NATIONAL TECHNICAL UNIVERSITY OF ATHENS
SCHOOL OF ELECTRICAL & COMPUTER ENGINEERING
DIVISION OF COMPUTER SCIENCE

Hardware-Software Co-Design of Deep Learning Accelerators: From Custom to Automated Design Methodologies

Ph.D. Thesis

Dimitrios N. Danopoulos

Athens, September 2024

NATIONAL TECHNICAL UNIVERSITY OF ATHENS
SCHOOL OF ELECTRICAL & COMPUTER ENGINEERING
DIVISION OF COMPUTER SCIENCE

Hardware-Software Co-Design of Deep Learning Accelerators: From Custom to Automated Design Methodologies

Ph.D. Thesis

Dimitrios N. Danopoulos

Supervising Committee: Dimitrios Soudris
Kiamal Pekmestzi
Dionisios Pnevmatikatos

Approved by the advisory committee on September, 2024.

....	....	....
Dimitrios Soudris	Kiamal Pekmestzi	Dionisios Pnevmatikatos
Professor N.T.U.A	Professor Emer. N.T.U.A	Professor N.T.U.A

....	....
Sotirios Xydis	Christoforos Kachris
Assistant Professor N.T.U.A	Assistant Professor UNIWA
....	....
Panayiotis Tsanakas	Georgios Zervakis
Professor N.T.U.A	Assistant Professor U.P.

....
Dimitrios N. Danopoulos
Ph.D. Electrical & Computer Engineer N.T.U.A.

The content of this Ph.D. Thesis does not reflect the official opinion of the National Technical University of Athens. Responsibility for the information and views expressed in this thesis lies entirely with the author.

Content that is reused from publications that the author has (co-)authored (excerpts, figures, tables, etc.) is under copyright with the respective paper publishers (IEEE, ACM, Springer etc) and is cited accordingly in the current text. Content that is reused from third-party publications appears with the appropriate copyright note. Reuse of any such content by any interested party requires the publishers' prior consent, according to the applicable copyright policies. Content that has not been published before is copyrighted jointly as follows:

Copyright ©Dimitrios N. Danopoulos, 2024
Ph.D. Electrical & Computer Engineer N.T.U.A.
All rights reserved

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ
ΥΠΟΛΟΓΙΣΤΩΝ

Συσχεδίαση Υλικού και Λογισμικού για Επιταχυντές Βαθιάς Μάθησης: Από Προσαρμοσμένες σε Αυτοματοποιημένες Μεθοδολογίες Σχεδίασης

ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

Δημήτριος Ν. Δανόπουλος

Αθήνα, Σεπτέμβριος 2024

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Συσχεδίαση Υλικού και Λογισμικού για Επιταχυντές Βαθιάς Μάθησης: Από Προσαρμοσμένες σε Αυτοματοποιημένες Μεθοδολογίες Σχεδίασης

ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

Δημήτριος Ν. Δανόπουλος

Συμβουλευτική Επιτροπή : Δημήτριος Σούντρης
Κιαμάλ Πεκμεστζή
Διονύσιος Πνευματικάτος

Εγκρίθηκε από την επταμελή εξεταστική επιτροπή τον Σεπτέμβριο, 2024.

....
Δημήτριος Σούντρης Κιαμάλ Πεκμεστζή Διονύσιος Πνευματικάτος
Καθηγητής ΕΜΠ Ομότιμος Καθηγητής ΕΜΠ Καθηγητής ΕΜΠ

....
Σωτήριος Ξύδης Χριστόφορος Κάχρης
Επίκουρος Καθηγητής ΕΜΠ Επίκουρος Καθηγητής ΠΑΔΑ

....
Παναγιώτης Τσανάκας Γεώργιος Ζερβάκης
Καθηγητής ΕΜΠ Επίκουρος Καθηγητής ΠΠ

....
Δημήτριος Ν. Δανόπουλος
Διδάκτωρ Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Copyright © Δημήτριος Ν. Δανόπουλος, 2024
Με επιφύλαξη παντός δικαιώματος. All rights reserved.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα.
Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

“Those who have a 'why' to live, can bear with almost any 'how'.”
Viktor E. Frankl, Man’s Search for Meaning.

Abstract

In the past few years, Deep Learning (DL), a subset within the broader field of Artificial Intelligence (AI), has achieved remarkable success across a wide spectrum of applications, such as computer vision and autonomous systems. It has emerged as one of the most powerful and accurate techniques, often employing Deep Neural Networks (DNNs) that frequently surpass human performance. However, the ongoing AI research, heavily relies on high-performance systems to handle the vast amounts of computations and data involved. Advancements in technology and architecture have led to the integration of co-processors like Graphics Processing Units (GPUs) or Field Programmable Gate Arrays (FPGAs). These devices are types of hardware accelerators that play crucial roles in accelerating AI workloads and have facilitated the deployment but also the development of increasingly sophisticated AI models. To address the substantial computational demands of AI algorithms, such specialized hardware requires significant engineering effort to achieve optimal performance and efficiency. Moreover, the current state-of-the-art leverages approximate computing, an approach that permits computations to be less precise, trading off some precision for additional efficiency gains. However, evaluating the accuracy of approximate DNNs is cumbersome due to the lack of adequate support for approximate arithmetic in Deep Learning frameworks, such as PyTorch or Tensorflow.

In this dissertation, we first employ FPGAs as accelerators for a wide range of AI applications and present various strategies and techniques to improve the efficiency and performance for such applications. We also examine automated frameworks to convert trained neural network models into optimized FPGA firmware, alleviating the hardware development challenges faced by engineers in the process. Additionally, we investigate the use of approximate computing to leverage the intrinsic error resilience of DNN models. We address the challenge of evaluating approximate DNNs by introducing two frameworks, AdaPT and TransAxx. These frameworks, built on PyTorch and optimized using CPU and GPU hardware acceleration respectively, facilitate approximate inference and approximation-aware retraining for various DNN models, Last, we propose a hardware-driven Monte Carlo Tree Search (MCTS) algorithm to efficiently search the space of possible approximate configurations on Vision Transformer (ViT) models and achieve significant trade-offs between accuracy and power.

Keywords: AI, DNNs, PyTorch, Hardware Accelerator, Approximate Computing

Περίληψη

Τα τελευταία χρόνια, η Βαθιά Μάθηση (Deep Learning), ένα υποσύνολο στο ευρύτερο πεδίο της Τεχνητής Νοημοσύνης (AI), έχει σημειώσει αξιοσημείωτη επιτυχία σε ένα ευρύ φάσμα εφαρμογών, όπως η όραση υπολογιστών και τα αυτόνομα συστήματα. Έχει αναδειχθεί ως μια από τις πιο ισχυρές και ακριβείς τεχνικές, χρησιμοποιώντας συχνά Βαθιά Νευρωνικά Δίκτυα (DNNs) που πολλές φορές ξεπερνούν την ανθρώπινη απόδοση. Ωστόσο, η συνεχιζόμενη έρευνα της τεχνητής νοημοσύνης βασίζεται σε μεγάλο βαθμό σε υπολογιστικά συστήματα υψηλής απόδοσης για τη διαχείριση των τεράστιων όγκων υπολογισμών και δεδομένων που εμπλέκονται. Οι εξελίξεις στην τεχνολογία και την αρχιτεκτονική οδήγησαν στην ενσωμάτωση συνεπεξεργαστών όπως Graphics Processing Units (GPUs) ή Field Programmable Gate Arrays (FPGAs). Αυτές οι συσκευές είναι τύποι επιταχυντών υλικού που παίζουν κρίσιμο ρόλο στην επιτάχυνση του φόρτου εργασίας της τεχνητής νοημοσύνης και έχουν διευκολύνει την ανάπτυξη ολοένα και πιο εξελιγμένων μοντέλων τεχνητής νοημοσύνης. Για την αντιμετώπιση των σημαντικών υπολογιστικών απαιτήσεων των αλγορίθμων AI, τέτοιο εξειδικευμένο υλικό απαιτεί σημαντική προγραμματιστική προσπάθεια για την επίτευξη βέλτιστης απόδοσης και αποδοτικότητας. Επιπλέον, η τελευταία λέξη της τεχνολογίας αξιοποιεί την μέθοδο προσεγγιστικών υπολογισμών (approximate computing), μια μέθοδο που επιτρέπει στους υπολογισμούς να είναι λιγότερο ακριβείς, ανταλλάσσοντας κάποια ακρίβεια για κέρδη αποδοτικότητας. Ωστόσο, η αξιολόγηση της ακρίβειας των προσεγγιστικών νευρωνικών δικτύων είναι δύσκολη λόγω της έλλειψης επαρκούς υποστήριξης για την προσεγγιστική αριθμητική σε περιβάλλοντα Βαθιάς Μάθησης, όπως το PyTorch ή Tensorflow.

Στόχος της παρούσας διατριβής είναι, αρχικά, η χρήση των FPGAs ως επιταχυντές για ένα ευρύ φάσμα εφαρμογών τεχνητής νοημοσύνης, ενώ παρουσιάζονται διάφορες στρατηγικές και τεχνικές για τη βελτίωση της αποτελεσματικότητας και της απόδοσης για τέτοιες εφαρμογές. Προτείνονται επίσης διάφορες βελτιστοποιήσεις για αυτοματοποιημένα περιβάλλοντα με στόχο την μετατροπή εκπαιδευμένων μοντέλων νευρωνικών δικτύων σε βελτιστοποιημένο υλικολογισμικό για FPGA, μειώνοντας τις προκλήσεις ανάπτυξης υλικού που αντιμετωπίζουν οι μηχανικοί στην συγκεκριμένη διαδικασία. Επιπλέον, εξετάζεται η χρήση προσεγγιστικών υπολογισμών για να αξιοποιηθεί η εγγενής ανθεκτικότητα σφάλματος των Βαθιών Νευρωνικών Δικτύων. Αντιμετωπίζεται η δυσκολία στην αξιολόγηση προσεγγιστικών δικτύων, εισάγοντας για τον σκοπό αυτό δύο περιβάλλοντα, το AdaPT και το TransAxx. Αυτά τα περιβάλλοντα, που είναι βασισμένα στο PyTorch και είναι βελτιστοποιημένα με χρήση επιτάχυνσης CPU και GPU αντίστοιχα, επιτρέπουν προσεγγιστικό inference και retraining για διάφορα βαθιά νευρωνικά μοντέλα. Τέλος, προτείνεται ένας αλγόριθμος Monte Carlo

Tree Search (MCTS) που οδηγείται από υλικό για την αποτελεσματική αναζήτηση του χώρου των προσεγγιστικών παραμέτρων σε μοντέλα Vision Transformer (ViT) με στόχο την επίτευξη σημαντικών αντισταθμίσεων μεταξύ ακρίβειας και ισχύος.

Λέξεις Κλειδιά: Τεχνητή Νοημοσύνη, Βαθιά Νευρωνικά Δίκτυα, PyTorch, Επιταχυντής Υλικού, Προσεγγιστικός Υπολογισμός

Acknowledgments

Above all, I express my gratitude to Professor Dimitrios Soudris, the supervisor of this project, for entrusting me with this thesis. His continual support, guidance, and encouragement have proven invaluable throughout the entire journey. I am also sincerely thankful to Asst. Professor Christoforos Kachris, my advisor from the undergraduate thesis to the first years of this Ph.D. thesis, for his firm support and inspiration. Additionally, I wish to express my sincere thanks to Asst. Professors George Lentaris, Kostas Siozios and Georgios Zervakis for their invaluable guidance throughout the final progression of my research. Special gratitude is also extended to the entire staff of the Microprocessors and Digital Systems Laboratory (MicroLab) of NTUA. Last but not least, I would like to thank my entire family for their constant encouragement, motivation and unconditional support throughout my university studies.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

AI	Artificial Intelligence
ANN	Artificial Neural Network
ASIC	Application-Specific Integrated Circuit
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DDR	Double Data Rate
DNN	Deep Neural Network
DRAM	Dynamic Random-Access Memory
DSP	Digital Signal Processing
FLOPs	Floating-point Operations Per Second
FPGA	Field-Programmable Gate Array
FPS	Frames Per Second
GAN	Generative Adversarial Network
GPU	Graphics Processing Unit
HBM	High Bandwidth Memory
HPC	High-Performance Computing
IoT	Internet of Things
MLP	Multi-Layer Perceptron
MSB	Most Significant Bit
MSE	Mean Squared Error
NN	Neural Network
OpenCL	Open Computing Language
OPS	Operations Per Second
PCIe	Peripheral Component Interconnect Express
RAM	Random Access Memory
ReLU	Rectified Linear Unit
RL	Reinforcement Learning
RMSE	Root Mean Squared Error
SGD	Stochastic Gradient Descent
SIMD	Single Instruction Multiple Data
SoC	System-on-Chip
SoTA	State of The Art
TPU	Tensor Processing Unit
VAE	Variational Autoencoder
ViT	Vision Transformer

1 Introduction

1.1 Motivation for the Research

The motivation for this research is rooted in the increasing demand for efficient artificial intelligence (AI) processing. As the use of AI technologies continues to grow across a wide range of industries, it has become increasingly important to develop hardware and software solutions that can handle the computational demands of these applications. This has led to a growing interest in the use of hardware accelerators, such as graphics processing units (GPUs) and field-programmable gate arrays (FPGAs), which can significantly improve the speed and energy efficiency of AI processing.

To gain a deeper understanding of the motivation behind this work, it is important to trace its origins and foundational elements. Over the past six decades, Moore's Law has played a pivotal role in driving the trajectory of computing. Throughout this extended period, the industry's solid emphasis on transistor scaling, a key aspect of Moore's Law, has consistently yielded increased transistor performance and density. While it might be premature to definitively declare the demise of Moore's Law, there are indications suggesting that we have encountered the physical constraints inherent in silicon-based CPUs (Central Processing Units).

"Moore's law is the observation that the number of transistors in a dense integrated circuit doubles approximately every two years. The period is often quoted as 18 months because of Intel executive David House, who predicted that chip performance would double every 18 months. — G. E. Moore, 1965

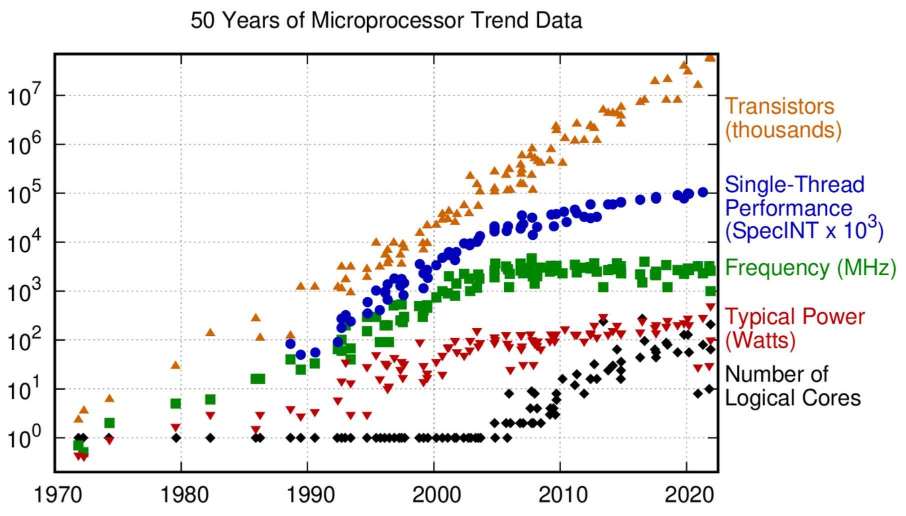


Figure 1-1. 50 years of microprocessor trend data [1].

As observed from the above figure, the exponential growth in computing power predicted by Moore's Law is slowing down. This trend poses a challenge to the continued advancement of AI, as the increased computational power is crucial for handling the complex calculations and vast datasets, especially involved in Deep Learning tasks. The traditional approach of relying solely on general-purpose CPUs for AI workloads has become increasingly inefficient and unsustainable. CPUs, designed for a wide range of tasks, are not optimized for the specific computational demands of AI algorithms, leading to suboptimal performance and high energy consumption. In Figure 1-2, we show the enormous computational demands needed for training recent Deep Learning models.

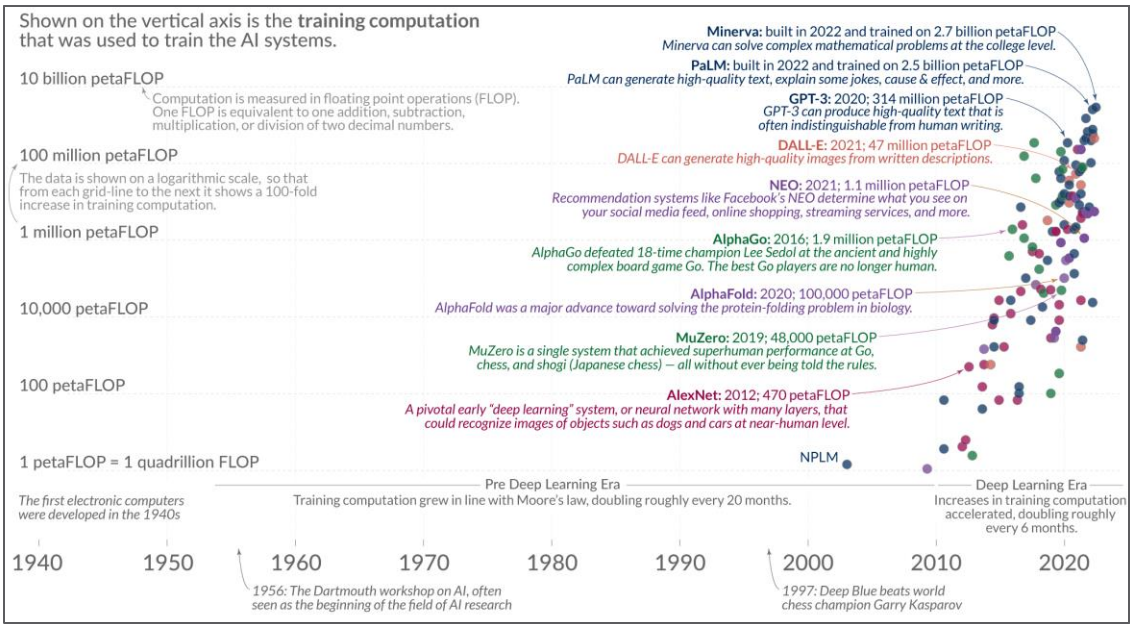


Figure 1-2. Training computation for popular Deep Learning models [2].

It is evident that CPUs cannot keep up with the pace of the computational demands of AI and especially Deep Learning. Alternative computing architectures and specialized processors for AI are being developed ensuring an uninterrupted progression of the AI research. Despite the integration of AI accelerators to enhance computational performance, there always remains a notable concern regarding increased power consumption. The demand for substantial processing power in deep learning tasks, coupled with the growing complexity of neural networks, has contributed to an increased energy requirement, *presenting a challenge in achieving optimal power efficiency even with the deployment of specialized AI accelerators*.

Consumption	**CO2e (lbs)**
Air travel, 1 passenger, NY↔SF	1984
Human life, avg, 1 year	11.023
American life, avg, 1 year	36.156
Car, avg incl. fuel, 1 lifetime	126.000
Transformer (big) w/ neural architecture search	626.155

Table 1-1. Estimated CO2 emissions from LLM training, compared to familiar consumption.

In Table 1-1 we present the estimated CO2 emissions from training large language models (LLMs) in GPU accelerators. The consumption is compared to common human activities highlighting the extreme power demands needed for AI processing. One of the key challenges in AI processing is the development of algorithms that can run on fast and power efficient hardware, especially for many computer vision algorithms where response latency and energy consumption are crucial. Deep Neural Networks (DNNs) for example have emerged as a popular choice for the core algorithms of many computer vision tasks due to their ability to learn complex features from raw image data. This has led to a growing interest in the use of hardware accelerators, such as GPUs and FPGAs, to speed up the computations of these algorithms and improve overall performance. Programming such devices and developing software-hardware solutions that can handle the computational demands of these algorithms in real time is not trivial and often requires great programming effort from the engineer's perspective.

Additionally, current state-of-the-art employs approximate multipliers to address the highly increased power demands of DNN accelerators. Approximate computing refers to the idea of sacrificing the accuracy of computation in favor of efficiency, often through the use of reduced precision or simplified operations. Approximate multipliers can significantly reduce the computational complexity of AI models, making them more efficient and practical for real-world applications. Evaluating the accuracy of approximate DNNs proves challenging due to the absence of dedicated approximate hardware. Understanding how these DNNs behave on such hardware is crucial before constructing the hardware, making it a prerequisite for accurate assessment. When hardware is unavailable, the only feasible option is to perform simulation of the approximate multiplier arithmetic. This can be done by leveraging a deep learning framework capable of supporting this functionality but common DNN frameworks lack built-in support. Emulating the behavior of the approximate multiplier using these frameworks will be cumbersome resulting in prolonged execution times. Unlike the optimized libraries available for the standard versions of the frameworks across various DNN layers, there are no equivalents for accelerating the simulation process of the approximate multiplier.

In addition, towards optimizing approximate computations in DNNs, a critical consideration involves identifying the most suitable approximate multiplier for each DNN layer, a concept referred to as cross-layer optimization. This aspect is particularly crucial when the goal is to maximize power gains while adhering to constraints on acceptable accuracy loss. While numerous studies have explored automated methods for determining optimal per-layer quantization in quantized DNNs, the field of approximate DNNs has received comparatively less attention in terms of an automated search flow. In other words, there's a gap in the existing research when it comes to systematically and automatically optimizing the configuration of approximate computations within the entire network. Also, determining the optimal configuration of approximate multipliers between each layer of a DNN model in order to find the best trade-off between accuracy and power is liable to cause a significant computational overhead. The design space becomes large and measuring the accuracy of every configuration is impractical. Addressing these challenges could pave the way for more widespread and practical adoption of approximate DNNs, particularly in resource-intensive applications.

1.2 Thesis Objectives

This thesis makes several significant contributions to the field of efficient computing for deep learning. The objectives of our research can be summarized below:

a) *Acceleration of AI Applications:* Our dissertation will involve accelerating various AI applications by leveraging Field-Programmable Gate Arrays (FPGAs), demonstrating enhanced computational efficiency and performance. We will describe how we can apply various hardware/software optimization techniques for deploying AI algorithms efficiently on FPGAs and compare the execution and efficiency vs other high performance systems such as GPUs. This work aims to go beyond standard FPGA acceleration techniques (such as pipelining or loop unrolling) by exploring advanced optimizations tailored to FPGA architectures. We will apply various hardware/software co-optimization techniques (e.g., SLR partitioning, memory access optimization, Xilinx int8 DSP optimization) specifically tuned for Xilinx FPGA architectures. We will also focus on cloud FPGAs, aiming to bridge the gap between optimized edge AI solutions and cloud-based AI deployments by creating a unified interface using OpenCL FPGA APIs. This approach enables seamless integration and scalability across both edge and cloud environments.

b) *Automatic FPGA firmware from trained CNN models:* We will introduce optimizations for an end-to-end framework towards generating FPGA firmware from trained Convolutional Neural Networks (CNNs). The ultimate goal is to alleviate the engineering effort from optimizing a CNN for FPGA hardware. This framework utilizes HLS4ML [3], enabling the seamless conversion of trained CNN models into optimized FPGA firmware specifically designed for cloud FPGA architectures. Our goal is to enhance the framework by implementing additional optimization techniques, including more FPGA-friendly layer implementations, to provide seamless support for larger network topologies. Furthermore, we will leverage the Xilinx DSP's packed int8 multiplier to improve computational efficiency and performance. Last, as HLS4ML has been previously applied in smaller-scale AI applications like particle physics, we aim to extend its use to large-scale cloud FPGA architectures through a common OpenCL API for both edge and server FPGAs.

c) *Rapid approximate DNN emulation:* In order to enable fast and seamless support of approximate DNN evaluation we will propose two frameworks. AdaPT and TransAxx, both based on PyTorch with CPU and GPU acceleration respectively, will enable support of various DNNs and model architectures. They will also support post-training quantization and approximation-aware finetuning so as to recover the accuracy introduced by approximation. Our objective is that our frameworks will allow developers to test various DNN architectures and approximation levels seamlessly, a feature lacking in most existing tools, which typically focus on static, non-adaptive solutions. In addition to CNNs, which are commonly evaluated in previous work, our frameworks will support a broader range of DNN topologies, including LSTMs, GANs and Vision Transformers. This will expand the applicability of our approach to diverse models, offering more comprehensive support for various approximate neural network architectures.

d) *Automatic Design Space Exploration:* We will address the challenge of finding optimal approximate configuration in-between the layers of a DNN model. We will show we can leverage Monte Carlo Tree Search (MCTS) algorithm as a HW-driven automated search that can achieve significant trade-offs in the power-accuracy space of an approximate DNN model. MCTS will allow for intelligent exploration of the design space by balancing exploration (trying new configurations) and exploitation (refining promising configurations), making it more efficient than previous brute-force or heuristic methods. Additionally, this will be one of the first applications of MCTS for exploring the power-accuracy parameter space of approximate DNNs, specifically for Vision transformer models.

1.3 Overview of the Thesis

The rest of this thesis is organized as follows:

- Chapter 2 provides a background on the research area, specifically related to AI and hardware acceleration, as well as a literature review that explores the key concepts, theories, and methodologies that have shaped the field of this work.

- Chapter 3 presents a detailed analysis of software and hardware optimization techniques applied to Deep Neural Networks for efficient inference along with the demonstration on several real-world AI applications. Also, an end-to-end tool is presented for automatic FPGA firmware generation from trained CNNs.

- Chapter 4 describes the two frameworks, AdaPT and TransAxx for rapid approximate DNN simulation. It also demonstrates the use of an MCTS-based algorithm for automated design space search towards achieving significant trade-offs in the power-accuracy space of an approximate DNN model.

- Chapter 5 concludes this thesis by summarizing the presented results and discusses the future directions of this work.

- The following chapter presents an extended abstract in Greek language.

- Appendix A, in Appendices, describes the SERRANO platform. Specifically it presents a platform towards seamless application development & deployment in the heterogeneous edge-cloud continuum.

2

Background and Literature Review

In this chapter, we provide a concise summary of the previous work and background that form the foundation of the present dissertation. The chapter provides a detailed background on the research area, shedding light on the progression of AI and hardware accelerators which the current study is situated. Additionally, in order to provide a comprehensive understanding of the subject matter, a thorough examination of existing research, theories, and studies has been conducted. The literature review explores the key concepts, theories, and methodologies that have shaped the field, highlighting the gaps, limitations, and unresolved questions that motivate the current study. By presenting a comprehensive background and providing relevant literature, this chapter sets the stage for the subsequent analysis and findings presented in the following chapters.

2.1 Computer Vision and Neural Networks

Computer vision encompasses a branch of Machine Learning dedicated to the analysis and comprehension of images and videos. Its primary objective is to enable computers to "see" by effectively interpreting visual information.

Within computer vision, models are specifically designed to decode visual data by extracting relevant features and contextual information acquired during the training process. This capability empowers these models to comprehend images and videos and apply those interpretations to tasks involving prediction or decision making. However, computer vision heavily relies on abundant data for its operations. Nowadays to accomplish such tasks, neural networks which are a type of deep learning model are needed to be trained to acquire knowledge and enhance their accuracy through iterative learning processes. Once these algorithms are refined for optimal precision, they become powerful tools for artificial intelligence. They enable rapid classification and clustering of data, often surpassing the efficiency of manual identification by human experts.

2.1.1 Historical Development and Milestones

This section will provide a comprehensive overview of the key advancements and significant milestones in the field of neural networks. By exploring the historical context, the reader can highlight the evolution of neural networks, which is essential for understanding their current state and future potential. Besides the historical milestones there were also “AI winters”, a term which was used as an analogy to show periods when funding and interest for artificial intelligence was in decline, specifically around 1970s—mid 1980s and mid-1990s — mid-2000s.

1940: Prehistory: Artificial Neurons

In the 1940s, Warren McCulloch, a neuroscientist, and Walter Pitts, a logician, embarked on pioneering work in the field of artificial neurons. Their collaboration led to the development of the first artificial neuron, which aimed to simulate the operations of organic neurons found in biological systems. This groundbreaking achievement demonstrated the potential of utilizing basic computational units to replicate logical functions [4].

1950: Artificial Neural Networks

Frank Rosenblatt, a research psychologist who worked at Cornell Aeronautical Laboratory, drew inspiration from the influential work of Warren McCulloch and Walter Pitts. Building upon their contributions, Rosenblatt dedicated his efforts during the 1950s to developing the *perceptron* which consisted of a single layer of neurons with the capability to classify images composed of several hundred pixels. The perceptron can be regarded as a significant precursor to the advanced neural networks we employ today [5].

1985: Backpropagation

In his 1974 doctoral thesis, Paul Werbos was the first to propose the utilization of *backpropagation* as a means to optimize neural networks [6]. However, due to the prevailing state of the initial neural network winter, his groundbreaking work went largely unnoticed by the research community. It was only later, with the influential research conducted by Rumelhart et al. in 1986, that backpropagation gained widespread recognition as a powerful training technique for neural networks [7]. Equipped with the capabilities of backpropagation, researchers could now explore numerous applications of neural networks. Notably, Yann LeCun proposed a technique in 1989 for recognizing handwritten digits using neural networks [8]. However, a new challenge emerged, leading to slow and unstable training, posing further obstacles to overcome.

2010: The Rise of Deep Learning

While the roots of deep learning can be traced back to the 1940s and 1950s with familiar terms up until now, such as backpropagation, gradient descent, ReLU, etc., its true boom occurred after 2010. In 2012, deep learning gained significant attention and popularity due to a pivotal moment in the field: the *ImageNet* competition. The ImageNet Large Scale Visual Recognition Challenge, initiated by Fei-Fei Li and her team at Stanford University, provided a dataset of millions of labeled images for researchers to develop algorithms that could accurately identify objects within images. It was during this competition that a deep learning model called *AlexNet* [9], developed by Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, outperformed other traditional computer vision techniques by a significant margin. AlexNet's success demonstrated the enormous potential of deep learning in computer vision tasks. Specifically Convolutional Neural Networks (CNN) models with various architectures, such as AlexNet, begin to gain traction achieving significant results in the competition year after year.

2014: Generative Adversarial Networks

In 1991, Juergen Schmidhuber introduced the concept of adversarial neural networks, wherein two neural networks engage in a zero-sum game, with one network's gain equating to the other network's loss. In 2014, Ian Goodfellow et al. applied the adversarial principle to develop a generative adversarial network (GAN) [10]. Basically, Goodfellow and his colleagues employed a sophisticated statistical analysis to detect the most important features comprising a photograph with the aim to enable machines to generate new realistic images automatically with similar features. In the next years, GANs have found applications in image-to-image translation tasks, such as converting summer photos to winter or altering day scenes into night scenes. Additionally, GANs exceled at generating photorealistic images of objects, scenes, and people to such an extent that humans cannot distinguish them as artificial. By setting neural networks against one another, Goodfellow created a powerful AI tool that is now essential in many AI tasks.

2017: The Transformer model

Ashish Vaswani et al. in their 2017 paper titled "Attention Is All You Need," marked a significant advancement in the field of deep learning and specifically natural language processing [11]. Transformers introduced a novel architecture that revolutionized the way information is processed from human text and represented in neural networks by using a mechanism called *self-attention*. This attention mechanism allowed the model to focus on different parts of the input sequence during processing, enabling better capturing of long-range dependencies. Moreover, Transformers are increasingly being employed in computer vision tasks, expanding their application beyond natural language processing.

2.1.2 Key Components in Neural Networks

Traditionally, computer vision algorithms heavily relied on handcrafted features and traditional machine learning techniques. However, the advent of deep learning has revolutionized the field, leading to remarkable advancements in visual understanding. Deep learning models, particularly convolutional neural networks (CNNs), have emerged as the backbone of computer vision tasks. These models are inspired by the structure and functioning of the human visual cortex. CNNs excel at automatically learning hierarchical representations and extracting discriminative features from raw image data, eliminating the need for manual feature engineering. This paragraph discusses the importance of deep learning models in computer vision and gives an overview of the key components comprising such models.

Brain analogy Before proceeding with the fundamentals of neural networks we will emphasize on the brain analogy that helps in understanding how Convolutional Neural Networks (CNNs) work. CNNs draw inspiration from biological processes observed in the animal visual cortex. Individual cortical neurons respond to stimuli only in a restricted region of the visual field known as the receptive field. To cover the entire visual field, the receptive fields of different neurons partially overlap. This concept mirrors the behavior of the human brain, as CNNs simulate the densely interconnected brain cells using digital neurons. These artificial neurons are designed to trigger or respond when they detect certain features, irrespective of the features' positions in the visual field.

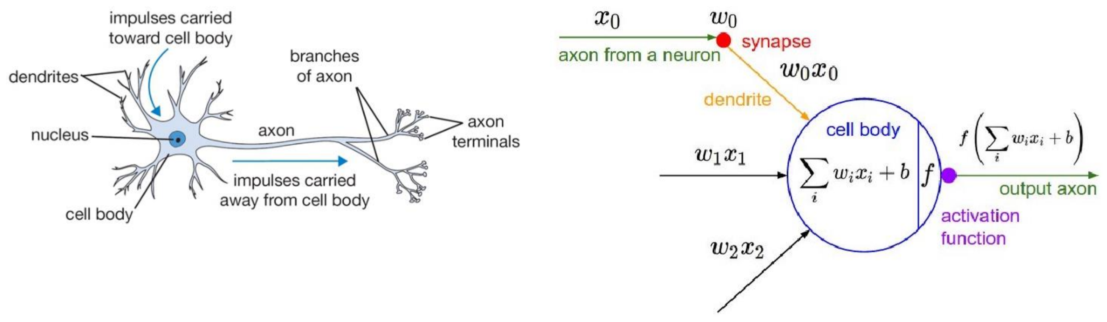


Figure 2-1. A biological neuron (left) and its mathematic model (right)

Neural Network A neural network is created by connecting multiple artificial neurons together. These neurons are organized in a feed-forward network, typically in a directed acyclic graph, although some architectures employ multilayer perceptrons where neurons are grouped into layers. This means that the output of certain neurons can serve as input to other neurons. Neural networks consist of input and output layers, as well as hidden layers that often increase the size and complexity of the network. Each neuron assigns a weight to its input, indicating its degree of correctness or incorrectness with respect to the task at hand. The final output is determined by the sum of these weightings.

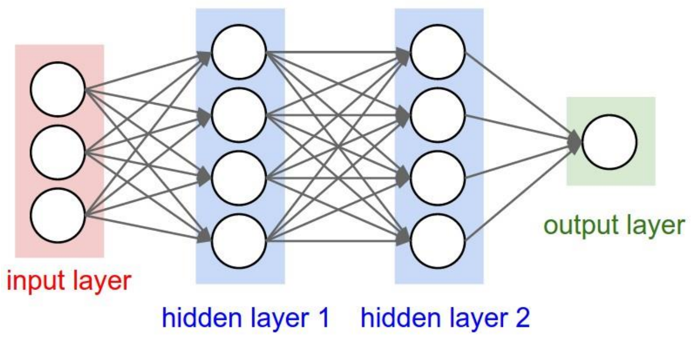


Figure 2-2. The organization of multiple layers of neurons

Layer Types Neural Networks use many different interconnected layers, as shown in the previous figure, specifically designed among other tasks to recognize or detect 2-dimensional image data. A Neural Network can have different layers performing unique tasks aiming to give a specific output that is passed through the other layers.

Some of the fundamental layers found in many models are:

- *Convolutional Layer* applies a convolution operation to the input, passing the result to the next layer. The convolution emulates the response of an individual neuron to visual stimuli. Each convolutional neuron processes data only for its receptive field. Convolutional layers take several feature maps as input and using convolution with the filter weights $k\ x\ k$ acquired from the training process they produce feature maps as output. For filters larger than $1\ x\ 1$, border effects reduce the output dimensions. To avoid this effect, the input image is typically padded with zeros on each side thus reducing the output dimensions.
- *Activation Layer* applies a non-linear activation function to each input pixel. The most popular activation function is the *Rectified Linear Unit (ReLU)* which computes $f(x) = \max{(0,\mathrm{x})}$ and clips all negative elements to zero. Other networks use sigmoidal functions such as $f(x) = 1/(1 + e^{-x})$ or $f(x) = \tanh{(x)}$.
- *Pooling Layer* combines the outputs of neuron clusters at one layer into a single neuron in the next layer by summarizing multiple input pixels into one output pixel. For example, max pooling uses the maximum value from each of a cluster of neurons at the prior layer. Another example is average pooling, which uses the average value from each of a cluster of neurons at the prior layer. They are usually applied to a patch of $2x2$ or $3x3$ input pixels, but can also be applied as global pooling to the whole input image.
- *Fully Connected Layer* connects every neuron in one layer to every neuron in another layer. Each output value of a Fully Connected layer looks at every value in the input layer, multiplies them all by the corresponding

weight it has for that input index, and sums the results to get its output. They can be visualized as one dimensional and perform the high level reasoning in the neural network.

- *Dropout Layer* is a popular method to combat overfitting in large CNNs. These layers randomly drop a selectable percentage of their connections during training which prevents the network from learning very precise mappings and forces some abstraction and redundancy into the weights.
- *Softmax Layer* are often used in the final layer of a neural network-based classifier. It converts the raw class scores z_i into class probabilities P_i according to $P_i = e^{z_i} / \sum_k e^{z_k}$, which result in a vector P that sums to 1.
- *Transformer-Encoder Layer* is found in Transformer models, for example in Vision Transformers (ViT) models which are used in computer vision. This layer is responsible for capturing the spatial relationships and dependencies between different parts of the image. It uses self-attention mechanisms to focus on relevant image regions and extract meaningful features. The Transformer encoder layer plays a crucial role in processing the input image and producing high-level representations that can be further utilized for classification or other tasks.

Training a Neural Network involves the learning of its parameters through a process of optimization. By defining a loss function and employing the backpropagation algorithm, the network can discover the optimal weights. The commonly used approach is supervised training, which requires a set of labeled examples. Initially, the network begins with small or random weights, and each example is repeatedly fed through the network to improve performance (feed-forward pass). Training is considered complete when the network achieves the desired performance on the training data. The backpropagation algorithm calculates the loss between the current network output and the ground truth, then propagates the error backward through the network to compute weight updates (backward-pass). The objective of the learning process is to minimize this loss by adjusting the training weights. The learning rate determines the magnitude of these updates. An example of a basic optimization algorithm called Stochastic Gradient Descent is provided below. It utilizes a subset of examples to compute the parameter gradients relative to the loss function.

$$\theta_{t+1} = \theta_t - \lambda \cdot \nabla_{\theta_t} L\big(f_{\theta_t}(x_i), y_i\big)$$

The convergence of this algorithm lacks formal proof, but in practice, it often converges to good local minima, even with random parameter initialization. One reason for this could be the stochastic nature of the algorithm, which enables it to optimize various loss functions and escape unfavorable minima. Another reason could be that many local minima are nearly as accurate as the global minimum. Ongoing research continues to explore these questions.

Image Classification Once training is complete, the neural network is prepared for image recognition on new data, which is accomplished through the process of *inference*. In this context, the objective is to compute the network's output. The image's colors are represented as RGB values, which range from 0 to 255 and combine red, green, and blue components. Computers can extract the RGB value of each pixel and organize the results into an array for interpretation, which is then fed through all the network layers. The input image is then scanned for features using small filters. Feature extraction begins with the input image, where each pixel serves as input for neurons grouped into features. The neurons in the feature maps are arranged in two-dimensional grids.

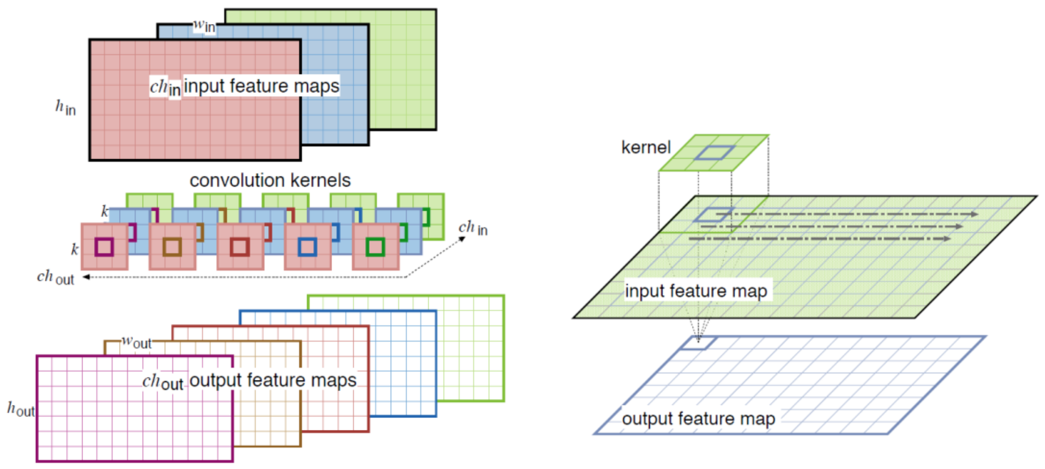


Figure 2-3. Left: Illustration of data inside a CNN Layer (left). The ch_{in} input feature maps are transformed into ch_{out} output feature maps by applying ch_{in} x ch_{out} filter kernels of size k x k. Right: Illustration of a 2D convolution between 3x3 kernel and an input feature map by sliding the kernel over the input pixels and performing MAC operations at each pixel position.

The input enters alternating layers of convolution, pooling and others. Numerous hidden layers can be involved where data is fed through. After passing all layers, the network produces a final vector with a possibility P_i for each category of our model.

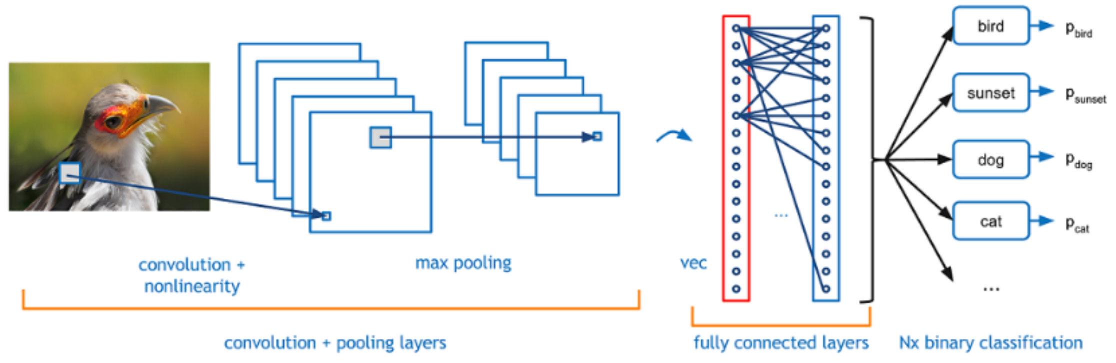


Figure 2-4. The organization of multiple layers of neurons.

2.1.3 The Challenges of Neural Network Inference

Neural networks can address intricate problems across various domains, such as computer vision or natural language processing as we already mentioned. Nonetheless, these approaches encounter challenges and constraints that impede their complete potential and widespread application. Below, we summarize the main existing challenges and limitations associated with neural networks inference.

1. Computational Complexity: Neural networks can be computationally intensive, especially deep neural networks with a large number of layers and parameters. Inference requires performing extensive matrix multiplication operations and activation function evaluations, which can be time-consuming and resource-intensive, particularly on devices with limited computational power.
2. Memory Requirements: Neural networks often require significant memory to store the model parameters and intermediate activations during inference. This can be problematic on resource-constrained devices, such as mobile phones or embedded systems, where memory capacity is limited.
3. Energy Efficiency: Inference on power-limited devices, such as mobile devices or edge devices, requires careful optimization for energy efficiency. Running complex neural networks can quickly drain the device's battery. Therefore, minimizing the energy consumption of the inference process is crucial for real-world deployment.
4. Latency: Many applications demand low-latency predictions. However, neural network inference can introduce delays due to the computational requirements involved. Reducing the inference time to achieve near-instantaneous predictions is a significant challenge, especially when dealing with large and complex models.
5. Model Size: Deep neural networks can be quite large in terms of the number of parameters they possess. This poses challenges in terms of storage, transmission, and memory requirements during inference. Reducing the model size without significant loss in accuracy is a research area of ongoing interest.
6. Edge Deployment: Deploying neural networks on edge devices, such as smartphones, IoT devices, or embedded systems, presents unique challenges. These devices typically have limited computational resources, power constraints, and intermittent connectivity. Efficiently adapting neural networks to operate effectively under such constraints is essential for edge deployment.
7. Privacy and Security: Neural networks trained on sensitive data can raise concerns regarding privacy and security during inference. Protecting the privacy of user data and preventing malicious attacks, such as adversarial examples or model stealing, are significant challenges that need to be addressed.

2.2 Hardware Acceleration for AI

In this section, we will explore the role and significance of leveraging specialized hardware for accelerating AI computations, such as neural network inference. In summary, we will focus on the following key aspects:

- What is AI Acceleration and why we need it: We will provide an overview of hardware acceleration and its relevance in the context of AI. This will involve discussing the need for dedicated hardware to enhance the computational efficiency and performance of AI algorithms.
- Types of Hardware Accelerators: We will explore various types of hardware accelerators commonly employed for AI tasks. This includes Central Processing Units (CPUs), Graphics Processing Units (GPUs), Field-Programmable Gate Arrays (FPGAs) and Application-Specific Integrated Circuits (ASICs). We will examine their architectural characteristics, advantages, and limitations.
- Case Studies and Challenges: To illustrate the effectiveness of hardware acceleration, we will present real-world case studies showcasing hardware acceleration in AI. We will analyze the performance improvements achieved by utilizing hardware accelerators and compare them with traditional computing platforms. Also, we identify the challenges and limitations regarding current implementations.

2.2.1 What is AI Acceleration and why we need it

Hardware acceleration in the context of AI refers to a specialized hardware device or computer system designed explicitly to enhance the performance of artificial intelligence (AI) and machine learning (ML) applications. It targets tasks involving artificial neural networks, machine vision, and other data-intensive or sensor-driven applications. These accelerators often adopt many-core designs and prioritize low-precision arithmetic, novel dataflow architectures, and in-memory computing capabilities. Notably, as of 2018, AI integrated circuit chips typically consist of billions of MOSFET transistors [12]. Within this category, various vendor-specific terms exist to describe these devices, highlighting an emerging technology landscape with no prevailing design standard.

To understand the parallelization of MAC (Multiply-Accumulate) operations in most accelerators, which are common in neural network computations, Figure 2-5 presents an illustration of a generic accelerator architecture. In this architecture, weights and inputs, which can be inputs to the network or activation outputs from a previous layer, traverse through a grid of processing elements (PEs) in a synchronized manner.

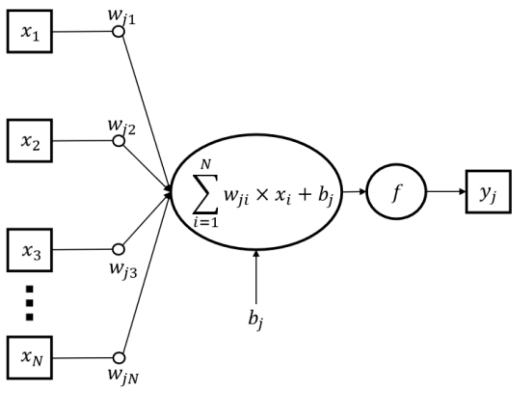


Figure 2-5. Graph diagram of a single neuron in a traditional neural network. [13]

Specifically focusing on the fully connected (FC) layer in an artificial neural network (ANN), each neuron is represented by a row (or column, depending on the implementation) in this grid. The fundamental building block of most AI accelerators is the systolic array, which comprises simple arithmetic logical units (ALUs) integrated within each PE. The arrangement and utilization of buffers (such as input, weight, and output buffers seen in Figure 2-6) may differ among accelerators, but the underlying principle remains consistent: input data and weights are loaded into their respective buffers and propagate through the systolic array. The array performs multiplication operations and accumulates the results, either within the PEs themselves or at the end of a lane of PEs, depending on the employed dataflow technique.

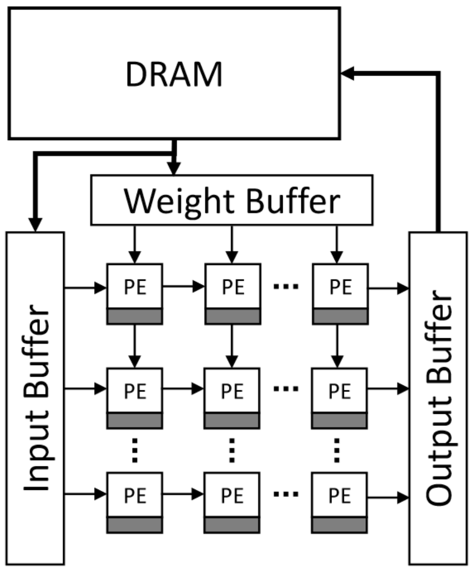


Figure 2-6. Typical architecture of an ANN accelerator. Each PE has a local memory attached to them (gray box) for storing partial sums. The structure of PEs varies depending on the dataflow. [13]

The need for AI advancement To answer the question of why we need acceleration we can observe Figure 2-7 below. It illustrates the progression since the introduction of AlexNet [9] in 2012, which was evaluated in the ImageNet-1k dataset for a 1000-class ImageNet Large-Scale Visual Recognition Competition. The research community has since focused on designing more accurate but complex networks [14]. This progress has been facilitated by the availability of fast hardware devices such as GPUs that can do efficient AI computations and can handle the extensive memory bandwidth and computational demands of large-scale DNNs. As a result, DNN architectures have evolved to become wider and deeper, with models comprising hundreds of layers and billions of parameters [15]. However, this poses a challenge when deploying DNNs on resource-constrained edge devices (e.g., phones, embedded devices, robots) with limited hardware resources such as memory, bandwidth, and energy. Consequently, there is an urgent need to develop efficient methods for deploying and *accelerating* DNNs on such devices without compromising their performance. Thus, AI acceleration is needed both for the advancement of new novel AI models but also for the deployment and efficient execution on devices with limited resources as we will discuss in the next paragraph.

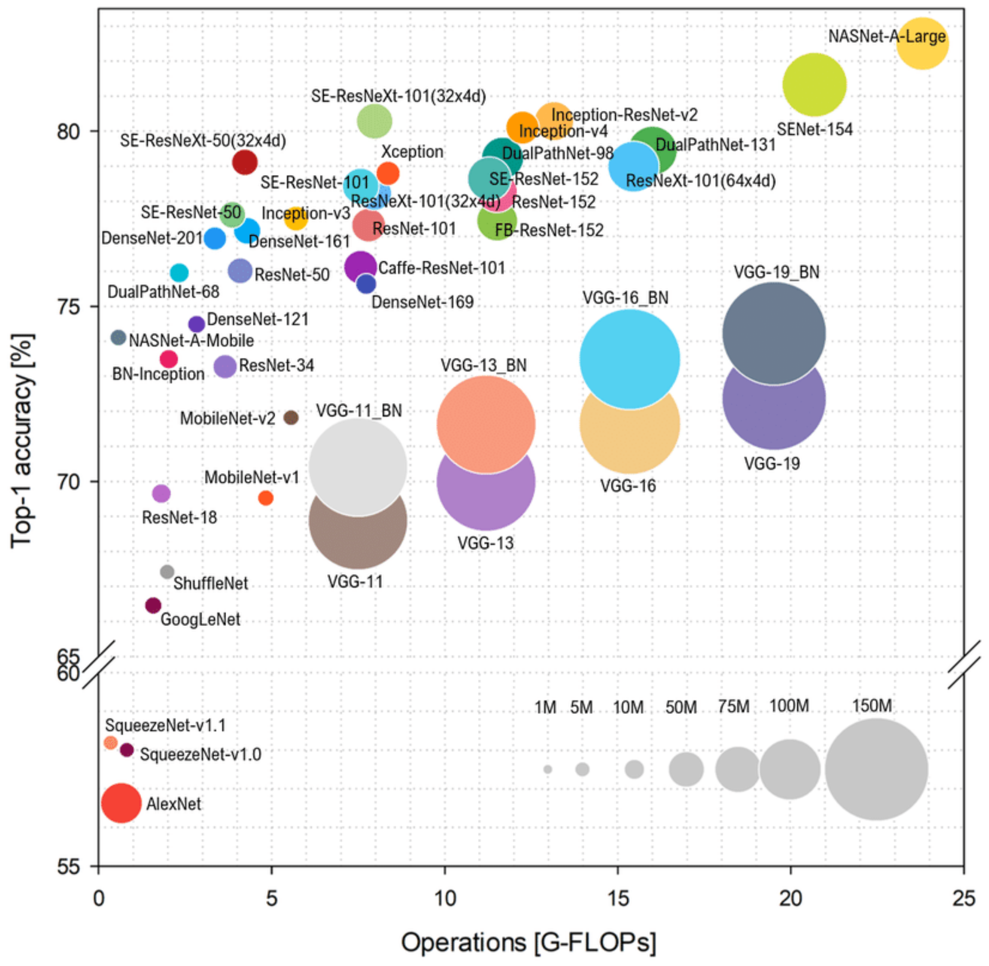


Figure 2-7. Ball chart reporting the Top-1 ImageNet accuracy vs. computational complexity. The size of each ball corresponds to the model's parameters. (Reprinted from [14])

The need for energy efficiency and performance As the size of AI models increases, the number of memory access operations required also grows. Comparatively, computation operations like matrix-matrix and matrix-vector computations are significantly more energy-efficient than memory access operations. When considering the energy consumption of read access from memory versus addition and multiplication operations [16], it becomes evident that memory access requires several orders of magnitude more energy than computation operations. Due to the inability of large networks to fit into on-chip storage, the frequency of energy-intensive DRAM accesses increases significantly. In contrast, AI accelerators can incorporate specific design elements aimed at reducing the frequency of memory access, providing larger on-chip cache, and integrating dedicated hardware features to enhance matrix-matrix computations. By virtue of being purpose-built devices, AI accelerators are more specific to the algorithms they execute, allowing them to leverage their dedicated features more efficiently compared to general-purpose processors.

Operation	Energy [pJ]	Relative Cost
32 bit int ADD	0.1	1
32 bit float ADD	0.9	9
32 bit Register File	1	10
32 bit int MULT	3.1	31
32 bit float MULT	3.7	37
32 bit SRAM Cache	5	50
32 bit DRAM Memory	**640**	**6400**

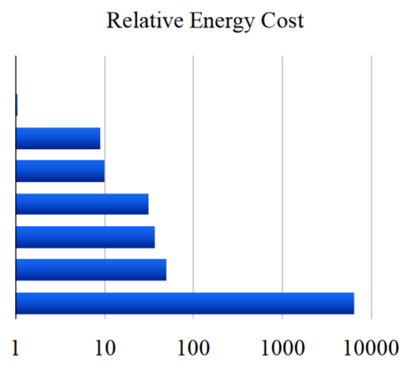


Figure 2-8. Energy metrics for 45nm CMOS process with each type of operation. [16]

Analyzing the provided data, we observe that the energy consumption per neural network connection is predominantly influenced by memory access. The energy required varies, ranging from 5pJ for 32-bit coefficients stored in on-chip SRAM to 640pJ for 32-bit coefficients stored in off-chip DRAM. However, due to the limited capacity of on-chip storage, large networks cannot be accommodated and therefore need more expensive DRAM accesses. To illustrate the magnitude of this energy demand, let's consider the scenario of running a 10 billion connection neural network. The calculation results in $10G \cdot 640pJ = 6.4W$. This might seem low but we are not taking into account the large number of computations which in total will often surpass the power limitations of a typical mobile device. Consequently, accommodating such energy-intensive DRAM access becomes impractical within the power envelope of common mobile devices. AI accelerators need to address these issues while also providing high parallelism. The need for real-world low-latency results becomes apparent, thus such hardware devices must operate at high energy efficiency and surpass at the same time the performance of generic hardware such as CPUs.

2.2.2 Tools and Technologies

Commonly, AI accelerators are coprocessors. Their purpose is to take AI-related workloads from the central processor to improve the efficiency of the overall system. Examples of these workloads include machine learning, deep learning, image processing, and natural language processing, among others. Their purpose and advantages also rest on the purpose and advantages of hardware accelerators. However, there is not a single type of accelerator and there are several reasons why we have multiple technologies today:

- *The field of AI is constantly evolving, and new algorithms and applications are being developed all the time.* This means that there is no single "best" AI accelerator for all purposes. Different devices are better suited for different tasks, and the best choice for a particular application will depend on a number of factors, such as the size and complexity of the model, the desired accuracy, and the power and thermal constraints of the system.

- *The different characteristics of AI acceleration devices are a result of the different ways in which they are designed and implemented.* Some devices, such as GPUs, are designed for general-purpose computing and can be used for a variety of tasks, including AI. Other devices, such as ASICs and FPGAs, are specifically designed for AI and can achieve higher performance for certain types of workloads.
- *The diversity of AI acceleration devices is beneficial to the community, as it gives developers a choice of different options to meet their specific needs.* This diversity is also likely to continue to grow as the field of AI continues to evolve.

In this subsection, we will analyze the most important hardware devices designed for AI computations along with the tools they can be programmed. Most of these architectures were used throughout the experiments of this dissertation.

CPU-based accelerators

While accelerators, which are mentioned next, excel in parallel processing of large-scale data, in the 2000s specific CPU components were designed, driven by video and gaming workloads. CPUs started to support vectorized instructions, such as SIMD (Single Instruction, Multiple Data) extensions like Intel's AVX and ARM's Neon. These instructions enable CPUs to perform parallel operations on multiple data elements simultaneously. With suitable optimizations, CPUs can achieve reasonable performance gains in certain AI workloads as well, especially when the computational demands are not highly parallelizable. CPUs are very efficient for DNNs with small or medium-scale parallelism, for sparse DNNs and in low-batch-size scenarios. It’s worth noting also that the kind of AI acceleration we are referring to regarding CPU-based approaches is typically inference. One other reason that CPU accelerators might be preferred is the host-device latency which is more prominent in the other devices. Many CPU vendors provide specific low level instructions to take advantage of the CPU parallelism. For instance, Intel leverages Advanced Vector Extensions 512 (Intel® AVX-512) and several more AI specific extensions such as DL Boost Vector Neural Network Instructions (VNNI), which consolidates three instructions into one. In general, these optimization strategies maximize the utilization of compute resources, improve cache utilization, and avoid potential bandwidth limitations, resulting in a significant performance boost.

To conclude, CPU acceleration is based on SIMD acceleration which is applied when the same value is being added to (or subtracted from) a large number of data points. This utilization of SIMD instructions allows for efficient parallel processing and can greatly enhance the performance of algorithms that exhibit data parallelism. However, it is important to note that not all algorithms can easily benefit from SIMD, as certain tasks with complex flow control may pose challenges for vectorization. Nevertheless, advancements in research and manual implementation techniques are paving the way for better support and automatic vectorization in compilers, ensuring that the potential of SIMD acceleration can be harnessed more effectively in a wider range of applications. Below, we summarize the advantages/disadvantages of using CPUs for AI acceleration:

CPU advantages :

- They are more suitable for small data or small AI models as they provide ultra-low latency in these scenarios.
- They eliminate the host to device data transfer time because the data is always near the computation in contrast with other accelerators.
- They are affordable and widely available. This enables the use of AI acceleration more easily applicable without requiring additional resources.

CPU disadvantages :

- Some algorithms are not easily adaptable for vectorization. Additional complexity may arise in order to avoid data dependencies as data independence is a requirement for vectorization.
- Often the implementation of algorithms using SIMD instructions typically involves manual effort, as most compilers do not generate SIMD instructions automatically from typical C programs. For example, gathering data into SIMD registers and scattering it to the correct destination locations is tricky.
- Different architectures provide different register sizes (e.g. 64, 128, 256 and 512 bits) and instruction sets, meaning that programmers must provide multiple implementations of vectorized code to operate optimally on any given CPU.
- CPU acceleration is not suitable for medium to large workloads as they cannot provide the massive parallelism other devices can.

GPU-based accelerators

A GPU is a computational processor that executes rapid calculations for image and graphic rendering purposes. GPUs leverage parallel processing techniques to accelerate their operations. In fact, some of the high end GPUs have a higher transistor count than the average CPU. The principle behind their operation is breaking down tasks into smaller segments and distributing them among numerous processor cores, often reaching hundreds or thousands of cores, operating within the same GPU. Historically, GPUs primarily handled the rendering of 2D and 3D images, videos, and animations but now they include a broader array of applications, include DL and big data analytics.

The very large number of cores or threads these devices have often translates to a very high parallelism which is particularly beneficial for AI tasks that involve complex mathematical operations, such as deep learning algorithms. For example, matrix multiplication in neural network training or inference is common and GPUs can do this kind of operation very efficiently. Furthermore, they often have a dedicated video random-access memory (VRAM). The nature of the applications GPUs usually execute require substantial memory bandwidth. Thus, VRAM is a very important component and needs to be physically near the computations to provide data with high throughput to the processing cores of the device. Besides its computational capabilities, a GPU employs specialized programming to facilitate data analysis and utilization. Nvidia dominates the GPU AI market, thus the following analysis will focus on this type of architecture.

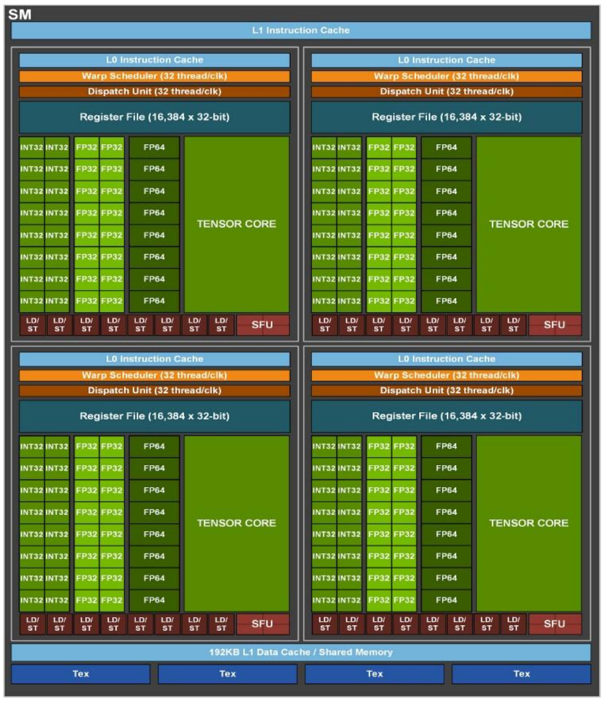

SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
INT32
FP32
FP64
TENSOR CORE
LD/ST
SFU
192KB L1 Data Cache / Shared Memory
Tex

Figure 2-9. Nvidia Ampere SM block diagram [17]

GPU Microarchitecture — The fundamental unit in Nvidia GPUs is the *Streaming Multiprocessor*, or SM. It comprises multiple compute engines positioned together, operating in parallel while awaiting tasks to be assigned to them (Figure 2-9). SMs are responsible for executing the instructions and performing computations such as floating-point arithmetic, matrix operations, texture mapping, and others. Concerning the memory access, GPUs fetch data from the memory hierarchy based on the instructions and store the results back into the appropriate memory locations. Below, are the main memory locations, each with distinct purpose and varying performance characteristics.

- Local *registers* per thread.
- A parallel data cache or *shared memory* that is shared by all the threads.
- A read-only *constant* cache that is shared by all the threads.

- A read-only *texture* cache that is shared by all the processors.
- A *local* cached memory like registers.

Also, a very important component in GPUs is *Tensor Cores*, the heart of Deep Learning processing. These cores are specialized hardware units found in almost any GPU today designed to accelerate matrix and tensor operations, which are fundamental to deep learning and other compute-intensive workloads. They perform matrix multiplications using lower-precision data types instead of the typical floating point 32bit datatype (i.e., FP16, INT8, INT4) which significantly speeds up computation due to reduced memory bandwidth requirements and increased parallelism. Developers also can easily utilize tensor cores using specialized programming interfaces like Nvidia's CUDA and software libraries such as cuBLAS and cuDNN or even more automated end-to-end tools for DNN inference such as TensortRT [18]. The combination of lower precision with the utilization of tensor cores has immensely enhanced the value of low precision in inference tasks as well as training.

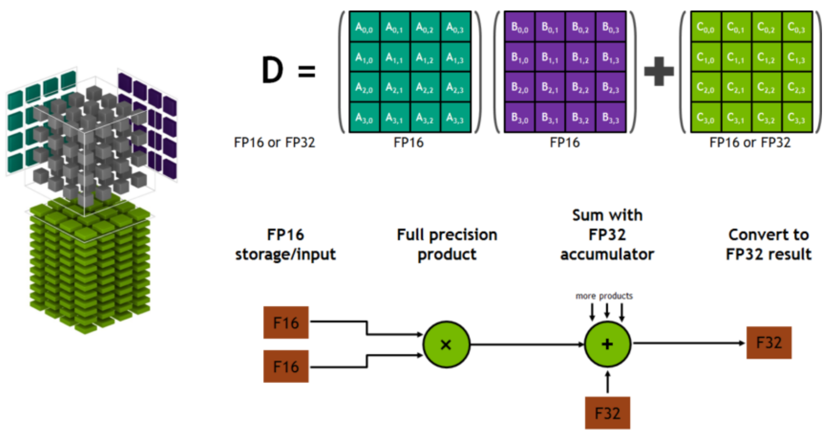


Figure 2-10. Matrix processing operations on Nvidia Tensor Cores. [19]

With each generation of GPU microarchitecture, new techniques have been introduced to enhance the performance of Tensor Core operations. These advancements have expanded their functionalities, enabling them to handle a wider range of numerical formats. For example, Tensor Cores available on Nvidia Volta and Turing GPUs operate on FP16 inputs in order to provide great speedup when performing convolutions or matrix multiply operations (Figure 2-10. Matrix processing operations on Nvidia Tensor Cores. Figure 2-10). Newer architectures such as Nvidia Hopper provide a novel 8-bit floating point format. FP8 reduces deviations from established IEEE 754 floating-point formats with a good balance between hardware and software to leverage existing implementations and enhance developer productivity. While these reduced precision formats can offer improved performance, it is important to note that not all operations should be executed using these data formats due to their limited dynamic range, which may lead to potential overflows.

Many Deep Learning frameworks such as Tensorflow [20] or PyTorch [21] provide GPU support inherently. The integration with GPUs is seamless, allowing users to leverage the computational power of GPUs for accelerated deep learning tasks. These ecosystems continuously evolve, providing a rich set of tools, pre-trained models and libraries advancing the AI development with their compute capabilities. Below, we summarize the most important GPU advantages and disadvantages when considering AI acceleration. Besides the great support and ecosystem of GPUs, it's important to carefully assess the specific requirements of each application and workload to determine whether utilizing GPUs will provide the desired benefits and outweigh any potential disadvantages.

GPU advantages :

- They offer high parallel processing power allowing them to perform multiple computations simultaneously.
- They have become the go-to hardware for training and deploying deep learning models and Nvidia ecosystem is already across various domains and industries.
- Their price-to-performance ratio makes them attractive for tasks requiring massive parallelism.
- They can be easily scaled by adding multiple GPUs to a system, allowing for increased performance and computational capabilities.

GPU disadvantages :

- They have increased power consumption which results in higher electricity costs and may require additional cooling measures.
- They excel at executing a single task on a massive scale but are not well-suited for general-purpose computing tasks or real-time serial tasks.
- They are currently more costly than CPUs especially compared with large-scale GPU systems which can reach significantly high price points.

