμένη σχεδίαση σε ASICs/FPGAs και ενσωματωμένους πολυπύρηνους επεξεργαστές. Σε επίπεδο κυκλώματος, υιοθετούμε το πρότυπο σχεδίασης του Προσεγγιστικού Υπολογισμού, και προτείνουμε νέες τεχνικές αριθμητικής προσέγγισης, οι οποίες στη συνέχεια χρησιμοποιούνται σε προσεγγιστικούς επιταχυντές υλικού. Σε επίπεδο πλατφόρμας, στοχεύουμε να ξεκλειδώσουμε πλήρως τις δυνατότητες νέων ενσωματωμένων συσκευών, όπως τα FPGAs διαστημικού βαθμού και οι ετερογενείς VPUs.

Η κύρια **διαφοροποίηση** της Διατριβής σε σχέση με τη βιβλιογραφία συνοψίζεται ως εξής:

- Σε επίπεδο τεχνικής σχεδίασης, προτείνουμε νέες μεθόδους προσέγγισης, τις οποίες και συνδυάζουμε ώστε να δημιουργήσουμε έναν πολύ μεγάλο χώρο σχεδίασης με πολλαπλές διαμορφώσεις προσέγγισης, επιτρέποντας έτσι τη μεγιστοποίηση των κερδών σε πόρους κάτω από έναν καθορισμένο περιορισμό σφάλματος.

- Σε επίπεδο κυκλώματος, προτείνουμε προσεγγιστικά αριθμητικά κυκλώματα που μπορούν απρόσκοπτα να διαμορφώσουν την προσέγγιση τους κατά τον χρόνο εκτέλεσης.

- Σε επίπεδο επιταχυντή υλικού, πραγματοποιούμε μια εκτεταμένη εξερεύνηση του χώρου σχεδίασης πάνω στις τεχνικές προσέγγισης, τις αριθμητικές αναπαραστάσεις και τις τεχνικές σχεδίασης υλικού, ώστε να υλοποιήσουμε προσεγγιστικούς επιταχυντές σε ASIC/FPGA τεχνολογία.

- Σε επίπεδο υπολογιστικής πλατφόρμας, εξετάζουμε συστηματικά τις δυνατότητες των εργαλείων προγραμματισμού και εκμεταλλευόμαστε πλήρως τις αρχιτεκτονικές υλικού ώστε να επιταχύνουμε αποδοτικά DSP/AI αλγορίθμους υψηλής πολυπλοκότητας.

Στην συνέχεια, αναλύουμε περιληπτικά τις προτεινόμενες τεχνικές και μεθοδολογίες, και παρουσιάζουμε τα κύρια πειραματικά αποτελέσματα.

3. Βελτιστοποίηση Αριθμητικής: DLSB Κωδικοποίηση

*Η ενότητα αυτή αφορά το **Κεφάλαιο 3** και βασίζεται στη **δημοσίευση [226]**.*

Εισαγωγή

Η ενσωμάτωση καινοτόμων αριθμητικών αναπαραστάσεων στη σχεδίαση κυκλωμάτων αποτελεί μία αποτελεσματική λύση για τη μείωση της κατανάλωσης ενέργειας/ισχύος, της επιφάνειας και της καθυστέρησης ενός κυκλώματος [227]. Στο πλαίσιο αυτό, έχουν προταθεί στη βιβλιογραφία διάφορα εναλλακτικά αριθμητικά πρότυπα [228–230] που έχουν ως στόχο τη βελτίωση του υλικού των αριθμητικών μονάδων. Στην παρούσα Διατριβή, εστιάζουμε στην αριθμητική αναπαράσταση DLSB [231], η οποία προσθέτει στη συμβατική αριθμητική ένα επιπλέον Λιγότερο Σημαντικό Bit (LSB). Δηλαδή, όλοι οι αριθμοί αναπαρίστανται με ένα ακόμα bit που έχει το ίδιο βάρος με το συμβατικό LSB. Το πρότυπο DLSB καθιστά το εύρος αναπαράστασης αριθμών πλήρως συμμετρικό, δηλαδή για την αριθμητική του συμπληρώματος ως προς 2, το εύρος αναπαράστασης είναι

$[-2^{n-1},\ 2^{n-1}]$ αντί για $[-2^{n-1},\ 2^{n-1} - 1]$. Επίσης, παρέχει πολλά πλεονεκτήματα, όπως: (i) η απλούστερη διαδικασία αντιστροφής αριθμών, (ii) η απλούστερη διαδικασία στρογγυλοποίησης στους υπολογισμούς κινητής υποδιαστολής, και (iii) η αυξημένη αποτελεσματικότητα στους υπολογισμούς Αριθμητικής Υπολοίπου (RNS). Τα πλεονεκτήματα της DLSB αριθμητικής στη σχεδίαση κυκλωμάτων αναλύονται στις δημοσιεύσεις [226, 231].

Όμως, η χρήση της αριθμητικής DLSB έχει ως αποτέλεσμα να αυξάνονται η καθυστέρηση και η επιφάνεια των κυκλωμάτων σε σύγκριση με τα αντίστοιχα συμβατικά κυκλώματα. Σύμφωνα με τη θεωρητική ανάλυση της δημοσίευσης [231], τα κυκλώματα πολλαπλασιασμού παρουσιάζουν τις μεγαλύτερες αυξήσεις σε πόρους. Επομένως, με κίνητρο τα πλεονεκτήματα της αριθμητικής DLSB, και στοχεύοντας να μειώσουμε τις αυξήσεις σε πόρους που προκύπτουν, προτείνουμε έναν βελτιωμένο αλγόριθμο για την εκτέλεση του πολλαπλασιασμού δύο αριθμών DLSB συμπληρώματος ως προς 2. Παρέχουμε μία εκτενή ανάλυση σε επίπεδο δυαδικού ψηφίου και αξιολογούμε τον προτεινόμενο αλγόριθμο θεωρητικά και πειραματικά, αποδεικνύοντας ότι οι χειρισμοί και οι βελτιστοποιήσεις χαμηλού επιπέδου προσφέρουν σημαντικά κέρδη σε πόρους. Επιπλέον, παρακινούμε περαιτέρω την εφαρμογή της αριθμητικής DLSB και αξιολογούμε την αποτελεσματικότητα της προτεινόμενης τεχνικής σε ένα ρεαλιστικό σενάριο. Συγκεκριμένα, δείχνουμε ότι οι πολλαπλασιαστές μεγάλου μεγέθους μπορούν να υλοποιηθούν αποδοτικά χρησιμοποιώντας μικρούς πολλαπλασιαστές DLSB ως δομικά στοιχεία.

Σχεδίαση Κυκλώματος DLSB Πολλαπλασιαστή

Έστω ότι ο $X = \langle x_{n-1}x_{n-2}\cdots x_0 \rangle_{2's}$ είναι ένας n-bit αριθμός σε συμπλήρωμα ως προς 2. Ο X μετατρέπεται στον αριθμό DLSB X^+ επισυνάπτοντας ένα επιπλέον LSB, όπως φαίνεται στην Εξ. (1).

$$X^+ = X + x_{0+} = \langle x_{n-1}x_{n-2}\cdots x_0 \rangle_{2's} + x_{0+} \tag{1}$$

Με βάση τον αλγόριθμο πολλαπλασιασμού Modified Booth [227], σχεδιάζουμε τον πολλαπλασιαστή για DLSB αριθμούς. Ένας από τους δύο όρους του πολλαπλασιασμού, έστω ο B^+, κωδικοποιείται με το επιπλέον LSB b_{0+} να συμπεριλαμβάνεται στο λιγότερο σημαντικό Modified Booth ψηφίο, δηλαδή $b_{-1} = b_{0+}$ αντί για $b_{-1} = 0$ (συμβατική κωδικοποίηση). Σε αυτή την περίπτωση, ο πολλαπλασιασμός εκτελείται σύμφωνα με την Εξ. (2).

$$A^+ \times B^+ = \langle a_{n-1}a_{n-2}\cdots a_0 \rangle_{2's} \cdot B^+ + a_{0+} \cdot B^+ =$$

$$= \langle a_{n-1}a_{n-2}\cdots a_0 \rangle_{2's} \cdot \sum_{j=0}^{n/2-1} 4^j b_j^{MB} + a_{0+} \cdot \left(\langle b_{n-1}b_{n-2}\cdots b_0 \rangle_{2's} + b_{0+} \right) \tag{2}$$

Η υλοποίηση του κυκλώματος με βάση την παραπάνω εξίσωση απαιτεί έναν συμβατικό πολλαπλασιαστή Modified Booth για τον υπολογισμό του πρώτου όρου της παράστασης, καθώς και επιπρόσθετο υλικό για τον υπολογισμό του όρου $a_{0+} \cdot \left(\langle b_{n-1} b_{n-2} \cdots b_0 \rangle_{2\text{'s}} + b_{0+} \right)$. Για να μειωθεί το κόστος που προκύπτει από τον υπολογισμό του επιπλέον όρου, θεωρούμε μια εναλλακτική αναπαράσταση για τον A^+. Συγκεκριμένα, πραγματοποιούμε χαμηλού επιπέδου λογικές βελτιστοποιήσεις, και ο A^+ κωδικοποιείται όπως φαίνεται στην Εξ. (3).

$$A^+ = (-1)^{a_{0+}} \cdot A', \qquad \text{όπου} \ \ A' = \langle a'_{n-1} a'_{n-2} \cdots a'_0 \rangle_{2\text{'s}}, \ \ a'_i = a_i \oplus a_{0+} \qquad (3)$$

Το επόμενο βήμα είναι να ενσωματώσουμε τον όρο $(-1)^{a_{0+}}$ στον υπολογισμό του b_j^{MB}, δηλαδή του Modified Booth ψηφίου. Λαμβάνοντας υπόψη ότι ο όρος $(-1)^{a_{0+}}$ επηρεάζει το πρόσημο (ανάλογα με την τιμή του a_{0+}), οδηγούμε το σήμα για το πρόσημο του Modified Booth ψηφίου σε μια πύλη XOR μαζί με το a_{0+}. Ως αποτέλεσμα, η απόλυτη τιμή του νέου Modified Booth ψηφίου (b_j^{MB+}) είναι η ίδια με του b_j^{MB}, ωστόσο, το πρόσημο του εξαρτάται και από το a_{0+}.

Με βάση τις παραπάνω τροποποιήσεις, ο πολλαπλασιασμός δύο n-bit DLSB αριθμών απαιτεί τη δημιουργία και την άθροιση $n/2$ μερικών γινομένων PP_j, όπως φαίνεται στις Εξ. (4)–(6).

$$A^+ \times B^+ = A' \cdot \sum_{j=0}^{n/2-1} 4^j b_j^{MB+} = \sum_{j=0}^{n/2-1} 4^j PP_j \qquad (4)$$

$$PP_j = A' \cdot b_j^{MB+} = 2^n \, \overline{p}_{j,n} + \sum_{i=0}^{n-1} 2^i p_{j,i} \qquad (5)$$

$$b_j^{MB+} = (-1)^{s'_j} \cdot (2two_j + one_j) = (-1)^{s_j \oplus a_{0+}} \cdot (2two_j + one_j) \qquad (6)$$

Οι λογικές εξισώσεις των σημάτων κωδικοποίησης s_j, one_j και two_j δημιουργούνται από τον πίνακα αλήθειας της συμβατικής Modified Booth κωδικοποίησης [227]. Το προτεινόμενο κύκλωμα κωδικοποίησης απεικονίζεται στο Σχήμα 1α΄. Σε σύγκριση με τον συμβατικό κωδικοποιητή Modified Booth (Σχήμα 1β΄), ο προτεινόμενος κωδικοποιητής έχει μία επιπλέον πύλη XOR για τον υπολογισμό του σήματος προσήμου s'_j, το οποίο περιλαμβάνει το a_{0+}. Η δημιουργία του i bit του μερικού γινομένου PP_j, η οποία προκύπτει από την Εξ. (5), απεικονίζεται σε επίπεδο πύλης στο Σχήμα 1γ΄. Το κύκλωμα παραγωγής μερικών γινομένων του DLSB πολλαπλασιαστή είναι το ίδιο με αυτό που χρησιμοποιείται στον συμβατικό αλγόριθμο Modified Booth.

Για λόγους σύγκρισης, το Σχήμα 2 απεικονίζει τα δέντρα των μερικών γινομένων των 16-bit πολλαπλασιαστών που εξετάστηκαν, δηλαδή του κλασσικού πολλαπλασιαστή Modified Booth (Σχήμα 2α΄), του μη-βελτιστοποιημένου πολλαπλασιαστή DLSB (Σχήμα
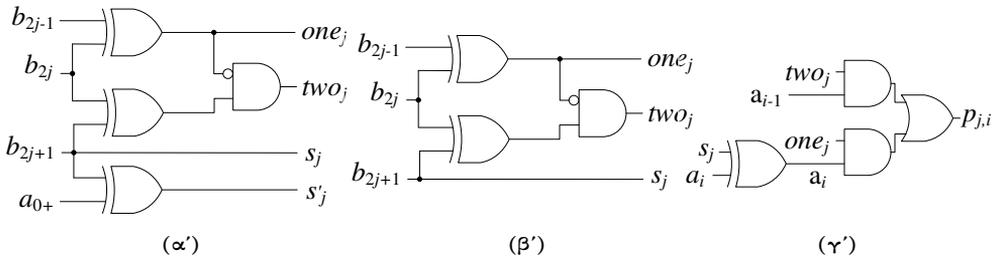
Σχήμα 1: Τα κυκλώματα για τον πολλαπλασιασμό DLSB αριθμών: **(α′)** προτεινόμενος DLSB Modified Booth κωδικοποιητής, **(β′)** κλασικός Modified Booth κωδικοποιητής, **(γ′)** κλασσικός γεννήτορας μερικών γινομένων (a_i: i bit του A, $\mathrm{a}_i = s_j \oplus a_i$).
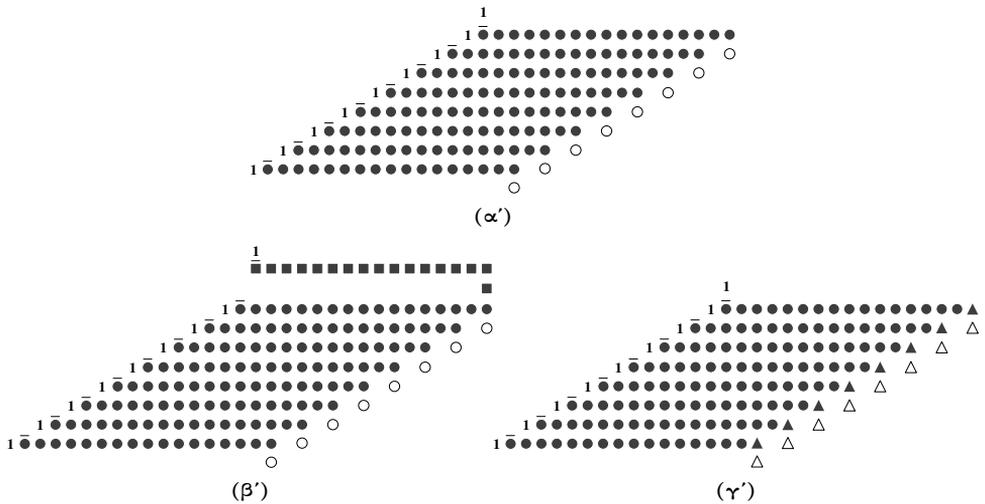
Σχήμα 2: Τα δέντρα μερικών γινομένων 16-bit πολλαπλασιαστών: **(α′)** συμβατικός Modified Booth πολλαπλασιαστής, **(β′)** μη-βελτιστοποιημένος DLSB Modified Booth πολλαπλασιαστής, **(γ′)** προτεινόμενος DLSB Modified Booth πολλαπλασιαστής.
Σύμβολα: ●: $p_{j,i}$ ○: όρος διόρθωσης ■: $a_{0+} \cdot b_i$ ▲: $p'_{j,0}$ △: όρος διόρθωσης

2β′), και του προτεινόμενου πολλαπλασιαστή DLSB (Σχήμα 2γ′). Στα σχήματα έχουν συμπεριληφθεί όλοι οι όροι που απαιτούνται για την ορθή εκτέλεση του πολλαπλασιασμού (bits μερικών γινομένων, σταθεροί όροι και όροι διόρθωσης). Όπως φαίνεται, ο μη-βελτιστοποιημένος πολλαπλασιαστής DLSB, ο οποίος εκφράζεται από την Εξ. (2), υλοποιεί τον κλασικό πολλαπλασιαστή Modified Booth συν επιπλέον λογική για το γινόμενο $a_{0+} \cdot B^+$. Αντίθετα, το προτεινόμενο κύκλωμα, το οποίο βασίζεται στην Εξ. (4), υλοποιεί ένα δέντρο ίδιου βάθους με αυτό του κλασσικού πολλαπλασιαστή Modi-

fied Booth. Η μόνη διαφορά του σε σχέση με αυτόν είναι ο υπολογισμός του σήματος $s'_j = s_j \oplus a_{0+}$. Αυτό το σήμα χρησιμοποιείται αντί για το s_j στην παραγωγή του LSB των μερικών γινομένων και των όρων διόρθωσης. Δηλαδή, σε σύγκριση με τον πολλαπλασιαστή Modified Booth για κλασσικούς αριθμούς, η μόνο επιβάρυνση του προτεινόμενου κυκλώματος για τον πολλαπλασιασμό DLSB αριθμών είναι $n/2$ πύλες XOR.

Πειραματική Αξιολόγηση

Για την πειραματική αξιολόγηση, αναπτύσσουμε τα κυκλώματα στη γλώσσα περιγραφής υλικού Verilog και πραγματοποιούμε τη σύνθεση τους με το εργαλείο Design Compiler της Synopsys και τη βιβλιοθήκη των 45nm της TSMC. Θεωρούμε τους ακόλουθους συμβολισμούς για τα εξεταζόμενα κυκλώματα πολλαπλασιαστών:

- CMB: συμβατικός Modified Booth πολλαπλασιαστής για κανονικούς αριθμούς συμπληρώματος ως προς 2.

- DLSB1: μη-βελτιστοποιημένος Modified Booth πολλαπλασιαστής για DLSB αριθμούς συμπληρώματος ως προς 2.

- DLSB2: προτεινόμενος Modified Booth πολλαπλασιαστής για DLSB αριθμούς συμπληρώματος ως προς 2.

Ο Πίνακας 1 παρουσιάζει τα πειραματικά αποτελέσματα από τη σύνθεση των πολλαπλασιαστών για διάφορα μεγέθη ($n = 8, 16, 32$). Επιπλέον, αναφέρονται οι επιβαρύνσεις σε ενέργεια και επιφάνεια των πολλαπλασιαστών σε σύγκριση με τον CMB. Όσον αφορά τα κρίσιμα μονοπάτια των κυκλωμάτων, όπως αναμενόταν, οι DLSB πολλαπλασιαστές έχουν σχεδόν τις ίδιες καθυστερήσεις με τον CMB. Ο DLSB2 εμφανίζει παρόμοια καθυστέρηση ακόμη και για το μικρότερο μήκος bit, δηλαδή 0.38ns έναντι 0.37ns. Αυτό δικαιολογείται από το γεγονός ότι το βάθος του δέντρου των μερικών γινομένων του δεν έχει αυξηθεί. Σε σύγκριση με τον DLSB1, η βελτιστοποιημένη σχεδίαση του DLSB2 επιτυγχάνει καλύτερα αποτελέσματα για όλους τους πόρους. Κατά μέσο όρο, ο DLSB2 επιβάλλει μικρές επιβαρύνσεις επιφάνειας και ενέργειας (3.1% και 3.3%), ενώ οι αντίστοιχες επιβαρύνσεις του μη-βελτιστοποιημένου DLSB1 είναι αρκετά μεγαλύτερες (11.9% και 15.4%). Όσον αφορά την κλιμάκωση του μήκους bit, τα αποτελέσματα δείχνουν ότι όσο αυξάνεται το μέγεθος του πολλαπλασιαστή, ο αντίκτυπος του επιπλέον LSB γίνεται μικρότερος τόσο στην επιφάνεια όσο και στην ενέργεια.

Στη συνέχεια, δείχνουμε πώς ο n-bit DLSB2 μπορεί να χρησιμοποιηθεί αποτελεσματικά ως δομικό στοιχείο για την υλοποίηση πολλαπλασιασμών με μεγάλους k-bit αριθμούς. Χωρίς απώλεια της γενικότητας, θεωρούμε $k = 2n$. Πρακτικά, έχουμε κυκλώματα πολλαπλασιασμού με μικρό σταθερό μήκος bit και θέλουμε να εκτελέσουμε πολλαπλασιασμούς μεγαλύτερων αριθμών.

Πίνακας 1: Αποτελέσματα σύνθεσης των n-bit DLSB πολλαπλασιαστών στη βιβλιοθήκη των 45nm της TSMC.

Κύκλωμα		Καθυστέρηση (ns)	Επιφάνεια (μm^2)	(%)[1]	Ενέργεια ($\mu W \cdot ns$)	(%)[1]
$n = 8$	CMB	0.37	485	–	384	–
	DLSB1	0.40	572	+18	480	+25
	DLSB2	0.38	510	+5.2	406	+5.7
$n = 16$	CMB	0.51	1519	–	1186	–
	DLSB1	0.52	1701	+12	1344	+13.3
	DLSB2	0.51	1562	+2.8	1217	+2.6
$n = 32$	CMB	0.67	5189	–	4340	–
	DLSB1	0.67	5491	+5.8	4685	+7.9
	DLSB2	0.67	5256	+1.3	4412	+1.7

[1] Αναφέρεται σε % επιβάρυνση επιφάνειας/ενέργειας (σχετική αύξηση) σε σύγκριση με τον CMB.

Έστω ότι οι A και B είναι δύο k-bit αριθμοί συμπληρώματος ως προς 2. Χωρίζουμε τους A και B σε δύο DLSB λέξεις των n bits. Πιο συγκεκριμένα, όπως φαίνεται στις Εξ. (7)–(8), χωρίζουμε τους αριθμούς σε δύο λέξεις, επισυνάπτοντας τα πιο σημαντικά bits των λιγότερο σημαντικών λέξεων, δηλαδή τα a_{n-1} και b_{n-1}, ως επιπλέον LSBs στις πιο σημαντικές λέξεις. Θεωρώντας ένα επιπλέον LSB ίσο με 0 για τις λιγότερο σημαντικές λέξεις, οι k-bit αριθμοί μετασχηματίζονται σε συναρτήσεις δύο n-bit DLSB αριθμών συμπληρώματος ως προς 2.

$$A = \left(\langle a_{2n-1} a_{2n-2} \cdots a_n \rangle_{2's} + a_{n-1} \right) \cdot 2^n + \left(\langle a_{n-1} a_{n-2} \cdots a_0 \rangle_{2's} + 0 \right) =$$
$$= A_1^+ \cdot 2^n + A_2^+ \tag{7}$$

$$B = \left(\langle b_{2n-1} b_{2n-2} \cdots b_n \rangle_{2's} + b_{n-1} \right) \cdot 2^n + \left(\langle b_{n-1} b_{n-2} \cdots b_0 \rangle_{2's} + 0 \right) =$$
$$= B_1^+ \cdot 2^n + B_2^+ \tag{8}$$

Ο πολλαπλασιασμός των παραπάνω κωδικοποιημένων αριθμών A και B απαιτεί την εκτέλεση 4 πολλαπλασιασμών μικρότερου μεγέθους, όπως προκύπτει από την επιμεριστική ιδιότητα. Αυτοί οι πολλαπλασιασμοί μπορούν να εκτελεστούν από το n-bit DLSB2 κύκλωμα. Αντίθετα, με βάση την ανάλυσή μας [226], αν χωρίσουμε τους αριθμούς θεωρώντας την συμβατική αριθμητική, χρειάζεται να χρησιμοποιηθεί ο κλασικός $(n+1)$-bit CMB ώστε να εκτελεστούν σωστά όλοι οι μικρότεροι πολλαπλασιασμοί που προκύπτουν. Για τον λόγο αυτό, στο Σχήμα 3 παρουσιάζουμε τα κέρδη σε πόρους του n-bit DLSB2 έναντι του $(n+1)$-bit CMB. Τα αποτελέσματα δείχνουν ότι ο DLSB2 παρέχει σημαντικά κέρδη, ειδικά για μικρό μήκος bit: 30.6% και 43.3% σε επιφάνεια και ενέργεια,
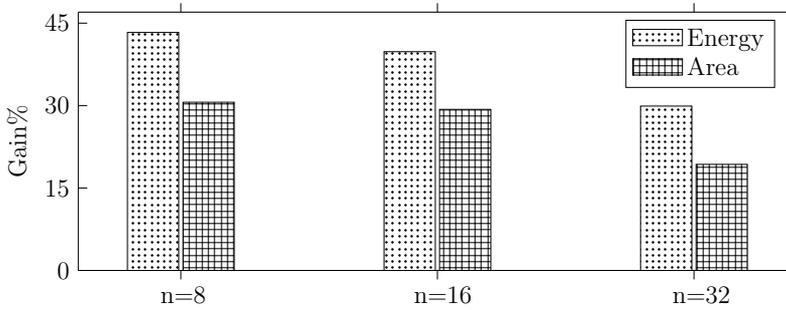
Σχήμα 3: Τα κέρδη σε ενέργεια και επιφάνεια κυκλώματος που παρέχονται από τον *n*-bit DLSB2 πολλαπλασιαστή σε σύγκριση με τον $(n+1)$-bit CMB (στοχεύοντας στη χρήση του DLSB2 αντί του CMB ως δομικό στοιχείο σε μεγάλους πολλαπλασιαστές).

αντίστοιχα, για $n = 8$. Η αναποτελεσματικότητα του CMB δικαιολογείται από το επιπλέον bit στο μήκος του πολλαπλασιαστή, το οποίο έχει ως αποτέλεσμα τη δημιουργία ενός ακόμη μερικού γινομένου και την προσθήκη ενός επιπλέον bit σε κάθε μερικό γινόμενο.

4. Αριθμητική Προσέγγιση: Υβριδική High-Radix Κωδικοποίηση

*Η ενότητα αυτή αφορά το **Κεφάλαιο 4** και βασίζεται στη **δημοσίευση [153]**.*

Εισαγωγή

Ο κύριος στόχος του Προσεγγιστικού Υπολογισμού στο επίπεδο της λογικής είναι τα αριθμητικά κυκλώματα [216], τα οποία αποτελούν βασικές υπολογιστικές μονάδες των επεξεργαστών γενικού σκοπού και των επιταχυντών υλικού. Στη βιβλιογραφία υπάρχει μία εκτενής έρευνα για τους προσεγγιστικούς αθροιστές [130, 246–249], οι οποίοι παρέχουν σημαντικά κέρδη στην καθυστέρηση του κρίσιμου μονοπατιού και την κατανάλωση ισχύος. Από την άλλη πλευρά, οι ερευνητικές δραστηριότητες που αφορούν προσεγγιστικούς πολλαπλασιαστές [142,143,149,151,154,250–256] έχουν αφήσει αρκετά ζητήματα ανοικτά σε σύγκριση με τις αντίστοιχες για τους αθροιστές. Στο κύκλωμα του πολλαπλασιασμού μπορούν να εφαρμοστούν προσεγγίσεις στην παραγωγή των μερικών γινομένων [149, 151, 255] και στην άθροιση των μερικών γινομένων [154, 250, 251, 253]. Και οι δύο τύποι προσέγγισης μπορούν να εφαρμοστούν σε συνεργασία για την επίτευξη υψηλότερης μείωσης της ισχύος και της επιφάνειας του κυκλώματος [149, 151]. Αν και σημαντική έρευνα αφορά την άθροιση των μερικών γινομένων, οι τεχνικές προσέγγισης

για την παραγωγή των μερικών γινομένων έχουν λάβει λιγότερη προσοχή. Ένας άλλος περιορισμός των προσεγγιστικών πολλαπλασιαστών της βιβλιογραφίας είναι ότι η πλειονότητά τους (π.χ., [154,250,252,256]) δεν εξετάζει τον πολλαπλασιασμό προσημασμένων αριθμών.

Με κίνητρο τα κέρδη που παρέχονται από τα αριθμητικά κυκλώματα στους πόρους ενός επιταχυντή, προτείνουμε μία υβριδική κωδικοποίηση για έναν από τους δύο όρους του πολλαπλασιαστή. Αυτή η κωδικοποίηση έχει ως αποτέλεσμα την αποδοτική δημιουργία των μερικών γινομένων στο κύκλωμα πολλαπλασιασμού. Η προτεινόμενη τεχνική αντιμετωπίζει ορισμένα από τα ζητήματα των προηγούμενων σχετικών κυκλωμάτων, όπως είναι ο προσημασμένος πολλαπλασιασμός και η μείωση των κρίσιμων μονοπατιών, ενώ παρέχει επίσης καλύτερα αποτελέσματα από σημαντικές δουλειές της βιβλιογραφίας. Για την εφαρμογή της, χωρίζουμε τον αριθμό σε δύο λέξεις από bits και τον κωδικοποιούμε με βάση δύο διαφορετικές κωδικοποιήσεις ακτίνας (radix): τα Περισσότερο Σημαντικά Bits (MSBs) κωδικοποιούνται με την ακριβή κωδικοποίηση radix-4, ενώ τα k Λιγότερο Σημαντικά Bits (LSBs) κωδικοποιούνται με την προσεγγιστική κωδικοποίηση high-radix-2^k (όπου $k \geq 4$). Για να απλοποιηθεί η αυξημένη πολυπλοκότητα που προκαλείται από τις συμβατικές κωδικοποιήσεις high-radix, αλλάζουμε τους πίνακες αλήθειας των σημάτων κωδικοποίησης εισάγοντας προσεγγίσεις. Η χρήση αυτής της υβριδικής κωδικοποίησης έχει ως αποτέλεσμα τη μείωση του αριθμού των μερικών γινομένων και τη δημιουργία απλούστερων δέντρων για την άθροιση τους.

Σχεδίαση Κυκλώματος Υβριδικού High-Radix Πολλαπλασιαστή

Έστω ότι ο B είναι ένας n-bit αριθμός συμπληρώματος ως προς 2. Θεωρούμε έναν ζυγό αριθμό k που ανήκει στο διάστημα $[4, n-2]$, δηλαδή $k = 2m$: $m \in \mathbb{Z}$ και $2 \leq m \leq (n-2)/2$. Ο B χωρίζεται σε δύο τμήματα με βάση το k: το πιο σημαντικό τμήμα που περιέχει τα $n - k + 1$ MSBs του και το λιγότερο σημαντικό τμήμα που περιέχει τα k LSBs του (υπάρχει ένα κοινό bit μεταξύ των δύο τμημάτων). Τα $n - k + 1$ MSBs κωδικοποιούνται με την κωδικοποίηση radix-4, ενώ τα k LSBs κωδικοποιούνται με την κωδικοποίηση radix-2^k, όπως φαίνεται στις Εξ. (9)–(11).

$$B = \langle b_{n-1}b_{n-2}\cdots b_0 \rangle_{2\text{'s}} = -2^{n-1}b_{n-1} + \sum_{i=0}^{n-2} 2^i b_i = \sum_{j=k/2}^{n/2-1} 4^j y_j^{R4} + y_0^{R2^k} \qquad (9)$$

$$y_j^{R4} = -2b_{2j+1} + b_{2j} + b_{2j-1} \quad \implies \quad y_j^{R4} \in \{0, \pm 1, \pm 2\} \qquad (10)$$

$$y_0^{R2^k} = -2^{k-1}b_{k-1} + 2^{k-2}b_{k-2} + \cdots + b_0 \implies y_0^{R2^k} \in \{0, \pm 1, \ldots, \pm(2^{k-1}-1), -2^{k-1}\} \ (11)$$

Η Εξ. (10) αναφέρεται στα κωδικοποιημένα radix-4 ψηφία, ενώ η Εξ. (11) αναφέρεται στο κωδικοποιημένο radix-2^k ψηφίο. Πιο συγκεκριμένα, η κωδικοποίηση radix-4 δημιουργεί $(n-k)/2$ ψηφία $y_j^{R4} \in \{0, \pm 1, \pm 2\}$, ενώ η κωδικοποίηση radix-2^k δημιουργεί το ψηφίο $y_0^{R2^k} \in \{0, \pm 1, \pm 2, \pm 3, \ldots, \pm(2^{k-1}-1), -2^{k-1}\}$. Συνολικά, ο B κωδικοποιείται με $(n-k)/2 + 1$ ψηφία.

Η παραπάνω ακριβής υβριδική κωδικοποίηση χαρακτηρίζεται από αυξημένη πολυπλοκότητα λόγω των πολλών τιμών που παίρνει το ψηφίο $y_0^{R2^k}$, καθώς και των τιμών που δεν είναι δυνάμεις του δύο. Δηλαδή, το κύκλωμα κωδικοποίησης όλων των πιθανών τιμών του $y_0^{R2^k}$ είναι αρκετά μεγάλο. Επομένως, προτείνουμε μια εναλλακτική προσεγγιστική παραλλαγή της κωδικοποίησης radix-2^k, ενώ διατηρούμε την χαμηλής πολυπλοκότητας κωδικοποίηση radix-4 των MSBs ώστε να αποφευχθούν τα μεγάλα λάθη. Η προσέγγιση της κωδικοποίησης radix-2^k γίνεται μέσω της αντιστοίχισης όλων των πιθανών τιμών του $y_0^{R2^k}$ στην πλησιέστερη από τις 4 μεγαλύτερες δυνάμεις του δύο ή το 0. Με αυτόν τον τρόπο, το προσεγγιστικό ψηφίο $\hat{y}_0^{R2^k}$ παίρνει τιμές από ένα μικρότερο σύνολο τιμών, το οποίο περιλαμβάνει μόνο 4 απόλυτες τιμές συν το 0. Επίσης, το άθροισμα των τιμών αυτών είναι 0, όπως στην περίπτωση του σετ τιμών του y_j^{R4}. Επιλέγουμε να διατηρήσουμε μόνο τις 4 μεγαλύτερες δυνάμεις του δύο, έτσι ώστε ο προσεγγιστικός κωδικοποιητής radix-2^k να απαιτεί περίπου την διπλάσια επιφάνεια από τους $(n-k)/2$ ακριβούς κωδικοποιητές radix-4.

Ο Πίνακας 2 παρουσιάζει την κλασική ακριβή κωδικοποίηση radix-4. Το σήμα $sign_j$ αναφέρεται στο πρόσημο του y_j^{R4}, δηλαδή ενεργοποιείται όταν το ψηφίο έχει αρνητική τιμή. Τα σήματα $\times 1_j$ και $\times 2_j$ υποδεικνύουν στο μέτρο του y_j^{R4}, δηλαδή ενεργοποιούνται όταν το ψηφίο έχει μέτρο 1 και 2, αντίστοιχα. Ο Πίνακας 3 παρουσιάζει την προσεγγιστική κωδικοποίηση radix-2^k. Οι δύο πρώτες στήλες δείχνουν την προτεινόμενη αντιστοίχιση για να δημιουργηθεί το προσεγγιστικό σετ τιμών ($y_0^{R2^k} \rightarrow \hat{y}_0^{R2^k}$). Για παράδειγμα, εάν η τιμή του ακριβούς radix-2^k ψηφίου, όπως αυτή υπολογίζεται από την Εξ. (11), ανήκει στο διάστημα $[2^{k-5}, 2^{k-4}+2^{k-5})$, τότε αντιστοιχίζεται στην τιμή 2^{k-4}. Παρόμοια με την κωδικοποίηση radix-4, ενεργοποιούνται τα σήματα κωδικοποίησης για να υποδείξουν την τιμή του $\hat{y}_0^{R2^k}$. Οι λογικές εξισώσεις των σημάτων κωδικοποίησης radix-4 και radix-2^k αναφέρονται στην δημοσίευση [153] (εξάγονται από τους πίνακες με χρήση χαρτών Karnaugh).

Συνοπτικά, ξεκινώντας από την ακριβή υβριδική κωδικοποίηση των Εξ. (9)–(11), προσεγγίζουμε την κωδικοποίηση high-radix των LSBs μέσω της αντιστοίχισης του Πίνακα 3. Με αυτόν τον τρόπο, ο B κωδικοποιείται προσεγγιστικά ως $\tilde{B}$, όπως φαίνεται στις Εξ. (12)–(14).

$$\tilde{B} = \sum_{j=k/2}^{n/2-1} 4^j y_j^{R4} + \hat{y}_0^{R2^k} \tag{12}$$

Πίνακας 2: Η ακριβής radix-4 κωδικοποίηση.

Bits Αριθμού			R4 Ψηφίο	Σήματα Κωδικοποίησης		
b_{2j+1}	b_{2j}	b_{2j-1}	y_j^{R4}	$sign_j$	$\times 2_j$	$\times 1_j$
0	0	0	0	0	0	0
0	0	1	1	0	0	1
0	1	0	1	0	0	1
0	1	1	2	0	1	0
1	0	0	-2	1	1	0
1	0	1	-1	1	0	1
1	1	0	-1	1	0	1
1	1	1	0	1	0	0

Πίνακας 3: Η προσεγγιστική high-radix-2^k κωδικοποίηση.

Ακριβές Ψηφίο	Προσεγγ. Ψηφίο	Σήματα Κωδικοποίησης				
$y_0^{R2^k}$	$\hat{y}_0^{R2^k}$	$sign$	$\times 2^{k-1}$	$\times 2^{k-2}$	$\times 2^{k-3}$	$\times 2^{k-4}$
$[0,\ 2^{k-5})$	0	0	0	0	0	0
$[2^{k-5},\ 2^{k-4}+2^{k-5})$	2^{k-4}	0	0	0	0	1
$[2^{k-4}+2^{k-5},\ 2^{k-3}+2^{k-4})$	2^{k-3}	0	0	0	1	0
$[2^{k-3}+2^{k-4},\ 2^{k-2}+2^{k-3})$	2^{k-2}	0	0	1	0	0
$[2^{k-2}+2^{k-3},\ 2^{k-1})$	2^{k-1}	0	1	0	0	0
$[-2^{k-1},\ -2^{k-2}-2^{k-3})$	-2^{k-1}	1	1	0	0	0
$[-2^{k-2}-2^{k-3},\ -2^{k-3}-2^{k-4})$	-2^{k-2}	1	0	1	0	0
$[-2^{k-3}-2^{k-4},\ -2^{k-4}-2^{k-5})$	-2^{k-3}	1	0	0	1	0
$[-2^{k-4}-2^{k-5},\ -2^{k-5})$	-2^{k-4}	1	0	0	0	1
$[-2^{k-5},\ 0)$	0	1	0	0	0	0

$$\text{όπου }\ y_j^{R4} \in \{0,\ \pm 1,\ \pm 2\} \tag{13}$$

$$\text{και }\ \hat{y}_0^{R2^k} \in \{0,\ \pm 2^{k-4},\ \pm 2^{k-3},\ \pm 2^{k-2},\ \pm 2^{k-1}\} \tag{14}$$

Με βάση την παραπάνω κωδικοποίηση, σχεδιάζεται ο προσεγγιστικός πολλαπλασιαστής RAD2^k, του οποίου τα βασικά στάδια απεικονίζονται στο Σχήμα 4. Στο πρώτο στάδιο, ο B κωδικοποιείται με την προτεινόμενη υβριδική κωδικοποίηση. Το δεύτερο στάδιο λαμβάνει ως είσοδο τον A και τα σήματα κωδικοποίησης του $\tilde{B}$, και παράγει $(n-k)/2$ ακριβή μερικά γινόμενα από την κωδικοποίηση radix-4 (ψηφία y_j^{R4}) και 1 προσεγγιστι- κό μερικό γινόμενο από την κωδικοποίηση radix-2^k (ψηφίο $\hat{y}_0^{R2^k}$). Το προσεγγιστι-

Σχήμα 4: Τα βασικά στάδια του προσεγγιστικού πολλαπλασιαστή RAD2^k, ο οποίος βασίζεται στην υβριδική κωδικοποίηση high-radix.
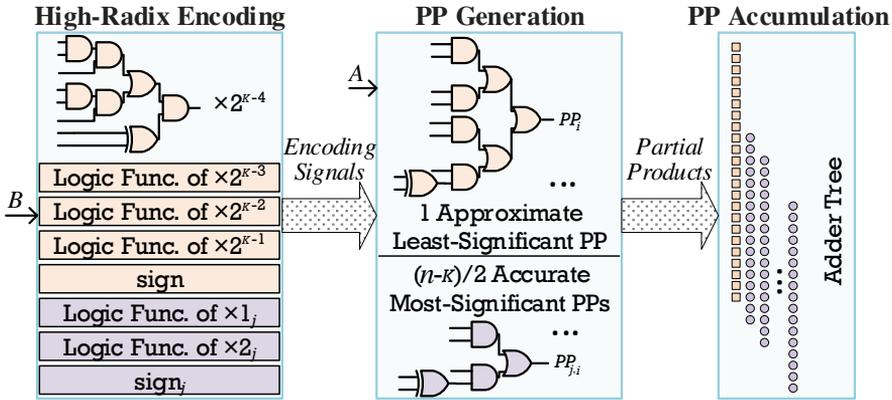
κό μερικό γινόμενο πρακτικά αντικαθιστά τα $k/2$ λιγότερο σημαντικά μερικά γινόμενα του ακριβούς πολλαπλασιαστή που χρησιμοποιεί μόνο την κωδικοποίηση radix-4. Δηλαδή, μειώνουμε τα μερικά γινόμενα κατά $k/2 - 1$, δεδομένου ότι ο ακριβής πολλαπλασιαστής radix-4 παράγει $n/2$ μερικά γινόμενα συνολικά, ενώ ο RAD2^k παράγει $(n - k)/2 + 1$.