FPGA-based accelerators

A Field-Programmable Gate Array (FPGA) is an integrated circuit that can be configured after manufacturing. The configuration is typically specified using a hardware description language (HDL) or higher abstraction languages such as High Level Synthesis (HLS). FPGAs consist of an array of programmable logic blocks and reconfigurable interconnects, enabling the connection of these blocks (Figure 2-11). The logic blocks can be set up to perform complex combinational functions or act as simple logic gates like AND and XOR. Many FPGAs also include memory elements within the logic blocks, ranging from basic flip-flops to more comprehensive memory units. This reconfigurable nature allows FPGAs to be reprogrammed to implement different logic functions, enabling flexible and adaptable computing akin to software programming.

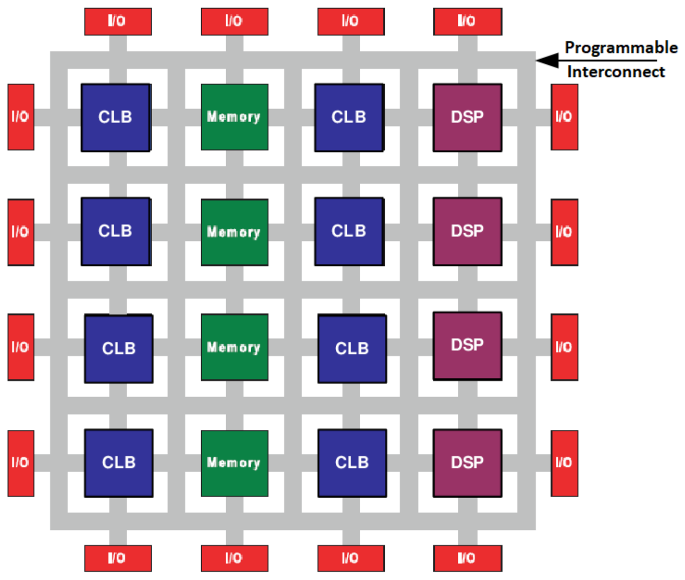


Figure 2-11. Internal structure of Xilinx FPGA [22]

More specifically, FPGAs, beyond being a mere array of gates, possess a sophisticated network of interconnected digital subcircuits, designed with precision to efficiently execute common functions and provide high flexibility. FPGAs mainly have 3 parts:

- *Configurable Logic Blocks* — At the heart of an FPGA's programmable-logic capabilities lies a collection of configurable logic blocks (CLBs) that implement logic functions.
- *Programmable Interconnects* — The CLBs within the FPGA need communication with each other achieved through a matrix of programmable interconnects.
- *Programmable I/O Blocks* — In order to connect with external circuitry FPGAs have programmable I/O blocks.

FPGA programming — FPGA programming involves utilizing an HDL to tailor circuits based on desired device capabilities. Unlike programming a GPU or CPU in a sequential manner, the process entails creating circuits and physically modifying the hardware to suit specific requirements. This process bears similarities to software programming, as the code which is written is converted into a binary file and loaded onto the FPGA. However, the distinction lies in the HDL's ability to perform physical changes to the hardware, rather than merely optimizing the device for software execution.

Unified software platforms enable software developers to utilize their preferred languages to program FPGAs from Xilinx or Intel without extensive knowledge of HDLs. These platforms work by translating higher-level languages into lower-level ones, allowing FPGAs to execute the desired functions. Languages compatible with unified software platforms for FPGA programming include:

- C and C++: High-level synthesis (HLS) now enables C-based languages for FPGA design. The Vivado HLS compiler from AMD-Xilinx provides a programming environment that optimizes C and C++ programs, allowing software engineers to optimize code using specific directives or “pragmas”.
- Python: Designers can use the Python language and libraries to create high-performance applications and program FPGAs. For example, Xilinx PYNQ, an open-source project from AMD simplifies FPGA usage on Xilinx platforms.
- AI development frameworks: AI scientists can use FPGAs without hardware knowledge. For example, with Vitis AI [23] engineers can directly apply their trained deep learning models from TensorFlow or Pytorch and compile them for FPGA acceleration. This eliminates the need for low-level hardware programming and achieves fast compilation times, similar to the software experience of CPUs.

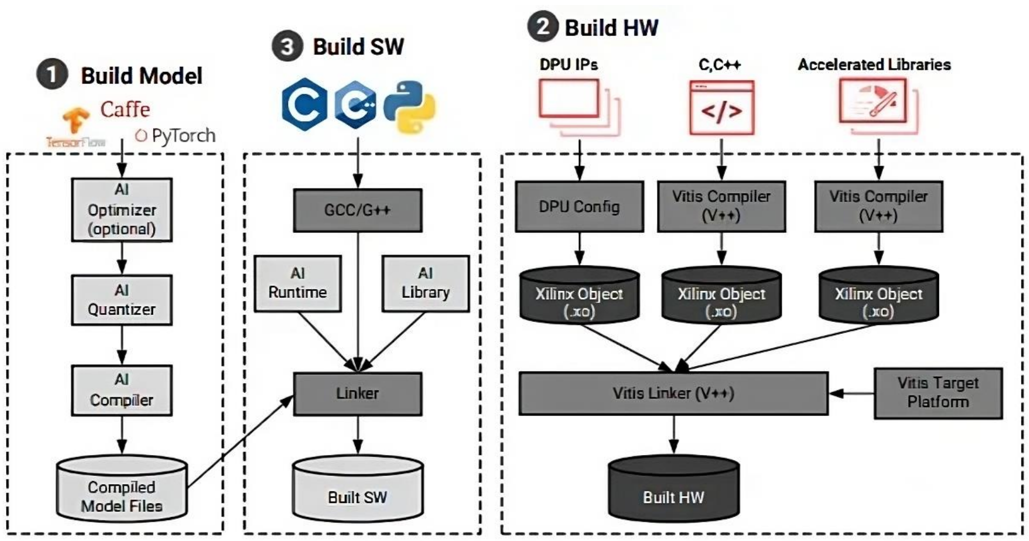


Figure 2-12. Xilinx AI Development stack

In Figure 2-12, the whole Xilinx Development stack is shown. As already mentioned, engineers can use high level tools such as Vitis for synthesizing a C/C++ function into RTL. This high-level code represents the desired functionality of the hardware module that is going to be implemented. For *optimization*, HLS tools offer various optimization options to fine-tune the code for better results. The generated RTL code is then subjected to the synthesis process and then transformed into a gate-level representation, specifying the actual logic gates and interconnections that make up the hardware design. Once synthesis is complete, the tool performs the *place-and-route* process. During this step, the gate-level netlist is mapped onto the FPGA's physical resources, such as lookup tables, flip-flops, and interconnects. After this process, the FPGA *bitstream* is generated. The bitstream contains the configuration data required to program the FPGA, defining the connections and functionality of the hardware module.

Additionally, AI engineers can take advantage of AI development stacks from FPGA companies such as Xilinx Vitis AI. Vitis AI fully supports AI inference algorithms to be accelerated in both edge and cloud environments. At the core of the Vitis AI stack lies the Xilinx Deep Learning Processor Unit (DPU), a programmable logic-based unit optimized for implementing Convolutional Neural Networks (as depicted in Figure 2-12). The DPU takes advantage of the parallel nature of the FPGA's programmable logic to perform fast and efficient computations required for neural network inference. The process of using Vitis AI and DPUs typically involves 1) *optimization*, an optional process to reduce the overall computational complexity of the model, 2) *quantization*, where the model is quantized to a lower bit precision, 3) *compilation*, where the DPU is configured to execute the specific model and the FPGA is programmed with the DPU's bitstream.

Below, we summarize the main advantages and disadvantages of using FPGAs for AI:

FPGA advantages :

- They are known for their power efficiency, as they can be optimized to execute to reduce energy consumption compared to general-purpose hardware
- They can achieve extremely low latency in processing data, making them ideal for real-time AI applications where quick responses are critical.
- They have high flexibility allowing them to fit to specific AI workloads

FPGA disadvantages :

- The development and programming require specialized hardware engineering skills, effort and time from the engineer perspective.
- At the same performance level, FPGAs are generally more expensive than GPUs, and the cost of hiring or training FPGA is significant.
- They have limited on-chip resources, such as logic cells and memory blocks, thus complex AI models often cannot fit in the resources of a single device.
- The ecosystem is not as mature as the ecosystem for CPUs and GPUs.

ASIC-based accelerators

An Application Specific Integrated Circuit (ASIs) is an integrated circuit (IC) chip customized for a particular use, for example AI model acceleration. They typically contain a systolic array which is composed of a large network of basic computing nodes, which can be either hardwired or software configured for specific applications. These nodes are usually fixed and identical, while the interconnect between them is programmable. Unlike the conventional Von Neumann architecture, where program execution follows a sequence of instructions stored in common memory, addressed and sequenced under the CPU's program counter (PC), individual nodes within a systolic array are triggered by the arrival of new data and process the data in a consistent manner.

Due to its ability to handle multiple data streams through data counters, the systolic array supports data parallelism. This enables the systolic array to efficiently process multiple data streams simultaneously.

There are many devices developed by various companies and organizations that are classified as ASIC AI accelerators. The most popular as of today is Google TPU (Tensor Processing Unit) [24] which shares many common characteristics with many accelerators of the same category. Google developed its custom ASIC AI accelerator, the TPU, to accelerate machine learning workloads, particularly neural network inference (
Figure 2-13). TPUs are extensively used within Google's data centers to accelerate various AI applications. They incorporate specialized features like the matrix multiply unit (MXU), which optimizes complex matrix operations, and utilize optical circuit switch (OCS) technology to achieve high-bandwidth memory (HBM). These TPUs can be seamlessly grouped into clusters known as Pods, enabling scalable and accelerated machine learning training and inference. Developers leverage cloud as well as edge TPUs to benefit from high performance, seamless development processes, and cost efficiency.

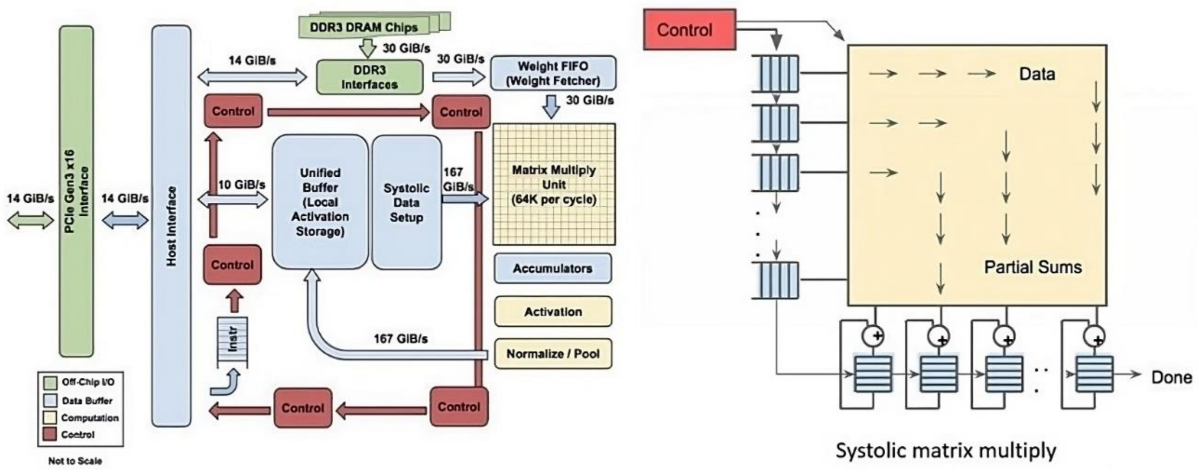


Figure 2-13. Google's TPU architecture and operation

TPUs, just like many AI accelerators, primarily focus on matrix processing combining multiply and accumulate operations. They consist of numerous multiply-accumulators directly interconnected to create a large physical matrix, utilizing a systolic array architecture as shown in Figure 2-13. For example, in the case of Cloud TPU v3, two systolic arrays with 128 x 128 ALUs each are present on a single processor. The TPU's operation involves a stream of data into an infeed queue, where data is loaded into HBM memory. After computation, the results are placed into an outfeed queue. The TPU host then retrieves and stores the results in its memory. For matrix operations, the TPU fetches parameters from HBM memory into the Matrix Multiplication Unit (MXU). Similarly, data is loaded from HBM memory, and as each multiplication is executed, the result moves to the next multiply-accumulator. The output is the accumulation of all multiplication results between data and parameters. This process requires no memory access during matrix multiplication, leading to a high computational throughput.

Other ASIC-based AI accelerators utilize different kind of techniques to improve the performance. Often, the multiplier circuit reduces its area and power consumption by discarding certain partial products used in computing the final result [25]. These circuits trade off precision and circuit complexity to enhance speed and power efficiency. This approach is often referred to as the *approximate computing paradigm*, which allows for design approximations with an acceptable loss of accuracy. In AI, many deep learning models and algorithms can tolerate some loss of precision without significantly affecting the final output or model accuracy. Thus, custom-designed hardware chips often integrate these kinds of optimizations to efficiently perform neural network operations.

Through specialized design ASIC-based accelerators have achieved remarkable levels of performance and energy efficiency. As the demand for AI accelerators grows, many companies are investing in custom ASIC development to gain a competitive edge in the AI market. This has led to a surge in the number of startups and established tech companies specializing in AI accelerator chips, contributing to a vibrant ecosystem of AI hardware innovation [26, 27, 28]. However, these devices have both pros and cons:

ASIC advantages :

- They are often designed specifically for tensor operations, resulting in faster training and inference times for DNNs compared to other accelerators.
- Due to their specialized design, ASICs are very energy-efficient.
- ASIC-based accelerators such as TPUs can scale efficiently to handle large-scale AI workloads and models.

ASIC disadvantages :

- The design process can be time consuming and the development cost high.
- They have limited reusability as the AI landscape changes significantly fast.

2.2.3 The Role of Approximate Computing

The failure of Dennard scaling resulted in the emergence of the "dark silicon problem" [29], compelling computer designers to explore innovative approaches to maintain and enhance the efficiency of computing systems. Within the field of computing, several groundbreaking paradigms emerged, and over the past decade, one of the most remarkable developments has been in Approximate Computing (AxC) research. Approximate computing involves techniques that leverage the inherent error resilience of various applications to achieve enhanced efficiency in terms of energy and performance at all levels of the computing stack. AI applications offer many opportunities for the implementation of AxC techniques due to several factors:

- *Inherent Error Resilience*: Many AI tasks often involve handling large datasets and complex models. These tasks exhibit a level of inherent error resilience, meaning that small deviations in computation or approximations may not significantly affect the overall quality of the results.
- *Function Approximation*: AI tasks frequently involve approximating complex functions to model patterns and relationships within data. AxC techniques can be applied to these approximations, optimizing the trade-off between accuracy and computational resources.
- *Computational Intensity of AI*: AI applications, especially deep learning models, can be computationally intensive, requiring substantial resources for inference and training. AxC techniques can significantly reduce these computational demands without compromising the quality of the results, making AI applications more efficient and cost-effective.
- *Parallelism and Hardware Acceleration*: AI workloads often lend themselves well to parallel processing and hardware acceleration. In specific parallel computing systems and specialized hardware [24] AxC techniques can be efficiently implemented, further enhancing their benefits in terms of performance or energy.
- *Real-time and Embedded Systems*: In certain AI applications, such as those in real-time or embedded systems, power and resource constraints are critical considerations. AxC techniques can address these limitations.
- *Emergence of Low-Precision Hardware*: AxC aligns well with the trend of developing specialized low-precision hardware for AI. AxC necessitates low-precision hardware to fully realize its potential, leading to the creation of such hardware that leveraged AxC for improved performance and energy efficiency.
- *Trade-off Flexibility*: AxC was developed to manage the complex trade-offs between accuracy, performance, energy etc. AxC allows for flexible optimization of these parameters, enabling more effective adjustments based on specific needs.

Taxonomy of AxC techniques for AI — In the past decade, there has been a focused effort to develop Approximate Computing techniques for AI/ML applications, driven by the prospect of achieving significant improvements in performance and computational efficiency. These AxC techniques have been explored across the entire computing stack, ranging from algorithms to circuits. Figure 2-14 illustrates a taxonomy of various approximate computing techniques targeting AI/ML applications. Notably, the most successful techniques are inherently cross-layered, wherein multiple layers of the compute stack are jointly designed to maximize the benefits of approximations while minimizing their impact on the quality of the final output of the applications.

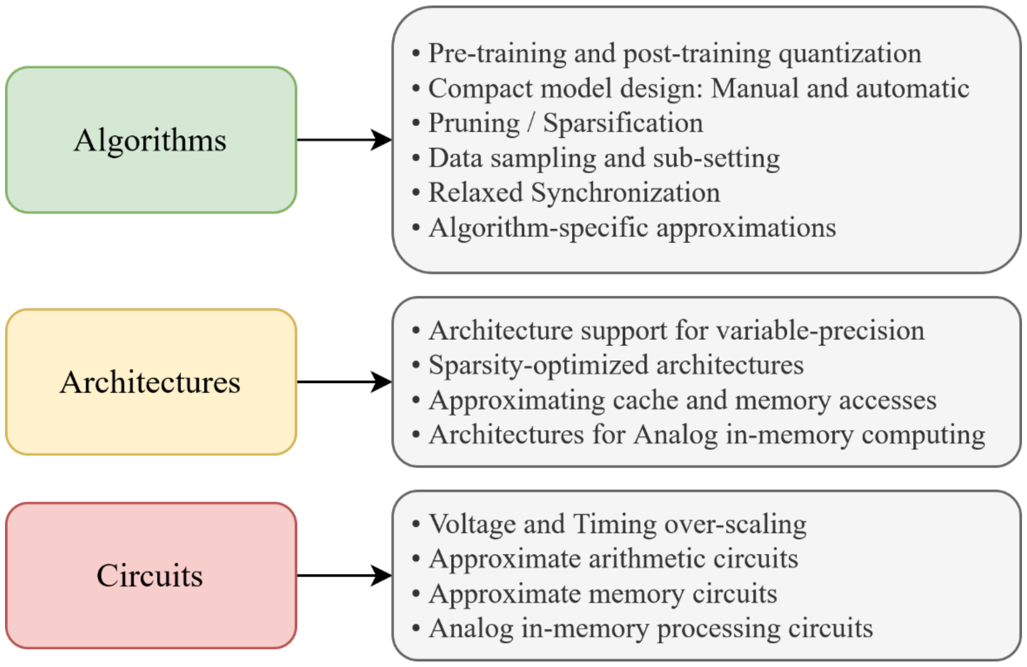

Algorithms
• Pre-training and post-training quantization
• Compact model design: Manual and automatic
• Pruning / Sparsification
• Data sampling and sub-setting
• Relaxed Synchronization
• Algorithm-specific approximations
Architectures
• Architecture support for variable-precision
• Sparsity-optimized architectures
• Approximating cache and memory accesses
• Architectures for Analog in-memory computing
Circuits
• Voltage and Timing over-scaling
• Approximate arithmetic circuits
• Approximate memory circuits
• Analog in-memory processing circuits

Figure 2-14. Taxonomy of Approximate Computing techniques

Delving into Figure 2-14, we can observe that there are several optimizations that can be applied on the *algorithm* level. Most techniques concern the reduction of the computations or memory of the model with the most popular being pruning or quantization. Also, at the *architectural* level, hardware accelerators for AI/ML typically consist of arrays of processing elements. AxC techniques leverage these processing elements to perform neural network operations more efficiently, optimizing both performance and energy consumption. Last but not least, *circuits* play a crucial role in the implementation of approximate computing systems. Previous techniques such as quantization heavily rely on efficient approximate circuit design to achieve maximum benefits. These types of approximations are usually categorized as i) logic approximations, which involve making slight modifications to the logic functionality of a circuit, and ii) timing approximations, where the circuit is designed and operated at an over-scaled voltage-frequency point.

2.2.4 Review of Relevant Studies and Research in the field

This subsection aims to offer a comprehensive review of the most influential architectures used for accelerating Deep Learning (DL). It highlights various approaches that support DL acceleration, including GPU-based accelerators, FPGA-based accelerators and ASIC-based accelerators that we introduced in the previous subsection. In this study, we have examined the research on DL accelerators over the past years that have influenced the ecosystem of AI hardware innovation. It is important to note that due to the prolific and rapidly evolving nature of DL acceleration, we acknowledge that our review may not encompass all research works to date. This survey can serve as a connecting point for the next chapters which focus on specific aspects of hardware optimization, such as reconfigurable hardware and approximate computing. To organize the subsection effectively, we have structured it into different categories based on the area of computer architecture and hardware design. Within each classification, we have cited research papers that were significant for the community but also for our dissertation. Additionally, we have selectively chosen the most notable and influential works under approximate computing for AI which Chapter 4 centers on.

Our main interest was to mainly investigate the trends in performance over several hardware accelerators along with how numerical precision (or other approximation techniques) have impacted these trends. In Figure 2-15, we show a small group of accelerators from [30] over the 10-year period plotted against their peak performance for one or more precision formats. This plot highlights the substantial performance gains achieved over the past decade by supporting lower precision formats.

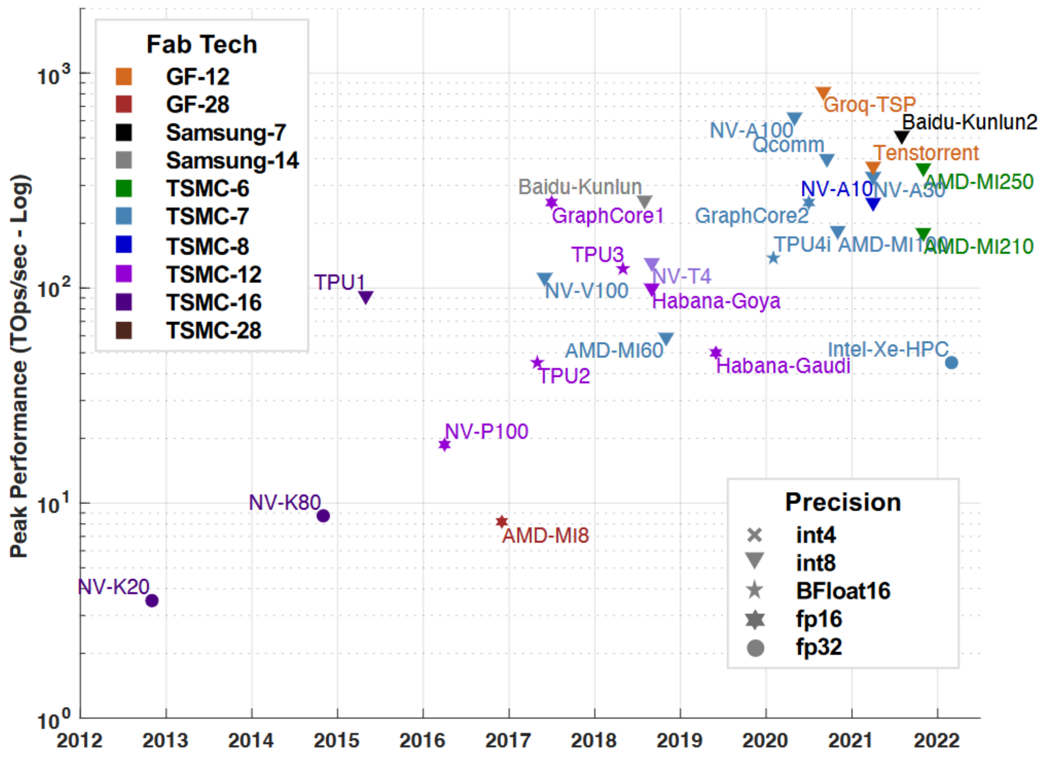


Figure 2-15. Peak performance and fabrication technology vs. release date. [30]

Review on GPU accelerators

Various vendors, including Nvidia, AMD and Intel, have ventured into the development of GP-GPU architectures, primarily focusing on High-Performance Computing (HPC) and, more recently, Artificial Intelligence (AI) computing. Despite their shared hardware details, there is a lack of consistency in the terminology employed by these different vendors. For instance, while AMD refers to it as "Compute Unit," Nvidia calls it "Streaming Multiprocessor," and Intel uses "Compute Slice" or "Execution Unit." Furthermore, Nvidia refers to the set of instructions scheduled and executed at each cycle as "Warp," while AMD adopts the term "Wavefront," and Intel utilizes "EU-Thread." In terms of the execution model, Nvidia employs the "Single Instruction Multiple Thread (SIMT)" approach, while both AMD and Intel use "Single Instruction Multiple Data (SIMD)". Table 3 presents the primary hardware characteristics of the three most recent GP-GPU architectures, namely Nvidia H100 , AMD, and Intel. The comparison includes the peak performance in Teraflops related to 32-bit single-precision (SP), 64-bit double-precision (DP) and half-precision (FP16) along with Thermal Design Power (TDP).

Model (Vendor)	**H100 (Nvidia)**	**Instinct MI250X (AMD)**	**Arc 770 (Intel)**
#physical-cores	132	220	32
#logical-cores	16896	14080	4096
Clock (GHz)	1.6	1.7	2.4
Peak perf. DP (TF)	30	47.9	4.9
Peak perf. SP (TF)	60	95.8	19.7
Peak perf. FP16 (TF)	120	383	39.3
Max Memory (GB)	80 HBM2e	128GB HBM2e	16GB GDDR6
Mem BW (TB/s)	2.0	3.2	0.56
TDP Power (Watt)	350	560	225

Table 2-1. Hardware characteristics of recent GPU systems developed by NVIDIA, AMD, and Intel

Nvidia GPUs have established their dominance in the AI market, positioning themselves as the prevailing choice for artificial intelligence applications [31]. With their high performance and specialized architecture tailored for parallel processing, Nvidia GPUs have demonstrated superior capabilities in accelerating deep learning algorithms and large-scale neural network training. The extensive support for AI frameworks and libraries, such as PyTorch and Tensorflow coupled with advanced software optimizations, further makes these GPUs as the go-to solution for AI researchers, data scientists, and industries who aim to find seamless solutions for their AI tasks. Considering the above, we examine the Nvidia architectures through the years and give a comprehensive review on the related work and research, specifically for AI inference applications.

Advancements in Nvidia GPU technologies — Considering each generation of Nvidia architecture, some minor differences occurred. Through the years, Nvidia developed more efficient architectures and AI-specific circuits to implement efficiently AI algorithms.

With the *Kepler* architecture, Nvidia introduced K20, K40, and K80 GPUs. The K40 processor offers more global memory than the K20, leading to slight improvements in memory bandwidth and floating-point throughput. On the other hand, the K80 features two enhanced Kepler GPUs with additional registers and shared memory, along with extended GPUBoost capabilities. The peak single-precision performance of the Kepler K20 and K40 is approximately 5 Tflops, while the K80's aggregate performance from two GPUs reaches around 5.6 Tflops. The peak memory bandwidth is 250 GB/s for the K20X, 288 GB/s for the K40, and a combined 480 GB/s for the K80.

Moving to the *Pascal* architecture, the P100 board was designed to address memory challenges with stacked memory technology, integrating High Bandwidth Memory 2 (HBM2). This resulted in greater bandwidth, more than double the capacity, and higher energy efficiency compared to previous generations using off-package GDDR5. The P100 achieves peak performances of about 10.5 Tflops in single precision and 5.3 Tflops in double precision. The peak memory bandwidth is increased to 732 GB/s.

The *Volta* architecture combined High-Performance Computing (HPC) and AI capabilities, incorporating Tensor Cores optimized for deep learning and second-generation NVLink with increased throughput. The Tesla V100 offers 7.5 Tflops of DP computing throughput and 900 GB/s peak memory bandwidth, 1.4× and 1.2× improvements over Pascal, respectively.

In the *Ampere* architecture, Tensor Cores have been significantly enhanced, boosting deep learning throughput by 10 times compared to V100. The A100 GPU delivers peak performances of 9.7 Tflops in DP and 19.5 Tflops with FP32.

Finally, the *Hopper* architecture represents Nvidia’s latest generation, introducing new streaming multiprocessors with advanced features. With Tensor Cores that are 6× faster than A100's chip-to-chip, a memory subsystem based on HBM3 modules providing nearly a 2× bandwidth increase, and fourth-generation NVLink providing 3× bandwidth increase, the Hopper achieves remarkable peak performances of 24 Tflops in DP and 48 Tflops using FP64 Tensor Core and FP32 operations. The Hopper's H100 SXM5 GPU supports 80 GB of fast HBM3 memory, delivering over 3 TB/sec of memory bandwidth, effectively doubling the memory bandwidth of the A100. Additionally, the H100 introduces DPX instructions to accelerate Dynamic Programming algorithms with support for advanced fused operands. The PCIe version of H100 provides 80 GB of fast HBM2e with over 2 TB/sec of memory bandwidth.

It’s worth mentioning that Nvidia has established a list of key technologies to accelerate workloads both in edge and cloud domains. Consequently, the availability of this computing power and the novel compute capabilities of GPUs targeting AI workloads facilitated the advancement of new DNN applications with higher computational needs.

Related research on GPUs — Many research papers utilized GPU accelerators to develop or deploy their applications. Specifically, research related to computer vision and neural network inference and training started to grow with the influential paper of Alex Krizhevsky et al. [9]. It introduced the AlexNet architecture, a deep convolutional neural network that achieved a significant breakthrough in image classification accuracy on the ImageNet dataset. The use of GPUs was instrumental in accelerating the training of deep neural networks. After, several other works introduced new AI models that solved more abstract tasks.

Object detection algorithms such as Faster R-CNN [32] or YOLO [33] proposed new model architectures for object detection. They achieved state-of-the-art performance in object detection and benefited from GPU acceleration during both training and inference. However, real-time inference required efficient processing of the entire image in a single pass, which can be computationally challenging for large images and complex scenes. As Shi et al. [18] also suggested on the rationale for edge processing, the tremendous data generated by IoT nodes creates significant challenges in edge applications. In their comprehensive analysis of edge processing applications, they advocate the execution of video analytics close to the data source. Nvidia and other GPU vendors introduced tensor mixed-precision computing, dynamically adapting calculations to decrease latency. Several previous work took advantage of the computing capabilities of the tensor cores to efficiently deploy AI algorithms [34] [35] [36] [37]. Tensor cores, however, are optimized for typically for 16-bit or 8-bit arithmetic. While this is often sufficient for training and inference in deep learning, it can lead to potential issues with numerical stability or data format constraints. It also makes balancing accuracy and power a difficult task as there is limited support for other precision datatypes in contrast with other AI accelerators [38] [39] [40].

Another important work of the last years that influenced the AI ecosystem was "Attention Is All You Need" (2017) by Vaswani et al [11] with the advent of the novel Transformer AI model. The parallelizable nature of the transformer model made it suitable for hardware acceleration. This paper inspired efforts to optimize and accelerate these operations on specialized hardware but also other AI accelerators, to handle large-scale language models efficiently. However, it is not trivial to handle the model's massive data dependencies and memory bandwidth requirements. Researchers have addressed these challenges through techniques like quantization, sparsity, and model distillation [41] [42] [43] but this remains an open issue.

Review on FPGA accelerators

FPGA-based acceleration, especially for neural network inference, has emerged as a compelling area of research, presenting a promising alternative to GPUs in terms of speed and energy efficiency. Researchers have been developing specialized hardware designs to unlock the full potential of FPGA as a viable solution. Numerous FPGA-based accelerator concepts have been proposed, incorporating both software and hardware optimization techniques to achieve impressive levels of speed and energy efficiency. A comprehensive investigation spanning from software to hardware, and from circuit to system levels, has been conducted to provide a holistic analysis of FPGA-based neural network inference accelerator designs. This study serves as a valuable reference and guide for future research but also for comparison with the use-cases of the next chapters which focus on reconfigurable computing.

According to market share research, AMD-Xilinx holds the dominant position as the largest FPGA manufacturer with 3.06 billion in revenue as reported in the 2019 annual reports. Following Xilinx is Intel, with 1.987 billion in revenue in the same year. Apart from these major players in the FPGA market, there are also other FPGA manufacturers such as Lattice Semiconductor and Archonix Semiconductor. Among the leading companies, Xilinx, Intel, offer high-performance FPGA platforms that include the ARM CPU processor but also PCI compatible cards that attach to servers. Intel focuses on GPU and machine learning processor examples, with their VPU primarily targeting image processing. Table 2-2 shows a summary of FPGA hardware platforms. It's worth noting that Xilinx offers a wide range of devices spanning from edge to cloud and hence, this paragraph will primarily focus on examining Xilinx's technologies in this context.

FPGA	**Series**	**Company**	**DSP Slices**	**Description**
Agilex	AGF027	Intel	8528	Quad core ARM Cortex-A53, PCIe Gen5
Stratix 10	DX 2800	Intel	5760	Quad-core ARM Cortex-A53 HPS, HBM2 16G, Intel Optane DC
Stratix 10	GX 2800	Intel	3456	DDR4
Arria 10	GT1150	Intel	1518	DDR4
Virtex UltraScale+	VU19P	Xillinx	3840	DDR4, server Class DIMM
Zynq UltraScale+	ZU7EV	Xillinx	1728	Quad-Core ARM Cortex-A53 MP Core, Dual-core ARM Cortex-R5 MPCore and Mali 400, MP2 GPU, DDR4
Zynq UltraScale+	ZU7CG	Xillinx	1728	Dual-Core ARM Cortex-A53 MP Core and ARM Cortex-R5 MPCore, DDR4
Alveo U200	XCU200	Xillinx	5867	Server Class, DDR4

Table 2-2. Summary Table of FPGA platforms for Xilinx and Intel

Advancements in Xilinx FPGA technologies — Xilinx, one of the leading FPGA manufacturers, has introduced several generations of FPGAs over the years, each representing significant advancements in technology and capabilities. Below is an overview of some key Xilinx FPGA generations and their evolution:

The *Virtex* series: The Virtex series, introduced in the 1990s, marked Xilinx's first foray into high-end FPGAs. These devices offered higher logic density, improved performance, and more resources compared to previous FPGA families. Virtex FPGAs also included specialized features such as Block RAM (BRAM) and Digital Signal Processing (DSP) slices, making them suitable for a wide range of applications. With the Virtex-6 series years after, Xilinx moved to a 40nm process, further improving performance and power efficiency. These FPGAs featured advanced DSP slices and on-chip memory, making them suitable for applications in communications and signal processing.

With the *7 series*, including Artix-7, Kintex-7, and Virtex-7 families, Xilinx brought significant improvements, such as the adoption of 28nm process technology. This generation offered higher logic capacity, improved power efficiency, and higher-speed serial transceivers.

The *UltraScale* series represented a major leap in FPGA technology, introducing 20nm process nodes. It brought massive increases in logic and memory capacity, as well as advancements in high-speed serial transceivers and DSP resources. UltraScale FPGAs offered a higher level of integration and performance for demanding applications.

The *UltraScale+* series continued the evolution, incorporating 16nm process technology. It further enhanced performance, power efficiency, and integration capabilities. UltraScale+ devices included features like High Bandwidth Memory (HBM) integration, versatile programmable I/O (PIO) banks, and advanced system-level features.

The newer generation of devices was the *Versal* series built on the TSMC 7 nm FinFET process technology. These devices included a typical programmable logic with some improvements in DSPs and other components, but also introduced the new AI engines. These intelligent engines provided up to 5× higher compute density for vector-based algorithms compared with previous approaches. They were optimized for real-time DSP and AI/ML computation built from the ground up to be software programmable and hardware adaptable. In particular, AI Engines are an array of very-long instruction word (VLIW) processors with single instruction multiple data (SIMD) vector units that are highly optimized for compute-intensive applications. With this new technology, AMD-Xilinx introduced a new paradigm of AI acceleration.

Related research on FPGAs — The versatility of FPGAs have made them particularly attractive for research, development, and deployment in various applications, including edge devices, data centers, and Internet of Things (IoT) devices. FPGA-based accelerators have shown to significantly reduce power consumption, contributing to energy-efficient and environmentally friendly computing solutions [44, 45].

One common method of achieving parallelization in FPGAs is by reducing the computation's bit-width directly which consequently leads to a reduction in the size of computation units. In state-of-the-art FPGA designs, 32-bit floating-point units are often replaced with fixed-point units. For instance, Podili et al. [46] implemented 32-bit fixed-point units for their proposed system, while other works like [47, 48, 49] widely adopted 16-bit fixed-point units. Notably, some works explore narrower bit-width designs. For example, Guo et al. [50] employed 8-bit units for their embedded FPGA design. Surprisingly, experiments with extremely narrow networks, such as Binarized Neural Networks (BNN) that employ 1-bit weights/activations, have shown remarkable results, exhibiting little accuracy loss [51, 52, 53]. Low bitwidth fixed-point arithmetic has been appealing to the FPGA designers and the latest state-of-the-art designs have been leveraging them to reduce latency and power. Often re-training or calibration is needed if aggressive quantization is applied so as to optimize the weights or activations to the new value distributions [54, 55].

Several prior works focused on optimizing convolution operations in convolutional layers by adopting alternative mathematical functions. The convolutional layer is the most common layer for acceleration in neural networks in computer vision. For example, in [56] they used Discrete Fourier Transformation (DFT) from digital signal processing for faster convolutions. Zhang et al. introduced a hardware design utilizing 2D DFT. Also, according to Ding et al. [57], a block-wise circular constraint can be utilized on the weight matrix. By doing so, the matrix-vector multiplication in Fully Connected (FC) layers is transformed into a series of 1D convolutions, enabling acceleration in the frequency domain. Last, several papers on neural network acceleration on FPGAs proposed the fast Winograd algorithm to reduce the number of arithmetic operations required for convolutions [58, 59].

Despite the efforts from the researchers to reduce memory or compute requirements of neural networks to achieve acceleration, there are always design or productivity limitations. High-level synthesis tools and design frameworks can help improve design productivity, but they may still face limitations in handling complex CNN architectures. Several previous works have proposed frameworks to automate the optimization of CNN models for FPGAs [60, 61]. Another popular framework is FINN [62], an end-to-end tool that enables design-space exploration on reduced precision neural networks. HLS4ML [3] is another widely adopted tool that automatically creates firmware implementations of AI algorithms using high level synthesis language. In these frameworks, however, the produced designs were not always optimal while there were many limitations and partial support for neural networks layers or operations.

Review on ASIC accelerators

ASIC accelerators play a crucial role in the acceleration of deep learning tasks, providing highly efficient and powerful solutions for a wide range of applications. As the demand for AI and deep learning grows, ASIC technology continues to advance further enhancing the capabilities of these specialized accelerators. Google has been a major contributor to AI hardware with its custom-designed Tensor Processing Units (TPUs). Also, Intel's AI hardware division, Habana, has been developing dedicated ASICs. These chips are designed to deliver high performance and efficiency for deep learning workloads. Graphcore has also gained attention for its Intelligence Processing Units (IPUs), custom-designed ASICs for AI acceleration. IPUs focus on parallelism and use a novel graph-based approach to accelerate neural network computations. Cerebras Systems has developed the Wafer-Scale Engine, a revolutionary AI accelerator that covers an entire silicon wafer with a massive number of cores and memory, providing exceptional performance for deep learning. While there are many ASIC AI accelerators most of them include some common characteristics such as customized instruction sets and data paths tailored for AI or specialized tensor processing units. In Table 2-3 we summarize some important AI accelerators.

Company	Product	Label	Form Factor
Cerebras	CS-1	CS-1	System
Cerebras	CS-2	CS-2	System
Google	TPU Edge	TPU Edge	System
Google	TPU1	TPU1	Chip
Google	TPU2	TPU2	Chip
Google	TPU3	TPU3	Chip
Google	TPU4	TPU4	Chip
Google	TPU4i	TPU4i	Chip
GraphCore	C2	GraphCore	Chip
GraphCore	C2	GraphCoreNode	System
GraphCore	Colossus Mk2	GraphCore2	Chip
GraphCore	Bow-2000	GraphCoreBow	Chip
Habana	Gaudi	Gaudi	Card
Habana	Goya HL-1000	Goya	Card
Qualcomm	Cloud AI 100	Qcomm	Card

Table 2-3. List of ASIC AI accelerators

Many of these accelerators leverage *dataflow* processing, a unique computational model that organizes computations based on the availability of data rather than control flow. Neural network training and inference computations can be effectively mapped in a deterministic manner, making them well-suited for dataflow processing. This kind of architectures offers advantages in terms of flexibility, resource utilization, and parallelism compared to traditional instruction-based architectures.

Advancements in ASIC technologies — The *first generation of ASICs* dedicated to deep learning emerged around 2015. Google's Tensor Processing Unit (TPU) was one of the pioneering ASICs in this category, optimized for inference tasks. These ASICs demonstrated significant performance improvements compared to GPUs and CPUs for deep learning workloads.

In the following years, *second-generation ASICs* further improved upon the performance and efficiency of early AI chips. These chips incorporated more advanced architectures and optimizations, enabling better parallelism, support for better quantization techniques, and memory management. NVIDIA's Volta and Xavier GPUs, equipped with Tensor Cores, were notable examples of this generation which can be characterized as AI chips.

With *large-Scale and wafer-scale ASICs*, later generations pushed the boundaries of scale and size. Cerebras Systems introduced the Wafer-Scale Engine (WSE), an innovative AI accelerator that spans an entire silicon wafer, featuring thousands of cores and a large-size memory. Such large-scale ASICs provided unparalleled performance for both training and inference tasks and addressed the problem of on-chip memory limitations which is a common challenge faced in the design of AI accelerators. Cerebras tried to overcome the on-chip memory limitation which arises due to the physical constraints of the chip's size by creating a large amount of memory that could fit into their custom chip. Thus neural network memory such as intermediate data, activations, and model parameters could be successfully processed on-chip during computation.

Next generations focused on *high-efficiency training ASICs*. While early ASICs focused primarily on inference, later generations addressed the demands of AI training workloads. Companies like Habana Labs or Graphcore developed ASICs optimized for AI training tasks, with features such as large memory bandwidth, higher precision arithmetic, and sophisticated communication mechanisms between chips.

It’s worth mentioning that due to the substantial number of AI accelerators available, efforts have been made to establish a standardized benchmark that facilitates fair comparisons among these devices. An initiative with the name MLPerf aimed to provide an impartial and objective performance standard for software frameworks, hardware platforms, and cloud solutions involved in machine learning. A diverse consortium of AI community researchers and developers from over 30 organizations collaboratively designed and continually enhanced these benchmarks. Various companies uploaded their device benchmarks over MLPerf which through the years made MLPerf to become a widely recognized and respected benchmarking suite. Now it is an industry standard as it has independent and objective evaluations, recognition and wide adoption over many major companies in the field.

Related research on ASICs — ASIC AI architectures often incorporate innovative approaches, such as dataflow processing, systolic arrays, and specialized units for matrix multiplication and tensor operations. There are also heterogeneous approaches exploring hybrid architectures [24, 28, 27] that combine different processing units (e.g., scalar processors, tensor processors, vector units) to achieve a balance between flexibility and efficiency .

Reduced precision (e.g., 16-bit, 8-bit) continues to be the default numerical precision for these devices whether targeting embedded, autonomous or data center applications. The precision provided by these formats is generally sufficient for most AI/ML applications that involve a reasonable number of classes [63, 64]. It is also very common for AI chips to appear in embedded system-on-chip (SoC) solutions, which often include low-power CPU cores, audio and video analog-to-digital converters (ADCs), encryption engines, network interfaces, etc. Also, most AI-chip companies like Cerebras or GraphCore have highly scalable inter-networking technologies that allow thousands of cards to be interconnected. This scalability is particularly crucial for dataflow accelerators and facilitates the networking of multiple cards, effectively handling even extremely large models like transformers [65]. Nevertheless, the ever-increasing computational demands of modern complex DNNs cannot be met by these devices alone and often more aggressive optimizations are needed.

While some approaches like the previous have reduced precision arithmetic, a large number of state-of-the-art works employ approximate multipliers to address the highly increased compute demands of DNN accelerators. Approximate computing techniques are based on the intuitive observation that achieving exact computation (for example exact multiplication) is not often needed. [66] employed approximate multipliers to different convolution layers and [67] proposed a compact and energy-efficient multiplier-less artificial neuron. Most of previous works focus on layer-wise approximation [68] where each layer has a different multiplier. Extensive search although is needed to find the optimal approximation per layer. Also, usually additional finetuning or retraining is required to mitigate the error induced by the approximate multipliers.

Evaluating the accuracy on approximate DNNs is cumbersome due to the limited support from DL frameworks which do not have optimized approximate mathematical operations. Thus, there have been many works that proposed approximate DNN frameworks [69, 70, 71] but there are still open challenges such as overcoming the slow inference time due to limited approximate arithmetic support from DL frameworks, re-training on approximate operations or finding optimal approximation per layer using search algorithms.

3

Optimization of Deep Learning Accelerators

In this section, we present a detailed analysis of software and hardware optimization techniques applied to Deep Neural Networks for efficient inference. The first section describes several optimizations that were explored throughout this thesis with a particular focus on their use and impact on computer vision tasks such as image classification. In the next three subsections, three distinct AI tasks are examined with implementation details regarding acceleration using reconfigurable hardware. The last section presents a comprehensive end-to-end framework for automatic acceleration of Convolutional Neural Networks on FPGAs and shows its performance through different experiments.

3.1 Overview

We begin this chapter by discussing various strategies and techniques to improve the efficiency and performance of neural networks for both software and hardware implementations. From the software perspective, we discuss the optimization techniques of quantization and pruning to reduce the precision of weights and activations, along with layer fusing to overcome the overhead due to memory access and intermediate data storage. From the hardware perspective, we cover the most important optimizations, which are universal for many hardware platforms, but with a focus on reconfigurable hardware. These include the parallelization strategies, data movement and precision scaling techniques. Next, we give three real world use-cases where various ML/AI algorithms were accelerated using FPGA platforms, along with in-depth explanations of the implementation process and the outcomes obtained. Last, we present our end-to-end tool for automatic FPGA firmware generation from CNNs, which is based on HLS4ML [3] tailored for cloud FPGA architectures.

3.2 Common Techniques for Efficient Inference

In scenarios requiring the execution of AI algorithms with constrained energy and low latency, there arises a demand for achieving energy-efficient deep learning execution. Numerous techniques can be employed to streamline the DNN inference process and improve its efficiency, making it more feasible to deploy on various hardware platforms with constrained resources such as FPGAs. However, we focus on the most important ones, namely quantization, pruning, and layer fusion.

3.2.1 Software: Quantization, Pruning and Layer Fusion

Quantization

The main challenge occurs when running on smaller, less powerful devices, but often on cloud devices as well, where there is high memory and computational demand. It involves adapting a relatively large neural network to operate efficiently. To this end, the most effective technique is known as quantization. To understand the concept of quantization, it's crucial to first examine why neural networks, in general, require substantial memory.

As we described in 2.1.2, a neural network comprises interconnected neurons organized into layers. Each layer involves a set of interconnected neurons, each having its weight, bias, and associated activation function. During training, the weights, biases, and activations are adjusted to optimize the neural network's performance. These values represent the majority of data stored in memory by the network. Typically, they are represented as 32-bit floating-point values to ensure high precision and accuracy. This precision comes at the cost of memory usage, especially for large networks with millions of parameters and activations. For instance, the 50-layer ResNet architecture includes approximately 26 million weights and 16 million activations. When stored as 32-bit floating-point values, the entire architecture requires around 168 MB of storage. Consequently, this is why neural networks tend to consume significant memory resources.

Quantization, however, is important not only for reducing memory requirements but also for optimizing computations in hardware accelerators. It plays a crucial role in improving the overall efficiency and performance of DNNs on hardware devices such as GPUs, FPGAs and other specialized accelerators. Generally, quantized neural networks use integer arithmetic format as we will see next, although new FP8 datatypes have been recently proposed and adopted by various companies [72]. Integer operations are generally faster and require less power than floating-point operations. This allows the accelerator to process more data in parallel and achieve higher throughput.

Quantization concept — Quantization involves reducing the precision of neural network parameters, including weights, biases, and activations, to decrease their memory consumption. For example, 32-bit floating-point values which are used to represent parameters are converted into a more compact representation, such as 8-bit integers. int8 quantization has become a popular approach for such optimizations not only for deep learning frameworks like TensorFlow and PyTorch but also for hardware toolchains like NVIDIA TensorRT and Xilinx Vitis AI framework.

To transition from floating-point to more efficient fixed-point operations, we require a method to convert floating-point vectors into integers. This conversion can be achieved by expressing a floating-point vector x as an approximate scalar multiplied by a vector of integer values:

$$\hat{\mathbf{x}} = s_{\mathbf{x}} \cdot \mathbf{x}_{\text{int}} \approx \mathbf{x}$$

We use a floating-point scale factor, $s_{\mathbf{x}}$, and an integer vector (e.g., INT8) denoted as $\mathbf{x}_{\text{int}}$ to quantize the vector x. The quantized version of the vector is represented as $\hat{\mathbf{x}}$. By applying quantization to both the weights and activations, we can express the quantized version of the accumulation equation:

$$\begin{aligned}\hat{A}_n &= \hat{\mathbf{b}}_n + \sum_m \widehat{\mathbf{W}}_{n,m}\hat{\mathbf{x}}_m \\ &= \hat{\mathbf{b}}_n + \sum_m \left(s_{\mathbf{w}}\mathbf{W}_{n,m}^{\text{int}}\right)\left(s_{\mathbf{x}}\mathbf{x}_m^{\text{int}}\right) \\ &= \hat{\mathbf{b}}_n + s_{\mathbf{w}}s_{\mathbf{x}}\sum_m \mathbf{W}_{n,m}^{\text{int}}\mathbf{x}_m^{\text{int}}\end{aligned}$$

Often, separate scale factors are used for weights ($s_{\mathbf{w}}$) and activations ($s_{\mathbf{x}}$) as they offer increased flexibility and reduce quantization errors. By applying each scale factor to the entire tensor, we can factor them out of the summation in the previous equation, allowing for fixed-point format MAC operations. Bias quantization is currently omitted, as biases are typically stored with a higher bit-width (32-bits), and their scale factor depends on that of the weights and activations [63]. Maintaining a higher bit-width for the accumulators, is crucial to prevent potential loss due to overflow during the computation.

The quantization scheme referred to as *uniform affine* is widely used because it allows for an efficient implementation of fixed-point arithmetic. Uniform affine quantization, also known as *asymmetric quantization*, is characterized by three quantization parameters: the scale factor (s), the zero-point (z), and the bit-width (b). These parameters work together to map a floating-point value to the integer grid, whose size is determined by bit-width. Once the three quantization parameters are defined, we can proceed with the quantization operation. Beginning with a real-valued vector x, we map it to the *unsigned* integer grid $\{0, ..., 2^b - 1\}$:

$$\mathbf{x}_{\text{int}} = \text{clamp}\left(\left\lfloor \frac{\mathbf{x}}{s} \right\rceil + z; 0, 2^b - 1\right),$$

where $\lfloor . \rceil$ is the round-to-nearest operator and clamping is defined as:

$$\text{clamp}\,(x; a, c) = \begin{cases} a, & x < a \\ x, & a \leq x \leq c \\ c, & x > c. \end{cases}$$

To approximate the real-valued input x we perform a *de-quantization* step:

$$\mathbf{x} \approx \hat{\mathbf{x}} = s(\mathbf{x}_{\text{int}} - z)$$

By combining the two steps mentioned earlier, we can present a comprehensive definition for the *quantization* function, denoted as $q(.)$:

$$\hat{\mathbf{x}} = q(\mathbf{x}; s, z, b) = s\left[\text{clamp}\,\left(\left\lfloor \frac{\mathbf{x}}{s} \right\rceil + z; 0, 2^b - 1\right) - z\right]$$

Through the de-quantization step, we can determine the quantization grid limits (q_{min}, q_{max}), where $q_{min} = -sz$ and $q_{max} = s(2^b - 1 - z)$. Values of $\mathbf{x}$ outside this range will be clipped to these limits, resulting in a clipping error. To reduce the clipping error, we can expand the quantization range by increasing the scale factor. However, this also leads to increased rounding error, as it falls within the range $[-\frac{1}{2}s, \frac{1}{2}s]$.

Quantization techniques — Normally, the quantization type can be Post-training quantization (PTQ) or Quantization-aware training (QAT). In the first, the model is quantized after it has been trained while on the latter the model is further trained with quantization in mind. PTQ can be data free or require a small calibration dataset while QAT will often require data. While PTQ is more straightforward to implement as it doesn't require retraining, QAT can often result in more accurate quantized models due to its training-awareness. However, finding optimal quantization ranges is critical for minimizing quantization errors in PTQ, whereas QAT needs to handle the complexities of training with quantization constraints. Last, quantization may need to be tailored to match the capabilities and preferences of the target hardware, ensuring the best performance and accuracy.

Pruning

Considering the redundancy in neural network parameters, network pruning involves the elimination of certain parameters, specifically setting them to zero, that have no significant impact on the model's performance (i.e., its accuracy). The concept of pruning was initially explored in Optimal Brain Damage [73], where weights with minimal influence on the loss function during training were pruned. A simpler approach [74] involves pruning weights with small magnitudes after training, followed by fine-tuning the remaining weights to recover any potential accuracy loss. This straightforward and linear method allows for a substantial reduction in the number of parameters in models such as the well-known AlexNet model, achieving up to a 10x reduction [74].

Pruning concept — There are various methods for pruning a neural network. One approach involves pruning weights, which entails setting specific parameters to zero, effectively making the network sparse. This results in a reduction of parameters in the model while maintaining the original architecture intact. Another method involves eliminating entire nodes from the network. By doing so, the overall architecture of the network becomes smaller, with the goal of preserving the accuracy achieved by the initial larger network.

Weight-based pruning is more popular due to its ease of implementation without compromising the network's performance. However, for it to be effective, sparse computations are necessary, which demands hardware support and a certain level of sparsity to achieve efficiency. On the other hand, node pruning enables dense computation, which is highly optimized and allows the network to run normally without relying on sparse computation. Dense computation is generally better supported by hardware. Nevertheless, the removal of entire neurons can more readily impact the accuracy of the neural network. Below is the visualization of pruning weights/synapses:

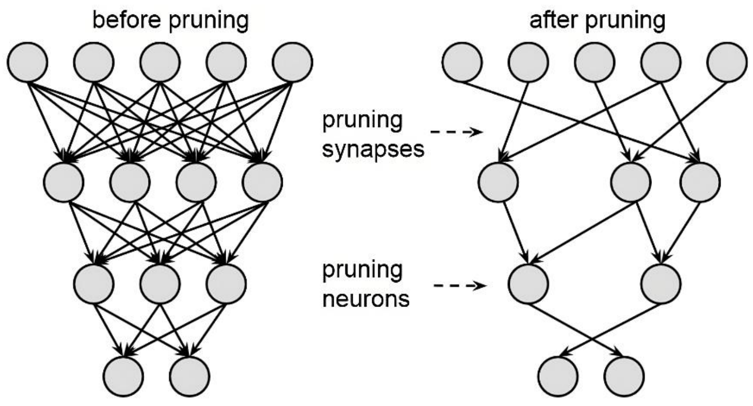


Figure 3-1. Visualization of pruning weights/synapses vs nodes/neurons

Pruning techniques — A significant challenge in the pruning process lies in deciding what to remove. When eliminating weights or nodes from a model, the objective is to target parameters that are less valuable. There exist various heuristics and methods for identifying less important nodes to be pruned while preserving accuracy to the maximum extent possible. These heuristics often rely on analyzing weights or activations of neurons to assess their significance in the model's performance. The ultimate goal is to eliminate a greater proportion of the less crucial parameters.
One of the simplest pruning techniques is based on the *magnitude of the weight*. Removing a weight essentially involves setting it to zero. To minimize the impact on the network, it is prudent to remove weights that are already close to zero, indicating low magnitudes. This can be practically achieved by discarding all weights below a certain threshold. When pruning a neuron based on weight magnitude, the L2 norm of the neuron's weights can be utilized as a helpful criterion.

Rather than relying solely on weights, the *activations from training data* can serve as a criterion for pruning. During the dataset's pass through the network, certain statistics of the activations can be observed. For instance, some neurons may consistently produce near-zero output values, suggesting that these neurons can likely be removed with minimal impact on the model. The underlying intuition is that if a neuron rarely activates with a high value, it is also rarely involved in the model's task.

It's worth noting that activation-based pruning can be more aggressive compared to weight-based pruning since it directly considers the importance of neurons based on their behavior with the training data. However, the effectiveness of activation-based pruning may vary depending on the specific dataset, architecture, and task. As with any pruning method, it's essential to evaluate the pruned network's performance and, if necessary, apply additional techniques to maintain or enhance its accuracy and capabilities. Ideally, in a neural network, all neurons should possess unique parameters and output activations of significant magnitude, without redundancy. The goal is to ensure that each neuron contributes something distinct, while removing those that fail to do so.

Last, when evaluating a pruning method, several metrics come into play besides accuracy such as size and computation time. Accuracy is crucial for assessing the model's performance on its task. Model size refers to the amount of storage required to store the model's parameters. Computation time can be measured using FLOPs (Floating Point Operations), providing a consistent metric that is independent of the system on which the model runs. Pruning involves a tradeoff between model performance and efficiency. Conversely, lighter pruning may yield a highly performant network, but it could be larger and more expensive to operate. Different applications of neural networks require careful consideration of this trade-off especially when considering resource constrained devices.

Layer Fusion

Layer fusion, also known as kernel/operator fusion, stands as a crucial optimization in numerous cutting-edge DNN execution frameworks like TensorFlow or Pytorch. The primary objective of this optimization is to enhance the efficiency of DNN inference especially when targeting hardware accelerators. This can be described as a software optimization that has a direct impact on hardware computations.

Fusion concept — The fundamental concept behind this fusion technique aligns with traditional loop fusion performed by optimizing compilers [75], resulting in the following advantages:

- Eliminating the need for unnecessary intermediate result storage.
- Reducing unnecessary input scans.
- Enabling additional opportunities for optimization on hardware compilers.

In particular, in traditional deep learning models, each operation generates intermediate results that are stored in memory before being passed to the next operation. With operator fusion, these intermediate results are eliminated, and computations are performed on the fly as data flows through the fused layer. This reduces the memory footprint and the overhead associated with intermediate storage. Also, when operations are fused, redundant computations on the same input data are avoided. Last, by fusing operations, the resulting composite operation can offer new optimization opportunities. For example, the fused operation may allow for better utilization of hardware-specific features, such as specialized instruction sets or tensor cores, which can lead to further acceleration.

Fusion techniques — Generally, the deep learning model is represented as a computational graph or a sequence of operations. Each layer in the model is typically represented as a node in the graph. The fusion process begins by analyzing the computational graph to identify fusion opportunities, that is consecutive operations that can be efficiently combined into a single fused operation. After fusion, the computational graph is updated to reflect the changes. The nodes representing the individual operations are replaced with a single node representing the fused operation. During inference, the fused layer is executed instead of the individual operations, leading to reduced memory transfers and improved execution speed.

By combining multiple operations into a single fused layer, redundant computations and memory accesses are minimized, leading to faster and more memory-efficient execution of DNN models. These advantages are significant in accelerating the deployment of DNN models on various hardware platforms, including CPUs, GPUs, and FPGAs.

3.2.2 Hardware: Parallelism, Data Movement and Precision Scaling

In this section, we delve into the most important aspects of leveraging hardware resources to optimize neural networks. By utilizing parallel processing, optimizing data movement, and exploring precision scaling, researchers and engineers can accelerate the execution DNNs. These techniques serve as a standard for neural network optimization and often apply to different hardware architectures. This exploration of hardware-oriented optimization techniques focuses on *spatial architectures.* Spatial architectures are designed to process data in parallel, making them well-suited for tasks that require simultaneous computations on large datasets. It does not cover, however, other popular architectures which include more specialized optimizations such as neuromorphic or Processing-in-Memory (PIM) architectures as they are beyond the scope of this thesis.

Parallelism

Various techniques and hardware architectures have been developed to exploit parallelism effectively in DNNs. Below are the common categories of parallelism in DNNs from the level of the algorithm:

- *Data Parallelism:* Data parallelism involves distributing the training data across multiple processing units (e.g., GPUs or CPUs) and processing different data batches simultaneously. Each processing unit performs the same set of operations on its assigned data batch, and the gradients are aggregated and updated synchronously or asynchronously. This approach allows for faster training by parallelizing the computation across multiple examples.
- *Model Parallelism*: In model parallelism, different parts or layers of the neural network are distributed across multiple processing units. This is particularly useful when a DNN is too large to fit entirely in the memory of a single device. Each processing unit handles the computations for its allocated portion of the model, and data is communicated between the units as needed.
- *Pipeline Parallelism*: Pipeline parallelism splits the computation of a DNN into stages, and each stage is processed independently by different processing units. The output from one stage is passed to the next stage for further processing. This approach can reduce the memory requirements for intermediate data, allowing for larger models to be trained.
- *Layer-Level Parallelism*: Layer-level parallelism involves parallelizing the computation of individual layers within a DNN. GPUs or FPGAs for example, are well-suited for this type of parallelism, as they can efficiently handle computations for multiple layers simultaneously due to their massively parallel architecture.

DNNs' compute intensive part — In DNNs, the most important operation which is usually responsible for the most computations in the model is the matrix multiplication. In fact, *GEMM* (General Matrix Multiply) plays a crucial role in Deep Neural Networks as it is a fundamental operation used in many layers of neural network models, particularly in fully connected layers and convolutional layers. GEMM can be highly optimized and parallelizable, and its efficient implementation is essential for accelerating DNN training and inference. The most important optimizations commonly used for many accelerators in parallelizing GEMM operations include:

- Tiling/Blocking: Tiling or blocking the input matrices into smaller submatrices allows for efficient use of cache memory. By breaking down the large matrices into smaller tiles, the data can fit into the limited cache memory of the processing units, reducing the need for frequent data access to main memory. When using the blocking technique in a GPU, the submatrices or tiles of the input matrices in a GEMM operation are stored in the local memory of each SM. The smaller tiles can fit entirely within the local memory, allowing the GPU to perform computations on these smaller data chunks without the need to fetch data from the slower global memory repeatedly. When using the blocking technique in an FPGA, the tiling of matrices enables the data to be stored on the on-chip memory of the device, which is usually BRAMs or LUTs, minimizing data transfer latency and enhancing performance.

- Parallel Processing: Utilizing multiple processing cores or computing units to perform GEMM computations in parallel significantly speeds up the matrix multiplication. GPUs, for example, with the massive number of CUDA cores or with the most specialized Tensor cores distribute the workload of the matrix multiplication which can be computed independently by the rows or columns. Similarly, FPGAs can expose parallelism using multiple DSPs that operate in parallel or they can be partitioned into multiple computation units which can operate simultaneously at the task level. FPGAs can also implement pipelining, where the computation is divided into multiple stages, and each stage processes different data points reducing the critical path delay.

- Vectorization: Vectorization involves transforming scalar operations into vector operations, where a single instruction operates on multiple elements of a vector. This optimization is especially effective on SIMD-capable processors. GPUs, have SIMD units capable of executing the same instruction on multiple data elements (vectors) in parallel. FPGA also provide SIMD-like parallelism. FPGA vendors provide libraries and tools that allow developers to create SIMD-based designs to take advantage of parallelism in various applications.

Data movement

In neural network inference, the process of moving data between memory and the processing units can often become a bottleneck, limiting the overall throughput and energy efficiency of the system. To address this issue, hardware vendors and researchers have been working on various techniques to optimize data movement in neural network inference on different hardware accelerators. Several techniques we have already mentioned involve optimizing data such as quantization or layer fusion. In this paragraph, however, we focus on three major techniques regarding data optimization from a hardware perspective. The techniques are described below:

- *Data Reuse:* In neural networks, some data elements, such as weights and activations, are used multiple times during the computation. Exploiting data reuse can lead to significant reductions in data movement overhead. There are several techniques to achieve data reuse optimization:
    - Activation Reuse: In CNNs, the same activation maps are reused for multiple filter convolutions across different spatial locations. By reusing activations, hardware accelerators can avoid transferring the same data multiple times, reducing memory access and bandwidth requirements.
    - Weight Reuse: In convolutional or fully connected layers, the same weight values are reused across different input neurons. Similarly, in CNNs, weights are reused for different input channels. Hardware accelerators can take advantage of this property to minimize data transfers and improve inference speed.
    - Buffering: By buffering intermediate results and reusing them when necessary, data movement can be minimized. Hardware accelerators can store and reuse the outputs of certain layers to avoid recalculating them during subsequent operations.

- *Data Layout Optimization:* Data layout refers to how tensors (activations, weights etc.) are organized in memory. Optimizing data layout is crucial for maximizing data locality and minimizing data movement overhead. Some key techniques include:
    - Weight and Activation Tiling: Breaking down the weight and activation tensors into smaller tiles allows hardware accelerators to load only the relevant data needed for a computation, reducing unnecessary data transfers.
    - Blocked Data Formats: Using blocked or tiled data formats improves data locality, as the data is organized in blocks with contiguous elements. This reduces cache thrashing and enhances the efficiency of memory accesses.

- Transpose and Data Reordering: For certain operations or hardware architectures, reordering data elements can lead to more efficient memory access patterns, reducing data movement overhead. For instance, in the BLAS (Basic Linear Algebra Subprograms) library, which is widely used for efficient GEMM computations, data reordering techniques such as column-major ordering are commonly employed. Column-major ordering involves organizing the elements of a matrix in memory by storing the columns of the matrix in contiguous blocks, allowing for optimized data access patterns during matrix operations. This approach ensures better data locality and reduces data movement overhead, thereby enhancing the overall performance of GEMM operations on various hardware accelerators.

- *Memory Hierarchy Utilization:* Modern hardware accelerators often have multiple levels of on-chip memory hierarchy, such as registers, cache, etc. Efficiently utilizing these memory levels is crucial to minimize data movement between on-chip and off-chip memory. Some techniques include:
  - Data Prefetching: Prefetching data from off-chip memory to on-chip memory in advance of their actual usage can hide data transfer latencies and ensure data is readily available when needed.
  - Cache Blocking: Dividing the data into smaller blocks that fit into the cache can improve data locality and reduce cache misses, enhancing data reuse and minimizing data movement.

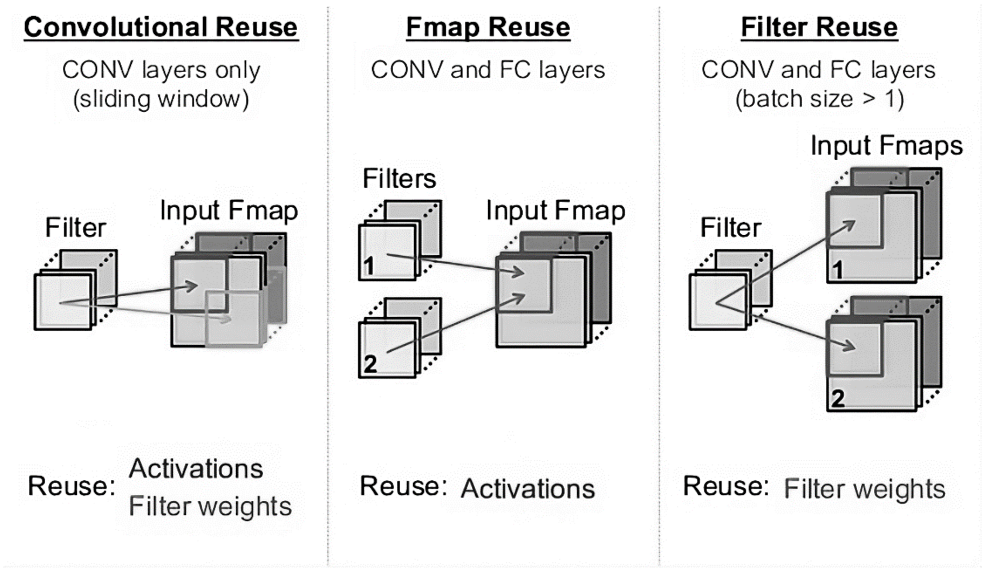


Figure 3-2. Data-reuse opportunities in DNNs

Precision Scaling

Precision scaling in hardware, especially in the context of neural networks, refers to the process of using reduced numerical precision to represent and compute values during various operations within a neural network model. We have covered reduced numerical precision optimization, namely quantization, from the software perspective. However, in this paragraph we focus on hardware-related techniques regarding precision optimization. This is typically done to achieve a balance between computational efficiency and model accuracy. As we have seen, in DNNs, many computations involve large matrices and tensors. The precision of these numerical values significantly impacts the computational requirements and memory usage of the hardware. Higher precision (e.g., 32-bit floating-point numbers) provides more accurate results but requires more memory and computational power. Lower precision (e.g., 8-bit fixed-point numbers or even 1-bit) uses less memory and computational resources but might lead to some loss of accuracy.