Στο Σχήμα 5 παρουσιάζουμε τα κυκλώματα που παράγουν το κάθε bit του προσεγγιστικού μερικού γινομένου των RAD2^k πολλαπλασιαστών ($k = 6, 8, 10$). Όπως φαίνεται, η επιφάνεια του κυκλώματος παραγωγής μερικών γινομένων είναι ίδια για όλες τις κωδικοποιήσεις, δηλαδή είναι ανεξάρτητη της επιλεγμένης τιμής του k. Η διαφοροποίηση έγκειται στα bits του A και στο μέγεθος της πληροφορίας που μεταφέρουν τα σήματα κωδικοποίησης του B. Το αντίστοιχο κύκλωμα παραγωγής των ακριβών μερικών γινομένων που βασίζονται στην radix-4 κωδικοποίηση είναι αυτό που απεικονίζεται στο Σχήμα 1γ΄.

Το Σχήμα 6 απεικονίζει τα δέντρα των μερικών γινομένων των 16-bit RAD2^k πολλαπλασιαστών ($k = 6, 8, 10$). Εκτός από τα bits των μερικών γινομένων, τα δέντρα περιλαμβάνουν σταθερούς και διορθωτικούς όρους. Παρατηρούμε ότι με κάθε αύξηση της παραμέτρου k παράγεται ένα λιγότερο ακριβές radix-4 μερικό γινόμενο, το οποίο πρακτικά κωδικοποιείται σε ένα μεγαλύτερο προσεγγιστικό radix-2^k μερικό γινόμενο. Το μήκος bit κάθε ακριβούς μερικού γινομένου είναι $n+1$. Αντίστοιχα, το προσεγγιστικό μερικό γινόμενο, το οποίο είναι και το λιγότερο σημαντικό στο δέντρο, αποτελείται από $n + k - 1$ bits.
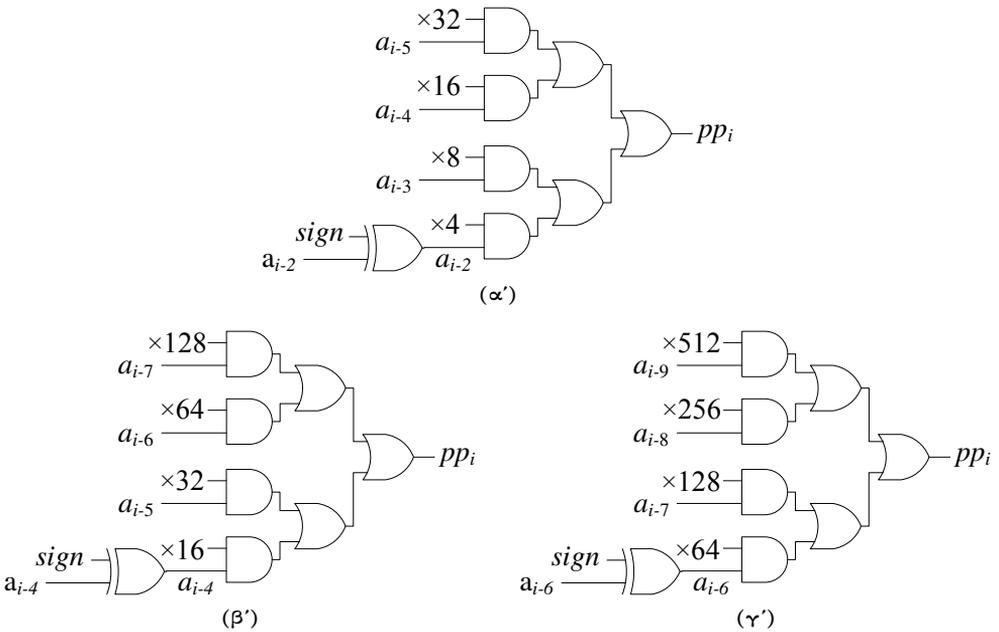
Σχήμα 5: Οι γεννήτορες ενός bit των μερικών γινομένων για τους πολλαπλασιαστές που βασίζονται στις προσεγγιστικές κωδικοποιήσεις high-radix: **(α′)** radix-64 ($k = 6$), **(β′)** radix-256 ($k = 8$), **(γ′)** radix-1024 ($k = 10$).

Συμβολισμοί: a_i: i-bit του A, $a_i = s_j \oplus a_i$
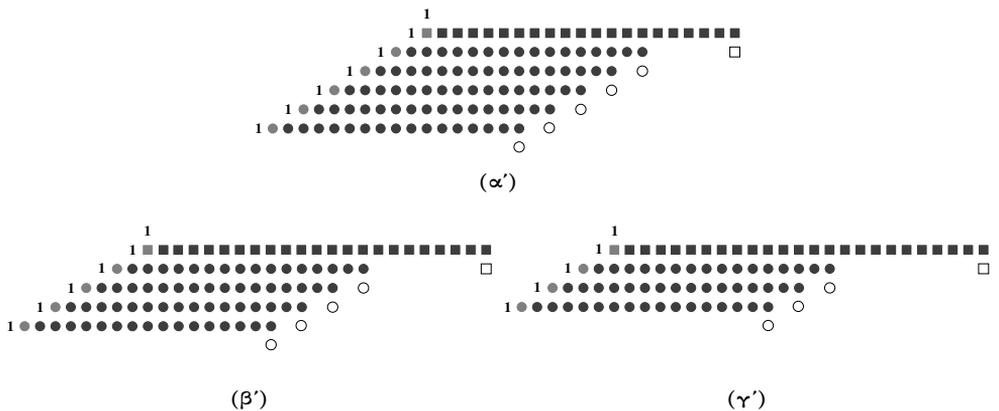
Σχήμα 6: Δέντρα μερικών γινομένων των 16-bit RAD2^k πολλαπλασιαστών, οι οποίοι βασίζονται στην ακριβή κωδικοποίηση radix-4 και την προσεγγιστική κωδικοποίηση high-radix-2^k: **(α′)** radix-64 ($k = 6$), **(β′)** radix-256 ($k = 8$), **(γ′)** radix-1024 ($k = 10$).

Σύμβολα: ●: bits radix-4 μερικών γινομένων ■: bits radix-2^k μερικών γινομένων ●, ■: αντεστραμμένα MSBs μερικών γινομένων ○, □: διορθωτικοί όροι

Πειραματική Αξιολόγηση

Για την πειραματική αξιολόγηση, υλοποιούμε τους προτεινόμενους RAD2^k πολλαπλασιαστές με τη γλώσσα περιγραφής υλικού Verilog και τους συνθέτουμε με το εργαλείο Design Compiler της Synopsys και τη βιβλιοθήκη των 65nm της TSMC. Για λόγους σύγκρισης, υλοποιούμε διάφορα κυκλώματα πολλαπλασιαστή: τον ακριβή radix-4 (ACCR4), τον ακριβή radix-8 (ACCR8), καθώς και τους προσεγγιστικούς R8ABM1 & R8ABM2-15 [151], R4ABM1-14, R4ABM1-16, R4ABM2-14, & R4ABM2-16 [149], και DRUM6 [142]. Εκτός από την αξιολόγηση των πόρων υλικού (επιφάνεια, καθυστέρηση, ισχύς, ενέργεια), οι οποίοι παρέχονται από τα εργαλεία υλοποίησης, μελετάμε και τα λάθη των προσεγγιστικών κυκλωμάτων. Συγκεκριμένα, χρησιμοποιούμε τη Μέση Σχετική Απόσταση Λάθους (MRED) και την Πιθανότητα Σχετικής Απόστασης Λάθους πάνω από 2% (PRED$_2$). Στη δημοσίευση [153] υπάρχει εκτενέστερη ανάλυση του λάθους των κυκλωμάτων. Ο Πίνακας 4 παρουσιάζει τα κύρια πειραματικά αποτελέσματα για τους πολλαπλασιαστές αριθμών των 16 bits.

Αρχικά, παρατηρούμε την αναποτελεσματικότητα του ACCR8 συγκριτικά με τον ACCR4, γεγονός που αποδεικνύει ότι οι υψηλότερες συμβατικές ακριβείς κωδικοποιήσεις radix αυξάνουν τα κόστη σε πόρους. Ο RAD64 δίνει το καλύτερο MRED, καθώς οι προσεγγίσεις του εκτελούνται μόνο στα 6 LSBs του B. Επιπλέον, οι προσεγγιστικές τιμές του έχουν μικρή απόλυτη απόσταση από τις ακριβείς. Αντίθετα, ο DRUM6 προσφέρει το χειρότερο MRED μεταξύ όλων των πολλαπλασιαστών με 1.47%, αν και η κατανομή σφαλμάτων του είναι περιορισμένη. Γενικά, οι προτεινόμενοι πολλαπλασιαστές RAD2^k παρέχουν MRED μικρότερο από 1%, πράγμα που σημαίνει ότι μπορούν να χρησιμοποιηθούν σε πραγματικές εφαρμογές που ανέχονται μέσα σφάλματα 5% ή 10% [258].

Ο DRUM6 έχει τη χειρότερη καθυστέρηση κρίσιμου μονοπατιού, καθώς αρχικά υπολογίζει την απόλυτη τιμή των γινομένων [142], ενώ οι RAD2^k αποτελούν τα ταχύτερα κυκλώματα (με καθυστερήσεις 0.65ns–0.72ns). Όσον αφορά την ενέργεια, ο R8ABM1 προσφέρει τη χειρότερη κατανάλωση μεταξύ όλων των πολλαπλασιαστών, καθώς μπορεί να χρησιμοποιεί έναν προσεγγιστικό αθροιστή για τον υπολογισμό των γινομένων 3A, αλλά περιέχει έναν ακριβή αθροιστή των 7 bits για την αποφυγή πολύ μεγάλων σφαλμάτων [151]. Η ενέργεια βελτιώνεται στον R8ABM2-15 λόγω της περικοπής των bits των μερικών γινομένων. Ωστόσο, καθώς οι RAD2^k έχουν μικρό MRED, η τεχνική της περικοπής μπορεί επίσης να εφαρμοστεί και σε αυτούς, παρέχοντας ακόμα καλύτερα κέρδη σε πόρους. Οι πολλαπλασιαστές του [149] παρουσιάζουν παρόμοια αποτελέσματα, με τον R4ABM2-16 να προσφέρει τα καλύτερα αποτελέσματα. Ο RAD1024 διαθέτει την καλύτερη κατανάλωση ενέργειας, με τον DRUM6 να είναι δεύτερος, παρέχοντας, ωστόσο, σημαντικά μεγαλύτερο MRED από τους άλλους πολλαπλασιαστές. Το ίδιο συμβαίνει και για την μετρική PRED$_2$, η οποία είναι αυξημένη στον DRUM6, ενώ στους

Πίνακας 4: Πειραματικά αποτελέσματα των 16-bit πολλαπλασιαστών στη βιβλιοθήκη των 65nm της TSMC.

Κύκλωμα	Καθυστέρηση (ns)	Ισχύς (μW)	Επιφάνεια (μm^2)	Ενέργεια (μW·ns)	MRED (%)	PRED$_2$ (%)
ACCR4	0.75	4998	4153	3749	–	–
ACCR8	0.80	5343	4639	4274	–	–
RAD64	0.72	4497	3489	3238	0.08	0.42
RAD256	0.69	3493	2769	2410	0.28	1.69
RAD1024	0.65	3422	2624	2224	0.93	6.74
R8ABM1 [151]	0.77	5058	4210	3895	0.15	1.05
R8ABM2-15 [151]	0.74	3377	2926	2499	0.61	2.70
R4ABM1-14 [149]	0.74	4676	3958	3460	0.12	0.41
R4ABM1-16 [149]	0.73	4447	3725	3246	0.49	1.49
R4ABM2-14 [149]	0.73	4648	3732	3393	0.24	0.68
R4ABM2-16 [149]	0.72	4307	3467	3101	1.18	2.50
DRUM6 [142]	1.07	2148	3993	2298	1.47	28.85

άλλους πολλαπλασιαστές είναι μικρότερη από 3% (εκτός από τον RAD1024 που έχει PRED$_2$ 6.74%).

Το Σχήμα 7 απεικονίζει το διάγραμμα λάθους–ενέργειας για τους εξεταζόμενους πολλαπλασιαστές. Ο σκοπός του είναι να αναδείξει τα πιο εξέχοντα κυκλώματα, δηλαδή την καλύτερη τεχνική προσέγγισης για πολλαπλασιαστές. Όπως φαίνεται, το σύνορο Pareto σχηματίζεται αποκλειστικά από τους πολλαπλασιαστές RAD2^k. Οι πολλαπλασιαστές αυτοί αποτελούν τις πιο αποτελεσματικές λύσεις, καθώς παρουσιάζουν την καλύτερη ισορροπία μεταξύ ενέργειας και λάθους. Δηλαδή, παρέχουν καλύτερη κατανάλωση ενέργειας για το ίδιο ή μικρότερο λάθος από τους άλλους πολλαπλασιαστές της βιβλιογραφίας. Ο RAD64 επιτυγχάνει τη μικρότερη τιμή για το MRED και θα πρέπει να προτιμάται όταν η ακρίβεια των αποτελεσμάτων είναι σημαντική, ενώ ο RAD1024 προσφέρει τη μεγαλύτερη μείωση της ενέργειας για τυπικές τιμές λάθους ($< 1\%$). Τέλος, ο RAD256 επιτυγχάνει σημαντική μείωση της ενέργειας για πολύ μικρό λάθος.

Είναι επίσης σημαντικό να αναφέρουμε τα κέρδη των RAD2^k πολλαπλασιαστών σε σύγκριση με το ακριβές κύκλωμα αναφοράς, δηλαδή τον συμβατικό radix-4 πολλαπλασιαστή. Ως κέρδος ορίζουμε την σχετική μείωση των πόρων που επιτυγχάνουν οι RAD2^k από το κύκλωμα αναφοράς. Συγκεκριμένα, τα κέρδη τους είναι ως και 56% σε ενέργεια και 55% σε επιφάνεια με αντάλλαγμα ένα MRED μικρότερο του 1%. Τέλος, εξετάζουμε τους RAD2^k πολλαπλασιαστές για μεγαλύτερα μήκη bit ($n = 24, 32$). Σε αυτή την περίπτωση, τα κέρδη φτάνουν ως και 64% σε ενέργεια για ένα σταθερό MRED ίσο με 0.28%. Με βάση την ανάλυσή μας [153], όσο το μήκος bit αυξάνεται, τόσο μεγαλύτερα κέρδη προσφέρονται από τους RAD2^k για το ίδιο λάθος.
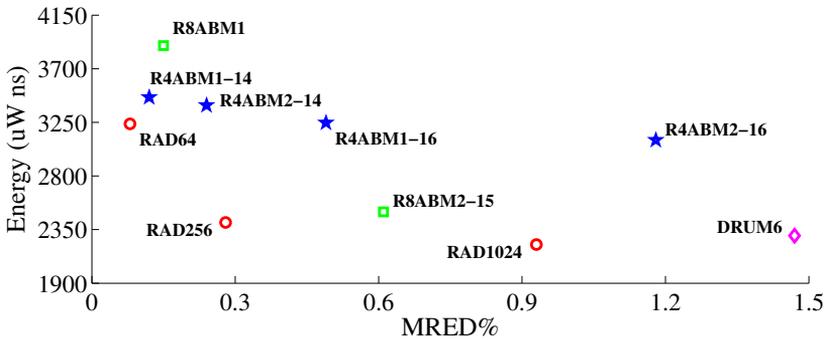
Σχήμα 7: Η συγκριτική ανάλυση Pareto των προσεγγιστικών πολλαπλασιαστών ως προς το μέσο σχετικό σφάλμα και την κατανάλωση ενέργειας.

5. Δυναμική Προσέγγιση: Κυκλώματα Επαναδιαμορφώσιμα στον Χρόνο Εκτέλεσης

Η ενότητα αυτή αφορά το **Κεφάλαιο 5** *και βασίζεται στις* **δημοσιεύσεις [144, 145]**.

Εισαγωγή

Η αυξημένη ποικιλομορφία των ανθεκτικών-σε-λάθη εφαρμογών απαιτεί ευέλικτους προσεγγιστικούς σχεδιασμούς που προσφέρουν πολλαπλές διαμορφώσεις προσέγγισης. Πιο συγκεκριμένα, τα προσεγγιστικά συστήματα και κυκλώματα πρέπει να έχουν την δυνατότητα να προσαρμόζουν τον βαθμό της προσέγγισης τους, έτσι ώστε να ικανοποιείται συνεχώς η επιθυμητή ακρίβεια της εφαρμογής, παρέχοντας παράλληλα επαρκή απόδοση εντός του περιορισμένου εύρους ισχύος. Ακόμη και για την ίδια εφαρμογή, ενδέχεται να χρησιμοποιούνται διαφορετικές κατανομές εισόδου που να απαιτούν μια νέα διαμόρφωση της προσέγγισης προκειμένου να παρέχεται αποδεκτή ποιότητα αποτελεσμάτων (π.χ., σκεφτείτε έναν προσεγγιστικό επεξεργαστή βίντεο που δέχεται εικόνες με διαφορετικό περιεχόμενο). Ως αποτέλεσμα, υπάρχει μια αυξανόμενη ανάγκη για προσεγγιστικούς σχεδιασμούς που να μπορούν: (i) να παρέχουν πολλαπλές διαμορφώσεις προσέγγισης (δηλαδή διαφορετικά επίπεδα ακρίβειας) και (ii) να διαμορφώνουν δυναμικά τον βαθμό της προσέγγισης τους, είτε πριν ξεκινήσει η εκτέλεση μιας εφαρμογής είτε κατά τον χρόνο εκτέλεσης, ανάλογα με τους περιορισμούς ακρίβειας και ισχύος.

Με κίνητρο την ανάγκη για δυναμική διαμόρφωση της προσέγγισης, στοχεύουμε στα α-ριθμητικά κυκλώματα και σχεδιάζουμε προσεγγιστικούς πολλαπλασιαστές που μπορούν να ρυθμίσουν τον βαθμό προσέγγισης τους κατά τον χρόνο εκτέλεσης. Τα προτεινόμενα κυκλώματα παρέχουν μεγαλύτερο χώρο προσέγγισης και απρόσκοπτη δυναμική διαμόρ-φωση σε σύγκριση με τους $\mathrm{RAD2}^k$ πολλαπλασιαστές, των οποίων ο βαθμός προσέγγισης είναι σταθερός και επιλέγεται κατά τον χρόνο σχεδίασης/υλοποίησης. Σημειώνουμε ότι δεν εξετάζουμε ποια διαμόρφωση τους βαθμού προσέγγισης θα πρέπει να επιλεγεί ούτε εφαρμόζουμε αυτόματη διαμόρφωση του (π.χ., με βάση το λάθος). Στόχος μας είναι να υλοποιήσουμε κυκλώματα που μπορούν να αλλάξουν αποτελεσματικά και εύκολα βαθμό προσέγγισης κατά τον χρόνο εκτέλεσης (χωρίς δηλαδή σημαντικές επιβαρύνσεις στην επιφάνεια κυκλώματος ή στην ταχύτητα). Επιπροσθέτως, εκτός από την αριθμητική ακε-ραίων και σταθερής υποδιαστολής, υλοποιούμε τα κυκλώματα μας και για πραγματικούς αριθμούς κινητής υποδιαστολής.

Για να αυξήσουμε τον χώρο προσέγγισης, εφαρμόζουμε δύο ανεξάρτητες τεχνικές προ-σέγγισης στα κυκλώματα μας: την παράλειψη μερικών γινομένων (partial product per-foration) και την στρογγυλοποίηση μερικών γινομένων (partial product rounding). Η πρώτη τεχνική απαλείφει ολόκληρα μερικά γινόμενα (ξεκινώντας από τα λιγότερο ση-μαντικά), ενώ η δεύτερη στρογγυλοποιεί τα εναπομείναντα σε μικρότερο μήκος bit. Εκτός από τη δημιουργία ενός μεγάλου χώρου προσέγγισης, επιλέγουμε αυτές τις δύο τεχνικές επειδή η διαμόρφωσή τους σχετίζεται με τους αριθμούς εισόδου του πολλα-πλασιαστή, και έτσι μπορούμε εύκολα να την αλλάξουμε κατά τον χρόνο εκτέλεσης. Συγκεκριμένα, προσθέτουμε μόνο $2n$ πύλες AND και δύο n-bit σήματα ελέγχου (ένα ανά αριθμό εισόδου), όπου n είναι το μήκος bit των αριθμών. Έτσι, η διαμόρφωση της προσέγγισης ανάγεται σε ρύθμιση των bits των σημάτων ελέγχου, τα οποία τα οδηγούμε είτε στο λογικό '1' είτε στο '0' για να εφαρμόσουμε τον επιθυμητό βαθμό προσέγγισης.

Σχεδίαση Κυκλωμάτων Πολλαπλασιαστή με Δυναμική Προσέγγιση

Έστω ότι οι $A = \langle a_{n-1}a_{n-2}\cdots a_0 \rangle_{2's}$ και $B = \langle b_{n-1}b_{n-2}\cdots b_0 \rangle_{2's}$ είναι δύο n-bit αριθ-μοί συμπληρώματος ως προς 2. Ο ακριβής πολλαπλασιασμός $A \times B$ που βασίζεται στην radix-4 κωδικοποίηση απαιτεί την παραγωγή και την άθροιση $n/2$ μερικών γινομένων. Η εφαρμογή της perforation παραλείπει τη δημιουργία των P λιγότερο σημαντικών μερι-κών γινομένων, ενώ η εφαρμογή της rounding στρογγυλοποιεί τα εναπομείναντα μερικά γινόμενα περικόπτοντας τα $R-1$ LSBs τους και προσθέτοντας το $(R-1)$-ο bit στην υπόλοιπη (πιο σημαντική) λέξη από bits. Η εφαρμογή των δύο τεχνικών προσέγγισης είναι ανεξάρτητη και παρουσιάζεται στο Σχήμα 8. Ονομάζουμε αυτή την οικογένεια προσεγγιστικών πολλαπλασιαστών $\mathrm{AxFXU}|_{P,R}$ για να δηλώσουμε τον προσεγγιστικό πολλαπλασιασμό αριθμών σταθερής υποδιαστολής με στατικό βαθμό προσέγγισης P
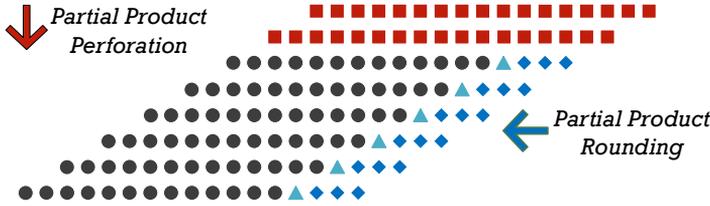
Σχήμα 8: Το προσεγγιστικό δέντρο μερικών γινομένων των τεχνικών perforation ($P = 2$) και rounding ($R = 4$).

Σύμβολα: ■: περικομμένα bits ◆: περικομμένα bits ▲: bits που προστίθενται στην υπόλοιπη λέξη για στρογγυλοποίηση ●: ακριβή bits

και R για perforation και rounding, αντίστοιχα. Ο πολλαπλασιασμός του κυκλώματος AxFXU εκτελείται όπως φαίνεται στις Εξ. (15)–(18).

$$A \times B|_{P,R} = \sum_{j=P}^{n/2-1} 4^j \tilde{P}P_j = \sum_{j=P}^{n/2-1} 4^j A_R \cdot y_j^{R4} \tag{15}$$

$$\text{όπου} \quad P \in [0, n/2 - 1), \quad R \in [0, n-1) \tag{16}$$

$$y_j^{R4} = -2b_{2j+1} + b_{2j} + b_{2j-1} \implies y_j^{R4} \in \{0, \pm 1, \pm 2\} \tag{17}$$

$$A_R = \langle a_{n-1} a_{n-2} \cdots a_R \rangle_{2\text{'s}} + a_{R-1} \tag{18}$$

Επομένως, ο πολλαπλασιασμός του AxFXU υπολογίζεται αθροίζοντας τα $n/2 - P$ πιο σημαντικά στρογγυλεμένα μερικά γινόμενα $\tilde{P}P_j$. Όπως φαίνεται και στο Σχήμα 8, το δέντρο των μερικών γινομένων μειώνεται κατακόρυφα μέσω της perforation και οριζόντια μέσω της rounding.

Η τεχνική perforation αφαιρεί P radix-4 κωδικοποιητές και $P \times n$ 1-bit γεννήτορες μερικών γινομένων από τον ακριβή πολλαπλασιαστή. Αυτά τα κυκλώματα απαιτούνται για την παραγωγή των μερικών γινομένων που απαλείφονται (κόκκινα τετράγωνα στο Σχήμα 8). Για τη βελτίωση της rounding και την αποφυγή επιβαρύνσεων λόγω της πρόσθεσης των a_{R-1} (μπλε ρόμβοι στο Σχήμα 8) με τις πιο σημαντικές λέξεις, χρησιμοποιούμε τις βελτιστοποιήσεις που προτείνουμε για την αριθμητική DLSB, θέτοντας $a_{0+} = a_{R-1}$. Με αυτόν τον τρόπο, τα bits των μερικών γινομένων παράγονται όπως στον ακριβή πολλαπλασιαστή, και μόνο μια πύλη XOR προστίθεται στον υπολογισμό του κάθε όρου διόρθωσης.

Στη συνέχεια, παρουσιάζουμε το κύκλωμα του αντίστοιχου προσεγγιστικού πολλαπλασιαστή για αριθμητική κινητής υποδιαστολής, το οποίο ονομάζεται AxFPU|$_{P,R}$. Το

κύκλωμα αυτό βασίζεται στον συμβατικό πολλαπλασιαστή αριθμών κινητής υποδιαστο-
λής, με τη διαφορά ότι χρησιμοποιεί τον AxFXU για να πολλαπλασιάσει τα σημαντικά
ψηφία των αριθμών (γνωστά με την ονομασία mantissa). Επίσης, ενσωματώνει επι-
πλέον λειτουργίες για να χειριστεί κάποιες ειδικές περιπτώσεις αποτελεσμάτων που
προκύπτουν λόγω των προσεγγίσεων. Θεωρώντας κανονικούς αριθμούς κινητής υποδια-
στολής με βάση το πρότυπο της IEEE-754 [263] (S: πρόσημο, E: πολωμένος εκθέτης,
e_{max}: πόλωση εκθέτη), το μπλοκ διάγραμμα του AxFPU$|_{P,R}$ απεικονίζεται στο Σχήμα
9 και η συνάρτηση του παρατίθεται στην Εξ. (19).

$$A_f \times B_f|_{P,R} = (-1)^{(S_A+S_B)} \cdot 2^{(E_A+E_B-2e_{max})} \cdot \underbrace{A_R \cdot \sum_{j=P}^{m/2-1} 4^j y_j^{R4}}_{A \times B|_{P,R}} \qquad (19)$$

Σε σύγκριση με τον ακριβή πολλαπλασιαστή, υπάρχει μία επιπλέον μονάδα που ανιχνεύει
αποτελέσματα που δεν είναι στη σωστή μορφή κινητής υποδιαστολής. Συγκεκριμένα,
ο προσεγγιστικός πολλαπλασιασμός των mantissas πολύ μεγάλων αριθμών μπορεί να
δημιουργήσει ένα αποτέλεσμα της μορφής $00.xx \ldots x$, ενώ τα αποτελέσματα πρέπει να
είναι πάντα στη μορφή $01.xx \ldots x$, $10.xx \ldots x$, ή $11.xx \ldots x$ [263]. Αυτό μπορεί να
συμβεί λόγω της απαλοιφής αρνητικών μερικών γινομένων από την perforation. Αυτή
η περίπτωση αντιμετωπίζεται ως υπερχείλιση, ωστόσο, σημειώνουμε ότι σύμφωνα με τα
πειραματικά μας αποτελέσματα, η πιθανότητα να συμβεί είτε κανονική υπερχείλιση είτε
αυτή η ειδική υπερχείλιση είναι περίπου 0.35% κατά μέσο όρο για κινητή υποδιαστολή
μισής και μονής ακρίβειας.

Τα κυκλώματα AxFXU$|_{P,R}$ και AxFPU$|_{P,R}$ εφαρμόζουν στατική προσέγγιση, δηλαδή
οι παράμετροι P και R δεν μπορούν να αλλάξουν μετά την υλοποίηση των κυκλωμάτων.
Για να σχεδιάσουμε τις δυναμικές εκδοχές τους (DyFXU$|_{P,R}$ και DyFPU$|_{P,R}$) χρη-
σιμοποιούμε ολόκληρο τον συμβατικό radix-4 πολλαπλασιαστή και βασιζόμαστε στις
ακόλουθες δύο παρατηρήσεις:

(i) τα $2P-1$ LSBs του B χρησιμοποιούνται μόνο για την παραγωγή των P λιγότερο
σημαντικών μερικών γινομένων.

(ii) τα $R-1$ LSBs του A χρησιμοποιούνται μόνο για την παραγωγή των $R-1$ LSBs
του κάθε μερικού γινομένου.

Από την πρώτη διαπίστωση, συμπεραίνουμε ότι η περικοπή των $2P-1$ LSBs του B
ισοδυναμεί με την εφαρμογή perforation βαθμού P. Από τη δεύτερη διαπίστωση, συ-
μπεραίνουμε ότι η περικοπή των $R-1$ LSBs του A μαζί με την πρόσθεση του a_{R-1}
με το πιο σημαντικό μέρος του A ισοδυναμούν με την εφαρμογή rounding βαθμού R.
Επομένως, για να ελέγχουμε δυναμικά τις τιμές των παραμέτρων P και R, οδηγούμε το
i bit του κάθε αριθμού εισόδου σε μια πύλη AND μαζί με ένα σήμα s_i ($i = 1, 2, \ldots, n$).
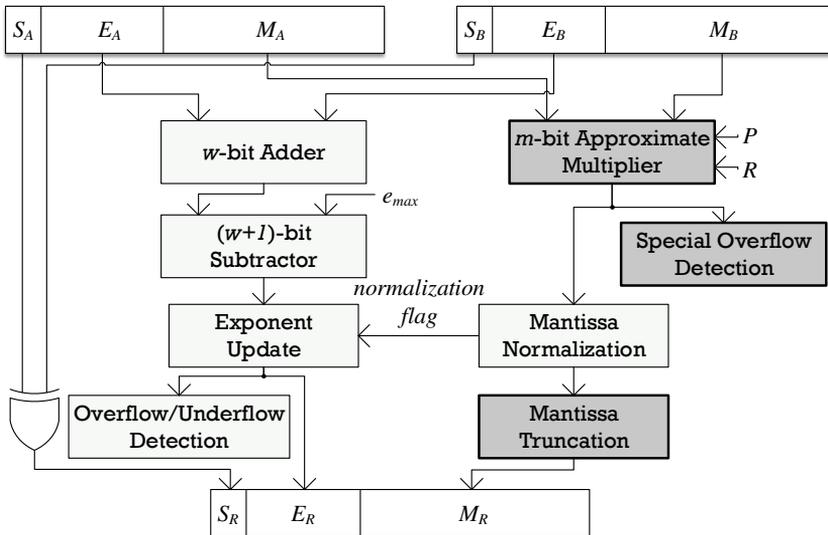
Σχήμα 9: Ο προσεγγιστικός πολλαπλασιαστής αριθμών κινητής υποδιαστολής AxFPU$|_{P,R}$, ο οποίος βασίζεται στις τεχνικές perforation (P) και rounding (R). Οι τονισμένες μονάδες υποδεικνύουν τις διαφοροποιήσεις σε σύγκριση με τον συμβατικό ακριβή πολλαπλασιαστή.

Η τιμή του σήματος s_i προσδιορίζει αν το i bit του αριθμού θα περικοπεί "εικονικά" ($s_i = 0$) ή όχι ($s_i = 1$).

Το Σχήμα 10 απεικονίζει τη δυναμική διαμόρφωση της προσέγγισης σε $P = 2$ και $R = 4$. Το σήμα s_i παίρνει την τιμή '0' στις πύλες AND που οδηγούνται από:

(i) τα 3 LSBs του B, και έτσι, τα 2 λιγότερο σημαντικά μερικά γινόμενα δεν υπολογίζονται (είναι 0) → η perforation ρυθμίζεται σε $P = 2$.

(ii) τα 3 LSBs του A και έτσι, τα 3 LSB κάθε μερικού γινομένου δεν υπολογίζονται (είναι 0) → η rounding ρυθμίζεται σε $R = 4$.

Στις υπόλοιπες πύλες AND, το s_i οδηγείται στο λογικό '1' ώστε τα αντίστοιχα bits των μερικών γινομένων να παραχθούν ορθά.

Πειραματική Αξιολόγηση

Για την πειραματική αξιολόγηση, υλοποιούμε διάφορες διαμορφώσεις των προτεινόμενων πολλαπλασιαστών για αριθμητική σταθερής (AxFXU/DyFXU) και κινητής υποδιαστολής (AxFPU/DyFPU) με τη γλώσσα περιγραφής υλικού Verilog. Η σύνθεση γίνεται με το εργαλείο Design Compiler της Synopsys και τη βιβλιοθήκη των 65nm της TSMC. Εκτός από την αξιολόγηση των πόρων υλικού, στις δημοσιεύσεις [144, 145] παρέχουμε
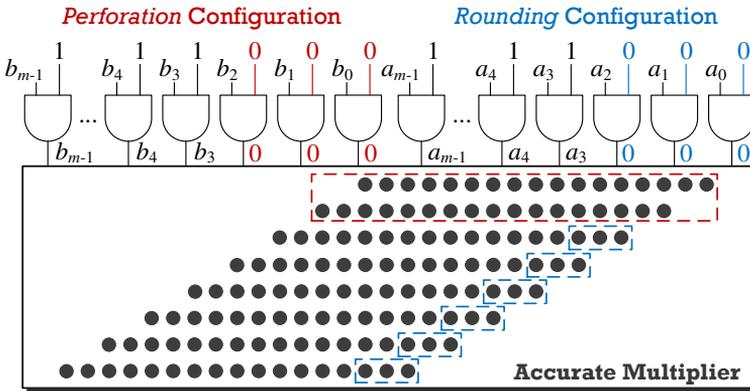
Σχήμα 10: Η δυναμική διαμόρφωση του βαθμού προσέγγισης των πολλαπλασιαστών. Η "εικονική" περικοπή των 3 LSBs του A και του B θέτει τα σημειωμένα bits των μερικών γινομένων στο 0, ενεργοποιώντας $P = 2$ (διαμόρφωση της perforation) και $R = 4$ (διαμόρφωση της rounding).

μία εκτεταμένη ανάλυση των λαθών, ειδικά για τα κυκλώματα κινητής υποδιαστολής, στα οποία εμφανίζονται ιδιόμορφα αποτελέσματα (υπερχείλιση, υποχείλιση και τύποι δεδομένων όπως NaN και $\pm\infty$).

Στο Σχήμα 11 παρουσιάζεται η ανάλυση Pareto για τους πολλαπλασιαστές σταθερής υποδιαστολής. Το διάγραμμα περιλαμβάνει αρκετούς AxFXU πολλαπλασιαστές και τους αντίστοιχους DyFXU, καθώς και σημαντικά κυκλώματα της βιβλιογραφίας [142, 143, 149, 151, 153, 255, 266]. Υπενθυμίζουμε ότι ο $\text{AxFXU}|_{P,R}$ έχει σχεδιαστεί να εφαρμόζει πάντα την προσέγγιση $\{P, R\}$, δηλαδή δεν έχουν υλοποιηθεί κάποιες μονάδες του συμβατικού ακριβούς κυκλώματος. Αντίθετά, ο $\text{DyFXU}|_{P,R}$ διαμορφώνεται δυναμικά για να εφαρμόσει την προσέγγιση $\{P, R\}$, δηλαδή έχει υλοποιηθεί όλο το συμβατικό ακριβές κύκλωμα και επιλέγεται η προσέγγιση μέσω σημάτων εισόδου. Όπως φαίνεται στο διάγραμμα, το σύνορο Pareto σχηματίζεται αποκλειστικά από διαφορετικές διαμορφώσεις του AxFXU. Δηλαδή, το προτεινόμενο κύκλωμα αποτελεί την καλύτερη σχεδιαστική λύση, λαμβάνοντας υπόψη τόσο το μέσο λάθος όσο και την κατανάλωση ενέργειας. Ο $\text{AxFXU}|_{2,2}$ παρέχει σημαντική μείωση της ενέργειας με αντάλλαγμα ένα πολύ μικρό λάθος. Ομοίως, οι $\text{AxFXU}|_{2,6}$, $\text{AxFXU}|_{3,4}$ και $\text{AxFXU}|_{3,6}$ προσφέρουν αξιοσημείωτα ενεργειακά κέρδη για τιμές MRED έως 0.78%. Όσον αφορά τον DyFXU, παρόλο που έχει χειρότερη κατανάλωση ενέργειας από τον AxFXU, διατηρεί σε όλες σχεδόν τις περιπτώσεις το σύνορο Pareto σε σχέση με τους υπόλοιπους προσεγγιστικούς πολλαπλασιαστές.