- *Exploiting 8-bit precision in hardware:* Using 8-bit for weights in a DNN is a common practice nowadays in computer vision applications. It often provides good tradeoff of performance and accuracy while having a broad support in many hardware accelerators.
    - *GPUs:* Modern GPUs, such as NVIDIA's Volta and Ampere architectures, come equipped with the specialized Tensor Cores that are designed to accelerate matrix multiplication operations. Tensor Cores can perform mixed-precision (FP16 and INT8) computations with significantly higher throughput compared to traditional floating-point units. Also, memory bandwidth often becomes a bottleneck in deep learning tasks, as fetching data from memory consumes time and energy. With 8-bit computations, more data can be loaded into the GPU's cache, reducing the need for frequent data fetches and optimizing memory utilization.
    - *FPGAs*: These devices inherently operate on fixed-point arithmetic, which is well-suited for implementing lower precision computations. Models need to be quantized and weights transformed to fixed-point formats before deployment on FPGAs. Dedicated processing pipelines can be designed for convolution, pooling, and other operations to achieve high throughput and low latency. Pipelining techniques can be applied more efficiently by reducing the overall critical path as FPGAs can handle more input samples in a given time frame. Techniques like loop unrolling and data reuse are commonly employed in these scenarios and DSP-specific optimizations can be tailored to provide more parallelism using 8-bit data. Last, with 8-bit fixed-point arithmetic, fewer FPGA resources are required such as LUT, BRAMs, etc. allowing more on-chip memory bandwidth.

- *Exploiting very low-bit precision in hardware:* Sub-8-bit DNN inference has been widely investigated from the research community, especially on customized hardware devices like FPGAs. It includes a range of numerical formats, such as 4-bit down to 1-bit representations. These formats use a reduced number of bits to represent the values of weights, activations, and intermediate computations within neural network layers. The most common very-low bit-width representation names are *ternary* and *binary*. Ternary weights can be -1, 0 or +1 but are less common in comparison to binary or higher-bit representations due to their limited range and the challenges they can pose in terms of training and computation. In contrast, binary weights or binary neural networks (BNNs) have been explored as a way to create highly efficient and low-power neural network models.
    - *GPUs:* Some GPUs, such as Nvidia T4 or A100 have support for 4-bit data formats. Several DNN frameworks have tried to incorporate the tensor core 4-bit inference scheme as an alternative in order to further reduce the computations and memory requirement for inference tasks. Naturally, however, Nvidia Tensor Cores have been designed for 8-bit inference, hence the reason for 4-bit format not being widely applicable.
    - *FPGAs:* In contrast to GPUs, FPGAs have greater flexibility in terms of customizing the design. Designers can use arbitrary bit-width from 1-bit to even 512-bit in a given register. This however comes with the cost of additional design time as careful calibration is needed to find the optimal ranges for these datatypes. For example, BNNs often involve XOR operations instead of the typical multiply and accumulate operation (MAC). FPGAs allow to design and implement custom XOR computation circuits tailored to the network's needs. This level of customization can lead to optimized hardware for XOR-related computations achieving very high parallelism from a single XOR instruction as it packs multiple data.

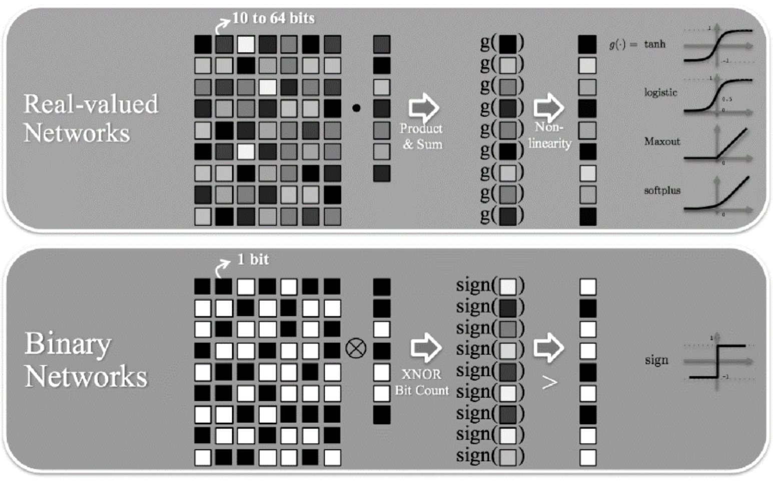


Figure 3-3. Binary Matrix Multiplication in Neural Networks

3.3 Accelerated Similarity Search using Vector Indexing

This section contains the first of the three scenarios in which we utilized FPGAs for AI acceleration. In particular, it concerns a novel integration of FPGAs into the popular FAISS (Facebook AI Similarity Search) framework [76] in order to accelerate the algorithm of similarity search. One of the most significant algorithms in ML employed for conducting similarity searches is referred to as the K-Nearest Neighbor algorithm (KNN). It finds extensive use in tasks such as predictive analysis, text categorization, and image recognition. However, this algorithm comes with a trade-off, often requiring substantial computational resources. To tackle this challenge, large companies dealing with large-scale datasets in modern data centers combine the KNN technique with algorithmic approximations, enabling the computation of crucial workloads on a real-time basis. Nevertheless, the computation demands and energy consumption escalate further when dealing with high-dimensional nearest neighbor queries. In this study, we introduce an innovative approach: a hardware-accelerated approximate KNN algorithm integrated into the FAISS framework through FPGA-OpenCL platforms. The FPGA architecture in this framework effectively addresses the intricacies of vector indexing during training and the incorporation of large-scale, high-dimensional data. The proposed solution leverages an FPGA-based in-memory format, which surpasses multi-core high-performance systems in terms of both speed and energy efficiency. Empirical experiments performed on the Xilinx Alveo U200 FPGA revealed significant results. The acceleration achieved is up to 98 times faster than a single-core CPU when utilizing only the accelerator, and the end-to-end system speed is improved by 2.1 times compared to a 36-thread Xeon CPU. Additionally, the design's performance per watt exhibits a notable boost of 3.5× compared to the same CPU, and 1.2× compared to a Kepler-class GPU.

3.3.1 Introduction and Related Work

In the era characterized by the immense expanse of big data, the operational requirements of modern data centers entail the processing of substantial workloads, often exceeding several terabytes of data daily. Notably, emerging machine learning applications deployed on the cloud have made remarkable strides, continually learning from large real-world datasets. Responding to the escalating computational complexities of these tasks, recent efforts have been directed towards augmenting their performance through specialized hardware. This is achieved by leveraging diverse heterogeneous architectures, including central CPUs, GPUs and FPGAs.

Addressing the surging demand for efficient and real-world execution of the latest-generation algorithms, the FPGA has emerged as potential platform. This architecture is distinguished by its high parallelization capabilities and adaptability, making it particularly suited for tasks characterized by repetition, such as the K-Nearest Neighbor (KNN) algorithm. Remarkably, numerous companies seek optimized solutions for achieving high performance while minimizing energy consumption costs, with FPGAs taking center stage in this transformative evolution [77, 78]. The field of KNN search within data centers is marked by the presence of big datasets and frequently high-dimensional inputs. Even state-of-the-art high-performance computers often struggle with these demanding requests, a phenomenon termed the "curse of dimensionality" [79]. To address the resource-intensive nature of standard KNN, the Approximate Nearest Neighbor (ANN) algorithm emerges as a solution, strategically managing computational complexity and data bandwidth through partial query searches [80].

However, the optimizations do not focus only on optimizing the query time even though it has a major importance. Novel data structures and algorithms that speed up KNN queries center around data point training, often employing specialized indexing techniques to enable efficient search of thousands of vectors without necessitating complete dataset access. Given the impracticality of exact results in vast databases, innovative techniques involving vector indexing or compression using quantization methods have been introduced [81]. The algorithms employed for constructing KNN graphs require substantial time effort, rendering them energy-inefficient and often ill-suited for the substantial scale of input vectors. Particularly when faced with statistically different data requiring frequent retraining, scalability becomes a significant challenge.

FAISS is an optimized library tailored for similarity searches of this nature, developed by Facebook [76]. FAISS includes approximate algorithms capable at managing big-scale inputs, effectively addressing the complexities of large datasets. Leveraging FPGA as a parallel platform presents an opportunity to alleviate this computational challenge through a specialized hardware-based solution, thereby elevating performance in these complex tasks. Implementing a finely-grained approach can leverage FPGA's hardware resources for superior performance, coupled with markedly reduced energy consumption – an attribute of utmost importance in modern data centers which try to operate on a lower energy footprint. Within this paper, we introduce an innovative implementation of the FAISS framework using FPGA technology, achieving a notable acceleration in contrast to alternative CPU multi-core solutions. More specifically, our contributions are as follows:

- We speed-up the index creation and addition of data points for the approximate KNN search by implementing an FPGA accelerator for Xilinx Alveo U200.
- We introduce an efficient FPGA integration for FAISS framework that can benefit directly from our hardware seamlessly.
- We successfully integrated and ran the full hardware accelerated framework on a Alveo U200 OpenCL-FPGA achieving superior performance and power efficiency compared with a CPU multi-core system.

Related Work — Numerous researchers have examined various KNN algorithms, particularly in the field of software and hardware optimizations tailored for this specific task. Moreover, large companies are actively exploring novel power-efficient computation methods, an aspect that held minimal significance in the past. Given the surge in processing large-scale data on cloud platforms, developers are compelled to devise ingenious heterogeneous architectures to facilitate the migration of these applications. Yuliang Pu et al. [82] introduced an enhanced KNN algorithm employing bubble sort within an FPGA-based computing framework. While their focus lies on query searches, their algorithm rests upon an exhaustive naive KNN implementation, which falls short in terms of performance when compared with an approximate solution. Even their proposed FPGA-accelerated approach fails to outperform the performance of FAISS executed on a conventional CPU. This is due to the fact that FAISS represents a CPU/GPU-optimized library designed for approximate similarity searches, delivering fast query results while maintaining generally negligible reduction in accuracy. Jialiang Zhang et al. [83] presented a technique for PQ-based approximated nearest neighbor search utilizing OpenCL FPGAs. Their focus lies in diminishing the codebook size to mitigate memory overhead while performing query searches. Yet, the computation cost of training/clustering, a crucial part of approximate KNN graph construction, is not reflected in the measurements. Furthermore, they provide only a partial implementation of the algorithm in pseudocode, without a detailed explanation of the hardware implementation/architecture of the accelerator. Notably, their proposed solution does not extend to FPGA implementation in cloud environments. Last, Hanaa M. Hussain et al. [84] introduced a K-means clustering methodology in FPGAs, tailored for processing large datasets. However, their implementation falls short when compared to the partition-based accelerator presented in this study, primarily concerning performance metrics. They report achieving a time of 0.0042 seconds per iteration for a dataset with N = 65500, K = 4 clusters, and D = 9 dimensions. K-means, characterized by its algorithmic complexity of $O(n \cdot d \cdot i \cdot k)$, translates to approximately 0.56 GFLOPs which is proportionately (of our FPGA resources) smaller than the performance we achieve in this work as we will see in the next paragraph.

Background

The FAISS framework employs an inverted indexing technique (IVF) as a preliminary step prior to conducting similarity searches through the clustering of dense vectors. In the ensuing section, we describe the operation of a conventional KNN algorithm and an approximate KNN approach within the FAISS framework. Subsequently, we present our FPGA implementation and the tools leveraged to develop the environment required for hosting the reconfigurable architecture of FPGA and application dataflow.

KNN [84, 85] stands as one of the most extensively employed machine learning techniques in scenarios including classification, recommender systems, and even financial research. This algorithm, when applied, returns the K-nearest neighboring points in relation to a designated object (referred to as a "query") within a given dataset. Each object's unique attributes, encapsulated within a data dimension D, define its distinct "weights" or "attributes" in the form of a specific vector. The decision-making process of this algorithm is frequently defined by the distances between the query points. In this context, the Euclidean distance prevails as the preferred distance metric due to its often-intuitive interpretation [86] and its computational scalability. Furthermore, the Hamming distance is commonly adopted when the input consists of discrete variables. In the field of Cartesian coordinates, the Euclidean n-space is described as follows:

$$\mathrm{dist}(x_i, y_i) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

Employing the aforementioned algorithm for exhaustive query searches, particularly with large datasets, results in a substantial number of operations. This is due to the necessity of computing distances between each point within the sample, leading to a considerable computational load. Consequently, the integration of approximate solutions becomes imperative to efficiently identify the K most probable nearest neighbors.

Approximate KNN — Practice says that an approximate nearest neighbor approach is almost as good as the accuracy of the exact solution, given that discrepancies in distance calculations are frequently insignificant. Consequently, KNN search becomes computationally viable even when dealing with extensive datasets and high-dimensional spaces. This feasibility is attributed to the substantial reduction in the total number of distance evaluations due to the implementation of approximation techniques. In the domain of approximate nearest neighbor search, two primary methods emerge, each emphasizing either on data reduction, dimension reduction, or a combination of both for query searches. One category revolves around spatial clustering-based techniques, with some approaches being K-means clustering [87] or hierarchical KD-trees [88]. These methodologies adopt a partitioning algorithm, constructing a k-nearest neighbor graph through the segmentation and clustering of samples into distinct regions. An alternative approach in the field of approximate nearest neighbor search entails hashing-based techniques like Locality Sensitive Hashing (LSH) [89]. Such methods group data points into "buckets" according to the distance metric. Near vectors are assigned to the same bucket, facilitating the algorithm's retrieval of the nearest neighbors to the target vector.

FAISS framework operation — Faiss leverages a variety of methods designed for performing similarity searches on dense vectors containing real or integer values. These methods identify vectors that are near a given query vector, which is achieved by minimizing the L2 distance or maximizing the dot product between the vectors. The underlying structure of Faiss involves employing diverse indexing approaches to store vectors, and the computation of distance involves multiple metrics, as mentioned earlier. The indexing methods within Faiss exhibit diversity, ranging from exact search methods to techniques like product quantization, which have demonstrated superior effectiveness compared to binary codes. This efficacy, however, is accompanied by a trade-off involving factors such as accuracy, search speed, training time, and memory consumption. Within this section, we will look into an exploration of fundamental indexing techniques employed within the Faiss framework. This exploration aims to provide a comprehensive understanding of the architecture before delving into the optimizing process via FPGA hardware. Our subsequent implementation, detailed in the following paragraph, focuses on utilizing IVFFlat indexing as a representative use case (best for high-accuracy regimes) but the design can be easily applied to other indexes such as IndexIVFPQ. Both approaches, particularly the latter one, exhibit slightly lower precision compared to exhaustive search. However, they have been demonstrated to effectively handle billions of vectors given ample memory resources on a single server.

I. *IVFFlat Indexing:* Several previous research works have exploited the characteristics of Voronoi diagrams to enhance variations of the nearest neighbor search [90]. A Voronoi diagram partitions a plane into distinct regions based on distances from points within a designated subset of the plane [91]. FAISS constructs the IndexIVFFlat index by establishing Voronoi cells through a codebook C_{coarse} in the d-dimensional space; each database vector is assigned to one of these cells. During the search process, only the database vectors y situated within the same cell as the query x, along with a few neighboring ones, are compared against the query vector. Consequently, two critical parameters govern the query process: *ncells*, representing the number of cells, and *nprobe*, denoting the count of cells (out of *ncells*) that are explored during a search. The concept of the number of cells aligns with the quantity of inverted lists, which could also be referred to as *nlist*. This probing mechanism operates as a partition-oriented approach rooted in Multi-probing (reminiscent of a variant of the best-bin KD-tree) [92]. The assignment of database vectors to cells is facilitated by a hashing function, notably K-means (closest query to centroids), and these vectors are stored within an inverted file structure. As a result, IVFFlat efficiently reduces the search space and significantly accelerates the search process compared to exhaustive search methods. However, to ensure result accuracy, it is imperative that the outcomes align with the Voronoi cells being visited during the search operation.

II. *IVFPQ Indexing*: This approach extends the concept of inverted indices by integrating them with product quantization, thereby circumventing the need for exhaustive search. The conventional product quantization method is referred to as PQ, and its non-exhaustive variant is termed IVFPQ [93]. Vectors continue to be stored within Voronoi cells, but their dimensions are reduced to a configurable byte count, denoted as 'm' (with the vector's dimension being a multiple of 'm'). The compression step introduces an additional layer of quantization, focusing on encoding sub-vectors of the original vectors. Since the vectors are not stored exactly in this scenario, the distances returned by the search method represent approximations and may exhibit somewhat lower accuracy compared to methods like IVFFlat. Additionally, when dealing with uniform data, this indexing approach encounters challenges due to the absence of inherent regularity that could be leveraged for clustering or dimensionality reduction. Nevertheless, the IndexIVFPQ structure proves highly valuable for conducting large-scale searches and holds potential for integration within our FPGA design. This indexing strategy offers a pragmatic solution for efficient search operations, making it well-suited for deployment in FPGA-based systems.

III. *LSH Technique:* Another widely recognized cell-probe approach is likely the original method of Locality Sensitive Hashing (LSH). This method aims to diminish the dimensionality of high-dimensional data by employing hashing to assign input items into akin buckets. Points that are in proximity to one another according to a specific distance metric (such as the Euclidean distance) are assigned to the same bucket with a high likelihood. While Faiss incorporates this algorithm, it does lack certain attributes present in other algorithms, including memory optimization. The incorporation of numerous hash functions introduces additional memory requirements, a factor that becomes impractical for extensive datasets hosted on cloud platforms. As a result, this approach is not particularly well-suited for the scope of this work due to the limitations posed by memory.

IV. *FPGA OpenCL framework*: The OpenCL specification within FPGAs comprises both host code and kernels. In this setup, the host is situated on an x86-64 CPU and manages the dataflow of the application. Our FAISS application, running on the host, takes advantage of the hardware kernels responsible for translating the OpenCL hardware abstraction into an FPGA implementation within the device fabric. For this task, we utilized the SDAccel environment [94], which provides both software and hardware emulation capabilities. Through careful examination of system reports and checking the device data tracing and application's timeline, we identified potential bottlenecks within our design down to the granularity of clock cycles. Thus, this analysis allowed us to finalize the optimized application, seamlessly integrating the hardware into the FAISS framework.

3.3.2 Implementation and Results

To determine which components of Faiss should be prioritized for hardware implementation, a comprehensive framework profiling was conducted. This section begins by detailing the process involved in this profiling, during which the IVFFlat index was chosen as a representative use-case. Subsequently, the hardware algorithm is specified, along with optimizations executed on both the host and kernel sides to optimize throughput and minimize overall design latency. The final portion of this section delves into the novel custom framework. This framework employs an FPGA with optimized memory transfer scheme which seamlessly integrates our designed accelerators to ensure minimal latency overhead during memory transactions.

Framework Profiling — Both prior research and our own practical experience have demonstrated that IVF indices generally exhibit substantial speed and accuracy. Notably, among the various indexing techniques employed in Faiss for typical scenarios, IVFFlat stands out as the most proficient technique. Subsequently, we employed call-graph techniques offered by profiling tools such as valgrind/callgrind to identify memory and computational bottlenecks. Functions consuming the majority of execution time present suitable candidates for offloading and acceleration onto FPGAs (see Figure 3-4). Profiling outcomes revealed that the most substantial computational workload arises during index creation and data addition to the index. This holds true even when considering modest query searches encompassing thousands of vectors. This phenomenon stems from the fact that while approximate nearest neighbor search is highly efficient, it incurs extended training times, especially when numerous cluster points are required to sample the dataset. Notably, the training algorithm incorporates numerous Multiply-Accumulate operations (MAC), which Faiss presently executes using CPU-optimized BLAS routines by default. Given the prevalent characteristics of these algorithms, it's very common to map them for hardware, owing to their high operations-to-bytes transferred ratio.

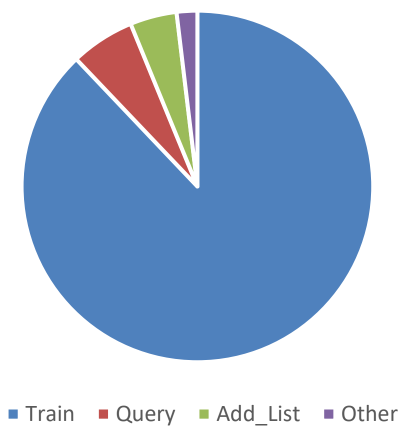


Figure 3-4. IVF indexing (Flat) workload distribution

Optimization schemes — Consideration of specific hardware design principles was imperative to ensure an efficient FPGA implementation of the accelerator. As an initial step, these principles include the thoughtful design of both the x86 host and kernel aspects of the accelerator function. This dual-pronged approach was aimed at effective alignment with Faiss operation while maintaining application correctness. Moreover, due to the significant importance of memory optimization, we dedicated efforts to streamline data movement between the Linux host and global memory, as well as between global memory and kernels. Furthermore, careful attention was directed toward designing the FPGA kernels in a manner that minimizes latency within our custom logic. To elaborate on the specifics:

1. *Defining the accelerator:* The key transformations of high-level synthesis lie in the host code. To establish a scalable function for acceleration, we devised a bespoke column-major General Matrix Multiplication (GEMM) routine, where the first matrix is transposed:
$$C = alpha * T(A) * B + beta * C$$
This approach optimizes hardware efficiency by accessing both arrays with consecutive elements along the second dimension, in line with the data storage format. Subsequently, we determined the matrix dimensions for our function's application, particularly during index creation and data addition phases. Within Faiss, the GEMM's first input is characterized by an $nlist \times vector$ dimension matrix, while the second input comprises a vector $dimension \times centroid\ points$ matrix. Notably, the integer *nlist*, often small compared to the database, signifies the number of inverted lists, as previously described. This value, generally a multiple of $\sqrt{dataset}$, (e.g., IVFFlat4096), informs the number of lists. Additionally, the vector dimension (*vector_dim*), representing the dimension of the vectors, typically remains modest (below a thousand) due to data dimension compression. The larger matrix dimension pertains to centroid points (*centroid_points*), denoting the points sampled for clustering in each iteration. Our algorithmic design and data flow are intricately constructed around these facts, ensuring maximum optimization aligned with these guidelines.

2. *Optimizing data movement*: A pivotal observation we made was that GEMM is executed multiple times within each iteration of the training process, accessing segments of the B matrix. This realization underpins our strategy, which hinges on a blocking technique. Moreover, our theoretical understanding confirms that each iteration employs consistent clustering data (total centroid points), leading us to infer that the B matrix remains constant across iterations. To obviate redundant data transfers to global memory during each iteration, we made a one-time transfer of the aggregate centroid points to DDR once in the initial iteration. This approach not only eliminates memory requests for the same data but also

facilitates accelerated burst data transfers to global memory, benefiting from larger memory chunks being transferred at higher rates. To further enhance data throughput, we implemented a 512-bit user interface on each kernel side, aligned with the maximum memory bandwidth supported by our FPGA through OpenCL vector datatypes. By employing the float16 datatype, we maintained peak accuracy for the dataset while abstaining from introducing further layers of approximation into the Approximate Nearest Neighbor (ANN) search process. By leveraging all four DDRs of the device for read-write operations, we achieved optimal memory transfer rates spanning between host-DDRs and DDRs-kernels. It's worth mentioning that careful allocation of kernels to Super Logic Regions (SLRs) was undertaken. Our design placement was strategically arranged to ensure that SLR resource limits were not surpassed, and kernels were matched with the memory banks to which they exhibited the most connections according to their respective SLRs. This deliberate placement strategy avoids SLR crossings and mitigates critical path complexities, which often translates to inefficient synthesis outcomes and unnecessary power consumption. Lastly, for improved communication efficiency, we allocated matrix blocks within physically contiguous memory utilizing on-chip Block RAMs (BRAMs), strategically positioned near kernel computations. This arrangement ensured high speed communication, enabling one-cycle read-write operations of 512-bit data, optimizing the performance using the most efficient data movers.

3. *Optimizing kernel*: To achieve substantial throughput, our focus centered on introducing a significant level of finely-tuned parallelism in application execution within the Programmable Logic (PL) fabric. This entailed mitigating data dependencies. Employing appropriate OpenCL directives like 'pipeline' or 'unroll,' we formulated a highly parallel and pipelined architecture characterized by minimal latency. This architecture exhibited exceptional efficiency in executing Multiply-Accumulate (MAC) operations. By ensuring an initiation interval (II) of 1 for each loop and utilizing the dataflow directive, the kernel operated optimally, rapidly consuming incoming data from memory interfaces and rapidly writing results back to DDRs. Moreover, we identified that creating larger and fewer Compute Units (CUs) to access global memory chunks yielded increased efficiency compared to generating multiple smaller CUs. This strategic choice averted excessive FPGA resource utilization and area consumption, mitigating timing failures. Furthermore, we generalized our implementation to host multiple FPGAs. This entailed an even distribution of the workload and automatic synchronization of the dataflow across any number of kernels. Our design, tailored for a single kernel parallelizes up to 192 output data achieving the maximum of 300 Mhz, thus in every cycle 192 output elements are produced, translating into $2 \cdot 192 \cdot 300$ MHz = 115 GFLOPS for a single kernel.

Final system design — In order to integrate the hardware accelerator into Faiss and transfer it to the FPGA, a necessary step was to export it as a shared library. This library was then connected to the rest of Faiss framework through library linking during compilation. To ensure seamless compilation of the entire framework using the SDAccel compiler and loading successfully the required OpenCL runtime libraries, we performed targeted adjustments to the framework *Makefile*. The FPGA API operates underneath the Faiss framework, and enabling it involves a simple switch from the 'SW' to 'HW' build target option. Within the FPGA memory model of Faiss, relevant Faiss functions are mapped onto FPGA memory, granting access to device specifications across all framework source files. This setup enables any function to execute in HW mode immediately and seamlessly. Moreover, all cluster points resided in the FPGA global memory throughout the application execution, with hardware accelerators in Faiss accessing the necessary memory segments as required. The optimized CPU BLAS function, "sgemm" was replaced with a customized function in our implementation. This custom function meticulously handles input data manipulation and utilizes OpenCL task synchronization through command queues on the host side. A representative hardware dataflow is depicted in Figure 3-5, showcasing the concurrency achieved via OpenCL Command Queues, as well as the synchronization established between the host and kernels. These elements were carefully implemented to operate concurrently in alignment with FPGA design principles, thus achieving elevated performance levels.

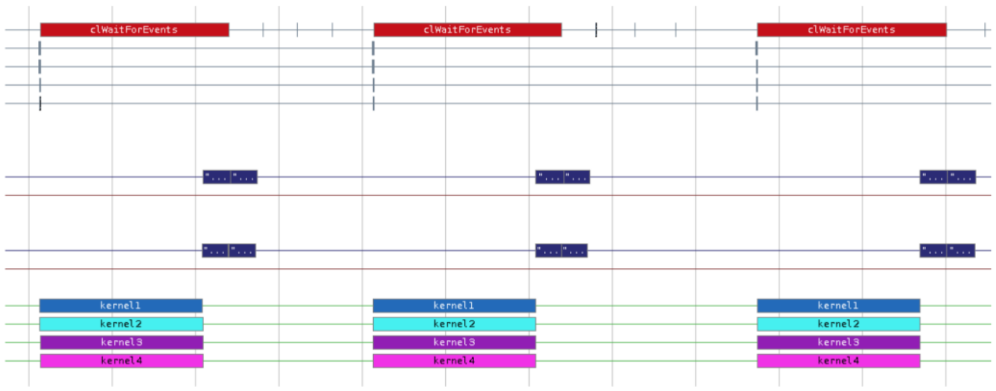


Figure 3-5. Kernel Synchronization

Also, on the following Figure 3-6, we show the illustration of the end-to-end application from SW to HW. Host code optimization (concurrency from OpenCL Command Queue), buffer management regarding data exchange between the host and kernels, general pipelining on the FPGA, and synchronization between host and kernels were all precisely constructed according to FPGA design principles for high performance. As depicted in the figure, the query vectors are loaded from the host x86 CPU and communicate with the dynamic library of Faiss which we have linked with the rest of the framework. Also, the inverted index is loaded in order to fetch the results according to the clusters.

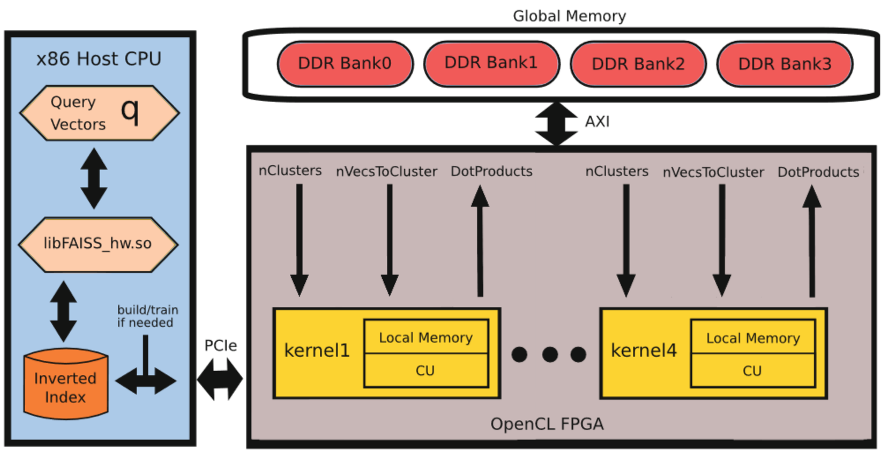


Figure 3-6. Software and hardware dataflow

Evaluation and Results

To assess the design, we verified the accuracy of the accelerator and quantified its performance. Subsequently, following the integration with Faiss, we conducted assessments of the final system's precision and efficiency using real-world data. Additionally, we measured power efficiency in comparison to alternative systems like CPU and GPU setups. The configuration of the system was done on OpenCL-FPGAs, specifically employing an Xilinx Alveo U200 datacenter card equipped with four DDR4 channels. This was coupled with a host system utilizing a Xeon CPU. For a comprehensive like-for-like comparison against a high-performance CPU, we selected a c4.8xlarge instance from AWS Cloud, equipped with a Xeon CPU with 36 vCPUs and 60 GiB of RAM. Remarkably, this instance bears an equivalent cost (per hour) to an f1.2xlarge instance featuring a similar FPGA, the VU9P. Moreover, for the conclusive evaluation of the final system's performance, we assessed the efficiency in terms of performance per watt against both the same CPU and a Kepler-class K40 GPU. Our FPGA hardware design maximizes the utilization of all DDRs and optimally leverages the resources within each SLR. However, the primary limitation to further scalability resides in the routing across the three SLRs. The resource utilization of a single kernel on the FPGA device is outlined in

Table 3-1 for reference.

Utilization summary				
Name	**BRAM**	**DSP**	**FF**	**LUT**
Total	502	1036	156137	89206
Percentage (%)	11	15	6	7

Table 3-1. FPGA resource utilization per kernel

Accelerator performance — To assess the design's effectiveness, our initial step involved validating the accelerator's accuracy and quantifying its performance on the Alveo FPGA. To thoroughly test the hardware functionality, we simulated Faiss inputs across diverse dataset scenarios. This involved generating random cluster data across various list sizes. Figure 3-7 illustrates the notable speed-up achieved exclusively by the FPGA accelerator, with enhancements reaching approximately 98× for larger list sizes. The arrow shows the maximum speed-up achieved when compared with a single-core CPU. This comes from the fact that for more cells the impact of data transfer is less evident. It's worth mentioning here that the lower efficiency value which happens to be from ∼500 cells and below does not impact the overall performance of the algorithm. Usually in real-world datasets, especially in larger ones used in data centers, the Voronoi cells are multiples of thousands for satisfactory clustering, even for a modest 1-million dataset.

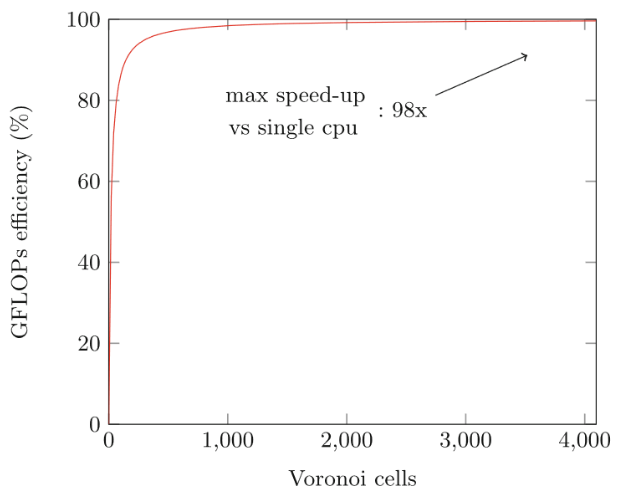


Figure 3-7. Hardware accelerator efficiency

Final system and evaluation — In the final evaluation of the system's performance, our initial comparison involved measuring the end-to-end execution against the previously mentioned 36-thread Xeon CPU, resulting in an approximate 2.1× speed-up. To provide a demonstrative scenario, we employed a million-scale dataset, specifically the SIFT 1M dataset, a very popular dataset that comfortably fits within available RAM. In Figure 3-8, we perform an accuracy assessment on this dataset using our FPGA design to evaluate the algorithm's efficacy with real-world data. The evaluation of KNN models generally employs the recall measure, which calculates the ratio of correctly predicted positive observations to all observations in the actual class. However, we use the more appropriate "$R - recall\ at\ R$" also known as intersection, to assess the effectiveness of our custom Inverted Index method. This measure quantifies the fraction of the R nearest neighbors found by the model that are within the ground-truth R nearest neighbors, with R set to 100 in our case.

In the provided figure, we analyze the accuracy of two Inverted Index (Flat) methods using varying probe values. Notably, constructing the IVF4096 index consumes more time due to a larger number of cells to train (4096 compared to 256). However, even while maintaining the same accuracy values (y-axis), the query search on IVF4096 is considerably faster. This efficiency is attributed to a smaller fraction of the database being compared to the query (nprobe/ncells). For instance, to sustain 0.6 accuracy, probing involves $\frac{2}{256}$ cells of the dataset on IVF256, whereas IVF4096 requires only $\frac{8}{4096}$ cells from the dataset, as illustrated in Figure 3-8. From these observations, we conclude that investing computational time in a robust index-building process yields more efficient query results. Consequently, by adopting this approach, users can achieve faster similarity search outcomes. As a result, the index-building algorithm, which was a focal point of acceleration in our FPGA design, plays an undeniable and pivotal role in enabling more efficient search capabilities.

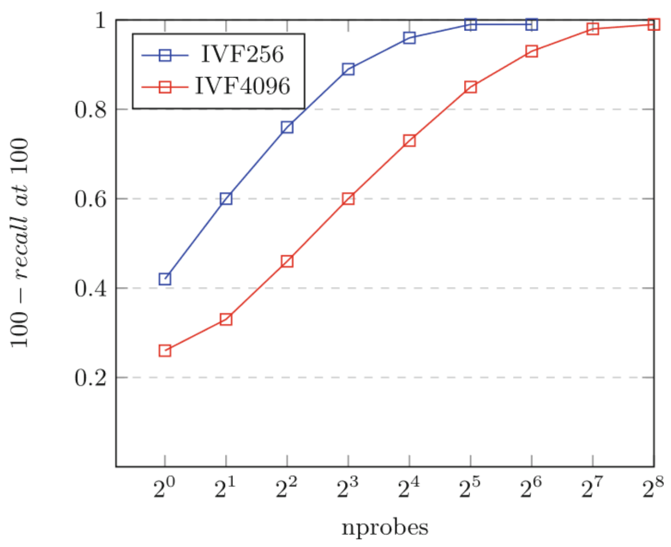


Figure 3-8. Inverted index accuracy for different probe values on SIFT 1M

Subsequently, we advanced to conducting actual real-world measurements of our FPGA board within a live environment, utilizing AWS metric tools to encompass data transfers. Following this, we engaged in a comparison of the measured performance per watt from our hardware design with the theoretical maximum performance per watt attainable by the Xeon CPU and a K40 GPU. The outcomes, as depicted in Figure 5, exhibit a distinct advantage for our FPGA architecture. For the other devices, we employed the following equation to facilitate a comparison of their potential maximum power efficiency against our FPGA design. This equation was utilized to determine how efficiently those devices could perform in relation to our FPGA architecture.

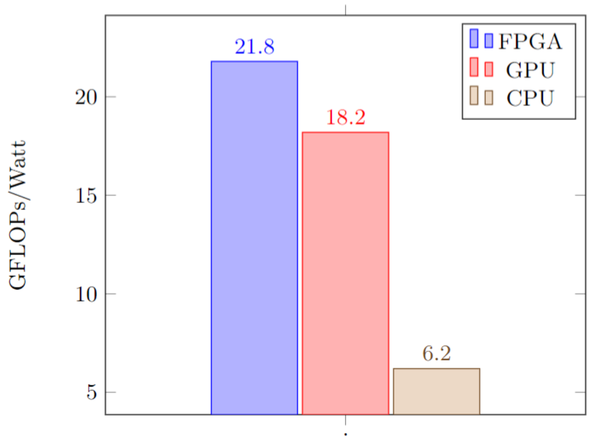


Figure 3-9. Power efficiency comparison

Conclusion — Hardware accelerators have the potential to significantly enhance the performance of ML applications. However, numerous frameworks, including Faiss, lack transparent support for effectively incorporating such acceleration modules. This study introduced an innovative approach that seamlessly integrates FPGA hardware into the popular Faiss framework for large-scale similarity searches, a vital component of cloud computing. Our findings demonstrate that our hardware accelerator surpasses the capabilities of a 36-thread Xeon CPU. Furthermore, it exhibits superior performance per watt in comparison to both the same CPU and a Kepler-class GPU. This underscores the effectiveness of a software/hardware codesign approach for addressing the demands of cloud computing workloads, specifically in scenarios like persistent indexing times for approximated KNN algorithms. The increased performance and efficiency of our design hold the potential to revolutionize the utilization of FPGA hardware in cloud environments and expansive data centers, given the growing significance of power efficiency amidst escalating workload requirements. Looking ahead, in order to solve the memory issue of billion-scale datasets which was our only restriction, the distributing of the application to a number of FPGAs is needed. Our algorithm was designed in such a way that the host application can easily distribute the dataset and thus the workload on a number of FPGAs on the cloud which due to limited infrastructure at that time we could not accomplish. Despite the limitations imposed by the available infrastructure at the time of this study, this distributed approach remains a promising avenue for future exploration.

Acknowledgement: This project has received funding from the Xilinx University program and the Hellenic Foundation for Research and Innovation (HFRI) and the Genal Secretariat for Research and Technology (GSRT), under grant agreement No 2212-Hardware Acceleration of Machine Learning Applications in the Cloud.

3.4 Accelerated Image Reconstruction using GANs

This section contains the second of the four scenarios in which we utilized FPGA acceleration for AI execution. Precise and efficient ML algorithms hold immense significance across various challenges, particularly in tasks involving classification or clustering. In recent times, a novel category of Machine Learning known as Generative Adversarial Networks (GANs) has emerged. GANs operate using two neural networks: a generative network (generator) and a discriminative network (discriminator). These networks engage in a competitive process with the objective of generating new unseen data, such as images. For instance, a GAN can reconstruct an image that is corrupted by noise or contains damaged segments. This image reconstruction concept has diverse applications in computer vision, augmented reality, human-computer interaction, animation, and medical imaging. Nonetheless, this algorithmic approach demands a substantial number of MAC (multiply-accumulate) operations and consumes considerable power to function. In this section, we describe the implementation of an image reconstruction algorithm utilizing GANs. Specifically, we focus on training a model to restore images of clothing utilizing the fashion-MNIST dataset as a case study. Furthermore, we deploy and optimize this algorithm on a Xilinx FPGA SoC. These platforms have demonstrated notable proficiency in effectively addressing such challenges in terms of performance and power management. The designed approach also outperforms CPU and GPU setups, achieving an average reconstruction time of 0.013 milliseconds per image and a peak signal-to-noise ratio (PSNR) of 43 dB on the FPGA's quantized configuration.

3.4.1 Introduction and Related Work

In the era characterized by the prevalence of big datasets, modern applications spanning from edge computing to cloud-based solutions confront the substantial challenge of processing several terabytes of data on a daily basis. Emerging machine learning algorithms, notably Neural Networks (NNs), which continuously learn from real-world large-scale data, have exhibited remarkable progress, largely attributed to their ability to achieve high levels of accuracy. Recently, a fresh category of Machine Learning, named as Generative Adversarial Networks (GANs) , was introduced by Ian Goodfellow and his collaborators [10]. Within this framework, two neural networks engage in a competitive game resembling a zero-sum scenario, where gains for one agent are losses for the other. Essentially an unsupervised learning task, GANs employ provided training data to acquire the skill of generating novel samples mirroring the statistical properties of the training dataset. This includes the capacity to reconstruct incomplete data, such as partial images. The setup involves a generator model trained to create new instances, and a discriminator model tasked with categorizing instances as genuine (pertaining to the domain) or counterfeit (generated). The applications of GANs have rapidly gained substantial traction, particularly in domains like science, video games, and computer vision, owing to their versatile capabilities [95].

More specifically, the focal point of our work is image reconstruction, a domain that has found application across multiple sectors. Notably, it is utilized in computer vision and image processing, including tasks like resolution upscaling. Moreover, it plays a pivotal role in human-computer interaction, including tasks such as face reconstruction, and extends its utility to medical imaging where it aids in the early diagnosis of incomplete medical images [96]. For these challenges, the utilization of cutting-edge models such as GANs presents a marked improvement in accuracy and output quality compared to older techniques. However, the surge in demand for efficient and rapid processing of the latest generation algorithms, like CNNs (or GANs in our context), has spurred efforts to optimize their performance through hardware-specific enhancements. This involves harnessing the potential of heterogeneous architectures, including CPUs, GPUs, and FPGAs [97]. FPGA implementations have made remarkable strides, proven to be exceptionally effective in tasks involving CNNs due to their inherent parallelism and configurable nature at the bit level [98, 99]. This architecture aligns well with tasks characterized by repetition, such as the computations within GANs. As demonstrated in our work, the incorporation of hardware accelerators enables us to achieve low latency and substantial overall throughput. Notably, these high-performance platforms also excel in power efficiency, a critical factor for both edge and cloud-based workloads [100, 96].

However, GANs represent a relatively recent yet highly significant area within this domain, and scant prior research has combined GANs with hardware acceleration. As a result, this study introduces an innovative approach: the deployment of GANs on a small embedded Xilinx FPGA SoC, which not only achieves high-quality image restoration but also operates within a short timeframe. To summarize, the primary contributions of this work are as follows:

- Development and Training of GAN Model: We construct and train a GAN model with the ability to generate novel, previously unseen images, employing clothing images as a practical use case. Through a series of key modifications, we adapt this model to excel at reconstructing images with an impressive degree of precision.
- Hardware Architecture Implementation: A hardware architecture is devised for the Generator model, specifically tailored for a Xilinx Zynq 7000 FPGA. This architecture is optimized to accelerate the image reconstruction algorithm, involving enhancements in host processing, memory management, and kernel operations.

- Performance and Quality Evaluation: The restored images are subjected to comprehensive evaluation in terms of both performance and quality. This evaluation encompasses varying bit precisions within the hardware configuration. Furthermore, the outcomes are compared against alternative platforms, such as CPUs and GPUs.

Related Work — Significant prior research within the domain GANs, especially in conjunction with FPGAs, is relatively scarce due to its novelty within the research community. Nevertheless, a number of earlier studies have tackled image reconstruction algorithms by leveraging hardware acceleration. In this paragraph the related work will include a very similar problem domain and compare our contributions with previous work in terms of quality of results and acceleration speed. In a study by Ghasemzadeh et al. [101], they introduce a reconfigurable design for tomographic image reconstruction, aimed at a Xilinx Virtex 2 Pro FPGA. Their focus centers on a reconfigurable design of filtered backprojection (FBP) for parallel beam imaging, achieving an operational frequency of 144MHz while utilizing nearly 14% of FPGA resources. However, the reconstructed image presented in their paper exhibits a comparatively more pronounced reduction in quality when contrasted with images generated using GANs, as demonstrated in this study. S.O. Memik et al. [102] explored FPGA implementation of an iterative image restoration technique. Their investigation includes various metrics like result quality and speed, concentrating on a Xilinx FPGA platform. Specifically, their highest reported speed for a single image restoration, with kernel execution only, using their largest FPGA, stands at 0.28 seconds for a 256×256 image. Even if we extrapolate this acceleration to our image size (28×28), our design showcases relatively higher speed. S. Kumar [103] developed a noise reduction algorithm tailored to eliminating diverse types of noise, particularly from digital images, with remarkable accuracy. Employing an approximated fractional integrator (AFI) on grayscale images, they propose a hardware implementation on an Artix-7 FPGA. Although they validate accuracy, no explicit performance outcomes are provided. There have been some earlier examples of combining GANs with FPGAs, such as in works like [104] and [105]. These works present memory-efficient architectures to accelerate the generator and/or discriminative network of GANs. Although this aligns closely with our problem domain, they showcase performance results on significantly larger FPGAs, with no explicit application testing or mention. While these studies meticulously outline FPGA architecture designs, they lack details concerning application outcomes or the quality of GAN-generated content. To summarize, some preceding research has addressed image restoration algorithms through FPGA implementations but falls short in matching the quality and/or speed of outcomes achieved through our GAN-related work. Others have delved into GAN-related concepts on FPGAs but omit application-specific details or insights into the quality of generated outputs. In subsequent sections, we will introduce a unique FPGA task that hasn't been previously explored, a GAN generative model designed for partial image restoration.

3.4.2 Implementation and Results

In the following paragraph we describe the optimization aspects of the software and then the hardware design of our GAN-related application for image restoration.

Software Implementation — We are going to outline the software-level design of the application. As previously indicated, we have conducted training on two custom MLP (multilayer perceptron) networks – one for the generator model and another for the discriminator model. The training was performed on the fashion-MNIST dataset, which poses a slightly greater challenge than the standard MNIST dataset. The primary generative model produces novel image samples, and by implementing several adjustments, we have configured the GAN model to reconstruct partial images. Furthermore, we have optimized the model parameters in a memory-efficient manner to ensure they can be accommodated within the FPGA on-chip BRAMs (Block RAMs) without causing any substantial loss in quality.

- *Model Parameters:* The discriminator is composed of an MLP featuring a 4-layer architecture. Each layer consists of a Dense layer, a LeakyReLU activation function, and a dropout layer. The final layer employs a Sigmoid activation function, featuring in a total of 1.5 million parameters. Similarly, the generator model, which excels in generating synthetic images, also adopts a 4-layer configuration. However, it incorporates a *Tanh* output layer and contains a total of 1.1 million parameters. It's important to note that these parameters have been notably reduced for deployment on hardware, which will be elaborated upon in the subsequent sections.

- *Data Reconstruction Technique:* During the training of the GAN, the generator tries to approximate a particular distribution, while the discriminator evaluates its performance, resulting in mutual iterative enhancements. The generator is supplied with random noise during each iteration to create random samples adhering to the distribution. However, in the context of image reconstruction, instead of random noise, we feed the generator with half of an image from the dataset as input, expecting it to generate an approximation of the missing half. For the generator model, both the training and test sets have been modified to only include the top half of the images. Ultimately, the outputted half-image is combined with the original counterpart to create a complete image, which is then used to train the discriminator model. The training process for the generator model is in line with the fundamental principles of the typical discriminator model, utilizing binary cross-entropy loss (as indicated in the next equation). The primary objective here is to devise a model that maximizes the likelihood of the training data.

$$H_{y'}(y) := -\sum_{i} (y'_i \log(y_i) + (1 - y')\log(1 - y_i))$$

- *HW-aware training:* To adapt the neural network model for FPGA synthesis, we undertook several hardware-friendly adjustments within the model architecture itself. Firstly, the substantial parameter count of the generative model (1.1 million) necessitated reduction to ensure compatibility with the FPGA's on-chip memory, thus maximizing data bandwidth. To achieve this, we downsized the network from four layers to three, as well as reducing the number of neurons. Despite these modifications, we retained the ReLU activation function, which is generally suitable for hardware implementation. Consequently, this yielded a mere 32,000 parameters, demonstrating exceptional memory efficiency without significant compromising of outcomes. Additionally, it's important to highlight our application of a *MinMax* constraint during the Keras training phase. This constraint limits weight values to a narrow range, specifically within the interval of (-2, 2) (as depicted in Figure 3-10). By doing so, the necessity for extensive bitwidth in multipliers is obviated, leading to a reduction in overall resource usage. Lastly, for the final output layer, we employed a Tanh function. While this poses slightly more intricacy in hardware implementation compared to ReLU, we successfully implemented it on the FPGA through the use of a pre-computed value table. The Tanh function's output neuron count, totaling 392, is derived from the dimensions of the predicted half-image ($\frac{28 * 28}{2}$). This design approach ensures efficient hardware compatibility while achieving the desired functionality.

In the following Figure, we can observe the loss achieved for discriminator and new generator model. Also, at the bottom of the Figure we can see the parameter range for each layer as acquired from the MinMax constraint we already mentioned.

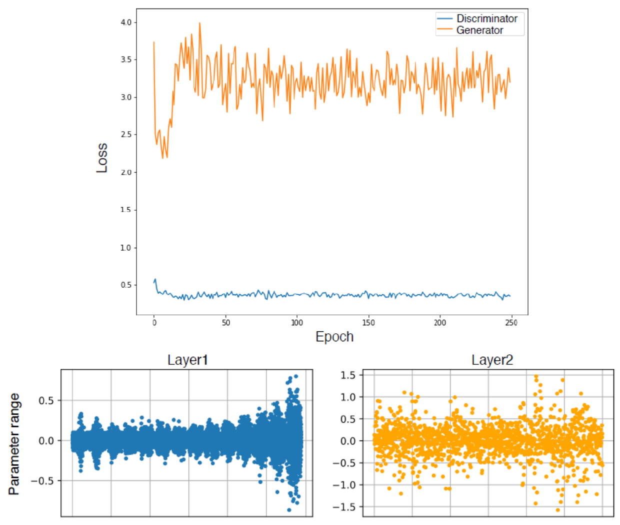


Figure 3-10. Training Results (top: Loss for Discriminator and Generator model, bottom: illustration of parameter range in each layer)

Figure 3-11 displays a collection of diverse images captured during both the initial and final epochs of training. Evidently, in the initial epoch, the lower halves of the images (which constitute the output generated by the generator model) appear as if they consist of random noise. However, in the concluding epoch (on the right), the showcased images constitute a compilation of complete clothing ensembles, bearing a strong resemblance to actual clothing.

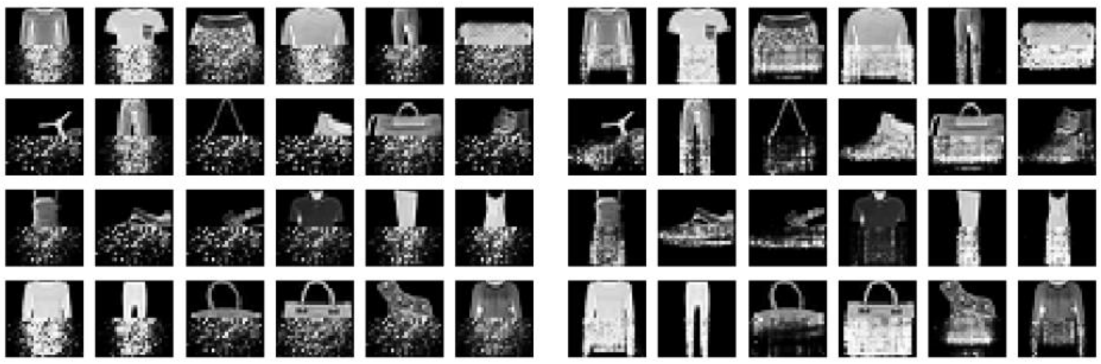


Figure 3-11. Generator results for image reconstruction from first (left) and last

Hardware Implementation — The primary objective behind FPGA acceleration was to enhance the speed of the generator model responsible for image synthesis and reconstruction. To ensure the creation of a fast reconfigurable design, we executed the implementation of a memory-efficient neural network, as explained in the preceding paragraph. This network's layers were then synthesized onto the hardware, employing a pipelined approach. The optimization of the host code, efficient management of buffers for enabling data exchange between the host and kernels, general FPGA pipelining strategies, and specific synchronization mechanisms between the host and kernels were all meticulously orchestrated in according to FPGA design principles for high performance. The final configuration of the system is presented in Figure 3-12.

- *Host optimizations:* The initial phase involves refining the design of the host component of the accelerator function to align effectively with the requirements of the application. The accelerator takes in a 14×28 image, which is then processed. The input image is stored in contiguous memory using C++ vectors, facilitating the utilization of the most efficient data transfer mechanisms for the accelerator. The outcome produced by the accelerator is the projected 14×28 image that was previously absent. This result is subsequently transferred back to the host system, where it's combined with the input image to yield the final reconstructed 28×28 image.

- *Memory optimizations*: Enhancing the data movement and the organization of memory is pivotal to achieving high performance levels. Through the strategic partitioning of BRAMs (Block RAMs) within the fabric, we attained optimal data bandwidth. This enabled the instantiation of a greater number of DSPs (Digital Signal Processors) capable of parallel operation, with simultaneous access to the model's weights. Additionally, we adopted fixed-point arithmetic for the MAC operations during model quantization. This switch from floating-point arithmetic ensured more efficient hardware synthesis for our design.

- *Kernel optimizations:* For the purpose of achieving substantial throughput, it was necessary to activate a high degree of finely-grained parallelism during application execution within the Programmable Logic (PL) fabric. This was achieved by strategically circumventing data dependencies. By utilizing appropriate pragma directives, we constructed an architecture for the *Generator* model that was entirely parallel and pipelined, minimizing latency. Furthermore, each layer's execution overlaps with the subsequent layer's operation in a dataflow manner. Intermediate results are transferred downstream without needing additional memory.

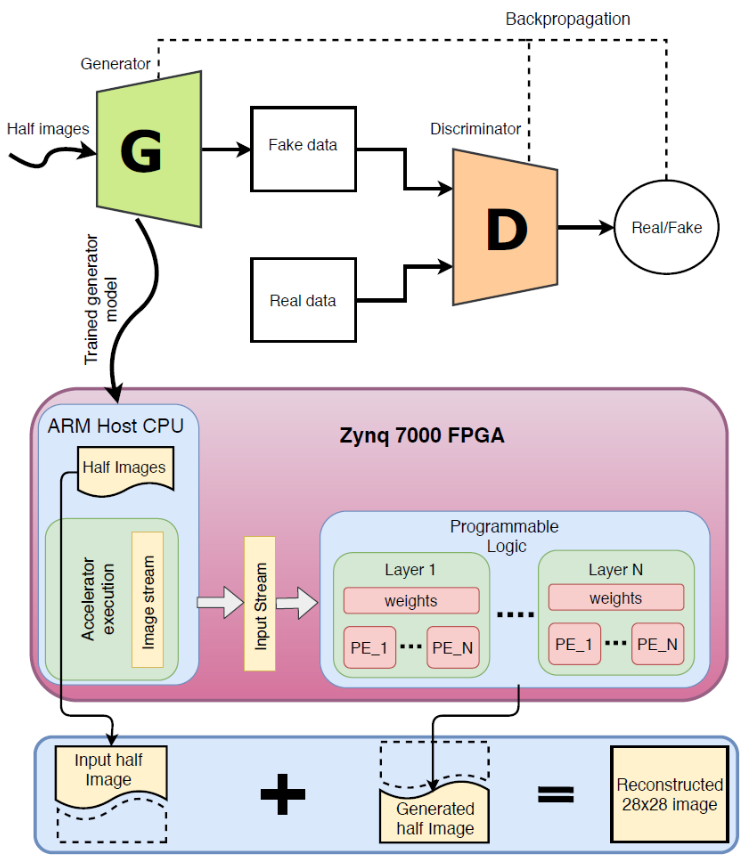


Figure 3-12. Final System design for GAN image reconstruction

Evaluation and Results

In this paragraph, we assess and analyze our application. Our evaluation will initiate with an examination of the reconstructed images yielded by the GAN model in hardware, involving a comparative study of errors across various bit precisions. Furthermore, we will provide an evaluation of the hardware accelerator's performance, resource utilization, and power efficiency. This assessment will be compared against other platforms.

I. *Assessment of the Model:* Our evaluation strategy advances to the model examination phase. Figure 3-13 presents a visual representation of the image quality generated by the Generator model through diverse fixed-point precisions in the hardware setting. The perceptible outcome is the variation in the quality of the lower portion of the image, which is subject to approximation, dependent on the bitwidths of the multipliers in the FPGA. After careful consideration, we opt to retain an 8-bit precision approach, as it yields the most optimal results without imposing a significant overhead on resources.

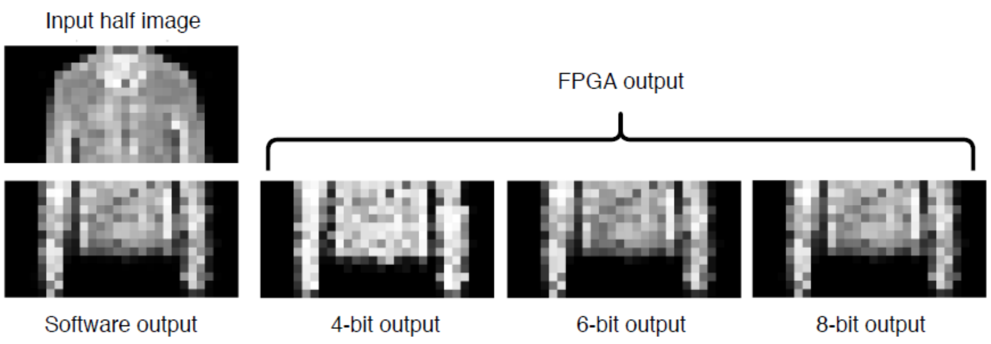


Figure 3-13. Image reconstruction quality for different bit precision in the FPGA

Furthermore, Figure 3-14 presents the distribution of pixel errors obtained from the complete set of test images (normalized on a scale of -1 to 1) across varying bitwidths on the FPGA hardware. The error values are compared against the software-based execution of the generator using 32-bit floating-point representation. This depiction also clarifies that the 8-bit configuration consistently yields the most favorable outcomes, achieving a Peak Signal-to-Noise Ratio (PSNR) of 43.14 dB (scaled from 0 to 255 pixels). Notably, within the context of 8-bit normalized images, the maximum error remains under 0.1. In contrast, configurations with fewer bits witness the maximum error rising to approximately 2. This escalation in error magnitude can potentially lead to complete pixel inversion, shifting from white to black or vice versa. This outcome, in turn, contributes to a discernible decline in output quality.

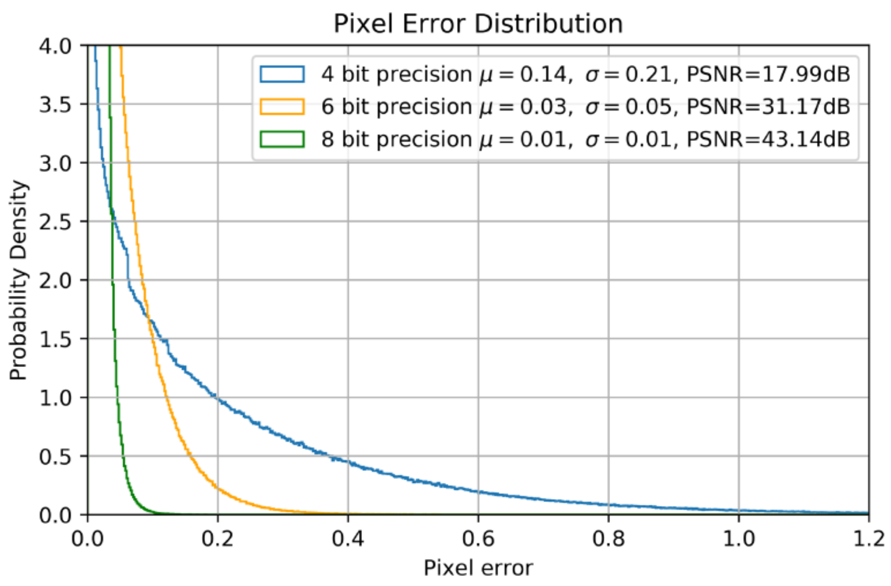


Figure 3-14. Pixel error distribution for different FPGA bitwidths

II. Regarding the system configuration, we employed a Xilinx ZC702, which is equipped with a Zynq-7000 System on Chip (SoC) featuring a Dual-core ARM Cortex-A9 processor and 512 MB of DDR3 memory. The resource allocation of our FPGA-based hardware accelerator, coupled with latency timings and the attained frames per second (FPS), is detailed in Table 3-2.

	Utilization Summary				Timing	
Name	**BRAM**	**DSP**	**FF**	**LUT**	**Latency**	**FPS**
Used	54	110	18907	9855	-	-
Percentage	38.57%	50%	11.77%	18.52%	0.013 (ms)	77K

Table 3-2. Resource utilization and latency per kernel

III. To comprehensively assess our system's performance, including memory transfers, we implemented the identical generator model on alternative systems (CPU, GPU). This was carried out to facilitate a fair performance and performance per watt (PPW) comparison. The acceleration achieved is evaluated against the baseline of the single-core testing on the ARM Cortex-A9 CPU within the FPGA SoC. The relatively modest scale of the problem rendered an embedded FPGA SoC advantageous compared to a GPU, considering the device overheads. Remarkably, this approach yielded favorable outcomes across all platforms, both in terms of performance and the performance-to-power ratio (PPW) metric as seen from the metrics in Table 3-3.

Device Information		**Evaluation**			
System	**Model**	**Time/Img**	**Speed-up**	**Power**	**PPW**
CPU	ARM A9	2.06ms	1×	3.2W	1×
GPU	Nvidia K80	0.033ms	62×	74W	2.7×
FPGA	ZC702	0.013ms	158×	3.6W	140×

Table 3-3. Performance and power evaluation vs other systems.

Conclusion — In this study, we examined the utilization of a highly promising Deep Learning technique known as GAN (Generative Adversarial Network) for the purpose of image reconstruction on an FPGA. This particular application domain had yet remained unexplored in the context of FPGA implementation. Our investigation unveiled the supremacy of GANs over conventional algorithms in terms of image restoration quality. Concurrently, we demonstrated a successful proof-of-concept, showcasing the efficacy of employing FPGAs for such tasks, yielding substantial gains in accuracy, speed, and power efficiency. The generator model, meticulously trained with optimizations tailored for hardware, displayed exceptional performance in reconstructing high-quality images. It not only minimized latency but also operated with remarkable power efficiency, outperforming equivalent CPU and GPU platforms. While the potential trade-offs within the design space are expansive, our study has illuminated a path within this field, yielding prosperous outcomes. The overall objective is to establish FPGAs as major contributors to the open software-hardware landscape, and this research takes a significant stride towards realizing that vision.

Acknowledgements: This project was funded from the Xilinx University program and the Hellenic Foundation for Research and Innovation (HFRI) and the General Secretariat for Research and Technology (GSRT), under grant agreement No 2212 - Hardware Acceleration of Machine Learning Applications in the Cloud.

3.5 Hardware Accelerated AI for Covid detection

Within this section, we explore the third scenario out of the four in which we employed FPGA acceleration to enhance the execution of AI applications. The scenario we investigated here is a medical application aimed at combating the Covid-19 pandemic. The Covid-19 pandemic had devastated both social life and the global economy, causing a relentless surge in daily cases and fatalities. While chest X-Rays serve as a widely accessible and cost-effective screening method, the sheer volume of respiratory illness cases impedes rapid testing and timely quarantine for every patient. Consequently, there is a pressing need for an automated solution, driven by the research community's dedication. In response to this demand, we present a Deep Neural Network (DNN) topology designed to categorize chest X-Ray images into three classes: Covid-19, Viral Pneumonia, and Normal. The accurate identification of Covid-19 infections through X-Rays holds utmost significance, supporting medical professionals in their diagnostic tasks. Nonetheless, the substantial amount of data to be processed consumes valuable time and computational resources. Taking a significant stride forward, we implement and deploy this Neural Network on a Xilinx Cloud FPGA platform. These devices are known for their remarkable speed and power efficiency. The ultimate goal is to provide a cloud-based medical solution for hospitals, streamlining medical diagnoses with precision, speed, and low energy efficiency. To the best of our knowledge, this application has not been explored for FPGAs previously. Notably, the achieved accuracy and speed surpass any known implementations of Neural Networks for X-Ray Covid detection. Specifically, our system classifies X-Ray images at an impressive rate of 3600 frames per second (FPS) with an accuracy of 96.2%. Furthermore, it outperforms GPUs with a speed-up of 3.1× and surpasses CPUs with a remarkable 17.6× in terms of performance. In terms of power efficiency, the FPGA platform excels, demonstrating a 4.6× improvement over GPUs and an impressive 13.1× over CPUs.

3.5.1 Introduction and Related work

The abrupt surge in COVID-19 cases, stemming from a novel respiratory virus, has imposed an unprecedented burden on healthcare systems globally. The pandemic's profound impact on both the health and economy of the worldwide population is undeniable. A crucial aspect of the battle against COVID-19 involves the efficient screening of infected individuals. This is vital to ensure that those diagnosed can promptly receive treatment and be quarantined, particularly those at elevated risk. The conventional method for detecting the disease involves a manual examination of chest X-Ray radiographs (CXRs) by highly trained specialists. While widely adopted, this approach is inherently time-consuming and intricate, adding strain to healthcare systems.

Moreover, X-Ray images of pneumonia associated with COVID-19 are often indistinct, leading to potential misclassification and subsequent errors in medication or delayed quarantine [106, 107, 108]. Recognizing the urgency to address these challenges, there is a pressing need for an automated method to categorize chest X-rays and identify specific diseases. In response to the collaborative efforts of the research community and the imperative to combat the COVID-19 pandemic, we propose the development of a Deep Learning application for the automatic detection of COVID-19 through chest X-Rays. The ultimate goal is to deploy this tool in the Cloud, providing doctors and clinics worldwide with remote access to a Cloud Medical AI Assistance.

While Cloud Computing facilitates on-demand access to computing resources, it is acknowledged that CNN models, due to their computational intensity, pose significant demands on compute power. This is particularly related in the current pandemic era, where vast amounts of patient data need processing daily. Addressing this modern challenge, our project introduces a novel solution leveraging hardware-specific optimizations through Field-Programmable Gate Array (FPGA) architecture. FPGA-based acceleration has exhibited substantial promise by offloading specific tasks from the CPU, enhancing system performance, and reducing dynamic power consumption. This section marks a significant advancement in Deep Learning, especially in CNN tasks, benefiting from FPGA's parallelism and reconfigurability at the bit level. Deploying our CNN model on FPGA platforms accelerates the Image Recognition process, ensuring both speed and power efficiency, crucial in datacenter workloads.

In this project, we will describe several highly accurate Deep Learning models using custom and novel Convolutional Neural Network topologies that can detect Covid-19 disease in chest X-Ray images. The application is also accelerated through a Xilinx Cloud FPGA platform using the latest Xilinx's development stack for AI inference. Furthermore, we investigate how our model makes predictions in an attempt to gain deeper insights into critical factors associated with Covid cases but also make the proposed testing technology scalable on the Cloud to be available globally with support for massive input data.
In summary, the main contributions of the paper are as follows:

- We expand upon three efficient CNN model architectures with a focus on memory and size efficiency, aimed at classifying chest X-Rays into three categories (Covid, Viral Pneumonia, Normal). Trained on the Tensorflow Deep Learning framework, we attain a maximum accuracy of approximately 97%, surpassing the performance of previous CNNs designed for chest X-Ray Covid detection.
- We introduce FPGA-specific optimizations to the model topologies, employing 8-bit quantization for arithmetic precision. The most efficient model is selected and accelerated on a Xilinx Alveo U50 FPGA using a heterogeneous architecture that is both scalable and seamlessly portable to data centers for cloud workloads.