Στη συνέχεια αξιολογούμε τα προτεινόμενα δυναμικά κυκλώματα. Όπως αναμενόταν, ο DyFXU παρέχει την ίδια καθυστέρηση κρίσιμου μονοπατιού και ελαφρώς αυξημένη επιφάνεια σε σύγκριση με τον συμβατικό ακριβή πολλαπλασιαστή. Όσον αφορά την κα-

Σχήμα 11: Η συγκριτική ανάλυση Pareto των προσεγγιστικών πολλαπλασιαστών σταθερής υποδιαστολής ως προς το μέσο σχετικό σφάλμα και την κατανάλωση ενέργειας.
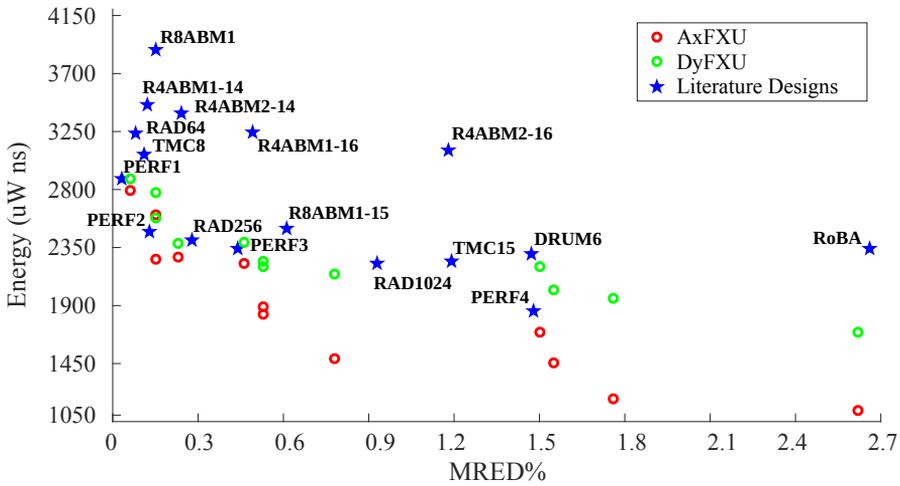
τανάλωση ενέργειας, η οποία έχει μετρηθεί όταν ο DyFXU λειτουργεί για προσέγγιση P και R, είναι βελτιωμένη σε σχέση με τον ακριβή πολλαπλασιαστή λόγω των λογικών μονοπατιών που δεν ενεργοποιούνται. Τέλος, είναι ενδιαφέρουσα η σύγκριση του DyFXU$|_{P,R}$ με τον αντίστοιχο AxFXU$|_{P,R}$. Η κατανάλωση ενέργειας του DyFXU$|_{P,R}$ είναι αυξημένη λόγω της ισχύος διαρροής των ανενεργών κόμβων του κυκλώματος και της ισχύος που καταναλώνεται από τις πύλες AND και τα σήματα ελέγχου που δίνονται ως είσοδοι στο κύκλωμα. Ενδεικτικά, ο DyFXU$|_{1,2}$ καταναλώνει από $1.03\times$ έως και $1.2\times$ περισσότερη ενέργεια από AxFXU$|_{1,2}$ ανάλογα με την επιλεγμένη συχνότητα ρολογιού. Αντίστοιχα, για μεγαλύτερες προσεγγίσεις, ο DyFXU$|_{4,6}$ καταναλώνει $1.3\times-$ $1.7\times$ περισσότερη ενέργεια από τον AxFXU$|_{4,6}$. Συνοψίζοντας, ο DyFXU δεν παρέχει κέρδη σε επιφάνεια, ωστόσο, οι επιπλέον επιβαρύνσεις είναι αμελητέες. Ως προς την ενέργεια, παρέχει σημαντικά κέρδη και ξεπερνά πολλούς πολλαπλασιαστές της βιβλιογραφίας, παρόλο που αυτά τα κέρδη είναι μειωμένα σε σύγκριση με τον AxFXU. Σε κάθε περίπτωση, η πολύ μικρή επιβάρυνση επιφάνειας και τα μειωμένα (αλλά εξίσου σημαντικά) ενεργειακά κέρδη επιτυγχάνονται ενώ υποστηρίζεται η δυναμική διαμόρφωση της προσέγγισης.

Όσον αφορά τα κυκλώματα κινητής υποδιαστολής, θεωρούμε μισή και μονή ακρίβεια, δηλαδή 16-bit και 32-bit αριθμούς, αντίστοιχα. Αρχικά, πραγματοποιούμε μία λεπτομερή εξερεύνηση του χώρου σχεδίασης ώστε να εξάγουμε τις καλύτερες διαμορφώσεις του AxFPU μέσω ανάλυσης Pareto [145]. Η εξερεύνηση μας δείχνει ότι υπάρχει ένα ευρύ φάσμα κερδών σε επιφάνεια και ενέργεια με αντάλλαγμα τυπικές τιμές MRED. Αυτό

καθιστά τον AxFPU ως μία αποτελεσματική λύση για σενάρια με διαφορετικούς περιορισμούς σε επιφάνεια/ενέργεια. Σύμφωνα με τα αποτελέσματα, η τεχνική rounding έχει αξιοσημείωτο αντίκτυπο στην ακρίβεια των αποτελεσμάτων για μικρές διαμορφώσεις της perforation. Αντίθετα, για μεγαλύτερες προσεγγίσεις perforation, το MRED δεν επηρεάζεται τόσο πολύ από τον βαθμό προσέγγισης της rounding. Ένα άλλο συμπέρασμα που προκύπτει από αυτή την ανάλυση είναι ότι οι προτεινόμενες προσεγγίσεις ευνοούνται από μεγαλύτερα μήκη bit, όπως στην περίπτωση της μονής ακρίβειας αριθμητικής κινητής υποδιαστολής. Επίσης, αξίζει να αναφέρουμε ότι τα ενεργειακά κέρδη σε σύγκριση με τους συμβατικούς πολλαπλασιαστές κινητής υποδιαστολής κυμαίνονται έως 53.5% για μισή και 82.4% για μονή ακρίβεια. Αυτά τα κέρδη προσφέρονται για τιμές MRED έως 2.2% και 3.3%, αντίστοιχα.

Στη συνέχεια, συγκρίνουμε τον AxFPU με προσεγγιστικούς πολλαπλασιαστές κινητής υποδιαστολής της βιβλιογραφίας [149, 151, 267, 268]. Στον Πίνακα 5, παρουσιάζουμε τα κέρδη πόρων και τα αποτελέσματα της ακρίβειας των εξεταζόμενων πολλαπλασιαστών. Ο πίνακας περιλαμβάνει δύο νέες μετρικές ακρίβειας, οι οποίες είναι προσαρμοσμένες στην αριθμητική κινητής υποδιαστολής. Συγκεκριμένα, η μετρική PON εκφράζει την πιθανότητα το προσεγγιστικό αποτέλεσμα να είναι υπερχείλιση ενώ το ακριβές είναι κανονικός αριθμός (ή το αντίστροφο). Αντίστοιχα, η μετρική PUN εκφράζει την πιθανότητα το προσεγγιστικό αποτέλεσμα να είναι υποχείλιση ενώ το ακριβές είναι κανονικός αριθμός (ή το αντίστροφο). Σημειώνουμε ότι επιλέγουμε διαμορφώσεις για τον AxFPU που έχουν τιμές λάθους συγκρίσιμες με αυτές των άλλων κυκλωμάτων. Όσον αφορά την ακρίβεια, ο CFPU [267] είναι ο πιο αναποτελεσματικός πολλαπλασιαστής, καθώς προσφέρει περίπου $4\times$–$6\times$ μεγαλύτερο MRED και $PRED_2$ ίσο με 70%. Είναι σημαντικό να αναφέρουμε ότι τόσο το PON όσο και το PUN, δηλαδή οι μετρικές που εξετάζουν απροσδόκητα αποτελέσματα σχετικά με υπερχείλιση και υποχείλιση, είναι κάτω από 0.5% για όλα τα κυκλώματα. Στην περίπτωση της μισής ακρίβειας, ο RMAC [268] έχει ελαφρώς καλύτερα αποτελέσματα ακρίβειας από τον AxFPU, ωστόσο, το κύκλωμα μας παρέχει αυξημένα ενεργειακά κέρδη. Όσον αφορά την απλή ακρίβεια, ο AxFPU προσφέρει καλύτερες τιμές MRED και $PRED_2$ από τον RMAC, καθώς και υψηλότερα κέρδη ενέργειας. Συνολικά, ο προτεινόμενος πολλαπλασιαστής AxFPU παρέχει μεγαλύτερη εξοικονόμηση ενέργειας από τον RMAC [268] για παρόμοια λάθη, καθώς και καλύτερη ενεργειακή απόδοση από τον CFPU [267] με σημαντικά καλύτερη ακρίβεια. Σχετικά με την καθυστέρηση του κρίσιμου μονοπατιού, οι CFPU [267] και RMAC [268] αποδίδουν μεγαλύτερα κέρδη, καθώς μοντελοποιούν τον πολλαπλασιασμό των mantissas με λιγότερο πολύπλοκες πράξεις. Τέλος, τα κέρδη των R4ABM [149] και R8ABM [151] θεωρούνται μικρά σε σύγκριση με αυτά των άλλων κυκλωμάτων. Το πλεονέκτημα τους είναι οι μικρές τιμές λάθους. Ωστόσο, σε περίπτωση που υπάρχει ανάγκη για μικρές τιμές λάθους, ο AxFPU με έναν μικρότερο βαθμό προσέγγισης μπορεί να προσφέρει μεγαλύτερα κέρδη σε πόρους. Ενδεικτικά, αναφέρουμε ότι το MRED του $AxFPU16|_{3,6}$ είναι 1.10% και αυτό του $AxFPU32|_{10,18}$ είναι 1.66%.

Πίνακας 5: Πειραματικά αποτελέσματα των προσεγγιστικών πολλαπλασιαστών κινητής υποδιαστολής μισής (16-bit) και μονής (32-bit) ακρίβειας στη βιβλιοθήκη των 65nm της TSMC.

Κύκλωμα	Καθυστέρηση (%)[1]	Επιφάνεια (%)[1]	Ενέργεια (%)[1]	MRED (%)	PRED₂ (%)	PON (%)	PUN (%)
AxFPU16$\|_{4,6}$	32.9	70.8	67.1	3.33	57.40	0.43	0.10
CFPU16 [267]	44.7	71.6	64.6	12.96	70.68	0.46	0.11
RMAC16 [268]	42.2	71.1	63.2	3.16	49.42	0.20	0
R4ABM16 [149]	7.8	43.3	39.1	2.12	20.61	0.09	0.01
R8ABM16 [151]	10.1	47.6	41.3	1.56	7.14	0	0
AxFPU32$\|_{10,20}$	46.0	89.9	87.4	2.20	44.86	0.26	0.01
CFPU32 [267]	53.9	86.5	79.7	12.80	70.63	0.05	0.02
RMAC32 [268]	50.6	83.9	78.3	2.92	49.73	0.02	0
R4ABM32 [149]	9.1	53.3	50.7	1.44	16.32	0.02	0
R8ABM32 [151]	14.3	59.5	51.1	0.99	5.46	0	0

[1] Αναφέρεται σε % κέρδος καθυστέρησης/επιφάνειας/ενέργειας (σχετική μείωση) σε σύγκριση με το ακριβές κύκλωμα.

Τέλος, αξιολογούμε τον DyFPU, ο οποίος είναι η παραλλαγή του AxFPU με δυνατότητα διαμόρφωσης της προσέγγισης κατά τον χρόνο εκτέλεσης. Υπενθυμίζουμε ότι ο DyFPU$|_{P,R}$ υποδηλώνει τον μοναδικό πολλαπλασιαστή DyFPU που έχει ρυθμιστεί μέσω των σημάτων εισόδου για να εκτελεί τον πολλαπλασιασμό με βαθμό perforation P και βαθμό rounding R. Αντίθετα, ο AxFPU$|_{P,R}$ διαμορφώνεται κατά τον χρόνο σχεδίασης, δηλαδή, ένα νέο κύκλωμα υλοποιείται για κάθε διαμόρφωση της προσέγγισης. Τα αποτελέσματα από αυτή τη σύγκριση είναι παρόμοια με εκείνα της σύγκριση των κυκλωμάτων σταθερής υποδιαστολής (AxFXU και DyFXU). Πιο συγκεκριμένα, όπως αναμενόταν, ο DyFPU επιβάλλει αύξηση της επιφάνειας κατά 4.3% και 2.1% για μισή και απλή ακρίβεια, αντίστοιχα. Όσον αφορά την κατανάλωση ενέργειας, ο DyFPU είναι χειρότερος από τον αντίστοιχο AxFPU, παρέχοντας ∼1.4× και ∼1.6× λιγότερο κέρδος για μεγάλες προσεγγίσεις στη μισή και απλή ακρίβεια, αντίστοιχα. Ωστόσο, εξακολουθεί να είναι πιο ενεργειακά αποδοτικός από τον ακριβή πολλαπλασιαστή, παρέχοντας κέρδη έως και 49.7%, ενώ επιτρέπει επίσης τη δυναμική ρύθμιση του βαθμού προσέγγισης μέσω της απρόσκοπτης αλλαγής των τιμών των παραμέτρων P και R.

6. Συνεργατική Προσέγγιση: Συνδυασμός Αριθμητικών Κωδικοποιήσεων

*Η ενότητα αυτή αφορά το **Κεφάλαιο 6** και βασίζεται στη **δημοσίευση [147]**.*

Εισαγωγή

Ένας από τους στόχους του Προσεγγιστικού Υπολογισμού είναι να προσφέρει σχεδιαστικές λύσεις που μπορούν να χειριστούν διαφορετικούς περιορισμούς σε ακρίβεια και πόρους. Αυτό το χαρακτηριστικό είναι εξαιρετικά σημαντικό, λαμβάνοντας υπόψη ότι οι ανθεκτικές-σε-λάθη εφαρμογές [29, 240] επιβάλλουν ποικίλες απαιτήσεις σχετικά με την ποιότητα των αποτελεσμάτων και παρουσιάζουν διαφορετική διάδοση σφαλμάτων στους υπολογισμούς τους. Οι τεχνικές προσέγγισης εφαρμόζονται σε επίπεδο λογισμικού, αρχιτεκτονικής και υλικού [16, 17, 19]. Για την επέκταση του χώρου προσέγγισης, οι σχεδιαστές εξετάζουν την ταυτόχρονη εφαρμογή πολλαπλών τεχνικών προσέγγισης, είτε από διαφορετικό είτε από το ίδιο επίπεδο. Αν και πρόσφατα εμφανίστηκαν στρατηγικές κάθετης προσέγγισης, δηλαδή σε πολλαπλά επίπεδα [18], το πλήρες δυναμικό της οριζόντιας προσέγγισης, δηλαδή εντός του ίδιου επιπέδου σχεδίασης, παραμένει ένα ανοιχτό ζήτημα για περαιτέρω εξερεύνηση. Σε αυτό το πλαίσιο, διερευνούμε, για πρώτη φορά, την αποτελεσματικότητα του συνδυασμού τεχνικών αριθμητικής προσέγγισης στη σχεδίαση ενεργειακά αποδοτικών κυκλωμάτων πολλαπλασιασμού. Εστιάζουμε στον συνδυασμό προσεγγίσεων στο επίπεδο της αριθμητικής, καθώς αυτή επηρεάζει εγγενώς τόσο τη δυναμική όσο και τη στατική κατανάλωση ενέργειας των κυκλωμάτων. Επιπλέον, η υλοποίηση που παρέχεται σε αυτό το επίπεδο μπορεί να υιοθετηθεί άμεσα σε μια στρατηγική κάθετης σχεδίασης σε πολλαπλά επίπεδα.

Σε αυτή την ενότητα, πραγματοποιούμε μια εκτενή εξερεύνηση του χώρου σχεδίασης συνδυάζοντας τεχνικές αριθμητικής προσέγγισης. Στόχος μας είναι η επέκταση του χώρου προσέγγισης με ποικίλα σύνολα διαμορφώσεων και η εύρεση της πιο αποτελεσματικής λύσης. Χρησιμοποιούμε τον όρο "συνεργατική προσέγγιση" για να δηλώσουμε ότι δύο τεχνικές εφαρμόζονται στη σχεδίαση του ίδιου προσεγγιστικού κυκλώματος. Το αποτέλεσμα της εξερεύνησής μας για τη συνεργατική προσέγγιση είναι 5 οικογένειες προσεγγιστικών υβριδικών πολλαπλασιαστών, οι οποίοι συνδυάζουν τεχνικές που έχουν ήδη εξεταστεί (high-radix, perforation, rounding).

Σχεδίαση Κυκλωμάτων Πολλαπλασιαστή με Συνεργατική Προσέγγιση

Το πρώτο βήμα είναι να εξερευνήσουμε το χώρο σχεδίασης, δηλαδή να εξετάσουμε ποιοι συνδυασμοί των τεχνικών high-radix, perforation και rounding είναι εφικτοί. Υπενθυμίζουμε ότι κάθε μία από αυτές τις τεχνικές προσέγγισης πρακτικά ανάγεται σε προσεγγιστική κωδικοποίηση ενός από τους αριθμούς εισόδου. Ο Πίνακας 6 παραθέτει όλους τους συνδυασμούς θεωρώντας μία κωδικοποίηση ανά όρο του πολλαπλασιασμού. Όπως φαίνεται, υπάρχουν δύο συνδυασμοί, δηλαδή rounding & rounding και perforation & perforation που δεν μπορούν να υλοποιηθούν. Ο συνδυασμός perforation & high-radix ισοδυναμεί με απλή perforation, καθώς το προσεγγιστικό μερικό γινόμενο που παράγεται από τη high-radix απαλείφεται. Επομένως, μπορεί να αποκλειστεί με ασφάλεια από

Πίνακας 6: Συνδυασμός αριθμητικών τεχνικών προσέγγισης.

1ος Όρος	2ος Όρος	Εφαρμοσιμότητα	Όνομα Συνδυασμού
high-radix	high-radix	✓	DRAD
high-radix	perforation	✓	ισοδύναμο με perforation
high-radix	rounding	✓	RADR
rounding	rounding	✗	–
rounding	perforation	✓	ROUP
perforation	perforation	✗	–

Έξτρα Συνδυασμοί	Εφαρμοσιμότητα	Όνομα Συνδυασμού
high-radix & high-radix + perforation	✓	DRADP
high-radix & rounding + perforation	✓	ισοδύναμο με ROUP
rounding & perforation + perforation	✓	ισοδύναμο με ROUP

περαιτέρω εξέταση. Επιπλέον, εξετάζουμε την εφαρμογή της perforation σε όλους τους εφικτούς συνδυασμούς (κάτω μέρος του πίνακα). Ο μόνος επιπλέον ουσιαστικός συνδυασμός είναι η perforation με τη διπλή high-radix, καθώς οι υπόλοιποι ισοδυναμούν με rounding & perforation.

Στη συνέχεια, αναλύουμε τις εφικτές κωδικοποιήσεις και παραθέτουμε τις εξισώσεις των προσεγγιστικών πολλαπλασιασμών.

DRAD: High-Radix & High-Radix

Έστω ότι τα m και k είναι οι παράμετροι διαμόρφωσης της κωδικοποίησης high-radix των A και B, αντίστοιχα. Οι δύο αριθμοί μετασχηματίζονται με τις κωδικοποιήσεις high-radix-2^m και high-radix-2^k, όπως φαίνεται στις Εξ. (20)–(25). Σημειώνουμε ότι ο B κωδικοποιείται ακριβώς όπως στον RAD πολλαπλασιαστή, δηλαδή με την υβριδική κωδικοποίηση radix-4–radix-2^k, ενώ για τον A, τον χωρίζουμε σε δύο λέξεις και κωδικοποιούμε μόνο τη λιγότερο σημαντική με την radix-2^m. Για να δημιουργήσουμε το ψηφίο high-radix στον A, προσθέτουμε και αφαιρούμε τον όρο $2^{m-1}a_{m-1}$ στην αναπαράσταση του.

$$A = -2^{n-1}a_{n-1} + \sum_{i=0}^{n-2} 2^i a_i = A_1 + x_0^{R2^m} \tag{20}$$

$$\text{όπου} \quad A_1 = -2^{n-1}a_{n-1} + \sum_{i=m-1}^{n-2} 2^i a_i + 2^{m-1}a_{m-1} \tag{21}$$

$$x_0^{R2^m} = -2^{m-1}a_{m-1} + 2^{m-2}a_{m-2} + \cdots + a_0 \tag{22}$$

$$B = -2^{n-1}b_{n-1} + \sum_{i=0}^{n-2} 2^i b_i = B_1 + y_0^{R2^k} \tag{23}$$

$$\text{όπου} \quad B_1 = \sum_{j=k/2}^{n/2-1} 4^j y_j^{R4}, \qquad y_j^{R4} = -2b_{2j+1} + b_{2j} + b_{2j-1} \tag{24}$$

$$y_0^{R2^k} = -2^{k-1}b_{k-1} + 2^{k-2}b_{k-2} + \cdots + b_0 \tag{25}$$

Χρησιμοποιώντας τα προσεγγιστικά ψηφία high-radix, δηλαδή $\hat{x}_0^{R2^m} \in \{0, \pm 2^{m-4}, \pm 2^{m-3}, \pm 2^{m-2}, \pm 2^{m-1}\}$ και $\hat{y}_0^{R2^k} \{0, \pm 2^{k-4}, \pm 2^{k-3}, \pm 2^{k-2}, \pm 2^{k-1}\}$, ο πολλαπλασιασμός του $\text{DRAD}|_{k,m}$ υπολογίζεται από την Εξ. (26).

$$\text{DRAD}|_{k,m} = A_1 \cdot B_1 + B_1 \cdot \hat{x}_0^{R2^m} + A \cdot \hat{y}_0^{R2^k} \tag{26}$$

DRADP: High-Radix & High-Radix + Perforation

Συνδυάζοντας τη διπλή κωδικοποίηση high-radix του DRAD με perforation, δημιουργούμε τον πολλαπλασιαστή $\text{DRADP}|_{k,m}$, ο οποίος πρακτικά απαλείφει το λιγότερο σημαντικό μερικό γινόμενο, όπως φαίνεται στην Εξ. (27).

$$\text{DRADP}|_{k,m} = A_1 \cdot B_1 + B_1 \cdot \hat{x}_0^{R2^m} \tag{27}$$

RADR: High-Radix & Rounding

Για αυτόν τον συνδυασμό, θεωρούμε την προσεγγιστική κωδικοποίηση του B με high-radix, δηλαδή, $B_1 + \hat{y}_0^{R2^k}$, ενώ ο A κωδικοποιείται εμμέσως μέσω της εφαρμογής της rounding στα μερικά γινόμενα. Χρησιμοποιούμε μία διαφορετική παραλλαγή της rounding από αυτή που χρησιμοποιήθηκε στους AxFXU/AxFPU. Συγκεκριμένα, περικόπτουμε τις R λιγότερο σημαντικές στήλες του πίνακα (δέντρου) των μερικών γινομένων που δημιουργείται από το γινόμενο $A \cdot B_1$. Ως αποτέλεσμα, ο πολλαπλασιασμός του $\text{RADR}|_{k,R}$ υπολογίζεται από την Εξ. (28).

$$\text{RADR}|_{k,R} = A \cdot B_1|_R + A \cdot \hat{y}_0^{R2^k} \tag{28}$$

Για να αντισταθμίσουμε το σφάλμα της περικοπής, εισάγουμε έναν διορθωτικό όρο στη θέση του περικομμένου MSB του κάθε μερικού γινομένου. Αυτός ο όρος είναι διαφορετικός για κάθε μερικό γινόμενο και σχετίζεται με την κωδικοποίηση radix-4, δηλαδή ισούται με $one_j + two_j$. Υπενθυμίζουμε ότι τα σήματα one_j και two_j ενεργοποιούνται εάν το j μερικό γινόμενο είναι ίσο με $\pm 1A$ ή $\pm 2A$, αντίστοιχα. Για να βελτιώσουμε

περαιτέρω την rounding, επισυνάπτουμε μία σταθερά '1' στη θέση που ο υποπίνακας μερικών γινομένων του $A \cdot B_1$ περικόπτεται.

ROUP: Rounding & Perforation

Ο συνδυασμός rounding & perforation έχει ήδη εξεταστεί στους πολλαπλασιαστές AxFXU/AxFPU. Επομένως, σε αυτή την ενότητα χρησιμοποιούμε δύο διαφορετικές παραλλαγές της rounding. Η κύρια διαφοροποίηση τους σε σχέση με τη rounding των AxFXU/AxFPU είναι ότι δεν πραγματοποιείται στρογγυλοποίηση στο ίδιο bit για κάθε μερικό γινόμενο. Αντίθετα, το bit στρογγυλοποίησης του κάθε μερικού γινομένου επιλέγεται με βάση τη σημαντικότητα του στο δέντρο γινομένων. Ονομάζουμε αυτή την τεχνική ως ασύμμετρη rounding.

Ο πρώτος συνδυασμός φέρει την ετικέτα ROUP1$|_{P,R}$ και χρησιμοποιεί την rounding του RADR. Συγκεκριμένα, απαλείφουμε τα P λιγότερο σημαντικά γινόμενα, και στη συνέχεια εφαρμόζουμε rounding περικόπτοντας τις R λιγότερο σημαντικές στήλες του πίνακα μερικών γινομένων και εισάγοντας όρους διόρθωσης. Ο πολλαπλασιασμός του ROUP1 υπολογίζεται από την Εξ. (29).

$$ \text{ROUP1}|_{P,R} = \left. \sum_{j=P}^{n/2-1} 4^j \tilde{P} P_j \right|_R = \left. \sum_{j=P}^{n/2-1} 4^j A \cdot y_j^{R4} \right|_R \tag{29} $$

Ο δεύτερος συνδυασμός φέρει την ετικέτα ROUP2$|_{P,R}$ και μετατρέπει την rounding των AxFXU/AxFPU από συμμετρική σε ασύμμετρη. Ο πολλαπλασιασμός του ROUP2 εκτελείται με την άθροιση των στρογγυλεμένων μερικών γινομένων, όπως φαίνεται στις Εξ. (30)–(31).

$$ \text{ROUP2}|_{P,R} = \sum_{j=P}^{n/2-1} 4^j \tilde{P} P_j = \sum_{j=P}^{n/2-1} 4^j A_{R_j} \cdot y_j^{R4} \tag{30} $$

$$ \text{όπου} \quad A_{R_j} = \langle a_{n-1} a_{n-2} \cdots a_{R_j} \rangle_{2's} + a_{R_j - 1} \tag{31} $$

Σε σύγκριση με τις αντίστοιχες εξισώσεις της rounding των AxFXU/AxFPU, η μόνη διαφορά είναι ότι υπάρχουν j A_R λέξεις (μία ανά μερικό γινόμενο) αντί για μία. Οι υπόλοιπες βελτιστοποιήσεις χαμηλού επιπέδου είναι οι ίδιες και έχουν ως αποτέλεσμα να ενσωματωθούν τα bits $a_{R_j - 1}$ με μια πύλη XOR στους όρους διόρθωσης, δηλαδή $(sign_j \oplus a_{R_j - 1}) \cdot (one_j + two_j)$. Σημειώνουμε ότι ο συμβολισμός ROUP2$|_{P,R}$ θεωρεί $R = R_P$, δηλαδή δείχνει τη διαμόρφωση της rounding του λιγότερο σημαντικού μερικού γινομένου από τα εναπομείναντα.

Πειραματική Αξιολόγηση

Αρχικά, εξετάζουμε την ακρίβεια των προτεινόμενων πολλαπλασιαστών, χρησιμοποιώντας την μετρική MRED (μέσος όρος των σχετικών αποστάσεων λάθους για ένα σύνολο από ζεύγη εισόδων πολλαπλασιαστή). Ο χώρος σχεδίασης των προτεινόμενων πολλαπλασιαστών είναι πάρα πολύ μεγάλος, καθώς σε κάθε κύκλωμα ο βαθμός προσέγγισης διαμορφώνεται από δύο ανεξάρτητες παραμέτρους. Για αυτόν τον λόγο, για κάθε διαμόρφωση της προσέγγισης υπολογίζουμε το MRED για ένα σύνολο από 200K ζεύγη εισόδου, τα οποία είναι ομοιόμορφα κατανεμημένα στο εύρος των 16-bit αριθμών.

Το Σχήμα 12 παρουσιάζει τη διακύμανση του MRED σε σχέση με τη διαμόρφωση προσέγγισης. Όσον αφορά τον DRAD, παρόλο που το εύρος λαθών του είναι μικρό, δηλαδή [0.15%, 1.65%], το λάθος κλιμακώνεται γρήγορα δημιουργώντας κενά διαστήματα. Ωστόσο, η κλιμάκωση του λάθους του παρέχει αυξημένη πυκνότητα σε σύγκριση με τον RAD, ο οποίος έχει μόνο τρεις διαμορφώσεις προσέγγισης ($k = 6, 8, 10$) με τιμές MRED 0.08%, 0.28% και 0.93%. Όπως ήταν αναμενόμενο, για τις ίδιες διαμορφώσεις, ο DRADP εμφανίζει μεγαλύτερες τιμές λάθους από τον DRAD, ξεκινώντας από 0.49%, ενώ παρατηρείται και πάλι ταχεία κλιμάκωση λάθους. Το πλεονέκτημα του RADR είναι ότι εξομαλύνει την ταχεία κλιμάκωση σφαλμάτων του RAD, προσθέτοντας πολλαπλές τιμές μεταξύ δύο διαδοχικών προσεγγίσεων, ειδικά για τις μικρές τιμές του k. Το εύρος λαθών και η κλιμάκωση των ROUP1 και ROUP2 είναι παρόμοια με αυτά του AxFXU. Ο ROUP1 διαθέτει αργή κλιμάκωση λάθους από 0.04% ($P = 1$) έως 2.47% ($P = 4$) με πολλές ενδιάμεσες τιμές. Οι τιμές MRED του ROUP2 είναι παρόμοιες, ωστόσο, το μέγιστο λάθος του είναι μικρότερο (2.07%) σε σύγκριση με αυτό του ROUP1 (2.47%).

Στο Σχήμα 13 παρουσιάζουμε το διάγραμμα MRED–ενέργειας με όλα τα προτεινόμενα προσεγγιστικά κυκλώματα της Διατριβής. Όσον αφορά τους πολλαπλασιαστές high-radix που σχεδιάστηκαν με συνεργατική προσέγγιση (DRAD, DRADP, RADR), αυξάνουν την ανάλυση της καμπύλης σημείων του απλού RAD. Η καμπύλη επίσης βελτιώνεται από τον DRADP, ο οποίος αποτελεί την καλύτερη σχεδιαστική λύση high-radix στις περισσότερες περιπτώσεις. Σημειώνουμε ότι παρουσιάζουμε μόνο τους πολλαπλασιαστές RAD για $k = 6, 8, 10$, επειδή για $k \geq 12$ παράγονται μεγάλες τιμές λάθους. Αυτός ο περιορισμός της απλής τεχνικής high-radix ξεπερνιέται με τη μέθοδο της συνεργατικής προσέγγισης. Όσον αφορά τους πολλαπλασιαστές που εφαρμόζουν perforation & rounding, οι τεχνικές ασύμμετρης rounding των ROUP1 και ROUP2 υπερτερούν της συμμετρικής rounding του AxFXU. Συμπεραίνουμε ότι οι πιο λεπτομερείς βελτιστοποιήσεις σε επίπεδο bit, όπως η προσαρμοσμένη rounding ανά μερικό γινόμενο του ROUP2, παρέχουν καλύτερα αποτελέσματα από καθολικές λύσεις, όπως η συμμετρική rounding του AxFXU. Επιπλέον, για μικρές τιμές λάθους, ο AxFXU δεν θεωρείται ο πιο ενεργειακά αποδοτικός πολλαπλασιαστής, καθώς αρκετές διαμορφώσεις
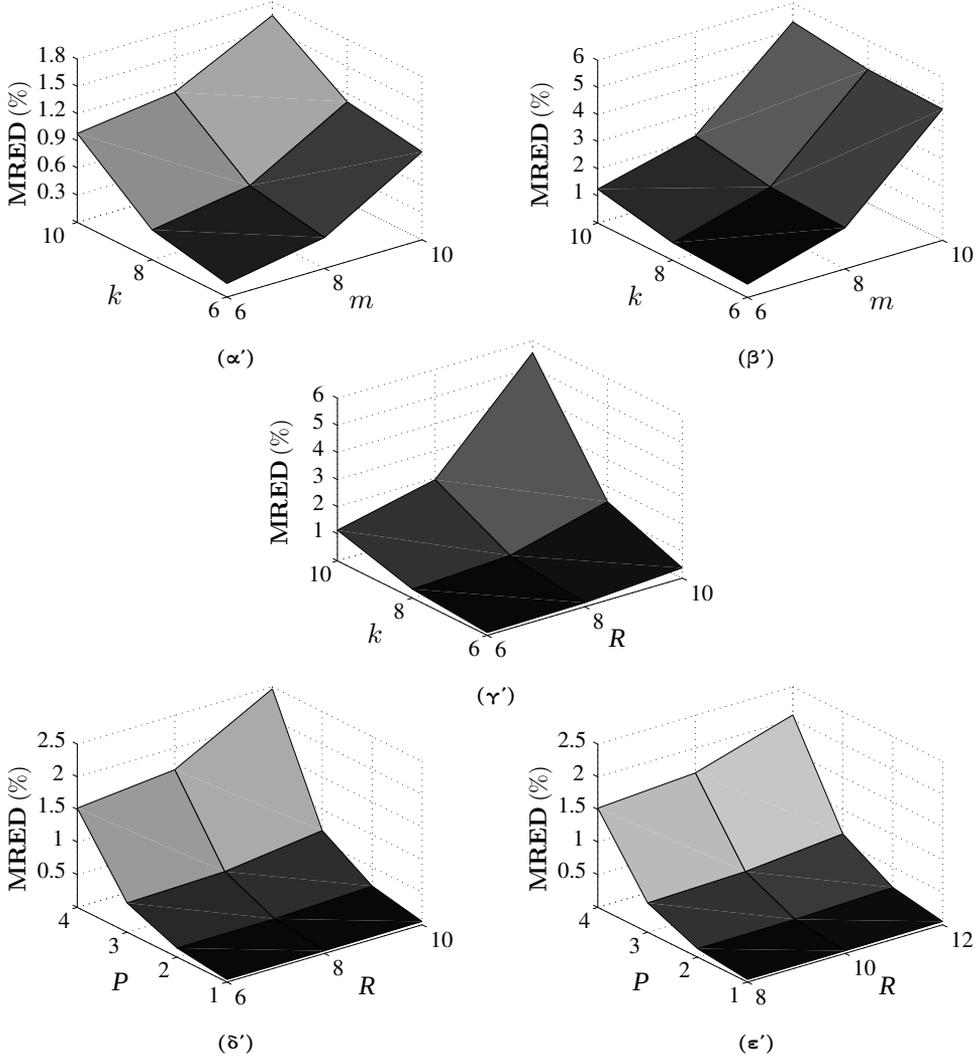
Σχήμα 12: Η διακύμανση του MRED των 16-bit πολλαπλασιαστών που βασίζονται στη συνεργατική προσέγγιση: **(α′)** DRAD$|_{k,m}$, **(β′)** DRADP$|_{k,m}$, **(γ)** RADR$|_{k,R}$, **(δ′)** ROUP1$|_{P,R}$, **(ε′)** ROUP2$|_{P,R}$.

της συνεργατικής προσέγγισης παρέχουν καλύτερη κατανάλωση ενέργειας. Ωστόσο, σημειώνουμε ότι το πλεονέκτημα του AxFXU είναι ότι διευκολύνει τη δυναμική διαμόρφωση της προσέγγισης, όπως δείξαμε στην προηγούμενη ενότητα, η οποία θα ήταν πιο δύσκολη στους ROUP1 και ROUP2 λόγω της διαφορετικής rounding ανά μερικό γινόμενο.
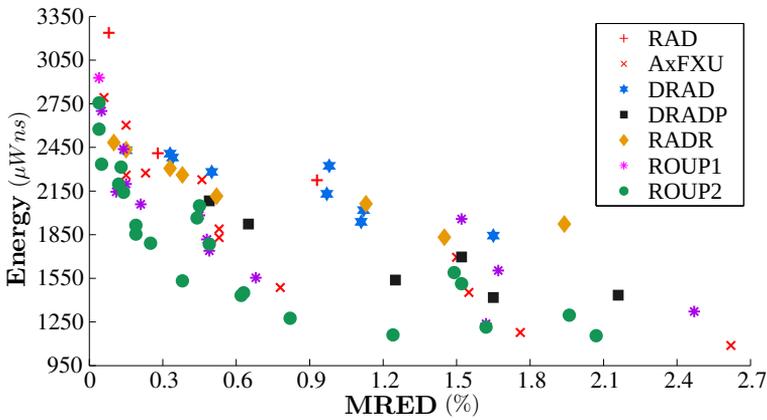
Σχήμα 13: Η συγκριτική ανάλυση Pareto όλων των προσεγγιστικών πολλαπλασιαστών της Διατριβής ως προς το μέσο σχετικό σφάλμα και την κατανάλωση ενέργειας.

Συνολικά, όπως φαίνεται στο διάγραμμα, το σύνορο Pareto σχηματίζεται αποκλειστικά από τους πολλαπλασιαστές ROUP2. Η οικογένεια πολλαπλασιαστών ROUP2 παρέχει την καλύτερη ισορροπία ενέργειας–λάθους και βελτιώνει περαιτέρω το σύνορο Pareto των RAD [153] και AxFXU [144] κατά 1.5×–2×. Επιπλέον, είναι σημαντικό να αναφέρουμε ότι αυξάνει την ανάλυση του συνόρου (της καμπύλης σημείων/διαμορφώσεων), δηλαδή επεκτείνει ακόμη περισσότερο τον ήδη μεγάλο χώρο προσέγγισης, παρέχοντας μεγάλη ποικιλία επιλογών σχεδίασης. Ως τελικό στάδιο αξιολόγησης, συγκρίνουμε το νέο σύνορο Pareto με άλλα σημαντικά κυκλώματα της βιβλιογραφίας [142,149,151,266]. Συγκεκριμένα, αξιολογούμε την κατανάλωση ενέργειας των κυκλωμάτων για διάφορους περιορισμούς MRED στην περιοχή 0.15%–1.47%. Ο μεγάλος χώρος προσέγγισης και τα πυκνά διαστήματα λάθους του ROUP2 παρέχουν αυξημένη ευελιξία και επομένως, μπορούμε να επιλέξουμε διαμορφώσεις με MRED ίσο ή ελαφρώς μικρότερο από έναν δεδομένο περιορισμό MRED. Συνεπώς, ο ROUP2 μπορεί να μεγιστοποιήσει τα κέρδη του για οποιοδήποτε περιορισμό λάθους, παρέχοντας σημαντικά κέρδη που κυμαίνονται από 29% έως 63% σε σύγκριση με τους πολλαπλασιαστές της βιβλιογραφίας.

7. Προσεγγιστικοί DSP & AI Επιταχυντές Υλικού

*Η ενότητα αυτή αφορά το **Κεφάλαιο 7** και βασίζεται στις **δημοσιεύσεις [144, 145, 147, 153, 274–277]**.*

Εισαγωγή

Σε αυτή την ενότητα, εστιάζουμε στην ανάπτυξη προσεγγιστικών DSP και AI επιταχυντών με Γλώσσας Περιγραφής Υλικού (HDL), στοχεύοντας τόσο σε τεχνολογία FPGA όσο και σε ASIC. Η γενική μέθοδος προσέγγισης έγκειται στη χρήση των προτεινόμενων προσεγγιστικών πολλαπλασιαστών της Διατριβής σε κλασικές αρχιτεκτονικές υλικού για DSP και AI. Τα αριθμητικά κυκλώματα αποτελούν σημαντικές μονάδες επεξεργασίας στους επιταχυντές υλικού, καθώς επηρεάζουν εγγενώς την κατανάλωση ενέργειας και την απόδοση ολόκληρης της εφαρμογής. Ο αντίκτυπός τους γίνεται ακόμη μεγαλύτερος αν λάβουμε υπόψη ότι οι σύγχρονοι επιταχυντές υλοποιούν παράλληλες αρχιτεκτονικές με πολλαπλές μονάδες επεξεργασίας.

Η ενσωμάτωση των προσεγγιστικών κυκλωμάτων μας σε επιταχυντές βασίζεται σε μια μεθοδολογία σχεδίασης που εξερευνά τον χώρο σχεδίασης σε επίπεδο λογισμικού και υλικού. Στόχος μας είναι να αξιολογήσουμε τα προσεγγιστικά κυκλώματα της Διατριβής σε πραγματικές εφαρμογές DSP/AI και αντίστροφα, να διερευνήσουμε την ανθεκτικότητα των εφαρμογών σε λάθη και να ποσοτικοποιήσουμε τα κέρδη πόρων των προσεγγιστικών επιταχυντών. Επιπλέον, η κύρια μέθοδος προσέγγισης που εφαρμόζουμε (προσέγγιση των πολλαπλασιασμών) επιτρέπει την απρόσκοπτη μείωση των πόρων και την βελτίωση της απόδοσης εφαρμογών DSP/AI, χωρίς να τροποποιείται η αρχική αρχιτεκτονική υλικού. Συγκεκριμένα, βελτιώνουμε την απόδοση των DSP επιταχυντών χωρίς να ρυθμίζουμε την αριθμητική τους (μία τυπική εργασία στην ανάπτυξης πυρήνων υλικού). Ειδικά για τα Συνελικτικά Νευρωνικά Δίκτυα (CNNs), παρέχουμε προσεγγιστικούς επιταχυντές χωρίς επανεκπαίδευση, η οποία μπορεί να απαιτείται ώστε να μειωθεί η απώλεια της ακρίβειας που προκαλείται από τις προσεγγίσεις. Η εξάλειψη της ανάγκης για πρόσθετη εκπαίδευση/εξερεύνηση θεωρείται πλεονέκτημα, καθώς ενδέχεται να μην είναι διαθέσιμα ιδιόκτητα σύνολα δεδομένων ή τα μοντέλα των δικτύων. Επίσης, ο χρόνος εκπαίδευσης συνήθως αυξάνεται λόγω της εξομοίωσης των προσεγγίσεων υλικού και της χρήσης προσαρμοσμένων προσεγγιστικών αριθμητικών τελεστών.