- The FPGA application is executed in a containerized environment, and results are quantified in terms of accuracy, speed, and power efficiency. Our performance surpasses other high-performance devices (Xeon CPU, V100 GPU), excelling in both performance and performance per watt. Additionally, we evaluate the model's classification ability using heatmaps on X-Rays and present other important classification metrics.

It is imperative to mention once more that the models presented are not intended for self-diagnosis. Individuals should seek assistance from local health authorities if needed. The tool is designed as an assistant resource for healthcare systems, offering highly accurate identification of the type of disease, whether it is Covid-19, Viral Pneumonia, or a Normal chest image. The rapid and accurate detection of COVID-19 infections in chest X-Rays is of paramount importance as it facilitates the rapid diagnosis and quarantine of high-risk patients.

Related Work — X-Ray detection algorithms have been explored by many researchers especially in the past, usually for lung diseases such as Viral Pneumonia. In the recent year of 2020, the global Covid-19 pandemic has sparked an increasing interest within the research community regarding the development of AI models tailored for the identification of Covid-19. The pressing demand to support healthcare professionals in their medical diagnoses through automated tools, particularly leveraging Deep Learning, is evident. However, given the substantial computational complexity of these AI models and the exponential growth of laboratory data from new patients, there is an immediate need for a robust platform capable of handling these requests with both high speed and efficiency. The subsequent literature review encompasses comparable design approaches addressing analogous issues or presenting related problems within our specific problem domain.

Several studies have investigated the use of Neural Networks towards Covid detetion through chest X-Rays. Mangal et al. [109] presented CovidAID, a deep neural network based model to triage patients for appropriate testing. On the publicly available covid-chestxray dataset their model gave 90.5% accuracy for the COVID-19 infection. On the same scheme, Wang et al. introduced CovidNet [110], a deep convolutional neural network design tailored for the detection of COVID-19 cases from CXR images. They showed a classification report with 93.3% achieved accuracy. Although, these projects were some of the early work towards the fight against Covid and were very useful for the community, they lack the performance of our CNN model which achieves an accuracy of 96.2%.

Furthermore, several deep learning approaches have been devised for the identification of Covid-19, as documented in [111, 112, 113]. However, these models often fall short in terms of accuracy or precision when compared to our proposed model. For instance, Jain et al. introduced an Xception model topology for the same problem domain, achieving slightly higher accuracy (97.9%) than our model. Nonetheless, their approach lacks a method for accelerating or enhancing the efficiency of the inference procedure. Additionally, numerous studies have focused on a binary classification scenario, distinguishing between Covid and non-Covid images [114, 115]. In contrast, our model incorporates a third classification category, Viral Pneumonia, which holds significance in treatment strategies, as patients with Viral Pneumonia require distinct treatment plans.

It is noteworthy that, to the best of our knowledge, there is no existing research on Covid detection utilizing Field-Programmable Gate Arrays (FPGAs). Consequently, we will compare our work with related projects in the same problem domain, particularly those involving hardware acceleration of Convolutional Neural Networks (CNNs) for Pneumonia detection. For example, Chouhan et al. [116] developed a CNN model using transfer learning from ImageNet models, reporting an average inference computation time of 0.043s on an Nvidia GTX 1070 GPU card. Similarly, Azemin et al. [117] implemented a ResNet-101 CNN model architecture for Covid-19 detection, achieving a speed of 453 images/min on CPU. In conclusion, while various CNN-based approaches for Covid-19 detection exist, our project introduces a novel CNN with superior accuracy compared to prior work. Moreover, we propose an acceleration method suitable for deployment on cloud FPGAs, a domain that has not been explored extensively in previous research.

3.5.2 Implementation and Results

We will divide this paragraph into two parts; the software and hardware implementation of our proposed solution. These are two separate flows that are needed before deploying the model to the actual hardware. The software flow is related to the training and finetuning of the CNN model using a deep learning framework while the hardware flow is the optimization and deployment procedure regarding the FPGA execution.

Software Implementation

In this part, we formulated multiple Convolutional Neural Network (CNN) architectures for the classification of Chest X-Ray (CXR) images within the dataset. The selection of the optimal AI model was based on considerations of both accuracy and efficiency, with the intention of deploying it on an FPGA, as elucidated in the subsequent section. This section is dedicated to delving into the problem formulation, the dataset utilized, the training process, and the hardware-centric optimizations implemented on the models to facilitate their efficient deployment on the FPGA device.

1. **Dataset:** The Covid-19 X-Ray image database utilized in this study was curated from the Italian Society of Medical and Interventional Radiology (SIRM) COVID-19 DATABASE [118] . The dataset comprises a total of 2,905 CXR images, categorized into 219 for Covid, 1,345 for Viral Pneumonia, and 1,341 for the Normal class, used for training and evaluating the AI models. Despite the relatively modest size and irregularity of the dataset, we employed various techniques to address these challenges, as elaborated in the following sections. The selection of this dataset was driven by its open-source nature and full accessibility to the research community and the general public. As datasets expand, we remain committed to refining and adapting the models accordingly. The bar chart below illustrates the distribution of CXR images for each infection type, segmented into training and test sets, with the test dataset accounting for 25% of the total dataset.

2. **Model Topology:** We propose three different topologies for this problem in order to have a better evaluation on the dataset and select the most suitable model for acceleration on the FPGA afterwards. We developed separate models that each has different prediction accuracy, architectural complexity (in terms of number of parameters) and computational complexity (in terms of number of MAC operations). A CustomCNN which is a classic convolution neural network, a lightResNet which is a ResNet50 variant and DenseNetX which is based on DenseNet architecture but it also includes the Bottleneck layers and Compression factor.

3. **Training:** Last we will analyze several techniques that we applied on the training procedure. These optimizations mainly had to do with the specific dataset characteristics but also include several hardware-aware optimizations on the model that helped us deploy and accelerate the CNNs in the FPGA more efficiently. The first is related to *Class weighting.* Training with a dataset like ours with very few Covid-19 images as opposed to Normal or Viral Pneumonia images constitutes a class-imbalanced problem. This is a complexity that poses significant challenge to the converging of our models as CNNs are normally assume to be trained on identical distribution datasets. To overcome the class variance we imposed specific class weights (i.e., 6× on Covid class) which applied to the model's loss for each sample and eventually helped the model learn from the imbalanced data. Next we apply *HW-aware optimizations* to the model compilation. The CNNs' topology needed some minor modifications in order to be compatible and efficient with Vitis AI quantizer and compiler. In particular, the order of the Batch Normalization (BN), Rectified Linear Unit (ReLU) activation and Convolution layers has been altered from BN → ReLU → Conv to Conv → BN → ReLU. This order of layers is ambiguous from the research

community, however it ensures that for any parameter values the network always produces activations with the desired distribution. Also, another optimization that we did is in the case of GlobalAveragePooling2D, which we needed for example in DenseNetX and we replaced it with AveragePooling2D plus a Flatten layer. Last, softmax was implemented in the DPU and not in SW (proved to be more than 100x times faster) using an AXI master interface named `sfm_interrupt`. The softmax module used `m_axi_dpu_aclk` as the AXI clock for `SFM_M_AXI` as well as for computation.

Hardware Implementation

In this segment, we will describe the sequence of steps encompassing quantization, evaluation, compilation, and ultimately, the execution of AI models on the Alveo FPGA platform. Additionally, we will examine the comprehensive architecture of the FPGA design, functioning within a heterogeneous system that facilitates efficient communication with the host processor. Finally, we will establish a complete end-to-end environment accessible for seamless testing and utilization of our project through an FPGA-containerized application.

Acceleration Approach — Given the potential utilization of our application by many users globally, an efficient and expeditious solution is imperative. Consequently, we opted to leverage the Vitis AI environment to deploy our Convolutional Neural Network (CNN) models on an Xilinx Alveo U50 FPGA. This strategic decision aims to yield an application with high inference throughput and a compact memory footprint—critical factors for optimal performance in cloud workloads. Further exploration of the steps involved in model quantization, evaluation, and the subsequent compilation will be discussed. This compilation process generates DPU (Deep Learning Processing Unit) instructions, facilitating the effective utilization of the FPGA's compute units (CUs).

1. **Quantization:** Initially, we converted our models into a Tensorflow-compatible floating-point frozen graph as a prerequisite for the quantization process. Subsequently, we opted for the quantization of the trained weights of our Convolutional Neural Networks (CNNs) using 8-bit precision. This choice, widely acknowledged in similar CNN applications, has demonstrated the ability to maintain acceptable accuracy levels. Finally, we supplied a representative sample set of the training data to calibrate the quantization process. The data underwent a complete forward pass through the model, and the weights were adjusted based on the data range required for inference by the application.

2. **Evaluation of Quantized Model:** The transformation from a floating-point model, where values can exhibit a broad dynamic range, to an 8-bit model, limiting values to one of 256 possibilities, inherently introduces a slight loss of accuracy. Therefore, it was crucial to evaluate the quantized graph on Tensorflow before proceeding with the model compilation. Notably, this technique generally yielded nearly identical accuracy results when compared with the actual application tested on the board. Moreover, the quantized graph on the FPGA, in contrast to the floating-point graph on the CPU, had a minimal impact on the final accuracy (less than 0.5%).

3. **Model Compilation:** In the final phase, we compiled the graph into a set of micro-instructions encapsulated in a `.xmodel` file format. The Vitis AI compiler undertook the conversion and optimization of the quantized deployment model, resulting in the generation of the final "executable" for CNN inference. The generated instructions were tailored to the specific configuration of our Deep Learning Processing Unit (DPU). In our case, the `DPUCAHX8H` DPU IP was selected. To specify the parameters of the DPU for the target Alveo U50 board, we provided these parameters in a `.dcf` file.

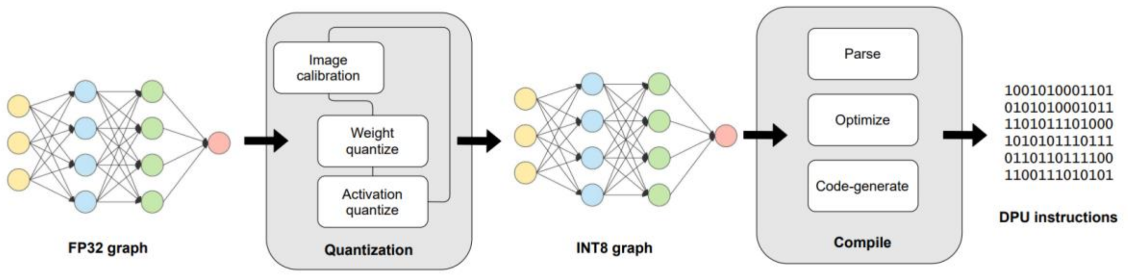


Figure 3-15. CNN graph quantization and compilation for the FPGA DPU

Evaluation and Results

In this segment, we will assess and profile our application comprehensively. Initially, we will analyze the model performance, considering factors such as validation loss, accuracy, and various classification metrics for the neural networks. Subsequently, we will delve into the evaluation of the hardware accelerator's performance, exploring aspects such as resource utilization, acceleration, and power efficiency in comparison to the CPU and GPU. Finally, we will showcase qualitative results, shedding light on areas within the X-Ray images that are particularly indicative of Covid or Viral Pneumonia.

Model evaluation — In this study, experiments were conducted using Tensorflow and Keras, employing the common 224 × 224 image dimensions typical in many Convolutional Neural Networks (CNNs). All models underwent training with the Adam optimizer, accompanied by EarlyStopping and best model callbacks. The optimization of classification models was achieved through the minimization of the cross-entropy loss function. Additionally, various parameters and hyperparameters for each model underwent tuning during training, including Learning Rate (LR) and epochs. Table 3-4 provides an overview of the key characteristics of each model, encompassing training hyperparameters, model specifications, and model evaluation metrics.

	Hypermaramaters		Model Specs		Evaluation	
Model	**LR**	**Epochs**	**Params**	**FLOPs**	**Accuracy**	**Loss**
CustomCNN	0.0001	70	2.033G	1.025G	96.2%	0.16
lightResNet	0.001	60	2.697G	2.814G	96.5%	0.408
DenseNetX	0.005	80	0.758G	1.722G	94.9%	0.264

Table 3-4. CNN model characteristics and performance

Qualitative analysis — In the preceding tables, we presented various performance criteria to assess the effectiveness of our classification models. Depending on specific requirements, one can opt for a model with distinct characteristics that align with the desired balance between performance efficiency (FLOPs) and accuracy. For the purpose of demonstration and the FPGA implementation, we chose the CustomCNN model. This selection is based on its efficient performance, achieving a high level of accuracy comparable to that of lightResNet while maintaining minimal computational requirements — a crucial factor for compute-intensive workloads. Furthermore, we provide several insightful activation maps derived from the last convolutional layer of the CustomCNN network. These maps play a significant role by offering an understanding of the model's classification capability and validating the regions of attention regarding the disease.

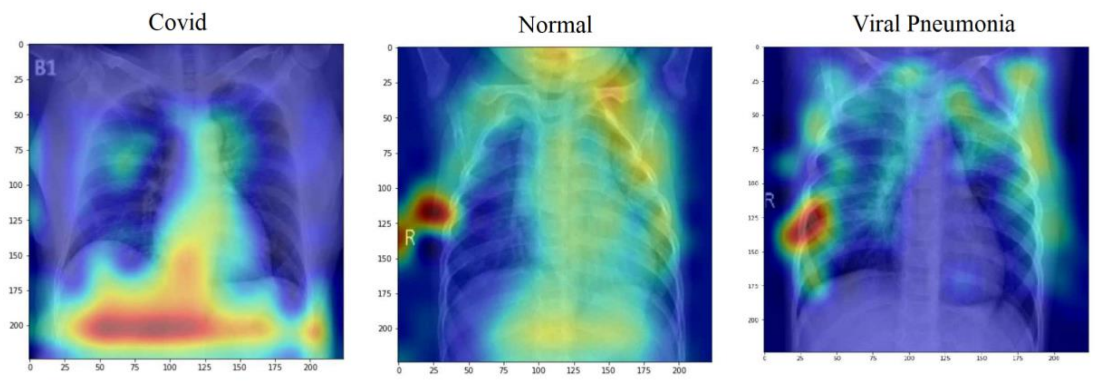


Figure 3-16. X-Ray visualizations using attention heatmaps

System performance — In assessing the system design, we initially verified the resource utilization of the FPGA's Deep Learning Processing Unit (DPU). The hardware configuration for deployment comprised a Xilinx Alveo U50 Cloud FPGA featuring an 8GB High Bandwidth Memory (HBM) capacity and a total bandwidth of 316 GB/s. The device was integrated into a Gen4x8 PCI Express setup, operating at a kernel clock frequency of 300MHz. Table 3-5 provides an overview of the resource utilization for a DPUv3E kernel equipped with five batch engines (our design employed two kernels).

	Utilization Summary				
Name	**BRAM**	**URAM**	**DSP**	**FF**	**LUT**
Used	628	320	2600	310752	250290
Percentage	46.7%	50%	43.6%	21.2%	28.7%

Table 3-5. Resource utilization of a single DPU kernel

Subsequently, we conducted an assessment of inference using the CustomCNN model on alternative high-performance systems, specifically an Nvidia V100 GPU and a 10-core Intel Xeon Silver 4210. The inference on these alternate devices was carried out using Tensorflow with default settings and suitable batch sizes. The left side of Figure 3-17 illustrates the maximum throughput achieved by each device, measured in X-Rays per second (FPGA: 3600, GPU: 1157, CPU: 204). Additionally, we annotated the latency, measured in milliseconds, for single X-Ray image inference on each device. Furthermore, the right side of Figure 3-17 depicts the power efficiency measurement for each device in X-Rays/Sec/Watt (FPGA: 51.3, GPU: 11.1, CPU: 3.9).

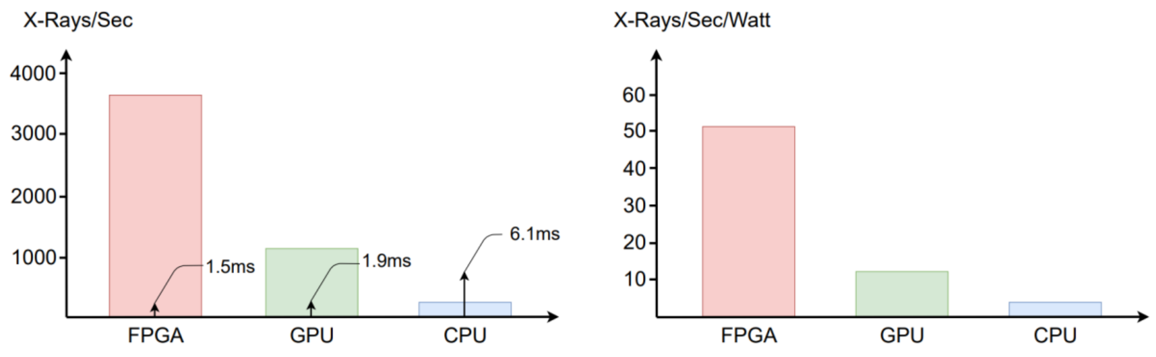


Figure 3-17. Performance and Performance/Watt metrics across different architectures

The throughput metric holds significant importance for cloud workloads dealing with extensive patient data, while the latency metric becomes crucial in edge scenarios, such as mobile phones, where time sensitivity demands an immediate response.

A quantitative comparison reveals that, in large batch size scenarios in the cloud, the FPGA achieves the highest inference speed, demonstrating a 3.1× speed-up from the GPU and a remarkable 17.6× speed-up from the CPU in terms of throughput. It is noteworthy that the 8-bit quantized CNN employed on the FPGA incurred less than a 0.5% accuracy loss, a trade-off deemed acceptable for the gains in performance and power efficiency. Furthermore, the FPGA outperforms the other two devices in the power efficiency metric, measured in X-Rays/Sec/Watt. Specifically, it attains a 4.6× speed-up from the GPU and an impressive 13.1× speed-up from the CPU. The FPGA's metric of 51.3 X-Ray/Sec/Watt implies that to identify a disease in a single chest X-Ray image, only 0.019 seconds and 1 Watt of compute power would be required. This efficiency is particularly crucial for cloud providers aiming to minimize energy consumption in data centers while meeting the performance demands of various applications.

Conclusion — In this study, we introduced multiple AI models, each possessing distinct characteristics, designed for the detection of COVID-19 cases from CXR images. These models are open source and available to the general public. Notably, our study showcased substantial enhancements in both accuracy and performance when compared to previous related work. Additionally, we delved into the interpretability of our model's predictions by employing an attention heatmap method, seeking deeper insights into critical factors associated with COVID-19 cases. This not only aids clinicians in refining screening processes but also enhances trust and transparency when utilizing our Convolutional Neural Network (CNN). Furthermore, we quantized, compiled, and accelerated the AI model for deployment on an Alveo U50 FPGA, aiming to accelerate computer-aided screening. The application was containerized, allowing seamless portability to a cluster of FPGAs operating in the cloud, exhibiting high performance and energy efficiency in comparison to other architectures. It's essential to note that this is not a production-ready solution intended for self-diagnosis. From a research standpoint, our focus remains on enhancing performance and introducing additional features within our AI Health framework, particularly as new data is collected. This may involve areas such as risk stratification for survival analysis or predicting hospitalization durations. While the realm of AI automated systems is vast, this work sheds light on the potential contributions of FPGAs in fundamentally shaping computer-aided Medical Diagnosis.

Acknowledgements: This project was funded from the Xilinx University program and the Hellenic Foundation for Research and Innovation (HFRI) and the General Secretariat for Research and Technology (GSRT), under grant agreement No 2212- Hardware Acceleration of Machine Learning Applications in the Cloud.

3.6 Creating Optimized Firmware from CNNs for FPGAs

This section contains the forth of the four scenarios in which we utilized FPGAs for AI acceleration. In particular, this section does not present a specific AI application but entails a more generalized way for accelerating AI applications for FPGAs. Specifically, it describes on efficient framework to convert trained CNN models into optimized FPGA firmware. Effective AI algorithms hold immense significance across numerous challenges, particularly in tasks involving classification or clustering. However, a standardized universal AI model and seamless optimization is necessary. Consolidating various machine learning models into a shared ecosystem can substantially reduce development time and enhance compatibility within frameworks. The Open Neural Network Exchange Format (ONNX) stands as a widely recognized open format for representing deep learning models. Its purpose is to enable smoother transitions of models among cutting-edge tools for AI developers. Notably, hardware companies like Nvidia and Intel are striving to stay aligned with this trend. They are producing hardware runtimes optimized for CPUs and GPUs that proficiently manage these open format AI models like ONNX. This empowers developers to harness an assorted range of hardware and utilize their preferred AI frameworks. Yet, FPGAs pose a more intricate challenge. However, they are a proven platform for effectively addressing such challenges concerning performance and power efficiency. Our study is based on an early-stage development project known as HLS4ML [3], initially designed for particle physics applications. The project's core innovation involves the automatic generation of neural networks (NNs) for embedded Xilinx FPGAs. Our work takes this a step further by incorporating hardware-aware NN training and a comprehensive optimization strategy on top of HLS4ML. This extension significantly enhances the library’s performance and power efficiency. Additionally, it introduces functionality for cloud FPGA firmware deployment from any NN model. Our methodology begins with FPGA-aware training of a model in Keras, tailored for image recognition. The model is then converted into the ONNX open format before being adapted and fine-tuned for cloud FPGAs. This process employs a novel scheme that optimizes various aspects, including the host environment, memory management, and kernel operations. Multiple levels of network precision are also leveraged. To the best of our knowledge, this approach stands as an unique innovation. It leads to a remarkable speed-up, achieving performance gains of up to 102× compared to a single CPU, and up to 5.5× improvements in performance per watt compared to GPUs.

Subsequent paragraphs will provide an initial overview and contextual insights into frameworks designed for the automated acceleration of AI hardware. Our implementation and integration within the HLS4ML framework will be detailed, followed by evaluation metrics concerning the acceleration of two popular AI models.

3.6.1 Introduction and Related work

Recently, ML techniques have achieved remarkable success across many applications and have emerged as pivotal tools, particularly in fields like image and speech recognition. Convolutional Neural Networks (CNNs) have gained substantial traction due to their exceptional accuracy and performance in visual recognition tasks [119]. These deep learning models have proven to be revolutionary, permeating numerous industries and finding their way into an increasing array of commercial products, thereby significantly impacting people's daily lives.
Nonetheless, the demands of contemporary companies necessitate the processing of massive volumes of data, often in the order of terabytes or more each day, owing to the prevalence of these algorithms. As machine learning applications continually learn from extensive real-world datasets [120], the requirement for faster processing speeds becomes ever more pressing. This computational complexity has spurred initiatives to enhance these tasks through hardware-specific optimizations, making use of diverse heterogeneous architectures that combine platforms such as CPUs, GPUs, and FPGAs [121]. While the utilization of multicore systems holds promise [122], the challenge of mitigating the considerable energy costs and processing times persists [123]. Notably, FPGA implementations have progressed significantly, showcasing their exceptional effectiveness in CNN-related tasks due to their remarkable parallelism and reconfigurability at the bit level. Leveraging hardware accelerators contributes to an increased overall throughput, stemming from the highly parallelizable nature of the numerous multiply-accumulate operations (MACs) inherent to these algorithms.

Within the field of CNN hardware acceleration, a notable challenge is the interoperability of deep learning frameworks. The task of moving models seamlessly between cutting-edge tools while selecting the optimal configuration remains a challenge for AI developers. As a response to this, there has been a shift towards adopting open format AI models such as ONNX, which fosters an ecosystem with standardized representations [98]. ONNX offers an open-source format for AI models, including both deep learning and traditional machine learning paradigms. Offering an adaptable computation graph model and definitions of pre-built operators and standard data types, ONNX gathers significant support from prominent corporations like Alibaba, ARM, AWS, IBM, Huawei, Intel, Nvidia, and others [124]. However, adapting AI models to run efficiently on platforms like FPGAs can be exceptionally demanding, prompting developers to explore ways to circumvent code rewriting and overcome code optimization challenges. The process of hardware acceleration for varying neural network architectures on FPGAs is far from straightforward.
In this paper, we present an innovative approach for seamlessly translating Convolutional Neural Networks (CNNs) into OpenCL FPGA devices within cloud environments. We extend the capabilities of an early-stage open-source project known as HLS4ML [125, 3],

which involves a compiler for high-level language code tailored for embedded Xilinx FPGAs, thereby enabling neural network model implementation. Our contribution introduces novel architectures and optimization methodologies for automating the translation of neural networks onto cloud-based FPGAs, enabling full execution within FPGA hardware and taking advantage of the server class FPGAs. As a result, the development process for ONNX-based deep learning applications on FPGAs gains speed and power efficiency, simplifying neural network compilation and acceleration overall in the cloud as well. To sum up, the primary contributions of this paper include the following:

- Introduction of a novel optimization scheme for automatically generating cloud FPGA firmware from ONNX models, around a generalized approach adaptable to various neural network types with the capacity for design space exploration.

- Introduction of a new template for additional optimizations at the kernel, memory, and host levels for FPGAs, such as the Xilinx Alveo U200. These optimizations are applied on top of HLS4ML library utilizing a flexible heterogeneous streaming architecture, involving different precision configurations between neural network layers.

- Proposition of a hardware-specific training method for neural networks, demonstrated using a small MNIST-based model and a larger CIFAR-based model. We port ONNX-converted models for automatic High-Level Synthesis (HLS) translation for the Alveo board. In performance and performance per watt, we achieve superior results compared to other high-performance devices like a Xeon CPU and P100 GPU.

Related Work — Numerous researchers have experimented with CNN algorithms, particularly focusing on optimizing both software and hardware aspects. These optimizations are applicable across various domains, including image recognition and object detection. In the following overview of related work, similar design approaches are presented, addressing analogous problems within our problem domain. Weijie You et al. [126] introduced a design for a DNN pipeline accelerator based on grouping techniques, tailored for FPGAs. Their evaluation uses AlexNet and VGG16 networks using Xilinx ZC706, an embedded FPGA SoC. In contrast, our paper utilizes a cloud FPGA instead of the embedded counterpart. In a similar vein, Hao [127] proposed an FPGA/DNN co-design method facilitated by an Auto-HLS engine to generate FPGA-synthesizable C code. Their work also centers on an embedded FPGA, the Xilinx PYNQ-Z1. Furthermore, Sitao Huang [128] demonstrated a versatile sparse DNN inference accelerator on FPGA, adaptable for both mobile and high-performance computing scenarios. Notably, they omitted a GPU comparison from their study. Ghasemzadeh et al. [96] showcased

ReBNet, a Residual Binarized Neural Network running on Xilinx FPGAs. This implementation employs 1-bit precision per neural network layer, utilizing Xnor Popcount-style computations. However, their approach's performance falls short when compared to our solution. Despite running at a slightly different kernel frequency, their hardware accelerator's CNN model is also based on the MNIST dataset. Their reported peak throughput is 64000 Images/s, whereas our MNIST model achieves a maximum of 158000 Images/s. This performance advantage is achieved by using 8-bit and higher precision weights while utilizing only half of the available resources. Jiong Si et al. [129] conducted testing on an FPGA platform for inference using various precisions, including 8-bit, on the MNIST dataset. Despite their FPGA operating at a lower frequency, their performance with 8-bit data is proportionally lower than ours. They indicate the use of a 25 MHz clock, while our FPGA reaches the maximum device frequency of 300 MHz. Notably, our clock speed is 12 times faster, leading to a significant performance advantage of 60 times compared to their proposed solution. This implies an overall 5-fold performance enhancement from our system. Alemdar et al. [99] implemented fully-connected ternary-weight neural networks on FPGAs, reporting a latency of 20.5 μs for the MNIST dataset. Their ternary networks inherently achieve sparsity through pruning smaller weights to zero during training, enhancing energy efficiency. It's noteworthy that their work utilized a custom Xilinx Kintex 7 FPGA board named Sakura-X, operating at 200 MHz. In contrast, our FPGA board, utilizing similar resources, operates at 300 MHz and achieves a latency of 6.3 μs while employing higher precision weights and activations. Lastly, Makrani et al. [130] introduced a model to optimize the performance/cost ratio of scale-out applications in cloud environments across varying memory configurations. Their methodology, called Mena, focuses on tuning memory and processor parameters to match system configurations with application requirements and budget constraints, although it does not extend to the realm of FPGAs.

In conclusion, extensive research has explored the use of lower precision neural networks on FPGAs. However, in the subsequent paragraphs, we will present a novel approach for deploying CNNs on FPGAs automatically. This approach boasts the advantage of seamlessly and efficiently generating FPGA code from ONNX models, offering low latency and high power efficiency.

Preliminaries — In this work, we focused on utilizing an open neural network format that can be beneficial for our framework as it promotes interoperability. The Open Neural Network Exchange (ONNX) serves as an open ecosystm, empowering AI developers to streamline the AI model development process. ONNX offers an open-source format for AI models, encompassing both deep learning and traditional machine learning. It employs an adaptable computation graph model and defines a range of built-in operators and standard data types for seamless deployment in inference tasks. ONNX has broad support across numerous frameworks such as Tensorflow, Caffe, Pytorch, MxNet, Matlab, and is compatible with various hardware runtimes, including Nvidia's and Qualcomm's. Javier Duarte et al [3] explored the utilization of FPGAs for rapid inference within particle physics applications. They identified a distinctive requirement in particle physics for FPGA-based trigger and data acquisition systems, demanding exceptionally low, sub-microsecond latency. Consequently, they introduced a package designed to automate the creation of machine learning models for FPGAs through High-Level Synthesis (HLS). This compiler autonomously translates pre-trained neural networks into HLS code, relying on the model's architecture, weights, and biases. It extends support to models trained in Keras, PyTorch, and even ONNX. However, it's important to note that this tool is still in its early stages and offers limited support for certain layers. In our specific case, the model tailored for FPGA hardware initially encountered compilation challenges when used with hls4ml. It necessitated several modifications to address memory issues, but this marked the inception of our solution approach.

3.6.2 Integration with Existing Tools and Evaluation

Exporting machine learning models from frameworks like TensorFlow or PyTorch necessitates robust hardware with substantial computational capabilities, highlighting the importance of a well-defined design flow. To gain a clearer insight into our design strategy, Figure 3-18 provides an overview of the proposed design flow. As depicted, the process begins with the training and optimization of an AI model using Keras, incorporating hardware-aware optimizations. Subsequently, the model is converted into the ONNX format. Next, the open-format neural network model undergoes conversion into FPGA HLS code, facilitated by the HLS4ML package, complemented by our neural network and FPGA optimizations tailored for cloud deployment. Finally, the image recognition application is executed on board with meticulous synthesis for high-frequency performance and adequate hardware exploration. Performance, modularity, and re-configurability serve as pivotal sides of a high-performance and adaptable design.

- *Performance:* To enhance design efficiency and reduce latency, we employ a streaming dataflow approach. Each layer operates in parallel as an independent module, transmitting its output directly to the subsequent layer, which takes the previous output layout as input, thereby overlapping operations and obviating the need to store intermediate results in off-chip memory. This results in a faster and more power-efficient design. Additionally, we consider calculation precision, optimizing multiplier usage while minimizing accuracy errors.

- *Modularity:* Our design accommodates varying precision levels among layers, creating a heterogeneous neural network with accuracy tailored to each layer's activation requirements. Precision settings for weights, biases, and activations can be seamlessly adjusted to align with application needs. Furthermore, each layer functions as a standalone accelerated module, offering independent control.

- *Re-configurability:* Our framework is configured to generate custom FPGA firmware compatible with a wide array of OpenCL FPGA devices, including multiple copies of the same kernel for batch processing.

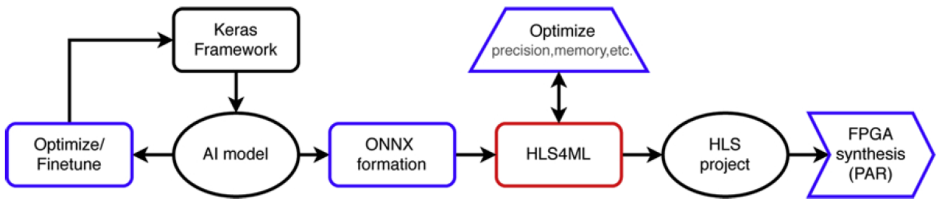


Figure 3-18. Design steps for ONNX model deployment to FPGAs. (Blue tasks indicate our work while red are taken from previous work. The HLS4ML python API was modified to support the new optimizations and changes.

Architecture Design

We adhere to FPGA design principles towards optimizing high performance and power efficiency, accompanied by the development of a custom OpenCL host API designed to accommodate cloud FPGAs for data centers. Our semi-automated OpenCL-centric tool flow methodology for deploying neural networks on FPGAs supports various software and hardware optimizations. We commence with hardware-specific training of a neural network, applicable to a wide range of neural network models. This approach offers substantial flexibility in parallelization, particularly beneficial when working with unsigned neural network weights. Furthermore, the translation of the generalized ONNX model format into a fused hardware model with independent modules for each layer provides opportunities for further parameterization and optimization of the neural network. Lastly, through the OpenCL host API, we enhance memory access to kernels, maximizing device data bandwidth. Utilizing OpenCL command queues and precise host-kernel synchronization enables substantial parallelization at a coarse-grained level.

I. *Hardware Accelerator*. Our hardware accelerator, as previously mentioned, comprises several accelerated modules derived from the ONNX model, which are pipelined with an initiation interval (II) of 1 and interconnected sequentially in a streaming fashion. Of particular significance are the dense/convolution layers, primarily due to their substantial computational demands, accounting for nearly 90% of the total network execution time. To optimize the performance, we have implemented a layer fusion technique specifically targeting Convolution-ReLU layers. This optimization combines the convolution and ReLU activation operations into a single, streamlined processing step. By fusing these layers, we eliminate redundant data movements and reduce the overall computation time, leading to significant improvements in both latency and throughput.

II. *Multiplier Optimization.* We aimed to maintain 8-bit precision for all weights and biases, as this typically yields satisfactory inference accuracy [131, 132]. However, we employed mixed-precision data types for the inner activations. To determine the minimal required precision for each layer in the network, a statistical analysis was conducted on the weight and activation parameters.An innovative approach to multiplier design involved efficiently performing two multiplication operations within a single clock cycle using Xilinx's DSP48E2. Our objective was to devise an efficient method for encoding inputs a, b, and c in a way that the multiplication (a + b) × c could be easily separated into a × c and b × c. We achieved this by packing the two 8-bit inputs, a and b, into the 27-bit port of the DSP48E2 multiplier via the pre-adder, ensuring that the vectors were as far apart as possible, as illustrated in Figure 3-19.

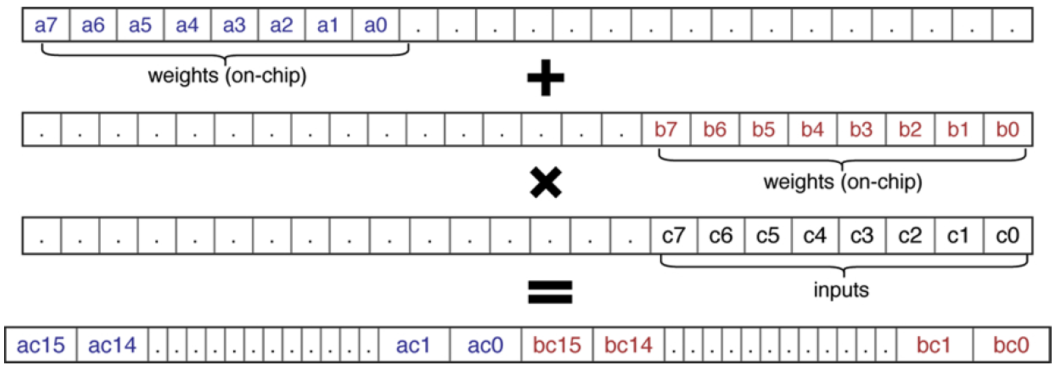


Figure 3-19. Packing two INT8 multiplications with a single DSP Slice.

As observed, the product of the packed port and an 8-bit coefficient produced the result without the bits of each vector influencing the computation of the other. To prevent any interference between the upper and lower bits, our 8-bit input required a minimum total input size of 24 bits (16 bits + 8 bits).

III. *Memory Optimization:* In neural networks, the same set of inputs or weights is often heavily reused in convolutional or dense layers. We opted for input sharing, as depicted in Figure 3-20, which involved parallel MAC operations of the type $a_i \times w_i$ and $a_i \times w_j$. By left-shifting the values using the discussed INT8 optimization technique, each DSP slice contributed to a partial and independent portion of the final output values.

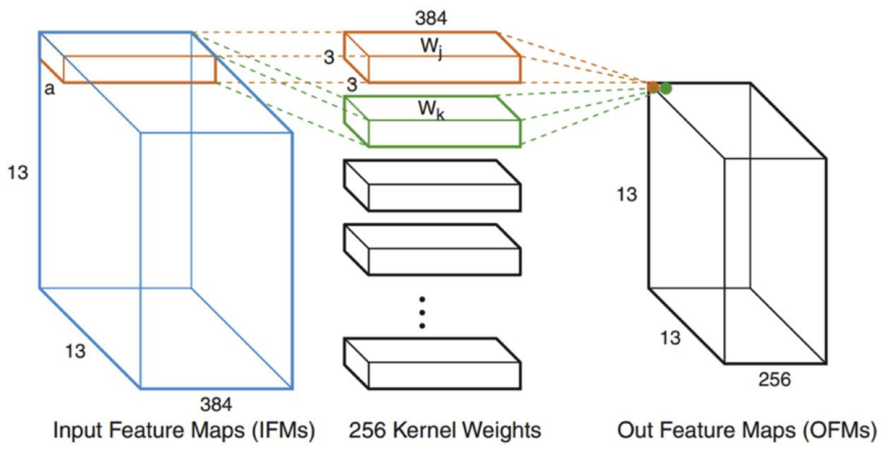
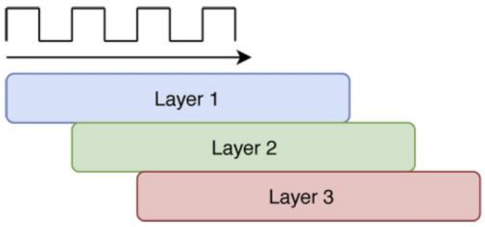


Figure 3-20. Illustration of input sharing technique

The common coefficient "c" in the multiplication a × c + b × c corresponds to the input ai of the partial sum of products $O_k = \sum_{i=1}^{N} a_k w_{i,k}$ in a convolutional layer. Additionally, we configured the neural network weights to be stored in the fast BRAMs of the FPGA device, allocating weights into matrix blocks within physically contiguous memory, situated near the computation of kernels. This ensured rapid communication, allowing one-cycle read-writes of vectorized data using the most efficient data movers. Furthermore, the blocks were fully partitioned (or partially for larger networks), enabling the FPGA device to simultaneously access multiple weight values and facilitate extensive parallelism.

IV. *Dataflow Optimization:* Following the optimization of multipliers and memory, we proceeded with the final integration of the acceleration modules by executing all layers in a unified kernel with a streaming architecture. We established channels based on FIFOs, enabling consumer layers to initiate operations before producer layers had completed, as illustrated in Figure 3-21. Each layer transmitted its output to the next layer using similar datatype layouts, allowing them to overlap their operations once sufficient data had accumulated in the previous layer. This increased the concurrency of the RTL design. Moreover, in the streaming architecture, there was no need to store intermediate results of each layer in off-chip memory since they were promptly passed downstream. This resulted in significant resource reduction and enhanced power efficiency, employing only the minimal resources required for our design. We also enabled batch processing for the forward operation, allowing users to input a packed array of N images for batch processing and receive results in an N × classes array containing classification probabilities. The *softmax* layer generated this array and provided all classification results after processing all images.



```
void forward(img, res) {
   save_weights_onchip();

   layer1(img, out1);
   layer2(out1, out2);
   layer3(out2, res);

   return;
}
```

Figure 3-21. Task level pipelining between layers

V. *Model and parameters:* Initially, we normalized the image values within a range of 0 to 1 by dividing them by 255 before inputting them into the neural network model. Subsequently, we divided the dataset into two parts: 60,000 images for training and 10,000 images for the test set. The MNIST model primarily comprises two dense layers, encompassing over 200,000 synapses, with activation layers interspersed. In contrast, the other model, designed for the CIFAR-10 dataset, is a variant of VGG-16, featuring three convolutional and pooling layers with activation layers in between. We opted for the ReLU activation function for both models due to its stability, speed, and efficiency, particularly when deployed on FPGA hardware.

VI. *The sign extension issue:* As previously explained, we've implemented an INT8 optimization for the multiplier in our hardware accelerator, which utilizes a packed port to perform two multiplications in a single DSP operation. However, the original neural network computations in both dense and convolutional layers involve signed-by-unsigned multiplications. This arises from the fact that the activations, stemming from ReLU or input layers, are consistently positive numbers, while the weights may include negative values. When multiplying these types of numbers, it becomes imperative to sign-extend the inputs, as illustrated below. This sign extension is necessary to execute the multiplications in an unsigned format and subsequently obtain the correct result in two's complement form; otherwise, the outcome may prove invalid. For instance, when we multiply two 8-bit numbers with one of them being signed, the sign extension process results in a 16 by 8-bit multiplication, ensuring that the outcome is represented in the appropriate two's complement format. This principle applies not only to integers but also to fixed-point numbers, as demonstrated in Figure 3-22, since we treat them as integers and subsequently ascertain their binary point position. Consequently, the INT8 optimization cannot be efficiently applied by default, as the packed port designed for single DSP multiplication cannot accommodate two 16-bit numbers simultaneously.

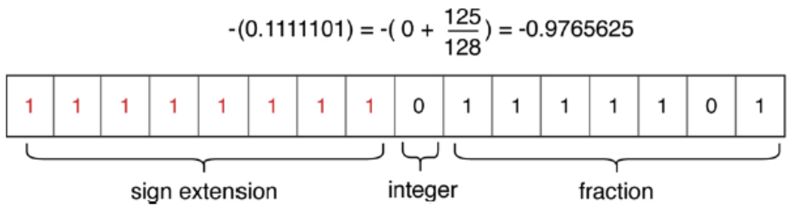


Figure 3-22. Sign extension for an 8-bit fixed point number

VII. *The weight regulation:* To address the aforementioned issue, we devised a strategy within the Tensorflow/Keras framework, which we employ for training our models. This strategy involves imposing specific constraints during the training process, akin to employing L2 regularization techniques to direct the growth of model weights. We opted for a per-layer non-negativity constraint on both weights and biases. Initially, this approach yielded suboptimal results, with some weights becoming excessively large positive values due to overfitting. To rectify this, we introduced additional guiding mechanisms, such as weight and bias initialization, as well as extending the number of training epochs. Ultimately, a combination of iterative retraining steps and pruning was instrumental in achieving a weight distribution within the [0, 2.5] range, with the majority of weights concentrated in the [0, 0.05] interval. In the final stages of post-training refinement, we reduced the integer width of the 8-bit fixed-point values, retaining only 1 bit for the integer component of the weights, since only a small fraction of values necessitated 2 integer bits.

Final system design using OpenCL — Following the successful implementation of a low-latency design, which leveraged fine-grain parallelism at the kernel level within the PL fabric, we proceeded to create an efficient OpenCL host API using standard OpenCL API calls. Specifically, we allocated image inputs using C++ vectors, with each vector sized at 28 * 28 * N, where N represents the number of images when batch processing is preferred. This allocation method ensured that the image matrix occupied physically contiguous memory, enabling the utilization of the most efficient data transfer mechanisms to and from DDRs. To optimize data throughput, we implemented a 512-bit user interface on each kernel side, leveraging the maximum memory bandwidth supported by Xilinx OpenCL FPGAs. Simultaneously, we made use of all available DDRs on the device, achieving peak data transfer rates. Additionally, we carefully crafted concurrency within OpenCL command queues for kernel initiation and synchronized interactions between the host and kernels, ensuring smooth operation within a constant dataflow paradigm.

Moreover, we took measures to minimize SLR crossing whenever feasible, as this tends to result in less efficient designs in terms of latency and power due to the creation of longer critical paths. Fortunately, our kernels generally remained within the resource boundaries of each SLR. Figure 3-23 provides an overview of the entire system, starting from the host CPU, traversing through the FPGA, and finishing in the calculations performed at the multiplier level in each Processing Element (PE). Notably, the final system, complete with all the optimizations described, was integrated into the HLS4ML package, accessible through the Python API, thereby implementing our custom architecture.

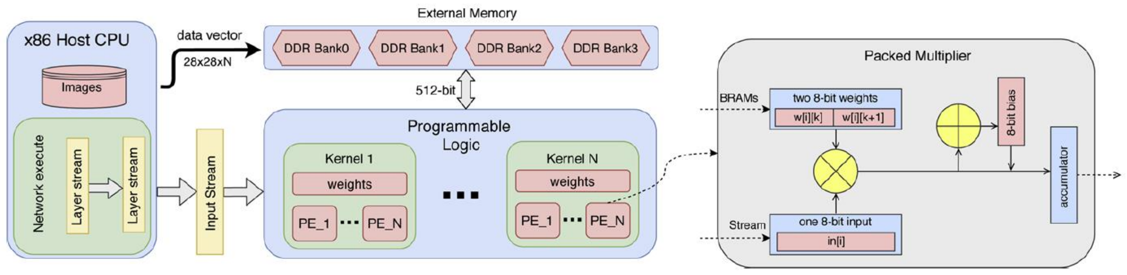


Figure 3-23. Final system dataflow

Evaluation and Results

In this section, we will conduct an assessment and profiling of our application. We will start by examining the hardware-aware training of a neural network, which serves as a test case, and assess its accuracy. It's worth noting that post-training quantization was employed prior to determining the ultimate inference accuracy. Following that, we will proceed with the performance evaluation of both the hardware accelerator and the fully operational system, assessing their performance on both the compact MNIST model and the larger CIFAR-10 model.

To illustrate our approach, we implemented two customized neural networks, as previously described, utilizing the Keras library. These models were specifically chosen because they are commonly used in cloud industry applications, addressing clothing classification and object recognition tasks, respectively. Figure 3-24 displays the validation accuracy of the first model across various training steps, comparing the default model with the regulated one. The slightly lower accuracy observed in the hardware-optimized model can be attributed to the weight initialization and constraints we introduced to fully leverage our proposed multiplier unit, enabling double packed MAC (Multiply-Accumulate) operations. The difference in validation accuracy between the two models is approximately 4%.

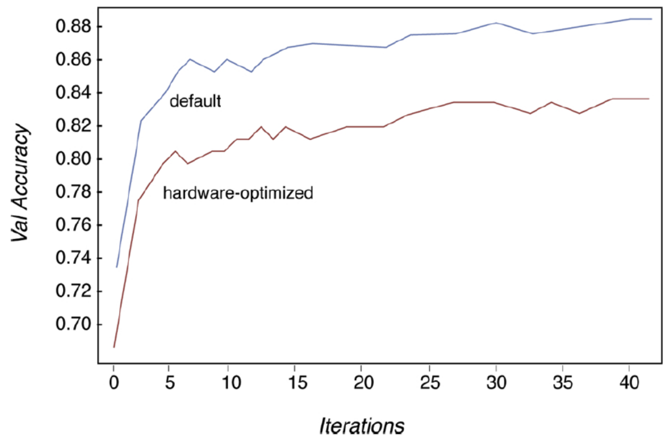
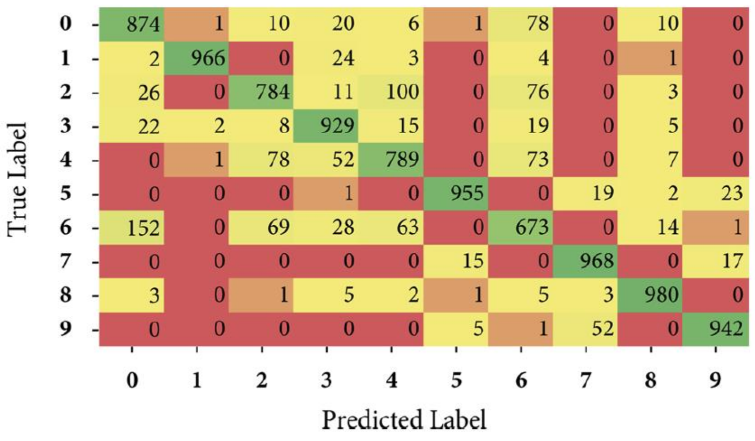


Figure 3-24. Accuracy comparison between neural network models

However, when evaluating the inference accuracy on the final system (which predominantly utilizes 8-bit weights and activations), we note only a 6% reduction in accuracy compared to the original model on average. This trade-off is quite reasonable, considering the substantial increase in performance achieved using the same computational resources, along with the advantages of 8-bit precision, such as reduced resource usage, lower latency, and power efficiency. In our specific case, we opted to visualize the classification accuracy through a confusion matrix (as depicted in Figure 3-25). This choice was made because conventional accuracy metrics often conceal the details of the classification model's performance. A confusion matrix provides a breakdown of correct and incorrect predictions for each class, offering insights not only into the classifier's errors but also into the impact of our hardware optimizations.



	0	1	2	3	4	5	6	7	8	9
0	874	1	10	20	6	1	78	0	10	0
1	2	966	0	24	3	0	4	0	1	0
2	26	0	784	11	100	0	76	0	3	0
3	22	2	8	929	15	0	19	0	5	0
4	0	1	78	52	789	0	73	0	7	0
5	0	0	0	1	0	955	0	19	2	23
6	152	0	69	28	63	0	673	0	14	1
7	0	0	0	0	0	15	0	968	0	17
8	3	0	1	5	2	1	5	3	980	0
9	0	0	0	0	0	5	1	52	0	942

Figure 3-25. Heatmap on the confusion matrix of the classification model

Accelerator performance — To evaluate the design, our initial focus was on verifying the correctness of the accelerator, particularly our custom multiplier, as incorrect DSP outputs could potentially disrupt the entire convolutional neural network. We conducted the hardware evaluation on a Xilinx Alveo U200 FPGA. This system boasts 64GB of off-chip RAM with a bandwidth of 77 GB/s and operates on a Gen3x16 PCI Express interface, running at a kernel clock speed of 300MHz. Table 3-6 provides detailed resource utilization and timing data for each neural network layer, categorized by layer type, in the case of the MNIST model.

	Utilization summary				**Timing**	
Layer	BRAM	DSP	FF	LUT	Latency (cycles)	FPS
Dense	192	64	2449	12656	1709	-
ReLu	0	0	16	127	130	-
Softmax	7	0	985	2401	51	-
Total	199	64	3450	15184	1890	158K

Table 3-6. FPGA resource utilization and latency per layer

Our streaming architecture exhibits a total latency that closely approximates the sum of initiation intervals for each layer, amounting to 1890 kernel cycles. The kernel also achieved the maximum device frequency of 300MHz, with the theoretical potential to reach up to 400MHz, as evident from the Worst Negative Slack (WNS) of 0.8ns. Consequently, our neural network accelerator can complete a full forward pass in 1890 kernel cycles or 6.3 microseconds, which translates to an impressive 158,000 frames per second (FPS). For instance, consider the scenario of processing a full HD 1920 × 1080 video stream at 30 FPS, where the video is segmented into 28 × 28 tiles for neural network inference. To handle this task in real-time, a neural network inference rate of 80,000 FPS would be required, a demand that our system successfully meets. Lastly, Figure 3-26 provides a visualization of coarse-grained concurrency at the task level, highlighting the parallelism achieved by employing four kernels simultaneously, with data computation and transfer occurring concurrently. This illustration was directly generated from the SDAccel framework.

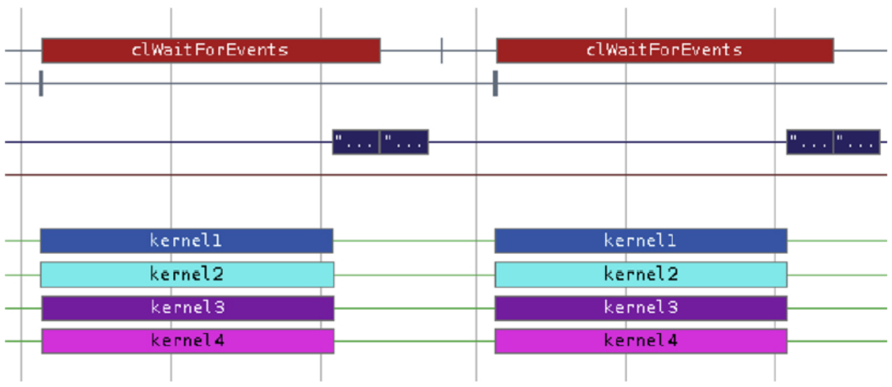


Figure 3-26. Task level parallelism with concurrent kernel execution

Final system performance — In the final evaluation of our system, we initiated tests to determine the end-to-end execution time required for a single forward pass through our two neural networks, namely MNIST and CIFAR, encompassing memory transfers within the process. Subsequently, we compared these results with the execution times achieved on a single-core Xeon CPU and an Nvidia Tesla P100 GPU, with the observation that the timings for the two neural networks might appear similar due to device overhead considerations. As indicated in Table 3-7, our final FPGA system excels in terms of performance and demonstrates lower average power consumption, especially for the smaller MNIST model. This superiority in performance and efficiency can be attributed to several factors. Firstly, our system employs reduced-precision weights and activations, as opposed to the default 32-bit floating-point representations. Combined with the utilization of a packed multiplier, our system achieves minimal latency, a crucial attribute for applications demanding rapid processing. Furthermore, it's worth noting that our design utilizes only a small fraction of the available device resources, rendering it highly power-efficient. Consequently, it can be deployed not only in data centers but also in smaller embedded FPGA SoCs, catering to critical applications that necessitate low latency and energy efficiency.

Device Information			**Performance Evaluation**				**Power Evaluation**	
System	Model	Architecture	CIFAR	Speed-up	MNIST	Speed-up	Watt(avg)	Perf./Watt (max)
CPU	Xeon 2.4 GHz	22-nm	58 ms	1 ×	43 ms	1 ×	9 W *	1 ×
GPU	Nvidia P100	16-nm	1.1 ms	52.7 ×	0.75 ms	57 ×	95 W	5.4 ×
FPGA	Alveo U200	16-nm	2.7 ms	21.5 ×	0.42 ms	102.3 ×	31 W	29.7 ×

* Scaled to single-core

Table 3-7. Evaluation vs other architectures

To measure the performance of our systems in comparison to other projects sharing a similar architectural context, we conducted a meticulous examination of related projects involving FPGA designs aimed at addressing comparable problems or domains (for further details, refer to the Related Work section). Among these projects, one of the closest in resemblance was an 8-bit accelerator developed by . Jiong Si et al. [129] for MNIST , which our design significantly outperformed, achieving a fivefold increase in performance.When comparing our design with systems utilizing different architectures, we executed identical benchmarks on both a CPU and a GPU system, with the baseline reference being the Xeon CPU. Notably, our FPGA design excels in terms of performance on the MNIST model, outpacing both CPU and GPU systems. It attains a remarkable 102.3× speed-up over the CPU and a substantial 1.8× improvement over the GPU.

Furthermore, the power efficiency of our FPGA architecture excels, particularly evident in the Performance/Watt metric. Here, our FPGA achieves a substantial 29.7-fold speed-up compared to the CPU and a 5.5-fold improvement compared to the GPU. It's noteworthy that the FPGA exhibits superior power efficiency in both the CIFAR and MNIST model cases.

Conclusion — In this study, we introduced a novel and revamped framework designed to autonomously generate FPGA firmware using High-Level Synthesis (HLS) based on neural network models. We implemented various optimizations and extended the functionality of the existing hls4ml package, enhancing its speed and efficiency. For illustrative purposes, we meticulously trained and fine-tuned two customized neural networks, one smaller and one larger in size, with optimizations tailored for hardware acceleration. These networks were specifically designed for image classification tasks, such as recognizing clothing or objects, which are common applications in the cloud computing industry. Our research findings demonstrated that the proposed architecture can surpass the performance and power efficiency of other high-end platforms like CPUs or GPUs. Additionally, we compared the performance of our accelerator with other FPGA designs mentioned in the Related Work section, including those utilizing reduced-precision neural networks, and consistently showcased its superiority. From a research perspective, we are actively working on further enhancing performance and incorporating additional features into the package. One such feature under consideration is the integration of hardware-aware training, as discussed in this work. The area of potential design space trade-offs is extensive, and our research contributes valuable insights to this domain, achieving successful results and aiming to establish FPGAs as fundamental contributors to the evolving open software-hardware ecosystem.

Acknowledgments

This project was funded from the Xilinx University program and the Hellenic Foundation for Research and Innovation (HFRI) and the General Secretariat for Research and Technology (GSRT), under grant agreement No 2212- Hardware Acceleration of Machine Learning Applications in the Cloud.

4

Simulation of Approximate Deep Neural Networks

On the previous chapter we saw how we can optimize AI/ML applications for hardware accelerators such as FPGAs. Approximate hardware, on the other hand, involves trading off some level of precision in calculations to gain performance or energy-efficiency benefits. AI workloads, particularly deep learning, are often tolerant of minor errors or approximations in calculations. Approximate hardware designs can exploit this tolerance by using reduced-precision arithmetic or other approximation techniques to speed up AI computations while sacrificing minimal accuracy. Computing in these architectures is called *approximate computing* and can help reduce latency, power consumption or area, which are critical considerations in AI, especially for edge devices with limited power budgets. Nevertheless, the development period for these hardware devices is lengthy, thus determining the error or the impact of approximate hardware on an AI model without having the hardware yet can be challenging. One method is to use simulation tools and software libraries that allow to apply approximate computing techniques to an AI model's computations. However, popular DNN frameworks do not support approximate arithmetic because only libraries of accurate mathematical functions are inherently supported, thus emulation becomes extremely slow. In this chapter, we formulate, validate, and assess a framework for emulating approximate DNNs to address this challenge. This enabled us to estimate the impact of approximation on accuracy and power for several approximate multipliers on the popular Pytorch framework. We describe how our framework was built to provide acceleration support for both CPU but also for GPU for arbitrary approximate multipliers and neural network models in a seamless flow in order to perform fast approximate inference. Last, we introduce a Monte Carlo Tree Search (MCTS) algorithm to efficiently search the space of possible configurations using a hardware-driven hand-crafted policy, allowing us to derive close to pareto-optimal solutions.

4.1 Motivation and Challenges in Simulating Approximate DNNs

Deep learning methods, which are based on neural networks, have demonstrated remarkable success in various applications like image processing due to their high effectiveness and precision. Deep Neural Networks (DNNs), for instance, excel in achieving exceptional accuracy and performance in tasks like visual recognition and complex regression algorithms. Nevertheless, when it comes to neural networks, the inference process of the model demands a substantial number of multiply-accumulate operations (MACs) and memory accesses, resulting in significant energy consumption and time overhead [133]. The computational intricacy associated with this, along with the inherent error resilience of deep learning, has spurred significant research into the development of approximate DNN accelerators [38].

The primary objective of approximate computing is to realize substantial reductions in computational resource and memory usage. This approach entails the reduction of model parameters or activations to a lower numerical precision through the use of fixed-point arithmetic, as opposed to the conventional 32-bit floating-point precision. Prior research has demonstrated that selecting an optimal bitwidth for the model's multiply-accumulate (MAC) operations can result in negligible errors [134]. Many accelerators employ integer quantization, such as INT8. Consequently, implementing approximate MAC operations, which predominantly rely on integer arithmetic, can yield significant enhancements in performance, power efficiency, and energy efficiency on the hardware side [135]. For instance, energy savings ranging from 35% to 81% have been reported, with less than a 1% loss in accuracy [136].

The vast and diverse landscape of approximate arithmetic units (e.g., [137, 68]) and their intricate impact on DNN accuracy increase the complexity of design, underscoring the need for an approximate emulation framework. Furthermore, as DNNs increase in depth, they become more sensitive to approximation [138]. Consequently, fine-tuning DNNs with approximation awareness is essential to mitigate the errors introduced by naive approximation methods (e.g., replacing exact multipliers with approximate ones) and attain high inference accuracy [136]. Mainstream DNN frameworks lack support for approximate arithmetic, as they inherently prioritize libraries of accurate mathematical functions, leading to slow emulation.

This chapter introduces two high-speed emulation frameworks for approximate DNN accelerators developed in PyTorch: AdaPT, which leverages AVX intrinsics and multithreading for CPU acceleration, and TransAxx, which utilizes GPU acceleration with CUDA. The frameworks' primary goal is to accelerate and facilitate the simulation of AI models on approximate hardware. They function as an integrated PyTorch plugin that seamlessly integrate with most AI models, including Convolutional Neural Networks (CNNs), Variational Autoencoders (VAEs), Long Short-Term Memories (LSTMs) and Generative Adversarial Networks (GANs) for AdaPT and even Vision Transformers (ViT) for TransAxx. They represent innovative emulation platforms, enabled first time for PyTorch which is a popular ecosystem within the AI research community [139]. They are also capable of performing approximate inference across a wide range of bitwidth representations, including mixed precision. State-of-the-art techniques are employed for model quantization and approximate retraining, enabling further accuracy enhancements.

4.2 AdaPT: Operation and Optimization Techniques

We conceived the AdaPT framework as a rapid cross-layer DNN approximation emulation tool, presented as a PyTorch plugin. Users have the flexibility to enable or disable it, reverting to the PyTorch default flow when needed. The framework seamlessly supports a wide array of layers and model architectures. We offer support for two key techniques to enhance accuracy: post-training quantization, employing cutting-edge calibration, and approximate-aware retraining. Users have the freedom to select an approximate compute unit (ACU) for integration into AdaPT as a self-contained component or to stick with the default precise flow. Furthermore, AdaPT is equipped to handle mixed precision and mixed approximation, allowing the use of different ACUs between layers. However, it's important to note that fine-grained approximation, such as per-filter approximation, is not currently supported. To accelerate the approximate DNN emulation, AdaPT leverages the power of OpenMP threads and Intel AVX2 intrinsics for advanced vectorization.

Related Work — Popular deep learning frameworks like Caffe and TensorFlow have undergone extensive examination within the research community when it comes to simulating approximate convolutional neural networks (CNNs) for image recognition [68, 70]. However, in recent years, PyTorch has emerged as the standard for both DNN training and inference. To the best of our knowledge, no prior work has demonstrated support for approximate DNN emulation within the PyTorch framework. Furthermore, most previous studies have primarily concentrated on 8-bit quantized CNNs exclusively for image recognition simulations. In contrast, AdaPT offers a broader spectrum of support, accommodating various bitwidths (e.g., 4-bit, 8-bit, 12-bit, and so on) for diverse types of DNNs and applications. This includes CNNs for image recognition, LSTMs for text classification, and VAEs and GANs for image reconstruction, with built-in support for approximation-aware retraining. Several pre-RTL simulation frameworks have been developed for approximate DNNs, including AxDNN [140] and TypeCNN [141].

TypeCNN's evaluation focused on two Neural Networks (NNs) based on the Lenet architecture, utilizing a custom C++ framework without CPU optimizations. AxDNN, on the other hand, combines precision-scaling and pruning techniques with the simulation of approximate hardware, resulting in approximately a 20-fold simulation speedup compared to default RTL simulation, primarily for power analysis purposes. Other frameworks like Ristretto [142] can assess various bitwidth representations but lack support for approximate arithmetic. In contrast, ALWANN [68] and TFApprox [70] implemented different variations of ResNets using approximate units with 8-bit weights. Notably, ALWANN reported extended simulation times (approximately 1 hour for ResNet50). Meanwhile, TFApprox, along with ProxSiM [143] performed emulation and evaluation on a GPU, achieving low inference times on the Tensorflow framework, although limited to 8-bit inference. ProxSiM also featured re-training capabilities for 8-bit multipliers but did not present results for popular DNNs.

Quantization optimization — To effectively simulate approximate compute units, it is essential to implement an efficient quantization scheme that minimizes the impact of errors. Previous research has predominantly concentrated on 8-bit quantization [70, 143]. However, in AdaPT, we have integrated a versatile bitwidth quantizer based on Nvidia's TensorRT toolkit [144], which offers support for both lower and higher precision levels. This flexibility is particularly valuable when simulating higher precision ACUs for various DNNs that exhibit limited error tolerance, such as compact CNNs [145, 146]. We opted for this approach due to its open-source nature and its state-of-the-art capabilities. The development of a new quantization method falls outside the scope of this paper. Our quantizer is built for mapping real numbers to integers and can be applied to both weights and activations, typically found within Convolutional or Linear layers. This mapping involves an affine relationship, which means that the real value is calculated as the product of a scale factor (A) and the quantized value, with an offset (B), which is often set to zero, expressed by the equation: *real_value* = A × *quantized_value* + B.
To determine the optimal quantization parameters for the scale values, we employed the calibrator class from TensorRT to collect data statistics. In our quantization modules, we implemented the histogram calibrator, specifically targeting the 99.9th percentile, as it demonstrated the best overall performance. However, it's worth noting that other methods, such as Mean Squared Error (MSE) or entropy, can be used seamlessly. Instead of simply identifying the maximum absolute value in our dataset, our calibrator learns the offline *calib_max*, which represents the absolute maximum input value that can be represented in the quantized space at the 99.9th percentile. Weight ranges are determined on a per-channel basis, while activation ranges are calculated per tensor, a strategy supported by previous research, and known for its efficacy [146]. By processing just a single batch of images, our learnable *calib_max* can be optimally configured for most DNNs, resulting in approximately 0.1% error for the majority of 8-bit CNNs. Optionally, following post-training quantization, we can enable Quantization Aware Training (QAT)

by continuing to train the calibrated model based on the Straight Through Estimator (STE) derivative approximation. The process is visually depicted in Figure 4-1. AdaPT mitigates the impact of approximation during training by incorporating fake quantization modules, which operate with quantized floating-point values to mimic the rounding effects associated with true integer quantization. This approach effectively computes the layer gradients. During QAT, the model performs propagation through our ACUs, typically for 10% of the default training schedule, ensuring that retraining is aware of the approximation.

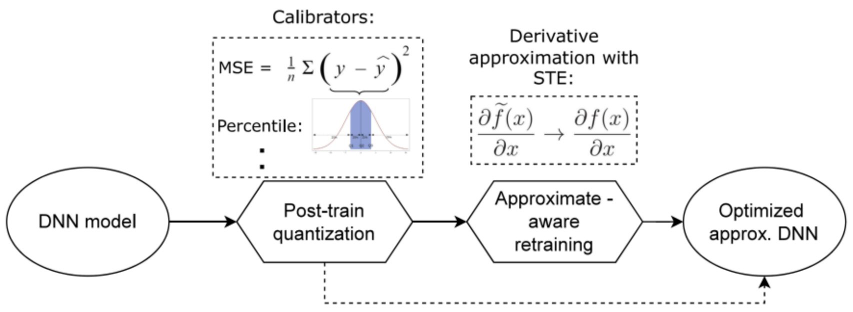

Calibrators:
MSE = $\frac{1}{n}\Sigma\left(y-\hat{y}\right)^2$
Percentile:
Derivative approximation with STE:
$\frac{\partial \tilde{f}(x)}{\partial x} \rightarrow \frac{\partial f(x)}{\partial x}$
DNN model
Post-train quantization
Approximate - aware retraining
Optimized approx. DNN

Figure 4-1. Quantization & approximation-aware retraining flow before inference

Approximate units — In AdaPT, users have the flexibility to specify whether each DNN layer should be considered as accurate or approximate during its definition. It is possible to designate any desired ACU for each layer (or for all layers), provided that the output of the multiplier remains deterministic. This section highlights the most commonly encountered layers that we have re-engineered and adapted for approximation.

1. *Convolution Layer*: Typically applied in 2D convolution scenarios, commonly found in CNNs, this layer takes an input tensor X with (N, C_{in}, H_{in}, W_{in}), where N represents the batch size, C_{in} is the number of channels, , H_{in} is the height, and W_{in} is the width. The output is represented by a tensor Y with dimensions (N, C_{out}, H_{out}, W_{out}). We transformed the filters into a 2-D matrix and the input matrix into another matrix, ensuring that the product of these two matrices computes the same dot product as the original 2D convolution. This transformation aims to facilitate a more efficient implementation for accelerating AdaPT's emulation by simplifying the computation through matrix multiplication. Our convolution layer accommodates various input dimensions, kernel sizes, padding, striding, and groups, making it capable of simulating a wide range of DNN configurations.

2. *Separable Convolution:* In the case of separable convolution, the core idea involves breaking it down into a two-step computation: depthwise and pointwise 2D convolutions, as expressed in the equations:

$$y = \mathrm{Conv2D}(C_{in}, C_{in}, H_{in}, W_{in}, groups = C_{in})\ (x)$$

$$out = \mathrm{Conv2D}(C_{in}, C_{out}, H'_{in}, W'_{in}, groups = 1)\ (y)$$

The first equation comprises the depthwise convolution which is equivalent to a Conv2D with groups equal to C_{in}channels. Next the output is fed to the pointwise convolution which is same as Conv2D with 1 × 1 kernel size.

3. *Linear Layer:* The Linear Layer is commonly used in Multi-Layer Perceptrons (MLPs), often appearing in the final layers of DNNs, as well as in models like GANs and VAEs. Similar to 2D convolution, the equivalent PyTorch layer performs a matrix multiplication, expressed as $y = xA^T + b$, where the input matrix is multiplied by the weight matrix and an optional bias vector is added.

4. *RNN Layer:* Recurrent Neural Network (RNN) layers are typically employed for tasks involving temporal relationships, such as text classification or speech recognition. We have incorporated the feedback loop within the recurrent layer to enable it to retain information over time, following a mathematical approach equivalent to the vanilla PyTorch RNN layer. It also utilizes our custom Linear layer, rendering it compatible with approximation. Likewise, for Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) layers, we have included the 'memory cell,' which can store information over extended time intervals.

Framework operation — The operation of the AdaPT framework is illustrated in Figure 4-2. Initially, the user configures the desired DNN model, specifying quantization parameters such as precision and the calibrator to be used. Additionally, the user defines the approximate module to be utilized from the library, along with the dataset for the DNN models. It's important to note that for the training dataset, only a representative subset is required, which typically constitutes around 10% of the original training set, primarily for calibration purposes. Subsequently, AdaPT identifies the supported layers within the DNN and retrieves the appropriate layer class from its layer library. For the approximate multiplier, the corresponding Look-up Table (LUT) is generated from AdaPT's LUT generator, organized as a cache-line aligned C-array. This design allows CPU cores to efficiently access data from the same cache segment. Furthermore, an additional tool is employed to translate a hardware description into a C function. In cases where larger bitwidths may substantially increase LUT sizes, AdaPT can replace LUT-based multiplication with function-based multiplication, where the approximate multiplier is represented in C-code. While this approach can mitigate memory-related challenges associated with large LUTs (greater than 15 bits), it may introduce overhead in DNN execution time. Importantly, both approaches provide an equivalent high-level representation of the ACU, ensuring consistent results during quantization or retraining.

AdaPT aims to populate the CPU cores' cache with LUTs as much as possible to minimize cache misses. Finally, just-in-time (JIT) compilation dynamically loads the layer extension using the Ninja build system. The produced inference and retrain engines are then linked with the final approximate DNN layers, which replace the corresponding vanilla PyTorch layers through a graph re-transform tool. This tool analyzes the layers and recursively substitutes PyTorch layers with their approximate equivalents. Ultimately, users have the option to fine-tune the model using the provided training subset to achieve even higher accuracy or proceed with approximate evaluation.

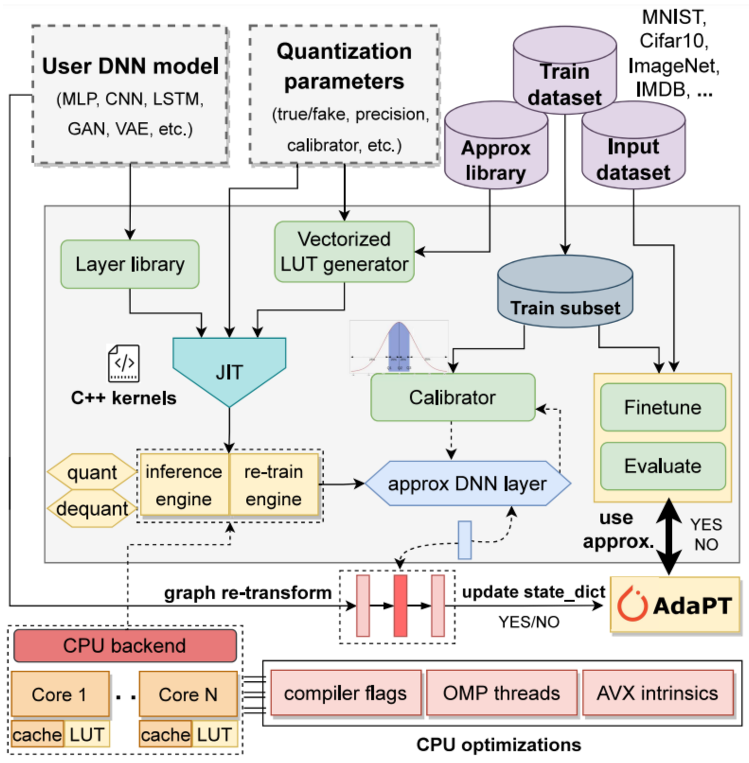


Figure 4-2. AdaPT framework operation

Acceleration approach — The most demanding computational task within the implemented layers revolves around matrix multiplications. For instance, the original 2D convolution is reconfigured into a matrix multiplication. To accommodate approximate units, we devised a solution by implementing LUT-based multiplications between individual input and filter values. This approach allows us to compute any approximate unit without the necessity of directly implementing its corresponding function, except in cases where the LUT sizes are exceptionally large, as previously mentioned. Subsequently, the table look-ups are executed in parallel using a hybrid model of parallel programming, which leverages OpenMP threads and CPU vectorization.

- *Thread Parallelism:* In AdaPT, we have developed an efficient batched Conv2D implementation that seamlessly scales with input size, avoiding memory errors by leveraging the thread parallelism provided by OpenMP. This approach allows us to implement loop-based parallelism shared across batches, achieving nearly linear scaling as the input data size increases. The primary objective of this approach is to unify the utilization of incremental parallelism through a common interface, simplifying the application development process.
- *Vector Parallelism:* The second layer of parallelism is introduced during the execution of each thread, which pertains to the implementation of parallel table lookup using Single Instruction Multiple Data (SIMD) instructions. The goal here is to accelerate the gather operation for the lookup table data, a process involving data retrieval from disparate memory locations and its consolidation into a continuous memory space. To facilitate efficient memory access, all values of the SIMD are organized in contiguous memory, which aligns with the structure of our AdaPT tensors. The indices, comprising activations and weights, are packed into vector registers. The AVX2 instruction set is then employed to execute the gather instruction and vectorize the task, collecting memory locations from the lookup table into the destination vector register. We have chosen to use AVX2 intrinsics because of their wide support in Intel CPUs manufactured from 2013 onward and their compatibility with AMD CPUs. Figure 4-3 illustrates the process of utilizing vectorized loads in a 2D convolution scenario.

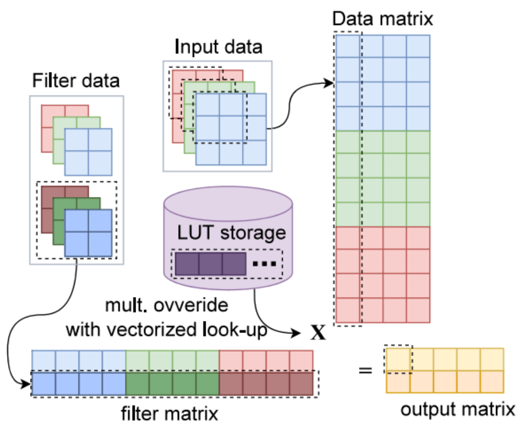


Figure 4-3. 2D convolution to matrix multiplication with LUT override

Acknowledgments

This work has been supported in parts by the E.C. funded program SERRANO under H2020 Grant Agreement No: 101017168 and in parts by the German Research Foundation (DFG) project “ACCROSS” HE 2343/16-1.

4.3 TransAxx: Operation and Optimization Techniques

Previously, we have described a framework for emulation of approximate DNNs using CPUs. Now, we introduce a novel platform that harnesses the power of GPU acceleration through CUDA cores and the CUDA programming model. This tool named TransAxx encompasses all the key features of AdaPT, including approximate-aware retraining, support for arbitrary multipliers, various precision levels, and diverse model architectures. Notably, it was built with the aim to emulate Vision Transformers (hence the name), a first in the field of DNN simulation frameworks, but can also be used with various CNN models as well. Furthermore, it boasts a more streamlined design, enabling the automatic use of pretrained models without requiring manual intervention in the model's code from the user’s perspective. Its execution on GPUs results in increased speed enabling to emulate large AI models if needed without significant execution time while keeping a very user-friendly interface. Using TransAxx, we were able to analyze the sensitivity of transformer models on the ImageNet dataset to approximate multiplications and perform approximate-aware finetuning to regain accuracy as we will show in the experimental evaluation.

Motivation behind TransAxx — CNNs demonstrate the capability to achieve impressive accuracy and performance in visual recognition and complex regression algorithms. Beyond CNNs, recent advancements in deep learning have given rise to Vision Transformer (ViT) models. These models, based on the self-attention mechanism of transformers, have achieved state-of-the-art performance in various computer vision tasks. However, ViT models are computationally expensive due to their numerous parameters and the self-attention mechanism, limiting their use on resource-constrained devices. The application of approximate computing has shown potential in enhancing the efficiency of deep learning models by reducing computational complexity and memory requirements [147]. This involves sacrificing a small amount of accuracy for significant gains in speed and power efficiency through the use of inexact arithmetic components instead of accurate counterparts [136, 148]. While approximate computing is promising for improving DNN efficiency, prior research has not explored its application to transformers. The broad domain of approximate compute units (ACUs) and their non-trivial impact on DNN accuracy complicates the design of such hardware. Therefore, there is a need for an approximate emulation framework to address this complexity. As DNNs become deeper, they become more sensitive to approximation [138], necessitating approximation-aware retraining to correct errors introduced by approximation [138, 70]. In the case of ViT models, the distortion of the self-attention map, involving a large number of operations, may lead to even greater errors with lower precision [149]. Moreover, determining the appropriate approximate multiplier for each DNN layer is crucial when aiming to maximize power gains under accuracy loss constraints [70]. While many works have explored automated methods for determining optimal per-layer quantization in quantized DNNs [150, 151, 152], few have focused on an automatic search flow in approximate DNNs [137]. If ViT models are intended for use, especially on large datasets like ImageNet, the computational cost poses a significant design challenge.

***Related Work* —** Taking cue from the remarkable achievements of transformer models in the domain of natural language processing (NLP), scientists have recently utilized transformer models to address computer vision (CV) challenges. However, just like CNNs, ViT models are troubled with heavy computational cost, thus applying quantization techniques, which is lowering the bit-width of the weights or activations can often address the memory requirement and computational cost of the model. Despite the recent progress made in developing quantized ViT models, approximate ViT models and their behavior remains an open issue. Also, automated mixed precision techniques for quantized CNNs have been already investigated by the community to strike a balance between the computational efficiency and the numerical stability but there is no research for approximate ViT models.

Li et al. proposed Q-ViT [149], a fully differentiable quantization technique for ViT models that focuses on a head-wise bit-width and switchable scale. Ding et al. [153] proposed APQ-ViT which introduced a unified bottom-elimination blockwise calibration scheme to surpasses the typical post-training quantization which causes significant performance drops. Last, Liu et al. [42] presented an effective post-training quantization algorithm that finds the optimal low-bit quantization intervals for weights and inputs and introduced a ranking loss to keep the relative order of the attention layer values.

Approximate computing takes a different approach, on top of quantization, by intentionally introducing errors into computations in exchange for reduced computational cost. Several previous works have presented custom DNN frameworks to simulate the accuracy of approximate CNNs. For example, TFapprox [70] or ProxSim [143], are frameworks built on top of TensorFlow to emulate approximate circuits in CNNs using GPU accelerators. Mrazek et al. proposed ALWANN [68], a methodology to apply layer-wise approximation on 8bit ACUs with finetuning capabilities. The experimental results of these works, however, are limited mainly to ResNet CNN models for small datasets like Cifar-10. Other frameworks, such as AdaPT [147] and ApproxTrain [154] experimented on a wide range of DNNs but did not propose a design space exploration strategy while posing difficulties for the users to test their custom models.

Support for the transformer architecture — The transformer layer stands as the foundational element within the vision transformer architecture. Its primary role involves taking a sequence of image patches as input, leveraging the self-attention mechanism to establish long-range relationships, and generating a new feature sequence. Additionally, there are supplementary blocks positioned at the start or end of a ViT model, such as patch embedding or a classification head. In patch embedding, the input image undergoes division into fixed-size, non-overlapping patches, with each patch linearly embedded into a flat vector. These patch embeddings subsequently serve as input tokens for the transformer model. Toward the conclusion of the ViT model, a classification head is typically present, tasked with making predictions based on the learned features. Our approach focuses on applying approximation exclusively to the core transformer blocks that involve the self-attention mechanism, which significantly dominate the execution time (usually exceeding 98%). These blocks are frequently expanded into multiple transformer encoder blocks, each primarily comprising normalization, multi-head self-attention, and a feed-forward layer. Towards incorporating approximate arithmetic, we concentrate on the latter two, which entail the majority of mathematical operations.

The attention mechanism can be precisely defined through the following equation, where the softmax function is employed to compute weights determining the significance of each element in the input. In this context, Q represents the query vector, K is the key vector, and V is the value vector:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

In summary, it computes the correlation between the Query vector and the Key vector, followed by multiplying the corresponding Value for each Key. The division by the square root of d_k (dimensionality of the KK vector) is carried out to ensure an appropriate variance of attention values.

- Multi-head Attention: The multi-head attention technique involves employing multiple sets of (Q, K, V) triplets instead of just a single set. This approach addresses scenarios where an element in a sequence relies on dependencies with more than one other element. The utilization of multiple weights associated with the same element facilitates a more comprehensive weighting of the sequence. The multi-head attention mechanism is explained further below. Each head has its own set of learned parameters and performs the scaled dot-product attention operation separately.

$$MultiHead(Q, K, V) = Concat(head_1, \dots, head_h)W^O$$
$$\text{where } \; head_i = Attention(QW_i^Q, KW_i^K, VW_i^V).$$

- Feed-Forward Network: The Feed-Forward Network (FFN) is a two-layer classification network featuring a GELU (Gaussian Error Linear Unit) activation layer. Its purpose is to facilitate non-linear interactions among patches or tokens in the input feature map, enabling the model to grasp intricate patterns and relationships. Given the substantial computational requirements of the FFN layer, there is often a need to apply approximate computing methods to enhance efficiency. The formulation of the FFN layer is as follows:

$$FFN\left(\vec{X}\right) = GeLU\left(\vec{X}W_1 + b_1\right)W_2 + b_2$$

$$\text{where } W_1 \in \mathbb{R}^{d\times d_f}, b_1 \in \mathbb{R}^{d_f} \text{ and } W_2 \in \mathbb{R}^{d_f\times d}, b_2 \in \mathbb{R}^{d}.$$

$$d_f \text{ is the inner hidden size of the Feed} - \text{Forward Network.}$$

Quantization and fine-tuning strategies — To simulate approximate computing units effectively, it is crucial to implement an efficient quantization scheme that minimizes the impact of quantization errors. It's worth noting that the majority of approximate multipliers, especially those designed for Deep Neural Networks (DNNs), support low bit-width integer (fixed-point) arithmetic [136]. Previous efforts in approximate CNN simulators have predominantly focused on standard 8-bit quantization [70, 143]. However, in TransAxx, we employ a versatile bitwidth quantizer, similar to AdaPT's quantizer. As our paper does not aim to propose a new quantization method, we utilized a state-of-the-art open-source quantizer based on Nvidia's TensorRT toolkit, offering flexibility for both lower and higher precisions. This adaptability becomes crucial when simulating higher precision Arithmetic Computing Units (ACUs) for deep neural networks that may lack error resilience [145, 146].

The mapping between real and quantized values must be affine, following the equation $\text{Real\_value} = \text{A} \times \text{Quantized\_value} + \text{B}$,, where A represents the scale and B is the zero point (often set to zero). To determine suitable quantization parameters for the scale values, we employed a calibration technique to gather data statistics. Our quantization modules were implemented with the histogram calibrator, targeting a 99.9% percentile, as it consistently delivered optimal performance. However, alternative methods like Mean Squared Error (MSE) or entropy can be seamlessly applied in our framework if desired. Moreover, optionally after post-training quantization, we can perform Quantization Aware Training (QAT) by continuing to train the calibrated model based on the Straight Through Estimator (STE) derivative approximation. Notably, in TransAxx, QAT is approximation-aware as it simulates the approximate noise of ACUs during the fine-tuning stage, eventually demonstrating increased robustness to the applied multipliers. During approximate-aware retraining, TransAxx propagates computations through our ACUs (typically for 2.5% of the default training schedule), effectively computing layer gradients using STE, and ultimately enhancing the final accuracy of the approximate ViT model at the end.

Toy experiment — A toy experiment was conducted to test the effectiveness of backpropagation on a simple attention layer that used an 8-bit approximate multiplier instead of the default FP32 arithmetic of PyTorch. The core attention mechanism works by focusing on specific parts of the input sequence based on their relevance to the current output. The goal of this experiment is to see if a simple layer with the core attention mechanism can backpropagate correctly and ensure, thus, that its functionality is not compromised by the approximate hardware emulation.

To perform the experiment, the attention layer was modified to use a `mul8s_1L2H` approximate multiplier from the Evoapprox lib [137] for the forward and backward passes. The modified layer was then trained using a standard Stochastic Gradient Descent (SGD) algorithm for 500 iterations on data taken from $\mathcal{N}(0,1)$.The results presented in Figure 4-4 showed that backpropagation on our framework works as expected minimizing the MSE while the target values in the layer exhibit similarity with the accurate layer. Mathematically, the output data Y converges in probability to the target data X because as the number of samples (n) approaches infinity, the probability that the difference between Y_n and X being greater than some small value epsilon

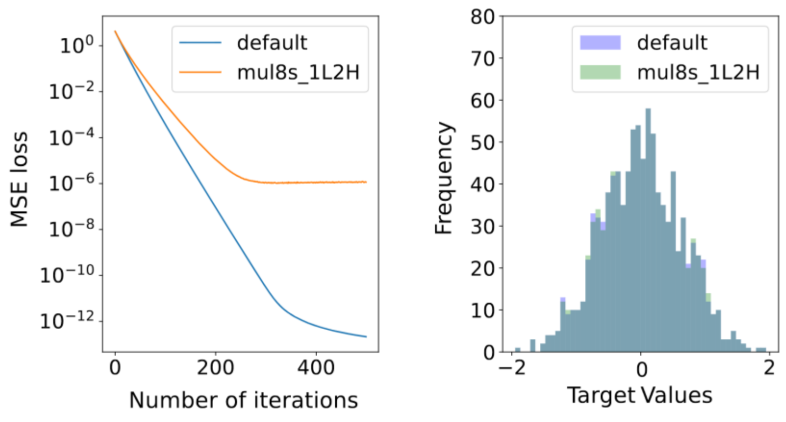


Figure 4-4. Preliminary testing with an approximate attention layer. Left: MSE loss per training iteration. Right: Histograms of target data (using FP32) and output data (using approx. multiplier) distributions from the layer's inference.

Framework operation — TransAxx framework, developed for the rapid emulation of cross-layer DNN approximation, is accessible as a PyTorch plugin. Users have the flexibility to activate or deactivate this plugin as needed, allowing the utilization of the PyTorch default flow when required. TransAxx seamlessly supports a broad spectrum of layer and model architectures without necessitating user intervention. In this work, our focus centers on ViT models, as there has been no prior exploration of approximations for such models. Additionally, we introduce two key techniques to enhance accuracy: post-training quantization with advanced calibration and approximate-aware retraining.

Furthermore, TransAxx accommodates mixed approximation, involving different multipliers between layers. One of the primary challenges associated with incorporating approximate components in DNNs is the imperative to emulate approximate operations swiftly. This is particularly crucial as existing DNN GPU-based accelerators do not inherently support such computations. To address this challenge, we implement a universal GPU accelerator designed to operate across all Nvidia GPU architectures, thereby accelerating the emulation of approximate ViT models.

Designing the framework — Our framework, shown in Figure 4-5, offers an orthogonal approach to simulating approximate ViT models. The primary functionalities are outlined below:

- *Extension of default PyTorch modules:* TransAxx aims to handle computations within approximate ViT models that involve non-differentiable operations or dependencies on non-PyTorch libraries. To achieve this, it extends the default PyTorch modules, allowing our custom functions to seamlessly integrate with the existing computational graph. During the model compilation, our framework automatically swaps the vanilla PyTorch layers with the custom layers, converting the default model to the desired approximate equivalent. These layers are instantiated on-the-fly using just-in-time (JIT) compilation, ensuring efficient integration with the model's computational graph. JIT also makes the model flexible and easy to modify during runtime. This method supports incremental compilation, which means that only the parts of the code that have changed are recompiled. This significantly reduces the overhead of repeatedly compiling and loading TransAxx's layer extensions during experimentation.

- *Layer initialization and kernel dispatching:* Regular Tensor objects within PyTorch are leveraged to handle the initialization of weights or biases for the custom kernels. This ensures consistency with PyTorch's initialization mechanisms, maintaining compatibility and ease of use within the framework. Then the weights/activations are quantized based on the layer's multiplier bitwidth and calibrated using the statistics of the layer's activations. Last, our framework utilizes C++ macros to dispatch the appropriate GPU kernel per layer.

- *Generation of Look-up Tables (LUTs):* For each approximate multiplier, a corresponding LUT is generated from its high-level description (e.g., in C, Matlab, or behavioral HDL). We have an integrated tool within TransAxx that can generate this LUT for any arbitrary approximate multiplier, whose behavior is described in C or HDL. This is facilitated by running all possible hardware multiplications x×y (e.g., using an RTL simulation) for the given approximate multiplier and then storing the results in a LUT. Hence, LUT[x][y] gives the approximate product of x and y. During the forward pass, TransAxx uses these LUTs and substitutes the default (exact) multiplication operator with the approximate product (i.e., loading the value LUT[x][y]). The LUT was a design choice to help reduce the emulation time of TransAxx. Initially, LUTs are stored in the global memory as it has a large size and it's the most appropriate option for the random access patterns of the LUTs. However, next, we show how we improved these memory accesses.

- *GPU kernel optimization*: It is crucial to carefully consider memory transfers, as operations involving LUTs can quickly become memory-bound. Notably, the cache behavior of the GPU is influenced by both hardware specifications and the specific memory access patterns of the ViT model. However, given that LUT data is read-only, we can guide the Nvidia compiler to maximize memory access throughput (i.e., by using CUDA intrinsics and compiler flags). Using this approach, the LUT array will be typically cached through the L1 GPU cache, which offers low latency and can be shared among all threads within a CUDA core. This facilitates efficient caching and access to LUT data across multiple threads.

- *Handling large bitwidths:* For scenarios where LUTs may grow substantially in size, particularly with large bitwidths (> 12 bits), our framework provides a flexible solution. TransAxx can dynamically substitute LUT-based multiplication with functional-based multiplication (in which the approximate multiplier is alternatively described in C-code). This process can introduce computational overhead in the DNN execution time but ensures that our framework remains efficient and scalable. It's worth mentioning that both approaches provide a 1-1 representation of the multiplier at high-level thus the results would be the same in inference or retraining. Also, it is important to note that transformers work well with low precision values [149], and higher bit-width, which might hinder TransAxx emulation time performance, is often not required.

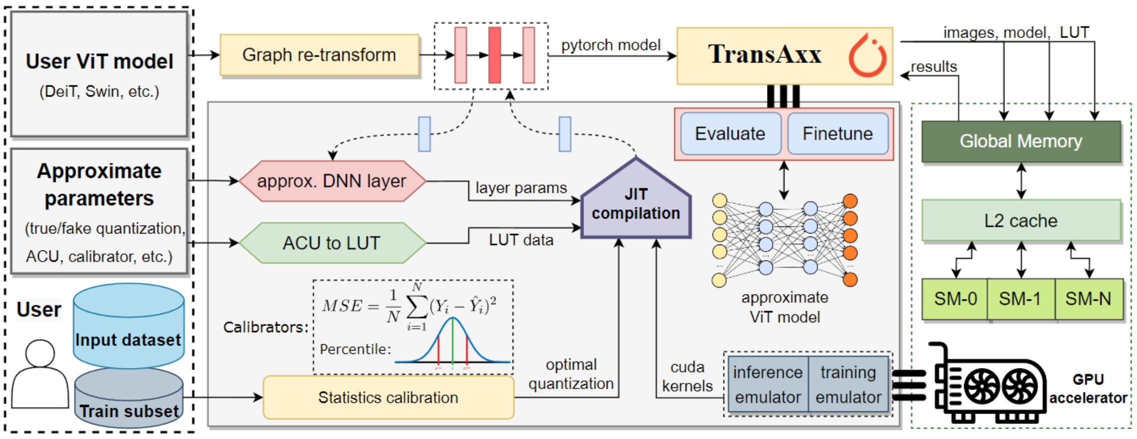


Figure 4-5. TransAxx framework operation

4.4 Automated Design Space Exploration

Automated Design Space Exploration plays a pivotal role in the field of optimizing Deep Neural Networks (DNNs), particularly when addressing the delicate balance between model accuracy and computational efficiency. This field becomes especially crucial in scenarios where deploying large and intricate models may be impractical due to resource constraints or real-time requirements. In the context of optimal approximation within DNNs, researchers aim to systematically explore the vast design space, encompassing hyperparameters, architectural choices, and optimization techniques. The objective is to discover configurations that strike an optimal trade-off, leveraging techniques like quantization, pruning, knowledge distillation, and algorithmic approximations. In our work, we focused on quantization/approximation optimization, that is applying specific approximate multipliers in each DNN layer in order to achieve an optimal balance between accuracy and power consumption. Specifically the method is applied for ViT models which pose a challenge for their complexity. The synthesis of approximate accelerators can be described as a multi-objective optimization problem. Let A denote the synthesized approximate accelerator. The search process can be modeled as an optimization problem below:

$$A = arg_X \text{Cost}\,(X)$$

where X is a vector representing the design parameters and characteristics of the accelerator, and Cost(X) is a cost function that captures the trade-offs between factors such as accuracy, resource utilization, and energy efficiency.

4.4.1 Literature Review

For automatic exploration of the design space for accelerators usually machine learning techniques are used. In the context of hardware design, the design space refers to the set of possible configurations and parameters that can be chosen to meet specific performance or efficiency criteria. Accelerators, in this case, typically refer to specialized hardware components designed to accelerate specific computational tasks, often used in the context of deep learning or other compute-intensive applications. The automatic design space exploration for approximate ViT models in our case is accomplished by simulating the approximate multipliers used in the model. It’s a systematic approach towards finding optimal configurations for neural network models by considering different levels of precision/approximation in arithmetic operations, particularly in the multiplication operations performed within the network. With the use of our proposed frameworks, AdaPT and TransAxx, instead of implementing each potential multiplier configuration in hardware and measuring its impact through physical synthesis, the methodology becomes seamless by employing their simulation capabilities. Due to the extensive range of approximate implementations within libraries such as EvoAapprox [137], users gain access to a diverse set of implementation options. This diversity allows them to effectively navigate the tradeoff between Quality of Results (QoR) and energy consumption (or other hardware parameters) at the accelerator level. However, even for accelerators involving only a few operations, determining the optimal combination of approximate compute units (ACUs) becomes an intractable task. There are several previous works that addressed the challenge of identifying the most suitable replacements for arithmetic operations within a target accelerator, drawing from the available approximate circuits in such libraries. Given that this is a multi-objective optimization problem, there isn't a singular optimal solution; rather, multiple solutions typically exist. The focus is on pinpointing approximate circuits within the Pareto frontier, which encompasses the non-dominated solutions.

There have been many works, especially in the recent years, that involve the automation of the end-to-end process of optimizing DNNs. Automated Machine Learning (AutoML) is a field that includes a variety of tasks, and optimizing neural network quantization/approximation is a part of it in which we focus on this work. Algorithms and methods such as Neural Architecture Search (NAS) or genetic algorithms have been employed to automatically discover the optimal architecture or configurations for neural networks. This includes exploring different quantization and approximation schemes to find the right balance between accuracy and computational efficiency. Also, Reinforcement Learning (RL) - based approaches can also guide the search process, where the model learns from its past experiences to make decisions on the quantization and approximation strategies for DNN layers.

For example, mixed-precision quantization shows promise in providing an extra boost in speed and reducing model size by taking advantage of model redundancy and assigning lower bit-width to less sensitive or less useful layers in the model. However, the challenge lies in accurately measuring each layer's *sensitivity* score and mapping it to the appropriate bit-width. A lot of techniques have been proposed such as HAWQV3 [152], a hardware-aware mixed-precision quantization formulation that uses Integer Linear Programming (ILP), or deep reinforcement learning (DRL) based quantization methods such as AutoQ [151] and HAQ [150]. While training RL agents for common hardware might be feasible, training such algorithms for simulated approximate DNNs, especially ViTs, would immensely exceed the computational and timing constraints. Approximate DNN frameworks run much slower than the default DL frameworks (i.e., [70, 68]) due to the lack of adequate support for approximate arithmetic. Additionally, the learned policies of RL agents to find optimal approximation per layer would degrade when evaluated on new ACUs or models as each ACU may behave very differently on layers with different data distributions.

4.4.2 Overview of Monte Carlo Tree Search (MCTS) Algorithm

This paragraph will present a comprehensive background of Monte Carlo Tree Search (MCTS) algorithm. MCTS is a proven intelligent stochastic search algorithm particularly effective in handling problems characterized by large branching factors, such as in the domain of games. Grounded in the search for optimal hardware configurations that balance accuracy and performance, our research navigates the intricate field of approximate accelerator design, leveraging the heuristic search algorithm of MCTS to efficiently traverse the vast space of hardware solutions. Before we proceed with our proposed algorithm implementation, in this paragraph we will analyze the fundamentals of the classic MCTS algorithm. Our novel method for this problem will enable to deal with the large Design Space Exploration, which unveils a broad range of potential approximations through lightweight random simulations as we will discuss in the following paragraphs.

Principle of operation — Monte Carlo Tree Search centers its attention on evaluating the most promising moves by systematically expanding the search tree through random sampling of the search space. In the context of game applications, MCTS relies on numerous playouts, commonly referred to as roll-outs. In each playout, the game unfolds to its conclusion by randomly selecting moves. The outcome of each playout serves as a basis for weighting the nodes within the game tree, favoring superior nodes for future playouts. A fundamental approach to employing playouts involves applying an identical number of them after each legal move by the current player. The selection of the move that led to the most victories in these playouts characterizes this method, known as Pure Monte Carlo Game Search. The effectiveness of this strategy typically improves over time as more playouts are assigned to moves that have demonstrated success for the current player. The iterative process of Monte Carlo tree search encompasses four key steps:

1. *Selection*: Commencing from the root R, successive child nodes are chosen until a leaf node L is reached. The root represents the current game state, and a leaf node is any node with a potential child from which no simulation (playout) has been initiated.

2. *Expansion*: Unless L concludes the game definitively (e.g., win/loss/draw), one or more child nodes are created, and node C is chosen from them. Child nodes encompass any valid moves from the game position defined by L.

3. *Simulation*: A single random playout is conducted from node C, also referred to as a playout or rollout. This process can be as straightforward as selecting uniformly random moves until the game reaches a resolution (e.g., victory, defeat, or draw), as observed in chess.

4. *Backpropagation*: The result of the playout is utilized to update information in the nodes along the path from C to R. This step, often termed backpropagation, ensures that the knowledge gained from the simulated playout influences the evaluation of subsequent moves in the search tree.

The above steps can be visualized as in the following figure.

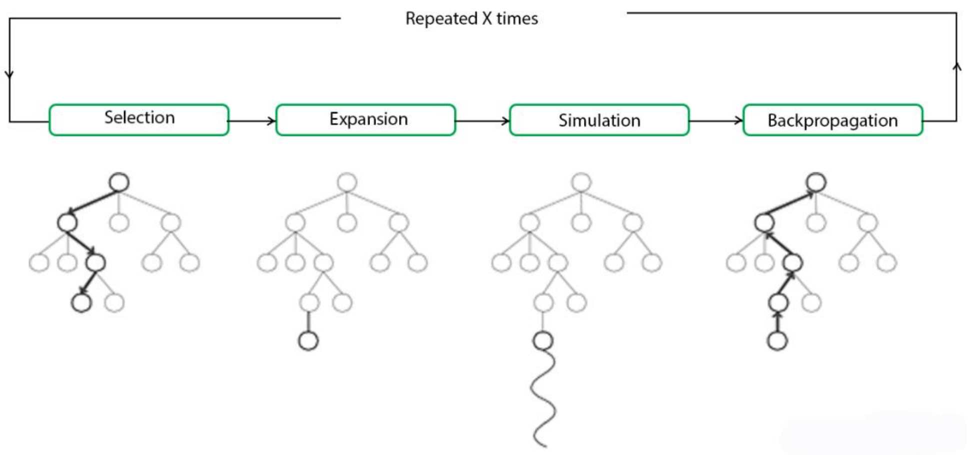


Figure 4-6. Steps of Monte Carlo tree search.

Exploration and exploitation — The primary challenge in selecting child nodes lies in achieving a delicate balance between exploiting deep variants following moves with a high average win rate and exploring moves with limited simulations. The initial formulation addressing this balance, known as UCT (Upper Confidence Bound 1 applied to trees), was pioneered by Levente Kocsis and Csaba Szepesvári. UCT builds upon the UCB1 formula developed by Auer, Cesa-Bianchi, and Fischer, incorporating the probably convergent Adaptive Multi-stage Sampling (AMS) algorithm initially employed in multi-stage decision-making models, particularly Markov Decision Processes, by Chang, Fu, Hu, and Marcus. Kocsis and Szepesvári advocate the selection of moves within each node of the game tree based on the expression:

$$\frac{w_i}{n_i} + c\sqrt{\frac{\ln N_i}{n_i}}$$

In this formula:

- w_i stands for the number of wins for the node considered after the i-th move.
- $n_i$stands for the number of simulations for the node considered after the i-th move.

- N_i stands for the total number of simulations after the i-th move run by the parent node of the one considered.
- c is the exploration parameter—theoretically equal to √2; in practice usually chosen empirically.

The first component of the formula above corresponds to exploitation; it is high for moves with high average win ratio. The second component corresponds to exploration; it is high for moves with few simulations. We will take advantage of these characteristics in our MCTS-based design space search as we will discuss next.

Below, we summarize the main advantages and disadvantages of using MCTS:

MCTS advantages :

- Implementation of MCTS proves straightforward, rendering it user-friendly.
- As a heuristic approach, Monte Carlo Tree Search (MCTS) demonstrates its efficacy in the absence of specific domain knowledge, relying solely on the understanding of rules and end conditions. Through the exploration of random playouts, MCTS autonomously identifies optimal moves and learns from them.
- MCTS allows for the preservation of its state at any intermediate point, facilitating future utilization as needed.
- The versatility of MCTS extends to supporting asymmetric expansion of the search tree, adapting dynamically to the operational circumstances it encounters

MCTS disadvantages :

- The rapid growth of the tree structure after a few iterations demands a substantial amount of memory, posing a resource challenge.
- Monte Carlo Tree Search exhibits a reliability issue in certain circumstances, particularly in turn-based games, where a single branch or path might lead to a disadvantageous outcome against the opponent. This challenge arises from the vast number of possible combinations, with some nodes not being visited frequently enough to discern their long-term results.
- The effectiveness of the MCTS algorithm is often contingent on a large number of iterations, introducing a speed issue as it requires significant computational effort to determine the most efficient path.

4.4.3 MCTS-Based Automated Search for Optimal Approximation

In this part of our work, we will propose the use of Monte Carlo Tree Search for approximate accelerator design space exploration. However, recognizing the inherent dissimilarities between the two domains, that is game theory and approximate computing, substantial modifications were essential to seamlessly tailor MCTS for approximate accelerator simulation. To the best of our knowledge, this is the first work that adapts MCTS for approximate DNN simulation, specifically for Vision Transformer models.

Design Space Size — The exploration-oriented design space exploration process heavily relies on an extensive search to discover and validate potential solutions. However, the overall size of the search space, indicating the total number of conceivable solutions, becomes exceedingly vast and expands exponentially, particularly for medium-sized DNNs as the number of candidates and approximate transformations increases. Given that many existing MCTS-related works depict the search space as a tree, in our case tree nodes signify distinct approximate DNN variants generated during the DSE. Assuming the total number of combinations (or approximate variants) achievable for an DNN accelerator, denoted as N, with the number of candidates as C (which is the number of model layers in our case) and the total possible approximations applicable to a candidate represented as A, the growth of N follows an arithmetic progression, expressed as A0 * A1 * A2 * ... * AC as the tree deepens. For example, for a DNN with 10 layers, the number of possible hardware configurations, that is applying in each layer an approximate multiplier from a set of 4 multipliers/ACUs equals 4^{10} = 1048576.

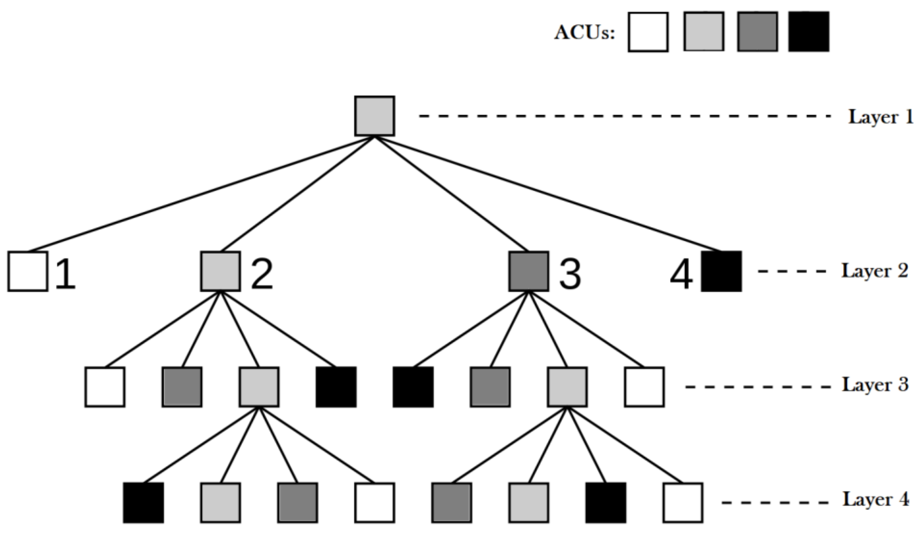


Figure 4-7. Visualization of the tree expansion in DNN layers when 4 ACUs are used.

As the number of candidates, along with their associated transformations, increases, the assessment of all potential combinations or nodes within the search tree becomes unfeasible due to the exponential expansion of the tree's size. The previous figure illustrates the total number of combinations (or nodes within a tree) in relation to the escalating number of candidates (assuming only four possible transformations for each candidate). In practical terms, considering the average time taken for an approximate DNN inference simulation (considering medium-size CNNs using GPU acceleration) – approximately 3 minutes – it translates to a simulation time of 3min*1048576 $\approx 6\ years$. In some applications commonly found in AutoML, where the number of candidates and approximate transformations tends to be even higher than the aforementioned example, thoroughly exploring such an extensive search space is impractical within a reasonable timeframe. To efficiently explore the approximate design space we will use a *hardware-driven* version of Monte Carlo tree search to narrow down the architecture space for the approximate ViT models, maximizing accuracy while still meeting our given power constraints. To further reduce feedback time, we also developed an accuracy predictor for the inference and reach a near Pareto-optimal curve of power and accuracy.

Rationale for employing MCTS — Monte Carlo Tree Search is an AI search technique, often used in board games, that uses probabilistic and heuristic-driven algorithms to combine the classic implementation of tree search with principles from machine learning and particularly reinforcement learning. There has been only a few previous works that utilized MCTS on the AutoML domain, however it is a different domain than ours. Specifically some previous works have investigated the use of MCTS-based methods for hyperparameter tuning regarding CNNs [155, 156]. Also, regarding the design of electronic circuits there are also a few number of works but they focus on solving the routing problem [157, 158, 159]. Now, in our research area regarding approximate simulation frameworks, MCTS has not been investigated for finding optimal approximate configurations in DNNs, specifically ViTs.
MCTS has the ability to dynamically balance exploration and exploitation, making it less susceptible to getting stuck in local optima compared with other methods such as greedy algorithms which are often used for quantization precision search [160, 161]. Specifically, the case of using inexact arithmetic can introduce additional sources of error that can make the optimization problem more complex to solve as it requires a greater degree of exploration than the greedy methods. In addition to these methods, RL agents and genetic algorithms have also been used for finding optimal hardware configurations in CNNs [150], though applying them for our case would be unrealistic. The reason over not training an RL agent is twofold. Training and converging an agent would require a lot of data which would greatly exceed the computational and timing constraints by simulating a vast number of ViT models. Also, the effectiveness of the learned policies of the agent for determining the optimal approximation per layer may deteriorate when assessed on new ACUs or models, given that different ACUs exhibit significant variability in their

behavior across layers with distinct data distributions. Besides RL approaches, comparing with the use of genetic algorithms, MCTS has the potential to find a good solution faster and more reliably by balancing exploitation and exploration, which is crucial for this computationally expensive design space. Also, the effort required to optimize the parameters of the MCTS algorithm (e.g. exploration coefficient) is lower than the effort required for the genetic algorithms [162]. Thus it would be problematic to apply genetic algorithms towards this complex problem which inhibits high variability across different ACUs.

Operation of MCTS-based search — Exposing the optimal configuration of approximate multipliers between each layer of a DNN model in order to find the best trade-off between performance and power is liable to cause a significant computational overhead. The design space becomes large as mentioned earlier and measuring the accuracy of every configuration is not feasible even when using our GPU-based acceleration, particularly in our case, where the ViT simulation increases further the execution time. In order to systematically navigate the space of approximate design solutions, we employ a Monte Carlo tree search (MCTS) that is specifically tailored to be *hardware-driven*. This approach helps accelerate the exploration of architectural configurations for approximate Vision Transformer (ViT) models, striking a balance between maximizing accuracy and adhering to predefined power constraints. Additionally, to expedite the feedback loop, we have devised an accuracy predictor for estimating inference accuracy. As a result, our methodology achieves a nearly Pareto-optimal curve, effectively balancing power consumption and accuracy.

In MCTS, nodes are the building blocks of the search tree. The high level description of our approach (also shown as pseudocode in Algorithm 1) is as follows:

1. Create a root node with an initial state of the model.
2. Traverse the tree *selecting* the node with the best Upper Confidence Bound (UCB) value according to UCB equation until a leaf node is reached.
3. If the leaf node is not terminal, *expand* it by creating child nodes for all possible actions from that state.
4. Simulate a *rollout* from the selected child node based on the input policy *P* until a terminal state is reached. Here, we take actions by choosing a potential ACU for the layer l_i of the ViT model. We followed a head-wise approach similar to [149], making decisions for individual heads within the multi-head attention layers of the ViT. The terminal state is defined as the point when all layers l_i of the *L* layers in the model have been assigned an ACU A_i. This can be expressed as $\mathrm{Model}(A_i)_{i=1}^{L}$, indicating the model's configuration at the terminal state. Then, we compute the reward of the current approximate configuration of this rollout.
5. *Backpropagate* the reward obtained from the rollout up to root and update all UCB values of visited nodes.

UCB equation: $$S_i = x_i + c\sqrt{\frac{\ln N_i}{n_i}},$$

where S_i the value of node i, x_i the empirical mean of node i, c the exploration constant, N_i the total number of simulations up to i and n_i the number of visits of the node.

Algorithm 1 Pseudocode for our hw-driven MCTS

```
Input: 1) Model $M$, 2) Ground truth batch $B_g$, 3) Selected ACUs $A$,
4) Exploration constant $c$, 5) Rollout policy $P$, 6) No. of Simulations $N$
Output: 1) Optimal Approx. Configs $C_{out}$
 1: $rootNode \leftarrow Node(M)$
 2: for $i \leftarrow 1$ to $N$ do
 3:     $node \leftarrow rootNode$
 4:     while not $node.isTerminal()$ do
 5:         if $node.isFullyExpanded()$ then
 6:             $node \leftarrow node.getBestChild(c)$
 7:         else
 8:             $node \leftarrow expand(node)$
 9:             break
10:         end if
11:     end while
12:     $state \leftarrow node.state$
13:     while not $state.isTerminal()$ do
14:         $a \leftarrow chooseAction(state, P, A)$
15:         $state \leftarrow state.takeAction(a), a \in A$
16:     end while
17:     $Y_i \leftarrow AxxConfig(state)$
18:     $accuracy_i, power_i \leftarrow evaluate(M, Y_i, B_g)$
19:     $reward \leftarrow accuracy_i - \lambda \times power_i$
20:     $backprop(node, reward)$
21: end for
22: $C_{out} \leftarrow pareto(accuracy_i, power_i, Y_i), \forall i \in [1, N]$
```

The cycle of selection, expansion, simulation and backpropagation continues until it reaches the user-defined time limits. Every terminal state is expressed as $S = (A_1, A_2, \ldots, A_L)$ and all the environment-specific information that is relevant for the decision-making process has been included, specifically the power consumption of the current configuration and the output from the accuracy predictor. Also, the exploration-exploitation ratio can be tuned using c variable from the UCB formula. The accuracy from the predictor is computed in each rollout of the MCTS tree as evaluating the real accuracy on the whole dataset (ImageNet in our case) would be impractical. Root Mean Square Error (RMSE) was used as a metric to reflect the magnitude of the error of the target models. The output was compared with the ground truth using the square root of the typical MSE formula: $\sum_{i=1}^{N}(x_i - y_i)^2$ for 128 input samples which was proven to suffice the majority of our scenarios. In general, determining where to allocate the right multiplier is not straightforward. The aim of MCTS is to explore alternative paths which might be perceived as sub-optimal so that the system can avoid getting stuck in local optima. Figure 4-8 represents the normalized true and predicted accuracy (red and blue bars respectively) after applying the `mul8s_1L2H` ACU in the first 5 layers individually for every target ViT model of our study. Through these ablation studies we show our primary objective, which is not to achieve precise predictions of the accuracy, but rather to produce estimations that capture the general trend of the actual accuracy. Also, upon observation of the figure, it is evident that each layer block has a different *sensitivity* towards the perturbation of the final accuracy. This sensitivity list is beneficial for producing a better policy during the rollout phase of MCTS as we discuss next.

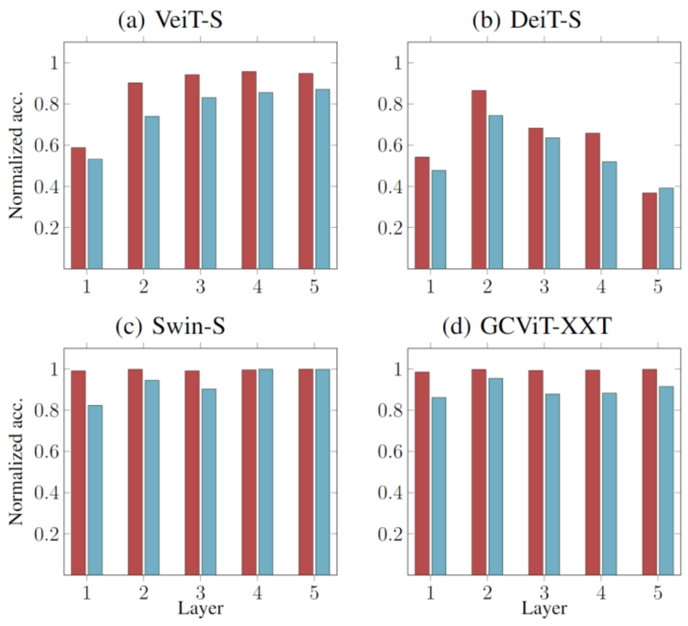


Figure 4-8. Comparison of actual (red) and predicted (blue) accuracy after applying approximation to each layer individually (from layer 1 to 5) across different ViT models.

Defining a better policy — Rollout policy is usually a simple heuristic to estimate the reward of a given state by randomly choosing actions until the terminal state is reached [163]. It is often implemented as a random policy, where actions are selected uniformly at random without any particular strategy but with the aim to explore a wider range of states. Using domain-specific knowledge of the sensitivity of each layer to approximation, we implemented a more sophisticated policy. Let $S = (s_{j,1}, s_{j,2}, \dots, s_{j,L})$ be the layer sensitivity list of an ACU A_j. $s_{j,i}$ is the normalized accuracy of the model when ACU A_j with $j \in [1, k]$ is applied only on layer *i*. Similarly, we can represent the total returned power of the approximate model when A_j is applied on layer i as $p_{j,i}$. Conclusively, we can now express the probability of taking a specific action in the rollout policy, that is to select an A_j out of *k* available ACUs for layer *i* as:

$$P(A_j)_i = \frac{e^{(s_{j,i} - \lambda \times p_{j,i})}}{\sum_{z=1}^{k} e^{(s_{z,i} - \lambda \times p_{z,i})}}$$

We incorporated expert knowledge to our problem by producing a better-informed rollout than the default heuristic. We took feedback from each layer but also incorporated randomness as well. This approach reflects *optimism in the face of uncertainty* which as a mathematical concept has underpinned many decision-making algorithms [164, 165].

4.5 Experimental Evaluation

In this section we present the evaluation of AdaPT and TransAxx frameworks regarding several DNN networks with their respective quantization calibration, approximation-aware training and simulation time. Also, the specifications regarding each model's parameters, number of MAC operations (OPs) and dataset used will be shown. The experiments were conducted on Intel Xeon Gold 6138 CPU at 2.00GHz and 64GB RAM for AdaPT CPU experiments while for TransAxx experiments the same system was used along with an Nvidia Tesla V100 GPU. Furthermore, we will show the effectiveness of our MCTS algorithm for efficiently searching the space of approximate DNN designs.

4.5.1 Experiments with AdaPT Framework

First, each target model (Table 4-1) undergoes assessment based on five metrics: accuracy in FP32, quantized (with and without calibration), approximate, and models subjected to approximate-aware re-training. Post-quantization calibration involves the utilization of two batches of images, each set at 128, for histogram collection using a 99.9% percentile method. For the retraining of the DNNs we employed Stochastic Gradient Descent (SGD) with a learning rate of 1e-4 and a batch size of 128, utilizing 10% of the corresponding training datasets as retrain subset. We showcase five DNNs across various tasks, including image recognition (ResNet50, VGG19, SqueezeNet), text classification (LSTM-IMDB), and image reconstruction (VAE-MNIST).

DNN	Type	Dataset	Params	OPs
ResNet50	CNN	CIFAR10	23.52M	0.33G
DenseNet121	CNN	CIFAR10	6.96M	0.23G
VGG19	CNN	CIFAR10	38.86M	0.42G
Fashion-GAN	GAN	Fashion MNIST	0.28M	0.29M
VAE-MNIST	VAE	MNIST	0.65M	0.66M
LSTM-IMDB	LSTM	IMDB	0.58M	0.55G
Inceptionv3	CNN	ImageNet	27.16M	2.85G
SqueezeNet	CNN	ImageNet	1.24M	0.36G
ShuffleNet	CNN	ImageNet	2.28M	0.15G

Table 4-1. Specifications for each DNN used in AdaPT's experiments

For retraining demonstration purposes, two approximate multipliers, implemented as Look-Up Tables (LUTs), are used with distinct Mean Relative Error (MRE) and Mean Absolute Error (MAE) values from the EvoApprox library [137]. One has 8-bit precision and low power consumption but higher MRE, while the other has 12-bit precision with lower MRE but higher power consumption. The top-1 accuracy metric is generally employed, except for ImageNet models, which use top-5. The results presented in Table Table 4-2 indicate that post-training quantization achieves low accuracy error compared to the original FP32 models (approximately 0.1%), primarily due to calibration. Calibration proves to be crucial for modern neural networks, especially larger ones, as emphasized in [166]. Through our approximate-aware retraining, DNNs can be adapted to the approximate backward engine, resulting in increased accuracy for the target DNN.

mul8s_1L2H	**MAE: 0.081 %, MRE: 4.41 %, power: 0.301mW[1]**					
DNN	**FP32**	**8bit**	**8bit calib.**	**8bit approx.**	**retrain[3]**	**time**
ResNet50	93.65%	93.55%	93.59%	82.69 %	93.44%	763s
VGG19	93.95%	93.80%	93.82%	90.7%	93.56%	318s
VAE-MNIST	99.99%	99.95%	99.96%	93.12%	99.88%	9.28s
LSTM-IMDB	83.10%	82.90%	82.95%	79.9%	82.63%	710s
SqueezeNet	80.6%	79.01%	80.16%	62.01%	76.21%	620s
mul12s_2KM	**MAE: 1.2e-6 %, MRE: 4.7e-4 %, power: 1.205mW[2]**					
DNN	**FP32**	**12bit**	**12bit calib.**	**12bit approx.**	**retrain[3]**	**time**
ResNet50	93.65%	93.60%	93.61%	93.52%	90.54%	798s
VGG19	93.95%	93.80%	93.81%	93.81%	93.71%	359s
VAE-MNIST	99.99%	99.98%	99.98%	99.98%	99.99%	10.11s
LSTM-IMDB	83.10%	82.94%	82.96%	82.96%	83.12%	1040s
SqueezeNet	80.6%	80.11%	80.3%	80.35%	80.50%	623s

Table 4-2. Accuracy and retrain time evaluation for AdaPT on various DNNs

[1]power of 8bit exact: 0.425mW. [2]power of 12bit exact: 1.210mW.
[3]approx. multiplier & approximation-aware retrain.

In line with prior research [167] most models are retrained for a single epoch, achieving significant performance improvements. The effectiveness of AdaPT's retrain engine is evident in substantial error recovery of approximation, particularly in the 8-bit ACU (approximately 7.5% increase on average). Further fine-tuning with learning rate annealing marginally reduces the error.

Furthermore, we provide a summary of the emulation time for each approximate DNN, as presented in Table 4-3. Moreover, the incorporation of quantization and dequantization in each layer introduces an overhead of approximately 10% for the optimized approximate solution. The comparison of inference is conducted using 8-bit precision to align with related work, ensuring an unbiased assessment (the specific approximate module can be arbitrary, given their implementation as LUTs). Notably, the emulation time experiences a linear increase when different ACUs are utilized within the same DNN, occurring in-between the layers. It is worth mentioning that employing smaller LUTs results in lower inference times due to improved cache utilization. Additionally, an observed average increase of approximately 2.1 times in time occurs when expanding the LUT bitwidth by two.

DNN	**Native CPU**	**Baseline Approx**	**AdaPT (w/ func)**	**AdaPT (w/ LUT)**	**AdaPT vs Baseline**
ResNet50	0.5 min	76.5 min	104 min	1.7 min	45×
DenseNet12	0.48 min	53.2 min	72 min	1.6 min	33.2×
VGG19	0.2 min	91.7 min	125 min	1.7 min	53.9×
Fashion-GAN	0.003 min	0.02 min	1.1 min	0.012 min	1.7×
VAE-MNIST	0.015 min	0.1 min	1.2 min	0.02 min	5×
LSTM-IMDB	1.36 min	48.5 min	449 min	7.6 min	6.4×
Inceptionv3	22.1 min	2909 min	4560 min	83 min	35.1×
SqueezeNet	11.6 min	443 min	576 min	20.6 min	21.5×
ShuffleNet	11.4 min	163 min	251 min	22.4 min	7.3×

Table 4-3. Inference emulation time in AdaPT for various DNNs

Specifically, in Table 4-3, we compare AdaPT with PyTorch's *native* FP32 optimized implementation, the *baseline* unoptimized approximate simulation (which utilizes LUTs but excludes our optimizations), and the functional C-implementation of the ACU (`mul8s_1L2H`). It is evident that the computation time for AdaPT has been significantly reduced in comparison to the baseline approach. When benchmarked against the state-of-the-art, AdaPT outperforms ALWANN [68], which also operates on a Xeon CPU, with a notable margin (1.7 minutes vs. 54.5 minutes on ResNet50). TypeCNN [141], running on a CPU and employing a custom C++ framework, does not provide inference results. Moving on to ProxSim [143], despite running on a GPU, AdaPT demonstrates very similar execution times (20.6 minutes vs. 17.5 minutes on SqueezeNet). TFApprox [70] exhibits faster execution on a GPU with ResNet50 (1.7 minutes vs. 0.26 minutes), but it's essential to note that the authors exclusively assess ResNets for 8-bit inference in image recognition.

AdaPT's strength lies in its diverse features, supporting various model architectures, application domains, and approximation techniques, along with approximation-aware retraining. This versatility contributes to the creation of a robust framework. A comprehensive comparison of AdaPT's functionalities with the state-of-the-art is presented in Table 4-4.

Tool Support	**AdaPT**	**[70]**	**[143]**	**[68]**	**[141]**
Framework	PyTorch	TF[1]	TF	TF	C++
Backend	CPU	GPU	GPU	CPU	CPU
Varying DNN types[2]	✓	✗	✗	✗	✗
Arbitrary ACU	✓	✗	✗	✗	✓
Quantization calibration	✓	✗	✗	✓	✗
Approximate-aware retraining	✓	✗	✓	✓	✓

[1]TF: Tensorflow. [2]For example: CNN, LSTM, GAN, etc.

Table 4-4. Qualitative comparison of AdaPT with state-of-the-art

To conclude, AdaPT is an end-to-end framework for fast cross-layer evaluation and re-training of approximate DNNs based on the popular PyTorch library. We showed how AdaPT simplifies and accelerates the process of DNN simulation using multi-threading and vectorization while at the same time it can support a wide range of DNN topologies and paved the way to new approximate DNN accelerators first time for PyTorch.

4.5.2 Experiments with TransAxx Framework

In this section, we conduct experiments to evaluate the performance of the TransAxx framework, focusing on the accuracy and execution time of popular Vision Transformer (ViT) models across various approximate multipliers. In terms of software versions, TransAxx was developed on PyTorch 1.13 with CUDA Version 11.7. The hardware setup employed for the experiments consisted of the same CPU as AdaPT's experiments, a 20-core Intel Xeon Gold 5218R server with 64GB of RAM, along with an Nvidia Tesla V100 GPU as the hardware accelerator of TransAxx.

Our framework prioritizes speed while maintaining flexibility, allowing users to efficiently test their custom Approximate Computing Units (ACUs). As mentioned earlier, two emulation methods are supported: the LUT-based and the functional-based approaches. In Table 4-5, we present inference times using both approaches and the retraining time for an epoch for each model. Our experiments involve four popular Vision Transformer models — ViT [168], DeiT [169], Swin [170], and GCViT [171] — on the ImageNet2012 dataset with batch sizes of 128 or 64. For retraining, we employ the Adam optimizer with a learning rate of $5e^{-5}$ for 2.5% of the ImageNet train dataset. In Table 4-5 the 8-bit `mul8s_1KV9w` [137] is used, but the execution time remains similar for any ACU of the same size. The LUT-based approach is generally unaffected by the type of ACU and is notably faster than the functional-based approach. The latter serves as a backup method in TransAxx to address unforeseen memory issues.

DNN	FLOPs	Params	Inference (w/ func.)	Inference (w/ LUT)	Retraining (w/ LUT)
ViT-S	4.2G	22.1M	121 min	6 min	5.5 min
DeiT-S	4.2G	22.1M	122 min	6.1 min	5.8 min
Swin-S	8.5G	49.6M	242 min	13.1 min	13 min
GCViT-XXT	1.9G	12M	43.5 min	3 min	3.5 min

Table 4-5. Emulation time in TransAxx for different ViTs

We further investigated the performance of LUT-based multiplication in our study, as depicted in Figure 4-9. On the left side of the figure, we can observe the inference time for each ViT model using different bitwidths of the LUT. As it is evident, the emulatiom time increases as the memory requirements increase because it takes more time to fetch the LUT from the GPU memory. Also, larger LUTs may not fit entirely into cache, leading to increased cache misses and longer memory access times. In contrast, smaller LUTs are more likely to fit into cache, resulting in better cache utilization and lower memory access latency. Now, on the right side of the figure, we show the caching effect on the LUT performance across the first batches of the inference.

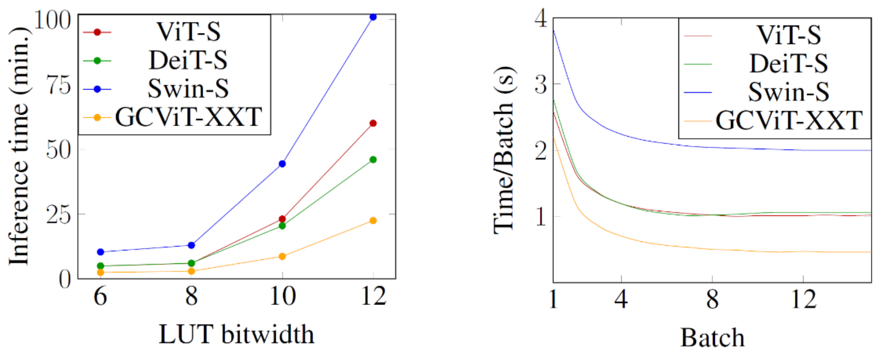


Figure 4-9. LUT-based multiplication performance. Left: LUT bitwidth impact on inference emulation time. Right: Caching effect on LUT performance during the first batches of inference.

As processing progresses through subsequent batches, the LUT caching mechanisms we introduced within TransAxx effectively come into play. These mechanisms allow for faster access to frequently used data, such as the LUT memory in our case, thereby reducing the computation time. As aforementioned, for > 12 bits where LUT memory might increase substantially TransAxx can always subsitute the LUT-based with functional based approach. We report that for a 12-bit multiplier `mul12s_2PP` from [137]) the inference emulation time using its C functional description can be ~5x slower than the LUT-based approach. Generally, the LUT-based execution times are deemed satisfactory and sufficiently fast, especially when considering that there is no other alternative for approximate ViT emulation. It is important to note that these performance metrics pertain to the complex and large ImageNet dataset. When compared to other frameworks for approximate simulation and taking into account the utilization of the large ImageNet dataset and complex ViT architectures, TransAxx proves to be faster. Additionally, TransAxx outperforms similar frameworks that lack support for ViT models.To facilitate a comparison with previous research, our focus can be directed towards the execution time of ViT-S, which closely aligns with ResNet50 in terms of FLOPs (4.2G vs. 3.87G). TransAxx demonstrates quicker inference times when contrasted with other GPU simulation frameworks such as ProxSim [143] (6min vs. 107min) and ApproxTrain [154] (6min vs. 10.4min).

Tool Support	**TransAxx**	**[147]**	**[154]**	**[70]**	**[143]**	**[68]**	**[141]**
Framework	PyTorch	PyTorch	TF	TF	TF	TF	C++
Backend	GPU	CPU	GPU	GPU	GPU	CPU	CPU
ViT model support	✓	✗	✗	✗	✗	✗	✗
Automatic layer swapping	✓	✗	✗	✗	✗	✗	✗
Quantization calibration	✓	✓	✗	✗	✗	✓	✗
HW-aware retraining	✓	✓	✓	✓	✓	✗	✓
Design space search	✓	✗	✗	✗	✗	✓	✗

Table 4-6. Qualitative comparison of TransAxx with state-of-the-art

We also perform a systematic analysis on the top-1 accuracy achieved on ImageNet-1K dataset for each of the four aforementioned ViT models on the default FP32, 8-bit quantized, approximate and retrained versions, as seen in Table 4-7. Apart from performing successful finetuning after approximation, we considered significantly the quantization part, as also described in previous paragraphs. In particular, prior to the approximation of the models, we performed a calibrated quantization scheme since finding a scale parameter correctly is known to have a large impact on the network's performance [172]. Regarding the approximate multipliers used for the experiments, we chose three distinct ACUs with different Mean Relative Error (MRE) and power characteristics obtained from the open-source EvoApprox [137], and an accurate ACU which corresponds to `mul8s_1KV6`.

Model specifications				ACU 1: **`mul8s_1KV9`** MRE 0.90%, power: 0.410mW			ACU 2: **`mul8s_1L2H`** MRE 4.41%, power: 0.301mW			ACU 3: **`mul8s_1L2L`** MRE 12.26%, power: 0.200mW		
Name	MACs approx.	FP32	8bit (calib.)	Initial	Retrained	Power ↓	Initial	Retrained	Power ↓	Initial	Retrained	Power ↓
ViT-S	98.54	74.64	71.86	34.95	67.31	3.45	1.264	66.74	28.75	0.090	0.15	52.18
DeiT-S	98.54	81.34	79.34	0.96	70.16	3.45	0.10	67.01	28.75	0.10	0.11	52.18
Swin-S	99.7	82.89	81.83	79.56	79.25	3.49	64.30	76.64	29.09	0.41	67.87	52.79
GCViT	75.5	79.72	78.91	73.50	78.346	2.64	51.56	76.93	22.03	0.26	63.01	39.98

Table 4-7. Accuracy and power benchmark [%] per multiplier and model

Table 4-7 summarizes our accuracy results obtained before and after retraining. Generally, we see that approximate-aware retraining reduces the accuracy gap successfully on the majority of approximate models, as the weights of the network can adapt to the distributions the ACUs represent. Additionally, we report the total MAC (multiply-and-accumulate) power reduction as it will be important for experiments of the design space exploration in the next paragraph. Clearly the actual power reduction would be influenced by numerous factors but the reduction from MAC operations usually has a cascading effect on the total consumption. The reduction percentage is in relation to the total MACs approximated in each model in order to have more precise measurements. The baseline is the power consumption of the accurate `mul8s_1KV6` multiplier (0.425mW).

4.5.3 Explored Design Space

The manual and simple method of performing approximation, that is to apply the same multiplier to all layers in the model gave us substantial power gains with the cost of some accuracy drop as seen from Table 4-7. However, in some cases, it is not possible to recover the large impact that approximate multiplications had on accuracy. Moreover, in many cases there might be a more power efficient solution that achieves similar accuracy. In this subsection, we conduct a comprehensive analysis of our automated search algorithm based on MCTS. The results of our automated search reveal a nuanced understanding of the trade-offs inherent in the power-accuracy space. We show that we can automatically find better solutions that are closer to the Pareto-front and thus offer better trade-off between power consumption and accuracy. For our experiment the four ACUs (`mul8s_1KV9, mul8s_1L2H, mul8s_1L2L, mul8s_1KV6`) are considered as possible candidates for each layer of every target model.

Policy Evaluation — Before proceeding with the Monte Carlo simulations it is essential to exhibit the stability of the algorithm and its ability to converge after some iterations. Naturally, our agent's performance is optimal when the rollout policy we used to estimate the expected rewards of each possible action is more likely to choose the best action. This is the reason we injected knowledge from the hardware configurations into the system using UCB1 formula so as to guide the search process towards the Pareto-optimal points. In Figure 4-10, for the case of ViT-S model, we measure the normalized reward on each simulation of the four possible starting paths/actions from the root node of the MCTS tree. In this way, we demonstrate the ability of our custom hardware-driven policy to converge to more stable rewards faster than the random policy.