Σχεδίαση Προσεγγιστικών DSP & AI Επιταχυντών Υλικού

Η προτεινόμενη μεθοδολογία για τη σχεδίαση προσεγγιστικών επιταχυντών DSP και AI απεικονίζεται στο Σχήμα 14. Αποτελείται από δύο στάδια: την εξερεύνηση σε επίπεδο λογισμικού και την ανάπτυξη του επιταχυντή σε επίπεδο υλικού. Σημειώνουμε ότι η μεθοδολογία μας ακολουθεί τα τυπικά βήματα της σχεδίασης υλικού. Ωστόσο, περιλαμβάνει πρόσθετες λειτουργίες (χρήση προσεγγιστικών αριθμητικών μονάδων), ανάλυση λαθών (λόγω των προσεγγίσεων) και εξερεύνηση του χώρου σχεδίασης για διαφορετικές τεχνικές προσέγγισης, αριθμητικά πρότυπα και αλγόριθμους των εφαρμογών. Η λεπτομερής εξερεύνηση του χώρου σχεδίασης είναι πολύ σημαντική, επειδή κάθε συνδυασμός
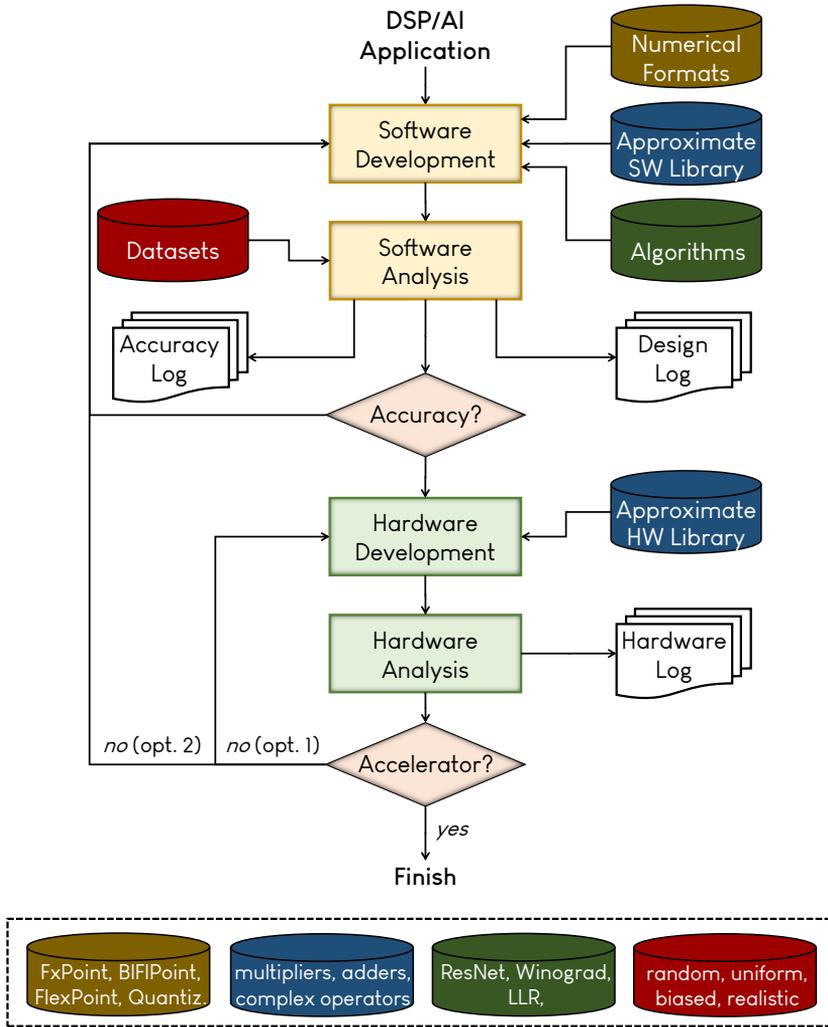
Σχήμα 14: Μεθοδολογία σχεδίασης για προσεγγιστικούς DSP & AI επιταχυντές υλικού.

προσεγγίσεων, αριθμητικών προτύπων και αλγορίθμων έχει διαφορετικό αντίκτυπο στην ακρίβεια, τους πόρους και την απόδοση του επιταχυντή.

Πριν αναλύσουμε τα βήματα της μεθοδολογίας μας, κάνουμε τις ακόλουθες διευκρινίσεις:

- Προσεγγιστική Βιβλιοθήκη Σχεδίασης: περιλαμβάνει τα μοντέλα λογισμικού και τις περιγραφές υλικού των προσεγγιστικών αριθμητικών μονάδων (π.χ., RAD

[153], AxFXU [144], AxFPU [145] και ROUP [147]), όπως επίσης και την α-
νάλυση των λαθών και τα πειραματικά τους αποτελέσματα.

- <u>Σύνολο Αριθμητικών Προτύπων</u>: περιλαμβάνει διάφορες αριθμητικές, π.χ., το συμ-
βατικό πρότυπο σταθερής και κινητής υποδιαστολής, το μπλοκ κινητής υποδιαστο-
λής [306], το Flexpoint [301] και την κβάντιση σταθερής υποδιαστολής [307].

- <u>Σύνολο Αλγορίθμων</u>: περιλαμβάνει αλγόριθμους για ολόκληρη την εφαρμογή (π.χ.,
το μοντέλο ResNet [308] ως CNN ή τη μέθοδο Log-Likelihood-Ratio [309] για
την αποδιαμόρφωση σήματος), καθώς και βασικές λειτουργίες (π.χ., τη μέθοδο
Winograd [310] για τη συνέλιξη εικόνας).

- <u>Σύνολο Σετ Δεδομένων</u>: περιλαμβάνει διάφορα σετ δεδομένων (π.χ., τυχαία, Gaus-
sian/ομοιόμορφα κατανεμημένα και ρεαλιστικά δεδομένα) για την αξιολόγηση της
ποιότητας των αποτελεσμάτων σε επίπεδο λογισμικού.

Αρχικά, αναπτύσσουμε την εφαρμογή σε λογισμικό χρησιμοποιώντας διάφορα αριθμητικά
πρότυπα και αλγόριθμους. Τα μοντέλα λογισμικού μας έχουν ακρίβεια bit, γεγονός που
μας επιτρέπει να μελετάμε την ακρίβεια της εφαρμογής όταν χρησιμοποιούμε προσεγγι-
στικές αριθμητικές μορφές ή/και προσεγγιστικούς τελεστές. Για να λάβουμε τα ακριβή
(ορθά) αποτελέσματα, εκτελούμε προσομοιώσεις με τα σετ δεδομένων. Στη συνέχεια,
αρχίζουμε να ενσωματώνουμε προσεγγιστικές αριθμητικές μονάδες και να δημιουργο-
ύμε προσεγγιστικές παραλλαγές του στοχευμένου επιταχυντή. Για κάθε παραλλαγή,
εκτελούμε προσομοιώσεις για να λάβουμε τα προσεγγιστικά αποτελέσματα και να αξιο-
λογήσουμε την ποιότητα τους με βάση τις μετρικές της εκάστοτε εφαρμογής. Επιπλέον,
για κάθε παραλλαγή, αναφέρουμε πιθανές επιβαρύνσεις σε υλικό, π.χ., σχετικά με την
παραλληλοποίηση ή την υλοποίηση πολύπλοκων λειτουργιών. Το επόμενο βήμα είναι να
εξετάσουμε ποιες προσεγγιστικές παραλλαγές ικανοποιούν την επιθυμητή ποιότητα των
αποτελεσμάτων και να επιλέξουμε τις πιο αποτελεσματικές για υλοποίηση σε τεχνολογία
FPGA/ASIC. Σε περίπτωση που δεν πληρούνται οι περιορισμοί ακρίβειας, σχεδιάζουμε
προσεγγιστικές παραλλαγές συνδυάζοντας διαφορετικές αριθμητικές, αλγόριθμους και
προσεγγιστικές μονάδες.

Μετά την εξερεύνηση σε επίπεδο λογισμικού, συνεχίζουμε με την ανάπτυξη της ε-
φαρμογής σε HDL. Υλοποιούμε τις επιλεγμένες προσεγγιστικές παραλλαγές ενσωμα-
τώνοντας τις σχεδιαστικές μας επιλογές για αριθμητικές, αλγόριθμους και τεχνικές
προσέγγισης. Τέλος, στη φάση της αξιολόγησης εξετάζουμε εάν ο επιταχυντής ικα-
νοποιεί τους περιορισμούς για απόδοση και κατανάλωση πόρων. Σε περίπτωση που
τα αποτελέσματα δεν είναι τα αναμενόμενα, υιοθετούμε εναλλακτικούς σχεδιασμούς
(π.χ., σχήμα παραλληλοποίησης) ή επιστρέφουμε στην αρχική εξερεύνηση σε επίπε-
δο λογισμικού για να δημιουργήσουμε νέες προσεγγιστικές παραλλαγές της εφαρμο-
γής.

Πειραματική Αξιολόγηση

Με βάση τη μεθοδολογία μας και τη χρήση κλασικών τεχνικών σχεδίασης και παραλληλο-ποίησης, υλοποιούμε αρκετούς προσεγγιστικούς επιταχυντές για επεξεργασία σημάτων και νευρωνικά δίκτυα. Οι τεχνικές λεπτομέρειες των επιταχυντών και των αρχιτεκτονι-κών υλικού αναλύονται στις δημοσιεύσεις μας [144,145,147,153,274–277]. Επίσης, στις ίδιες δημοσιεύσεις παραθέτουμε επίσης μία πληθώρα πειραματικών αποτελεσμάτων, αξιο-λογούμε τις σχεδιαστικές επιλογές μας, αναλύουμε τα ανταλλάγματα ακρίβειας–πόρων, και εξάγουμε τις καλύτερες λύσεις.

Τα πιο σημαντικά πειραματικά αποτελέσματα συνοψίζονται στον Πίνακα 7. Όσον α-φορά την ποιότητα των αποτελεσμάτων, οι προσεγγίσεις μας προκαλούν μικρή απώλεια ακρίβειας, η οποία μπορεί να ρυθμιστεί στο επιθυμητό επίπεδο λόγω των πολλαπλών διαφορετικών διαμορφώσεων της προσέγγισης που προσφέρονται από τις οικογένειες κυκλωμάτων της Διατριβής. Πιο συγκεκριμένα, επιτυγχάνουμε τυπικές τιμές στις με-τρικές επεξεργασίας εικόνας και μικρά λάθη σε εφαρμογές που βασίζονται σε υπολο-γισμούς πολλαπλασιασμού-άθροισης (φιλτράρισμα σήματος, πολλαπλασιασμός πινάκων και παραγοντοποίηση LU). Οι προσεγγίσεις μας παρέχουν επίσης καλά αποτελέσματα ακρίβειας σε εφαρμογές/αλγόριθμους από άλλους τομείς, όπως οι τηλεπικοινωνίες (α-ποδιαμόρφωση QAM) και η μηχανική μάθηση (αλγόριθμος K-means). Επιπλέον, τα προσεγγιστικά CNNs παρουσιάζουν μικρή απώλεια ακρίβειας σε διάφορες αριθμητικές (16-bit σταθερής υποδιαστολής, 32/16-bit κινητής υποδιαστολής, 8-bit κβαντισμένων ακεραίων). Σε επίπεδο επιταχυντή, επιτυγχάνουμε αξιοσημείωτα κέρδη πόρων τόσο σε ASIC όσο και σε FPGA. Ανάλογα με την εφαρμογή, προσφέρουμε κέρδη ισχύος/ενέρ-γειας στο διάστημα 30%–70% στην τεχνολογία ASIC, καθώς και πολύτιμη μείωση της λογικής των κυκλωμάτων, δηλαδή έως 65% στα LUTs του FPGA και έως 60%–70% στα λογικά κελιά του ASIC.

8. Επιτάχυνση DSP στα Νέα FPGAs Διαστημικού Βαθμού

*Η ενότητα αυτή αφορά το **Κεφάλαιο 8** και βασίζεται στη **δημοσίευση [319]**.*

Εισαγωγή

Τα τελευταία χρόνια, η επιτάχυνση υλικού με FPGAs έχει αποκτήσει αυξημένη δυναμική λόγω της ελκυστικής απόδοσης-ανά-Watt που προσφέρει [3,20,279]. Ως αποτέλεσμα, τα FPGAs βρίσχουν εφαρμογή σε διάφορους τομείς, όπως η ρομποτική, οι τηλεπικοινωνίες,

Πίνακας 7: Συνοπτικά αποτελέσματα των προσεγγιστικών DSP & AI επιταχυντών της Διατριβής.

Εφαρμογή	Προσέγγιση	Επιταχυντής	Κέρδος	Ακρίβεια
Sobel Edge Detector	RAD, AxFXU	65-nm ASIC	58% ενέργεια	CER > 98.5%
FIR Filter	RAD, AxFXU	65-nm ASIC	34% ενέργεια	MRED = 3.6%
Matrix Multiplication	RAD, AxFXU	65-nm ASIC	71% ενέργεια	MRED = 0.9%
QAM Demodulation	RAD	ZCU106 FPGA	65% LUTs	BER $\in [10^{-1}, 10^{-4}]$
ShipDetect CNN	RAD	Zynq-7020 FPGA	38% LUTs	0.2% απώλεια
Gaussian Blurring	AxFPU16	65-nm ASIC	40% ισχύς	PSNR = 51dB
Gaussian Blurring	AxFPU32	65-nm ASIC	57% ισχύς	PSNR = 55dB
CNN/MNIST	AxFPU16	65-nm ASIC	49% ισχύς	2.7% απώλεια
CNN/MNIST	AxFPU32	65-nm ASIC	66% ισχύς	5.5% απώλεια
CNN/CIFAR-10	AxFPU16	65-nm ASIC	39% ισχύς	2.5% απώλεια
CNN/CIFAR-10	AxFPU32	65-nm ASIC	58% ισχύς	5.4% απώλεια
K-Means Clustering	AxFPU32	–	–	ECR = 0%
LU Decomposition	AxFPU32	–	–	MRED $\in [0.1, 0.8]$%
ResNet-8/CIFAR-10	ROUP	45-nm ASIC	54% ενέργεια	4% απώλεια

και το διάστημα. Η βιομηχανία του διαστήματος είναι μία από τις κοινότητες που ανα-
ζητούν ενσωματωμένες πλατφόρμες υψηλής απόδοσης για να χειριστούν τις αυξημένες
υπολογιστικές απαιτήσεις και τις γρήγορες μεταφορές δεδομένων των σύγχρονων δια-
στημικών εφαρμογών. Η βελτιωμένη απόδοση των FPGAs διευκολύνει τον ενσωματω-
μένο υπολογισμό στο διάστημα, π.χ., για εργασίες παρατήρησης της Γης και πλοήγησης,
καθώς μειώνει την ανάγκη για μεταφορά των δεδομένων των αισθητήρων στους επίγειους
σταθμούς. Σε αυτό το πλαίσιο, τα FPGAs διαστημικού βαθμού (space-grade) και τα
κλασικά εμπορικά αξιολογούνται συνεχώς [322–329] ως ενσωματωμένοι επιταχυντές/ε-
πεξεργαστές διαστημικών εφαρμογών. Όταν η ανθεκτικότητα σε ακτινοβολία είναι υ-
ψίστης σημασίας, χρησιμοποιούνται FPGAs διαστημικού βαθμού αντί των εμπορικών
ώστε να αυξηθεί η αξιοπιστία. Η βιβλιογραφία περιλαμβάνει επίσης πολυάριθμες δουλειές
με αρχιτεκτονικές επεξεργασίας για το διάστημα που περιλαμβάνουν FPGAs [330–332].
Σχετικά με τους αλγόριθμους που υλοποιούνται στα FPGAs, εκτός από την επιτάχυν-
ση DSP, επιταχύνονται εργασίες κωδικοποίησης/αποκωδικοποίησης δεδομένων οργάνων
και αισθητήρων (π.χ., που μεταδίδονται μέσω του πρωτοκόλλου SpaceWire/SpaceFi-
bre [333]) και συμπίεσης δεδομένων [334, 335].

Η αγορά προσφέρει μία περιορισμένη ποικιλία από FPGAs διαστημικού βαθμού, όπως
φαίνεται στην ανάλυση μας στη δημοσίευση [319]. Πολύ πρόσφατα, υπάρχουν νέες πολ-
λά υποσχόμενες προσθήκες σε αυτήν την περιορισμένη δεξαμενή συσκευών διαστημικού
βαθμού. Η εταιρία NanoXplore [343] κατασκευάζει τα πρώτα Ευρωπαϊκά ανθεκτικά-
σε-ακτινοβολία FPGAs υψηλής πυκνότητας [344, 345], γνωστά και ως BRAVE (Big
Re-programmable Array for Versatile Environments). Τα BRAVE FPGAs έχουν προ-

σελκύσει το ενδιαφέρον της διαστημικής κοινότητας και αναμένεται να διαδραματίσουν βασικό ρόλο στις επερχόμενες διαστημικές αποστολές, ειδικά στην Ευρώπη. Αυτή η νέα διαστημική οικογένεια FPGAs περιλαμβάνει τσιπ χαμηλής και υψηλής χωρητικότητας που βασίζονται σε SRAM τεχνολογία. Τα τρία πρώτα FPGAs της NanoXplore είναι το NG-Medium (65nm), το NG-Large (65nm) και το NG-Ultra (28nm). Η εταιρεία παρέχει επίσης το εργαλείο λογισμικού NXmap για την ανάπτυξη και τον προγραμματισμό. Το NG-Medium είναι το πρώτο FPGA της σειράς BRAVE και έγινε διαθέσιμο το 2016 μαζί με τις αρχικές εκδόσεις εργαλείων. Τα BRAVE FPGAs ενσωματώνουν όλους τους παραδοσιακούς προγραμματιζόμενους λογικούς πόρους. Επιπλέον, τα NG-Large και NG-Ultra ενσωματώνουν τους επεξεργαστές ARM Cortex-R5 και ARM Cortex-R52, αντίστοιχα. Τέλος, τα BRAVE FPGAs περιλαμβάνουν λειτουργίες που είναι απαραίτητες για ενσωματωμένους υπολογιστές στο διάστημα, όπως η διεπαφή SpaceWire για γρήγορες μεταφορές δεδομένων και ο καθαρισμός της μνήμης προγραμματισμού για τη διασφάλιση συνεχούς λειτουργικότητας.

Η αποτελεσματική χρήση μιας νέας οικογένειας FPGAs όπως η BRAVE, καθώς και η πλήρης εκμετάλλευση των σχετικών εργαλείων λογισμικού, απαιτούν μια συστηματική και πειθαρχημένη προσέγγιση. Επιπροσθέτως, ο προγραμματιστής πρέπει να είναι σε θέση να ξεπεράσει τα θέματα και τους περιορισμούς των νέων εργαλείων ώστε να παρέχει αποδεκτή χρήση πόρων και επαρκή απόδοση. Η ανάπτυξη σε αυτά τα FPGAs διαστημικού βαθμού γίνεται ακόμη πιο δύσκολη λόγω των ιδιαίτερων χαρακτηριστικών τους σε σύγκριση με τα εμπορικά (π.χ., χαμηλότερη απόδοση). Για αυτούς τους λόγους, η European Space Agency (ESA) υποστηρίζει ένα σύνολο ερευνητικών δραστηριοτήτων (QUEENS1/QUEENS2/QUEENS3), οι οποίες περιλαμβάνουν συγκριτική αξιολόγηση του υλικού και δοκιμές των εργαλείων λογισμικού των BRAVE FPGAs. Αυτές οι δραστηριότητες στοχεύουν στη βελτίωση των νέων συσκευών και εργαλείων και στην αξιολόγηση της καταλληλότητας της χρήσης των BRAVE FPGAs ως ενσωματωμένων επεξεργαστών δεδομένων σε διαστημικές αποστολές. Στο πλαίσιο αυτών των δραστηριοτήτων, αναπτύσσουμε μια μεθοδολογία για την υποστήριξη της σχεδίασης στα BRAVE FPGAs και την αξιολόγηση των συσκευών/εργαλείων σε σύγκριση με λύσεις που παρέχονται από καθιερωμένους προμηθευτές της αγοράς. Σημειώνουμε ότι παρόλο που η μεθοδολογία μας εφαρμόζεται σε FPGAs διαστημικού βαθμού, μπορεί να υιοθετηθεί για να βοηθήσει τη σχεδίαση σε οποιαδήποτε πλατφόρμα FPGA ή να πραγματοποιήσει συγκριτική αξιολόγηση μεταξύ διαφορετικών FPGAs. Επιπλέον, ο στόχος μας δεν είναι να σχεδιάσουμε νέες παράλληλες αρχιτεκτονικές DSP. Αντίθετα, χρησιμοποιούμε πυρήνες υλικού κυρίως από το πεδίο της CV, οι οποίοι αναπτύχθηκαν σε προηγούμενες δραστηριότητες της ESA [326–329] και τους επιταχύνουμε στην τεχνολογία BRAVE. Πιο συγκεκριμένα, με συστηματικό τρόπο, αναπτύσσουμε αυτούς τους πυρήνες υψηλής απόδοσης στα FPGAs της NanoXplore και σε άλλα FPGAs της αγοράς με παρόμοια αρχιτεκτονική, και αξιολογούμε τα εργαλεία λογισμικού και το υποκείμενο υλικό.

Μεθοδολογία Σχεδίασης για τα BRAVE FPGAs Διαστημικού Βαθμού

Σε αυτήν την ενότητα, εισάγουμε μια μεθοδολογία για την ανάπτυξη πυρήνων DSP στα BRAVE FPGAs, καθώς και για την αξιολόγηση των δυνατοτήτων των εργαλείων και των τσιπ. Η μεθοδολογία μας αφορά τα κύρια στάδια της τυπικής ροής σχεδίασης στα FPGAs, δηλαδή τη σύνθεση, την υλοποίηση, τη δημιουργία του bit-stream, και τον προγραμματισμό. Επί του παρόντος, η μεθοδολογία εκτελείται χειροκίνητα από τον προγραμματιστή, ωστόσο, ορισμένα τμήματα, όπως η εξερεύνηση των ρυθμίσεων του εργαλείου θα μπορούσαν να εκτελεστούν με αυτοματοποιημένο τρόπο.

Από την άποψη της αξιολόγησης του εργαλείου ανάπτυξης και προγραμματισμού, η μεθοδολογία που σχετίζεται με τη σύνθεση στοχεύει:

(i) στον έλεγχο της σωστής λειτουργικότητας όλων των ρυθμίσεων και στρατηγικών του εργαλείου, εξετάζοντας εάν αυτά εφαρμόζουν την αναμενόμενη λειτουργία και εάν τα αποτελέσματα εξόδου της προσομοίωσης είναι τα ορθά.

(ii) στην αξιολόγηση της ποιότητας των αποτελεσμάτων για διαφορετικές διαμορφώσεις του εργαλείου, συγκρίνοντας τους πόρους του FPGA που αυτές χρησιμοποιούν.

(iii) στην αξιολόγηση της ικανότητας της σύνθεσης να αντιστοιχίζει αποτελεσματικά τα κυκλώματα στα μπλοκ του FPGA, εξετάζοντας τη χρήση των πόρων σε σύγκριση με την αναμενόμενη (θεωρητική).

(iv) στη βαθμολόγηση της χρήσης των πόρων μέσω συστηματικών συγκρίσεων με συσκευές/εργαλεία κατασκευαστών FPGAs που είναι ήδη καθιερωμένοι στην αγορά.

Από την άποψη της δημιουργίας αποδοτικών DSP επιταχυντών, δεδομένου ενός πυρήνα που έχει αναπτυχθεί με HDL, η μεθοδολογία μας στοχεύει:

(i) στον προσδιορισμό της αναμενόμενης χρήσης πόρων με βάση καθιερωμένους κατασκευαστές της αγοράς.

(ii) στην εξαγωγή των ρυθμίσεων του εργαλείου που οδηγούν στις πιο αποτελεσματικές παραλλαγές υλοποίησης.

(iii) στην επίλυση ζητημάτων που προκύπτουν από την "ανωριμότητα" του εργαλείου (επειδή κυκλοφόρησε πρόσφατα) ή λόγω του ότι ο πυρήνας αναπτύχθηκε αρχικά σε τεχνολογία άλλου κατασκευαστή FPGA.

Η προτεινόμενη μεθοδολογία για τη σύνθεση απεικονίζεται στο Σχήμα 15. Χωρίζεται σε τρεις κύριες φάσεις: τη διαμόρφωση των αλγοριθμικών παραμέτρων του πυρήνα DSP, την εξερεύνηση σε επίπεδο εργαλείου και την εξερεύνηση σε επίπεδο κώδικα/HDL. Η
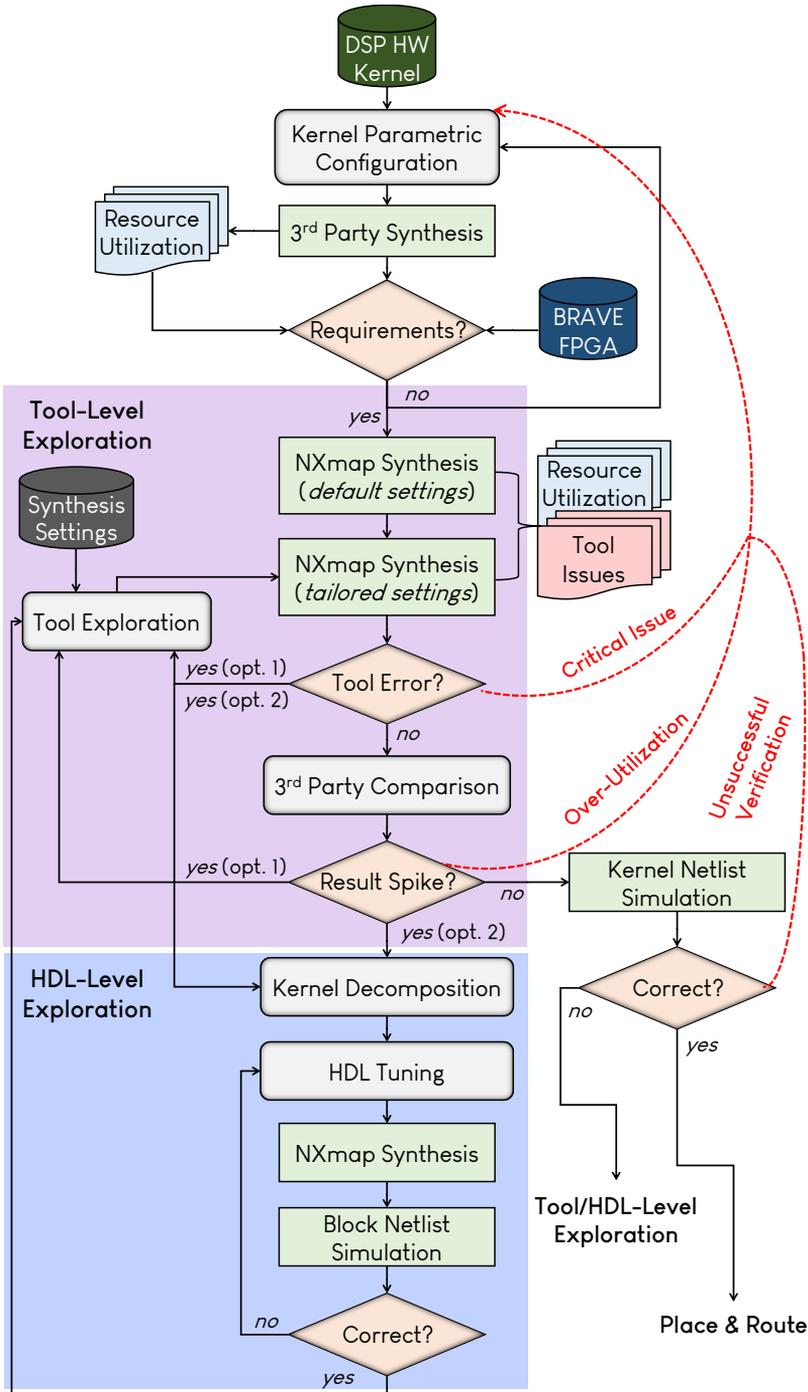
Σχήμα 15: Μεθοδολογία για τη σύνθεση πυρήνων DSP στα νέα BRAVE FPGAs.

ανάλυση της μεθοδολογίας παρατίθεται στη δημοσίευση [319], όπου επίσης απεικονίζο-
νται και αναλύονται μεθοδολογίες για τα στάδια της υλοποίησης, της παραγωγής του
bitstream και του προγραμματισμού.

Υλοποίηση CV Πυρήνων

Με βάση τη μεθοδολογία μας, επιταχύνουμε CV πυρήνες από προηγούμενες δραστηριό-
τητες της ESA [326–329] στο NG-Large FPGA. Συγκεκριμένα, χρησιμοποιούμε πυ-
ρήνες για εξαγωγή χαρακτηριστικών (ακμές και γωνίες εικόνας), δηλαδή τον Canny
Edge Detector [326] και τον Harris Corner Detector [327], καθώς και πυρήνες για ε-
ξαγωγή βάθους στην ανακατασκευή 3D σκηνής, δηλαδή τον GAD-Disparity [328] και
τον Spacesweep [329]. Αυτοί οι πυρήνες επιβάλλουν αυξημένες απαιτήσεις σε υπολογι-
σμούς και πόρους μνήμης, πιέζοντας τα νέα εργαλεία ως προς την αποτελεσματική και
υψηλής απόδοσης υλοποίηση. Οι επιλεγμένες αλγοριθμικές παράμετροι των πυρήνων
παρουσιάζονται στον Πίνακα 8.

Για την υλοποίηση τους στο BRAVE NG-Large, προέκυψαν αρκετά ζητήματα. Ένα
από αυτά ήταν η περιγραφή HDL των μονάδων μνήμης με δύο θύρες. Οι RAMBs του
NG-Large δεν είναι “ανάγνωσης-πρώτα μνήμες”, επομένως, έπρεπε να προσθέσουμε κα-
ταχωρητές για τη διοχέτευση των δεδομένων και των διευθύνσεων εγγραφής και επίσης
να εκτελέσουμε τη λειτουργία εγγραφής στην αρνητική ακμή του ρολογιού. Αυτό το
ζήτημα δεν αναφέρθηκε από το εργαλείο NXmap, ωστόσο, μετά από έρευνα με βάση τη
μεθοδολογία μας, καταφέραμε να απομονώσουμε τις σχετικές μονάδες και να εντοπίσου-
με το πρόβλημα μνήμης μέσω προσομοίωσης μετά τη σύνθεση. Ένα άλλο περιστατικό
που συνέβη αφορούσε την αναγνώριση των μνημών ROM. Το εργαλείο υλοποιούσε
πάντα τους πίνακες ROM (που δηλώνονται ως `constant` στη VHDL) στα LUTs. Επο-
μένως, χρησιμοποιήσαμε τη δήλωση `signal` στην περιγραφή των ROM μας. Αυτό το
ζήτημα δεν ήταν λειτουργικό, ωστόσο, δεν μας επέτρεψε να παραδώσουμε τις επιθυμητές
βελτιστοποιήσεις πόρων, ενώ ήταν δύσκολο να εντοπιστεί. Επιπλέον, σημειώνουμε ότι
το εργαλείο δεν μπορούσε να εφαρμόσει πολύ μεγάλους πολλαπλασιασμούς στα DSPs
μέσω σπασιμάτων, επομένως, σε ορισμένες περιπτώσεις, έπρεπε να τροποποιήσουμε τον
κώδικα HDL και να χωρίσουμε τους πολλαπλασιαστές σε μικρότερους (ή εναλλακτικά
να τους αντιστοιχίσουμε στα CYs).

Όσον αφορά την αντιστοίχιση των CV πυρήνων στο υποκείμενο υλικό, παρατηρήσαμε
αρκετές αναποτελεσματικές επιλογές από το εργαλείο NXmap. Ενδεικτικά αναφέρου-
με ότι το εργαλείο δεν χρησιμοποιούσε τους εσωτερικούς καταχωρητές των DSPs και
RAMBs αλλά τα DFFs των FEs (π.χ., για την υλοποίηση των καταχωρητών πριν από
τον πολλαπλασιασμό ή μετά την έξοδο της μνήμης). Επιπλέον, το εργαλείο δεν χρησιμο-
ποιούσε όλες τις διαθέσιμες διαμορφώσεις μνήμης για την αποτελεσματική αντιστοίχιση
των μονάδων RAM στις RAMBs, με αποτέλεσμα να υπάρχει πολύ μεγάλη χρήση πόρων.

Πίνακας 8: Αλγοριθμικές παράμετροι των CV πυρήνων.

Πυρήνας	Εικόνα			Μάσκα	
	Μέγεθος	Χώρισμα	I/O Bits	Μέγεθος	Bits
Canny Edge Detector [326]	1024×1024	–	8/4	3×3	8×3
Harris Corner Detector [327]	1024×1024	1024×32	8/32	7×7	8×14
GAD-Disparity Stereo Vision [328]	1024×1024	1024×32	8/10	7×7	8×7
Spacesweep Stereo Vision [329]	1024×1024	1024×16	8/32	13×13	8×8

Τέλος, σημειώνουμε ότι το εργαλείο δεν εκμεταλλεύτηκε καθόλου τα CYs (π.χ., για τις προσθέσεις), με αποτέλεσμα την υπερβολική χρήση των DSPs και τη μειωμένη απόδοση.

Ωστόσο, σημειώνουμε ότι όλα τα λειτουργικά ζητήματα έχουν επιλυθεί και οι CV πυρήνες έχουν υλοποιηθεί με επιτυχία, ενώ όσον αφορά την αποτελεσματικότητα της αντιστοίχισης τους στο υλικό, το NXmap βελτιώνεται συνεχώς. Σε κάθε περίπτωση, αναφέραμε ενδεικτικά ζητήματα που προέκυψαν κατά την υλοποίηση για να τονίσουμε τη δυσκολία χρήσης νέων εργαλείων σχεδίασης FPGA και να αναδείξουμε τη σημασία μεθοδολογιών όπως των προτεινόμενων.

Πειραματική Αξιολόγηση

Για την πειραματική αξιολόγηση, βασιζόμαστε στις υλοποιήσεις των CV πυρήνων και εξετάζουμε τόσο το εργαλείο λογισμικού NXmap όσο και τα BRAVE FPGAs της NanoXplore. Από τα BRAVE FPGAs, εκτός από το NG-Large, επιστρατεύουμε επίσης τον προκάτοχό του NG-Medium για λόγους σύγκρισης. Όσον αφορά τα FPGAs τρίτων κατασκευαστών, χρησιμοποιούμε τα Xilinx Virtex-5QV XQR5VFX130, Microsemi RTG4 RT4G150 και Intel Cyclone III EP3CLS150. Αντίστοιχα, χρησιμοποιούμε τα εργαλεία ανάπτυξης, δηλαδή τα NanoXplore NXmap v3.5.0.4, Xilinx Vivado, Microsemi Libero και Intel Quartus.

Ο Πίνακας 9 παραθέτει την τελική κατανάλωση πόρων των CV πυρήνων στο NG-Large. Αυτά τα αποτελέσματα αφορούν προσαρμοσμένες ρυθμίσεις του εργαλείου NXmap και η σύγκριση τους σε σχέση με αυτά που προσφέρουν οι προκαθορισμένες ρυθμίσεις αναλύεται στη δημοσίευση [319]. Συνολικά, συμπεραίνουμε ότι το εργαλείο NXmap αντιστοιχίζει τους πυρήνες όπως αναμενόταν. Λαμβάνοντας υπόψη ότι αυτοί οι πυρήνες επιβάλλουν διαφορετικές απαιτήσεις μνήμης, το εργαλείο χρησιμοποιεί σωστά αρκετές από τις διαθέσιμες διαμορφώσεις των RAMBs (π.χ., 24K×2 και 12K×4). Ως αποτέλεσμα, προκύπτει λογική χρήση των RAMBs για εικόνες πλάτους 1024-pixel, που κυμαίνονται από 36% (Harris) έως 93% (Canny). Ο Canny χρησιμοποιεί σχεδόν όλους τους πόρους

Πίνακας 9: Κατανάλωση πόρων των CV πυρήνων στο NG-Large FPGA.

Πυρήνας	Προσαρμοσμένες Ρυθμίσεις Εργαλείου					
	LUT	DFF	CY	DSP	RAMB	MHz
Canny	1843 (2%)	2349 (2%)	1086 (4%)	4 (2%)	177 (93%)	38
Harris	6105 (5%)	15413 (13%)	7112 (23%)	81 (22%)	69 (36%)	40
Disparity	998 (1%)	3672 (3%)	4548 (15%)	4 (2%)	85 (45%)	50
Spacesweep	5458 (5%)	10276 (8%)	6277 (20%)	50 (14%)	79 (42%)	52

* %: χρήση του NG-Large (137K LUTs, 129K DFFs, 32K CYs, 384 DSPs, 192 RAMBs).

μνήμης, καθώς λαμβάνει ολόκληρη την εικόνα για να την επεξεργαστεί, σε αντίθεση με τους άλλους πυρήνες που επεξεργάζονται λωρίδες της εικόνας. Όσον αφορά τα αριθμη-τικά και λογικά στοιχεία των πυρήνων, το NXmap αναγνωρίζει, αναφέρει και υλοποιεί με επιτυχία όλους τους αριθμητικούς τελεστές, τις μηχανές πεπερασμένων καταστάσεων και τις λογικές συναρτήσεις. Επίσης, οι συχνότητες ρολογιού επιτυγχάνονται έπειτα από ε-ξερεύνηση όλων των σχετικών ρυθμίσεων του εργαλείου, και είναι βελτιωμένες σε σχέση με αυτές που προσφέρουν οι προεπιλεγμένες ρυθμίσεις.

Ο Πίνακας 10 αναφέρει αποτελέσματα σχετικά με την ταχύτητα επεξεργασίας (συχνότη-τα ρολογιού, χρόνος εκτέλεσης για μία εικόνα εισόδου και απόδοση). Η μετρική της απόδοσης δεν λαμβάνει υπόψη τους χρόνους εισόδου και εξόδου (μεταφορές της ει-κόνας) και αναφέρεται μόνο στην επεξεργασία, δηλαδή στη μεμονωμένη εκτέλεση του πυρήνα. Για τους πυρήνες ανίχνευσης χαρακτηριστικών, μετράμε τα Καρέ Ανά Δευ-τερόλεπτο (FPS), ενώ για τους πυρήνες εξαγωγής βάθους, χρησιμοποιούμε τα MPixel Disparities ανά Δευτερόλεπτο (MPDS), η οποία είναι μία μετρική που συνδυάζει την ανάλυση της εικόνας και την ταχύτητα επεξεργασίας. Τα αποτελέσματα δείχνουν ότι το NG-Large παρέχει επαρκή απόδοση για εικόνα 1 MPixel, κρίνοντας από τις απαιτήσεις των αντίστοιχων διαστημικών εφαρμογών. Ο χρόνος που απαιτείται για μια πλήρη ανα-κατασκευή της σκηνής χρησιμοποιώντας τους Disparity και Spacesweep θα μπορούσε να βελτιώσει την εξαγωγή βάθους στα rovers του Άρη κατά 1 τάξη μεγέθους (όσον αφορά την ανάλυση και την ταχύτητα). Σημειώνουμε ότι στις επιλεγμένες παραμέτρους των πυρήνων, ο Spacesweep εξετάζει 3× περισσότερα επίπεδα βάθους από τον Dispar-ity (300 έναντι 100), και έτσι παρέχει πολύ μεγαλύτερη ακρίβεια. Επιπλέον, δεδομένου ότι οι περισσότερες αλγοριθμικές αλυσίδες διαστημικών εφαρμογών απαιτούν 1–10 FPS, συμπεραίνουμε ότι η απόδοση των Canny και Harris, που είναι 5.3 FPS και 10 FPS, αντίστοιχα, αφήνει αρκετό χώρο για τις υπόλοιπες συναρτήσεις να τελειώσουν την ε-κτέλεση στην ώρα τους. Σχετικά με την κατανάλωση ισχύος του NG-Large, αναφέρουμε μια λεπτομερή ανάλυση στη δημοσίευση [319]. Εν συντομία, η στατική ισχύς (όταν το FPGA δεν είναι προγραμματισμένο) είναι 2W, ενώ η δυναμική ισχύς ποικίλλει, καθώς ε-ξαρτάται από τη χρήση των πόρων (π.χ., είναι ∼200mW όταν χρησιμοποιούνται 10K FEs

Πίνακας 10: Απόδοση και ισχύς των CV πυρήνων στο NG-Large FPGA.