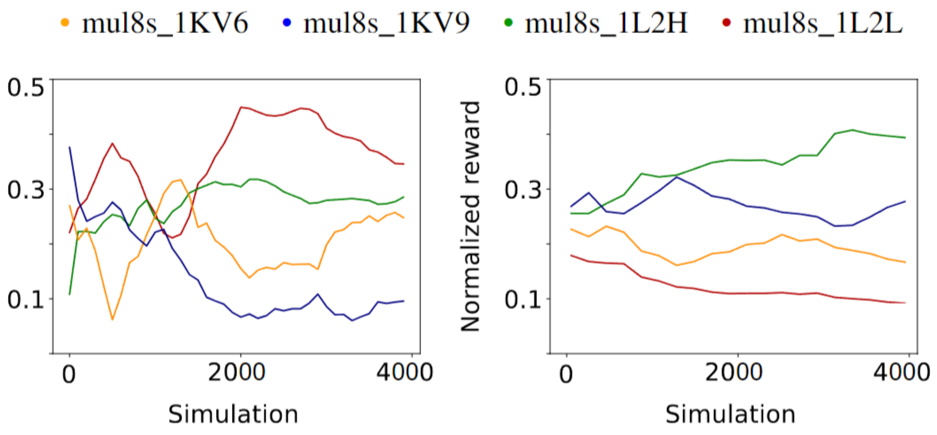


Figure 4-10. Rewards of each action from the MCTS root node using the random policy (left) and hw-driven policy (right).

Another valuable insight we can obtain from this figure is that the hardware-driven policy manages to find faster what might be a good or bad action to take according to the algorithm. For example, it prefers choosing `mul8s_1L2H` multiplier at the first layer as it might give an "appealing" accuracy-power tradeoff which is also indicated by our measurements in Table 4-7. In contrary, the lower power `mul8s_1L2L` multiplier is not often preferred as it significantly compromises the accuracy. We should note however, that intuitively these outcomes would vary across models, layers or multipliers.

Additionally, to demonstrate the convergence of our hw-driven MCTS-based search, we plot the reward values and their rolling mean (with a window of 50) targeting ViT-S model, over multiple simulations, as shown in Figure 4-11. As more simulations are performed, the MCTS tree is refined, and the algorithm converges towards optimal decisions or solutions. With each simulation, the search algorithm converges towards the approximate configurations that yield the best performance for the ViT model in terms of accuracy/power. The average reward stabilizes or reaches a plateau as the number of simulations increases. This means that our algorithm has explored the search space sufficiently enough and has found a solution or set of solutions that consistently achieve a certain level of performance.

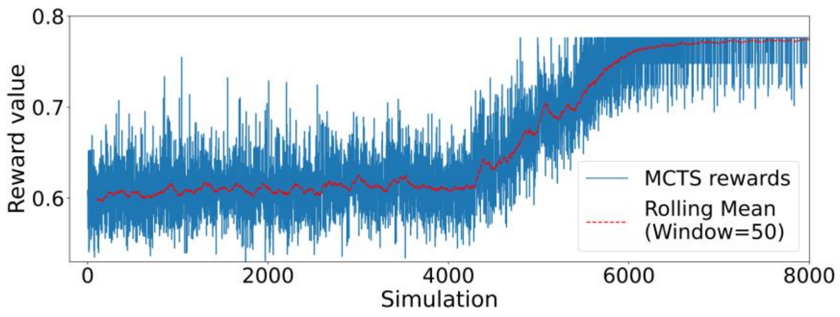


Figure 4-11. Convergence of the MCTS rewards.

MCTS simulations — Finally, we evaluate the use of our hardware-driven MCTS algorithms towards finding the Pareto-optimal curve for accuracy and power. In Figure 4-12, we illustrate the scatter plots from MCTS for every target ViT model using 2000 simulations (power consumption is normalized). On average, the exploration time on most models for this setup was about 35mins which is considered very acceptable, especially when compared with previous work ([152, 68]). The user has the flexibility to tune it according to their requirements; however, further exploration yielded minimal results on the accuracy-power trade-off. Each acquired Pareto (in red) represents the knowledge learned by the system towards finding the optimal multiplier configuration and it's basically the output of the search algorithm. To provide further comparisons, we experiment for two distinct λ parameters for the power bias, as described in Algorithm 1 and MCTS policy equation. For a state in the tree to be evaluated accurately, it must be visited a sufficient number of times (to gain the confidence about the statistics). Thus, the number of simulations clearly affects the quality of solutions, but we saw that around 2000 simulations were enough for most models to derive successful results in a reasonable time. Additionally, experimenting with different λ parameters or even exploration-exploitation ratios could potentially yield marginally improved results; however, our experiments served as a strong indication of the successful application of the proposed search algorithm and parameters.

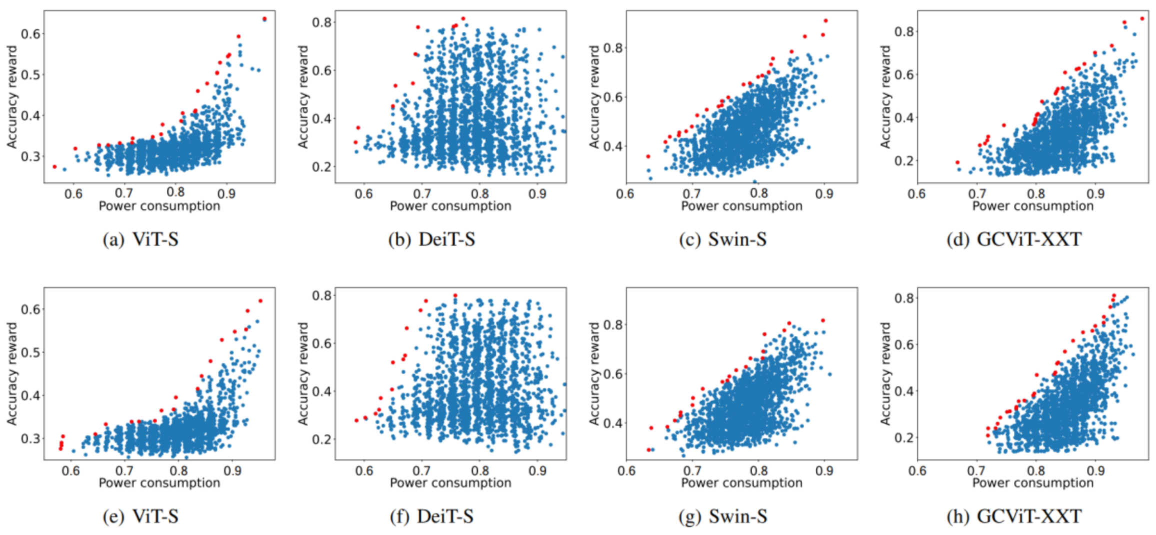


Figure 4-12. Scatter plots of hw-driven MCTS using 2000 simulations for λ=1.5 (top) and λ=0.5 (bottom) parameter.

In Figure 4-13, we summarize the solutions found with the baseline approximation of the models (in yellow), obtained from Table 4-7, along with the proposed optimal solutions obtained by our MCTS-based approach (in green). As baseline solutions we define the accuracy/power data points from Table 4-7 in which approximation is uniformly applied to all layers of the model without much customization. The corresponding scatter plots of Figure 4-13 illustrate the practical observation of the accuracy/power trade-off, with normalized power consumption along the x-axis and actual measured accuracy along the y axis. Notably, the study highlights the superior performance of our hardware-driven Monte Carlo tree search algorithm, particularly in achieving a more advantageous balance between power consumption and accuracy, as well as giving an increased flexibility from the designer's perspective in choosing a wider range of approximate solutions.

The green data points of Figure 4-13 are derived from the large exploration of the parameter space using MCTS, depicted in Figure 4-12, focusing only on the optimal solutions achieved through this algorithm (red points in Figure 4-12). These data points are evaluated on the whole ImageNet validation dataset and the Pareto points are depicted in the corresponding scatter plots of
Figure 4-13 as green triangular markers. The baseline solutions from Table 4-7, for each approximate multiplier, are deliberately included to provide a benchmark for evaluating the effectiveness of the proposed solutions using the MCTS search algorithm.

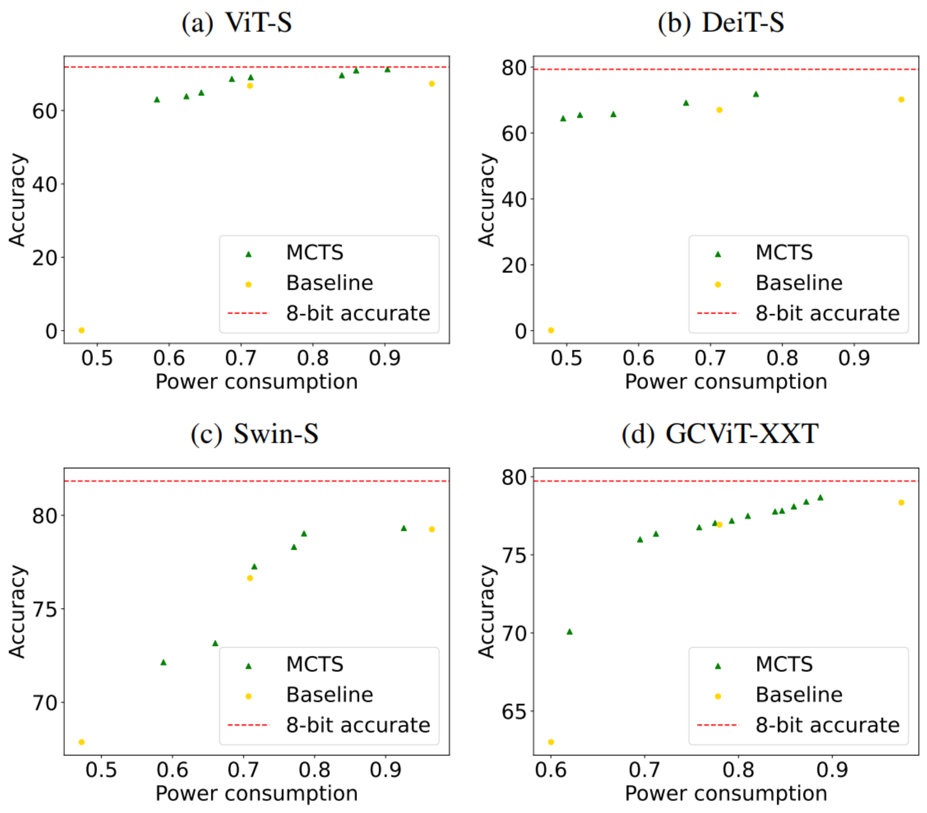


Figure 4-13. MCTS-optimal solutions based on real accuracy (green) and baseline approximate solutions (yellow).

The scatter plots of
Figure 4-13 visually highlight the trade-offs between power consumption and accuracy. Our solutions (green points) mainly form the Pareto-front in each subplot (i.e., deliver the best possible compromise between the two objectives, accuracy and power). For similar accuracy as the baseline approximate solutions (within ~1% maximum difference), MCTS delivers on average solutions with ~21% lower power. Additionally, our approach yields a considerable number of Pareto solutions in all models, providing the designer with a fine-grained set of choices. These choices aim to deliver the most effective balance between the two objectives, power and accuracy, while minimizing exploration time. In contrast, the baseline approximation offers a more limited selection.

In general, our MCTS search enables obtaining results from the Pareto-curve customized to fulfill the designer's requirements and eventually give a more fine-grained trade-off between accuracy and power with respect to the baseline approximation. Our requirements, relevant to the specific application, included constraints on power consumption, accuracy thresholds and exploration time of MCTS algorithm, but additional criteria can be involved for the system under consideration. Note that, to the

best of our knowledge, our work is the first to analyze the impact of approximate multipliers on ViT models and deliver a framework for a) evaluating the inference accuracy with reasonable speed, b) performing approximation-aware ViT re-training, and c) delivering a fine-grained accuracy-power trade-off when exploring the design space of ViT to approximation mapping. Despite the considerable savings reported in
Figure 4-13, higher power savings for similar accuracy loss have often been reported in approximate CNNs [136]. Hence, additional research is required for ViT models and/or dedicated approximate multipliers/approximation techniques are needed. Our work lays the groundwork for exploring this challenging domain and greatly facilitates designers in identifying solutions fast enough that align more closely with the desired balance of power and accuracy in ViT models.

Conclusion

This Ph.D. thesis has investigated the intricate field of efficient computing for Deep Learning (DL), with a primary focus on optimizing deep neural network accelerators for custom hardware. Through a comprehensive exploration of various software and hardware optimization strategies this research has contributed to our understanding of the underlying hardware mechanisms towards optimal AI model execution. Additionally, this thesis explored the use of approximate computing to exploit the capabilities of such hardware in inference speed and energy efficiency of DNNs. By proposing two custom emulation frameworks, AdaPT and TransAxx, it addresses the challenge of approximate DNN simulation. Specifically, it enables researchers and practitioners to rapidly prototype and evaluate approximate DNNs, fostering a more accessible and iterative approach to model development. Last, this thesis proposed an MCTS (Monte Carlo Tree Search) based method for rapid design space exploration towards find an optimal tradeoff between power consumption and accuracy in approximate DNNs, showing its efficacy on vision transformer models. As the field of deep learning continues to evolve, the findings of this thesis contribute to the ongoing discussion on efficient computing for AI inference. The insights gained from optimizing neural network inference on custom hardware, coupled with the emulation framework for approximate computing, signify a substantial step towards bridging the gap between software and hardware in the field of DL.

5.1 Summary of findings

Below is the summary of the main findings and contributions:

- *Optimization of novel AI applications*: Three distinct AI tasks were optimized for reconfigurable hardware. First, the popular FAISS (Facebook AI Similarity Search) framework [70] was optimized for FPGAs in order to accelerate the algorithm of similarity search. Second, a Generative Adversarial Network (GAN) was developed and deployed on an FPGA SoC with the aim to restore images of clothing. Third, towards combating the Covid-19 pandemic, a CNN was developed and accelerated for reconfigurable hardware with the ability to categorize chest X-Ray images into three classes: Covid-19, Viral Pneumonia, and Normal.

- *Automatic FPGA firmware from CNNs*: We proposed an end-to-end framework, based on HLS4ML [56]. Through this tool we convert trained CNN models into optimized FPGA firmware tailored for cloud FPGA architectures.
- *AdaPT Framework:* We created the AdaPT framework as a rapid DNN approximation emulation tool, presented as a PyTorch plugin. The framework leverages CPU acceleration and seamlessly supports a wide range of layers and model architectures along with post-training quantization and approximate-aware retraining.
- *TransAxx Framework:* We created TransAxx with the aim to emulate approximate Vision Transformers. It accommodates all the major features of AdaPT such as post-training quantization and approximate-aware retraining. Also, it has a more streamlined and seamless design and leverages GPU acceleration for the emulation of approximate AI models.
- *MCTS-based Design Space Exploration:* We proposed a novel method based on Monte Carlo Tree Search (MCTS) for a hw-driven automated search that can find near-optimal trade-offs in the power-accuracy space. With this method, we demonstrate that we can automatically find better solutions fast, which are closer to the Pareto front than the simple approximation method. In the latter, the approximation is uniformly applied to all layers of the model without much customization, while in our solution we achieve a better trade-off between power consumption and accuracy using mixed approximation between layers.

5.2 Future Directions

The research presented demonstrates that custom hardware for Deep Learning holds great promise in enhancing performance and energy efficiency. Nevertheless, with the rapid growth and expansion of next-generation AI models and the continuous emergence of new hardware architectures, new innovative challenges always arise that need attention in future applications. Given the rapid growth and expansion of next-generation AI such as *generative AI* and the emergence of new requirements from both technical and societal perspectives, there remain open and novel issues to be addressed in future efforts. Major technology companies, European networks of researchers, and Europe itself have already paved the way for the design and implementation of future computing systems. Recent years have witnessed a surge in the demand for computing systems capable of supporting large language models and other extensive AI applications. This has brought hardware accelerators to the forefront as promising solutions for optimizing resource efficiency and performance in datacenter and edge infrastructures. Future research will focus on enhancing the scalability and adaptability of accelerators to accommodate the evolving complexity of AI models, while also innovating new AI algorithms that can understand, interpret, and generate complex, context-aware responses, mimicking humans.

Future work should focus on next-generation computing systems and AI, specifically on several key technical areas to advance their capabilities and address emerging challenges:

1. *Scalability and Adaptability of Accelerators:*
   - Develop FPGA-based AI accelerators that can dynamically reconfigure to support various AI workloads, ensuring efficient resource utilization and performance optimization.
   - Enhance the scalability of these accelerators to handle the growing size and complexity of AI models, including large language models and deep neural networks.

2. *Energy-Efficient Computing:*
   - Implement AI/ML-based resource management policies that optimize energy consumption while maintaining system performance and reliability.
   - Integrate renewable energy sources with traditional power supplies, using AI to predict and manage energy demand, ensuring a sustainable and cost-effective energy mix.

3. *Approximate Computing:*
   - Explore approximate computing techniques for new novel architectures or material that allow for controlled trade-offs between computational accuracy and efficiency, reducing power consumption and processing time without significantly impacting the quality of results.
   - Develop algorithms that can leverage these techniques for various AI applications, such as natural language processing or generative AI.

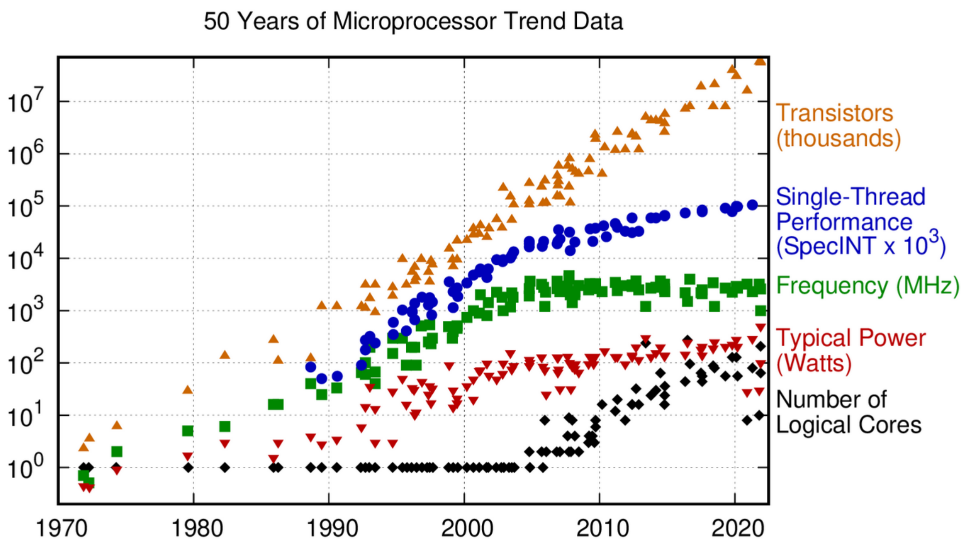
4. *Edge and Cloud Integration:*
   - Design systems that seamlessly integrate edge computing with cloud infrastructures, leveraging the strengths of both to provide low-latency, high-performance computing for AI applications.
   - Focus on the development of distributed AI models that can operate efficiently across these integrated environments.

Εκτεταμένη Περίληψη στα Ελληνικά

Το κίνητρο αυτής της έρευνας προήλθε από την αυξανόμενη ζήτηση για αποτελεσματική επεξεργασία τεχνητής νοημοσύνης (AI). Καθώς η χρήση τεχνολογιών τεχνητής νοημοσύνης συνεχίζει να αυξάνεται σε ένα ευρύ φάσμα βιομηχανιών, έχει γίνει όλο και πιο σημαντικό να αναπτυχθούν λύσεις υλικού και λογισμικού που μπορούν να χειριστούν τις υπολογιστικές απαιτήσεις αυτών των εφαρμογών. Αυτό οδήγησε σε αυξανόμενο ενδιαφέρον για τη χρήση επιταχυντών υλικού, όπως μονάδες επεξεργασίας γραφικών (GPU) και συστοιχία επιτόπια προγραμματιζόμενων πυλών (FPGA), οι οποίες μπορούν να βελτιώσουν σημαντικά την ταχύτητα και την ενεργειακή απόδοση της επεξεργασίας AI.
Για να αποκτήσουμε μια βαθύτερη κατανόηση του κινήτρου πίσω από αυτό το έργο, είναι σημαντικό να εντοπίσουμε την προέλευση και τα θεμελιώδη στοιχεία του. Τις τελευταίες έξι δεκαετίες, ο νόμος του Μουρ έπαιξε καθοριστικό ρόλο στην οδήγηση της πορείας των υπολογιστών. Κατά τη διάρκεια αυτής της εκτεταμένης περιόδου, η σταθερή έμφαση της βιομηχανίας στην μείωση του μεγέθους των τρανζίστορ, μια βασική πτυχή του νόμου του Moore, έχει σταθερά αποφέρει αυξημένη απόδοση και πυκνότητα τρανζίστορ. Αν και μπορεί να είναι πρόωρο να δηλωθεί οριστικά η κατάρρευση του νόμου του Moore, υπάρχουν ενδείξεις που υποδηλώνουν ότι έχουμε αντιμετωπίσει τους φυσικούς περιορισμούς που βασίζονται σε γενικού σκοπού μονάδες επεξεργασίες (πχ. CPU) που βασίζονται σε πυρίτιο (Κεντρικές Μονάδες Επεξεργασίας).

> *"Moore's law is the observation that the number of transistors in a dense integrated circuit doubles approximately every two years. The period is often quoted as 18 months because of Intel executive David House, who predicted that chip performance would double every 18 months. — G. E. Moore, 1965*

Όπως παρατηρείται από το παρακάτω σχήμα, η εκθετική αύξηση της υπολογιστικής ισχύος που προβλέπεται από το νόμο του Moore επιβραδύνεται. Αυτή η τάση αποτελεί πρόκληση για τη συνεχή πρόοδο της τεχνητής νοημοσύνης, καθώς η αυξημένη υπολογιστική ισχύς είναι ζωτικής σημασίας για τον χειρισμό των πολύπλοκων υπολογισμών και των τεράστιων συνόλων δεδομένων, ειδικά που εμπλέκονται σε εργασίες Βαθιάς Μάθησης (Deep Learning). Η παραδοσιακή προσέγγιση της βασιζόμενης αποκλειστικά σε CPU γενικής χρήσης για φόρτους εργασίας τεχνητής νοημοσύνης γίνεται όλο και πιο αναποτελεσματική και μη βιώσιμη. Οι CPU, σχεδιασμένες για ένα ευρύ φάσμα εργασιών, δεν είναι βελτιστοποιημένες για τις συγκεκριμένες υπολογιστικές απαιτήσεις των αλγορίθμων τεχνητής νοημοσύνης, οδηγώντας σε μη βέλτιστη απόδοση και υψηλή κατανάλωση ενέργειας.



Σχήμα 1. Δεδομένα τάσης μικροεπεξεργαστή 50 ετών [1].

Είναι προφανές ότι οι CPU δεν μπορούν να συμβαδίσουν με το ρυθμό των υπολογιστικών απαιτήσεων της τεχνητής νοημοσύνης και ιδιαίτερα του Deep Learning. Αναπτύσσονται εναλλακτικές αρχιτεκτονικές υπολογιστών και εξειδικευμένοι επεξεργαστές για την τεχνητή νοημοσύνη, διασφαλίζοντας την αδιάλειπτη πρόοδο της έρευνας για την τεχνητή νοημοσύνη. Παρά την ενσωμάτωση επιταχυντών τεχνητής νοημοσύνης για τη βελτίωση της υπολογιστικής απόδοσης, παραμένει πάντα μια αξιοσημείωτη ανησυχία σχετικά με την αυξημένη κατανάλωση ενέργειας. Η ζήτηση για ουσιαστική ισχύ επεξεργασίας σε εργασίες βαθιάς μάθησης, σε συνδυασμό με την αυξανόμενη πολυπλοκότητα των νευρωνικών δικτύων, έχει συμβάλει σε αυξημένες ενεργειακές απαιτήσεις, αποτελώντας πρόκληση για την επίτευξη βέλτιστης απόδοσης ισχύος ακόμη και με την ανάπτυξη εξειδικευμένων επιταχυντών τεχνητής νοημοσύνης.

Consumption	**CO2e (lbs)**
Air travel, 1 passenger, NY↔SF	1984
Human life, avg, 1 year	11.023
American life, avg, 1 year	36.156
Car, avg incl. fuel, 1 lifetime	126.000
Transformer (big) w/ neural architecture search	626.155

Πίνακας 1. Εκτιμώμενες εκπομπές CO2 από την εκπαίδευση LLM, σε σύγκριση με άλλες δραστηριότητες με γνωστή κατανάλωση.

Στον παραπάνω πίνακα παρουσιάζουμε τις εκτιμώμενες εκπομπές CO2 από την εκπαίδευση μεγάλων γλωσσικών μοντέλων (LLM) σε επιταχυντές GPU. Η κατανάλωση συγκρίνεται με κοινές ανθρώπινες δραστηριότητες υπογραμμίζοντας τις ακραίες απαιτήσεις ισχύος που απαιτούνται για την επεξεργασία τεχνητής νοημοσύνης. Μία από τις βασικές προκλήσεις στην επεξεργασία της τεχνητής νοημοσύνης είναι η ανάπτυξη αλγορίθμων που μπορούν να λειτουργούν σε γρήγορο και αποδοτικό υλικό, ειδικά για πολλούς αλγόριθμους υπολογιστικής όρασης όπου η καθυστέρηση απόκρισης και η κατανάλωση ενέργειας είναι κρίσιμης σημασίας. Τα βαθιά νευρωνικά δίκτυα (DNN) για παράδειγμα έχουν αναδειχθεί ως δημοφιλής επιλογή για τους βασικούς αλγόριθμους πολλών εργασιών όρασης υπολογιστή λόγω της ικανότητάς τους να μαθαίνουν πολύπλοκα χαρακτηριστικά από ακατέργαστα δεδομένα εικόνας. Αυτό οδήγησε σε ένα αυξανόμενο ενδιαφέρον για τη χρήση επιταχυντών υλικού, όπως GPU και FPGA, για την επιτάχυνση των υπολογισμών αυτών των αλγορίθμων και τη βελτίωση της συνολικής απόδοσης. Ο προγραμματισμός τέτοιων συσκευών και η ανάπτυξη λύσεων λογισμικού-υλισμικού που μπορούν να χειριστούν τις υπολογιστικές απαιτήσεις αυτών των αλγορίθμων σε πραγματικό χρόνο δεν είναι εύκολη διαδικασία και συχνά απαιτεί μεγάλη προσπάθεια προγραμματισμού από την πλευρά του μηχανικού.

Επιπλέον, η τρέχουσα τελευταία λέξη της τεχνολογίας χρησιμοποιεί κατά προσέγγιση πολλαπλασιαστές (approximate multipliers) για την αντιμετώπιση των εξαιρετικά αυξημένων απαιτήσεων ισχύος των επιταχυντών DNN. Ο κατά προσέγγιση υπολογισμός αναφέρεται στην ιδέα της θυσίας της ακρίβειας του υπολογισμού υπέρ της αποδοτικότητας, συχνά μέσω της χρήσης μειωμένης ακρίβειας ή απλοποιημένων λειτουργιών. Οι κατά προσέγγιση πολλαπλασιαστές μπορούν να μειώσουν σημαντικά την υπολογιστική πολυπλοκότητα των μοντέλων τεχνητής νοημοσύνης, καθιστώντας τα πιο αποτελεσματικά και πρακτικά για εφαρμογές του πραγματικού κόσμου. Η αξιολόγηση της ακρίβειας των κατά προσέγγιση DNN αποδεικνύεται πρόκληση λόγω της απουσίας αποκλειστικού κατά προσέγγιση υλικού. Η κατανόηση του τρόπου με τον οποίο συμπεριφέρονται αυτά τα DNN σε τέτοιο υλικό είναι ζωτικής σημασίας πριν από την κατασκευή του υλικού, καθιστώντας το απαραίτητη προϋπόθεση για ακριβή αξιολόγηση. Όταν το υλικό δεν είναι διαθέσιμο, η μόνη εφικτή επιλογή είναι η προσομοίωση της αριθμητικής του κατά προσέγγιση πολλαπλασιαστή. Αυτό μπορεί να γίνει αξιοποιώντας ένα περιβάλλον βαθιάς μάθησης (πχ. PyTorch, Tensorflow) ικανό να υποστηρίξει αυτήν τη λειτουργικότητα, αλλά τα κοινά περιβάλλοντα DNN δεν διαθέτουν ενσωματωμένη υποστήριξη. Η εξομοίωση της συμπεριφοράς του κατά προσέγγιση πολλαπλασιαστή με χρήση αυτών των προγραμμάτων θα είναι δύσκολη με αποτέλεσμα παρατεταμένους χρόνους εκτέλεσης. Σε αντίθεση με τις βελτιστοποιημένες βιβλιοθήκες που είναι διαθέσιμες για τις τυπικές εκδόσεις των DNN, δεν υπάρχουν ισοδύναμα για την επιτάχυνση της διαδικασίας προσομοίωσης του κατά προσέγγιση πολλαπλασιαστή.

Επιπροσθέτως, για τη βελτιστοποίηση των κατά προσέγγιση υπολογισμών σε DNN, μια κρίσιμη διαδικασία περιλαμβάνει τον εντοπισμό του καταλληλότερου κατά προσέγγιση πολλαπλασιαστή για κάθε επίπεδο DNN, μια έννοια που αναφέρεται ως βελτιστοποίηση πολλαπλών επιπέδων. Αυτή η πτυχή είναι ιδιαίτερα κρίσιμη όταν ο στόχος είναι να μεγιστοποιηθούν τα κέρδη ισχύος με ταυτόχρονη τήρηση των περιορισμών σχετικά με την αποδεκτή απώλεια ακρίβειας. Ενώ πολυάριθμες μελέτες έχουν εξερευνήσει αυτοματοποιημένες μεθόδους για τον προσδιορισμό της βέλτιστης κβαντοποίησης ανά στρώμα σε κβαντισμένα DNN, το πεδίο των κατά προσέγγιση DNN έχει λάβει συγκριτικά λιγότερη προσοχή όσον αφορά την αυτοματοποιημένη ροή αναζήτησης. Με άλλα λόγια, υπάρχει ένα κενό στην υπάρχουσα έρευνα όσον αφορά τη συστηματική και αυτόματη βελτιστοποίηση της διαμόρφωσης των κατά προσέγγιση υπολογισμών σε ολόκληρο το δίκτυο. Επίσης, ο καθορισμός της βέλτιστης διαμόρφωσης των κατά προσέγγιση πολλαπλασιαστών μεταξύ κάθε επιπέδου ενός μοντέλου DNN προκειμένου να βρεθεί η καλύτερη αντιστάθμιση μεταξύ ακρίβειας και ισχύος μπορεί να προκαλέσει σημαντική υπολογιστική επιβάρυνση. Ο χώρος σχεδιασμού γίνεται μεγάλος και η μέτρηση της ακρίβειας κάθε διαφορετικής διαμόρφωσης δεν είναι πρακτική. Η αντιμετώπιση αυτών των προκλήσεων θα μπορούσε να ανοίξει το δρόμο για πιο ευρεία και πρακτική υιοθέτηση κατά προσέγγιση DNN, ιδιαίτερα σε εφαρμογές έντασης πόρων.

Ανασκόπηση ιστορικού υποβάθρου και βιβλιογραφίας

Σε αυτό το κεφάλαιο, παρέχουμε μια συνοπτική περίληψη της εργασίας και του ιστορικού υπόβαθρου που αποτελούν τη βάση της παρούσας διατριβής. Το κεφάλαιο παρέχει ένα λεπτομερές υπόβαθρο για την ερευνητική περιοχή, ρίχνοντας φως στην εξέλιξη της τεχνητής νοημοσύνης και των επιταχυντών υλικού στους οποίους βρίσκεται η παρούσα μελέτη. Επιπρόσθετα, προκειμένου να παρέχεται μια ολοκληρωμένη κατανόηση του αντικειμένου, έχει διεξαχθεί μια διεξοδική εξέταση της υπάρχουσας έρευνας, θεωριών και μελετών. Η ανασκόπηση της βιβλιογραφίας διερευνά τις βασικές έννοιες, θεωρίες και μεθοδολογίες που έχουν διαμορφώσει το τρέχον πεδίο, επισημαίνοντας τα κενά, τους περιορισμούς και τα άλυτα ερωτήματα που παρακινούν την τρέχουσα μελέτη. Παρουσιάζοντας ένα περιεκτικό υπόβαθρο και παρέχοντας σχετική βιβλιογραφία, αυτό το κεφάλαιο θέτει τις βάσεις για την επακόλουθη ανάλυση και τα ευρήματα που παρουσιάζονται στα επόμενα κεφάλαια.

Η όραση υπολογιστών περιλαμβάνει έναν κλάδο της Μηχανικής Μάθησης που είναι αφιερωμένος στην ανάλυση και κατανόηση εικόνων και βίντεο. Ο πρωταρχικός του στόχος είναι να επιτρέψει στους υπολογιστές να «βλέπουν» ερμηνεύοντας αποτελεσματικά τις οπτικές πληροφορίες. Εντός της όρασης υπολογιστή, τα μοντέλα έχουν σχεδιαστεί ειδικά για την αποκωδικοποίηση οπτικών δεδομένων εξάγοντας σχετικά χαρακτηριστικά και πληροφορίες που αποκτώνται κατά τη διάρκεια της εκπαιδευτικής διαδικασίας. Αυτή η ικανότητα δίνει τη δυνατότητα σε αυτά τα μοντέλα να κατανοούν εικόνες και βίντεο και να εφαρμόζουν αυτές τις ερμηνείες σε εργασίες που περιλαμβάνουν πρόβλεψη ή λήψη αποφάσεων. Ωστόσο, η όραση υπολογιστή βασίζεται σε μεγάλο βαθμό σε άφθονα δεδομένα για τις λειτουργίες της. Στις μέρες μας για να επιτευχθούν τέτοιες εργασίες, τα νευρωνικά δίκτυα που είναι ένας τύπος μοντέλου βαθιάς μάθησης χρειάζονται για να εκπαιδευτούν ώστε να αποκτήσουν γνώση και να ενισχύσουν την ακρίβειά τους μέσω επαναληπτικών διαδικασιών μάθησης. Μόλις τελειοποιηθούν αυτοί οι αλγόριθμοι για βέλτιστη ακρίβεια, γίνονται ισχυρά εργαλεία τεχνητής νοημοσύνης. Επιτρέπουν την ταχεία ταξινόμηση και ομαδοποίηση δεδομένων, ξεπερνώντας συχνά την αποτελεσματικότητα της μη αυτόματης αναγνώρισης από ειδικούς.

Τα νευρωνικά δίκτυα αποτελούν ένα πολύ σημαντικό πεδίο στη μηχανική μάθηση και την τεχνητή νοημοσύνη. Η ανάπτυξη τους έχει ρίζες σε διάφορες έννοιες και αλγορίθμους που εξελίσσονται με την πάροδο του χρόνου. Συνοπτική ακολουθεί η ιστορική τους εξέλιξη:

- 1940: Τεχνητοί Νευρώνες
- 1950: Τεχνητά Νευρωνικά Δίκτυα
- 1985: Οπισθοδιάδοση
- 2010: Η άνοδος της βαθιάς μάθησης
- 2014: Δημιουργικά ανταγωνιστικά δίκτυα
- 2017: Transformer model

Τα νευρωνικά δίκτυα έχουν διάφορα επίπεδα, τα οποία εξυπηρετούν διάφορους σκοπούς και λειτουργίες. Κάποια από τα βασικά επίπεδα που χρησιμοποιούνται σε νευρωνικά δίκτυα είναι τα εξής:

- *Επίπεδο Εισόδου (Input Layer):* Αυτό το επίπεδο λαμβάνει τα εισερχόμενα δεδομένα και προωθεί τις εισόδους στο επόμενο επίπεδο. Ο αριθμός των νευρώνων σε αυτό το επίπεδο αντιστοιχεί στον αριθμό των χαρακτηριστικών (features) των δεδομένων.

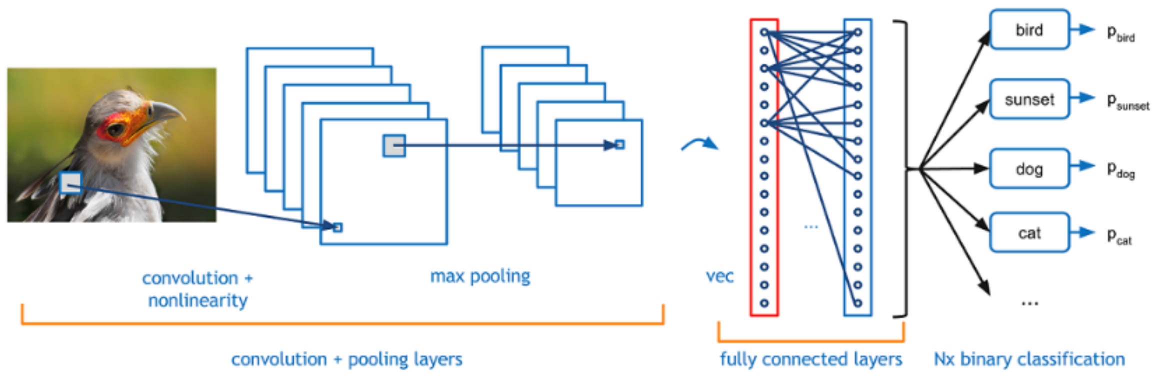
- *Κρυφό Επίπεδο (Hidden Layer):* Αυτά τα επίπεδα επεξεργάζονται τις εισόδους από το επίπεδο εισόδου. Ένα νευρωνικό δίκτυο με ένα κρυφό επίπεδο ονομάζεται δίκτυο με ένα κρυφό επίπεδο (single-layer perceptron), ενώ αν έχει περισσότερα από ένα κρυφά επίπεδα, το ονομάζουμε πολυεπίπεδο νευρωνικό δίκτυο (multilayer perceptron).

- *Επίπεδο Εξόδου (Output Layer):* Το επίπεδο αυτό παράγει την τελική έξοδο του δικτύου. Ο αριθμός των νευρώνων σε αυτό το επίπεδο εξαρτάται από τον τύπο του προβλήματος (π.χ., μία κλάση σε ένα πρόβλημα ταξινόμησης, ένας αριθμός σε ένα πρόβλημα παλινδρόμησης).

- *Συνελικτικά Επίπεδα (Convolutional Layers):* Χρησιμοποιούνται συνήθως σε συνελικτικά νευρωνικά δίκτυα (CNNs) για την εξαγωγή χαρακτηριστικών από εικόνες. Επιτρέπουν την αναγνώριση χαρακτηριστικών σε διάφορα μέρη της εικόνας.

- *Επίπεδα Κανονικοποίησης (Normalization Layers):* Χρησιμοποιούνται για την κανονικοποίηση των εξόδων των προηγούμενων επιπέδων και τη βελτίωση της σύγκλισης του μοντέλου.

Αυτά είναι μερικά από τα βασικά επίπεδα που χρησιμοποιούνται σε νευρωνικά δίκτυα, και τα δίκτυα συνήθως περιλαμβάνουν διαφορετικούς συνδυασμούς αυτών των επιπέδων ανάλογα με τον σκοπό τους και τη φύση των δεδομένων. Παρακάτω στην εικόνα φαίνεται πώς η είσοδος εισέρχεται σε εναλλασσόμενα επίπεδα συνέλιξης, συγκέντρωσης και άλλων για να εκτελεστεί μέσα από το νευρωνικό για αναγνώριση εικόνας. Πολλά κρυφά επίπεδα μπορούν να εμπλέκονται όπου τροφοδοτούνται δεδομένα. Αφού περάσει όλα τα επίπεδα, το δίκτυο παράγει ένα τελικό διάνυσμα με πιθανότητα P_i για κάθε κατηγορία κλάσης του μοντέλου μας (inference).



Σχήμα 2. Διαδικασία εκτέλεσης ενός νευρωνικού δικτύου για αναγνώριση εικόνας

Προκλήσεις στην εκτέλεση νευρωνικών — Τα νευρωνικά δίκτυα μπορούν να αντιμετωπίσουν περίπλοκα προβλήματα σε διάφορους τομείς, όπως η όραση υπολογιστή ή η επεξεργασία φυσικής γλώσσας, όπως αναφέραμε ήδη. Ωστόσο, αυτές οι προσεγγίσεις αντιμετωπίζουν προκλήσεις και περιορισμούς που εμποδίζουν την πλήρη δυνατότητα και την ευρεία εφαρμογή τους. Παρακάτω, συνοψίζουμε τις κύριες υπάρχουσες προκλήσεις και περιορισμούς που σχετίζονται με την εξαγωγή συμπερασμάτων νευρωνικών δικτύων.

1. Υπολογιστική πολυπλοκότητα: Τα νευρωνικά δίκτυα μπορεί να είναι υπολογιστικά δύσκολα, ειδικά τα βαθιά νευρωνικά δίκτυα με μεγάλο αριθμό επιπέδων και παραμέτρων. Η διαδικασία του inference απαιτεί την εκτέλεση εκτεταμένων λειτουργιών πολλαπλασιασμού πινάκων και αξιολογήσεων συναρτήσεων ενεργοποίησης, οι οποίες μπορεί να είναι χρονοβόρες και εντατικές σε πόρους, ιδιαίτερα σε συσκευές με περιορισμένη υπολογιστική ισχύ.
2. Απαιτήσεις Μνήμης: Τα νευρωνικά δίκτυα συχνά απαιτούν σημαντική μνήμη για την αποθήκευση των παραμέτρων του μοντέλου και τις ενδιάμεσες ενεργοποιήσεις κατά την εξαγωγή συμπερασμάτων. Αυτό μπορεί να είναι προβληματικό σε συσκευές με περιορισμένους πόρους, όπως κινητά τηλέφωνα ή ενσωματωμένα συστήματα, όπου η χωρητικότητα μνήμης είναι περιορισμένη.
3. Ενεργειακή απόδοση: Η εξαγωγή συμπερασμάτων σχετικά με συσκευές περιορισμένης ισχύος, όπως κινητές συσκευές ή συσκευές αιχμής, απαιτεί προσεκτική βελτιστοποίηση για την ενεργειακή απόδοση. Η λειτουργία πολύπλοκων νευρωνικών δικτύων μπορεί να εξαντλήσει γρήγορα την μπαταρία της συσκευής. Επομένως, η ελαχιστοποίηση της

κατανάλωσης ενέργειας της διαδικασίας συμπερασμάτων είναι ζωτικής σημασίας για την ανάπτυξη σε πραγματικό κόσμο.

4. Latency: Πολλές εφαρμογές απαιτούν προβλέψεις χαμηλής καθυστέρησης. Ωστόσο, η εξαγωγή συμπερασμάτων νευρωνικών δικτύων μπορεί να προκαλέσει καθυστερήσεις λόγω των σχετικών υπολογιστικών απαιτήσεων. Η μείωση του χρόνου εξαγωγής συμπερασμάτων για την επίτευξη σχεδόν στιγμιαίων προβλέψεων είναι μια σημαντική πρόκληση, ειδικά όταν έχουμε να κάνουμε με μεγάλα και πολύπλοκα μοντέλα.

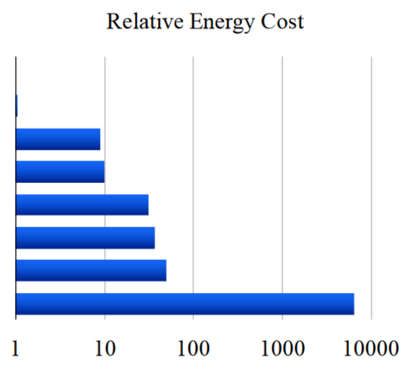
5. Μέγεθος μοντέλου: Τα βαθιά νευρωνικά δίκτυα μπορεί να είναι αρκετά μεγάλα όσον αφορά τον αριθμό των παραμέτρων που διαθέτουν. Αυτό δημιουργεί προκλήσεις όσον αφορά τις απαιτήσεις αποθήκευσης, μετάδοσης και μνήμης κατά την εξαγωγή συμπερασμάτων. Η μείωση του μεγέθους του μοντέλου χωρίς σημαντική απώλεια στην ακρίβεια είναι ένας τομέας έρευνας συνεχούς ενδιαφέροντος.

6. Ανάπτυξη στο άκρο: Η ανάπτυξη νευρωνικών δικτύων σε συσκευές στο άκρο, όπως smartphone, συσκευές IoT ή ενσωματωμένα συστήματα, παρουσιάζει μοναδικές προκλήσεις. Αυτές οι συσκευές έχουν συνήθως περιορισμένους υπολογιστικούς πόρους, περιορισμούς ισχύος και διακοπτόμενη συνδεσιμότητα. Η αποτελεσματική προσαρμογή των νευρωνικών δικτύων ώστε να λειτουργούν αποτελεσματικά κάτω από τέτοιους περιορισμούς είναι απαραίτητη για την ανάπτυξη αιχμής.

7. Απόρρητο και ασφάλεια: Τα νευρωνικά δίκτυα που εκπαιδεύονται σε ευαίσθητα δεδομένα μπορούν να εγείρουν ανησυχίες σχετικά με το απόρρητο και την ασφάλεια κατά τη διάρκεια της εξαγωγής συμπερασμάτων. Η προστασία του απορρήτου των δεδομένων χρήστη και η αποτροπή κακόβουλων επιθέσεων, όπως η κλοπή μοντέλων ή των βαρών τους, είναι σημαντικές προκλήσεις που πρέπει να αντιμετωπιστούν.

Η ανάγκη για ενεργειακή απόδοση και αποδοτικότητα: Καθώς το μέγεθος των μοντέλων τεχνητής νοημοσύνης αυξάνεται, αυξάνεται και ο αριθμός των απαιτούμενων λειτουργιών πρόσβασης στη μνήμη. Συγκριτικά, οι υπολογιστικές λειτουργίες όπως οι υπολογισμοί πίνακα και πίνακα-διανύσματος είναι σημαντικά πιο αποδοτικοί από πλευράς ενέργειας από τις λειτουργίες πρόσβασης στη μνήμη [16]. Όταν εξετάζουμε την κατανάλωση ενέργειας της πρόσβασης ανάγνωσης από τη μνήμη έναντι των πράξεων πρόσθεσης και πολλαπλασιασμού, γίνεται προφανές ότι η πρόσβαση στη μνήμη απαιτεί αρκετές τάξεις μεγέθους περισσότερη ενέργεια από τις πράξεις υπολογισμού. Λόγω της αδυναμίας των μεγάλων δικτύων να χωρέσουν σε αποθήκευση στο chip, η συχνότητα των ενεργοβόρων προσβάσεων DRAM αυξάνεται σημαντικά. Αντίθετα, οι επιταχυντές τεχνητής νοημοσύνης μπορούν να ενσωματώσουν συγκεκριμένα σχεδιαστικά στοιχεία που στοχεύουν στη μείωση της συχνότητας πρόσβασης στη μνήμη, στην παροχή μεγαλύτερης κρυφής μνήμης στο τσιπ και στην ενσωμάτωση αποκλειστικών χαρακτηριστικών υλικού για τη βελτίωση των υπολογισμών πίνακα-πίνακα. Λόγω του ότι είναι ειδικά κατασκευασμένες συσκευές, οι επιταχυντές τεχνητής νοημοσύνης είναι πιο συγκεκριμένοι στους αλγόριθμους που εκτελούν, επιτρέποντάς τους να αξιοποιούν τις αποκλειστικές λειτουργίες τους πιο αποτελεσματικά σε σύγκριση με τους επεξεργαστές γενικής χρήσης.

Operation	Energy [pJ]	Relative Cost
32 bit int ADD	0.1	1
32 bit float ADD	0.9	9
32 bit Register File	1	10
32 bit int MULT	3.1	31
32 bit float MULT	3.7	37
32 bit SRAM Cache	5	50
32 bit DRAM Memory	**640**	**6400**



Σχήμα 3. Μετρήσεις ενέργειας για τεχνολογία 45nm CMOS ανά πράξη. [16]

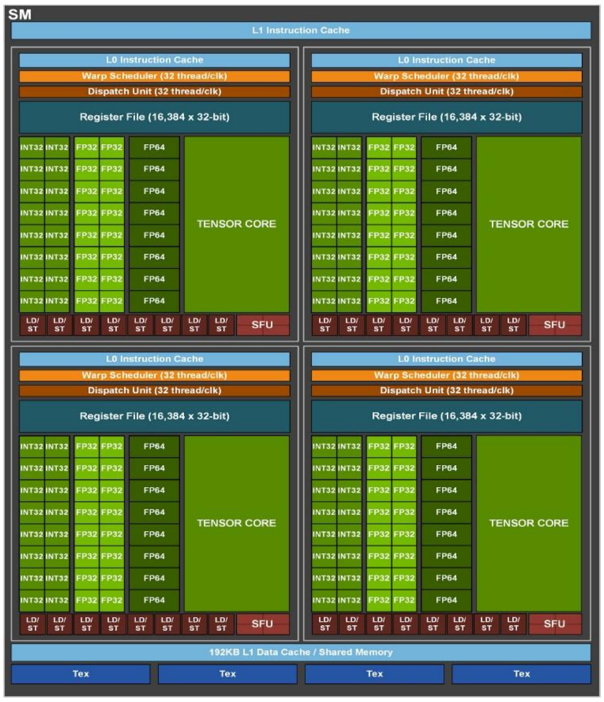
Επιταχυντές CPU

Ενώ οι επιταχυντές, οι οποίοι αναφέρονται στη συνέχεια, υπερέχουν στην παράλληλη επεξεργασία δεδομένων μεγάλης κλίμακας, στη δεκαετία του 2000 σχεδιάστηκαν συγκεκριμένα εξαρτήματα CPU, βασιζόμενα στον φόρτο εργασίας βίντεο και παιχνιδιών. Οι CPU άρχισαν να vector extensions, όπως επεκτάσεις SIMD (Single Instruction, Multiple Data), όπως το AVX της Intel και το Neon της ARM. Αυτές οι οδηγίες επιτρέπουν στις CPU να εκτελούν παράλληλες λειτουργίες σε πολλαπλά στοιχεία δεδομένων ταυτόχρονα. Με κατάλληλες βελτιστοποιήσεις, οι CPU μπορούν να επιτύχουν συγκεκριμένους φόρτους εργασίας AI, ειδικά όταν οι υπολογιστικές απαιτήσεις δεν είναι ιδιαίτερα παραλληλίσιμες. Οι CPU είναι πολύ αποδοτικές για DNN με παραλληλισμό μικρής ή μεσαίας κλίμακας, για αραιά DNN και σε σενάρια μικρού μεγέθους δεδομένων. Αξίζει επίσης να σημειωθεί ότι το είδος της επιτάχυνσης AI στο οποίο αναφερόμαστε όσον αφορά τις προσεγγίσεις που βασίζονται σε CPU είναι συνήθως συμπέρασμα. Ένας άλλος λόγος για τον οποίο μπορεί να προτιμώνται οι επιταχυντές CPU είναι η καθυστέρηση της συσκευής κεντρικού υπολογιστή που είναι πιο εμφανής στις άλλες συσκευές. Πολλοί προμηθευτές CPU παρέχουν συγκεκριμένες οδηγίες χαμηλού επιπέδου για να επωφεληθούν από τον παραλληλισμό της CPU. Για παράδειγμα, η Intel αξιοποιεί το Advanced Vector Extensions 512 (Intel® AVX-512) και πολλές ακόμη επεκτάσεις ειδικές για την τεχνητή νοημοσύνη, όπως τις οδηγίες DL Boost Vector Neural Network Instructions (VNNI), οι οποίες ενοποιούν τρεις εντολές σε μία. Γενικά, αυτές οι στρατηγικές βελτιστοποίησης μεγιστοποιούν τη χρήση υπολογιστικών πόρων, βελτιώνουν τη χρήση της κρυφής μνήμης με αποτέλεσμα σημαντική ενίσχυση της απόδοσης. Συμπερασματικά, η επιτάχυνση της CPU βασίζεται στην επιτάχυνση SIMD, η οποία εφαρμόζεται όταν η ίδια τιμή προστίθεται (ή αφαιρείται από) μεγάλο αριθμό σημείων δεδομένων. Αυτή η χρήση των οδηγιών SIMD επιτρέπει την αποτελεσματική παράλληλη επεξεργασία και μπορεί να βελτιώσει σημαντικά την απόδοση των αλγορίθμων που παρουσιάζουν παραλληλισμό δεδομένων. Ωστόσο, είναι σημαντικό να σημειωθεί ότι δεν μπορούν εύκολα να επωφεληθούν όλοι οι αλγόριθμοι από το SIMD, καθώς ορισμένες εργασίες με πολύπλοκη ροή μπορεί να θέτουν προκλήσεις για τη παραλληλοποίηση. Ωστόσο, οι εξελίξεις στην έρευνα και τις τεχνικές χειροκίνητης υλοποίησης ανοίγουν το δρόμο για καλύτερη υποστήριξη και αυτόματη διανυσματοποίηση στους μεταγλωττιστές, διασφαλίζοντας ότι η δυνατότητα της επιτάχυνσης SIMD μπορεί να αξιοποιηθεί πιο αποτελεσματικά σε ένα ευρύτερο φάσμα εφαρμογών.

Επιταχυντές GPU

Η GPU είναι ένας υπολογιστικός επεξεργαστής που εκτελεί γρήγορους υπολογισμούς για σκοπούς απόδοσης εικόνας και γραφικών. Οι GPU αξιοποιούν τεχνικές παράλληλης επεξεργασίας για να επιταχύνουν τις λειτουργίες τους. Στην πραγματικότητα, ορισμένες από τις GPU υψηλής τεχνολογίας έχουν υψηλότερο αριθμό τρανζίστορ από τη μέση CPU. Η αρχή πίσω από τη λειτουργία τους είναι ο διαχωρισμός των εργασιών σε μικρότερα τμήματα και η διανομή τους σε πολυάριθμους πυρήνες επεξεργαστών, που συχνά φθάνουν σε εκατοντάδες ή χιλιάδες πυρήνες, που λειτουργούν εντός της ίδιας GPU (CUDA cores). Ιστορικά, οι GPU χειρίζονταν κυρίως την απόδοση 2D και 3D εικόνων, βίντεο και κινούμενων εικόνων, αλλά τώρα περιλαμβάνουν ένα ευρύτερο φάσμα εφαρμογών, περιλαμβάνουν ανάλυση DL και big data.

Ο πολύ μεγάλος αριθμός πυρήνων ή νημάτων που έχουν αυτές οι συσκευές μεταφράζεται συχνά σε πολύ υψηλό παραλληλισμό που είναι ιδιαίτερα ωφέλιμο για εργασίες τεχνητής νοημοσύνης που περιλαμβάνουν πολύπλοκες μαθηματικές πράξεις, όπως αλγόριθμους βαθιάς μάθησης. Για παράδειγμα, ο πολλαπλασιασμός πινάκων στην εκπαίδευση ή το inference νευρωνικών δικτύων είναι συνηθισμένες διαδικασίες και οι GPU μπορούν να κάνουν αυτό το είδος λειτουργίας πολύ αποτελεσματικά. Επιπλέον, συχνά διαθέτουν ειδική μνήμη τυχαίας πρόσβασης βίντεο (VRAM). Η φύση των εφαρμογών που συνήθως εκτελούν οι GPU απαιτούν σημαντικό bandwidt μνήμης. Έτσι, η VRAM είναι ένα πολύ σημαντικό στοιχείο και πρέπει να είναι φυσικά κοντά στους υπολογισμούς για να παρέχει δεδομένα με υψηλή απόδοση στους πυρήνες επεξεργασίας της συσκευής. Εκτός από τις υπολογιστικές της δυνατότητες, μια GPU χρησιμοποιεί εξειδικευμένο προγραμματισμό για να διευκολύνει την ανάλυση και τη χρήση δεδομένων.


SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
INT32
FP32
FP64
TENSOR CORE
LD/ST
SFU
192KB L1 Data Cache / Shared Memory
Tex

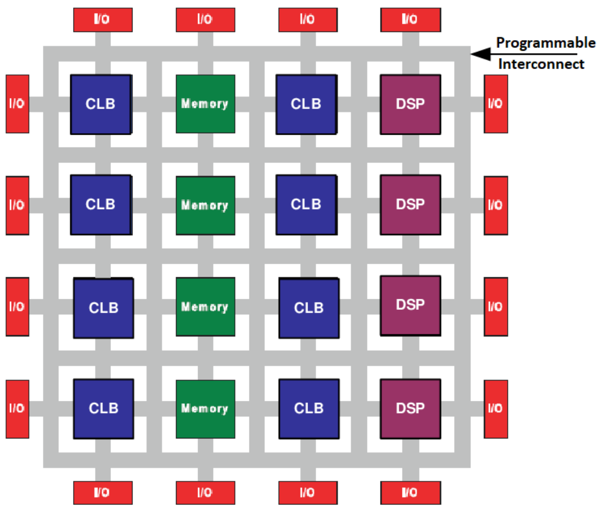
Σχήμα 4. Nvidia Ampere SM μπλοκ διάγρμμα [17]

Επιταχυντές FPGA

Μια συστοιχία επιτόπια προγραμματιζόμενων πυλών (FPGA) είναι ένα ολοκληρωμένο κύκλωμα που μπορεί να διαμορφωθεί μετά την κατασκευή. Η διαμόρφωση τυπικά καθορίζεται χρησιμοποιώντας μια γλώσσα περιγραφής υλικού (HDL) ή γλώσσες υψηλότερης αφαίρεσης, όπως η σύνθεση υψηλού επιπέδου (HLS). Τα FPGA αποτελούνται από μια σειρά προγραμματιζόμενων λογικών μπλοκ και επαναδιαμορφώσιμων διασυνδέσεων, επιτρέποντας τη σύνδεση αυτών των μπλοκ Τα λογικά μπλοκ μπορούν να ρυθμιστούν για να εκτελούν σύνθετες συνδυαστικές λειτουργίες ή να λειτουργούν ως απλές λογικές πύλες όπως AND και XOR (Σχήμα 5). Πολλά FPGA περιλαμβάνουν επίσης στοιχεία μνήμης μέσα στα λογικά μπλοκ, που κυμαίνονται από βασικά flip-flops έως πιο ολοκληρωμένες μονάδες μνήμης. Αυτή η αναδιαμορφώσιμη φύση επιτρέπει στα FPGA να επαναπρογραμματίζονται για την υλοποίηση διαφορετικών λογικών συναρτήσεων, επιτρέποντας ευέλικτους και προσαρμόσιμους υπολογιστές παρόμοιους με τον προγραμματισμό λογισμικού.

Πιο συγκεκριμένα, τα FPGAs, πέρα από το να είναι απλώς μια σειρά από πύλες, διαθέτουν ένα εξελιγμένο δίκτυο διασυνδεδεμένων ψηφιακών υποκυκλωμάτων, σχεδιασμένο με ακρίβεια για να εκτελεί αποτελεσματικά κοινές λειτουργίες και να παρέχει υψηλή ευελιξία. Τα FPGAs αποτελούνται κυρίως από 3 μέρη:

• Διαμορφώσιμα Λογικά Μπλοκ — Στην καρδιά των δυνατοτήτων προγραμματιζόμενης λογικής ενός FPGA βρίσκεται μια συλλογή από διαμορφώσιμα λογικά μπλοκ (CLBs) που υλοποιούν λογικές λειτουργίες.
• Προγραμματιζόμενες Διασυνδέσεις — Τα CLBs μέσα στο FPGA χρειάζονται επικοινωνία μεταξύ τους, η οποία επιτυγχάνεται μέσω ενός πλέγματος προγραμματιζόμενων διασυνδέσεων.
• Προγραμματιζόμενα Μπλοκ Εισόδου/Εξόδου — Για να συνδεθούν με εξωτερικά κυκλώματα, τα FPGAs διαθέτουν προγραμματιζόμενα μπλοκ εισόδου/εξόδου.

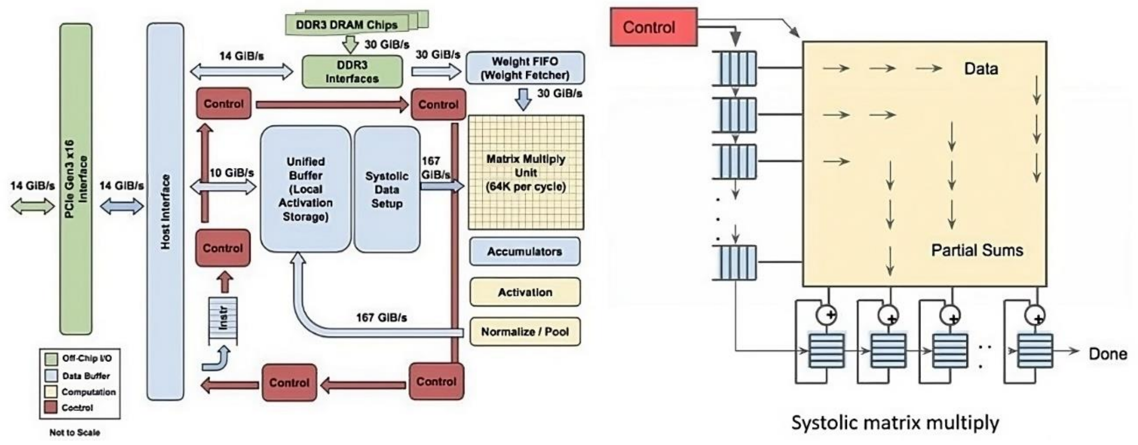


Σχήμα 5. Εσωτερική δομή ενός Xilinx FPGA [22]

Επιταχυντές ASIC

Ένα Εξειδικευμένο Ολοκληρωμένο Κύκλωμα (ASIC) είναι ένα ολοκληρωμένο κύκλωμα (IC) τσιπ προσαρμοσμένο για μια συγκεκριμένη χρήση, για παράδειγμα την επιτάχυνση μοντέλων τεχνητής νοημοσύνης. Συνήθως περιέχουν έναν συστολικό πίνακα, ο οποίος αποτελείται από ένα μεγάλο δίκτυο βασικών υπολογιστικών κόμβων, που μπορεί να είναι είτε ενσύρματοι είτε διαμορφωμένοι μέσω λογισμικού για συγκεκριμένες εφαρμογές. Αυτοί οι κόμβοι είναι συνήθως σταθεροί και πανομοιότυποι, ενώ η διασύνδεση μεταξύ τους είναι προγραμματιζόμενη. Σε αντίθεση με την παραδοσιακή αρχιτεκτονική Von Neumann, όπου η εκτέλεση του προγράμματος ακολουθεί μια σειρά εντολών αποθηκευμένων σε κοινή μνήμη, διευθυνόμενες υπό τον μετρητή προγράμματος (PC) της CPU, οι μεμονωμένοι κόμβοι μέσα σε έναν συστολικό πίνακα ενεργοποιούνται από την άφιξη νέων δεδομένων και επεξεργάζονται τα δεδομένα με συνεπή τρόπο. Λόγω της ικανότητάς του να χειρίζεται πολλαπλές ροές δεδομένων μέσω μετρητών δεδομένων, ο συστολικός πίνακας υποστηρίζει παράλληλη επεξεργασία δεδομένων. Αυτό επιτρέπει στον συστολικό πίνακα να επεξεργάζεται αποδοτικά πολλαπλές ροές δεδομένων ταυτόχρονα.

Υπάρχουν πολλές συσκευές που έχουν αναπτυχθεί από διάφορες εταιρείες και οργανισμούς και ταξινομούνται ως επιταχυντές AI ASIC. Ο πιο δημοφιλής μέχρι σήμερα είναι ο TPU (Tensor Processing Unit) της Google, ο οποίος μοιράζεται πολλά κοινά χαρακτηριστικά με πολλούς επιταχυντές της ίδιας κατηγορίας. Η Google ανέπτυξε τον δικό της εξατομικευμένο επιταχυντή AI ASIC, τον TPU, για να επιταχύνει τα φορτία εργασίας μηχανικής μάθησης, ιδιαίτερα την εκτέλεση νευρωνικών δικτύων. Οι TPUs χρησιμοποιούνται εκτενώς στα κέντρα δεδομένων της Google για να επιταχύνουν διάφορες εφαρμογές AI. Ενσωματώνουν εξειδικευμένα χαρακτηριστικά όπως η μονάδα πολλαπλασιασμού πινάκων (MXU), που βελτιστοποιεί σύνθετες λειτουργίες πινάκων συχνά με την χρήση μνήμης υψηλής ταχύτητας (HBM). Αυτοί οι TPUs μπορούν να ομαδοποιηθούν σε συμπλέγματα γνωστά ως Pods, επιτρέποντας κλιμακούμενη και επιταχυνόμενη εκπαίδευση και εκτέλεση μηχανικής μάθησης. Οι προγραμματιστές αξιοποιούν TPUs στο cloud καθώς και στο edge για να επωφεληθούν από υψηλή απόδοση, απρόσκοπτες διαδικασίες ανάπτυξης και οικονομική αποδοτικότητα.



Σχήμα 6. Ο επιταχυντής TPU της Google και η λειτουργία του.

Ο ρόλος του Approximate Computing

Η αποτυχία της κλιμάκωσης Dennard οδήγησε στην εμφάνιση του "προβλήματος του σκοτεινού πυριτίου" [29], αναγκάζοντας τους σχεδιαστές υπολογιστών να εξερευνήσουν καινοτόμες προσεγγίσεις για να διατηρήσουν και να βελτιώσουν την αποδοτικότητα των υπολογιστικών συστημάτων. Στον τομέα των υπολογιστών, αναδείχθηκαν αρκετά πρωτοποριακά παραδείγματα και την τελευταία δεκαετία, μία από τις πιο αξιοσημείωτες εξελίξεις έχει πραγματοποιηθεί στην έρευνα για την Προσεγγιστική Υπολογιστική (Approximate Computing). Η προσεγγιστική υπολογιστική περιλαμβάνει τεχνικές που εκμεταλλεύονται την εγγενή ανθεκτικότητα σφαλμάτων διάφορων εφαρμογών για να επιτύχουν αυξημένη αποδοτικότητα σε όρους ενέργειας και απόδοσης σε όλα τα επίπεδα των υπολογισμών. Οι εφαρμογές AI προσφέρουν πολλές ευκαιρίες για την εφαρμογή τεχνικών Approximate Computing λόγω διαφόρων παραγόντων:

• Εγγενής Ανθεκτικότητα Σφαλμάτων: Πολλές εργασίες AI συχνά περιλαμβάνουν τη διαχείριση μεγάλων συνόλων δεδομένων και σύνθετων μοντέλων. Αυτές οι εργασίες παρουσιάζουν ένα επίπεδο εγγενούς ανθεκτικότητας σφαλμάτων, που σημαίνει ότι μικρές αποκλίσεις στον υπολογισμό ή προσεγγίσεις ενδέχεται να μην επηρεάσουν σημαντικά τη συνολική ποιότητα των αποτελεσμάτων.
• Προσέγγιση Λειτουργιών: Οι εργασίες AI συχνά περιλαμβάνουν την προσέγγιση σύνθετων λειτουργιών για τη μοντελοποίηση προτύπων και σχέσεων μέσα στα δεδομένα. Οι τεχνικές προσεγγιστικών υπολογισμών μπορούν να εφαρμοστούν σε αυτές τις προσεγγίσεις, βελτιστοποιώντας την ανταλλαγή μεταξύ ακρίβειας και υπολογιστικών πόρων.
• Υπολογιστική Ένταση της AI: Οι εφαρμογές AI, ιδιαίτερα τα μοντέλα βαθιάς μάθησης, μπορεί να είναι υπολογιστικά έντονες, απαιτώντας σημαντικούς πόρους για την εκτέλεση και την εκπαίδευση. Οι τεχνικές προσεγγιστικών υπολογισμών μπορούν να μειώσουν σημαντικά αυτές τις υπολογιστικές απαιτήσεις χωρίς να υποβαθμίσουν την ποιότητα των αποτελεσμάτων, καθιστώντας τις εφαρμογές AI πιο αποδοτικές και οικονομικά αποδοτικές.
• Παραλληλισμός και Επιτάχυνση Υλικού: Τα φορτία εργασίας AI συχνά προσφέρονται καλά για παράλληλη επεξεργασία και επιτάχυνση υλικού. Οι τεχνικές προσεγγιστικών υπολογισμών μπορούν να εφαρμοστούν αποδοτικά σε παράλληλα υπολογιστικά συστήματα και εξειδικευμένο υλικό, ενισχύοντας περαιτέρω τα οφέλη τους σε όρους απόδοσης και ενεργειακής αποδοτικότητας.
• Συστήματα Πραγματικού Χρόνου και Ενσωματωμένα Συστήματα: Σε ορισμένες εφαρμογές AI, όπως αυτές σε συστήματα πραγματικού χρόνου ή ενσωματωμένα συστήματα, οι περιορισμοί ισχύος και πόρων είναι κρίσιμες παράμετροι. Οι τεχνικές προσεγγιστικών υπολογισμών μπορούν να αντιμετωπίσουν αυτούς τους περιορισμούς.

• Εμφάνιση Υλικού Χαμηλής Ακρίβειας: Η προσεγγιστικοί υπολογισμοί ευθυγραμμίζεται καλά με την τάση ανάπτυξης εξειδικευμένου υλικού χαμηλής ακρίβειας για το AI. Τέτοιο υλικό μπορεί να εκμεταλλευτεί την προσεγγιστική αριθμητική για να επιτύχει υψηλότερη απόδοση και ενεργειακή αποδοτικότητα σε σύγκριση με τις παραδοσιακές προσεγγίσεις υψηλής ακρίβειας.
• Ευελιξία Ανταλλαγών: Οι εφαρμογές AI συχνά περιλαμβάνουν πολύπλοκες ανταλλαγές μεταξύ ακρίβειας, απόδοσης και κατανάλωσης ενέργειας, επομένως η προσεγγιστική αριθμητική μπορεί να επιτρέψει την εξερεύνηση αυτών των ανταλλαγών.

Σχεδίαση Επιταχυντών Υλικού Βαθιάς Μάθησης

Σε αυτή την ενότητα, παρουσιάζουμε μια λεπτομερή ανάλυση των τεχνικών βελτιστοποίησης λογισμικού και υλικού που εφαρμόζονται σε Βαθιά Νευρωνικά Δίκτυα για αποδοτική εκτέλεση. Παρουσιάζουμε τρεις εφαρμογές, τρεις διακριτές εργασίες AI, με λεπτομέρειες υλοποίησης σχετικά με την επιτάχυνση χρησιμοποιώντας επαναδιαμορφώσιμο υλικό (FPGA). Ακόμα προτείνουμε διάφορες βελτιστοποιήσεις για ένα ολοκληρωμένο περιβάλλον για την αυτόματη επιτάχυνση των συνελικτικών νευρωνικών δικτύων σε FPGAs και δείχνουμε την απόδοσή του μέσω διαφόρων πειραμάτων.

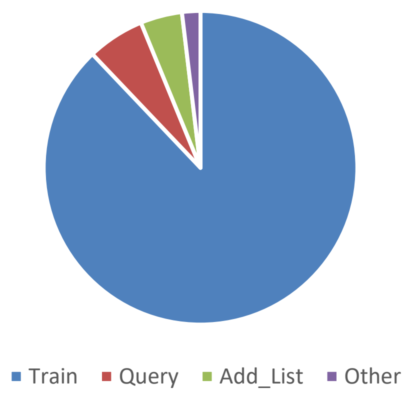
Επιταχυνόμενη αναζήτηση ομοιότητας διανυσμάτων μέσω ευρετηρίου

Αυτή η ενότητα περιέχει το πρώτο από τα τρία σενάρια στα οποία χρησιμοποιήσαμε FPGAs για επιτάχυνση AI. Συγκεκριμένα, αφορά μια νέα ενσωμάτωση των FPGAs στο δημοφιλές πλαίσιο FAISS (Facebook AI Similarity Search) [76] για την επιτάχυνση του αλγορίθμου αναζήτησης ομοιότητας. Ένας από τους πιο σημαντικούς αλγορίθμους στη Μηχανική Μάθηση που χρησιμοποιείται για την εκτέλεση αναζητήσεων ομοιότητας είναι ο αλγόριθμος K-Nearest Neighbor (KNN). Ο αλγόριθμος αυτός βρίσκει εκτεταμένη χρήση σε εργασίες όπως η προγνωστική ανάλυση, η κατηγοριοποίηση κειμένου και η αναγνώριση εικόνας. Ωστόσο, αυτός ο αλγόριθμος έρχεται με έναν συμβιβασμό, συχνά απαιτώντας σημαντικούς υπολογιστικούς πόρους. Για να αντιμετωπιστεί αυτή η πρόκληση, μεγάλες εταιρείες που διαχειρίζονται μεγάλα σύνολα δεδομένων σε σύγχρονα κέντρα δεδομένων συνδυάζουν την τεχνική KNN με αλγοριθμικές προσεγγίσεις, επιτρέποντας τον υπολογισμό κρίσιμων φορτίων εργασίας σε πραγματικό χρόνο. Παρ' όλα αυτά, οι απαιτήσεις υπολογισμού και η κατανάλωση ενέργειας αυξάνονται περαιτέρω όταν πρόκειται για ερωτήματα υψηλής διάστασης για τους πλησιέστερους γείτονες. Σε αυτή τη μελέτη, παρουσιάζουμε μια καινοτόμο προσέγγιση: έναν υλικο-επιταχυνόμενο κατά προσέγγιση αλγόριθμο KNN ενσωματωμένο στο περιβάλλον FAISS μέσω πλατφορμών FPGA-OpenCL. Η αρχιτεκτονική FPGA σε αυτό το πλαίσιο αντιμετωπίζει αποτελεσματικά τις πολυπλοκότητες της ευρετηρίασης διανυσμάτων κατά την εκπαίδευση και την ενσωμάτωση δεδομένων μεγάλης κλίμακας και υψηλής διάστασης. Η προτεινόμενη λύση αξιοποιεί μια μορφή ενσωματωμένης μνήμης βασισμένη σε FPGA, η οποία ξεπερνά τα πολυπύρηνα συστήματα υψηλής απόδοσης τόσο σε ταχύτητα όσο και σε ενεργειακή αποδοτικότητα. Τα εμπειρικά πειράματα που πραγματοποιήθηκαν στο FPGA Xilinx Alveo U200 αποκάλυψαν σημαντικά αποτελέσματα. Η επιτάχυνση που επιτεύχθηκε είναι έως και 98 φορές ταχύτερη από έναν μονοπύρηνο επεξεργαστή όταν χρησιμοποιείται μόνο ο επιταχυντής, και η ταχύτητα του συστήματος από άκρη σε άκρη βελτιώθηκε κατά 2,1 φορές σε σύγκριση με έναν επεξεργαστή Xeon με 36 νήματα. Επιπλέον, η απόδοση ανά βατ του υλικού παρουσιάζει αξιοσημείωτη αύξηση κατά 3.5× σε σύγκριση με τον ίδιο επεξεργαστή και 1.2× σε σύγκριση με μια GPU κατηγορίας Kepler.

Λειτουργία του FAISS

Το FAISS αξιοποιεί μια ποικιλία μεθόδων σχεδιασμένων για την εκτέλεση αναζητήσεων ομοιότητας σε πυκνά διανύσματα που περιέχουν πραγματικές ή ακέραιες τιμές. Η υποκείμενη δομή του FAISS περιλαμβάνει τη χρήση διαφόρων προσεγγίσεων ευρετηρίασης για την αποθήκευση διανυσμάτων, και ο υπολογισμός της απόστασης περιλαμβάνει πολλαπλές μετρικές. Η επόμενη υλοποίησή μας, η οποία περιγράφεται στην ακόλουθη παράγραφο, εστιάζει στη χρήση της ευρετηρίασης IVFFlat ως μια αντιπροσωπευτική περίπτωση χρήσης (κατάλληλη για σενάρια υψηλής ακρίβειας), αλλά ο σχεδιασμός μπορεί εύκολα να εφαρμοστεί σε άλλα ευρετήρια όπως το IndexIVFPQ. Για να καθοριστεί ποια στοιχεία του Faiss πρέπει να δοθούν προτεραιότητα για υλοποίηση σε υλικό, πραγματοποιήθηκε μια ολοκληρωμένη προφίλ του περιβάλλοντος. Αυτή η ενότητα ξεκινά με την περιγραφή της διαδικασίας που εμπλέκεται στο λεγόμενο profiling, κατά την οποία επιλέχθηκε το ευρετήριο IVFFlat ως αντιπροσωπευτική περίπτωση χρήσης. Στη συνέχεια, προσδιορίζεται ο αλγόριθμος υλικού, μαζί με τις βελτιστοποιήσεις που πραγματοποιήθηκαν τόσο στην πλευρά του κεντρικού υπολογιστή όσο και στην πλευρά του πυρήνα για τη βελτιστοποίηση της διαπερατότητας και την ελαχιστοποίηση της συνολικής καθυστέρησης σχεδιασμού. Το τελευταίο μέρος αυτής της ενότητας εξετάζει το νέο προσαρμοσμένο περιβάλλον. Αυτό το πλαίσιο χρησιμοποιεί ένα FPGA με βελτιστοποιημένο σχήμα μεταφοράς μνήμης που ενσωματώνει απρόσκοπτα τους σχεδιασμένους επιταχυντές μας για να εξασφαλίσει ελάχιστη καθυστέρηση κατά τη διάρκεια των συναλλαγών μνήμης.

Λειτουργίες που καταναλώνουν το μεγαλύτερο μέρος του χρόνου εκτέλεσης αποτελούν κατάλληλους υποψηφίους για εκφόρτωση και επιτάχυνση σε FPGAs (βλέπε Σχήμα Figure 3-4). Τα αποτελέσματα του profiling αποκάλυψαν ότι το μεγαλύτερο υπολογιστικό φόρτο προκύπτει κατά τη δημιουργία του ευρετηρίου και την προσθήκη δεδομένων σε αυτό. Αυτό ισχύει ακόμα και όταν εξετάζουμε μέτριες αναζητήσεις ερωτημάτων που περιλαμβάνουν μερικές χιλιάδες διανύσματα. Αυτό το φαινόμενο οφείλεται στο γεγονός ότι, ενώ η κατά προσέγγιση αναζήτηση πλησιέστερων γειτόνων είναι ιδιαίτερα αποδοτική, απαιτεί εκτεταμένους χρόνους εκπαίδευσης, ειδικά όταν απαιτούνται πολλαπλά σημεία συσσωμάτωσης για τη δειγματοληψία του συνόλου δεδομένων. Σημειώνεται ότι ο αλγόριθμος εκπαίδευσης ενσωματώνει πολυάριθμες λειτουργίες Πολλαπλασιασμού-Πρόσθεσης (MAC), τις οποίες το Faiss εκτελεί αυτή τη στιγμή χρησιμοποιώντας βελτιστοποιημένες ρουτίνες BLAS για CPU από προεπιλογή. Δεδομένων των κυρίαρχων χαρακτηριστικών αυτών των αλγορίθμων, είναι πολύ συνηθισμένο να αντιστοιχίζονται σε υλικό, λόγω του υψηλού λόγου των αριθμητικών λειτουργιών προς τα bytes που μεταφέρονται.



Σχήμα 7. Κατανομή υπολογιστικού φόρτου στο ευρετήριο IVF (Flat)

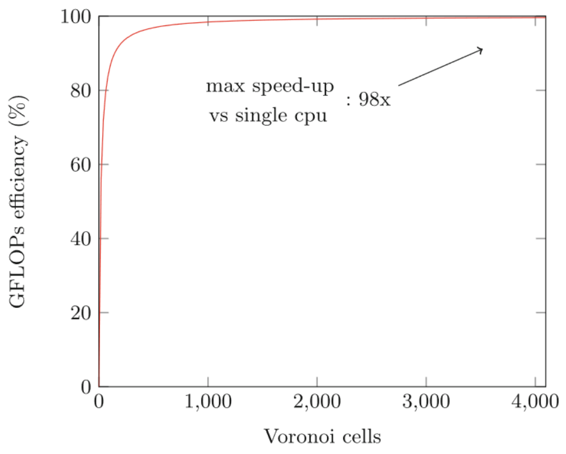
Αποτελέσματα

Για να αξιολογήσουμε τον σχεδιασμό, επαληθεύσαμε την ακρίβεια του επιταχυντή και ποσοτικοποιήσαμε την απόδοσή του. Στη συνέχεια, μετά την ενσωμάτωση με το Faiss, πραγματοποιήσαμε αξιολογήσεις της ακρίβειας και της αποδοτικότητας του τελικού συστήματος χρησιμοποιώντας δεδομένα από τον πραγματικό κόσμο. Επιπλέον, μετρήσαμε την ενεργειακή αποδοτικότητα σε σύγκριση με εναλλακτικά συστήματα, όπως οι διαμορφώσεις CPU και GPU. Η διαμόρφωση του συστήματος έγινε σε OpenCL-FPGAs, χρησιμοποιώντας συγκεκριμένα μια κάρτα Xilinx Alveo U200 για κέντρα δεδομένων, εξοπλισμένη με τέσσερα κανάλια DDR4. Αυτό συνδυάστηκε με ένα κεντρικό σύστημα που χρησιμοποιούσε μια CPU Xeon. Για μια ολοκληρωμένη σύγκριση με μια υψηλής απόδοσης CPU, επιλέξαμε μια περίπτωση c4.8xlarge από το AWS Cloud, εξοπλισμένη με μια CPU Xeon με 36 vCPUs και 60 GiB RAM. Αξιοσημείωτα, αυτή η περίπτωση έχει ισοδύναμο κόστος (ανά ώρα) με μια περίπτωση f1.2xlarge που διαθέτει παρόμοιο FPGA, το VU9P. Επιπλέον, για την τελική αξιολόγηση της απόδοσης του τελικού συστήματος, αξιολογήσαμε την αποδοτικότητα σε όρους απόδοσης ανά watt σε σύγκριση τόσο με την ίδια CPU όσο και με μια GPU κλάσης Kepler K40. Ο σχεδιασμός του υλικού FPGA μεγιστοποιεί τη χρήση όλων των DDRs και αξιοποιεί βέλτιστα τους πόρους εντός κάθε SLR. Ωστόσο, ο κύριος περιορισμός για περαιτέρω κλιμάκωση έγκειται στη δρομολόγηση μεταξύ των τριών SLRs. Η χρήση πόρων ενός μόνο πυρήνα στη συσκευή FPGA περιγράφεται στο παρακάτω πίνακα.

Σύνοψη χρήσης πόρων				
Name	**BRAM**	**DSP**	**FF**	**LUT**
Total	502	1036	156137	89206
Percentage (%)	11	15	6	7

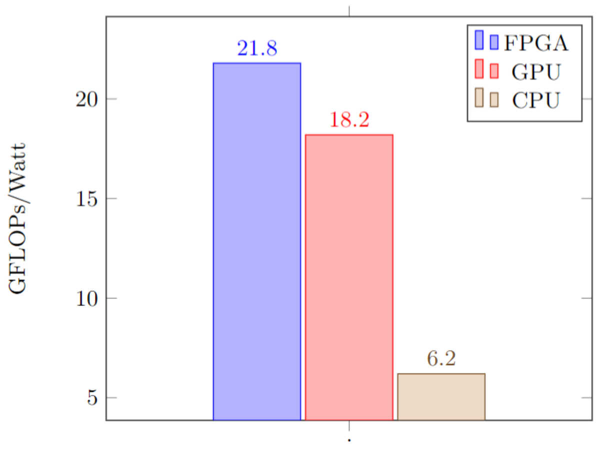
Πίνακας 2. Πόροι ανά πυρήνα στο FPGA

Το παρακάτω σχήμα απεικονίζει τη σημαντική επιτάχυνση που επιτεύχθηκε αποκλειστικά από τον επιταχυντή FPGA, με βελτιώσεις που φτάνουν περίπου 98 φορές για μεγαλύτερα μεγέθη κελιών. Το βέλος δείχνει τη μέγιστη επιτάχυνση που επιτεύχθηκε σε σύγκριση με έναν μονοπύρηνο επεξεργαστή CPU. Αυτό προκύπτει από το γεγονός ότι για περισσότερα κελιά, η επίδραση της μεταφοράς δεδομένων είναι λιγότερο εμφανής. Αξίζει να σημειωθεί ότι η χαμηλότερη τιμή αποδοτικότητας, η οποία συμβαίνει για περίπου 500 κελιά και κάτω, δεν επηρεάζει την συνολική απόδοση του αλγορίθμου. Συνήθως, σε πραγματικά datasets, ειδικά σε μεγαλύτερα που χρησιμοποιούνται σε κέντρα δεδομένων, τα κύτταρα Voronoi είναι πολλαπλάσια των χιλιάδων για ικανοποιητική συσσώρευση, ακόμα και για ένα μέτριο dataset του ενός εκατομμυρίου.



Σχήμα 8. Αποτελεσματικότητα πυρήνα ανά κύτταρο Voronoi

Ακόμα, παρουσιάζουμε και την αποτελεσματικότητα σε GFLOPs/Watt σε σχέση με άλλες αρχιτεκτονικές.



Σχήμα 9. Απόδοση κατανάλωσης ενέργειας.

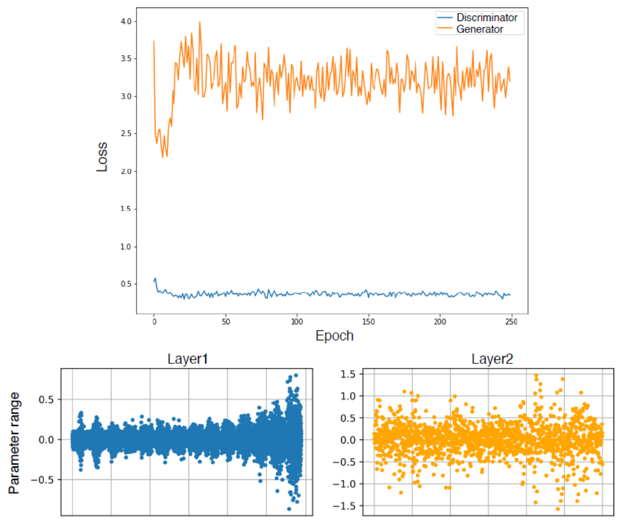
Τα ευρήματά μας δείχνουν ότι ο επιταχυντής μας υπερβαίνει τις δυνατότητες μιας CPU Xeon με 36 νημάτων. Επιπλέον, επιδεικνύει ανώτερη απόδοση ανά watt σε σύγκριση και με την ίδια CPU και με μια GPU κλάσης Kepler. Αυτό υπογραμμίζει την αποτελεσματικότητα μιας προσέγγισης κωδικοποίησης λογισμικού/υλικού για την αντιμετώπιση των απαιτήσεων των φορτίων εργασίας στο υπολογιστικό νέφος, ειδικά σε σενάρια όπως οι χρόνοι δείκτη για προσεγγισμένους αλγορίθμους KNN. Η αυξημένη απόδοση και αποδοτικότητα του σχεδιασμού μας κρατά το δυναμικό να επαναπροσδιορίσει τη χρήση του υλικού FPGA σε περιβάλλοντα νέφους και μεγάλα κέντρα δεδομένων, λαμβάνοντας υπόψη τη σημασία της ενεργειακής αποδοτικότητας σε αυξημένες απαιτήσεις φορτίων εργασίας. Σαν μελλοντικός στόχος, προκειμένου να λυθεί το πρόβλημα μνήμης για datasets σε δισεκατομμύρια καταχωρήσεις, θα απαιτηθεί η διανομή της εφαρμογής σε έναν αριθμό πολλών FPGA. Ο αλγόριθμός μας σχεδιάστηκε με τρόπο ώστε η κεντρική εφαρμογή να μπορεί εύκολα να διανέμει τα δεδομένα και έτσι το φορτίο εργασίας σε έναν αριθμό FPGA στο νέφος το οποίο λόγω περιορισμένης υποδομής την περίοδο της μελέτης δεν μπορούσαμε να επιτύχουμε.

Επιταχυνόμενη ανακατασκευή εικόνας μέσω GANs

Αυτή η ενότητα περιλαμβάνει το δεύτερο από τα τέσσερα σενάρια στα οποία χρησιμοποιήσαμε επιτάχυνση FPGA για την εκτέλεση της τεχνητής νοημοσύνης. Ακριβείς και αποδοτικοί αλγόριθμοι Μηχανικής Μάθησης έχουν μεγάλη σημασία σε διάφορες προκλήσεις, ιδίως σε εργασίες που αφορούν ταξινόμηση. Το τελευταίο διάστημα, έχει εμφανιστεί μια νέα κατηγορία Μηχανικής Μάθησης γνωστή ως Γεννητικά Ανταγωνιστικά Δίκτυα (GANs). Τα GANs λειτουργούν χρησιμοποιώντας δύο νευρωνικά δίκτυα: ένα δημιουργικό δίκτυο (γεννήτορας) και ένα διακριτικό δίκτυο (διακριτής).

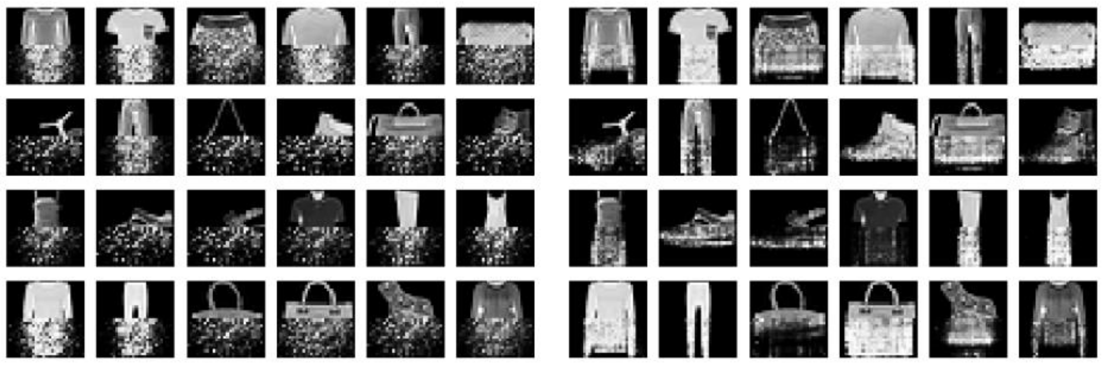
Αυτά τα δίκτυα συμμετέχουν σε μια ανταγωνιστική διαδικασία με στόχο τη δημιουργία νέων δεδομένων, όπως εικόνες. Για παράδειγμα, ένα GAN μπορεί να ανακατασκευάσει μια εικόνα που είναι παραμορφωμένη από θόρυβο ή περιέχει κατεστραμμένα τμήματα. Αυτή η έννοια ανακατασκευής εικόνας έχει ποικίλες εφαρμογές στην όραση υπολογιστών, την επαυξημένη πραγματικότητα, την αλληλεπίδραση ανθρώπου-υπολογιστή, την κινούμενη εικόνα και την ιατρική εικονική πραγματικότητα. Ωστόσο, αυτή η αλγοριθμική προσέγγιση απαιτεί έναν σημαντικό αριθμό λειτουργιών MAC (πολλαπλασιασμός-πρόσθεση) και καταναλώνει σημαντική ενέργεια για να λειτουργήσει. Σε αυτήν την ενότητα, περιγράφουμε την υλοποίηση ενός αλγορίθμου ανακατασκευής εικόνας με χρήση GANs. Ειδικότερα, επικεντρωνόμαστε στην εκπαίδευση ενός μοντέλου για την αποκατάσταση εικόνων ρούχων χρησιμοποιώντας το σύνολο δεδομένων fashion-MNIST ως περίπτωση μελέτης. Επιπλέον, εφαρμόζουμε και βελτιστοποιούμε αυτόν τον αλγόριθμο σε ένα Xilinx FPGA SoC. Αυτές οι πλατφόρμες έχουν επιδείξει σημαντική ικανότητα στην αντιμετώπιση τέτοιων προκλήσεων όσον αφορά την απόδοση και τη διαχείριση της ενέργειας. Η σχεδιασμένη προσέγγιση υπερτερεί επίσης σε σχέση με τις ρυθμίσεις CPU και GPU, επιτυγχάνοντας ένα μέσο χρόνο ανακατασκευής των εικόνων 0.013 χιλιοστά του δευτερολέπτου ανά εικόνα και ένα μέγιστο λόγο σήματος προς θόρυβο (PSNR) των 43 dB στην κβαντική ρύθμιση του FPGA.

Στο ακόλουθο Σχήμα, μπορούμε να παρατηρήσουμε την απώλεια που επιτυγχάνεται για τον διακριτή και το νέο μοντέλο γεννήτριας. Επίσης, στο κάτω μέρος του Σχήματος μπορούμε να δούμε την εύρος παραμέτρων για κάθε στρώμα όπως αποκτήθηκε από τον περιορισμό MinMax.

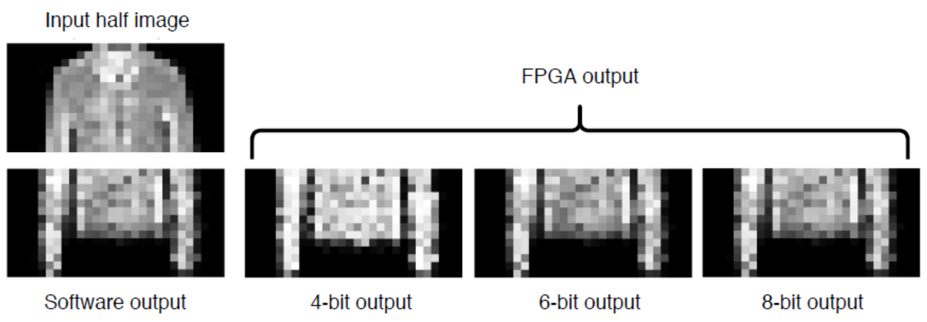


Σχήμα 10. Αποτελέσματα Εκπαίδευσης (επάνω: Απώλεια για το Διακριτή και το Μοντέλο της Γεννήτριας, κάτω: εικόνα του εύρους παραμέτρων σε κάθε επίπεδο)

Η παρακάτω εικόνα παρουσιάζει μια συλλογή από διαφορετικές εικόνες που καταγράφηκαν κατά τη διάρκεια και των αρχικών και των τελικών εποχών εκπαίδευσης. Εμφανώς, κατά την αρχική εποχή, τα κάτω μέρη των εικόνων (που αποτελούν την έξοδο που δημιουργείται από το μοντέλο της γεννήτριας) φαίνεται σαν να αποτελούνται από τυχαίο θόρυβο. Ωστόσο, κατά την ολοκλήρωση της εποχής (δεξιά), οι παρουσιαζόμενες εικόνες αποτελούν μια συλλογή ολοκληρωμένων σύνολων ρούχων, με μεγάλη ομοιότητα με πραγματικά ρούχα.


Σχήμα 11. Αποτελέσματα της γεννήτριας για την ανακατασκευή εικόνας από την πρώτη (αριστερά) και την τελευταία (δεξιά) εποχή.

Στη συνέχεια, αξιολογούμε και αναλύουμε την εφαρμογή μας. Η αξιολόγησή μας θα ξεκινήσει με μια εξέταση των ανακατασκευασμένων εικόνων που προκύπτουν από το μοντέλο GAN στο υλικό, συμπεριλαμβανομένης μιας συγκριτικής μελέτης των σφαλμάτων σε διάφορες ακρίβειες bit. Επιπλέον, θα παρέχουμε αξιολόγηση της απόδοσης του επιταχυντή υλικού, της χρήσης πόρων και της αποδοτικότητας ενέργειας. Αυτή η αξιολόγηση θα συγκριθεί με άλλες πλατφόρμες.



Σχήμα 12. Ποιότητα ανακατασκευής για διάφορες αριθμητικές ακρίβειες στο FPGA

Όσον αφορά τη διαμόρφωση του συστήματος, χρησιμοποιήσαμε ένα Xilinx ZC702, το οποίο είναι εξοπλισμένο με ένα σύστημα τσιπ (SoC) Zynq-7000 που διαθέτει έναν διπύρηνο επεξεργαστή ARM Cortex-A9 και 512 MB μνήμης DDR3. Η κατανομή πόρων του επιταχυντή υλικού μας βασισμένου σε FPGA, σε συνδυασμό με τους χρόνους καθυστέρησης και τις εικόνες ανά δευτερόλεπτο (FPS) που επιτεύχθηκαν, αναλύεται λεπτομερώς στον πίνακα παρακάτω.

	Σύνοψη χρήσης πόρων				Χρονισμός	
Name	**BRAM**	**DSP**	**FF**	**LUT**	**Latency**	**FPS**
Used	54	110	18907	9855	-	-
Percentage	38.57%	50%	11.77%	18.52%	0.013 (ms)	77K

Πίνακας 3. Χρήση πόρων και χρόνος ανά πυρήνα

Για να αξιολογήσουμε πλήρως την απόδοση του συστήματός μας, συμπεριλαμβανομένων των μεταφορών μνήμης, υλοποιήσαμε το ίδιο μοντέλο γεννήτριας σε εναλλακτικά συστήματα (CPU, GPU). Αυτό έγινε για να διευκολυνθεί μια δίκαιη σύγκριση απόδοσης και απόδοσης ανά βατ (PPW). Εντυπωσιακά, αυτή η προσέγγιση επέφερε ευνοϊκά αποτελέσματα σε όλες τις πλατφόρμες, τόσο σε όρους απόδοσης όσο και σε μετρική απόδοσης-σε-ισχύ (PPW), όπως φαίνεται από τις μετρήσεις παρακάτω.

Device Information		**Evaluation**			
System	**Model**	**Time/Img**	**Speed-up**	**Power**	**PPW**
CPU	ARM A9	2.06ms	1×	3.2W	1×
GPU	Nvidia K80	0.033ms	62×	74W	2.7×
FPGA	ZC702	0.013ms	158×	3.6W	140×

Πίνακας 4. Μέτρηση χρόνου και ενεργειακής απόδοσης ανά σύστημα

Η μελέτη εξέτασε τη χρήση του GAN για ανακατασκευή εικόνων σε FPGA, αποδεικνύοντας την υπεροχή του σε ακρίβεια, ταχύτητα και ενεργειακή απόδοση σε σχέση με CPU και GPU. Το μοντέλο της γεννήτριας (GAN) παρήγαγε υψηλής ποιότητας εικόνες με εξαιρετική απόδοση. Ο στόχος είναι να ενισχυθεί η παρουσία των FPGA στο λογισμικό-υλικό περιβάλλον και μέσα από αυτή την εφαρμογή επιτεύχθηκε το βήμα αυτό σε ένα σημαντικό βαθμό.

Επιταχυνόμενος εντοπισμός του Covid σε ακτινογραφίες

Μέσα σε αυτή την ενότητα, εξετάζουμε το τρίτο σενάριο από τα τέσσερα στα οποία χρησιμοποιήσαμε επιτάχυνση FPGA για να βελτιώσουμε την εκτέλεση εφαρμογών τεχνητής νοημοσύνης. Το σενάριο που ερευνήσαμε εδώ είναι μια ιατρική εφαρμογή με στόχο την καταπολέμηση της πανδημίας Covid-19. Η πανδημία Covid-19 είχε καταστρεπτικές συνέπειες τόσο στην κοινωνική ζωή όσο και στην παγκόσμια οικονομία, προκαλώντας αδιάκοπη αύξηση των καθημερινών κρουσμάτων και θανάτων. Ενώ οι ακτινογραφίες θώρακα λειτουργούν ως ευρέως διαθέσιμη και οικονομική μέθοδος διαλογής, ο τεράστιος όγκος των περιπτώσεων αναπνευστικών ασθενειών εμποδίζει τις γρήγορες δοκιμές και την έγκαιρη καραντίνα κάθε ασθενούς. Κατά συνέπεια, υπάρχει επιτακτική ανάγκη για μια αυτοματοποιημένη λύση, καθοδηγούμενη από την αφοσίωση της ερευνητικής κοινότητας. Σε απάντηση αυτής της ζήτησης, παρουσιάζουμε μια τοπολογία Βαθιού Νευρωνικού Δικτύου (DNN) σχεδιασμένη να κατηγοριοποιεί ακτινογραφίες θώρακα σε τρεις κατηγορίες: Covid-19, Ιογενής Πνευμονία και Φυσιολογική. Η ακριβής αναγνώριση των μολύνσεων από Covid-19 μέσω των ακτινογραφιών έχει ύψιστη σημασία, υποστηρίζοντας τους ιατρούς στα διαγνωστικά τους καθήκοντα. Ωστόσο, ο σημαντικός όγκος δεδομένων προς επεξεργασία καταναλώνει πολύτιμο χρόνο και υπολογιστικούς πόρους. Κάνοντας ένα σημαντικό βήμα μπροστά, υλοποιούμε και αναπτύσσουμε αυτό το Νευρωνικό Δίκτυο σε μια πλατφόρμα Xilinx Cloud FPGA. Αυτές οι συσκευές είναι γνωστές για την αξιοσημείωτη ταχύτητα και την ενεργειακή τους αποδοτικότητα. Ο τελικός στόχος είναι να παρέχουμε μια ιατρική λύση για νοσοκομεία, απλοποιώντας τις ιατρικές διαγνώσεις με ακρίβεια, ταχύτητα και χαμηλή ενεργειακή κατανάλωση. Μέχρι όσο γνωρίζουμε, αυτή η εφαρμογή δεν έχει εξερευνηθεί για FPGAs προηγουμένως. Αξιοσημείωτα, η επιτευχθείσα ακρίβεια και ταχύτητα ξεπερνούν οποιαδήποτε γνωστή υλοποίηση Νευρωνικών Δικτύων για την ανίχνευση Covid μέσω ακτινογραφιών. Συγκεκριμένα, το σύστημά μας κατηγοριοποιεί τις ακτινογραφίες με εντυπωσιακό ρυθμό 3600 καρέ ανά δευτερόλεπτο (FPS) με ακρίβεια 96.2%. Επιπλέον, ξεπερνά τις GPU με αύξηση ταχύτητας κατά 3.1× και ξεπερνά τις CPU με μια αξιοσημείωτη απόδοση κατά 17.6×. Όσον αφορά την ενεργειακή απόδοση, η πλατφόρμα FPGA υπερέχει, δείχνοντας βελτίωση κατά 4.6× σε σχέση με τις GPU και εντυπωσιακή βελτίωση κατά 13.1× σε σχέση με τις CPU.

Διαμορφώσαμε πολλές αρχιτεκτονικές Συνελικτικού Νευρωνικού Δικτύου (CNN) για την ταξινόμηση εικόνων ακτίνων X θώρακα (CXR) εντός του συνόλου δεδομένων. Η επιλογή του βέλτιστου μοντέλου τεχνητής νοημοσύνης βασίστηκε σε εκτιμήσεις τόσο της ακρίβειας όσο και της αποτελεσματικότητας, με την πρόθεση να αναπτυχθεί σε ένα FPGA, όπως διευκρινίστηκε στην επόμενη ενότητα. Αυτή η ενότητα είναι αφιερωμένη στην εμβάθυνση στη διαμόρφωση του προβλήματος, στο σύνολο δεδομένων που χρησιμοποιείται, στη διαδικασία εκπαίδευσης και στις βελτιστοποιήσεις με επίκεντρο το υλικό που εφαρμόζονται στα μοντέλα για τη διευκόλυνση της αποτελεσματικής ανάπτυξής τους στη συσκευή FPGA.

1. Σύνολο δεδομένων: Η βάση δεδομένων εικόνων με ακτίνες X Covid-19 που χρησιμοποιήθηκε σε αυτήν τη μελέτη επιμελήθηκε από την Ιταλική Εταιρεία Ιατρικής και Επεμβατικής Ακτινολογίας (SIRM) COVID-19 DATABASE [118]. Το σύνολο δεδομένων περιλαμβάνει συνολικά 2.905 εικόνες CXR, κατηγοριοποιημένες σε 219 για Covid, 1.345 για Ιογενή Πνευμονία και 1.341 για την τάξη Φυσιολογική, που χρησιμοποιούνται για την εκπαίδευση και την αξιολόγηση των μοντέλων AI. Παρά το σχετικά μέτριο μέγεθος και την παρατυπία του συνόλου δεδομένων, χρησιμοποιήσαμε διάφορες τεχνικές για να αντιμετωπίσουμε αυτές τις προκλήσεις, όπως αναλύεται στις επόμενες ενότητες.

2. Τοπολογία Μοντέλου: Προτείνουμε τρεις διαφορετικές τοπολογίες για αυτό το πρόβλημα προκειμένου να έχουμε καλύτερη αξιολόγηση στο σύνολο δεδομένων και να επιλέξουμε το καταλληλότερο μοντέλο για επιτάχυνση στο FPGA στη συνέχεια. Αναπτύξαμε ξεχωριστά μοντέλα που το καθένα έχει διαφορετική ακρίβεια πρόβλεψης, αρχιτεκτονική πολυπλοκότητα (από άποψη αριθμού παραμέτρων) και υπολογιστική πολυπλοκότητα (από άποψη αριθμού λειτουργιών MAC). Ένα CustomCNN που είναι ένα κλασικό νευρωνικό δίκτυο συνέλιξης, ένα lightResNet που είναι μια παραλλαγή ResNet50 και το DenseNetX που βασίζεται στην αρχιτεκτονική DenseNet αλλά περιλαμβάνει επίσης τα επίπεδα Bottleneck και τον παράγοντα συμπίεσης.

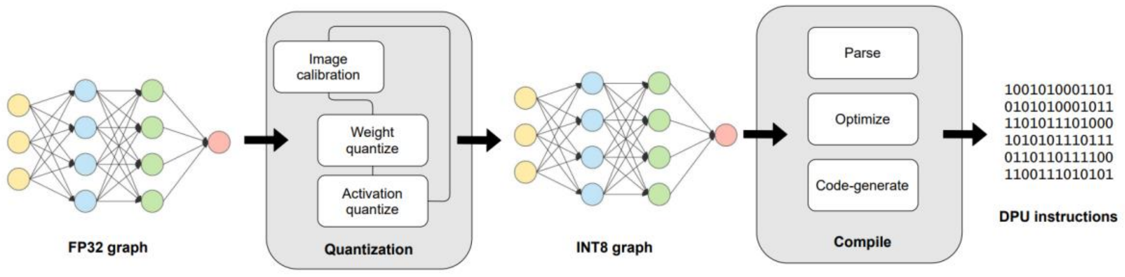
3. Εκπαίδευση: Τελευταία θα αναλύσουμε αρκετές τεχνικές που εφαρμόσαμε στη διαδικασία εκπαίδευσης. Το πρώτο σχετίζεται με τη στάθμιση της τάξης. Η εκπαίδευση με ένα σύνολο δεδομένων όπως το δικό μας με πολύ λίγες εικόνες Covid-19 σε αντίθεση με τις εικόνες φυσιολογικής ή ιογενούς πνευμονίας αποτελεί ένα πρόβλημα με ανισορροπία στην τάξη. Για να ξεπεραστεί η απόκλιση κλάσης, επιβάλαμε συγκεκριμένα βάρη κλάσεων (δηλαδή, 6× στην κατηγορία Covid) που εφαρμόστηκαν στην απώλεια του μοντέλου για κάθε δείγμα και τελικά βοήθησαν το μοντέλο να μάθει από τα μη ισορροπημένα δεδομένα. Στη συνέχεια εφαρμόζουμε βελτιστοποιήσεις με επίγνωση HW στη συλλογή μοντέλων. Η τοπολογία των CNN χρειαζόταν κάποιες μικρές τροποποιήσεις προκειμένου να είναι συμβατή και αποτελεσματική με τον κβαντιστή και τον μεταγλωττιστή Vitis AI. Συγκεκριμένα, η σειρά των επιπέδων ενεργοποίησης Batch Normalization (BN), Rectified Linear Unit (ReLU) και Convolution έχει αλλάξει από BN → ReLU → Conv σε Conv → BN → ReLU. Επίσης, μια άλλη βελτιστοποίηση που κάναμε είναι στην περίπτωση του GlobalAveragePooling2D, το οποίο χρειαζόμασταν για παράδειγμα στο DenseNetX και το αντικαταστήσαμε με το AveragePooling2D συν ένα επίπεδο Flatten. Τέλος, το softmax μπήκε στο DPU και όχι μέσω SW καθώς ήταν ταχύ.

Προσέγγιση επιτάχυνσης — Δεδομένης της πιθανής χρήσης της εφαρμογής μας από πολλούς χρήστες παγκοσμίως, μια αποτελεσματική και γρήγορη λύση είναι επιτακτική. Ως εκ τούτου, επιλέξαμε να αξιοποιήσουμε το περιβάλλον AI του Vitis για να αναπτύξουμε τα μοντέλα συνελικτικού νευρωνικού δικτύου (CNN) σε ένα Xilinx Alveo U50 FPGA. Αυτή η διαδικασία μεταγλώττισης δημιουργεί οδηγίες DPU (Deep Learning Processing Unit), διευκολύνοντας την αποτελεσματική χρήση των υπολογιστικών μονάδων (CUs) του FPGA.

1. Quantization: Αρχικά, μετατρέψαμε τα μοντέλα μας σε ένα παγωμένο γράφημα κινητής υποδιαστολής συμβατό με Tensorflow ως προϋπόθεση για τη διαδικασία κβαντοποίησης. Στη συνέχεια, επιλέξαμε την κβαντοποίηση των εκπαιδευμένων βαρών των Συνελικτικών Νευρωνικών Δικτύων μας (CNN) χρησιμοποιώντας ακρίβεια 8-bit. Τέλος, παράσχαμε ένα αντιπροσωπευτικό σύνολο δειγμάτων των δεδομένων εκπαίδευσης για τη βαθμονόμηση της διαδικασίας κβαντοποίησης.

2. Αξιολόγηση του κβαντισμένου μοντέλου: Ο μετασχηματισμός από ένα μοντέλο κινητής υποδιαστολής, όπου οι τιμές μπορούν να παρουσιάσουν ένα ευρύ δυναμικό εύρος, σε ένα μοντέλο 8-bit, που περιορίζει τις τιμές σε μία από τις 256 πιθανότητες, εισάγει εγγενώς μια μικρή απώλεια ακρίβειας. Ως εκ τούτου, ήταν κρίσιμο να αξιολογηθεί το κβαντισμένο γράφημα στο Tensorflow πριν προχωρήσουμε στη συλλογή του μοντέλου. Το κβαντισμένο γράφημα στο FPGA, σε αντίθεση με το γράφημα κινητής υποδιαστολής στην CPU, είχε ελάχιστη επίδραση στην τελική ακρίβεια (λιγότερο από 0,5%).

3. Σύνταξη μοντέλου: Στην τελική φάση, μεταγλωττίσαμε το γράφημα σε ένα σύνολο μικροεντολών που είναι ενθυλακωμένες σε μορφή αρχείου «.xmodel». Ο μεταγλωττιστής Vitis AI ανέλαβε τη μετατροπή και τη βελτιστοποίηση του κβαντοποιημένου μοντέλου ανάπτυξης, με αποτέλεσμα τη δημιουργία του τελικού «εκτελέσιμου» για το συμπέρασμα του CNN.



Σχήμα 13. Κβαντισμός γραφήματος CNN και μεταγλώττιση για το FPGA DPU

Αξιολόγηση μοντέλου — Σε αυτή τη μελέτη, διεξήχθησαν πειράματα χρησιμοποιώντας Tensorflow και Keras, χρησιμοποιώντας τις κοινές διαστάσεις εικόνας 224 × 224 τυπικές σε πολλά Συνελικτικά Νευρωνικά Δίκτυα (CNN). Όλα τα μοντέλα υποβλήθηκαν σε εκπαίδευση με το Adam optimizer, συνοδευόμενη από EarlyStopping. Η βελτιστοποίηση των μοντέλων ταξινόμησης επιτεύχθηκε μέσω της ελαχιστοποίησης της συνάρτησης απώλειας διασταυρούμενης εντροπίας. Επιπλέον, διάφορες παράμετροι και υπερπαράμετροι για κάθε μοντέλο υποβλήθηκαν σε συντονισμό κατά τη διάρκεια της εκπαίδευσης, συμπεριλαμβανομένου του Ρυθμού Μάθησης (LR) και των εποχών. Ο παρακάτω πίνακας παρέχει μια επισκόπηση των βασικών χαρακτηριστικών κάθε μοντέλου, περιλαμβάνοντας υπερπαραμέτρους εκπαίδευσης, προδιαγραφές μοντέλου και μετρήσεις αξιολόγησης μοντέλου.

	Hypermarameters		Model Specs		Evaluation	
Model	**LR**	**Epochs**	**Params**	**FLOPs**	**Accuracy**	**Loss**
CustomCNN	0.0001	70	2.033G	1.025G	96.2%	0.16
lightResNet	0.001	60	2.697G	2.814G	96.5%	0.408
DenseNetX	0.005	80	0.758G	1.722G	94.9%	0.264

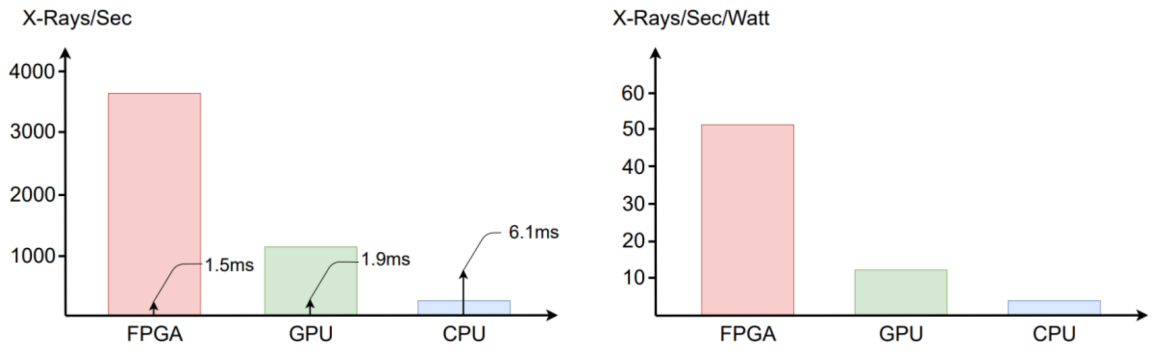
Πίνακας 5. Χαρακτηριστικά μοντέλων και ακρίβεια

Απόδοση συστήματος — Κατά την αξιολόγηση του σχεδιασμού του συστήματος, επαληθεύσαμε αρχικά τη χρήση πόρων της μονάδας επεξεργασίας βαθιάς μάθησης (DPU) του FPGA. Η διαμόρφωση υλικού για ανάπτυξη περιελάμβανε ένα Xilinx Alveo U50 Cloud FPGA με χωρητικότητα 8 GB High Bandwidth Memory (HBM) και συνολικό εύρος ζώνης 316 GB/s. Η συσκευή ενσωματώθηκε σε μια εγκατάσταση Gen4x8 PCI Express, που λειτουργεί σε συχνότητα ρολογιού πυρήνα 300 MHz. Ο παρακάτω πίνακας παρέχει μια επισκόπηση της χρήσης πόρων για έναν πυρήνα DPUv3E.

	Utilization Summary				
Name	**BRAM**	**URAM**	**DSP**	**FF**	**LUT**
Used	628	320	2600	310752	250290
Percentage	46.7%	50%	43.6%	21.2%	28.7%

Πίνακας 6. Χρήση πόρων για ένα πυρήνα DPU

Στη συνέχεια, πραγματοποιήσαμε μια αξιολόγηση συμπερασμάτων χρησιμοποιώντας το μοντέλο CustomCNN σε εναλλακτικά συστήματα υψηλής απόδοσης, συγκεκριμένα μια GPU Nvidia V100 και μια 10πύρηνη Intel Xeon Silver 4210. Το inference σε αυτές τις εναλλακτικές συσκευές πραγματοποιήθηκε χρησιμοποιώντας το Tensorflow με προεπιλεγμένες ρυθμίσεις. Η αριστερή πλευρά του παρακάτω σχήματος απεικονίζει τη μέγιστη απόδοση που επιτυγχάνεται από κάθε συσκευή, μετρημένη σε ακτίνες X ανά δευτερόλεπτο (FPGA: 3600, GPU: 1157, CPU: 204). Επιπλέον, σχολιάσαμε τον χρόνο, μετρούμενο σε χιλιοστά του δευτερολέπτου, για εξαγωγή μεμονωμένων εικόνων ακτίνων X σε κάθε συσκευή. Επιπλέον, η δεξιά πλευρά του σχήματος απεικονίζει τη μέτρηση απόδοσης ισχύος για κάθε συσκευή σε ακτίνες X/δευτερόλεπτα/Watt (FPGA: 51.3, GPU: 11.1, CPU: 3.9).



Σχήμα 14. Επιτάχυνση και απόδοση μεταξύ διάφορων αρχιτεκτονικών

Συμπέρασμα — Σε αυτή τη μελέτη, εισαγάγαμε πολλαπλά μοντέλα τεχνητής νοημοσύνης, καθένα από τα οποία διαθέτει ξεχωριστά χαρακτηριστικά, σχεδιασμένα για την ανίχνευση περιπτώσεων COVID-19 από εικόνες CXR. Κβαντίσαμε, μεταγλωττίσαμε και επιταχύναμε το μοντέλο AI για ανάπτυξη σε ένα Alveo U50 FPGA, με στόχο να επιταχύνουμε τον έλεγχο με τη βοήθεια υπολογιστή. Η εφαρμογή ήταν σε docker κοντέινερ, επιτρέποντας απρόσκοπτη φορητότητα σε ένα σύμπλεγμα FPGA που λειτουργεί στο cloud, παρουσιάζοντας υψηλή απόδοση και ενεργειακή απόδοση σε σύγκριση με άλλες αρχιτεκτονικές. Είναι σημαντικό να σημειωθεί ότι αυτή δεν είναι μια έτοιμη για παραγωγή λύση που προορίζεται για αυτοδιάγνωση. Από ερευνητικής σκοπιάς, η εστίασή μας παραμένει στη βελτίωση της απόδοσης και στην εισαγωγή πρόσθετων λειτουργιών στο πλαίσιο της AI Health, ιδιαίτερα καθώς συλλέγονται νέα δεδομένα. Αυτό μπορεί να περιλαμβάνει τομείς όπως η διαστρωμάτωση κινδύνου για ανάλυση επιβίωσης ή η πρόβλεψη της διάρκειας νοσηλείας. Ενώ η σφαίρα των αυτοματοποιημένων συστημάτων τεχνητής νοημοσύνης είναι τεράστια, αυτή η εργασία ρίχνει φως στις πιθανές συνεισφορές των FPGA στη θεμελιώδη διαμόρφωση της Ιατρικής Διάγνωσης με τη βοήθεια υπολογιστή.

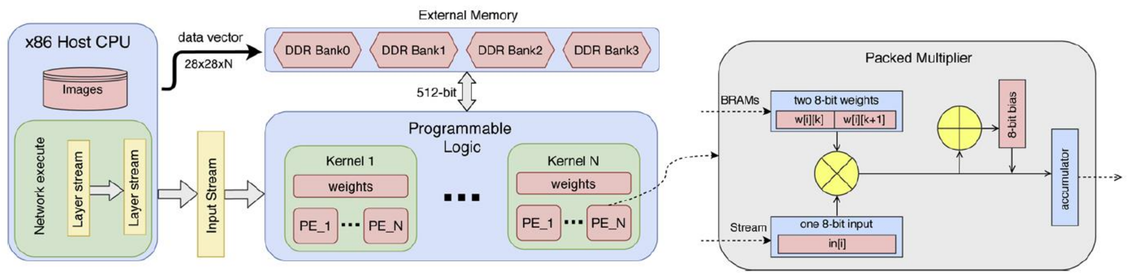
Παράγωντας αυτόματα υλικολογισμικό για FPGA από CNN

Αυτή η ενότητα περιέχει το τέταρτο από τα τέσσερα σενάρια στα οποία χρησιμοποιήσαμε FPGA για επιτάχυνση AI. Συγκεκριμένα, αυτή η ενότητα δεν παρουσιάζει μια συγκεκριμένη εφαρμογή τεχνητής νοημοσύνης, αλλά εισάγει έναν πιο γενικευμένο τρόπο και βελτιστοποιήσεις για την επιτάχυνση των εφαρμογών τεχνητής νοημοσύνης για FPGA. Συγκεκριμένα, περιγράφει ένα αποτελεσματικό περιβάλλον για τη μετατροπή εκπαιδευμένων μοντέλων CNN σε βελτιστοποιημένο υλικολογισμικό FPGA. Οι αποτελεσματικοί αλγόριθμοι τεχνητής νοημοσύνης έχουν τεράστια σημασία σε πολλές εφαρμογές, ιδιαίτερα σε εργασίες που περιλαμβάνουν ταξινόμηση ή ομαδοποίηση. Ωστόσο, είναι απαραίτητο ένα τυποποιημένο καθολικό μοντέλο AI και μια εύκολη βελτιστοποίηση. Η ενοποίηση διαφόρων μοντέλων μηχανικής μάθησης σε ένα κοινό οικοσύστημα μπορεί να μειώσει σημαντικά τον χρόνο ανάπτυξης και να βελτιώσει τη συμβατότητα μεταξύ διαφόρων frameworks. Το Open Neural Network Exchange Format (ONNX) αποτελεί μια ευρέως αναγνωρισμένη ανοιχτή μορφή για την αναπαράσταση μοντέλων βαθιάς μάθησης. Σκοπός του είναι να επιτρέψει ομαλότερη μετάβαση μοντέλων μεταξύ εργαλείων αιχμής για προγραμματιστές τεχνητής νοημοσύνης. Συγκεκριμένα, οι εταιρείες υλικού όπως η Nvidia και η Intel προσπαθούν να παραμείνουν ευθυγραμμισμένες με αυτήν την τάση. Παράγουν χρόνους εκτέλεσης υλικού βελτιστοποιημένους για CPU και GPU που διαχειρίζονται επιδέξια αυτά τα μοντέλα AI ανοιχτής μορφής όπως το ONNX. Αυτό δίνει τη δυνατότητα στους προγραμματιστές να αξιοποιήσουν μια ποικιλία από υλικό και να χρησιμοποιήσουν τα προτιμώμενα πλαίσια AI. Ωστόσο, τα FPGA αποτελούν μια πιο περίπλοκη πρόκληση. Αποτελούν μια αποδεδειγμένη πλατφόρμα για την αποτελεσματική αντιμετώπιση τέτοιων προκλήσεων σχετικά με την απόδοση και την απόδοση ισχύος. Η μελέτη μας βασίζεται σε ένα έργο ανάπτυξης πρώιμου σταδίου γνωστό ως HLS4ML [3], που αρχικά σχεδιάστηκε για εφαρμογές σωματιδιακής φυσικής. Η βασική καινοτομία του έργου περιλαμβάνει την αυτόματη δημιουργία νευρωνικών δικτύων (NN) για ενσωματωμένα Xilinx FPGA. Η δουλειά μας κάνει ένα βήμα παραπέρα ενσωματώνοντας εκπαίδευση NN με γνώση του υλικού και μια ολοκληρωμένη στρατηγική βελτιστοποίησης πάνω στο HLS4ML. Αυτή η επέκταση βελτιώνει σημαντικά την απόδοση και την ενεργειακή απόδοση της βιβλιοθήκης. Επιπλέον, εισάγει λειτουργικότητα για ανάπτυξη υλικολογισμικού Cloud FPGA από οποιοδήποτε μοντέλο NN. Η μεθοδολογία μας ξεκινά με εκπαίδευση μοντέλου στον Keras με επίγνωση FPGA, προσαρμοσμένη για αναγνώριση εικόνας. Στη συνέχεια, το μοντέλο μετατρέπεται σε ανοιχτή μορφή ONNX πριν προσαρμοστεί και τελειοποιηθεί για τα Cloud FPGA. Αυτή η διαδικασία χρησιμοποιεί ένα νέο σχήμα που βελτιστοποιεί διάφορες πτυχές, συμπεριλαμβανομένου του CPU, της διαχείρισης μνήμης και των λειτουργιών του πυρήνα. Επίσης, αξιοποιούνται πολλαπλά επίπεδα ακρίβειας δικτύου. Από όσο γνωρίζουμε, αυτή η προσέγγιση αποτελεί μοναδική καινοτομία. Οδηγεί σε αξιοσημείωτη επιτάχυνση, επιτυγχάνοντας κέρδη απόδοσης έως και 102× σε σύγκριση με μία μόνο CPU και έως και 5,5× βελτιώσεις στην απόδοση ανά watt σε σύγκριση με τις GPU.

Τηρούμε τις αρχές σχεδιασμού FPGA για τη βελτιστοποίηση της υψηλής απόδοσης και της απόδοσης ισχύος, που συνοδεύεται από την ανάπτυξη ενός προσαρμοσμένου API OpenCL που έχει σχεδιαστεί για να φιλοξενεί Cloud FPGA για κέντρα δεδομένων. Η ημι-αυτοματοποιημένη μεθοδολογία ροής εργαλείων με επίκεντρο το OpenCL για την ανάπτυξη νευρωνικών δικτύων σε FPGA υποστηρίζει διάφορες βελτιστοποιήσεις λογισμικού και υλικού. Ξεκινάμε με την εκπαίδευση ενός νευρωνικού δικτύου ειδικά για το υλικό, που εφαρμόζεται σε ένα ευρύ φάσμα μοντέλων νευρωνικών δικτύων. Αυτή η προσέγγιση προσφέρει ουσιαστική ευελιξία στην παραλληλοποίηση, ιδιαίτερα επωφελής για unsigned βάρη νευρωνικών δικτύων. Επιπλέον, η μετάφραση της γενικευμένης μορφής μοντέλου ONNX σε ένα μοντέλο συγχωνευμένου υλικού με ανεξάρτητες μονάδες για κάθε επίπεδο παρέχει ευκαιρίες για περαιτέρω παραμετροποίηση και βελτιστοποίηση του νευρωνικού δικτύου. Τέλος, μέσω του OpenCL host API, ενισχύουμε την πρόσβαση στη μνήμη στους πυρήνες, μεγιστοποιώντας το εύρος ζώνης δεδομένων της συσκευής. Η χρήση ουρών εντολών OpenCL και ο ακριβής συγχρονισμός κεντρικού πυρήνα επιτρέπει την ουσιαστική παραλληλοποίηση σε χονδρόκοκκο επίπεδο.

Τελικός σχεδιασμός συστήματος με χρήση OpenCL — Μετά την επιτυχή εφαρμογή ενός σχεδιασμού χαμηλής καθυστέρησης, ο οποίος αξιοποίησε τον παραλληλισμό σε επίπεδο πυρήνα εντός του PL, προχωρήσαμε στη δημιουργία ενός αποτελεσματικού OpenCL κεντρικού API χρησιμοποιώντας τυπικές κλήσεις OpenCL API. Συγκεκριμένα, κατανείμαμε εισόδους εικόνας χρησιμοποιώντας διανύσματα C++, με κάθε διάνυσμα να έχει μέγεθος 28 * 28 * N, όπου το N αντιπροσωπεύει τον αριθμό των εικόνων όταν προτιμάται η ομαδική επεξεργασία. Αυτή η μέθοδος κατανομής εξασφάλισε ότι η εικόνα καταλάμβανε συνεχόμενη μνήμη, επιτρέποντας τη χρήση των πιο αποτελεσματικών μηχανισμών μεταφοράς δεδομένων προς και από DDR. Για τη βελτιστοποίηση της απόδοσης δεδομένων, εφαρμόσαμε μια διεπαφή χρήστη 512-bit σε κάθε πλευρά του πυρήνα, αξιοποιώντας το μέγιστο εύρος ζώνης μνήμης που υποστηρίζεται από τα Xilinx OpenCL FPGA. Ταυτόχρονα, χρησιμοποιήσαμε όλα τα διαθέσιμα DDR στη συσκευή, επιτυγχάνοντας μέγιστες ταχύτητες μεταφοράς δεδομένων. Επιπλέον, δημιουργήσαμε προσεκτικά την ταυτόχρονη λειτουργία στις εντολές OpenCL για την εκκίνηση του πυρήνα και τις συγχρονισμένες αλληλεπιδράσεις μεταξύ του κεντρικού υπολογιστή και των πυρήνων, διασφαλίζοντας την ομαλή λειτουργία μέσα σε ένα παράδειγμα σταθερής ροής δεδομένων.

Επιπλέον, λάβαμε μέτρα για την ελαχιστοποίηση της διέλευσης SLR όποτε είναι εφικτό, καθώς αυτό τείνει να έχει ως αποτέλεσμα λιγότερο αποδοτικούς σχεδιασμούς όσον αφορά την καθυστέρηση και την ισχύ λόγω της δημιουργίας μακρύτερων κρίσιμων διαδρομών. Ευτυχώς, οι πυρήνες μας παρέμειναν γενικά εντός των ορίων πόρων κάθε SLR. Το σχήμα παρακάτω παρέχει μια επισκόπηση ολόκληρου του συστήματος, ξεκινώντας από τον κεντρικό υπολογιστή CPU, που διασχίζει το FPGA και τελειώνει στους υπολογισμούς που εκτελούνται σε επίπεδο πολλαπλασιαστή σε κάθε Στοιχείο Επεξεργασίας (PE). Συγκεκριμένα, το τελικό σύστημα, πλήρες με όλες τις βελτιστοποιήσεις που περιγράφηκαν, ενσωματώθηκε στο πακέτο HLS4ML, προσβάσιμο μέσω του Python API, υλοποιώντας έτσι την προσαρμοσμένη αρχιτεκτονική μας.



Σχήμα 15. Τελικό ολοκληρωμένο σύστημα

Απόδοση επιταχυντή — Για την αξιολόγηση του σχεδιασμού, η αρχική μας εστίαση ήταν στην επαλήθευση της ορθότητας του επιταχυντή, ιδιαίτερα του προσαρμοσμένου πολλαπλασιαστή μας, καθώς εσφαλμένες έξοδοι DSP θα μπορούσαν ενδεχομένως να διαταράξουν ολόκληρο το συνελικτικό νευρωνικό δίκτυο. Πραγματοποιήσαμε την αξιολόγηση υλικού σε ένα FPGA Xilinx Alveo U200. Αυτό το σύστημα διαθέτει 64 GB μνήμης RAM εκτός τσιπ με εύρος ζώνης 77 GB/s και λειτουργεί σε διεπαφή Gen3x16 PCI Express, που λειτουργεί με ταχύτητα ρολογιού πυρήνα 300 MHz. Ο πίνακας παρακάτω παρέχει λεπτομερή δεδομένα χρήσης πόρων και χρονισμού για κάθε επίπεδο νευρωνικού δικτύου, κατηγοριοποιημένα ανά τύπο στρώματος, στην περίπτωση του μοντέλου MNIST.

	Utilization summary				**Timing**	
Layer	BRAM	DSP	FF	LUT	Latency (cycles)	FPS
Dense	192	64	2449	12656	1709	-
ReLu	0	0	16	127	130	-
Softmax	7	0	985	2401	51	-
Total	199	64	3450	15184	1890	158K

Πίνακας 7. Χρήση πόρων και χρόνοι απόκρισης ανά επίπεδο.

Τελική απόδοση του συστήματος — Στην τελική αξιολόγηση του συστήματός μας, ξεκινήσαμε δοκιμές για τον προσδιορισμό του χρόνου εκτέλεσης από άκρο σε άκρο που απαιτείται για ένα μεμονωμένο πέρασμα προς τα εμπρός μέσω των δύο νευρωνικών μας δικτύων, δηλαδή του MNIST και του CIFAR, που περιλαμβάνει μεταφορές μνήμης στη διαδικασία. Στη συνέχεια, συγκρίναμε αυτά τα αποτελέσματα με τους χρόνους εκτέλεσης που επιτεύχθηκαν σε μια CPU Xeon μονού πυρήνα και μια GPU Nvidia Tesla P100, με την παρατήρηση ότι οι χρονισμοί για τα δύο νευρωνικά δίκτυα μπορεί να φαίνονται παρόμοιοι λόγω των επιβαρύνσεων της συσκευής. Όπως υποδεικνύεται στον πίνακα παρακάτω, το τελικό μας σύστημα FPGA υπερέχει όσον αφορά την απόδοση και δείχνει χαμηλότερη μέση κατανάλωση ενέργειας, ειδικά για το μικρότερο μοντέλο MNIST. Αυτή η υπεροχή στην απόδοση και την αποτελεσματικότητα μπορεί να αποδοθεί σε διάφορους παράγοντες. Πρώτον, το σύστημά μας χρησιμοποιεί βάρη και ενεργοποιήσεις μειωμένης ακρίβειας, σε αντίθεση με τις προεπιλεγμένες αναπαραστάσεις κινητής υποδιαστολής 32-bit. Σε συνδυασμό με τη χρήση ενός προσαρμοσμένου πολλαπλασιαστή, το σύστημά μας επιτυγχάνει ελάχιστη καθυστέρηση, ένα κρίσιμο χαρακτηριστικό για εφαρμογές που απαιτούν γρήγορη επεξεργασία. Επιπλέον, αξίζει να σημειωθεί ότι ο σχεδιασμός μας χρησιμοποιεί μόνο ένα μικρό κλάσμα των διαθέσιμων πόρων της συσκευής, καθιστώντας το εξαιρετικά αποδοτικό από πλευράς ενέργειας. Κατά συνέπεια, μπορεί να αναπτυχθεί όχι μόνο σε κέντρα δεδομένων αλλά και σε μικρότερα ενσωματωμένα FPGA SoC, καλύπτοντας κρίσιμες εφαρμογές που απαιτούν χαμηλό χρόνο και ενεργειακή απόδοση.

Device Information			**Performance Evaluation**				**Power Evaluation**	
System	Model	Architecture	CIFAR	Speed-up	MNIST	Speed-up	Watt(avg)	Perf./Watt (max)
CPU	Xeon 2.4 GHz	22-nm	58 ms	1 ×	43 ms	1 ×	9 W *	1 ×
GPU	Nvidia P100	16-nm	1.1 ms	52.7 ×	0.75 ms	57 ×	95 W	5.4 ×
FPGA	Alveo U200	16-nm	2.7 ms	21.5 ×	0.42 ms	102.3 ×	31 W	29.7 ×

Πίνακας 8. Σύγκριση με άλλες αρχιτεκτονικές.

Συμπέρασμα — Σε αυτή τη μελέτη, βελτιώσαμε ένα νέο και ανανεωμένο πλαίσιο σχεδιασμένο για να δημιουργεί αυτόνομα υλικολογισμικό FPGA χρησιμοποιώντας Σύνθεση υψηλού επιπέδου (HLS) που βασίζεται σε μοντέλα νευρωνικών δικτύων. Υλοποιήσαμε διάφορες βελτιστοποιήσεις και επεκτείναμε τη λειτουργικότητα του υπάρχοντος πακέτου hls4ml, ενισχύοντας την ταχύτητα και την αποτελεσματικότητά του. Για επεξηγηματικούς σκοπούς, εκπαιδεύσαμε σχολαστικά και τελειοποιήσαμε δύο προσαρμοσμένα νευρωνικά δίκτυα, ένα μικρότερο και ένα μεγαλύτερο σε μέγεθος, με βελτιστοποιήσεις προσαρμοσμένες στην επιτάχυνση υλικού. Τα ευρήματα της έρευνάς μας έδειξαν ότι η προτεινόμενη αρχιτεκτονική μπορεί να ξεπεράσει την απόδοση και την απόδοση ισχύος άλλων πλατφορμών υψηλής τεχνολογίας, όπως CPU ή GPU. Από ερευνητική άποψη, εργαζόμαστε ενεργά για την περαιτέρω βελτίωση της απόδοσης και την ενσωμάτωση πρόσθετων λειτουργιών στο πακέτο. Ο τομέας των πιθανών σχεδιασμών είναι εκτεταμένος και η έρευνά μας συνεισφέρει πολύτιμες γνώσεις σε αυτόν τον τομέα, επιτυγχάνοντας αξιοσημείωτα αποτελέσματα.