Πυρήνας	Ρολόι (MHz)	Χρ. Εκτέλεσης (s/frame)	Απόδοση[1] (FPS) | (MPDS)	Ισχύς[2] (W)
Canny	38	0.10	10	2.3
Harris	40	0.19	5.3	2.5
Disparity	50	6.7	18	2.3
Spacesweep	52	10.8	29	2.4

[1] Χωρίς I/O: FPS για τους Harris και Canny, MPDS για τους Disparity και Spacesweep.
[2] Εκτίμηση βασισμένη στη στατική ισχύ του FPGA και τη δυναμική ισχύ των μπλοκ του.

ή 200 DSPs και ~40mW όταν χρησιμοποιούνται 150 RAMBs). Με βάση τις μετρήσεις ισχύος και την ανάλυσή μας, αναφέρουμε μια εκτίμηση για την ισχύ στον Πίνακα 10, η οποία, ωστόσο, αφορά μόνο τη χρήση των πόρων και δεν λαμβάνει υπόψη τις εισόδους/εξόδους, τις διασυνδέσεις και την εναλλαγή δραστηριότητας. Όταν εξετάζουμε αυτές τις παραμέτρους, η κατανάλωση ισχύος του NG-Large βρίσκεται σε τυπικές τιμές για φόρτους εργασίας DSP υψηλής απόδοσης.

Στη συνέχεια, συγκρίνουμε τα αποτελέσματα με εκείνα τρίτων κατασκευαστών FPGAs. Για τον Canny, το NXmap παρέχει χρήση LUTs που είναι συγκρίσιμη με εκείνη των άλλων εργαλείων (τα LUTs είναι αυξημένα μόνο κατά 6%). Ωστόσο, σημειώνουμε ότι αν λάβουμε υπόψη και τα LUTs που καταναλώνονται λόγω χρήσης CYs, τα συνολικά LUTs αυξάνονται κατά 48%. Τα DFFs αυξάνονται κατά σχεδόν 50% στο NG-Large. Αυτή η τεράστια αύξηση οφείλεται στη μη χρήση των εσωτερικών καταχωρητών των RAMBs (δεδομένου ότι η χρήση RAMBs είναι 93% και οι μνήμες μας περιλαμβάνουν καταχωρητές). Όσον αφορά τα DSPs, το NXmap χρησιμοποιεί τους ίδιους πόρους με τα εργαλεία τρίτων κατασκευαστών, ενώ είναι πιο αποτελεσματικό στην αξιοποίηση των πόρων μνήμης (λαμβάνοντας υπόψη τη διαφορά μεγέθους RAMBs μεταξύ των FPGAs). Συνολικά, θεωρούμε τους πόρους του Canny ως λογικούς, λαμβάνοντας υπόψη ότι σχεδόν γεμίζει όλες τις RAMBs, και έτσι, το εργαλείο πιέζεται ώστε να παρέχει αποτελεσματική υλοποίηση για τις υπόλοιπες μονάδες. Για τον Harris, το NXmap παρέχει καλή χρήση LUTs, δηλαδή $3.2\times$ λιγότερα έναντι της μέσης τιμής των άλλων εργαλείων. Όταν εξετάζουμε τα επιπλέον LUTs λόγω της χρήσης CYs, ο συνολικός αριθμός των LUTs αυξάνεται, αλλά εξακολουθεί να είναι συγκρίσιμος με την μέση τιμή των άλλων εργαλείων. Η χρήση των LUTs θα πρέπει να εξετάζεται μαζί με τον αριθμό των χρησιμοποιούμενων DSPs, για τα οποία το NXmap χρησιμοποιεί $1.5\times$ λιγότερα. Σχετικά με τη μνήμη, το NXmap παρέχει 56% λιγότερα RAMBs, ενώ τα συνολικά Kbits των RAMBs που καταναλώνονται είναι μικρότερα από τη μέση τιμή των άλλων εργαλείων. Παρόμοια αποτελέσματα παρατηρούνται και για τους άλλους δύο CV πυρήνες. Για τον Disparity, το NXmap παρέχει καλή χρήση LUTs, καθώς χρησιμοποιεί έναν μικρό αριθμό ακόμη και όταν λαμβάνονται υπόψη οι πόροι των CYs. Για τις μνήμες, το NXmap

είναι κάτω από την τιμή τόσο σε μπλοκ όσο και σε Kbits. Για τον Spacesweep, το NXmap παρέχει μικρή χρήση LUTs, η οποία είναι καλύτερη κατά 52% σε σύγκριση με τα άλλα εργαλεία και σχεδόν η ίδια όταν εξετάζουμε και τα CYs. Η χρήση των DFFs είναι επίσης καλύτερη κατά 5%, ενώ η χρήση DSPs και RAMBs είναι εξαιρετική, καθώς είναι μικρότερη των μέσων τιμών των άλλων FPGAs κατά 20% και 30%, αντίστοιχα.

Τέλος, εξετάζουμε τη βελτίωση του χρόνου εκτέλεσης στο NG-Large σε σύγκριση με το NG-Medium. Και τα δύο FPGAs υλοποιούν τους ίδιους αλγορίθμους των πυρήνων, δηλαδή δεν αξιοποιείται η μεγαλύτερη χωρητικότητα του NG-Large. Ο Harris στο NG-Large είναι 4.6× ταχύτερος, καθώς η συχνότητα ρολογιού αυξάνεται κατά ~4× και το NG-Medium πρέπει να επεξεργαστεί 4× περισσότερες (μικρότερες όμως) λωρίδες εικόνας. Ο Canny επιτυγχάνει καλύτερη συχνότητα ρολογιού και χρόνο εκτέλεσης στο NG-Medium, αλλά για το 1/4 της εικόνας εισόδου του NG-Large. Για εικόνα εισόδου μεγέθους 1024×1024, το NG-Medium θα πρέπει να επεξεργαστεί κάθε 1/4 της εικόνας, να στείλει τον χάρτη ακμών πίσω στη CPU και στο τέλος, να επαναφέρει όλους τους χάρτες ακμών που έχουν διαμεριστεί στο FPGA. Αυτά τα επιπλέον βήματα θα προσθέσουν μια επιβάρυνση ~60ms στο καλό σενάριο, δηλαδή όταν χρησιμοποιείται η διεπαφή SpaceWire στα 100 Mbps για μετάδοση δεδομένων. Για αυτόν τον λόγο, η συνολική βελτίωση της απόδοσης στο NG-Large είναι 1.4×. Όσον αφορά τον Disparity, και οι δύο συσκευές επιτυγχάνουν σχεδόν την ίδια συχνότητα ρολογιού, δηλαδή περίπου 50MHz, ωστόσο, το NG-Medium πρέπει να επεξεργαστεί 2× περισσότερες (μικρότερες όμως) λωρίδες εικόνας. Αυτή η διαφοροποίηση στο χώρισμα της εικόνας εισόδου μεταφράζεται σε 1.2× βελτίωση της απόδοσης στο NG-Large. Για τον Spacesweep, λαμβάνοντας υπόψη ότι το NG-Medium πρέπει να επεξεργάζεται μικρότερη εικόνα εισόδου για να αποφευχθεί η υπερκατανάλωση πόρων (και με μειωμένη συχνότητα ρολογιού κατά 1.7×), το NG-Large προσφέρει περίπου 1.7× καλύτερη απόδοση. Σε κάθε περίπτωση, σημειώνουμε ξανά ότι εφαρμόζουμε τους ίδιους αλγόριθμους και στις δύο συσκευές και η απόδοση στο NG-Large μπορεί να βελτιωθεί σημαντικά με την εκμετάλλευση της μεγαλύτερης χωρητικότητάς του και την αύξηση της παραλληλοποίησης (π.χ., μέσω της υλοποίησης παράλληλων αριθμητικών τελεστών ή της παράλληλης επεξεργασίας λωρίδων εικόνας).

9. Επιτάχυνση DSP & AI σε Ετερογενή Πολυπύρηνα SoCs

*Η ενότητα αυτή αφορά το **Κεφάλαιο 9** και βασίζεται στις **δημοσιεύσεις [359–362]**.*

Εισαγωγή

Η τελευταία δεκαετία χαρακτηρίζεται από μια ταχεία ανάπτυξη ισχυρών AI και πολύπλοκων DSP αλγορίθμων. Αυτή η αλγοριθμική εξέλιξη μαζί με τη μεγάλη αύξηση των δεδομένων των αισθητήρων έχουν επηρεάσει σημαντικά τα υπολογιστικά συστήματα στην άκρη του δικτύου. Η παγκόσμια ζήτηση για ταχύτητα αμφισβητεί την χρήση AI/DSP λειτουργιών σε νέες εφαρμογές, ειδικά στα ενσωματωμένα συστήματα περιορισμένης ισχύος. Ετερογενή Συστήματα-σε-Τσιπ (SoCs) όπως οι VPUs εμφανίζονται ως μια πολλά υποσχόμενη λύση [25], προσφέροντας αυξημένη ευελιξία προγραμματισμού και ποικιλομορφία όσον αφορά την επεξεργασία και την αποθήκευση. Γενικά, οι VPUs προσφέρουν μια ποικιλία επεξεργαστών, όπως επεξεργαστές γενικού σκοπού, φίλτρα υλικού, διανυσματικούς πυρήνες και επιταχυντές νευρωνικών δικτύων. Χρησιμοποιούνται σε εφαρμογές υπολογιστικής όρασης χαμηλής κατανάλωσης, καλύπτοντας τομείς όπως η ρομποτική, η αυτοκινητοβιομηχανία και το διάστημα. Το κύριο χαρακτηριστικό τους είναι ότι προσφέρουν σημαντικά χαμηλότερη κατανάλωση από τις υπόλοιπες ανταγωνιστικές πλατφόρμες, όπως είναι τα FPGAs και οι GPUs.

Η Intel κατασκευάζει τις Myriad VPUs (τη Myriad 2 στα 28nm και τη Myriad X 16nm) [364], οι οποίες αποτελούν εξέχοντες και καθιερωμένους επεξεργαστές για ενσωματωμένες εφαρμογές DSP και AI. Προσφέροντας πολυάριθμους ετερογενείς πυρήνες επεξεργασίας, αυτές οι VPUs επιτρέπουν την αποτελεσματική παραλληλοποίηση απαιτητικών αλγορίθμων. Ένα μόνο τσιπ VPU ενσωματώνει πολλαπλά περιφερειακά υλικού, μια ποικιλία από φίλτρα χαμηλής ισχύος, επεξεργαστές γενικού σκοπού, διανυσματικούς πυρήνες επιτάχυνσης και μηχανές επιτάχυνσης AI. Οι κύριες μονάδες επεξεργασίες των Myriad VPUs είναι οι SHAVEs, οι οποίοι αποτελούν VLIW & SIMD διανυσματικούς επεξεργαστές. Παρόμοια ετερογένεια παρέχεται και στην αποθήκευση δεδομένων, για την οποία τα VPU SoCs προσφέρουν μια ιεραρχία μνήμης που αποτελείται από μνήμες DRAM, scratchpad και cache. Είναι προφανές ότι η πλήρης αξιοποίηση και εκμετάλλευση τέτοιων πολύπλοκων ετερογενών συστημάτων απαιτεί μια σχολαστική και συστηματική προσέγγιση της ανάπτυξης. Επιπλέον, λαμβάνοντας υπόψη ότι αυτά τα SoCs είναι κατασκευασμένα με γνώμονα περισσότερο τη χαμηλή ισχύ παρά την υψηλή απόδοση, η ανάγκη για αποτελεσματικό προγραμματισμό γίνεται ακόμα πιο κρίσιμη. Η αρχιτεκτονική και τα εργαλεία προγραμματισμού των Myriad VPUs παρουσιάζονται αναλυτικά στη δημοσίευση [359].

Τα πολλά υποσχόμενα χαρακτηριστικά των Myriad VPUs έχουν προσελκύσει το ενδιαφέρον της ESA, η οποία συμμετέχει πλήρως στο σαφάρι της εύρεσης νέων ενσωματωμένων επεξεργαστών. Για τον σκοπό αυτό, υποστηρίζει ερευνητικές δραστηριότητες (LEOTOME, HPCB) που αξιολογούν τη συνολική απόδοση των Myriad SoCs. Αυτές οι δραστηριότητες στοχεύουν στην αξιολόγηση της καταλληλότητας των VPUs κυρίως για επιτάχυνση DSP/AI. Στο πλαίσιο αυτών των δραστηριοτήτων, προτείνουμε μια μεθοδολογία για την υποστήριξη της ανάπτυξης και την επιτάχυνση απαιτητικού φόρτου

DSP/AI εργασιών. Στόχος μας είναι να ξεπεράσουμε τα ζητήματα του ενσωματωμένου υπολογισμού λόγω των περιορισμένων πόρων, να ξεκλειδώσουμε το πλήρες δυναμικό της ετερογένειας των VPUs και έτσι, να υλοποιήσουμε αποτελεσματικά πολύπλοκους αλγόριθμους. Ο τομέας εφαρμογών μας είναι το διάστημα, ωστόσο, τόσο η μεθοδολογία όσο και οι τεχνικές ανάπτυξης είναι γενικές, δηλαδή μπορούν να χρησιμοποιηθούν ως πρότυπο σχεδίασης για την παροχή επιτάχυνσης DSP/AI στις ετερογενείς VPUs. Σχετικά με τον προγραμματισμό, εφαρμόζουμε διάφορες τεχνικές παραλληλοποίησης υψηλού επιπέδου, ενώ σε χαμηλότερο επίπεδο βελτιστοποιούμε τις μνήμες και τους πυρήνες ε-πιτάχυνσης. Σχετικά με τους αλγόριθμους, υλοποιούμε προσαρμοσμένους DSP και AI πυρήνες στη Myriad 2, επιταχύνουμε μια εξελιγμένη αλγοριθμική CV αλυσίδα 5 στα-δίων [378] στη Myriad 2, και επιταχύνουμε ένα απαιτητικό CNN [379] στον NCS2 USB επιταχυντή που βασίζεται στη Myriad X. Τόσο το CV όσο και το CNN, παρακολουθούν την πόζα (στάση) ενός δορυφόρου στο διάστημα.

Μεθοδολογία Σχεδίασης για τις Ετερογενείς Myriad VPUs

Η αποτελεσματική χρήση των πολύ ετερογενών Myriad 2 και Myriad X απαιτεί μια μεθο-δική σχεδιαστική προσέγγιση. Για να αξιοποιηθεί πλήρως το δυναμικό της ετερογένειας, οι ποικίλες αλγοριθμικές συναρτήσεις της εφαρμογής θα πρέπει να αντιστοιχιστούν στις καταλληλότερες μονάδες επεξεργασίας, ενώ η ανάπτυξη τους θα πρέπει να προσαρμοστεί στο υποκείμενο υλικό. Προς αυτή την κατεύθυνση, προτείνουμε μια μεθοδολογία σχε-δίασης για τον αποτελεσματικό προγραμματισμό οποιασδήποτε εφαρμογής DSP/AI που αποτελείται από πολλαπλές αλγοριθμικές συναρτήσεις. Χωρίζουμε τη μεθοδολογία μας σε δύο κλάδους. Ο πρώτος κλάδος αφορά την υλοποίηση ενός αλγορίθμου DSP, όπως η αλγοριθμική CV αλυσίδα του [378]. Ο δεύτερος κλάδος αφορά την επιτάχυνση ενός δικτύου AI, όπως το CNN του [379]. Η DSP μεθοδολογία στοχεύει στα Myriad SoCs, ενώ η AI μεθοδολογία αναφέρεται στον NCS2 USB επιταχυντή, ο οποίος βασίζεται στην Myriad X και την AI μηχανή της.

Για την προγραμματισμό μίας DSP εφαρμογής, ξεκινάμε αναπτύσσοντας από την αρχή όλες τις αλγοριθμικές συναρτήσεις ή μεταφέροντας τον κώδικα C/C++ τους (που α-ναπτύχθηκε σε άλλες πλατφόρμες) στον επεξεργαστή γενικού σκοπού LEON. Μετά την επιτυχή μεταγλώττιση και εκτέλεση όλων των συναρτήσεων, αναλύουμε τον χρόνο εκτέλεσής τους, τις απαιτήσεις σε μνήμη, I/O και αριθμητική, ακόμη και την πολυπλο-κότητα προγραμματισμού. Με βάση αυτήν την ανάλυση, και σύμφωνα με τον σχεδιαστικό μας στόχο (χαμηλή κατανάλωση ή υψηλή απόδοση), διαμερίζουμε ολόκληρο τον αλγόριθ-μο στο SoC, δηλαδή καθορίζουμε το που θα εκτελεστεί κάθε αλγοριθμική συνάρτηση. Επιπλέον, βάσει της ανάλυσής μας καθώς και της μικρο-αρχιτεκτονικής του SoC και των βιβλιοθηκών που παρέχονται από τον προμηθευτή, αναπτύσσουμε βοηθητικό λογι-σμικό για εργασίες όπως ο χειρισμός των εισόδων/εξόδων, η διαχείριση της μνήμης και η παραλληλοποίηση του φόρτου εργασίας.

Στη συνέχεια, ξεκινάμε την υλοποίηση κάθε συνάρτησης τόσο σε υψηλό όσο και σε χαμηλό επίπεδο. Σε υψηλό επίπεδο, εφαρμόζουμε το πιο αποτελεσματικό σχήμα παραλληλοποίησης και αντιστοίχισης στους επεξεργαστές του SoC, και οργανώνουμε το λογισμικό όσον αφορά τον χρόνο και την κατανομή μνήμης. Σε χαμηλό επίπεδο, επιταχύνουμε τις συναρτήσεις αναδιατάσσοντας τις λειτουργίες τους για να διευκολύνουμε τόσο την εκτέλεση όλης της αλγοριθμικής αλυσίδας όσο και την επαναχρησιμοποίηση της μνήμης. Σε αυτό το πλαίσιο, εφαρμόζουμε βελτιστοποίηση του μήκους των λέξεων για περαιτέρω ελαχιστοποίηση της προσωρινής αποθήκευσης, και πραγματοποιούμε παραλληλοποίηση μέσω τεχνικών διανυσματοποίησης ή/και αποσύνθεσης δεδομένων. Στο επόμενο βήμα, ενσωματώνουμε όλες τις συναρτήσεις στο σύστημα και εξερευνούμε όλες τις παραμέτρους κωδικοποίησης για να βελτιστοποιήσουμε την υλοποίηση για το δεδομένο πρόβλημα. Σε περίπτωση που οι περιορισμοί της σχεδίασης δεν ικανοποιούνται, χρησιμοποιούμε έναν βρόχο ανατροφοδότησης για να επιστρέψουμε είτε στην αρχική κατάτμηση και στον προγραμματισμό ολόκληρου του αλγορίθμου είτε στην υλοποίηση υψηλού και χαμηλού επιπέδου των συναρτήσεων.

Όλα τα παραπάνω βήματα της μεθοδολογίας ανάπτυξης DSP συνοψίζονται ως εξής:

1. Ανάπτυξη/Μεταφορά της εφαρμογής/αλγόριθμου στον επεξεργαστή γενικού σκοπού LEON.

2. Ανάλυση κάθε αλγοριθμικής συνάρτησης για ένα ρεαλιστικό σύνολο δεδομένων.

3. Κατάτμηση και προγραμματισμός ολόκληρου του αλγορίθμου στο Myriad SoC.

4. Ανάπτυξη βοηθητικού λογισμικού για τον χειρισμό των εισόδων/εξόδων, τη διαχείριση μνήμης και τις βελτιστοποιήσεις χαμηλού επιπέδου.

5. Παραλληλοποίηση υψηλού επιπέδου κάθε αλγοριθμικής συνάρτησης στους πυρήνες επιτάχυνσης SHAVEs.

6. Υλοποίηση και βελτιστοποίηση χαμηλού επιπέδου στους SHAVEs.

7. Ενσωμάτωση όλων των αλγοριθμικών συναρτήσεων στο σύστημα.

8. Δοκιμή και συντονισμός με σύνολα δεδομένων για τη συγκεκριμένη εφαρμογή.

9. Επιστροφή στο βήμα 3 ή 5 μέχρι να ικανοποιηθούν όλοι οι περιορισμοί και να βελτιστοποιηθεί το σύστημα.

Αντίστοιχη μεθοδολογία προτείνουμε και για την εκτέλεση CNNs μέσω του εργαλείου OpenVINO της Intel. Τα βήματα συνοψίζονται ως εξής:

1. Δημιουργία του μοντέλου του CNN σε κάποιο framework ανάπτυξης AI (π.χ., TensorFlow).

2. Βελτιστοποίηση και ανάπτυξη προσαρμοσμένων λειτουργιών του δικτύου στο OpenVINO.

3. Δημιουργία του αρχείου προγραμματισμού του δικτύου στο OpenVINO.

4. Εκτέλεση του δικτύου στην Myriad X και ανάλυση των αποτελεσμάτων.

Πίνακας 11: Επισκόπηση των VPU επιταχυντών της Διατριβής.

Συνάρτηση	Δεδομένα Εισόδου	Δεδομένα Εξόδου	VPU
Averaging Binning	4-MPixel, 8 bits	1-MPixel, 8 bits	Myriad 2
Floating-Point Convolution	1-MPixel, 8 bits	1-MPixel, 8 bits	Myriad 2
ShipDetect CNN	1-MPixel×3, 16 bits	64×1, 32 bits	Myriad 2
Canny Edge Detection	1-MPixel, 8/16 bits	1K–5K, 5 bits	Myriad 2
Depth Rendering	6×1, 32 bits	1-MPixel, 16 bits	Myriad 2
Edge Matching	1-MPixel×2, 8&16 bits	1K–5K, 8 bits	Myriad 2
UrsoNet CNN	1-MPixel×3, 8 bits	4099×1, 32 bits	Myriad X

5. Επιστροφή στο βήμα 1 ή 2 μέχρι να ικανοποιηθούν όλοι οι περιορισμοί και να βελτιστοποιηθεί η εκτέλεση.

Υλοποίηση CV Αλγοριθμικής Αλυσίδας

Με βάση τη μεθοδολογία μας, υλοποιούμε διάφορους DSP πυρήνες, έναν CV αλγόριθμο και CNNs, τα οποία συνοψίζονται στον Πίνακα 11. Ενδεικτικά, αναφέρουμε κάποιες λεπτομέρειες της υλοποίησης για τον CV αλγόριθμο του [378], ο οποίος παρακολουθεί την πόζα (στάση) του δορυφόρου Envisat σε πραγματικό χρόνο. Συγκεκριμένα, εισάγει μια ακολουθία εικόνων υψηλής ευκρίνειας και εκτελεί συνεχώς ρεαλιστική απόδοση του μοντέλου, ανίχνευση χαρακτηριστικών, αντιστοίχιση χαρακτηριστικών και προσαρμογή δεδομένων μέσω ισχυρής παλινδρόμησης. Η έξοδος είναι η 6D στάση του δορυφόρου κατά τη διάρκεια ενός υποθετικού ελιγμού σε μία αποστολή απομάκρυνσης απορριμμάτων [326]. Αυτός ο εξελιγμένος αλγόριθμος, ο οποίος περιλαμβάνει συναρτήσεις όπως η Canny Edge Detection, η Depth Rendering και η Edge Matching, παρουσιάζεται αναλυτικά στη δημοσίευση [359].

Η ανάλυση του CV αλγορίθμου με βάση τη μεθοδολογία μας καταλήγει στον διαχωρισμό και τον προγραμματισμό των επιμέρους συναρτήσεων που φαίνεται στο Σχήμα 16. Ο αυξημένος χρόνος εκτέλεσης στον επεξεργαστή γενικού σκοπού LEON των Canny Edge Detection και Depth Rendering δείχνει ότι θα πρέπει να εντοπίσουμε όλες τις ευκαιρίες παραλληλοποίησης και να επιταχύνουμε τις συναρτήσεις αυτές στους SHAVEs. Παρόλο που η συνάρτηση Edge Matching δεν περιλαμβάνει απαιτητικούς υπολογισμούς (εφαρμόζει μόνο σάρωση εικόνων και συγκρίσεις), την υλοποιούμε επίσης στους SHAVEs. Διαφορετικά, ο LEON θα έπρεπε να σαρώσει μια εικόνα 1024×1024, με αποτέλεσμα να αυξηθεί ο χρόνος εκτέλεσης. Όσον αφορά τη συνάρτηση Pose Refinement, επιλέγουμε να εκτελεστεί στον LEON λόγω των εξαρτήσεών της από τις βιβλιοθήκες BLAS/LAPACK. Σχετικά με τον προγραμματισμό των επιμέρους συναρτήσεων που παρουσιάζεται στο Σχήμα 16, αρχίζουμε να ανιχνεύουμε ακμές σε μια νέα εικόνα ακόμα κι όταν ο LEON
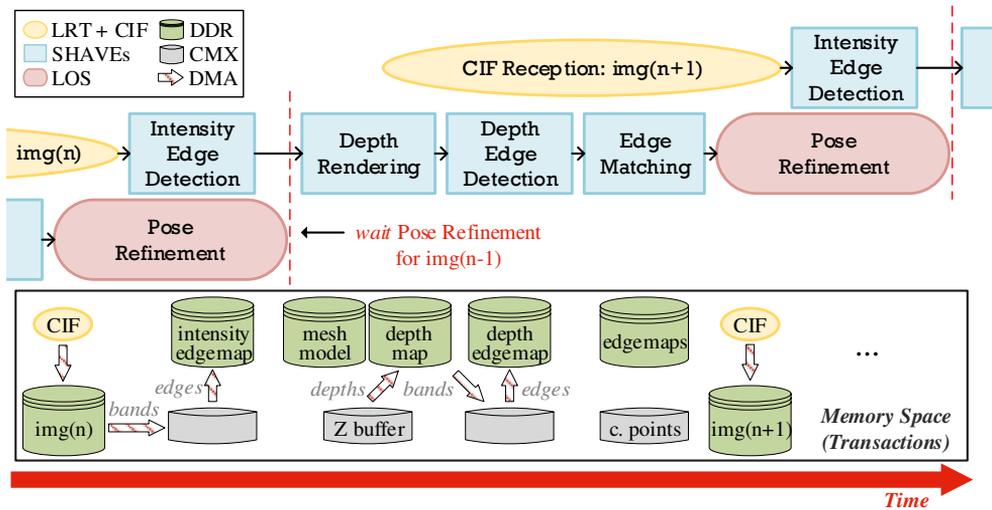
Σχήμα 16: Υλοποίηση του CV αλγορίθμου 5 σταδίων στη Myriad 2.

επεξεργάζεται το προηγούμενο καρέ για να εμφανίσει την πόζα του δορυφόρου. Όταν τελειώσουν και οι δύο αυτές εργασίες, ξεκινάει η εκτέλεση της Depth Rendering για την τρέχουσα εικόνα. Σημειώνουμε ότι η Myriad 2 λαμβάνει τις εικόνες εισόδου από το περιφερειακό Camera Interface (CIF). Η λήψη κάθε νέας εικόνας πραγματοποιείται παράλληλα με την κύρια επεξεργασία.

Για την παραλληλοποίηση των συναρτήσεων του CV αλγορίθμου στους SHAVEs, καθώς και των υπολοίπων πυρήνων επεξεργασίας εικόνας, εργαζόμαστε όπως απεικονίζεται στο Σχήμα 17. Οι εικόνες εισόδου των συναρτήσεων είναι αποθηκευμένες στη DDR DRAM μνήμη, χωρίζονται σε λωρίδες εικόνας, και μεταφέρονται στη CMX μνήμη εργασίας των SHAVEs μέσω της λειτουργίας DMA για να επεξεργαστούν. Οι επεξεργαστές LEON εκκινούν τους SHAVEs και εκτελούν όλες τις απαραίτητες εργασίες υψηλού επιπέδου σχετικά με τη λήψη των εικόνων, τον χωρισμό τους σε λωρίδες, και τις DMA μεταφορές. Όσον αφορά τις χαμηλού επιπέδου λεπτομέρειες υλοποίησης στους SHAVEs, αποφεύγουμε τη χρήση επιπλέον buffers εργασίας όσο αυτό είναι εφικτό, δηλαδή η επεξεργασία πραγματοποιείται επιτόπου στον buffer εισόδου, ενώ επίσης ρυθμίζουμε κατάλληλα τις κρυφές μνήμες. Συνοπτικά, για την ανάπτυξη εφαρμογών στις Myriad VPUs χρησιμοποιούμε τεχνικές όπως η προσαρμογή του τύπου δεδομένων των μεταβλητών, διανυσματικές SIMD εντολές για αριθμητικές πράξεις, και στατική/δυναμική δρομολόγηση των επιμέρους εργασιών. Όλες οι σχεδιαστικές λεπτομέρειες παρουσιάζονται αναλυτικά στις αντίστοιχες δημοσιεύσεις [359,360,362]. Ενδεικτικά, στο Σχήμα 18 παρουσιάζουμε κάποιες απο τις τεχνικές που χρησιμοποιήθηκαν.
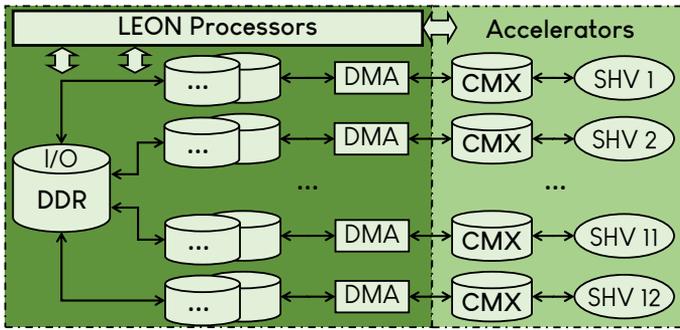
Σχήμα 17: Παραλληλοποίηση των πυρήνων επεξεργασίας εικόνας στη Myriad 2 VPU.
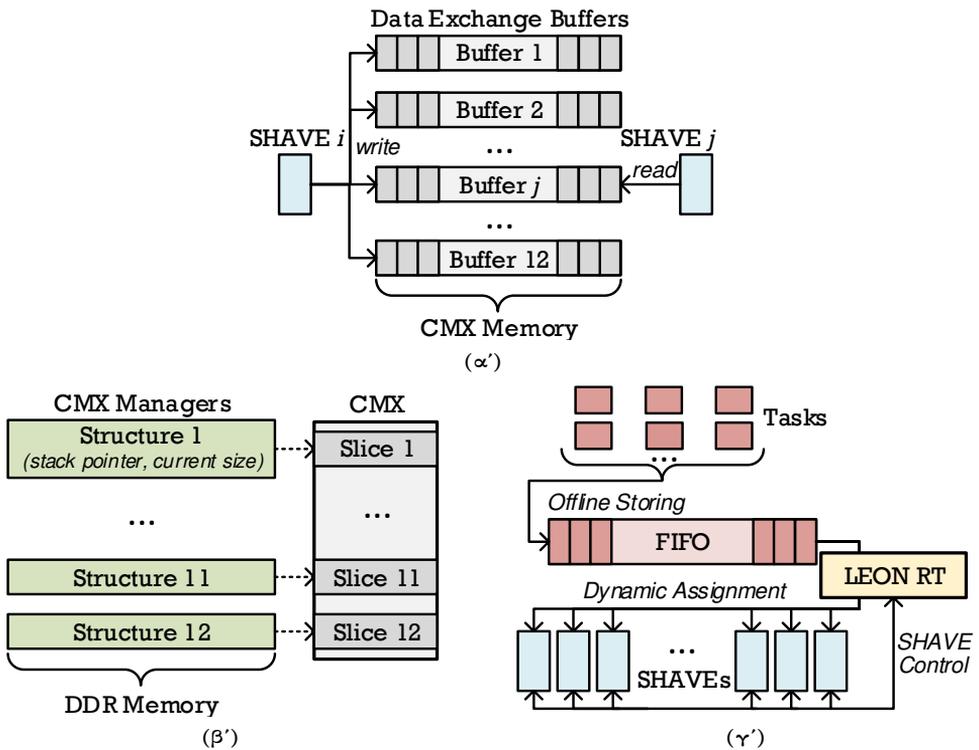
Σχήμα 18: Υλοποιημένες τεχνικές στη Myriad 2 για: **(α′)** επικοινωνία επιταχυντών, **(β′)** διαχείριση μνήμης, και **(γ′)** δυναμική ανάθεση εργασιών.

Πειραματική Αξιολόγηση

Σε αυτή την ενότητα παρουσιάζουμε τα πιο σημαντικά πειραματικά αποτελέσματα στις Myriad VPUs. Αρχικά, αξιολογούμε την υλοποίηση των DSP και CNN πυρήνων θεω-

ρώντας μία αρχιτεκτονική που αποτελείται από ένα FPGA, τη Myriad 2 VPU, και ένα Host-PC. Το Host-PC φορτώνει τα δεδομένα εισόδου στο FPGA, το οποίο ανταλλάσσει δεδομένα με τη Myriad 2 μέσω των περιφερειακών CIF και LCD. Η Myriad 2 πραγματοποιεί την επεξεργασία και επιστρέφει τα αποτελέσματα στο FPGA. Περισσότερες λεπτομέρειες για την αρχιτεκτονική και την επικοινωνία των δύο συσκευών υπάρχουν στη δημοσίευση [276]. Τα πειραματικά αποτελέσματα δείχνουν καλούς χρόνους μεταφοράς δεδομένων (π.χ., 21ms για μια 1-MPixel εικόνα), καθώς και αποδεκτούς χρόνους συστήματος (μεταφορά + επεξεργασία) λαμβάνοντας υπόψη άλλες συσκευές και αρχιτεκτονικές.

Αντίστοιχα, αξιολογούμε την υλοποίηση του CV αλγορίθμου στη Myriad 2, χρησιμοποιώντας μία παρόμοια αρχιτεκτονική, η όποια περιλαμβάνει επίσης και το διαστημικού βαθμού δίκτυο επικοινωνίας SpaceWire για τη μεταφορά δεδομένων. Περισσότερες λεπτομέρειες για αυτή την αρχιτεκτονική παραθέτονται στη δημοσίευση [359]. Αξιολογούμε το σύστημά μας με ένα συνθετικό σύνολο δεδομένων που περιλαμβάνει δύο ακολουθίες από 1000 1-MPixel εικόνες, οι οποίες προσομοιώνουν ρεαλιστικά την κίνηση του Envisat. Συγκεκριμένα, η πρώτη ακολουθία καρέ απεικονίζει μια περιστροφή του δορυφόρου με απόσταση ~50m από την κάμερα, ενώ η δεύτερη απεικονίζει μια κίνηση από τα 30m έως τα 20m μακριά από την κάμερα. Ο Πίνακας 12 αναφέρει τα κύρια πειραματικά αποτελέσματα της υλοποίησης του CV αλγορίθμου για τη δεύτερη ακολουθία δεδομένων εισόδου.

Οι χρόνοι εκτέλεσης ποικίλλουν κατά τη διάρκεια της ακολουθίας δοκιμής, επειδή ο αλγοριθμικός φόρτος εργασίας εξαρτάται από το περιεχόμενο, δηλαδή επηρεάζεται από το μέγεθος του Envisat στην εικόνα. Στη δημοσίευση [359], παρουσιάζουμε αναλυτικά το πως επιδρά η κάθε τεχνική μας στην βελτίωση της επιτάχυνσης κάθε συνάρτησης του CV αλγορίθμου. Η λήψη της εικόνας μέσω του περιφερειακού CIF, το οποίο λειτουργεί στα 5MHz, απαιτεί ~210ms, ωστόσο, στην τελική μας υλοποίηση καλύπτεται εξ' ολοκλήρου από την επεξεργασία του προηγούμενου καρέ (βλέπε Σχήμα 16). Ως εκ τούτου, η επιτάχυνση του συστήματος αυξάνεται σε 8.5×–12× (ο Πίνακας 12 δεν λαμβάνει υπόψη τα I/Os). Συνολικά, η απόδοση όλου του αλγορίθμου είναι 2.6–3.8 FPS για την ακολουθία 20m–30m (325ms ανά καρέ κατά μέσο όρο), ενώ για την ακολουθία 50m, επιτυγχάνουμε απόδοση 3.8–4.9 FPS (235ms ανά καρέ κατά μέσο όρο). Τέλος, στη δημοσίευση [359] παραθέτουμε συγκρίσεις με άλλους επεξεργαστές, από τις οποίες εξάγεται το εξής συμπέρασμα: η Myriad 2 VPU έχει σημαντικά χαμηλότερη κατανάλωση ισχύος από τα FPGAs και τις GPUs, αυξημένη επιτάχυνση σε σχέση με τις CPUs, και περίπου ίδια ή καλύτερη απόδοση-ανά-Watt από τα FPGAs και τις GPUs. Ενδεικτικά, για την τελευταία μετρική, η Myriad 2 ανταλλάσσει περίπου 3× σε ταχύτητα για 4× σε μέση ισχύ σε σύγκριση με το Zynq FPGA.

Ο Πίνακας 13 παρουσιάζει τα αποτελέσματα της εκτέλεσης του UrsoNet CNN σε VPU, CPU και GPU. Για αυτό το πείραμα, χρησιμοποιούμε εικόνες εισόδου που έχουν κλιμα-

Πίνακας 12: Πειραματικά αποτελέσματα του CV αλγορίθμου στη Myriad 2 VPU.

Συνάρτηση	LEON		SHAVEs			
	Χρ. Εκτέλεσης (ms)	Μνήμη (MB)	Χρ. Εκτέλεσης (ms)	Μνήμη (MB)	Ισχύς (mW)	Ακρίβεια (%)
I. Edge Detect.	370–375	6.1	36–37	2.3	771	99
D. Edge Detect.	375–380	7.1	39–40	3.8	771	99
Depth Rend.	1900–2100	6.4	119–212	3.3	980	100
Edge Matching	100–120	3	5–6	3	810	100
Pose Refinem.	100–130	6.3	100–130[1]	6.3	644	[359]
Αλγόριθμος	2845–3105	–	263–388[2]	–	898	[359]

[1] Η Pose Refinement δεν υλοποιήθηκε στους SHAVEs.

[2] Η Intensity Edge Detection δεν προστίθεται στη συνολική καθυστέρηση επειδή καλύπτεται χρονικά από την Pose Refinement.

Πίνακας 13: Πειραματικά αποτελέσματα για το UrsoNet CNN σε ενσωματωμένες συσκευές (εικόνα με μέγεθος 512×640×3).

Συσκευή	Χρ. Εκτέλεσης (ms)	Απόδοση (FPS)	Ισχύς (W)	FPS-ανά-Watt
Myriad X VPU (NCS2 + Pi 3)[1]	588	1.7	5	0.34
Cortex-A57 CPU (Jetson Nano)[2]	2830	0.4	10	0.04
	7519	0.1	5	0.02
Maxwell GPU (Jetson Nano)[3]	761	1.3	10	0.13
	958	1	5	0.2

[1] NCE & 16-core SHAVE @700MHz. Το NCS2 (2W) βρίσκεται πάνω σε ένα Raspberry Pi 3 (3W).

[2] 10W λειτουργία: 4-core @1.4GHz. 5W λειτουργία: 2-core @918MHz.

[3] 10W λειτουργία: 128-core @921MHz. 5W λειτουργία: 128-core @614MHz.

κωθεί σε 512×640×3, ώστε να πετύχουμε καλύτερους χρόνους για αυτό το απαιτητικό δίκτυο. Η πειραματική αξιολόγηση της υλοποίησης των αλγορίθμων δειγματοληψίας της εικόνας στη Myriad X VPU και οι επιπτώσεις τους στην ακρίβεια συμπεριλαμβάνονται στη δημοσίευση [362]. Η CPU είναι ο ARM Cortex-A57 της πλακέτας Jetson Nano της Nvidia, ενώ η GPU είναι η Maxwell 128 πυρήνων της ίδιας πλακέτας. Για το Jetson Nano, λαμβάνουμε υπόψη τις δύο λειτουργίες του: την "χαμηλής-κατανάλωσης" στα 5W και την "υψηλής-απόδοσης" στα 10W. Για να κάνουμε μια δίκαιη σύγκριση, θεωρούμε ότι η NCS2 USB VPU βρίσκεται πάνω σε μία πλακέτα, αυτή του Raspberry Pi 3, και επομένως, η συνολική κατανάλωση ισχύος είναι 5W. Όσον αφορά τον χρόνο εκτέλεσης, ο 4-πύρηνος Cortex-A57 που λειτουργεί στα 1.4GHz είναι σχετικά αργός, δηλαδή, ο NCS2 πετυχαίνει επιτάχυνση 6×. Σε σύγκριση με τη GPU, ο NCS2 προσφέρει μια μικρή βελτίωση 1.3×, αλλά με τη μισή κατανάλωση ισχύος της GPU.

Λαμβάνοντας υπόψη τόσο την απόδοση όσο και την ισχύ, ο NCS2 προσφέρει ~1.7×
περισσότερα FPS-ανά-Watt από τη GPU και το ~8.5× περισσότερα FPS-ανά-Watt από
τη CPU.

10. Επίλογος

Ο στόχος του παρούσας Διδακτορικής Διατριβής ήταν να σχεδιάσει και να αξιολογήσει
επιταχυντές υλικού για DSP και AI που πληρούν τις απαιτήσεις των σύγχρονων υπολο-
γιστικών συστημάτων. Προς την επίτευξη αυτού του στόχου, υιοθετήσαμε προσεγγίσεις
σχεδίασης από διαφορετικά επίπεδα της υπολογιστικής πυραμίδας. Ξεκινώντας από το
κατώτατο επίπεδο σχεδίασης και ανεβαίνοντας προς τα πάνω, η δουλειά μας περιλάμβανε
αριθμητικά κυκλώματα, επιταχυντές υλικού, υλοποιήσεις σε FPGAs και υλοποιήσεις σε
ενσωματωμένα SoCs. Συνολικά, οι βασικές συνεισφορές της Διατριβής συνοψίζονται
ως εξής:

- Εκτενής και ενημερωμένη έρευνα στην ευρύτερη επιστημονική περιοχή του Προ-
  σεγγιστικού Υπολογισμού, η οποία εξετάζει και ταξινομεί τεχνικές προσέγγισης
  λογισμικού και υλικού (**Κεφάλαιο 2**).

- Βελτιστοποιήσεις χαμηλού επιπέδου στο εναλλακτικό αριθμητικό πρότυπο DLSB
  (**Κεφάλαιο 3**).

- Νέες τεχνικές αριθμητικής προσέγγισης, οι οποίες δημιουργούν τις οικογένειες προ-
  σεγγιστικών πολλαπλασιαστών RAD, AxFXU/AxFPU και ROUP (**Κεφάλαια
  4–6**).

- Βελτίωση (3 φορές) του συνόρου Pareto του διαγράμματος ενέργειας-λάθους των
  προσεγγιστικών πολλαπλασιαστών της βιβλιογραφίας (**Κεφάλαια 4–6**).

- Σχήμα χαμηλής επιβάρυνσης για την απρόσκοπτη διαμόρφωση της προσέγγισης των
  πολλαπλασιαστών DyFXU/DyFPU κατά τον χρόνο εκτέλεσης (**Κεφάλαιο 5**).

- Σχεδίαση βασισμένη στην "συνεργατική προσέγγιση", η οποία συνδυάζει διάφορες
  ορθογώνιες τεχνικές αριθμητικής προσέγγισης ώστε να δημιουργήσει έναν πολύ
  μεγάλο χώρο σχεδίασης/προσέγγισης (**Κεφάλαιο 6**).

- Μεθοδολογία για την ανάπτυξη προσεγγιστικών επιταχυντών υλικού για DSP και
  AI, είτε σε ASIC είτε σε FPGA, η οποία βασίζεται σε μία λεπτομερή εξερεύνηση του
  χώρου σχεδίασης με διαφορετικές προσεγγίσεις, αλγόριθμους, αριθμητικά πρότυπα
  και τεχνικές σχεδίασης υλικού (**Κεφάλαιο 7**).

- Πειραματικά αποτελέσματα για διάφορους προσεγγιστικούς επιταχυντές υλικού για DSP και AI, συμπεριλαμβανομένων πυρήνων για επεξεργασία 1D/2D σήματος και νευρωνικών δικτύων (**Κεφάλαιο 7**).

- Μεθοδολογία για την αποτελεσματική υλοποίηση και επιτάχυνση απαιτητικών αλγορίθμων DSP στα νέα BRAVE FPGAs διαστημικού βαθμού (**Κεφάλαιο 8**).

- Πειραματικά αποτελέσματα από την υλοποίηση διαφόρων DSP πυρήνων (ανίχνευση χαρακτηριστικών και εξαγωγή βάθους εικόνας) σε FPGAs διαστημικού βαθμού της αγοράς (**Κεφάλαιο 8**).

- Μεθοδολογία για την επιτάχυνση απαιτητικών DSP και AI αλγορίθμων στις ετερογενείς πολυπύρηνες Myriad VPUs (**Κεφάλαιο 9**).

- Τεχνικές ενσωματωμένης σχεδίασης υψηλού και χαμηλού επιπέδου για τον αποδοτικό προγραμματισμό απαιτητικών αλγορίθμων στις ετερογενείς πολυπύρηνες Myriad VPUs (**Κεφάλαιο 9**).

- Πειραματικά αποτελέσματα από την υλοποίηση διαφόρων DSP και AI πυρήνων (συνελίξεις εικόνας, παρακολούθηση πόζας) στις ετερογενείς πολυπύρηνες Myriad VPUs (**Κεφάλαιο 9**).

- Αξιολόγηση των BRAVE FPGAs της NanoXplore και των Myriad VPUs της Intel ως υποψήφιων επεξεργαστών για χρήση σε διαστημικές αποστολές (**Κεφάλαια 8– 9**).

Γλωσσάρι

Accurate Circuit   Το ακριβές (ορθό) κύκλωμα. Αποτελεί το κύκλωμα αναφο-
ράς, το οποίο δεν εφαρμόζει κάποια προσέγγιση και παράγει
ορθά αποτελέσματα.

Application-   Τύπος ολοκληρωμένου κυκλώματος. Είναι ένα τσιπ που
Specific Integrated   έχει κατασκευαστεί για να εκτελεί αποδοτικά μία συγκεκρι-
Circuit (ASIC)   μένη διαδικασία/εφαρμογή και δεν μπορεί να προγραμματι-
στεί ξανά.

Approximate   Το προσεγγιστικό κύκλωμα. Έχει σχεδιαστεί με βάση
Circuit   κάποια τεχνική προσέγγισης ώστε να παράγει χαμηλότερης
ποιότητας αποτελέσματα και να προσφέρει κέρδη σε πόρους.

Approximate   Πρότυπο σχεδίασης υπολογιστικών συστημάτων. Χαλα-
Computing   ρώνει την ακρίβεια των υπολογισμών που εκτελούνται με
στόχο τα κέρδη σε πόρους (ισχύς, ενέργεια, επιφάνεια, τα-
χύτητα).

Computer   Τομέας της Επιστήμης των Υπολογιστών. Ασχολείται
Arithmetic   με την αναπαράσταση των αριθμών στα υπολογιστικά συ-
στήματα, καθώς και με το πως πραγματοποιούνται οι
πράξεις και η αποθήκευση τους.

Computer Vision   Τομέας της Επιστήμης των Υπολογιστών. Ασχολείται με
την αλγοριθμική αναπαράσταση της όρασης, μελετώντας τη
θεωρία και την τεχνολογία που απαιτείται για τη σχεδίαση
συστημάτων που λαμβάνουν και αναλύουν δεδομένα από
ψηφιακές εικόνες.

Convolutional Neural Network (CNN)	Τύπος τεχνητού νευρωνικού δικτύου. Αποτελεί μία αρχιτεκτονική δικτύου για την διαχείριση δεδομένων με κάποια χωρική τοπολογία (π.χ., εικόνα), η οποία αποτελείται από εκπαιδευμένα βάρη και βασίζεται στην πράξη της συνέλιξης.
Cooperative Approximation	Τύπος εφαρμογής του Προσεγγιστικού Υπολογισμού. Εφαρμόζει δύο ή περισσότερες τεχνικές προσέγγισης στο ίδιο κύκλωμα, αυξάνοντας σημαντικά τον χώρο σχεδίασης και προσφέροντας πολλαπλές διαμορφώσεις της προσέγγισης.
Critical Path Delay	Η καθυστέρηση του κρίσιμου μονοπατιού ενός κυκλώματος. Πρόκειται για το μονοπάτι από την είσοδο ως την έξοδο με την μέγιστη καθυστέρηση.
Design Space Exploration (DSE)	Η διαδικασία εξερεύνησης του χώρου σχεδίασης. Διερευνά πολλαπλές σχεδιαστικές παραλλαγές για την υλοποίηση ενός υπολογιστικού συστήματος με στόχο την εύρεση των καλύτερων λύσεων.
Digital Signal Processing (DSP)	Τομέας της Επιστήμης των Υπολογιστών. Ασχολείται με την επεξεργασία σημάτων στο πεδίο του χρόνου, του χώρου ή/και της συχνότητας μέσω εξειδικευμένων ψηφιακών επεξεργαστών.
Dynamic Approximation	Τύπος εφαρμογής του Προσεγγιστικού Υπολογισμού. Εφαρμόζει μία τεχνική προσέγγισης που μπορεί να διαμορφωθεί, δηλαδή να αλλάξει βαθμό προσέγγισης, κατά τη διάρκεια του χρόνου εκτέλεσης.
Edge Computing	Πρότυπο σχεδίασης υπολογιστικών συστημάτων. Πραγματοποιεί αποκεντρωμένη επεξεργασία και αποθήκευση των δεδομένων κοντά στις πηγές δημιουργίας τους, χωρίς να απαιτείται η μεταφορά τους σε υπολογιστικές δομές νέφους.
Embedded System	Τύπος υπολογιστικού συστήματος. Αποτελείται από επεξεργαστές, ειδικούς επιταχυντές, μνήμες και περιφερειακά συσκευών εισόδου/εξόδου, και επιτελεί μια αποκλειστική λειτουργία σε ένα μεγαλύτερο μηχανικό ή ηλεκτρονικό σύστημα.

Error Analysis   Η μελέτη του λάθους ενός προσεγγιστικού κυκλώματος. Περιλαμβάνει μετρικές λάθους και προσομοιώσεις ώστε να αναλυθεί το μέγεθος, η συχνότητα και η διάδοση των λαθών, και να αξιολογηθεί η ποιότητα των αποτελεσμάτων.

Error Constraint   Ο περιορισμός λάθους ενός προσεγγιστικού κυκλώματος. Ορίζεται από τον χρήστη ή την εφαρμογή και τα προσεγγιστικά αποτελέσματα θα πρέπει να το ικανοποιούν.

Error Distance   Μετρική λάθους. Ορίζεται ως η απόλυτη τιμή της διαφοράς του αποτελέσματος του προσεγγιστικού κυκλώματος από το αντίστοιχο του ακριβούς κυκλώματος.

Field-Programmable Gate Array (FPGA)   Τύπος προγραμματιζόμενου ολοκληρωμένου κυκλώματος. Αποτελείται από προγραμματιζόμενα λογικά μπλοκ και από μια ιεραρχία επαναδιαμορφώσιμων διασυνδέσεων που επιτρέπουν στα μπλοκ να διασυνδεθούν μεταξύ τους ώστε να υλοποιηθεί το επιθυμητό κύκλωμα.

Fixed-Point Arithmetic   Πρότυπο αναπαράστασης αριθμών. Αναπαριστά έναν πραγματικό αριθμό με έναν σταθερό αριθμό ακεραίων και δεκαδικών ψηφίων.

Floating-Point Arithmetic   Πρότυπο αναπαράστασης αριθμών. Αναπαριστά έναν πραγματικό αριθμό με έναν σταθερό αριθμό σημαντικών ψηφίων κλιμακώνοντας τον με έναν εκθέτη σε κάποια σταθερή βάση.

Hardware Accelerator   Επιταχυντής υλικού. Αποτελεί έναν εξειδικευμένο επεξεργαστή, ο οποίος επιταχύνει διαδικασίες και εφαρμογές σε σύγκριση με έναν επεξεργαστή γενικού σκοπού.

Heterogeneous Computing   Πρότυπο σχεδίασης υπολογιστικών συστημάτων. Χρησιμοποιεί πολλαπλούς τύπους επεξεργαστή και τεχνολογίες μνήμης.

Logic Synthesis   Η σύνθεση ενός κυκλώματος. Είναι η διαδικασία με την οποία η περιγραφή της επιθυμητής συμπεριφοράς του κυκλώματος μετατρέπεται σε υλοποίηση (λογικές πύλες) συγκεκριμένης τεχνολογίας μέσω ενός εργαλείου σύνθεσης.

Mean Relative Error Distance (MRED)	Μετρική λάθους. Ορίζεται ως η μέση τιμή των απόλυτων σχετικών τιμών των διαφορών των αποτελεσμάτων του προσεγγιστικού κυκλώματος από τα αντίστοιχα του ακριβούς κυκλώματος.
Multi-Core Processor	Τύπος επεξεργαστή. Ενσωματώνει σε ένα ολοκληρωμένο κύκλωμα δύο ή περισσότερους ξεχωριστούς επεξεργαστές (πυρήνες).
Pareto Analysis	Η διαδικασία εύρεσης βέλτιστων σχεδιαστικών λύσεων. Αναζητάει ένα σύνολο από βέλτιστες σχεδιαστικές λύσεις, τις οποίες χαρακτηρίζει με βάση τα κριτήρια βελτιστοποίησης (π.χ., κατανάλωση ενέργειας και ποσοστό λάθους).
Pareto Front	Σύνορο Pareto. Αποτελεί το σύνολο των βέλτιστων σχεδιαστικών λύσεων, οι οποίες είναι καλύτερες από τις υπόλοιπες σε τουλάχιστον ένα κριτήριο βελτιστοποίησης χωρίς να είναι χειρότερες σε κάποιο άλλο.
Possibility of Relative Error Distance (PRED)	Μετρική λάθους. Ορίζεται ως η πιθανότητα να υπάρχει σχετική απόσταση τιμών μεγαλύτερη από κάποια σταθερά (π.χ., 2%).
Radiation-Hardened Electronic Component	Ηλεκτρονικό εξάρτημα ανθεκτικό σε ακτινοβολία. Έχει κατασκευαστεί με ειδική τεχνολογία ώστε να είναι ανθεκτικό σε ζημιές ή δυσλειτουργίες που προκαλούνται από υψηλά επίπεδα ιονίζουσας ακτινοβολίας σε δριμύ περιβάλλοντα όπως το διάστημα.
Space-Grade Electronic Component	Ηλεκτρονικό εξάρτημα διαστημικού βαθμού. Πληροί μια αυστηρή διαδικασία ελέγχων που περιλαμβάνει πιστοποίηση εξαρτημάτων και υλικών και έχει ως στόχο την χρήση σε μελλοντικές διαστημικές αποστολές.
System-on-Chip (SoC)	Τύπος ολοκληρωμένου κυκλώματος. Ενσωματώνει τα περισσότερα ή όλα τα στοιχεία ενός υπολογιστικού συστήματος, όπως επεξεργαστές, ειδικούς επιταχυντές, μνήμες, διεπαφές εισόδου/εξόδου, σε ένα μικροτσίπ.

Vision Processing Unit (VPU)   Τύπος ενσωματωμένου επιταχυντή. Αποτελείται από πολλαπλούς διανυσματικούς επεξεργαστές, μηχανές Τεχνητής Νοημοσύνης και φίλτρα εικόνας, και εξειδικεύεται σε εφαρμογές Υπολογιστικής Όρασης.

Bibliography

[1] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of Things (IoT): A Vision, Architectural Elements, and Future Directions," *Elsevier Future Generation Computer Systems*, vol. 29, no. 7, pp. 1645–1660, 2013.

[2] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge Computing: Vision and Challenges," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, 2016.

[3] G. Lentaris, K. Maragos, I. Stratakos, L. Papadopoulos, O. Papanikolaou, D. Soudris, M. Lourakis, X. Zabulis, D. Gonzalez-Arjona, and G. Furano, "High-Performance Embedded Computing in Space: Evaluation of Platforms for Vision-Based Navigation," *AIAA Journal of Aerospace Information Systems*, vol. 15, no. 4, pp. 178–192, 2018.

[4] G. Georgis, G. Lentaris, and D. Reisis, "Acceleration Techniques and Evaluation on Multi-Core CPU, GPU and FPGA for Image Processing and Super-Resolution," *Springer Journal of Real-Time Image Processing*, vol. 16, no. 4, pp. 1207–1234, 2019.

[5] G. Gobieski, B. Lucia, and N. Beckmann, "Intelligence Beyond the Edge: Inference on Intermittent Embedded Systems," in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019, pp. 199–213.

[6] S. Mittal, "A Survey of Techniques for Improving Energy Efficiency in Embedded Computing Systems," *International Journal of Computer Aided Engineering and Technology*, vol. 6, no. 4, pp. 440–459, 2014.

[7] H. Ghasemzadeh and R. Jafari, "Ultra Low-Power Signal Processing in Wearable Monitoring Systems: A Tiered Screening Architecture with Optimal Bit Resolution," *ACM Transactions on Embedded Computing Systems*, vol. 13, no. 1, pp. 1–23, 2013.

[8] G. E. Moore, "Cramming More Components onto Integrated Circuits," *IEEE Solid-State Circuits Society Newsletter*, vol. 38, no. 8, pp. 1–4, 1965.

[9] R. H. Dennard, F. H. Gaensslen, H.-N. Yu, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of Ion-Implanted MOSFET's with Very Small Physical

Dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, 1974.

[10] J. Shalf, "The Future of Computing Beyond Moore's Law," *Philosophical Transactions of the Royal Society A*, vol. 378, no. 2166, pp. 1–15, 2020.

[11] M. Bohr, "A 30 Year Retrospective on Dennard's MOSFET Scaling Paper," *IEEE Solid-State Circuits Society Newsletter*, vol. 12, no. 1, pp. 11–13, 2007.

[12] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark Silicon and the End of Multicore Scaling," in *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2011, pp. 365–376.

[13] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Power Challenges May End the Multicore Era," *Communications of the ACM*, vol. 56, no. 2, pp. 93–102, 2013.

[14] M. Shafique, S. Garg, J. Henkel, and D. Marculescu, "The EDA Challenges in the Dark Silicon Era," in *Design Automation Conference (DAC)*, 2014, pp. 1–6.

[15] J. Han and M. Orshansky, "Approximate Computing: An Emerging Paradigm for Energy-Efficient Design," in *IEEE European Test Symposium (ETS)*, 2013, pp. 1–6.

[16] S. Mittal, "A Survey of Techniques for Approximate Computing," *ACM Computing Surveys*, vol. 48, no. 4, pp. 1–33, 2016.

[17] Q. Xu, T. Mytkowicz, and N. S. Kim, "Approximate Computing: A Survey," *IEEE Design & Test*, vol. 33, no. 1, pp. 8–22, 2016.

[18] M. Shafique, R. Hafiz, S. Rehman, W. El-Harouni, and J. Henkel, "Cross-Layer Approximate Computing: From Logic to Architectures," in *Design Automation Conference (DAC)*, 2016, pp. 1–6.

[19] P. Stanley-Marbell, A. Alaghi, M. Carbin, E. Darulova, L. Dolecek, A. Gerstlauer, G. Gillani, D. Jevdjic, T. Moreau, M. Cacciotti, A. Daglis, N. E. Jerger, B. Falsafi, S. Misailovic, A. Sampson, and D. Zufferey, "Exploiting Errors for Efficiency: A Survey from Circuits to Applications," *ACM Computing Surveys*, vol. 53, no. 3, pp. 1–39, 2021.

[20] C. Kachris, B. Falsafi, and D. Soudris, *Hardware Accelerators in Data Centers*. Springer, 2019.

[21] A. Shawahna, S. M. Sait, and A. El-Maleh, "FPGA-Based Accelerators of Deep Learning Networks for Learning and Classification: A Review," *IEEE Access*, vol. 7, pp. 7823–7859, 2019.

[22] L. Yang, Z. Yan, M. Li, H. Kwon, L. Lai, T. Krishna, V. Chandra, W. Jiang, and Y. Shi, "Co-Exploration of Neural Architectures and Heterogeneous ASIC Accelerator Designs Targeting Multiple Tasks," in *Design Automation Conference (DAC)*, 2020, pp. 1–6.

[23] M. Zahran, "Heterogeneous Computing: Here to Stay," *Communications of ACM*, vol. 60, no. 3, pp. 42–45, 2017.

[24] M. Zahran, *Heterogeneous Computing: Hardware and Software Perspectives*, 1st ed.   Association for Computing Machinery, 2019.

[25] S. Mittal and J. S. Vetter, "A Survey of CPU-GPU Heterogeneous Computing Techniques," *ACM Computing Surveys*, vol. 47, no. 4, pp. 1–35, 2015.

[26] Y. Turakhia, B. Raghunathan, S. Garg, and D. Marculescu, "HaDeS: Architectural Synthesis for Heterogeneous Dark Silicon Chip Multi-Processors," in *Design Automation Conference (DAC)*, 2013, pp. 1–7.

[27] A. Marongiu, A. Capotondi, G. Tagliavini, and L. Benini, "Simplifying Many-Core-Based Heterogeneous SoC Programming With Offload Directives," *IEEE Transactions on Industrial Informatics*, vol. 11, no. 4, pp. 957–967, 2015.

[28] J. N. Mitchell, "Computer Multiplication and Division Using Binary Logarithms," *IRE Transactions on Electronic Computers*, vol. EC-11, no. 4, pp. 512–517, 1962.

[29] S. T. Chakradhar and A. Raghunathan, "Best-Effort Computing: Re-thinking Parallel Software and Hardware," in *Design Automation Conference (DAC)*, 2010, pp. 865–870.

[30] M. Carbin, D. Kim, S. Misailovic, and M. C. Rinard, "Proving Acceptability Properties of Relaxed Nondeterministic Approximate Programs," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2012, p. 169–180.

[31] V. K. Chippa, D. Mohapatra, K. Roy, S. T. Chakradhar, and A. Raghunathan, "Scalable Effort Hardware Design," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 9, pp. 2004–2016, 2014.

[32] A. Sampson, "Hardware and Software for Approximate Programming," *University of Washington PhD Dissertation*, pp. 1–212, 2015.

[33] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt,

D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Kille-brew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-Datacenter Performance Analysis of a Tensor Process-ing Unit," in *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2017, pp. 1–12.

[34] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2017.

[35] Xilinx, *Zynq-7000 SoC FPGAs*, (accessed October 2022). [Online]. Available: https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html

[36] B. Barry, C. Brick, F. Connor, D. Donohoe, D. Moloney, R. Richmond, M. O'Riordan, and V. Toma, "Always-on Vision Processing Unit for Mobile Applications," *IEEE Micro*, vol. 35, no. 2, pp. 56–66, 2015.

[37] V. Leon, M. A. Hanif, X. Jiao, M. Shafique, K. Pekmestzi, and D. Soudris, "Ap-proximate Computing Survey: A Taxonomy of Software, Hardware & Architec-tural Techniques and Applications," *ACM Journal on Emerging Technologies in Computing Systems*, vol. N/A, no. N/A, pp. 1–45, 2023, under submission.

[38] A. Sampson, A. Baixo, B. Ransford, T. Moreau, J. Yip, L. Ceze, and M. Oskin, "ACCEPT: A Programmer-Guided Compiler Framework for Practical Approx-imate Computing," *University of Washington Technical Report*, pp. 1–14, 2015.

[39] S. Mitra, M. K. Gupta, S. Misailovic, and S. Bagchi, "Phase-Aware Optimiza-tion in Approximate Computing," in *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2017, pp. 185–196.

[40] H. Hoffmann, S. Misailovic, S. Sidiroglou, A. Agarwal, and M. C. Rinard, "Using Code Perforation to Improve Performance, Reduce Energy Consumption, and Respond to Failures," *Massachusetts Institute of Technology Technical Report*, pp. 1–21, 2009.

[41] S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. C. Rinard, "Quality of Service Profiling," in *ACM/IEEE International Conference on Software Engineering (ICSE)*, vol. 1, 2010, pp. 25–34.

[42] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. C. Rinard, "Managing Performance vs. Accuracy Trade-Offs with Loop Perforation," in *ACM SIGSOFT Symposium and European Conference on Foundations of Software Engineering (FSE)*, 2011, pp. 124–134.

[43] Q. Shi, H. Hoffmann, and O. Khan, "A Cross-Layer Multicore Architecture to Tradeoff Program Accuracy and Resilience Overheads," *IEEE Computer Architecture Letters*, vol. 14, no. 2, pp. 85–89, 2015.

[44] H. Omar, M. Ahmad, and O. Khan, "GraphTuner: An Input Dependence Aware Loop Perforation Scheme for Efficient Execution of Approximated Graph Algorithms," in *IEEE International Conference on Computer Design (ICCD)*, 2017, pp. 201–208.

[45] S. Li, S. Park, and S. Mahlke, "Sculptor: Flexible Approximation with Selective Dynamic Loop Perforation," in *ACM International Conference on Supercomputing (ICS)*, 2018, p. 341–351.

[46] F. Baharvand and S. G. Miremadi, "LEXACT: Low Energy N-Modular Redundancy Using Approximate Computing for Real-Time Multicore Processors," *IEEE Transactions on Emerging Topics in Computing*, vol. 8, no. 2, pp. 431–441, 2020.

[47] C. Tan, T. S. Muthukaruppan, T. Mitra, and J. Lei, "Approximation-Aware Scheduling on Heterogeneous Multi-Core Architectures," in *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2015, pp. 618–623.

[48] A. Kanduri, A. Miele, A. M. Rahmani, P. Liljeberg, C. Bolchini, and N. Dutt, "Approximation-Aware Coordinated Power/Performance Management for Heterogeneous Multi-cores," in *Design Automation Conference (DAC)*, 2018, pp. 1–6.

[49] J. Meng, S. Chakradhar, and A. Raghunathan, "Best-Effort Parallel Execution Framework for Recognition and Mining Applications," in *IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, 2009, pp. 1–12.

[50] S. Byna, J. Meng, A. Raghunathan, S. Chakradhar, and S. Cadambi, "Best-Effort Semantic Document Search on GPUs," in *Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU)*, 2010, p. 86–93.

[51] A. Raha, S. Venkataramani, V. Raghunathan, and A. Raghunathan, "Quality Configurable Reduce-and-Rank for Energy Efficient Approximate Computing," in *Design, Automation & Test in Europe Conference (DATE)*, 2015, pp. 665–670.

[52] V. Vassiliadis, K. Parasyris, C. Chalios, C. D. Antonopoulos, S. Lalis, N. Bellas, H. Vandierendonck, and D. S. Nikolopoulos, "A Programming Model and Runtime System for Significance-Aware Energy-Efficient Computing," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2015, p. 275–276.

[53] V. Vassiliadis, J. Riehme, J. Deussen, K. Parasyris, C. D. Antonopoulos, N. Bellas, S. Lalis, and U. Naumann, "Towards Automatic Significance Analysis for Approximate Computing," in *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2016, p. 182–193.

[54] M. C. Rinard, "Probabilistic Accuracy Bounds for Fault-Tolerant Computations That Discard Tasks," in *ACM International Conference on Supercomputing (ICS)*, 2006, p. 324–334.

[55] M. C. Rinard, "Using Early Phase Termination to Eliminate Load Imbalances at Barrier Synchronization Points," in *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2007, p. 369–386.

[56] Y. Lin, C. Sakr, Y. Kim, and N. Shanbhag, "PredictiveNet: An Energy-Efficient Convolutional Neural Network via Zero Prediction," in *IEEE International Symposium on Circuits and Systems (ISCAS)*, 2017, pp. 1–4.

[57] V. Akhlaghi, A. Yazdanbakhsh, K. Samadi, R. K. Gupta, and H. Esmaeilzadeh, "SnaPEA: Predictive Early Activation for Reducing Computation in Deep Convolutional Neural Networks," in *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2018, p. 662–673.

[58] J. S. Miguel, M. Badr, and N. E. Jerger, "Load Value Approximation," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2014, pp. 127–139.

[59] A. Yazdanbakhsh, G. Pekhimenko, B. Thwaites, H. Esmaeilzadeh, O. Mutlu, and T. C. Mowry, "RFVP: Rollback-Free Value Prediction with Safe-to-Approximate Loads," *ACM Transactions on Architecture and Code Optimization*, vol. 12, no. 4, pp. 1–26, 2016.

[60] O. Kislal and M. T. Kandemir, "Data Access Skipping for Recursive Partitioning Methods," *Elsevier Computer Languages, Systems & Structures*, vol. 53, pp. 143–162, 2018.

[61] M. Samadi, J. Lee, D. A. Jamshidi, A. Hormati, and S. Mahlke, "SAGE: Self-Tuning Approximation for Graphics Engines," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013, pp. 13–24.

[62] M. Karakoy, O. Kislal, X. Tang, M. T. Kandemir, and M. Arunachalam, "Architecture-Aware Approximate Computing," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 3, no. 2, pp. 1–24, 2019.

[63] Q. Zhang, T. Wang, Y. Tian, F. Yuan, and Q. Xu, "ApproxANN: An Approximate Computing Framework for Artificial Neural Network," in *Design, Automation & Test in Europe Conference (DATE)*, 2015, pp. 701–706.

[64] S. Chaudhuri, S. Gulwani, R. Lublinerman, and S. Navidpour, "Proving Programs Robust," in *ACM SIGSOFT Symposium and European Conference on Foundations of Software Engineering (FSE)*, 2011, p. 102–112.

[65] M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke, "Paraprox: Pattern-Based Approximation for Data Parallel Applications," in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014, p. 35–50.

[66] A. K. Mishra, R. Barik, and S. Paul, "iACT: A Software-Hardware Framework for Understanding the Scope of Approximate Computing," in *Workshop on Approximate Computing Across the System Stack (WACAS)*, 2014, pp. 1–6.

[67] I. Brumar, M. Casas, M. Moreto, M. Valero, and G. S. Sohi, "ATM: Approximate Task Memoization in the Runtime System," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2017, pp. 1140–1150.

[68] G. Keramidas, C. Kokkala, and I. Stamoulis, "Clumsy Value Cache: An Approximate Memoization Technique for Mobile GPU Fragment Shaders," in *Workshop on Approximate Computing (WAPCO)*, 2015, pp. 1–6.

[69] G. Tziantzioulis, N. Hardavellas, and S. Campanoni, "Temporal Approximate Function Memoization," *IEEE Micro*, vol. 38, no. 4, pp. 60–70, 2018.

[70] C. Alvarez, J. Corbal, and M. Valero, "Fuzzy Memoization for Floating-Point Multimedia Applications," *IEEE Transactions on Computers*, vol. 54, no. 7, pp. 922–927, 2005.

[71] A. Rahimi, L. Benini, and R. K. Gupta, "Spatial Memoization: Concurrent Instruction Reuse to Correct Timing Errors in SIMD Architectures," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 60, no. 12, pp. 847–851, 2013.

[72] G. Zhang and D. Sanchez, "Leveraging Hardware Caches for Memoization," *IEEE Computer Architecture Letters*, vol. 17, no. 1, pp. 59–63, 2018.

[73] Z. Liu, A. Yazdanbakhsh, D. K. Wang, H. Esmaeilzadeh, and N. S. Kim, "AxMemo: Hardware-Compiler Co-Design for Approximate Code Memoization," in *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2019, p. 685–697.

[74] L. Renganarayana, V. Srinivasan, R. Nair, and D. Prener, "Programming with Relaxed Synchronization," in *ACM Workshop on Relaxing Synchronization for Multicore and Manycore Scalability (RACES)*, 2012, p. 41–50.

[75] S. Misailovic, S. Sidiroglou, and M. C. Rinard, "Dancing with Uncertainty," in *ACM Workshop on Relaxing Synchronization for Multicore and Manycore Scalability (RACES)*, 2012, p. 51–60.

[76] S. Misailovic, D. Kim, and M. C. Rinard, "Parallelizing Sequential Programs with Statistical Accuracy Tests," *ACM Transactions on Embedded Computing Systems*, vol. 12, no. 2s, pp. 1–26, 2013.

[77] S. Campanoni, G. Holloway, G.-Y. Wei, and D. Brooks, "HELIX-UP: Relaxing Program Semantics to Unleash Parallelization," in *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2015, p. 235–245.

[78] G. Stitt and D. Campbell, "PANDORA: An Architecture-Independent Parallelizing Approximation-Discovery Framework," *ACM Transactions on Embedded Computing Systems*, vol. 19, no. 5, pp. 1–17, 2020.

[79] J. Sreeram and S. Pande, "Exploiting Approximate Value Locality for Data Synchronization on Multi-Core Processors," in *IEEE International Symposium on Workload Characterization (IISWC)*, 2010, pp. 1–10.

[80] J. Mengt, A. Raghunathan, S. Chakradhar, and S. Byna, "Exploiting the Forgiving Nature of Applications for Scalable Parallel Execution," in *IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, 2010, pp. 1–12.

[81] M. C. Rinard, "Unsynchronized Techniques for Approximate Parallel Computing," in *ACM Workshop on Relaxing Synchronization for Multicore and Manycore Scalability (RACES)*, 2012, pp. 1–7.

[82] F. de Dinechin, C. Lauter, and G. Melquiond, "Certifying the Floating-Point Implementation of an Elementary Function Using Gappa," *IEEE Transactions on Computers*, vol. 60, no. 2, pp. 242–253, 2011.

[83] M. D. Linderman, M. Ho, D. L. Dill, T. H. Meng, and G. P. Nolan, "Towards Program Optimization through Automated Analysis of Numerical Precision," in *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2010, p. 230–237.

[84] W.-F. Chiang, M. Baranowski, I. Briggs, A. Solovyev, G. Gopalakrishnan, and Z. Rakamarić, "Rigorous Floating-Point Mixed-Precision Tuning," in *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2017, p. 300–315.

[85] E. Darulova and V. Kuncak, "Towards a Compiler for Reals," *ACM Transactions on Programming Languages and Systems*, vol. 39, no. 2, pp. 1–28, 2017.

[86] C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough, "Precimonious: Tuning Assistant for Floating-Point Precision," in *ACM/IEEE SC, International Conference for High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 1–12.

[87] H. Guo and C. Rubio-González, "Exploiting Community Structure for Floating-Point Precision Tuning," in *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2018, p. 333–343.

[88] M. O. Lam, J. K. Hollingsworth, B. R. de Supinski, and M. P. Legendre, "Automatically Adapting Programs for Mixed-Precision Floating-Point Computation," in *ACM International Conference on Supercomputing (ICS)*, 2013, p. 369–378.

[89] M. O. Lam and J. K. Hollingsworth, "Fine-Grained Floating-Point Precision Analysis," *SAGE International Journal of High Performance Computing Applications*, vol. 32, no. 2, p. 231–245, 2018.

[90] W.-F. Chiang, G. Gopalakrishnan, Z. Rakamaric, and A. Solovyev, "Efficient Search for Inputs Causing High Floating-Point Errors," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2014, p. 43–52.

[91] C. Rubio-González, C. Nguyen, B. Mehne, K. Sen, J. Demmel, W. Kahan, C. Iancu, W. Lavrijsen, D. H. Bailey, and D. Hough, "Floating-Point Precision Tuning Using Blame Analysis," in *IEEE/ACM International Conference on Software Engineering (ICSE)*, 2016, p. 1074–1085.

[92] S. Yesil, I. Akturk, and U. R. Karpuzcu, "Toward Dynamic Precision Scaling," *IEEE Micro*, vol. 38, no. 4, pp. 30–39, 2018.

[93] Y. Tian, Q. Zhang, T. Wang, F. Yuan, and Q. Xu, "ApproxMA: Approximate Memory Access for Dynamic Precision Scaling," in *Great Lakes Symposium on VLSI (GLSVLSI)*, 2015, p. 337–342.

[94] H. Menon, M. O. Lam, D. Osei-Kuffuor, M. Schordan, S. Lloyd, K. Mohror, and J. Hittinger, "ADAPT: Algorithmic Differentiation Applied to Floating-Point Precision Tuning," in *ACM/IEEE SC, International Conference for High Performance Computing, Networking, Storage and Analysis*, 2018, pp. 614–626.

[95] H. Brunie, C. Iancu, K. Z. Ibrahim, P. Brisk, and B. Cook, "Tuning Floating-Point Precision Using Dynamic Program Information and Temporal Locality," in *ACM/IEEE SC, International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020, pp. 1–14.

[96] I. Laguna, P. C. Wood, R. Singh, and S. Bagchi, "GPUMixer: Performance-Driven Floating-Point Tuning for GPU Scientific Applications," in *ISC International Conference on High Performance Computing (HPC)*, 2019, pp. 227–246.

[97] S. Kang, K. Choi, and Y. Park, "PreScaler: An Efficient System-Aware Precision Scaling Framework on Heterogeneous Systems," in *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2020, p. 280–292.

[98] N. Laptev, K. Zeng, and C. Zaniolo, "Early Accurate Results for Advanced Analytics on MapReduce," *Proceedings of the VLDB Endowment*, vol. 5, no. 10, p. 1028–1039, 2012.

[99] I. Goiri, R. Bianchini, S. Nagarakatte, and T. D. Nguyen, "ApproxHadoop: Bringing Approximations to MapReduce Frameworks," in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015, p. 383–397.

[100] G. Hu, S. Rigo, D. Zhang, and T. Nguyen, "Approximation with Error Bounds in Spark," in *IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2019, pp. 61–73.

[101] D. L. Quoc, R. Chen, P. Bhatotia, C. Fetzer, V. Hilt, and T. Strufe, "StreamApprox: Approximate Computing for Stream Analytics," in *ACM/IFIP/USENIX International Middleware Conference*, 2017, p. 185–197.

[102] D. R. Krishnan, D. L. Quoc, P. Bhatotia, C. Fetzer, and R. Rodrigues, "IncApprox: A Data Analytics System for Incremental Approximate Computing," in *International Conference on World Wide Web (WWW)*, 2016, p. 1133–1144.

[103] D. L. Quoc, M. Beck, P. Bhatotia, R. Chen, C. Fetzer, and T. Strufe, "PrivApprox: Privacy-Preserving Stream Analytics," in *USENIX Annual Technical Conference (ATC)*, 2017, p. 659–672.

[104] Z. Wen, D. L. Quoc, P. Bhatotia, R. Chen, and M. Lee, "ApproxIoT: Approximate Analytics for Edge Computing," in *IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2018, pp. 411–421.

[105] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica, "BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data," in *ACM SIGOPS European Conference on Computer Systems (EuroSys)*, 2013, p. 29–42.

[106] S. Kandula, A. Shanbhag, A. Vitorovic, M. Olma, R. Grandl, S. Chaudhuri, and B. Ding, "Quickr: Lazily Approximating Complex Ad-Hoc Queries in Big Data Clusters," in *ACM SIGMOD International Conference on Management of Data (MOD)*, 2016, p. 631–646.

[107] X. Zhang, J. Wang, and J. Yin, "Sapprox: Enabling Efficient and Accurate Approximations on Sub-Datasets with Distribution-Aware Online Sampling," *Proceedings of the VLDB Endowment*, vol. 10, no. 3, p. 109–120, 2016.

[108] M. R. Anderson and M. Cafarella, "Input Selection for Fast Feature Engineering," in *IEEE International Conference on Data Engineering (ICDE)*, 2016, pp. 577–588.

[109] Y. Park, J. Qing, X. Shen, and B. Mozafari, "BlinkML: Efficient Maximum Likelihood Estimation with Probabilistic Guarantees," in *ACM SIGMOD International Conference on Management of Data (MOD)*, 2019, p. 1135–1152.

[110] J. Ansel, Y. L. Wong, C. Chan, M. Olszewski, A. Edelman, and S. Amarasinghe, "Language and Compiler Support for Auto-Tuning Variable-Accuracy Algorithms," in *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2011, pp. 85–96.

[111] J. Sorber, A. Kostadinov, M. Garber, M. Brennan, M. D. Corner, and E. D. Berger, "Eon: A Language and Runtime System for Perpetual Systems," in *ACM International Conference on Embedded Networked Sensor Systems (SenSys)*, 2007, p. 161–174.

[112] W. Baek and T. M. Chilimbi, "Green: A Framework for Supporting Energy-Conscious Programming Using Controlled Approximation," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2010, pp. 198–209.

[113] B. Boston, A. Sampson, D. Grossman, and L. Ceze, "Probability Type Inference for Flexible Approximate Programming," in *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2015, pp. 470–487.

[114] M. Carbin, S. Misailovic, and M. C. Rinard, "Verifying Quantitative Reliability for Programs That Execute on Unreliable Hardware," in *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2013, p. 33–52.

[115] S. Misailovic, M. Carbin, S. Achour, Z. Qi, and M. C. Rinard, "Chisel: Reliability- and Accuracy-Aware Optimization of Approximate Computational Kernels," in *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2014, p. 309–328.

[116] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn, "Flikker: Saving DRAM Refresh-Power through Critical Data Partitioning," in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011, p. 213–224.

[117] S. Achour and M. C. Rinard, "Approximate Computation with Outlier Detection in Topaz," in *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2015, pp. 711–730.

[118] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "EnerJ: Approximate Data Types for Safe and General Low-Power Computation," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011, p. 164–174.

[119] J. Park, H. Esmaeilzadeh, X. Zhang, M. Naik, and W. Harris, "FlexJava: Language Support for Safe and Modular Approximate Programming," in *ACM SIGSOFT Symposium and European Conference on Foundations of Software Engineering (FSE)*, 2015, p. 745–757.

[120] J. Park, X. Zhang, K. Ni, H. Esmaeilzadeh, and M. Naik, "ExpAX: A Framework for Automating Approximate Programming," *Georgia Institute of Technology Technical Report*, pp. 1–17, 2014.

[121] N. D. Goodman, V. K. Mansinghka, D. Roy, K. Bonawitz, and J. B. Tenenbaum, "Church: A Language for Generative Models," in *Conference on Uncertainty in Artificial Intelligence (UAI)*, 2008, p. 220–229.

[122] V. K. Mansinghka, D. Selsam, and Y. N. Perov, "Venture: A Higher-Order Probabilistic Programming Platform with Programmable Inference," *ArXiv*, vol. abs/1404.0099, pp. 1–78, 2014.

[123] D. Tolpin, J.-W. van de Meent, H. Yang, and F. Wood, "Design and Implementation of Probabilistic Programming Language Anglican," in *Symposium on*

Implementation and Application of Functional Programming Languages (IFL), 2016, pp. 1–12.

[124] J. Bornholt, T. Mytkowicz, and K. S. McKinley, "Uncertain<T>: A First-Order Type for Uncertain Data," in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014, p. 51–66.

[125] A. Sampson, P. Panchekha, T. Mytkowicz, K. S. McKinley, D. Grossman, and L. Ceze, "Expressing and Verifying Probabilistic Assertions," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2014, p. 112–122.

[126] V. Fernando, K. Joshi, and S. Misailovic, "Verifying Safety and Accuracy of Approximate Parallel Programs via Canonical Sequentialization," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–29, 2019.

[127] K. Joshi, V. Fernando, and S. Misailovic, "Statistical Algorithmic Profiling for Randomized Approximate Programs," in *ACM/IEEE International Conference on Software Engineering (ICSE)*, 2019, pp. 608–618.

[128] S. Cherubin and G. Agosta, "Tools for Reduced Precision Computation: A Survey," *ACM Computing Surveys*, vol. 53, no. 2, pp. 1–35, 2020.

[129] E. Schkufza, R. Sharma, and A. Aiken, "Stochastic Optimization of Floating-Point Programs with Tunable Precision," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2014, p. 53–64.

[130] V. Gupta, D. Mohapatra, A. Raghunathan, and K. Roy, "Low-Power Digital Signal Processing Using Approximate Adders," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 1, pp. 124–137, 2013.

[131] Z. Yang, A. Jain, J. Liang, J. Han, and F. Lombardi, "Approximate XOR/XNOR-Based Adders for Inexact Computing," in *IEEE International Conference on Nanotechnology (NANO)*, 2013, pp. 690–693.

[132] M. Pashaeifar, M. Kamal, A. Afzali-Kusha, and M. Pedram, "Approximate Reverse Carry Propagate Adder for Energy-Efficient DSP Applications," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 11, pp. 2530–2541, 2018.

[133] A. Dalloo, A. Najafi, and A. Garcia-Ortiz, "Systematic Design of an Approximate Adder: The Optimized Lower Part Constant-OR Adder," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 8, pp. 1595–1599, 2018.

[134] Y. Kim, Y. Zhang, and P. Li, "An Energy Efficient Approximate Adder with Carry Skip for Error Resilient Neuromorphic VLSI Systems," in *International Conference on Computer-Aided Design (ICCAD)*, 2013, pp. 130–137.

[135] J. Hu and W. Qian, "A New Approximate Adder with Low Relative Error and Correct Sign Calculation," in *Design, Automation & Test in Europe Conference (DATE)*, 2015, pp. 1449–1454.

[136] A. B. Kahng and S. Kang, "Accuracy-Configurable Adder for Approximate Arithmetic Designs," in *Design Automation Conference (DAC)*, 2012, p. 820–825.

[137] R. Ye, T. Wang, F. Yuan, R. Kumar, and Q. Xu, "On Reconfiguration-Oriented Approximate Adder Design and Its Application," in *International Conference on Computer-Aided Design (ICCAD)*, 2013, pp. 48–54.

[138] M. Shafique, W. Ahmad, R. Hafiz, and J. Henkel, "A Low Latency Generic Accuracy Configurable Adder," in *Design Automation Conference (DAC)*, 2015, pp. 1–6.

[139] O. Akbari, M. Kamal, A. Afzali-Kusha, and M. Pedram, "RAP-CLA: A Reconfigurable Approximate Carry Look-Ahead Adder," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 65, no. 8, pp. 1089–1093, 2018.

[140] W. Xu, S. S. Sapatnekar, and J. Hu, "A Simple Yet Efficient Accuracy-Configurable Adder Design," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 6, pp. 1112–1125, 2018.

[141] F. Ebrahimi-Azandaryani, O. Akbari, M. Kamal, A. Afzali-Kusha, and M. Pedram, "Block-Based Carry Speculative Approximate Adder for Energy-Efficient Applications," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 67, no. 1, pp. 137–141, 2020.

[142] S. Hashemi, R. I. Bahar, and S. Reda, "DRUM: A Dynamic Range Unbiased Multiplier for Approximate Applications," in *International Conference on Computer-Aided Design (ICCAD)*, 2015, pp. 418–425.

[143] R. Zendegani, M. Kamal, M. Bahadori, A. Afzali-Kusha, and M. Pedram, "RoBA Multiplier: A Rounding-Based Approximate Multiplier for High-Speed yet Energy-Efficient Digital Signal Processing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 2, pp. 393–401, 2017.

[144] V. Leon, G. Zervakis, S. Xydis, D. Soudris, and K. Pekmestzi, "Walking through the Energy-Error Pareto Frontier of Approximate Multipliers," *IEEE Micro*, vol. 38, no. 4, pp. 40–49, 2018.

[145] V. Leon, T. Paparouni, E. Petrongonas, D. Soudris, and K. Pekmestzi, "Improving Power of DSP and CNN Hardware Accelerators Using Approximate Floating-Point Multipliers," *ACM Transactions on Embedded Computing Systems*, vol. 20, no. 5, pp. 1–21, 2021.

[146] S. Vahdat, M. Kamal, A. Afzali-Kusha, and M. Pedram, "TOSAM: An Energy-Efficient Truncation- and Rounding-Based Scalable Approximate Multiplier," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 5, pp. 1161–1173, 2019.

[147] V. Leon, K. Asimakopoulos, S. Xydis, D. Soudris, and K. Pekmestzi, "Cooperative Arithmetic-Aware Approximation Techniques for Energy-Efficient Multipliers," in *Design Automation Conference (DAC)*, 2019, pp. 1–6.

[148] F. Frustaci, S. Perri, P. Corsonello, and M. Alioto, "Approximate Multipliers With Dynamic Truncation for Energy Reduction via Graceful Quality Degradation," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 67, no. 12, pp. 3427–3431, 2020.

[149] W. Liu, L. Qian, C. Wang, H. Jiang, J. Han, and F. Lombardi, "Design of Approximate Radix-4 Booth Multipliers for Error-Tolerant Computing," *IEEE Transactions on Computers*, vol. 66, no. 8, pp. 1435–1441, 2017.

[150] S. Venkatachalam, E. Adams, H. J. Lee, and S.-B. Ko, "Design and Analysis of Area and Power Efficient Approximate Booth Multipliers," *IEEE Transactions on Computers*, vol. 68, no. 11, pp. 1697–1703, 2019.

[151] H. Jiang, J. Han, F. Qiao, and F. Lombardi, "Approximate Radix-8 Booth Multipliers for Low-Power and High-Performance Operation," *IEEE Transactions on Computers*, vol. 65, no. 8, pp. 2638–2644, 2016.

[152] H. Waris, C. Wang, and W. Liu, "Hybrid Low Radix Encoding-Based Approximate Booth Multipliers," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 67, no. 12, pp. 3367–3371, 2020.

[153] V. Leon, G. Zervakis, D. Soudris, and K. Pekmestzi, "Approximate Hybrid High Radix Encoding for Energy-Efficient Inexact Multipliers," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 3, pp. 421–430, 2018.

[154] A. Momeni, J. Han, P. Montuschi, and F. Lombardi, "Design and Analysis of Approximate Compressors for Multiplication," *IEEE Transactions on Computers*, vol. 64, no. 4, pp. 984–994, 2015.

[155] O. Akbari, M. Kamal, A. Afzali-Kusha, and M. Pedram, "Dual-Quality 4:2 Compressors for Utilizing in Dynamic Accuracy Configurable Multipliers," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 4, pp. 1352–1361, 2017.

[156] F. Sabetzadeh, M. H. Moaiyeri, and M. Ahmadinejad, "A Majority-Based Imprecise Multiplier for Ultra-Efficient Approximate Image Multiplication," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 66, no. 11, pp. 4200–4208, 2019.

[157] D. Esposito, A. G. M. Strollo, E. Napoli, D. De Caro, and N. Petra, "Approximate Multipliers Based on New Approximate Compressors," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 65, no. 12, pp. 4169–4182, 2018.

[158] A. G. M. Strollo, E. Napoli, D. De Caro, N. Petra, and G. D. Meo, "Comparison and Extension of Approximate 4-2 Compressors for Low-Power Approximate Multipliers," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, no. 9, pp. 3021–3034, 2020.

[159] W. Liu, J. Xu, D. Wang, C. Wang, P. Montuschi, and F. Lombardi, "Design and Evaluation of Approximate Logarithmic Multipliers for Low Power Error-Tolerant Applications," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 65, no. 9, pp. 2856–2868, 2018.

[160] H. Saadat, H. Javaid, A. Ignjatovic, and S. Parameswaran, "REALM: Reduced-Error Approximate Log-based Integer Multiplier," in *Design, Automation & Test in Europe Conference (DATE)*, 2020, pp. 1366–1371.

[161] M. S. Ansari, B. F. Cockburn, and J. Han, "An Improved Logarithmic Multiplier for Energy-Efficient Neural Computing," *IEEE Transactions on Computers*, vol. 70, no. 4, pp. 614–625, 2021.

[162] R. Pilipović, P. Bulić, and U. Lotrič, "A Two-Stage Operand Trimming Approximate Logarithmic Multiplier," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 68, no. 6, pp. 2535–2545, 2021.

[163] S. Hashemi, R. I. Bahar, and S. Reda, "A Low-Power Dynamic Divider for Approximate Applications," in *Design Automation Conference (DAC)*, 2016, pp. 1–6.

[164] H. Jiang, L. liu, F. Lombardi, and J. Han, "Low-Power Unsigned Divider and Square Root Circuit Designs Using Adaptive Approximation," *IEEE Transactions on Computers*, vol. 68, no. 11, pp. 1635–1646, 2019.

[165] L. Chen, J. Han, W. Liu, and F. Lombardi, "Design of Approximate Unsigned Integer Non-Restoring Divider for Inexact Computing," in *Great Lakes Symposium on VLSI (GLSVLSI)*, 2015, pp. 51–56.

[166] L. Chen, J. Han, W. Liu, and F. Lombardi, "On the Design of Approximate Restoring Dividers for Error-Tolerant Applications," *IEEE Transactions on Computers*, vol. 65, no. 8, pp. 2522–2533, 2016.

[167] L. Chen, J. Han, W. Liu, P. Montuschi, and F. Lombardi, "Design, Evaluation and Application of Approximate High-Radix Dividers," *IEEE Transactions on Multi-Scale Computing Systems*, vol. 4, no. 3, pp. 299–312, 2018.

[168] E. Adams, S. Venkatachalam, and S.-B. Ko, "Approximate Restoring Dividers Using Inexact Cells and Estimation From Partial Remainders," *IEEE Transactions on Computers*, vol. 69, no. 4, pp. 468–474, 2020.

[169] R. Zendegani, M. Kamal, A. Fayyazi, A. Afzali-Kusha, S. Safari, and M. Pedram, "SEERAD: A High Speed yet Energy-Efficient Rounding-Based Approximate Divider," in *Design, Automation & Test in Europe Conference (DATE)*, 2016, pp. 1481–1484.

[170] S. Vahdat, M. Kamal, A. Afzali-Kusha, M. Pedram, and Z. Navabi, "TruncApp: A Truncation-Based Approximate Divider for Energy Efficient DSP Applications," in *Design, Automation & Test in Europe Conference (DATE)*, 2017, pp. 1635–1638.

[171] M. Imani, R. Garcia, A. Huang, and T. Rosing, "CADE: Configurable Approximate Divider for Energy Efficiency," in *Design, Automation & Test in Europe Conference (DATE)*, 2019, pp. 586–589.

[172] W. Liu, J. Li, T. Xu, C. Wang, P. Montuschi, and F. Lombardi, "Combining Restoring Array and Logarithmic Dividers into an Approximate Hybrid Design," in *IEEE Symposium on Computer Arithmetic (ARITH)*, 2018, pp. 92–98.

[173] H. Saadat, H. Javaid, and S. Parameswaran, "Approximate Integer and Floating-Point Dividers with Near-Zero Error Bias," in *Design Automation Conference (DAC)*, 2019, pp. 1–6.

[174] J. Schlachter, V. Camus, K. V. Palem, and C. Enz, "Design and Applications of Approximate Circuits by Gate-Level Pruning," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 5, pp. 1694–1702, 2017.

[175] I. Scarabottolo, G. Ansaloni, and L. Pozzi, "Circuit Carving: A Methodology for the Design of Approximate Hardware," in *Design, Automation & Test in Europe Conference (DATE)*, 2018, pp. 545–550.

[176] S. Venkataramani, K. Roy, and A. Raghunathan, "Substitute-and-Simplify: A Unified Design Paradigm for Approximate and Quality Configurable Circuits," in *Design, Automation & Test in Europe Conference (DATE)*, 2013, pp. 1367–1372.

[177] G. Liu and Z. Zhang, "Statistically Certified Approximate Logic Synthesis," in *International Conference on Computer-Aided Design (ICCAD)*, 2017, pp. 344–351.

[178] J. Castro-Godínez, H. Barrantes-García, M. Shafique, and J. Henkel, "AxLS: A Framework for Approximate Logic Synthesis Based on Netlist Transformations," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 68, no. 8, pp. 2845–2849, 2021.

[179] S. Venkataramani, A. Sabne, V. Kozhikkottu, K. Roy, and A. Raghunathan, "SALSA: Systematic Logic Synthesis of Approximate Circuits," in *Design Automation Conference (DAC)*, 2012, pp. 796–801.

[180] A. Ranjan, A. Raha, S. Venkataramani, K. Roy, and A. Raghunathan, "ASLAN: Synthesis of Approximate Sequential Circuits," in *Design, Automation & Test in Europe Conference (DATE)*, 2014, pp. 1–6.

[181] J. Miao, A. Gerstlauer, and M. Orshansky, "Approximate Logic Synthesis under General Error Magnitude and Frequency Constraints," in *International Conference on Computer-Aided Design (ICCAD)*, 2013, pp. 779–786.

[182] S. Hashemi, H. Tann, and S. Reda, "BLASYS: Approximate Logic Synthesis Using Boolean Matrix Factorization," in *Design Automation Conference (DAC)*, 2018, pp. 1–6.

[183] A. Yazdanbakhsh, D. Mahajan, B. Thwaites, J. Park, A. Nagendrakumar, S. Sethuraman, K. Ramkrishnan, N. Ravindran, R. Jariwala, A. Rahimi, H. Esmaeilzadeh, and K. Bazargan, "Axilog: Language Support for Approximate Hardware Design," in *Design, Automation & Test in Europe Conference (DATE)*, 2015, pp. 812–817.

[184] K. Nepal, Y. Li, R. I. Bahar, and S. Reda, "ABACUS: A Technique for Automated Behavioral Synthesis of Approximate Computing Circuits," in *Design, Automation & Test in Europe Conference (DATE)*, 2014, pp. 1–6.

[185] S. Lee, L. K. John, and A. Gerstlauer, "High-Level Synthesis of Approximate Hardware under Joint Precision and Voltage Scaling," in *Design, Automation & Test in Europe Conference (DATE)*, 2017, pp. 187–192.

[186] K. Nepal, S. Hashemi, H. Tann, R. I. Bahar, and S. Reda, "Automated High-Level Generation of Low-Power Approximate Computing Circuits," *IEEE Transactions on Emerging Topics in Computing*, vol. 7, no. 1, pp. 18–30, 2019.

[187] J. Castro-Godínez, J. Mateus-Vargas, M. Shafique, and J. Henkel, "AxHLS: Design Space Exploration and High-Level Synthesis of Approximate Accelerators using Approximate Functional Units and Analytical Models," in *International Conference On Computer Aided Design (ICCAD)*, 2020, pp. 1–9.

[188] L. Sekanina and Z. Vasicek, "Approximate Circuit Design by Means of Evolvable Hardware," in *IEEE International Conference on Evolvable Systems (ICES)*, 2013, pp. 21–28.

[189] Z. Vasicek and L. Sekanina, "Evolutionary Approach to Approximate Digital Circuits Design," *IEEE Transactions on Evolutionary Computation*, vol. 19, no. 3, pp. 432–444, 2015.

[190] V. Mrazek, R. Hrbacek, Z. Vasicek, and L. Sekanina, "EvoApprox8b: Library of Approximate Adders and Multipliers for Circuit Design and Benchmarking of Approximation Methods," in *Design, Automation & Test in Europe Conference (DATE)*, 2017, pp. 258–261.

[191] Z. Vasicek and L. Sekanina, "Search-based Synthesis of Approximate Circuits Implemented into FPGAs," in *International Conference on Field Programmable Logic and Applications (FPL)*, 2016, pp. 1–4.

[192] Z. Vasicek, V. Mrazek, and L. Sekanina, "Automated Circuit Approximation Method Driven by Data Distribution," in *Design, Automation & Test in Europe Conference (DATE)*, 2019, pp. 96–101.

[193] A. B. Kahng, S. Kang, R. Kumar, and J. Sartori, "Slack Redistribution for Graceful Degradation Under Voltage Overscaling," in *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2010, pp. 825–831.

[194] D. Mohapatra, V. K. Chippa, A. Raghunathan, and K. Roy, "Design of Voltage-Scalable Meta-Functions for Approximate Computing," in *Design, Automation & Test in Europe Conference (DATE)*, 2011, pp. 1–6.

[195] J. Chen and J. Hu, "Energy-Efficient Digital Signal Processing via Voltage-Overscaling-Based Residue Number System," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 21, no. 7, pp. 1322–1332, 2013.

[196] J. Zhang, K. Rangineni, Z. Ghodsi, and S. Garg, "ThUnderVolt: Enabling Aggressive Voltage Underscaling and Timing Error Resilience for Energy Efficient Deep Learning Accelerators," in *Design Automation Conference (DAC)*, 2018, pp. 1–6.

[197] P. Pandey, P. Basu, K. Chakraborty, and S. Roy, "GreenTPU: Improving Timing Error Resilience of a Near-Threshold Tensor Processing Unit," in *Design Automation Conference (DAC)*, 2019, pp. 1–6.

[198] J. Wang, X. Fu, X. Wang, S. Liu, L. Gao, and W. Zhang, "Enabling Energy-Efficient and Reliable Neural Network via Neuron-Level Voltage Scaling," *IEEE Transactions on Computers*, vol. 69, no. 10, pp. 1460–1473, 2020.

[199] G. Zervakis, S. Xydis, D. Soudris, and K. Pekmestzi, "Multi-Level Approximate Accelerator Synthesis Under Voltage Island Constraints," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 66, no. 4, pp. 607–611, 2019.

[200] Y. Liu, T. Zhang, and K. K. Parhi, "Computation Error Analysis in Digital Signal Processing Systems With Overscaled Supply Voltage," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 18, no. 4, pp. 517–526, 2010.

[201] D. Jeon, M. Seok, Z. Zhang, D. Blaauw, and D. Sylvester, "Design Methodology for Voltage-Overscaled Ultra-Low-Power Systems," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 59, no. 12, pp. 952–956, 2012.

[202] R. Ragavan, B. Barrois, C. Killian, and O. Sentieys, "Pushing the Limits of Voltage Over-Scaling for Error-Resilient Applications," in *Design, Automation & Test in Europe Conference (DATE)*, 2017, pp. 476–481.

[203] X. Jiao, D. Ma, W. Chang, and Y. Jiang, "LEVAX: An Input-Aware Learning-Based Error Model of Voltage-Scaled Functional Units," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 12, pp. 5032–5041, 2020.

[204] G. Zervakis, F. Ntouskas, S. Xydis, D. Soudris, and K. Pekmestzi, "VOSsim: A Framework for Enabling Fast Voltage Overscaling Simulation for Approximate Computing Circuits," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 6, pp. 1204–1208, 2018.

[205] T. Alan and J. Henkel, "SlackHammer: Logic Synthesis for Graceful Errors Under Frequency Scaling," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2802–2811, 2018.

[206] S. G. Ramasubramanian, S. Venkataramani, A. Parandhaman, and A. Raghunathan, "Relax-and-Retime: A Methodology for Energy-Efficient Recovery Based Design," in *Design Automation Conference (DAC)*, 2013, pp. 1–6.

[207] K. Shi, D. Boland, E. Stott, S. Bayliss, and G. A. Constantinides, "Datapath Synthesis for Overclocking: Online Arithmetic for Latency-Accuracy Trade-offs," in *Design Automation Conference (DAC)*, 2014, pp. 1–6.

[208] Y. Wang, J. Deng, Y. Fang, H. Li, and X. Li, "Resilience-Aware Frequency Tuning for Neural-Network-Based Approximate Computing Chips," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 10, pp. 2736–2748, 2017.

[209] M. R. Choudhury, V. Chandra, R. C. Aitken, and K. Mohanram, "Time-Borrowing Circuit Designs and Hardware Prototyping for Timing Error Resilience," *IEEE Transactions on Computers*, vol. 63, no. 2, pp. 497–509, 2014.

[210] R. Ragavan, C. Killian, and O. Sentieys, "Adaptive Overclocking and Error Correction Based on Dynamic Speculation Window," in *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2016, pp. 325–330.

[211] Z.-H. Li, T.-T. Zhu, Z.-J. Chen, J.-Y. Meng, X.-Y. Xiang, and X.-L. Yan, "Eliminating Timing Errors Through Collaborative Design to Maximize the Throughput," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 2, pp. 670–682, 2017.

[212] S. Roy and K. Chakraborty, "Predicting Timing Violations through Instruction-Level Path Sensitization Analysis," in *Design Automation Conference (DAC)*, 2012, pp. 1074–1081.

[213] J. Constantin, L. Wang, G. Karakonstantis, A. Chattopadhyay, and A. Burg, "Exploiting Dynamic Timing Margins in Microprocessors for Frequency-Over-Scaling with Instruction-Based Clock Adjustment," in *Design, Automation & Test in Europe Conference (DATE)*, 2015, pp. 381–386.

[214] X. Jiao, Y. Jiang, A. Rahimi, and R. K. Gupta, "WILD: A Workload-Based Learning Model to Predict Dynamic Delay of Functional Units," in *IEEE International Conference on Computer Design (ICCD)*, 2016, pp. 185–192.

[215] X. Jiao, Y. Jiang, A. Rahimi, and R. K. Gupta, "SLoT: A Supervised Learning Model to Predict Dynamic Timing Errors of Functional Units," in *Design, Automation & Test in Europe Conference (DATE)*, 2017, pp. 1183–1188.

[216] H. Jiang, F. J. H. Santiago, H. Mo, L. Liu, and J. Han, "Approximate Arithmetic Circuits: A Survey, Characterization, and Recent Applications," *Proceedings of the IEEE*, vol. 108, no. 12, pp. 2108–2135, 2020.

[217] S. Ullah, S. S. Murthy, and A. Kumar, "SMApproxLib: Library of FPGA-based Approximate Multipliers," in *Design Automation Conference (DAC)*, 2018, pp. 1–6.

[218] K. Chen, L. Chen, P. Reviriego, and F. Lombardi, "Efficient Implementations of Reduced Precision Redundancy (RPR) Multiply and Accumulate (MAC)," *IEEE Transactions on Computers*, vol. 68, no. 5, pp. 784–790, 2019.

[219] G. A. Gillani, M. A. Hanif, B. Verstoep, S. H. Gerez, M. Shafique, and A. B. J. Kokkeler, "MACISH: Designing Approximate MAC Accelerators With Internal-Self-Healing," *IEEE Access*, vol. 7, pp. 77 142–77 160, 2019.

[220] K. Manikantta Reddy, M. H. Vasantha, Y. B. Nithin Kumar, and D. Dwivedi, "Design of Approximate Booth Squarer for Error-Tolerant Computing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 5, pp. 1230–1241, 2020.

[221] G. A. Gillani, M. A. Hanif, M. Krone, S. H. Gerez, M. Shafique, and A. B. J. Kokkeler, "SquASH: Approximate Square-Accumulate With Self-Healing," *IEEE Access*, vol. 6, pp. 49 112–49 128, 2018.

[222] L. Chen, J. Han, W. Liu, and F. Lombardi, "Algorithm and Design of a Fully Parallel Approximate Coordinate Rotation Digital Computer (CORDIC)," *IEEE Transactions on Multi-Scale Computing Systems*, vol. 3, no. 3, pp. 139–151, 2017.

[223] I. Scarabottolo, G. Ansaloni, G. A. Constantinides, L. Pozzi, and S. Reda, "Approximate Logic Synthesis: A Survey," *Proceedings of the IEEE*, vol. 108, no. 12, pp. 2195–2213, 2020.

[224] F. J. Kurdahi, A. Eltawil, K. Yi, S. Cheng, and A. Khajeh, "Low-Power Multimedia System Design by Aggressive Voltage Scaling," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 18, no. 5, pp. 852–856, 2010.

[225] K. Shi, D. Boland, and G. A. Constantinides, "Accuracy-Performance Trade-offs on an FPGA through Overclocking," in *IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2013, pp. 29–36.

[226] V. Leon, S. Xydis, D. Soudris, and K. Pekmestzi, "Energy-Efficient VLSI Implementation of Multipliers with Double LSB Operands," *IET Circuits, Devices & Systems*, vol. 13, no. 6, pp. 816–821, 2019.

[227] B. Parhami, *Computer Arithmetic: Algorithms and Hardware Designs*, 1st ed. Oxford University Press, 2000.

[228] H. L. Garner, "The Residue Number System," *IRE Transactions on Electronic Computers*, vol. EC-8, no. 2, pp. 140–147, 1959.

[229] E. E. Swartzlander and A. Alexopoulos, "The Sign/Logarithm Number System," *IEEE Transactions on Computers*, vol. C-24, no. 12, pp. 1238–1242, 1975.

[230] A. K. Verma and P. Ienne, "Improved Use of the Carry-Save Representation for the Synthesis of Complex Arithmetic Circuits," in *IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, 2004, pp. 791–798.

[231] B. Parhami, "Double-Least-Significant-Bits 2's-Complement Number Representation Scheme with Bitwise Complementation and Symmetric Range," *IET Circuits, Devices and Systems*, vol. 2, no. 2, pp. 179–186, 2008.

[232] G. Jaberipur, "A One-Step Modulo $2^n + 1$ Adder Based on Double-LSB Representation of Residues," *CSI Journal on Computer Science and Engineering*, vol. 4, no. 2-4, pp. 10–16, 2006.

[233] G. Jaberipur and H. Alavi, "A Modulo $2^n + 1$ Multiplier with Double-LSB Encoding of Residues," in *CSI International Symposium on Computer Architecture and Digital Systems*, 2010, pp. 147–150.

[234] E. Vassalos, D. Bakalis, and H. T. Vergos, "On the Use of Double-LSB and Signed-LSB Encodings for RNS," in *International Conference on Digital Signal Processing (DSP)*, 2011, pp. 1–6.

[235] W. Wang, X. Huang, N. Emmart, and C. Weems, "VLSI Design of a Large-Number Multiplier for Fully Homomorphic Encryption," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 9, pp. 1879–1887, 2014.

[236] C. Rafferty, M. O'Neill, and N. Hanley, "Evaluation of Large Integer Multiplication Methods on Hardware," *IEEE Transactions on Computers*, vol. 66, no. 8, pp. 1369–1382, 2017.

[237] F. de Dinechin and B. Pasca, "Large Multipliers with Fewer DSP Blocks," in *International Conference on Field Programmable Logic and Applications (FPL)*, 2009, pp. 250–255.

[238] N. Weste and D. Harris, *CMOS VLSI Design: A Circuits and Systems Perspective*, 4th ed.   Addison-Wesley Publishing Company, 2010.

[239] K. Tsoumanis, S. Xydis, C. Efstathiou, N. Moschopoulos, and K. Pekmestzi, "An Optimized Modified Booth Recoder for Efficient Design of the Add-Multiply Operator," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 61, no. 4, pp. 1133–1143, 2014.

[240] V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Analysis and Characterization of Inherent Application Resilience for Approximate Computing," in *Design Automation Conference (DAC)*, 2013, pp. 1–9.

[241] A. Lingamneni, C. Enz, K. Palem, and C. Piguet, "Highly Energy-Efficient and Quality-Tunable Inexact FFT Accelerators," in *IEEE Custom Integrated Circuits Conference*, 2014, pp. 1–4.

[242] D. Shin and S. K. Gupta, "Approximate Logic Synthesis for Error Tolerant Applications," in *Design, Automation & Test in Europe Conference (DATE)*, 2010, pp. 957–960.

[243] S. Venkataramani, V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Quality Programmable Vector Processors for Approximate Computing," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013, pp. 1–12.

[244] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural Acceleration for General-Purpose Approximate Programs," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2012, pp. 449–460.

[245] M. Kamal, A. Ghasemazar, A. Afzali-Kusha, and M. Pedram, "Improving Efficiency of Extensible Processors by Using Approximate Custom Instructions," in *Design, Automation & Test in Europe Conference (DATE)*, 2014, pp. 1–4.

[246] V. Gupta, D. Mohapatra, A. Raghunathan, and K. Roy, "Low-Power Digital Signal Processing Using Approximate Adders," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 1, pp. 124–137, 2013.

[247] N. Zhu, W. L. Goh, W. Zhang, K. S. Yeo, and Z. H. Kong, "Design of Low-Power High-Speed Truncation-Error-Tolerant Adder and Its Application in Digital Signal Processing," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 18, no. 8, pp. 1225–1229, 2010.

[248] H. R. Mahdiani, A. Ahmadi, S. M. Fakhraie, and C. Lucas, "Bio-Inspired Imprecise Computational Blocks for Efficient VLSI Implementation of Soft-Computing Applications," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 57, no. 4, pp. 850–862, 2010.

[249] A. K. Verma, P. Brisk, and P. Ienne, "Variable Latency Speculative Addition: A New Paradigm for Arithmetic Circuit Design," in *Design, Automation & Test in Europe Conference (DATE)*, 2008, pp. 1250–1255.

[250] P. Kulkarni, P. Gupta, and M. Ercegovac, "Trading Accuracy for Power with an Underdesigned Multiplier Architecture," in *IEEE International Conference on VLSI Design*, 2011, pp. 346–351.

[251] C.-H. Lin and I.-C. Lin, "High Accuracy Approximate Multiplier with Error Correction," in *IEEE International Conference on Computer Design (ICCD)*, 2013, pp. 33–38.

[252] K. Y. Kyaw, W. L. Goh, and K. S. Yeo, "Low-Power High-Speed Multiplier for Error-Tolerant Application," in *IEEE International Conference of Electron Devices and Solid-State Circuits (EDSSC)*, 2010, pp. 1–4.

[253] C. Liu, J. Han, and F. Lombardi, "A Low-Power, High-Performance Approximate Multiplier with Configurable Partial Error Recovery," in *Design, Automation & Test in Europe Conference (DATE)*, 2014, pp. 1–4.

[254] S. Narayanamoorthy, H. A. Moghaddam, Z. Liu, T. Park, and N. S. Kim, "Energy-Efficient Approximate Multiplication for Digital Signal Processing and Classification Applications," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 23, no. 6, pp. 1180–1184, 2015.

[255] G. Zervakis, K. Tsoumanis, S. Xydis, D. Soudris, and K. Pekmestzi, "Design-Efficient Approximate Multiplication Circuits Through Partial Product Perforation," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, no. 10, pp. 3105–3117, 2016.

[256] Z. Babic, A. Avramovic, and P. Bulic, "An Iterative Logarithmic Multiplier," *Elsevier Microprocessors and Microsystems*, vol. 35, no. 1, pp. 23–33, 2011.

[257] J. Liang, J. Han, and F. Lombardi, "New Metrics for the Reliability of Approximate and Probabilistic Adders," *IEEE Transactions on Computers*, vol. 62, no. 9, pp. 1760–1771, 2013.

[258] J. Park, E. Amaro, D. Mahajan, B. Thwaites, and H. Esmaeilzadeh, "AxGames: Towards Crowdsourcing Quality Target Determination in Approximate Computing," in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016, pp. 623–636.

[259] M. Langhammer and B. Pasca, "Design and Implementation of an Embedded FPGA Floating Point DSP Block," in *IEEE Symposium on Computer Arithmetic*, 2015, pp. 26–33.

[260] H. Zhang, M. Putic, and J. Lach, "Low Power GPGPU Computation with Imprecise Hardware," in *Design Automation Conference (DAC)*, 2014, pp. 1–6.

[261] M. Imani, A. Sokolova, R. Garcia, A. Huang, F. Wu, B. Aksanli, and T. Rosing, "ApproxLP: Approximate Multiplication with Linearization and Iterative Error Control," in *Design Automation Conference (DAC)*, 2019, pp. 1–6.

[262] H. Jiang, C. Liu, F. Lombardi, and J. Han, "Low-Power Approximate Unsigned Multipliers With Configurable Error Recovery," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 66, no. 1, pp. 189–202, 2019.

[263] IEEE, "IEEE Standard for Floating-Point Arithmetic," *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pp. 1–84, 2019.

[264] J. Y. F. Tong, D. Nagle, and R. A. Rutenbar, "Reducing Power by Optimizing the Necessary Precision/Range of Floating-Point Arithmetic," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 8, no. 3, pp. 273–286, 2000.

[265] İ. Taştan, M. Karaca, and A. Yurdakul, "Approximate CPU Design for IoT End-Devices with Learning Capabilities," *MDPI Electronics*, vol. 9, no. 1, pp. 1–20, 2020.

[266] M. Schulte and E. Swartzlander, "Truncated Multiplication with Correction Constant [for DSP]," in *IEEE Workshop on VLSI Signal Processing*, 1993, pp. 388–396.

[267] M. Imani, D. Peroni, and T. Rosing, "CFPU: Configurable Floating Point Multiplier for Energy-Efficient Computing," in *Design Automation Conference (DAC)*, 2017, pp. 1–6.

[268] M. Imani, R. Garcia, S. Gupta, and T. Rosing, "RMAC: Runtime Configurable Floating Point Multiplier for Approximate Computing," in *International Symposium on Low Power Electronics and Design (ISLPED)*, 2018, pp. 1–6.

[269] A. Lingamneni, C. Enz, K. Palem, and C. Piguet, "Synthesizing Parsimonious Inexact Circuits Through Probabilistic Design Techniques," *ACM Transactions on Embedded Computing Systems*, vol. 12, no. 2s, pp. 1–26, 2013.

[270] R. Pilipović and P. Bulić, "On the Design of Logarithmic Multiplier Using Radix-4 Booth Encoding," *IEEE Access*, vol. 8, pp. 64 578–64 590, 2020.

[271] F. Zhu, S. Zhen, X. Yi, H. Pei, B. Hou, and Y. He, "Design of Approximate Radix-256 Booth Encoding for Error-Tolerant Computing," *IEEE Transactions on Circuits and Systems II: Express Briefs*, pp. 1–5, 2022.

[272] K.-J. Cho, K.-C. Lee, J.-G. Chung, and K. Parhi, "Design of Low-Error Fixed-Width Modified Booth Multiplier," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 12, no. 5, pp. 522–531, 2004.

[273] Z. Zhang and Y. He, "A Low-Error Energy-Efficient Fixed-Width Booth Multiplier With Sign-Digit-Based Conditional Probability Estimation," *IEEE Transactions on Circuits and Systems II: Express Briefs*, pp. 236–240, 2018.

[274] V. Leon, K. Pekmestzi, and D. Soudris, "Exploiting the Potential of Approximate Arithmetic in DSP & AI Hardware Accelerators," in *International Conference on Field-Programmable Logic and Applications (FPL)*, 2021, pp. 263–264.

[275] V. Leon, I. Stratakos, G. Armeniakos, G. Lentaris, and D. Soudris, "ApproxQAM: High-Order QAM Demodulation Circuits with Approximate Arithmetic," in *International Conference on Modern Circuits and Systems Technologies (MOCAST)*, 2021, pp. 1–5.

[276] G. Lentaris, G. Chatzitsompanis, V. Leon, K. Pekmestzi, and D. Soudris, "Combining Arithmetic Approximation Techniques for Improved CNN Circuit Design," in *IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, 2020, pp. 1–4.

[277] V. Leon, G. Makris, S. Xydis, K. Pekmestzi, and D. Soudris, "MAx-DNN: Multi-Level Arithmetic Approximation for Energy-Efficient DNN Hardware Accelerators," in *IEEE Latin American Symposium on Circuits and Systems (LASCAS)*, 2022, pp. 1–4.

[278] J. Fowers, G. Brown, P. Cooke, and G. Stitt, "A Performance and Energy Comparison of FPGAs, GPUs, and Multicores for Sliding-Window Applications," in *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2012, p. 47–56.

[279] I. Kuon and J. Rose, "Measuring the Gap Between FPGAs and ASICs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 203–215, 2007.

[280] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," *ArXiv*, vol. abs/1409.1556, pp. 1–14, 2014.

[281] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," *Communications of the ACM*, vol. 60, no. 6, pp. 84–90, 2012.

[282] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh, "From High-Level Deep Neural Models to FPGAs," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–12.

[283] K. Guo, L. Sui, J. Qiu, S. Yao, S. Han, Y. Wang, and H. Yang, "Angel-Eye: A Complete Design Flow for Mapping CNN onto Customized Hardware," *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pp. 24–29, 2016.

[284] K. Abdelouahab, M. Pelcat, J. Sérot, C. Bourrasset, and F. Berry, "Tactics to Directly Map CNN Graphs on Embedded FPGAs," *IEEE Embedded Systems Letters*, vol. 9, no. 4, pp. 113–116, 2017.

[285] J. Wang, J. Lin, and Z. Wang, "Efficient Hardware Architectures for Deep Convolutional Neural Network," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 65, no. 6, pp. 1941–1953, 2018.

[286] T. Yuan, W. Liu, J. Han, and F. Lombardi, "High Performance CNN Accelerators Based on Hardware and Algorithm Co-Optimization," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 68, no. 1, pp. 250–263, 2021.

[287] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Dian-Nao: A Small-Footprint High-Throughput Accelerator for Ubiquitous Machine-Learning," in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014, pp. 269–284.

[288] Y.-J. Lin and T. S. Chang, "Data and Hardware Efficient Design for Convolutional Neural Network," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 65, no. 5, pp. 1642–1651, 2018.

[289] Y. Wang, J. Lin, and Z. Wang, "FPAP: A Folded Architecture for Energy-Quality Scalable Convolutional Neural Networks," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 66, no. 1, pp. 288–301, 2019.

[290] D. Han, J. Lee, J. Lee, and H.-J. Yoo, "A Low-Power Deep Neural Network Online Learning Processor for Real-Time Object Tracking Application," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 66, no. 5, pp. 1794–1804, 2019.

[291] S. Jin, J. Cho, X. D. Pham, K. M. Lee, S.-K. Park, M. Kim, and J. W. Jeon, "FPGA Design and Implementation of a Real-Time Stereo Vision System," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 20, no. 1, pp. 15–26, 2010.

[292] L. Puglia, M. Vigliar, and G. Raiconi, "Real-Time Low-Power FPGA Architecture for Stereo Vision," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 64, no. 11, pp. 1307–1311, 2017.

[293] T. M. Khan, D. G. Bailey, M. A. U. Khan, and Y. Kong, "Efficient Hardware Implementation For Fingerprint Image Enhancement Using Anisotropic Gaussian Filter," *IEEE Transactions on Image Processing*, vol. 26, no. 5, pp. 2116–2126, 2017.

[294] H. Wang, Y. Ji, Y. Shen, W. Song, M. Li, X. You, and C. Zhang, "An Efficient Detector for Massive MIMO Based on Improved Matrix Partition," *IEEE Transactions on Signal Processing*, vol. 69, pp. 2971–2986, 2021.

[295] T. H. Pham, S. A. Fahmy, and I. V. McLoughlin, "An End-to-End Multi-Standard OFDM Transceiver Architecture Using FPGA Partial Reconfiguration," *IEEE Access*, vol. 5, pp. 21 002–21 015, 2017.

[296] I. Hong, I. Frosio, J. Clemons, B. Khailany, R. Venkatesan, and S. W. Keckler, "A Real-Time Energy-Efficient Superpixel Hardware Accelerator for Mobile Computer Vision Applications," in *Design Automation Conference (DAC)*, 2016, pp. 1–6.

[297] F. Min, H. Xu, Y. Wang, Y. Wang, J. Li, X. Zou, B. Li, and Y. Han, "Dadu-Eye: A 5.3 TOPS/W, 30 fps/1080p High Accuracy Stereo Vision Accelerator," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 68, no. 10, pp. 4207–4220, 2021.

[298] O. Castañeda, S. Jacobsson, G. Durisi, T. Goldstein, and C. Studer, "High-Bandwidth Spatial Equalization for mmWave Massive MU-MIMO With Processing-in-Memory," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 67, no. 5, pp. 891–895, 2020.

[299] D. Markert, X. Yu, H. Heimpel, and G. Fischer, "An All-Digital, Single-Bit RF Transmitter for Massive MIMO," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 64, no. 3, pp. 696–704, 2017.

[300] M. Gao, Q. Wang, A. S. K. Nagendra, and G. Qu, "A Novel Data Format for Approximate Arithmetic Computing," in *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2017, pp. 390–395.

[301] U. Köster, T. J. Webb, X. Wang, M. Nassar, A. K. Bansal, W. H. Constable, O. H. Elibol, S. Gray, S. Hall, L. Hornof, A. Khosrowshahi, C. Kloss, R. J. Pai, and N. Rao, "Flexpoint: An Adaptive Numerical Format for Efficient Training of Deep Neural Networks," in *International Conference on Neural Information Processing Systems (NIPS)*, 2017, pp. 1740–1750.

[302] I. Stratakos, V. Leon, G. Armeniakos, G. Lentaris, and D. Soudris, "Design Space Exploration on High-Order QAM Demodulation Circuits: Algorithms, Arithmetic and Approximation Techniques," *MDPI Electronics*, vol. 11, no. 1, pp. 1–14, 2022.

[303] V. Mrazek, Z. Vasicek, L. Sekanina, M. A. Hanif, and M. Shafique, "ALWANN: Automatic Layer-Wise Approximation of Deep Neural Network Accelerators without Retraining," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2019, pp. 1–8.

[304] J. Wu, C. Leng, Y. Wang, Q. Hu, and J. Cheng, "Quantized Convolutional Neural Networks for Mobile Devices," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 4820–4828.

[305] S. Anwar, K. Hwang, and W. Sung, "Structured Pruning of Deep Convolutional Neural Networks," *ACM Journal on Emerging Technologies in Computing Systems*, vol. 13, no. 3, pp. 1–18, 2017.

[306] X. Lian, Z. Liu, Z. Song, J. Dai, W. Zhou, and X. Ji, "High-Performance FPGA-Based CNN Accelerator With Block-Floating-Point Arithmetic," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 8, pp. 1874–1885, 2019.

[307] D. D. Lin, S. S. Talathi, and V. S. Annapureddy, "Fixed Point Quantization of Deep Convolutional Networks," in *International Conference on Machine Learning*, 2016, pp. 2849–2858.

[308] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.

[309] J. Hamkins, "Performance of Low-Density Parity-Check Coded Modulation," in *IEEE Aerospace Conference*, 2010, pp. 1–14.

[310] A. Lavin and S. Gray, "Fast Algorithms for Convolutional Neural Networks," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 4013–4021.

[311] N. Kanopoulos, N. Vasanthavada, and R. Baker, "Design of an Image Edge Detection Filter Using the Sobel Operator," *IEEE Journal of Solid-State Circuits*, vol. 23, no. 2, pp. 358–367, 1988.

[312] D. Bailey, *Design for Embedded Image Processing on FPGAs*, 1st ed. John Wiley & Sons, 2011.

[313] Kaggle, *Ships in Satellite Imagery*, (accessed October 2022). [Online]. Available: https://www.kaggle.com/rhammell/ships-in-satellite-imagery

[314] Y. LeCun, C. Cortes, and C. Burges, *MNIST Handwritten Digit Database*, (accessed October 2022). [Online]. Available: http://yann.lecun.com/exdb/mnist/

[315] A. Krizhevsky, V. Nair, and G. Hinton, *CIFAR-10 Dataset*, (accessed October 2022). [Online]. Available: https://www.cs.toronto.edu/~kriz/cifar.html

[316] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *IEEE International Symposium on Workload Characterization (IISWC)*, 2009, pp. 44–54.

[317] S. Na, L. Xumin, and G. Yong, "Research on K-Means Clustering Algorithm: An Improved k-means Clustering Algorithm," in *International Symposium on Intelligent Information Technology and Security Informatics*, 2010, pp. 63–67.

[318] G. Strang, *Introduction to Linear Algebra*, 5th ed.   Wellesley-Cambridge Press, 2016.

[319] V. Leon, I. Stamoulias, G. Lentaris, D. Soudris, D. Gonzalez-Arjona, R. Domingo, D. M. Codinachs, and I. Conway, "Development and Testing on the European Space-Grade BRAVE FPGAs: Evaluation of NG-Large Using High-Performance DSP Benchmarks," *IEEE Access*, vol. 9, pp. 131 877–131 892, 2021.

[320] C. Marantos, L. Papadopoulos, A.-A. Tsintzira, A. Ampatzoglou, A. Chatzigeorgiou, and D. Soudris, "Decision Support for GPU Acceleration by Predicting Energy Savings and Programming Effort," *Elsevier Sustainable Computing: Informatics and Systems*, vol. 34, pp. 1–13, 2022.

[321] C. Karakasis, K. Machairas, C. Marantos, I. S. Paraskevas, E. Papadopoulos, and D. Soudris, "Exploiting the SoC FPGA Capabilities in the Control Architecture of a Quadruped Robot," in *IEEE/ASME International Conference on Advanced Intelligent Mechatronics (AIM)*, 2020, pp. 501–507.

[322] A. Pérez, A. Rodríguez, A. Otero, D. González-Arjona, A. Jiménez-Peralo, M. A. Verdugo, and E. De La Torre, "Run-Time Reconfigurable MPSoC-Based On-Board Processor for Vision-Based Space Navigation," *IEEE Access*, vol. 8, pp. 59 891–59 905, 2020.

[323] X. Iturbe, D. Keymeulen, E. Ozer, P. Yiu, D. Berisford, K. Hand, and R. Carlson, "An integrated SoC for Science Data Processing in Next-Generation Space Flight Instruments Avionics," in *IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, 2015, pp. 134–141.

[324] D. Rudolph, C. M. Wilson, J. Stewart, P. Gauvin, A. D. George, H. Lam, G. Crum, M. J. Wirthlin, A. Wilson, and A. Stoddard, "CSP: A Multifaceted Hybrid Architecture for Space Computing," in *AIAA/USU Conference on Small Satellites*, 2014, pp. 1–7.

[325] K. Maragos, V. Leon, G. Lentaris, D. Soudris, D. Gonzalez-Arjona, R. Domingo, A. Pastor, D. M. Codinachs, and I. Conway, "Evaluation Methodology and Reconfiguration Tests on the New European NG-MEDIUM FPGA," in

NASA/ESA Conference on Adaptive Hardware and Systems (AHS), 2018, pp. 127–134.

[326] G. Lentaris, I. Stratakos, I. Stamoulias, D. Soudris, M. Lourakis, and X. Zabulis, "High-Performance Vision-Based Navigation on SoC FPGA for Spacecraft Proximity Operations," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 30, no. 4, pp. 1188–1202, 2020.

[327] G. Lentaris, I. Stamoulias, D. Soudris, and M. Lourakis, "HW/SW Codesign and FPGA Acceleration of Visual Odometry Algorithms for Rover Navigation on Mars," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 26, no. 8, pp. 1563–1577, 2016.

[328] G. Lentaris, D. Diamantopoulos, K. Siozios, D. Soudris, and M. A. Rodrigálvarez, "Hardware Implementation of Stereo Correspondence Algorithm for the ExoMars Mission," in *International Conference on Field Programmable Logic and Applications (FPL)*, 2012, pp. 667–670.

[329] G. Lentaris, K. Maragos, D. Soudris, X. Zabulis, and M. Lourakis, "Single- and Multi-FPGA Acceleration of Dense Stereo Vision for Planetary Rovers," *ACM Transactions on Embedded Computing Systems*, vol. 18, no. 2, pp. 1–27, 2019.

[330] C. Adams, A. Spain, J. Parker, M. Hevert, J. Roach, and D. Cotten, "Towards an Integrated GPU Accelerated SoC as a Flight Computer for Small Satellites," in *IEEE Aerospace Conference*, 2019, pp. 1–7.

[331] F. C. Bruhn, N. Tsog, F. Kunkel, O. Flordal, and I. Troxel, "Enabling Radiation Tolerant Heterogeneous GPU-based Onboard Data Processing in Space," *CEAS Space Journal*, vol. 12, pp. 551–564, 2020.

[332] J. E. Navarro, A. Samuelsson, H. Gingsjö, J. Barendt, A. Dunne, L. Buckley, D. Reisis, A. Kyriakos, E.-A. Papatheofanous, C. Bezaitis, P. Matthijs, J. P. Ramos, and D. Steenari, "High-Performance Compute Board – A Fault-Tolerant Module for On-Board Vision Processing," in *European Workshop on On-Board Data Processing (OBDP)*, 2021, pp. 1–7.

[333] S. Parkes, C. McClements, D. McLaren, B. Youssef, M. S. Ali, A. F. Florit, and A. G. Villafranca, "SpaceWire and SpaceFibre on the Microsemi RTG4 FPGA," in *IEEE Aerospace Conference*, 2016, pp. 1–8.

[334] Y. Barrios, A. J. Sánchez, L. Santos, and R. Sarmiento, "SHyLoC 2.0: A Versatile Hardware Solution for On-Board Data and Hyperspectral Image Compression on Future Space Missions," *IEEE Access*, vol. 8, pp. 54 269–54 287, 2020.

[335] A. Tsigkanos, N. Kranitis, D. Theodoropoulos, and A. Paschalis, "High-Performance COTS FPGA SoC for Parallel Hyperspectral Image Compression with CCSDS-123.0-B-1," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 11, pp. 2397–2409, 2020.

[336] M. Pignol, "COTS-based Applications in Space Avionics," in *Design, Automation & Test in Europe Conference (DATE)*, 2010, pp. 1213–1219.

[337] R. Glein, F. Rittner, A. Becher, D. Ziener, J. Frickel, J. Teich, and A. Heuberger, "Reliability of Space-Grade vs. COTS SRAM-based FPGA in N-Modular Redundancy," in *NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, 2015, pp. 1–8.

[338] BAE Systems, *RAD750 Processors*, (accessed October 2022). [Online]. Available: https://www.baesystems.com/en/product/radiation-hardened-electronics

[339] Microchip Technology, *AT697F*, (accessed October 2022). [Online]. Available: https://www.microchip.com/en-us/product/AT697F

[340] Xilinx, *Space-Grade Portofolio*, (accessed October 2022). [Online]. Available: https://www.xilinx.com/products/silicon-devices/fpga/rt-kintex-ultrascale.html#productTable

[341] Microsemi, *Rad-Tolerant FPGAs*, (accessed October 2022). [Online]. Available: https://www.microsemi.com/product-directory/fpga-soc/1640-rad-tolerant-fpgas

[342] Atmel, *Atmel Aerospace: Reprogrammable FPGA with Built-in SEU and SET Protections*, (accessed October 2022). [Online]. Available: http://ww1.microchip.com/downloads/en/DeviceDoc/4066I-FPGA-Reprogramable_US_051115_web.pdf

[343] NanoXplore, *RHBD FPGAs, SoCs, eFPGAs*, (accessed October 2022). [Online]. Available: https://www.nanoxplore.com/

[344] J. Le Mauff and E. Lepape, "From eFPGA Cores to RHBD System-On-Chip FPGA," in *Space FPGA Users Workshop (SEFUW)*, 2018, pp. 1–53.

[345] J. Le Mauff, "NX RHBD FPGA Solutions for OBDP Applications," in *European Workshop on On-board Data Processing (OBDP)*, 2019, pp. 1–34.

[346] Xilinx, *XQR Versal*, (accessed October 2022). [Online]. Available: https://www.xilinx.com/content/dam/xilinx/support/documentation/selection-guides/xqr-versal-product-selection-guide.pdf

[347] ESA and Roscosmos, *ExoMars 2022 Mission*, (accessed October 2022). [Online]. Available: https://directory.eoportal.org/web/eoportal/satellite-missions/e/exomars-2022

[348] T. Takashima, E. Ogawa, K. Asamura, and M. Hikishima, "Design of a Mission Network System using SpaceWire for Scientific Payloads Onboard the Arase Spacecraft," *Springer Earth, Planets and Space (EPS)*, vol. 70, pp. 1–9, 2018.

[349] F. Abazovic, *Mars Rover Perseverance Runs Xilinx*, (accessed October 2022). [Online]. Available: https://www.acanalysis.com/mars-rover-perseverance-runs-xilinx/

[350] NASA, *Mars 2020 Perseverance Landing Press Kit*, (accessed October 2022). [Online]. Available: https://www.jpl.nasa.gov/news/press_kits/mars_2020/landing/mission/spacecraft/perseverance_rover/

[351] D. Báscones, C. González, and D. Mozos, "FPGA Implementation of the CCSDS 1.2.3 Standard for Real-Time Hyperspectral Lossless Compression," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 11, no. 4, pp. 1158–1165, 2018.

[352] N. Kranitis, A. Tsigkanos, G. Theodorou, I. Sideris, and A. Paschalis, "A Single Chip Dependable and Adaptable Payload Data Processing Unit," in *IEEE International On-Line Testing Symposium (IOLTS)*, 2015, pp. 138–143.

[353] M. Bertolucci, F. Falaschi, R. Cassettari, D. Davalle, and L. Fanucci, "A Comprehensive Trade-off Analysis on the CCSDS 131.2-B-1 Extended Mod-Cod (SCCC-X) Implementation," in *Euromicro Conference on Digital System Design (DSD)*, 2020, pp. 126–132.

[354] J. Vidmar, P. Maillard, T. Jones, M. Sawant, G. Gambardella, and N. Fraser, "Space DPU: Constructing a Radiation-Tolerant, FPGA-based Platform for Deep Learning Acceleration on Space Payloads," in *European Workshop on On-Board Data Processing (OBDP)*, 2021, pp. 1–20.

[355] A. Severance, D. Nandi, and K. O'Neill, "Using the VectorBlox Software Development Kit to Create Programmable AI/ML Applications in Radiation-Tolerant RT PolarFire FPGAs," in *European Workshop on On-Board Data Processing (OBDP)*, 2021, pp. 1–8.

[356] K. Bravhar, V. Martins, L. Santos, and D. M. Codinachs, "BRAVE NG-MEDIUM FPGA Reconfiguration through SpaceWire: Example Use Case and Performance Analysis," in *NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, 2018, pp. 135–141.

[357] V. Leon, I. Stamoulias, G. Lentaris, D. Soudris, R. Domingo, M. Verdugo, D. Gonzalez-Arjona, D. Merodio Codinachs, and I. Conway, "Systematic Evaluation of the European NG-LARGE FPGA & EDA Tools for On-Board Processing," in *European Workshop on On-Board Data Processing (OBDP)*, 2021, pp. 1–8.

[358] Intel, *Cyclone III FPGA*, (accessed October 2022). [Online]. Available: https://www.intel.com/content/www/us/en/programmable/b/cyclone3.html

[359] V. Leon, G. Lentaris, E. Petrongonas, D. Soudris, G. Furano, A. Tavoularis, and D. Moloney, "Improving Performance-Power-Programmability in Space Avionics with Edge Devices: VBN on Myriad2 SoC," *ACM Transactions on Embedded Computing Systems*, vol. 20, no. 3, pp. 1–23, 2021.

[360] V. Leon, C. Bezaitis, G. Lentaris, D. Soudris, D. Reisis, E.-A. Papatheofanous, A. Kyriakos, A. Dunne, A. Samuelsson, and D. Steenari, "FPGA & VPU Co-Processing in Space Applications: Development and Testing with DSP/AI Benchmarks," in *IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, 2021, pp. 1–5.

[361] V. Leon, K. Pekmestzi, and D. Soudris, "Systematic Embedded Development and Implementation Techniques on Intel Myriad VPUs," in *IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, 2022, pp. 1–2.

[362] V. Leon, P. Minaidis, G. Lentaris, and D. Soudris, "Accelerating AI and Computer Vision for Satellite Pose Estimation on the Intel Myriad X Embedded SoC," *Elsevier Microprocessors and Microsystems*, vol. N/A, no. N/A, pp. 1–8, 2023, under submission.

[363] E. Petrongonas, V. Leon, G. Lentaris, and D. Soudris, "ParalOS: A Scheduling & Memory Management Framework for Heterogeneous VPUs," in *Euromicro Conference on Digital System Design (DSD)*, 2021, pp. 221–228.

[364] Intel, *Movidius Vision Processing Units*, (accessed October 2022). [Online]. Available: https://www.intel.com/content/www/us/en/products/details/processors/movidius-vpu.html

[365] A. D. George and C. M. Wilson, "Onboard Processing with Hybrid and Reconfigurable Computing on Small Satellites," *Proceedings of the IEEE*, vol. 106, no. 3, pp. 458–470, 2018.

[366] A. Crespo, A. Alonso, M. Marcos, J. A. de la Puente, and P. Balbastre, "Mixed Criticality in Control Systems," *IFAC Proceedings Volumes*, vol. 47, no. 3, pp. 12 261–12 271, 2014.

[367] S. Esposito and M. Violante, "System-Level Architecture for Mixed Criticality Applications on MPSoC: A Space Application," in *IEEE International Workshop on Metrology for AeroSpace (MetroAeroSpace)*, 2017, pp. 479–483.

[368] V. Leon, E. A. Papatheofanous, G. Lentaris, C. Bezaitis, N. Mastorakis, G. Bampilis, D. Reisis, and D. Soudris, "Combining Fault Tolerance Techniques and COTS SoC Accelerators for Payload Processing in Space," in *IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, 2022, pp. 1–6.

[369] L. Kosmidis, I. Rodriguez, Álvaro Jover, S. Alcaide, J. Lachaize, J. Abella, O. Notebaert, F. J. Cazorla, and D. Steenari, "GPU4S: Embedded GPUs in Space - Latest Project Updates," *Elsevier Microprocessors and Microsystems*, vol. 77, pp. 1–10, 2020.

[370] M. Benito, M. M. Trompouki, L. Kosmidis, J. D. Garcia, S. Carretero, and K. Wenger, "Comparison of GPU Computing Methodologies for Safety-Critical Systems: An Avionics Case Study," in *Design, Automation & Test in Europe (DATE)*, 2021, pp. 717–718.

[371] M. Lofqvist and J. Cano, "Accelerating Deep Learning Applications in Space," in *AIAA/USU Conference on Small Satellites*, 2020, pp. 1–19.

[372] G. Giuffrida, L. Fanucci, G. Meoni, M. Batič, L. Buckley, A. Dunne, C. Van Dijk, M. Esposito, J. Hefele, N. Vercruyssen, G. Furano, M. Pastena, and J. Aschbacher, "The Φ-Sat-1 Mission: the First On-Board Deep Neural Network Demonstrator for Satellite Earth Observation," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 60, pp. 1–14, 2022.

[373] J. E. Navarro, A. Samuelsson, H. Gingsjö, J. Barendt, A. Dunne, L. Buckley, D. Reisis, A. Kyriakos, E.-A. Papatheofanous, C. Bezaitis, P. Matthijs, J. P. Ramos, and D. Steenari, "High-Performance Compute Board – A Fault-Tolerant Module for On-Board Vision Processing," in *European Workshop on On-Board Data Processing (OBDP)*, 2021, pp. 1–7.

[374] S. Agarwal, E. Hervas-Martin, J. Byrne, A. Dunne, J. Luis Espinosa-Aranda, and D. Rijlaarsdam, "An Evaluation of Low-Cost Vision Processors for Efficient Star Identification," *MDPI Sensors*, vol. 20, no. 21, pp. 1–14, 2020.

[375] A. Dunne, "UB0100 AI & CV Compute Engine," in *ESA Workshop on Avionics, Data, Control and Software Systems (ADCSS)*, 2020, pp. 1–17. [Online]. Available: https://indico.esa.int/event/338/contributions/5696/attachments/4033/5938/1745

[376] E. Dunkel, J. L. Espinosa-Aranda, J. Romero-Canas, L. Buckley, Z. Towfic, F. Mirza, J. Swope, D. Russell, J. Sauvaeau, D. Sheldon, S. Chien, M. Fernandez, C. Knox, K. Wagstaff, S. Lu, M. Denbina, D. Atha, M. Swan, and M. Ono, "Benchmarking Machine Learning on The Myriad X Processor Onboard the ISS," in *International Space Station Research and Development Conference*, 2021, pp. 1–24. [Online]. Available: https://ai.jpl.nasa.gov/public/papers/ISS_RD_2021_Movidius.pptx

[377] G. Furano, G. Meoni, A. Dunne, D. Moloney, V. Ferlet-Cavrois, A. Tavoularis, J. Byrne, L. Buckley, M. Psarakis, K.-O. Voss, and L. Fanucci, "Towards the Use of Artificial Intelligence on the Edge in Space Systems: Challenges and Opportunities," *IEEE Aerospace and Electronic Systems Magazine*, vol. 35, no. 12, pp. 44–56, 2020.

[378] M. Lourakis and X. Zabulis, "Model-based Visual Tracking of Orbiting Satellites Using Edges," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2017, pp. 3791–3796.

[379] P. F. Proença and Y. Gao, "Deep Learning for Spacecraft Pose Estimation from Photorealistic Rendering," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2020, pp. 6007–6013.

[380] A. Yazdanbakhsh, K. K. Seshadri, B. Akin, J. Laudon, and R. Narayanaswami, "An Evaluation of Edge TPU Accelerators for Convolutional Neural Networks," *ArXiv*, vol. abs/2102.10423, pp. 1–11, 2021.

[381] K.-C. Hsu and H.-W. Tseng, "Accelerating Applications Using Edge Tensor Processing Units," in *ACM/IEEE SC, International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–14.

[382] V. Leon, G. Lentaris, D. Soudris, S. Vellas, and M. Bernou, "Towards Employing FPGA and ASIP Acceleration to Enable Onboard AI/ML in Space Applications," in *IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, 2022, pp. 1–4.

[383] J. Goodwill, G. Crum, J. MacKinnon, C. Brewer, M. Monaghan, T. Wise, and C. Wilson, "NASA SpaceCube Edge TPU SmallSat Card for Autonomous Operations and Onboard Science-Data Analysis," in *AIAA/USU Conference on Small Satellites*, 2021, pp. 1–13.

[384] A. Xygkis, L. Papadopoulos, D. Moloney, D. Soudris, and S. Yous, "Efficient Winograd-based Convolution Kernel Implementation on Edge Devices," in *ACM/ESDA/IEEE Design Automation Conference (DAC)*, 2018, pp. 1–6.

[385] F. Tsimpourlas, L. Papadopoulos, A. Bartsokas, and D. Soudris, "A Design Space Exploration Framework for Convolutional Neural Networks Implemented on Edge Devices," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2212–2221, 2018.

[386] X. Xu, J. Amaro, S. Caulfield, A. Forembski, G. Falcao, and D. Moloney, "Convolutional Neural Network on Neural Compute Stick for Voxelized Point-Clouds Classification," in *International Congress on Image and Signal Processing, BioMedical Engineering and Informatics (CISP-BMEI)*, 2017, pp. 1–7.

[387] J. Hochstetler, R. Padidela, Q. Chen, Q. Yang, and S. Fu, "Embedded Deep Learning for Vehicular Edge Computing," in *IEEE/ACM Symposium on Edge Computing (SEC)*, 2018, pp. 341–343.

[388] C. Marantos, N. Karavalakis, V. Leon, V. Tsoutsouras, K. Pekmestzi, and D. Soudris, "Efficient Support Vector Machines Implementation on Intel/Movidius Myriad 2," in *International Conference on Modern Circuits and Systems Technologies (MOCAST)*, 2018, pp. 1–4.

[389] L. Puglia, M. Ionică, G. Raiconi, and D. Moloney, "Passive Dense Stereo Vision on the Myriad2 VPU," in *IEEE Hot Chips Symposium (HCS)*, 2016, pp. 1–5.

[390] Intel, *OpenVINO Toolkit Overview*, (accessed October 2022). [Online]. Available: https://docs.openvino.ai/2021.2/index.html

[391] J. Canny, "A Computational Approach to Edge Detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-8, no. 6, pp. 679–698, 1986.

[392] STAR-Dundee, *SpaceWire EGSE*, (accessed October 2022). [Online]. Available: http://www.star-dundee.com/products/spacewire-egse-and-device-simulator-mk2/#product_features

[393] Y. Huang, Y. Bai, R. Li, and X. Huang, "Research of Canny Edge Detection Algorithm on Embedded CPU and GPU Heterogeneous Systems," in *International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery (ICNC-FSKD)*, 2016, pp. 647–651.

Publications

Journals

J10. **V. Leon**, M. A. Hanif, X. Jiao, M. Shafique, K. Pekmestzi, and D. Soudris, "Approximate Computing Survey: A Taxonomy of Software, Hardware & Architectural Techniques and Applications", ***ACM Journal on Emerging Technologies in Computing Systems***
DOI: under submission

J9. **V. Leon**, P. Minaidis, G. Lentaris, and D. Soudris, "Accelerating AI and Computer Vision for Satellite Pose Estimation on the Intel Myriad X Embedded SoC", ***Elsevier Microprocessors and Microsystems***
DOI: under submission

J8. I. Stratakos, **V. Leon**, G. Armeniakos, G. Lentaris, and D. Soudris, "Design Space Exploration on High-Order QAM Demodulation Circuits: Algorithms, Arithmetic and Approximation Techniques", ***MDPI Electronics***, vol. 11, no. 1, 2022
DOI: 10.3390/electronics11010039

J7. **V. Leon**, I. Stamoulias, G. Lentaris, D. Soudris, D. Gonzalez-Arjona, R. Domingo, D. Merodio Codinachs, and I. Conway, "Development and Testing on the European Space-Grade BRAVE FPGAs: Evaluation of NG-Large Using High-Performance DSP Benchmarks", ***IEEE Access***, vol. 9, 2021
DOI: 10.1109/ACCESS.2021.3114502

J6. **V. Leon**, T. Paparouni, E. Petrongonas, D. Soudris, and K. Pekmestzi, "Improving Power of DSP and CNN Hardware Accelerators using Approximate Floating-Point Multipliers", ***ACM Transactions on Embedded Computing***, vol. 20, no. 5, 2021
DOI: 10.1145/3448980

J5. **V. Leon**, G. Lentaris, E. Petrongonas, D. Soudris, G. Furano, A. Tavoularis, and D. Moloney, "Improving Performance-Power-Programmability in Space Avionics with Edge Devices: VBN on Myriad2 SoC", ***ACM Transactions***

on Embedded Computing, vol. 20, no. 3, 2021
DOI: [10.1145/3440885](10.1145/3440885)

J4. **V. Leon**, S. Mouselinos, K. Koliogeorgi, S. Xydis, D. Soudris, and K. Pekmestzi, "A TensorFlow Extension Framework for Optimized Generation of Hardware CNN Inference Engines", *MDPI Technologies*, vol. 8, no. 1, 2020
DOI: 10.3390/technologies8010006

J3. **V. Leon**, S. Xydis, D. Soudris and K. Pekmestzi, "Energy-Efficient VLSI Implementation of Multipliers with Double LSB Operands", *IET Circuits, Devices & Systems*, vol. 13, no. 6, 2019
DOI: [10.1049/iet-cds.2018.5039](10.1049/iet-cds.2018.5039)

J2. **V. Leon**, G. Zervakis, S. Xydis, D. Soudris and K. Pekmestzi, "Walking through the Energy-Error Pareto Frontier of Approximate Multipliers", *IEEE Micro*, vol. 38, no. 4, 2018
DOI: [10.1109/MM.2018.043191124](10.1109/MM.2018.043191124)

J1. **V. Leon**, G. Zervakis, D. Soudris and K. Pekmestzi, "Approximate Hybrid High Radix Encoding for Energy Efficient Inexact Multipliers", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 3, 2018
DOI: 10.1109/TVLSI.2017.2767858

Conferences

C14. **V. Leon**, G. Lentaris, D. Soudris, S. Vellas, and M. Bernou, "Towards Employing FPGA and ASIP Acceleration to Enable Onboard AI/ML in Space Applications", *IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, 2022
DOI: 10.1109/VLSI-SoC54400.2022.9939566

C13. **V. Leon**, E.-A. Papatheofanous, G. Lentaris, C. Bezaitis, N. Mastorakis, G. Bampilis, D. Reisis, and D. Soudris, "Combining Fault Tolerance Techniques and COTS SoC Accelerators for Payload Processing in Space", *IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, 2022
DOI: 10.1109/VLSI-SoC54400.2022.9939621

C12. <u>**V. Leon**</u>, K. Pekmestzi, and D. Soudris, "Systematic Embedded Development and Implementation Techniques on Intel Myriad VPUs", ***IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)***, 2022
DOI: 10.1109/VLSI-SoC54400.2022.9939592

C11. <u>**V. Leon**</u>, G. Makris, S. Xydis, K. Pekmestzi, and D. Soudris, "MAx-DNN: Multi-Level Arithmetic Approximation for Energy-Efficient DNN Hardware Accelerators", ***IEEE Latin American Symposium on Circuits and Systems (LASCAS)***, 2022
DOI: 10.1109/LASCAS53948.2022.9789055

C10. <u>**V. Leon**</u>, C. Bezaitis, G. Lentaris, D. Soudris, D. Reisis, E.-A. Papatheofanous, A. Kyriakos, A. Dunne, A. Samuelsson, and D. Steenari, "FPGA & VPU Co-Processing in Space Applications: Development and Testing with DSP/AI Benchmarks", ***IEEE International Conference on Electronics, Circuits and Systems (ICECS)***, 2021
DOI: 10.1109/ICECS53924.2021.9665462

C9. <u>**V. Leon**</u>, K. Pekmestzi, and D. Soudris, "Exploiting the Potential of Approximate Arithmetic in DSP & AI Hardware Accelerators", ***International Conference on Field Programmable Logic and Applications (FPL)***, 2021
DOI: [10.1109/FPL53798.2021.00049](10.1109/FPL53798.2021.00049)

C8. E. Petrongonas, <u>**V. Leon**</u>, G. Lentaris, and D. Soudris, "ParalOS: A Scheduling & Memory Management Framework for Heterogeneous VPUs", ***Euromicro Conference on Digital System Design (DSD)***, 2021
DOI: [10.1109/DSD53832.2021.00043](10.1109/DSD53832.2021.00043)

C7. <u>**V. Leon**</u>, I. Stratakos, G. Armeniakos, G. Lentaris, and D. Soudris, "ApproxQAM: High-Order QAM Demodulation Circuits with Approximate Arithmetic", ***International Conference on Modern Circuits and Systems Technologies (MOCAST)***, 2021
DOI: 10.1109/MOCAST52088.2021.9493421

C6. <u>**V. Leon**</u>, I. Stamoulias, G. Lentaris, D. Soudris, R. Domingo, M. Verdugo, D. Gonzalez-Arjona, D. Merodio Codinachs, and I. Conway, "Systematic Evaluation of the European NG-LARGE FPGA & EDA Tools for On-Board Processing", ***ESA/CNES/DLR European Workshop on On-Board Data Processing (OBDP)***, 2021
DOI: 10.5281/zenodo.5521055

C5.   G. Lentaris, G. Chatzitsompanis, **<u>V. Leon</u>**, K. Pekmestzi, and D. Soudris, "Combining Arithmetic Approximation Techniques for Improved CNN Circuit Design", ***IEEE International Conference on Electronics, Circuits and Systems (ICECS)***, 2020
DOI: [10.1109/ICECS49266.2020.9294869](https://doi.org/10.1109/ICECS49266.2020.9294869)

C4.   **<u>V. Leon</u>**, K. Asimakopoulos, S. Xydis, D. Soudris, and K. Pekmestzi, "Cooperative Arithmetic-Aware Approximation Techniques for Energy-Efficient Multipliers", ***ACM/IEEE Design Automation Conference (DAC)***, 2019
DOI: [10.1145/3316781.3317793](https://doi.org/10.1145/3316781.3317793)

C3.   S. Mouselinos, **<u>V. Leon</u>**, S. Xydis, D. Soudris, and K. Pekmestzi, "TF2FPGA: A Framework for Projecting and Accelerating TensorFlow CNNs on FPGA Platforms", ***International Conference on Modern Circuits and Systems Technologies (MOCAST)***, 2019
DOI: [10.1109/MOCAST.2019.8741940](https://doi.org/10.1109/MOCAST.2019.8741940)

C2.   K. Maragos, **<u>V. Leon</u>**, G. Lentaris, D. Soudris, D. Gonzalez-Arjona, R. Domingo, A. Pastor, D. Merodio Codinachs, and I. Conway, "Evaluation Methodology and Reconfiguration Tests on the New European NG-MEDIUM FPGA", ***NASA/ESA Conference on Adaptive Hardware and Systems (AHS)***, 2018
DOI: [10.1109/AHS.2018.8541492](https://doi.org/10.1109/AHS.2018.8541492)

C1.   C. Marantos, N. Karavalakis, **<u>V. Leon</u>**, V. Tsoutsouras, K. Pekmestzi, and D. Soudris, "Efficient Support Vector Machines Implementation on Intel/Movidius Myriad 2", ***International Conference on Modern Circuits and Systems Technologies (MOCAST)***, 2018
DOI: [10.1109/MOCAST.2018.8376630](https://doi.org/10.1109/MOCAST.2018.8376630)

Curriculum Vitae

Vasileios (Vasilis) Leon received the Diploma (M.Eng.) degree from the Department of Computer Engineering and Informatics, University of Patras (UoP) in 2016. During 2015-2016, he worked as junior DSP Engineer (started as internship) in Think Silicon, now an Applied Materials company. In 2016, he was admitted to the Ph.D. program of the School of Electrical and Computer Engineering, National Technical University of Athens (NTUA), where he joined the Microprocessors and Digital Systems Laboratory (MicroLab) to pursue his Doctoral Dissertation in the research field of computer hardware and embedded systems. From 2017, he is working as self-employed Researcher & Developer in projects of the European Space Agency (ESA) with embedded computing platforms, SoCs, co-processing architectures, and DSP/AI algorithms. Up to date, he has participated in the following research projects:

- **CAIRS21** (4000135491/21/NL/GLC/ov): *"COTS AI Accelerators in Mixed-Criticality High-Performance Avionics for Reconfigurable Satellites: TPU versus Prominent Embedded Devices, Mitigation Techniques, SW Frameworks and AI/ML Model Upload"*, 2021–2022

- **QUEENS3** (4000134874/21/NL/AR/va): *"Quality Assessment Of The New European Ultra BRAVE FPGA Software Tools"*, 2021–2023

- **HPCB** (num. 4000126129/18/NL/AF): *"FPGA Accelerated DSP Payload Data Processor Board"*, 2019–2021

- **QUEENS2** (4000128041/19/NL/AR/va): *"Quality Assessment Of The New European Large BRAVE FPGA Software Tools"*, 2019–2020

- **LEOTOME** (4000126083/18/NL/FE): *"Demonstration of Visual Based Navigation Algorithms on Myriad2 Processor"*, 2019

- **QUEENS1** (4000119331/17/NL/PS): *"Quality Assessment Of The New European Medium BRAVE FPGA Software Tools"*, 2017–2018

During his Ph.D. studies, Vasileios published **24 papers** in 14 international peer-reviewed conferences and 10 journals, including publications in the Design Automation Conference and IEEE/ACM Transactions journals. He was also a reviewer in various IEEE journals and conferences. Moreover, he co-supervised 17 undergraduate Diploma (M.Eng.) theses and offered teaching assistance in the laboratories of

the undergraduate courses *Microprocessors*, *Digital VLSI Systems* and *Introduction to VLSI.*

Research Interests

digital circuit design, hardware accelerators, FPGAs, embedded systems, approximate computing, computer arithmetic, DSP, computer vision.

Personal Web Profiles

linkedin.com/vasileiosleon

scholar.google.com/vasileiosleon

researchgate.net/vasileiosleon

NATIONAL TECHNICAL UNIVERSITY OF ATHENS
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING
DIVISION OF COMPUTER SCIENCE
MICROPROCESSORS AND DIGITAL SYSTEMS LABORATORY
9 Heroon Polytechniou, Zografou Campus
15780 Athens, Greece
Ph.D. Dissertation, Vasileios Leon