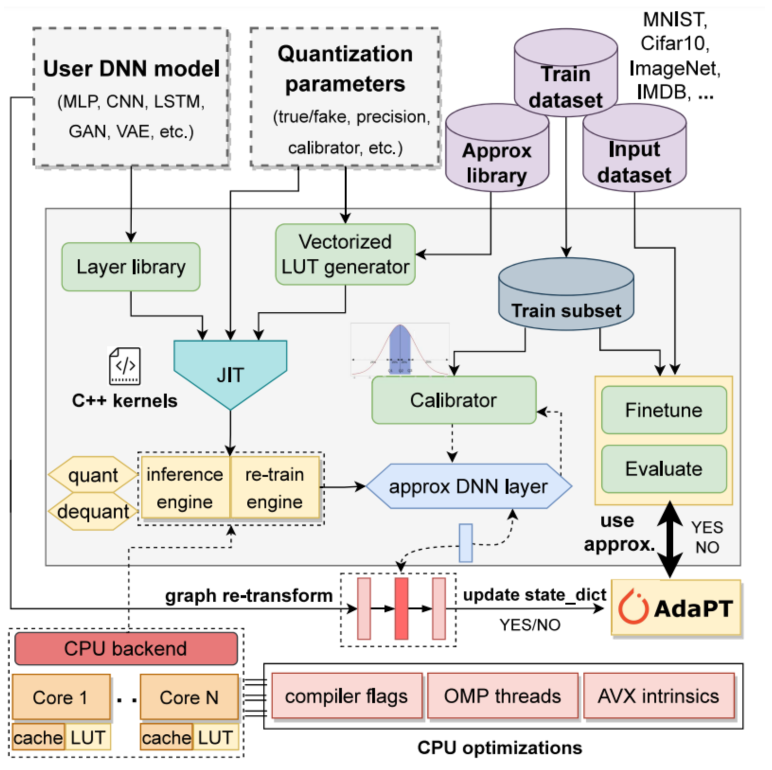
Προσομοίωση AI για κατά προσέγγιση υλικό

Στο προηγούμενο κεφάλαιο είδαμε πώς μπορούμε να βελτιστοποιήσουμε εφαρμογές AI/ML για επιταχυντές υλικού όπως τα FPGA. Το κατά προσέγγιση υλικό, από την άλλη πλευρά, περιλαμβάνει ανταλλαγή από κάποιο επίπεδο ακρίβειας στους υπολογισμούς για να έχουμε οφέλη απόδοσης ή ενεργειακής απόδοσης. Οι φόρτος εργασίας της τεχνητής νοημοσύνης, ιδιαίτερα στη βαθιά μάθηση, είναι συχνά ανεκτικοί σε μικρά λάθη ή προσεγγίσεις στους υπολογισμούς. Τα κατά προσέγγιση σχέδια υλικού μπορούν να εκμεταλλευτούν αυτήν την ανοχή χρησιμοποιώντας αριθμητικές τεχνικές μειωμένης ακρίβειας ή άλλες τεχνικές προσέγγισης για να επιταχύνουν τους υπολογισμούς τεχνητής νοημοσύνης θυσιάζοντας την ελάχιστη ακρίβεια. Ο υπολογισμός σε αυτές τις αρχιτεκτονικές ονομάζεται κατά προσέγγιση υπολογισμός και μπορεί να βοηθήσει στη μείωση του χρόνου, της κατανάλωσης ενέργειας ή πόρων, τα οποία είναι κρίσιμα ζητήματα στην τεχνητή νοημοσύνη, ειδικά για συσκευές αιχμής με περιορισμένους πόρους. Ωστόσο, η περίοδος ανάπτυξης για αυτές τις συσκευές υλικού είναι μακρά, επομένως ο προσδιορισμός του σφάλματος ή του αντίκτυπου του κατά προσέγγιση υλικού σε ένα μοντέλο τεχνητής νοημοσύνης χωρίς να έχουμε ακόμη το υλικό μπορεί να είναι δύσκολος. Μια μέθοδος είναι η χρήση εργαλείων προσομοίωσης και βιβλιοθηκών λογισμικού που επιτρέπουν την εφαρμογή κατά προσέγγιση υπολογιστικών τεχνικών στους υπολογισμούς ενός μοντέλου AI. Ωστόσο, τα δημοφιλή πλαίσια DNN δεν υποστηρίζουν κατά προσέγγιση αριθμητική, επειδή υποστηρίζονται εγγενώς μόνο βιβλιοθήκες ακριβών μαθηματικών συναρτήσεων, επομένως η εξομοίωση γίνεται εξαιρετικά αργή. Σε αυτό το κεφάλαιο, διατυπώνουμε, επικυρώνουμε και αξιολογούμε ένα πλαίσιο για την εξομοίωση κατά προσέγγιση DNN για την αντιμετώπιση αυτής της πρόκλησης. Αυτό μας επέτρεψε να εκτιμήσουμε τον αντίκτυπο της προσέγγισης στην ακρίβεια και την ισχύ για πολλούς κατά προσέγγιση πολλαπλασιαστές στο δημοφιλές πλαίσιο Pytorch. Περιγράφουμε πώς το πλαίσιο μας κατασκευάστηκε για να παρέχει υποστήριξη επιτάχυνσης τόσο για CPU όσο και για GPU για αυθαίρετους κατά προσέγγιση πολλαπλασιαστές και μοντέλα νευρωνικών δικτύων, προκειμένου να εκτελούνται γρήγορα προσεγγιστικό inference. Τέλος, εισάγουμε έναν αλγόριθμο Monte Carlo Tree Search (MCTS) για την αποτελεσματική αναζήτηση του χώρου των πιθανών διαμορφώσεων χρησιμοποιώντας μια χειροκίνητη πολιτική που βασίζεται στο υλικό, επιτρέποντάς μας να αντλήσουμε βέλτιστες λύσεις που πλησιάζουν τη καμπύλη pareto.

AdaPT: Λειτουργία και τεχνικές βελτιστοποίησης

Εμπνευστήκαμε το πλαίσιο AdaPT ως ένα εργαλείο εξομοίωσης προσέγγισης DNN σε πολλαπλά επίπεδα, που παρουσιάζεται ως πρόσθετο στο PyTorch. Οι χρήστες έχουν την ευελιξία να το ενεργοποιήσουν ή να το απενεργοποιήσουν, επιστρέφοντας στην προεπιλεγμένη ροή του PyTorch όταν χρειάζεται. Το πλαίσιο υποστηρίζει απρόσκοπτα ένα ευρύ φάσμα επιπέδων και αρχιτεκτονικών μοντέλων. Προσφέρουμε υποστήριξη για δύο βασικές τεχνικές για τη βελτίωση της ακρίβειας: κβαντοποίηση μετά την εκπαίδευση με τεχνική καλιμπραρίσματος και επανεκπαίδευση κατά προσέγγιση. Οι χρήστες έχουν την ελευθερία να επιλέξουν μια κατά προσέγγιση υπολογιστική μονάδα (ACU) για ενσωμάτωση στο AdaPT ως αυτοτελές στοιχείο ή να τηρήσουν την προεπιλεγμένη ακριβή ροή του PyTorch. Επιπλέον, το AdaPT είναι εξοπλισμένο για να χειρίζεται μικτή ακρίβεια και μικτή προσέγγιση, επιτρέποντας τη χρήση διαφορετικών ACU μεταξύ των επιπέδων. Ωστόσο, είναι σημαντικό να σημειωθεί ότι η ακριβής προσέγγιση, όπως η προσέγγιση ανά φίλτρο, δεν υποστηρίζεται επί του παρόντος. Για να επιταχύνει την κατά προσέγγιση εξομοίωση DNN, το AdaPT αξιοποιεί τη δύναμη των OpenMP threads και των εγγενών εντολών Intel AVX2 για προηγμένη διανυσματοποίηση.

Λειτουργία πλαισίου — Η λειτουργία του πλαισίου AdaPT απεικονίζεται στο παρακάτω σχήμα. Αρχικά, ο χρήστης διαμορφώνει το επιθυμητό μοντέλο DNN, καθορίζοντας παραμέτρους κβαντοποίησης, όπως η ακρίβεια και ο βαθμονομητής που θα χρησιμοποιηθεί. Επιπλέον, ο χρήστης ορίζει την κατά προσέγγιση λειτουργική μονάδα που θα χρησιμοποιηθεί από τη βιβλιοθήκη, μαζί με το σύνολο δεδομένων για τα μοντέλα DNN. Είναι σημαντικό να σημειωθεί ότι για το σύνολο δεδομένων εκπαίδευσης, απαιτείται μόνο ένα αντιπροσωπευτικό υποσύνολο, το οποίο συνήθως αποτελεί περίπου το 10% του αρχικού συνόλου εκπαίδευσης, κυρίως για λόγους βαθμονόμησης. Στη συνέχεια, το AdaPT προσδιορίζει τα υποστηριζόμενα επίπεδα εντός του DNN και ανακτά την κατάλληλη κλάση επιπέδου από τη βιβλιοθήκη των προσεγγιστικών επιπέδων του. Για τον κατά προσέγγιση πολλαπλασιαστή, το αντίστοιχο Look Up Table (LUT) δημιουργείται από τη γεννήτρια LUT του AdaPT, οργανωμένη ως πίνακας C. Αυτός ο σχεδιασμός επιτρέπει στους πυρήνες της CPU να έχουν αποτελεσματική πρόσβαση σε δεδομένα από το ίδιο τμήμα της κρυφής μνήμης. Επιπλέον, χρησιμοποιείται ένα πρόσθετο εργαλείο για τη μετάφραση μιας περιγραφής υλικού σε μια συνάρτηση C. Σε περιπτώσεις όπου μεγαλύτερα πλάτη bit μπορεί να αυξήσουν σημαντικά τα μεγέθη LUT, το AdaPT μπορεί να αντικαταστήσει τον πολλαπλασιασμό που βασίζεται σε LUT με πολλαπλασιασμό βάσει συνάρτησης, όπου ο κατά προσέγγιση πολλαπλασιαστής αναπαρίσταται στον κώδικα C. Ενώ αυτή η προσέγγιση μπορεί να μετριάσει τις προκλήσεις που αφορούν τη μνήμη που σχετίζονται με μεγάλα LUT (μεγαλύτερα από 15 bit), μπορεί να εισάγει επιβάρυνση στο χρόνο εκτέλεσης του DNN. Είναι σημαντικό ότι και οι δύο προσεγγίσεις παρέχουν μια ισοδύναμη αναπαράσταση υψηλού επιπέδου της ACU, διασφαλίζοντας σταθερά αποτελέσματα κατά την κβαντοποίηση ή την επανεκπαίδευση. Το AdaPT στοχεύει να συμπληρώσει την κρυφή μνήμη των πυρήνων της CPU με LUT όσο το δυνατόν περισσότερο για να ελαχιστοποιήσει τις αστοχίες της κρυφής μνήμης. Τέλος, η μεταγλώττιση just-in-time (JIT) φορτώνει δυναμικά τα μεταγλωττισμένα επίπεδα χρησιμοποιώντας το σύστημα κατασκευής Ninja. Οι παραγόμενες μηχανές συμπερασμάτων και επανεκπαίδευσης συνδέονται στη συνέχεια με τα τελικά κατά προσέγγιση επίπεδα DNN, τα οποία αντικαθιστούν τα αντίστοιχα αρχικά επίπεδα PyTorch μέσω ενός εργαλείου επαναμετασχηματισμού γραφήματος του νευρωνικού. Αυτό το εργαλείο αναλύει τα επίπεδα και αντικαθιστά αναδρομικά τα επίπεδα PyTorch με τα κατά προσέγγιση ισοδύναμά τους. Τελικά, οι χρήστες έχουν την επιλογή να προσαρμόσουν το μοντέλο χρησιμοποιώντας το παρεχόμενο υποσύνολο εκπαίδευσης για να επιτύχουν ακόμη μεγαλύτερη ακρίβεια ή να προχωρήσουν σε κατά προσέγγιση αξιολόγηση.



Σχήμα 16. Λειτουργία του AdaPT.

TransAxx: Λειτουργία και τεχνικές βελτιστοποίησης

Προηγουμένως, περιγράψαμε ένα πλαίσιο για την εξομοίωση κατά προσέγγιση DNN χρησιμοποιώντας CPU. Τώρα, παρουσιάζουμε μια νέα πλατφόρμα που αξιοποιεί τη δύναμη της επιτάχυνσης της GPU μέσω των πυρήνων CUDA και του μοντέλου προγραμματισμού CUDA. Αυτό το εργαλείο που ονομάζεται TransAxx περιλαμβάνει όλα τα βασικά χαρακτηριστικά του AdaPT, συμπεριλαμβανομένης της κατά προσέγγιση επανεκπαίδευσης, της υποστήριξης αυθαίρετων πολλαπλασιαστών, διαφόρων επιπέδων ακρίβειας και διαφορετικών αρχιτεκτονικών μοντέλων. Συγκεκριμένα, κατασκευάστηκε με στόχο να εξομοιώσει Vision Transformers (εξ ου και το όνομα), το πρώτο στον τομέα των πλαισίων προσομοίωσης DNN, αλλά μπορεί επίσης να χρησιμοποιηθεί και με διάφορα μοντέλα CNN. Επιπλέον, διαθέτει πιο βελτιωμένο σχεδιασμό, επιτρέποντας την αυτόματη χρήση προεκπαιδευμένων μοντέλων χωρίς να απαιτείται χειροκίνητη παρέμβαση στον κώδικα του μοντέλου από την οπτική γωνία του χρήστη. Η εκτέλεσή του σε GPU έχει ως αποτέλεσμα αυξημένη ταχύτητα που επιτρέπει την εξομοίωση μεγάλων μοντέλων τεχνητής νοημοσύνης, εάν χρειαστεί, χωρίς σημαντικό χρόνο εκτέλεσης, διατηρώντας παράλληλα μια πολύ φιλική προς το χρήστη διεπαφή. Χρησιμοποιώντας το TransAxx, μπορέσαμε να αναλύσουμε την ευαισθησία των μοντέλων Vision Transformer στο σύνολο δεδομένων ImageNet για να προσεγγίσουμε τους πολλαπλασιασμούς και να εκτελέσουμε επανεκπαίδευση κατά προσέγγιση για να ανακτήσουμε την ακρίβεια, όπως θα δείξουμε στην πειραματική αξιολόγηση.

Υποστήριξη για την αρχιτεκτονική του transformer — Το επίπεδο transformer αποτελεί το θεμελιώδες στοιχείο στην αρχιτεκτονική του Vision Transformer. Ο πρωταρχικός του ρόλος περιλαμβάνει τη λήψη μιας ακολουθίας κομματιών εικόνας ως είσοδο, τη μόχλευση του μηχανισμού attention για τη δημιουργία σχέσεων μεγάλης εμβέλειας και τη δημιουργία μιας νέας ακολουθίας χαρακτηριστικών. Επιπλέον, υπάρχουν συμπληρωματικά μπλοκ τοποθετημένα στην αρχή ή στο τέλος ενός μοντέλου ViT, όπως η ενσωμάτωση patch ή μια κεφαλή ταξινόμησης. Στην ενσωμάτωση patch, η εικόνα εισόδου υφίσταται διαίρεση σε ενημερωμένες εκδόσεις σταθερού μεγέθους, μη επικαλυπτόμενες, με κάθε patch να ενσωματώνεται γραμμικά σε ένα επίπεδο διάνυσμα. Αυτά τα patch embeddings χρησιμεύουν στη συνέχεια ως διακριτικά εισόδου για το μοντέλο του transformer. Προς το συμπέρασμα του μοντέλου ViT, συνήθως υπάρχει μια κεφαλή ταξινόμησης, η οποία έχει ως αποστολή να κάνει προβλέψεις με βάση τα μαθημένα χαρακτηριστικά. Η προσέγγισή μας επικεντρώνεται στην εφαρμογή της προσεγγιστικής αριθμητικής αποκλειστικά στα μπλοκ transformer που περιλαμβάνουν τον μηχανισμό attention, που κυριαρχούν σημαντικά στο χρόνο εκτέλεσης (συνήθως υπερβαίνει το 98%). Αυτά τα μπλοκ συχνά επεκτείνονται σε μπλοκ κωδικοποιητών πολλαπλών transformer, το καθένα από τα οποία περιλαμβάνει κυρίως κανονικοποίηση, attention πολλών κεφαλών και ένα επίπεδο τροφοδοσίας προς τα εμπρός. Προς την ενσωμάτωση της κατά προσέγγιση αριθμητικής, επικεντρωνόμαστε στις δύο τελευταίες, οι οποίες συνεπάγονται την πλειοψηφία των μαθηματικών πράξεων.

Ο μηχανισμός προσοχής μπορεί να οριστεί με ακρίβεια μέσω της ακόλουθης εξίσωσης, όπου η συνάρτηση softmax χρησιμοποιείται για τον υπολογισμό των βαρών που καθορίζουν τη σημασία κάθε στοιχείου στην είσοδο. Σε αυτό το πλαίσιο, το Q αντιπροσωπεύει το διάνυσμα ερωτήματος, το K είναι το διάνυσμα κλειδιού και το V είναι το διάνυσμα τιμής:

$$Attention(Q,K,V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

Λειτουργία πλαισίου — Το πλαίσιο μας, που φαίνεται στο σχήμα Σχήμα 17, προσφέρει μια ορθογώνια προσέγγιση για την προσομοίωση προσεγγιστικών μοντέλων ViT. Οι κύριες λειτουργίες περιγράφονται παρακάτω.

1) Επέκταση των προεπιλεγμένων μονάδων PyTorch: Το TransAxx στοχεύει στη διαχείριση υπολογισμών εντός προσεγγιστικών μοντέλων ViT. Για να το επιτύχει αυτό, επεκτείνει τις προεπιλεγμένες μονάδες PyTorch, επιτρέποντας στις προσαρμοσμένες λειτουργίες να ενσωματώνονται άμεσα με το υπάρχον υπολογιστικό γράφημα PyTorch.

Κατά τη διάρκεια της μεταγλώττισης του μοντέλου, το πλαίσιο μας ανταλλάσσει αυτόματα τα προεπιλεγμένα επίπεδα PyTorch με τα προσαρμοσμένα έπίπεδα, μετατρέποντας το προεπιλεγμένο μοντέλο στην επιθυμητή προσεγγιστική εκδοχή. Αυτά τα επίπεδα δημιουργούνται κατά τη διάρκεια της εκτέλεσης χρησιμοποιώντας την άμεση μεταγλώττιση JIT, εξασφαλίζοντας αποδοτική ενσωμάτωση με το υπολογιστικό γράφημα του μοντέλου. Η JIT κάνει επίσης το μοντέλο ευέλικτο και εύκολο στην τροποποίηση κατά τη διάρκεια της εκτέλεσης. Αυτή η μέθοδος υποστηρίζει σταδιακή μεταγλώττιση, που σημαίνει ότι μόνο τα μέρη του κώδικα που έχουν αλλάξει επαναμεταγλωττίζονται. Αυτό μειώνει σημαντικά το φόρτο επαναλαμβανόμενης μεταγλώττισης και φόρτωσης των επεκτάσεων του TransAxx κατά τη διάρκεια των πειραμάτων.

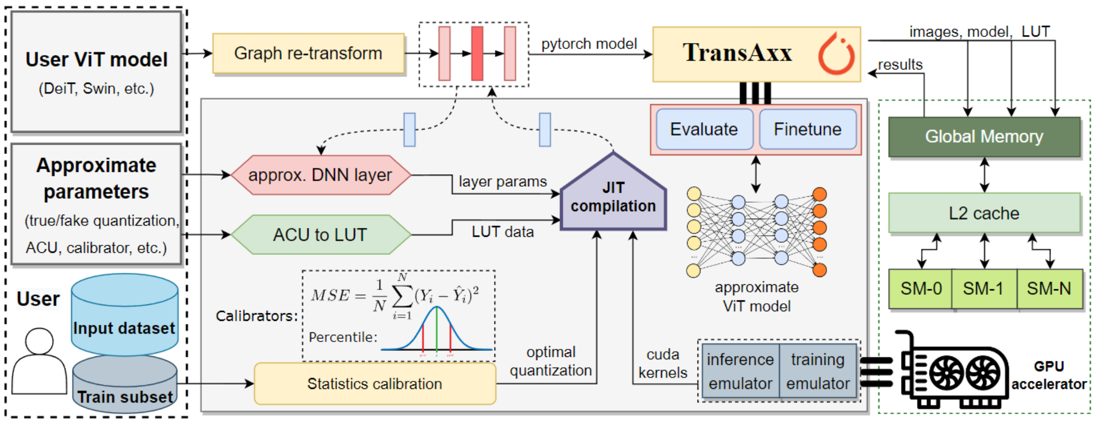
2) Αρχικοποίηση επιπέδου και εκτέλεση πυρήνων: Τα κανονικά αντικείμενα Tensor εντός του PyTorch χρησιμοποιούνται για τη διαχείριση της αρχικοποίησης των βαρών για τους προσαρμοσμένους πυρήνες. Αυτό εξασφαλίζει συνέπεια με τους μηχανισμούς αρχικοποίησης του PyTorch, διατηρώντας τη συμβατότητα και την ευκολία χρήσης εντός του πλαισίου. Στη συνέχεια, τα βάρη/ενεργοποιήσεις ποσοτικοποιούνται βάσει του πλάτους των bit του πολλαπλασιαστή του εκάστοτε επιπέδου και βαθμονομούνται χρησιμοποιώντας τα στατιστικά των ενεργοποιήσεων του επιπέδου. Τέλος, το πλαίσιο μας χρησιμοποιεί μακροεντολές C++ για να εκτελεί τον κατάλληλο πυρήνα GPU ανά στρώμα.

3) Δημιουργία LUTs: Για κάθε προσεγγιστικό πολλαπλασιαστή, δημιουργείται ένας αντίστοιχος πίνακας LUT από την περιγραφή του υψηλού επιπέδου (π.χ., σε C, Matlab, ή HDL συμπεριφοράς). Έχουμε ένα ενσωματωμένο εργαλείο στο TransAxx που μπορεί να δημιουργήσει αυτόν τον LUT για οποιοδήποτε αυθαίρετο προσεγγιστικό πολλαπλασιαστή, του οποίου η συμπεριφορά περιγράφεται σε C ή HDL. Αυτό διευκολύνεται εκτελώντας όλες τις δυνατές πολλαπλασιαστικές πράξεις $x \times y$ (π.χ., χρησιμοποιώντας μια προσομοίωση RTL) για τον δεδομένο προσεγγιστικό πολλαπλασιαστή και στη συνέχεια αποθηκεύοντας τα αποτελέσματα σε έναν LUT. Έτσι, το LUT[x][y] δίνει το προσεγγιστικό αποτέλεσμα του x και του y. Κατά τη διάρκεια της εμπρός περάσματος, το TransAxx χρησιμοποιεί αυτούς τους LUTs και αντικαθιστά τον προεπιλεγμένο (ακριβή) πολλαπλασιαστή με το προσεγγιστικό αποτέλεσμα (δηλαδή, φορτώνοντας την τιμή LUT[x][y]). Ο LUT ήταν μια επιλογή σχεδίασης για να βοηθήσει στη μείωση του χρόνου εξομοίωσης του TransAxx. Αρχικά, οι LUTs αποθηκεύονται στη μνήμη RAM, καθώς έχει μεγάλο μέγεθος και είναι η καταλληλότερη επιλογή για τα τυχαία πρότυπα προσπέλασης των LUTs. Ωστόσο, στη συνέχεια, δείχνουμε πώς βελτιώσαμε αυτές τις προσπελάσεις μνήμης.

4) Βελτιστοποίηση πυρήνων GPU: Είναι κρίσιμο να ληφθούν υπόψη οι μεταφορές μνήμης, καθώς οι λειτουργίες που περιλαμβάνουν LUTs μπορούν γρήγορα να γίνουν memory bound. Συγκεκριμένα, η συμπεριφορά της cache της GPU επηρεάζεται τόσο από τις προδιαγραφές του υλικού όσο και από τα συγκεκριμένα πρότυπα προσπέλασης μνήμης του μοντέλου ViT. Ωστόσο, δεδομένου ότι τα δεδομένα των LUTs είναι μόνο για ανάγνωση, μπορούμε να καθοδηγήσουμε τον μεταγλωττιστή της Nvidia να

μεγιστοποιήσει την απόδοση προσπέλασης μνήμης (δηλαδή, χρησιμοποιώντας τις εγγενείς εντολές της CUDA και τις σημαίες μεταγλώττισης). Χρησιμοποιώντας αυτήν την προσέγγιση, ο πίνακας LUT θα αποθηκεύεται συνήθως μέσω της cache L1 της GPU, που προσφέρει χαμηλή καθυστέρηση και μπορεί να διαμοιραστεί μεταξύ όλων των νημάτων εντός ενός πυρήνα CUDA. Αυτό διευκολύνει την αποδοτική προσωρινή αποθήκευση και προσπέλαση των δεδομένων LUT σε πολλαπλά νήματα.

5) Διαχείριση μεγάλων πλάτων bit: Για σενάρια όπου οι LUTs μπορεί να αυξηθούν σημαντικά σε μέγεθος, ιδιαίτερα με μεγάλα πλάτη bit (>12 bits), το πλαίσιο μας παρέχει μια ευέλικτη λύση. Το TransAxx μπορεί δυναμικά να αντικαταστήσει τον πολλαπλασιασμό βάσει LUT με πολλαπλασιασμό βάσει συνάρτησης (όπου ο προσεγγιστικός πολλαπλασιαστής περιγράφεται εναλλακτικά σε κώδικα C). Αυτή η διαδικασία μπορεί να εισάγει υπολογιστική υπερφόρτωση στον χρόνο εκτέλεσης του DNN, αλλά εξασφαλίζει ότι το πλαίσιο μας παραμένει αποδοτικό και κλιμακούμενο. Αξίζει να σημειωθεί ότι και οι δύο προσεγγίσεις παρέχουν μια 1-1 αναπαράσταση του πολλαπλασιαστή σε υψηλό επίπεδο, επομένως τα αποτελέσματα θα είναι τα ίδια κατά την πρόβλεψη ή την επανεκπαίδευση. Επίσης, είναι σημαντικό να σημειωθεί ότι οι vision transformers λειτουργούν καλά με τιμές χαμηλής ακρίβειας και οι υψηλότερες bit-width, που μπορεί να επηρεάσουν την απόδοση χρόνου εξομοίωσης του TransAxx, συχνά δεν απαιτούνται.



Σχήμα 17. Λειτουργία του TransAxx.

Αυτόματη Εξερεύνηση του σχεδιαστικού χώρου

Η Αυτοματοποιημένη εξερεύνηση του σχεδιαστικού χώρου διαδραματίζει κεντρικό ρόλο στον τομέα της βελτιστοποίησης των Βαθέων Νευρωνικών Δικτύων (DNN), ιδιαίτερα όταν αντιμετωπίζεται η λεπτή ισορροπία μεταξύ της ακρίβειας του μοντέλου και της υπολογιστικής απόδοσης. Αυτό το πεδίο καθίσταται ιδιαίτερα κρίσιμο σε σενάρια όπου η ανάπτυξη μεγάλων και περίπλοκων μοντέλων μπορεί να είναι μη πρακτική λόγω περιορισμών πόρων ή απαιτήσεων σε πραγματικό χρόνο. Στο πλαίσιο της βέλτιστης προσέγγισης στα DNN, οι ερευνητές στοχεύουν να εξερευνήσουν συστηματικά τον τεράστιο χώρο σχεδιασμού, που περιλαμβάνει υπερπαραμέτρους, αρχιτεκτονικές επιλογές και τεχνικές βελτιστοποίησης. Ο στόχος είναι να ανακαλύψουμε διαμορφώσεις που επιτυγχάνουν βέλτιστο συμβιβασμό, αξιοποιώντας τεχνικές όπως η κβαντοποίηση, οι αλγοριθμικές προσεγγίσεις και άλλα. Στην εργασία μας, εστιάσαμε στη βελτιστοποίηση κβαντοποίησης/προσέγγισης, δηλαδή στην εφαρμογή συγκεκριμένων προσεγγιστικών πολλαπλασιαστών σε κάθε επίπεδο DNN προκειμένου να επιτευχθεί η βέλτιστη ισορροπία μεταξύ ακρίβειας και κατανάλωσης ενέργειας. Συγκεκριμένα η μέθοδος εφαρμόζεται για μοντέλα ViT που αποτελούν πρόκληση για την πολυπλοκότητά τους. Εμπνευστήκαμε από το Monte Carlo Tree Search, μια τεχνική αναζήτησης τεχνητής νοημοσύνης, που χρησιμοποιείται συχνά σε επιτραπέζια παιχνίδια, η οποία χρησιμοποιεί αλγόριθμους πιθανοτήτων και ευρετικών για να συνδυάσει την κλασική εφαρμογή της αναζήτησης δέντρων με αρχές από τη μηχανική μάθηση και ιδιαίτερα την ενισχυτική μάθηση. Το MCTS έχει τη δυνατότητα να εξισορροπεί δυναμικά την εξερεύνηση και την εκμετάλλευση, καθιστώντας το λιγότερο επιρρεπές στο να κολλήσει στα τοπικά βέλτιστα σε σύγκριση με άλλες μεθόδους όπως οι άπληστοι αλγόριθμοι που χρησιμοποιούνται συχνά για κβαντοποίηση ακριβείας αναζήτησης [160, 161].

Λειτουργία αναζήτησης που βασίζεται σε MCTS — Η έκθεση της βέλτιστης διαμόρφωσης των κατά προσέγγιση πολλαπλασιαστών μεταξύ κάθε επιπέδου ενός μοντέλου DNN προκειμένου να βρεθεί η καλύτερη αντιστάθμιση μεταξύ απόδοσης και ισχύος ενδέχεται να προκαλέσει σημαντική υπολογιστική επιβάρυνση. Ο χώρος σχεδιασμού γίνεται μεγάλος όπως αναφέρθηκε προηγουμένως και η μέτρηση της ακρίβειας κάθε διαμόρφωσης δεν είναι εφικτή ακόμη και όταν χρησιμοποιείται η επιτάχυνση που βασίζεται σε GPU, ιδιαίτερα στην περίπτωσή μας, όπου η προσομοίωση ViT αυξάνει περαιτέρω τον χρόνο εκτέλεσης. Προκειμένου να περιηγηθούμε συστηματικά στο χώρο των κατά προσέγγιση λύσεων σχεδιασμού, χρησιμοποιούμε μια αναζήτηση δέντρου Monte Carlo (MCTS) που είναι ειδικά προσαρμοσμένη για να βασίζεται στο υλικό. Αυτή η προσέγγιση βοηθά στην επιτάχυνση της εξερεύνησης των αρχιτεκτονικών διαμορφώσεων για τα κατά προσέγγιση μοντέλα Vision Transformer (ViT), επιτυγχάνοντας μια ισορροπία μεταξύ της μεγιστοποίησης της ακρίβειας και της τήρησης προκαθορισμένων περιορισμών ισχύος. Επιπλέον, για να επιταχύνουμε τον βρόχο ανάδρασης, έχουμε επινοήσει έναν προγνωστικό παράγοντα ακρίβειας για την εκτίμηση της ακρίβειας συμπερασμάτων. Ως αποτέλεσμα, η μεθοδολογία μας επιτυγχάνει μια σχεδόν βέλτιστη καμπύλη Pareto, εξισορροπώντας αποτελεσματικά την κατανάλωση ενέργειας και την ακρίβεια.

Algorithm 1 Pseudocode for our hw-driven MCTS

Input: 1) Model M, 2) Ground truth batch B_g, 3) Selected ACUs A, 4) Exploration constant c, 5) Rollout policy P, 6) No. of Simulations N
Output: 1) Optimal Approx. Configs C_{out}

```
1: $rootNode \leftarrow Node(M)$
2: for $i \leftarrow 1$ to $N$ do
3:     $node \leftarrow rootNode$
4:     while not $node.isTerminal()$ do
5:         if $node.isFullyExpanded()$ then
6:             $node \leftarrow node.getBestChild(c)$
7:         else
8:             $node \leftarrow expand(node)$
9:             break
10:        end if
11:    end while
12:    $state \leftarrow node.state$
13:    while not $state.isTerminal()$ do
14:        $a \leftarrow chooseAction(state, P, A)$
15:        $state \leftarrow state.takeAction(a), a \in A$
16:    end while
17:    $Y_i \leftarrow AxxConfig(state)$
18:    $accuracy_i, power_i \leftarrow evaluate(M, Y_i, B_g)$
19:    $reward \leftarrow accuracy_i - \lambda \times power_i$
20:    $backprop(node, reward)$
21: end for
22: $C_{out} \leftarrow pareto(accuracy_i, power_i, Y_i), \forall i \in [1, N]$
```

Καθορισμός καλύτερης πολιτικής — Η πολιτική rollout είναι συνήθως μια απλή ευρετική για την εκτίμηση της ανταμοιβής μιας δεδομένης κατάστασης επιλέγοντας τυχαία ενέργειες μέχρι να επιτευχθεί η κατάσταση τερματικού [163]. Συχνά εφαρμόζεται ως τυχαία πολιτική, όπου οι ενέργειες επιλέγονται ομοιόμορφα τυχαία, χωρίς κάποια συγκεκριμένη στρατηγική, αλλά με στόχο την εξερεύνηση ενός ευρύτερου φάσματος καταστάσεων. Χρησιμοποιώντας τη γνώση του συγκεκριμένου τομέα σχετικά με την ευαισθησία κάθε επιπέδου στην προσέγγιση, εφαρμόσαμε μια πιο περίπλοκη πολιτική. Έστω $S = (s_{j,1}, s_{j,2}, \dots, s_{j,L})$ η λίστα ευαισθησίας επιπέδου ενός ACU A_j. $s_{j,i}$ είναι η κανονικοποιημένη ακρίβεια του μοντέλου όταν το ACU A_j με $j \in [1, k]$ εφαρμόζεται μόνο στο στρώμα i. Ομοίως, μπορούμε να αναπαραστήσουμε τη συνολική επιστρεφόμενη ισχύ του κατά προσέγγιση μοντέλου όταν το A_j εφαρμόζεται στο επίπεδο i ως $p_{j,i}$. Συμπερασματικά, μπορούμε τώρα να εκφράσουμε την πιθανότητα να κάνουμε μια συγκεκριμένη ενέργεια στην πολιτική rollout, δηλαδή να επιλέξουμε ένα A_j από k διαθέσιμα ACU για το επίπεδο i ως:

$$P(A_j)_i = \frac{e^{(s_{j,i} - \lambda \times p_{j,i})}}{\sum_{z=1}^{k} e^{(s_{z,i} - \lambda \times p_{z,i})}}$$

Πειραματική Αξιολόγηση

Σε αυτή την ενότητα παρουσιάζουμε την αξιολόγηση των πλαισίων AdaPT και TransAxx σχετικά με πολλά δίκτυα DNN με την αντίστοιχη βαθμονόμηση κβαντοποίησης, εκπαίδευση με επίγνωση προσέγγισης και χρόνο προσομοίωσης. Επίσης, μαζί με τις μετρήσεις απόδοσης, θα εμφανίζονται οι προδιαγραφές σχετικά με τις παραμέτρους κάθε μοντέλου, τον αριθμό των λειτουργιών MAC (OPs) και το σύνολο δεδομένων που χρησιμοποιούνται. Τα πειράματα διεξήχθησαν σε επεξεργαστή Intel Xeon Gold 6138 στα 2,00 GHz και 64 GB RAM για πειράματα CPU AdaPT ενώ για πειράματα TransAxx χρησιμοποιήθηκε το ίδιο σύστημα μαζί με μια GPU Nvidia Tesla V100. Επιπλέον, θα δείξουμε την αποτελεσματικότητα του αλγορίθμου MCTS για βέλτιστη αναζήτηση στον κατά προσέγγιση χώρο λύσεων για την εύρεση της καλύτερης ισορροπίας μεταξύ ακρίβειας και κατανάλωσης ενέργειας.

Αποτελέσματα στο AdaPT — Για σκοπούς επίδειξης επανεκπαίδευσης, δύο προσεγγιστικοί πολλαπλασιαστές, που υλοποιούνται ως πίνακες αναζήτησης (LUTs), χρησιμοποιούνται με διακριτές τιμές Μέσου Σχετικού Σφάλματος (MRE) και Μέσου Απόλυτου Σφάλματος (MAE) από τη βιβλιοθήκη EvoApprox [137]. Η μέτρηση ακρίβειας top-1 χρησιμοποιείται γενικά, εκτός από τα μοντέλα ImageNet, τα οποία χρησιμοποιούν το top-5.. Μέσω της κατά προσέγγιση επανεκπαίδευσής μας, τα DNN μπορούν να προσαρμοστούν στον προσαρμοσμένο κατά προσέγγιση υλικό, με αποτέλεσμα αυξημένη ακρίβεια για το κατά προσέγγιση DNN. Επιπλέον, παρέχουμε μια σύνοψη του χρόνου εξομοίωσης για κάθε κατά προσέγγιση DNN.

mul8s_1L2H	**MAE: 0.081 %, MRE: 4.41 %, power: 0.301mW[1]**					
DNN	**FP32**	**8bit**	**8bit calib.**	**8bit approx.**	**retrain[3]**	**time**
ResNet50	93.65%	93.55%	93.59%	82.69 %	93.44%	763s
VGG19	93.95%	93.80%	93.82%	90.7%	93.56%	318s
VAE-MNIST	99.99%	99.95%	99.96%	93.12%	99.88%	9.28s
LSTM-IMDB	83.10%	82.90%	82.95%	79.9%	82.63%	710s
SqueezeNet	80.6%	79.01%	80.16%	62.01%	76.21%	620s

mul12s_2KM	MAE: 1.2e-6 %, MRE: 4.7e-4 %, power: 1.205mW[2]					
DNN	**FP32**	**12bit**	**12bit calib.**	**12bit approx.**	**retrain[3]**	**time**
ResNet50	93.65%	93.60%	93.61%	93.52%	90.54%	798s
VGG19	93.95%	93.80%	93.81%	93.81%	93.71%	359s
VAE-MNIST	99.99%	99.98%	99.98%	99.98%	99.99%	10.11s
LSTM-IMDB	83.10%	82.94%	82.96%	82.96%	83.12%	1040s
SqueezeNet	80.6%	80.11%	80.3%	80.35%	80.50%	623s

Πίνακας 9. Ακρίβεια και χρόνος επανεκπαίδευσης στο AdaPT για διάφορα DNN.

DNN	Native CPU	Baseline Approx	AdaPT (w/ func)	AdaPT (w/ LUT)	AdaPT vs Baseline
ResNet50	0.5 min	76.5 min	104 min	1.7 min	45×
DenseNet12	0.48 min	53.2 min	72 min	1.6 min	33.2×
VGG19	0.2 min	91.7 min	125 min	1.7 min	53.9×
Fashion-GAN	0.003 min	0.02 min	1.1 min	0.012 min	1.7×
VAE-MNIST	0.015 min	0.1 min	1.2 min	0.02 min	5×
LSTM-IMDB	1.36 min	48.5 min	449 min	7.6 min	6.4×
Inceptionv3	22.1 min	2909 min	4560 min	83 min	35.1×
SqueezeNet	11.6 min	443 min	576 min	20.6 min	21.5×
ShuffleNet	11.4 min	163 min	251 min	22.4 min	7.3×

Πίνακας 10. Χρόνος προσομοίωσης για κάθε προσεγγιστικό νευρωνικό δίκτυο.

Αποτελέσματα στο TransAxx — Σε αυτήν την ενότητα, διεξάγουμε πειράματα για να αξιολογήσουμε την απόδοση του πλαισίου TransAxx, εστιάζοντας στην ακρίβεια και τον χρόνο εκτέλεσης των δημοφιλών μοντέλων Vision Transformer (ViT) σε διάφορους κατά προσέγγιση πολλαπλασιαστές. Όσον αφορά τις εκδόσεις λογισμικού, το TransAxx αναπτύχθηκε στο PyTorch 1.13 με την έκδοση CUDA 11.7. Η εγκατάσταση υλικού που χρησιμοποιήθηκε για τα πειράματα αποτελούνταν από την ίδια CPU με τα πειράματα του AdaPT, έναν επεξεργαστή Intel Xeon Gold 5218R 20 πυρήνων με 64 GB μνήμης RAM, μαζί με μια GPU Nvidia Tesla V100 ως τον επιταχυντή υλικού του TransAxx. Παρακάτω παρουσιάζονται οι χρόνοι εκτέλεσης, οι οποίοι κρίνονται ικανοποιητικοί και αρκετά γρήγοροι, ειδικά όταν ληφθεί υπόψη ότι δεν υπάρχει άλλη εναλλακτική για κατά προσέγγιση προσομοίωση ViT. Ακόμα συνοψίζουμε τα αποτελέσματα ακρίβειας που ελήφθησαν πριν και μετά την επανεκπαίδευση για κάθε πολλαπλασιαστή. Γενικά, βλέπουμε ότι η κατά προσέγγιση επανεκπαίδευση μειώνει το χάσμα ακρίβειας με επιτυχία στην πλειονότητα των κατά προσέγγιση μοντέλων, καθώς τα βάρη του δικτύου μπορούν να προσαρμοστούν στις κατανομές που αντιπροσωπεύουν τα ACU. Επιπλέον, αναφέρουμε τη συνολική μείωση ισχύος MAC, καθώς θα είναι σημαντικό για τα πειράματα της εξερεύνησης του χώρου σχεδιασμού στην επόμενη παράγραφο. Είναι σαφές ότι η πραγματική μείωση της ισχύος θα επηρεαζόταν από πολλούς παράγοντες, αλλά η μείωση από τις λειτουργίες MAC έχει συνήθως μια κλιμακωτή επίδραση στη συνολική κατανάλωση. Το ποσοστό μείωσης είναι σε σχέση με τα συνολικά MAC που προσεγγίζονται σε κάθε μοντέλο προκειμένου να έχουμε πιο ακριβείς μετρήσεις. Σαν σημείο αναφοράς έχουμε την κατανάλωση ισχύος του ακριβούς πολλαπλασιαστή mul8s_1KV6 (0,425 mW).

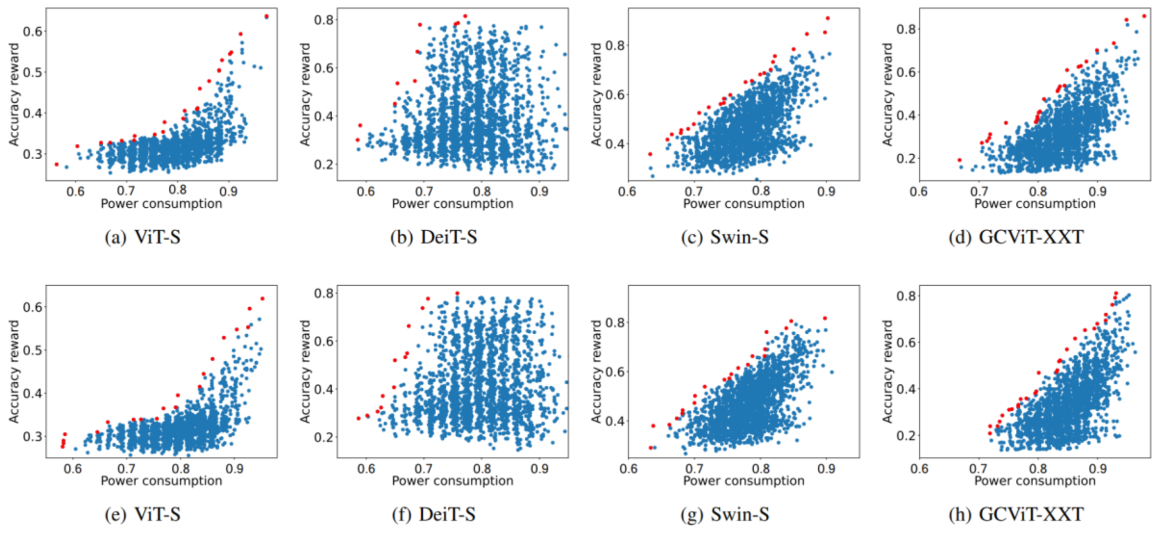
DNN	**FLOPs**	**Params**	**Inference (w/ func.)**	**Inference (w/ LUT)**	**Retraining (w/ LUT)**
ViT-S	4.2G	22.1M	121 min	6 min	5.5 min
DeiT-S	4.2G	22.1M	122 min	6.1 min	5.8 min
Swin-S	8.5G	49.6M	242 min	13.1 min	13 min
GCViT-XXT	1.9G	12M	43.5 min	3 min	3.5 min

Πίνακας 11. Χρόνοι προσομοίωσης για κάθε προσεγγιστικό μοντέλο ViT στο TransAxx.

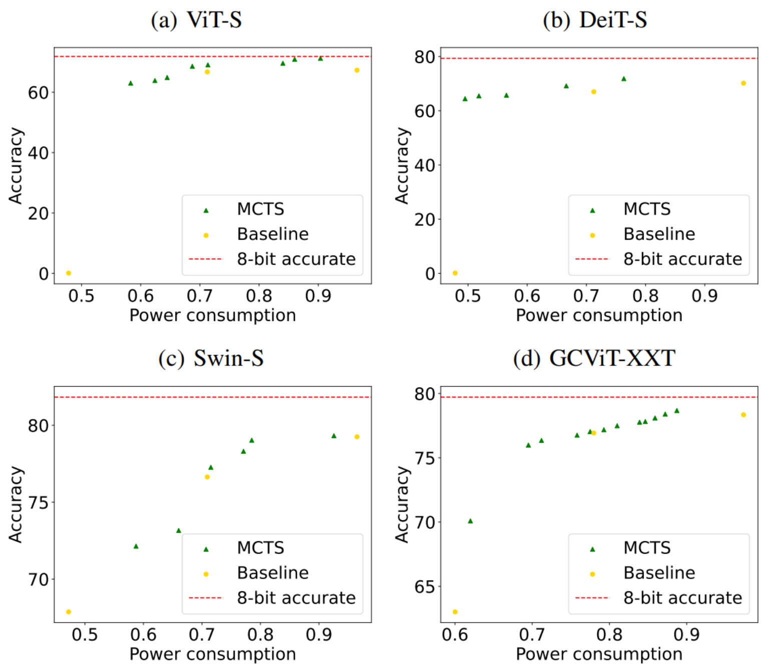
Model specifications				ACU 1: `mul8s_1KV9` MRE 0.90%, power: 0.410mW			ACU 2: `mul8s_1L2H` MRE 4.41%, power: 0.301mW			ACU 3: `mul8s_1L2L` MRE 12.26%, power: 0.200mW		
Name	MACs approx.	FP32	8bit (calib.)	Initial	Retrained	Power ↓	Initial	Retrained	Power ↓	Initial	Retrained	Power ↓
ViT-S	98.54	74.64	71.86	34.95	67.31	3.45	1.264	66.74	28.75	0.090	0.15	52.18
DeiT-S	98.54	81.34	79.34	0.96	70.16	3.45	0.10	67.01	28.75	0.10	0.11	52.18
Swin-S	99.7	82.89	81.83	79.56	79.25	3.49	64.30	76.64	29.09	0.41	67.87	52.79
GCViT	75.5	79.72	78.91	73.50	78.346	2.64	51.56	76.93	22.03	0.26	63.01	39.98

Πίνακας 12. Μετρικές ακρίβειας και κατανάλωσης ισχύος [%] για κάθε μοντέλο και πολλαπλασιαστή.

Αποτελέσματα του MCTS — Τέλος, αξιολογούμε τη χρήση των αλγορίθμων MCTS που βασίζονται σε υλικό για την εύρεση της βέλτιστης καμπύλης Pareto για ακρίβεια και ισχύ. Απεικονίζουμε τα διαγράμματα διασποράς από το MCTS για κάθε μοντέλο στόχου ViT χρησιμοποιώντας προσομοιώσεις 2000 (η κατανάλωση ενέργειας κανονικοποιείται). Κάθε αποκτημένο Pareto (με κόκκινο) αντιπροσωπεύει τη γνώση που έχει μάθει το σύστημα για την εύρεση της βέλτιστης διαμόρφωσης πολλαπλασιαστή και είναι βασικά η έξοδος του αλγορίθμου αναζήτησης. Για να παρέχουμε περαιτέρω συγκρίσεις, πειραματιζόμαστε για δύο διακριτές παραμέτρους λ για την πόλωση ισχύος, όπως περιγράφεται στον Αλγόριθμο 1 και στην εξίσωση πολιτικής MCTS. Ακόμα, συνοψίζουμε τις λύσεις που βρέθηκαν με την βασική προσέγγιση των μοντέλων (με κίτρινο), που ελήφθησαν από τον Πίνακας 12, μαζί με τις προτεινόμενες βέλτιστες λύσεις που λαμβάνονται από την προσέγγισή μας που βασίζεται στο MCTS (με πράσινο). Ως βασικές λύσεις ορίζουμε τα σημεία δεδομένων ακρίβειας/ισχύς από τον Πίνακας 12 στα οποία η προσέγγιση εφαρμόζεται ομοιόμορφα σε όλα τα επίπεδα του μοντέλου χωρίς μεγάλη προσαρμογή. Τα πράσινα σημεία δεδομένων του σχήματος προέρχονται από τη μεγάλη εξερεύνηση του χώρου παραμέτρων χρησιμοποιώντας MCTS, που απεικονίζεται στο Σχήμα Σχήμα 18, εστιάζοντας μόνο στις βέλτιστες λύσεις που επιτυγχάνονται μέσω αυτού του αλγόριθμου (κόκκινα σημεία στο Σχήμα Σχήμα 18). Αυτά τα σημεία δεδομένων αξιολογούνται σε ολόκληρο το σύνολο δεδομένων ImageNet και τα σημεία Pareto απεικονίζονται στα αντίστοιχα διαγράμματα διασποράς του Σχήματος Σχήμα 19 ως πράσινοι τριγωνικοί δείκτες. Οι βασικές λύσεις από τον Πίνακας 12, για κάθε κατά προσέγγιση πολλαπλασιαστή, περιλαμβάνονται σκόπιμα για να παρέχουν ένα σημείο αναφοράς για την αξιολόγηση της αποτελεσματικότητας των προτεινόμενων λύσεων χρησιμοποιώντας τον αλγόριθμο αναζήτησης MCTS.



Σχήμα 18. Διαγράμματα διασποράς MCTS που οδηγείται από hw χρησιμοποιώντας 2000 προσομοιώσεις για παράμετρο λ=1.5 (πάνω) και λ=0.5 (κάτω).



Σχήμα 19. MCTS-βέλτιστες λύσεις που βασίζονται σε πραγματική ακρίβεια (πράσινο) και αρχικές κατά προσέγγιση λύσεις (κίτρινο).

Σε γενικές γραμμές, η αναζήτησή μας MCTS επιτρέπει τη λήψη αποτελεσμάτων από την καμπύλη Pareto προσαρμοσμένη για να ικανοποιεί τις απαιτήσεις του σχεδιαστή και τελικά δίνει μια πιο λεπτομερή αντιστάθμιση μεταξύ ακρίβειας και ισχύος σε σχέση με την απλή προσεγγιστική υλοποίηση. Οι απαιτήσεις μας, σχετικές με τη συγκεκριμένη εφαρμογή, περιλάμβαναν περιορισμούς στην κατανάλωση ενέργειας, τα κατώφλια ακρίβειας και τον χρόνο εξερεύνησης του αλγόριθμου MCTS, αλλά μπορούν να ληφθούν υπόψη πρόσθετα κριτήρια για το υπό εξέταση σύστημα. Σημειώνουμε ότι, εξ όσων γνωρίζουμε, η εργασία μας είναι η πρώτη που αναλύει τον αντίκτυπο των κατά προσέγγιση πολλαπλασιαστών στα μοντέλα ViT και παρέχει ένα πλαίσιο για α) την αξιολόγηση της ακρίβειας συμπερασμάτων με λογική ταχύτητα, β) την εκτέλεση επανεκπαίδευσης ViT με επίγνωση της προσέγγισης , και γ) παροχή μιας λεπτομερούς αντιστάθμισης ακρίβειας-ισχύς κατά την εξερεύνηση του χώρου λύσεων των ViT. Για παρόμοια ακρίβεια με τις κατά προσέγγιση βασικές λύσεις (εντός ~1% μέγιστη διαφορά), το MCTS παρέχει κατά μέσο όρο λύσεις με ~21 % χαμηλότερη ισχύ. Παρά τις σημαντικές εξοικονομήσεις που αναφέρθηκαν, υψηλότερη εξοικονόμηση ενέργειας για παρόμοια απώλεια ακρίβειας έχουν συχνά αναφερθεί σε κατά προσέγγιση CNN [136]. Ως εκ τούτου, απαιτείται πρόσθετη έρευνα για μοντέλα ViT ή/και απαιτούνται ειδικοί κατά προσέγγιση πολλαπλασιαστές/τεχνικές προσέγγισης. Το έργο μας θέτει τις βάσεις για την εξερεύνηση αυτού του προκλητικού τομέα και διευκολύνει σημαντικά τους σχεδιαστές στον εντοπισμό λύσεων αρκετά γρήγορα που ευθυγραμμίζονται πιο στενά με την επιθυμητή ισορροπία ισχύος και ακρίβειας στα μοντέλα ViT.

Επίλογος

Αυτή η διατριβή έχει διερευνήσει το περίπλοκο πεδίο του αποδοτικού υπολογισμού για τη βαθιά μάθηση (DL), με κύρια εστίαση στη βελτιστοποίηση επιταχυντών βαθιών νευρωνικών δικτύων για προσαρμοσμένο υλικό. Μέσω μιας ολοκληρωμένης εξερεύνησης διαφόρων στρατηγικών βελτιστοποίησης λογισμικού και υλικού, αυτή η έρευνα συνέβαλε στην κατανόηση των υποκείμενων μηχανισμών υλικού προς τη βέλτιστη εκτέλεση μοντέλων τεχνητής νοημοσύνης. Επιπλέον, αυτή η διατριβή διερεύνησε τη χρήση προσεγγιστικών πολλαπλασιαστών για την εκμετάλλευση των δυνατοτήτων προσεγγιστικού υλικού για γρήγορη ταχύτητα εξαγωγής συμπερασμάτων και ενεργειακή απόδοση των DNN. Προτείνοντας δύο προσαρμοσμένα πλαίσια εξομοίωσης, το AdaPT και το TransAxx, αντιμετωπίζουμε την πρόκληση της προσομοίωσης προσεγγιστικών DNN. Συγκεκριμένα, δίνουμε τη δυνατότητα σε ερευνητές και επαγγελματίες να δημιουργήσουν γρήγορα πρωτότυπα και να αξιολογήσουν κατά προσέγγιση DNN, προωθώντας μια πιο προσιτή και επαναληπτική προσέγγιση για την ανάπτυξη μοντέλων. Τέλος, αυτή η διατριβή πρότεινε μια μέθοδο βασισμένη στον αλγόριθμο MCTS για ταχεία εξερεύνηση του χώρου παραμέτρων με στόχο την εύρεση μιας βέλτιστης αντιστάθμισης μεταξύ κατανάλωσης ενέργειας και ακρίβειας σε κατά προσέγγιση DNN, και συγκεκριμένα ViTs, δείχνοντας την αποτελεσματικότητά και χρησιμότητα του αλγορίθμου. Καθώς το πεδίο της βαθιάς μάθησης συνεχίζει να εξελίσσεται, τα ευρήματα αυτής της διατριβής συμβάλλουν στη συνεχιζόμενη συζήτηση σχετικά με τον αποτελεσματικό υπολογισμό για AI εφαρμογές. Οι γνώσεις που προέκυψαν από τη βελτιστοποίηση συμπερασμάτων νευρωνικών δικτύων σε προσαρμοσμένο υλικό, σε συνδυασμό με το πλαίσιο εξομοίωσης για κατά προσέγγιση υπολογιστές, σηματοδοτούν ένα ουσιαστικό βήμα προς τη γεφύρωση του χάσματος μεταξύ λογισμικού και υλικού στον τομέα της βαθιάς μάθησης.

Γλωσσάριο

AI	Τεχνητή Νοημοσύνη
ANN	Τεχνητό Νευρωνικό Δίκτυο
ASIC	Κυκλώματα Ειδικής Εφαρμογής
CNN	Συνελικτικό Νευρωνικό Δίκτυο
CPU	Μονάδα Κεντρικής Επεξεργασίας
DDR	Μνήμη Διπλής Ταχύτητας Δεδομένων
DNN	Βαθιά Νευρωνικά Δίκτυα
DRAM	Δυναμική Μνήμη Τυχαίας Προσπέλασης
DSP	Ψηφιακή Επεξεργασία Σήματος
FLOPs	Πράξεις Κινητής Υποδιαστολής Ανά Δευτερόλεπτο
FPGA	Συστοιχία Επιτόπια Προγραμματιζόμενων Πυλών
FPS	Καρέ Ανά Δευτερόλεπτο
GAN	Γεννητικά Ανταγωνιστικά Δίκτυα
GPU	Μονάδα Επεξεργασίας Γραφικών
HBM	Μνήμη Υψηλής Ευρυζωνικότητας
HPC	Υπολογιστές Υψηλής Απόδοσης
IoT	Διαδίκτυο των Πραγμάτων
MLP	Πολυεπίπεδο Αντίληπτρο
MSB	Πιο Σημαντικό Μπιτ
MSE	Μέσο Τετραγωνικό Σφάλμα
NN	Νευρωνικό Δίκτυο
OpenCL	Ανοιχτή Γλώσσα Υπολογιστικής (για παράλληλες εφαρμογές)
OPS	Πράξεις Ανά Δευτερόλεπτο
PCIe	Θυρίδα Διασύνδεσης Περιφερειακών Εξαρτημάτων Express
RAM	Μνήμη Τυχαίας Προσπέλασης
ReLU	Γραμμική Συνάρτηση Ενεργοποίησης
RL	Ενισχυτική με Μάθηση
RMSE	Τετραγωνική Ρίζα του Μέσου Τετραγωνικού Σφάλματος
SGD	Στοχαστική Απότομη Κάθοδος
SIMD	Ροή μονής εντολής πολλαπλών δεδομένων
SoC	Σύστημα σε τσιπ
SoTA	Τελευταία Λέξη της Τεχνολογίας
TPU	Μονάδα Επεξεργασίας Τενσορικών Προγραμμάτων
ViT	Οπτικός Μετασχηματιστής (είδος νευρωνικού δικτύου)

Appendices

Appendix A.

The SERRANO platform: Stepping towards seamless application development & deployment in the heterogeneous edge-cloud continuum.

The need for real-time analytics and faster decision-making mechanisms has led to the adoption of hardware accelerators such as GPUs and FPGAs within the edge cloud computing continuum. However, their programmability and lack of orchestration mechanisms for seamless deployment make them difficult to use efficiently. We address these challenges by presenting SERRANO, a project for transparent application deployment in a secure, accelerated, and cognitive cloud continuum. In this work, we introduce the SERRANO platform and its software, orchestration, and deployment services, focusing on its methods for automated GPU/FPGA acceleration and efficient, isolated, and secure deployments. By evaluating these services against representative use cases, we highlight SERRANO 's ability to simplify the development and deployment process without sacrificing performance.

A.1. Introduction

The explosive growth and increasing power of IoT devices, along with the emergence of 5G networks, has led to unprecedented data volumes. Emerging use cases around smart homes, autonomous vehicles, and smart factories require edge processing due to critical real-time requirements. To this end, multi-layered computing architectures are emerging where compute resources and applications are distributed from the edge of the network, closer to the point of data collection, to the cloud, realizing the edge-cloud computing continuum [173].

Even though edge-cloud architectures extend the compute capacity of the traditional cloud paradigm, the excessive compute demands of modern use-cases require the introduction of hardware accelerators in the computing stack. Specialized hardware acceleration platforms (e.g., GPUs, FPGAs) can achieve higher performance than typical processing systems for the same power envelope [147, 120]. Typical examples of accelerators include resource-constrained devices at the edge to high-performance, massively parallel devices at the cloud.

While such hardware platforms offer performance gains, these benefits do not come for free, as they typically require hardware knowledge to program. From the developer's perspective, a trade-off arises between performance and programmability. To relieve the developer of the programming burden, tools are needed for seamless development. In addition, the introduction of these devices into the edge-cloud computing continuum underscores the need for orchestration and deployment tools to ensure efficient and secure executions.

To this end, we present the H2020 project SERRANO, which provides a new ecosystem of technologies to address the challenges of introducing heterogeneity in the edge- cloud computing continuum. Specifically, the contributions of SERRANO are:

- *Software Services* for automatic optimization of applications targeting GPU and FPGA devices.
- *Orchestration Services* for end-to-end cognitive orchestration along with closed-loop control.
- *Deployment Services* for isolated and private execution of the heterogeneous compute units.

We evaluate the software and deployment services against a set of representative use cases. The experimental results highlight the ability of SERRANO to simplify the development and deployment of applications without sacrificing performance.

A.2. SERRANO Platform Overview

The SERRANO platform combines (a) a set of tools that simplify the development/deployment process of accelerated applications and (b) runtime mechanisms that ensure that the quality of service (QoS) requirements of deployed applications are met while efficiently leveraging the underlying heterogeneous infrastructure. These technologies and services are abstracted through APIs and SDKs to simplify their use (e.g., Plug&Chip [174]). Figure below shows an overview of the platform.

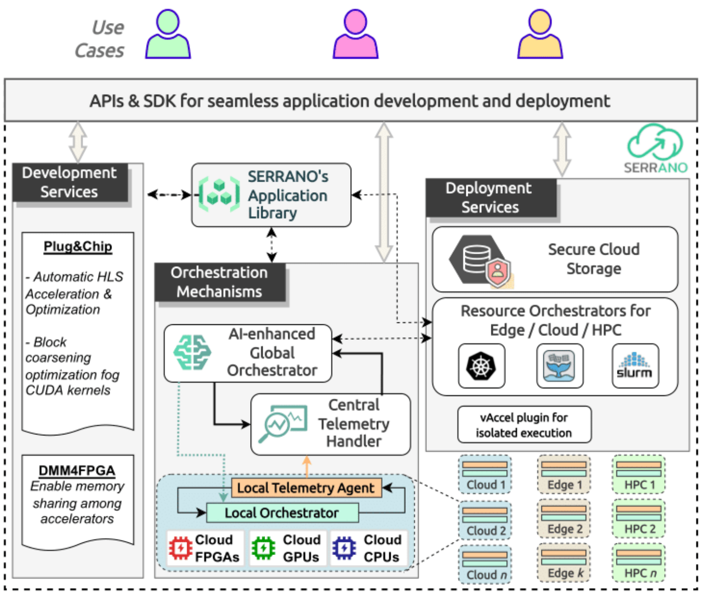

Use Cases
APIs & SDK for seamless application development and deployment
SERRANO
Development Services
SERRANO's Application Library
Deployment Services
Plug&Chip
- Automatic HLS Acceleration & Optimization
- Block coarsening optimization fog CUDA kernels
Orchestration Mechanisms
Secure Cloud Storage
Resource Orchestrators for Edge / Cloud / HPC
AI-enhanced Global Orchestrator
slurm
Central Telemetry Handler
vAccel plugin for isolated execution
DMM4FPGA
Enable memory sharing among accelerators
Local Telemetry Agent
Local Orchestrator
Cloud FPGAs
Cloud GPUs
Cloud CPUs
Cloud 1
Edge 1
HPC 1
Cloud 2
Edge 2
HPC 2
Cloud n
Edge k
HPC n

Figure A.2-1. The SERRANO platform

Service Development Kit: SERRANO contributes to the development and deployment of applications that leverage heterogeneous resources by providing a Service Development Kit (SDK) to increase developer productivity in building, deploying, and managing novel applications while having better control over compute, storage, and network infrastructure. In addition to increasing developer productivity and thus reducing time-to-market, the proposed toolkit provides mechanisms that can provide solutions for an application, i.e., different application versions, that compromise between performance, energy efficiency, and accuracy. Finally, the development services of SERRANO are used to optimize the computationally intensive parts of the applications of UC and to create a database of application versions with different trade-offs targeting the different resources available on the platform.

Orchestration Approach: SERRANO proposes an orchestration system that manages the underlying infrastructure in an abstract and disaggregated manner. This is achieved

through a hierarchical architecture consisting of three components: i) the central resource orchestrator, ii) the local resource orchestrators, and iii) the telemetry framework.
In this context, submitting an application to the SERRANO platform requires providing a set of QoS metrics (e.g., QPS, maximum error) that describe the desired state of the application. The central orchestrator, a mechanism that knows the current state of the underline resources, decides on the optimal placement, i.e., the edge, cloud, and HPC resources that can ensure that the application's requirements are met while minimizing resource consumption and hence energy consumption of the entire infrastructure. Once the optimal placement is decided, the central resource orchestrator assigns the workload to the selected resources along with the desired QoS metrics and coordinates the necessary data movement. Then, the local orchestrators are responsible for actually deploying the application using the appropriate application versions provided by the SDK. Finally, the proper functioning of the SERRANO orchestrator would not be possible without the telemetry framework, which captures information about the current infrastructure and application status.

Use Case Scenarios: The SERRANO platform is evaluated using three use-cases from different scientific domains that illustrate the platform's ability to solve multiple computationally intensive problems with different requirements.

Secure Data Storage: This use case focuses on security and distributed data storage. Protecting files from malicious third parties is done through encryption and novel erasure coding. Therefore, SERRANO will explore acceleration solutions for encoding and decoding tasks that fragment the encoded data into multiple pieces so that the encoded pieces can be stored in distributed locations, providing a secure storage solution.

High-Performance Fintech Analysis: The second use case comes from the field of financial technology, more specifically from the field of portfolio management and analysis. It deals with various AI algorithms accelerated by the heterogeneous computing resources of SERRANO to automatically manage multiple personalized portfolios simultaneously.

Machine Anomaly Detection in Manufacturing Environments: The third use-case belongs to the field of machine anomaly detection. In particular, high-frequency sensors generate large amounts of data that are processed in real time to automatically detect anomalies in machines. SERRANO will analyze the collected data and accelerate its processing.

A.3. SERRANO technologies and resources

SERRANO's hardware infrastructure

The SERRANO platform encapsulates hardware and HPC platforms both at the edge and in the cloud. The cloud infrastructure is coupled with both programmable FPGA accelerators and GPU devices. Accelerators are used to increase the performance and energy efficiency of the workloads being executed.

The cloud FPGAs and GPUs connected to the cloud side of SERRANO are: (i) a Xilinx Alveo U200 accelerator card, (ii) a Xilinx Alveo U50 accelerator card, and (iii) two NVIDIA Tesla T4 GPUs. In addition, NVIDIA BlueField-2 data processing units (DPUs) are used. System on a Chip (SoC) devices are used in the edge infrastructure. The edge FPGA and GPU devices in the infrastructure are: (i) a Xilinx MPSoC ZCU102 device, (ii) a Xilinx MPSoC ZCU104 device, (iii) an NVIDIA Xavier AGX, and (iv) an NVIDIA Xavier NX. In addition, the SmartBox, an industrial-ready box for data acquisition at the edge of the network with a 60 GB hard disk, is used for edge processing. Among the many HPC resources available on the SERRANO platform is the HPE Apollo 9000 Hawk.

SERRANO's Software Services

SERRANO provides a unified framework consisting of three tools for HLS and CUDA accelerator development and optimization.

Automatic HLS Optimization: SERRANO offers a tool that automatically optimizes synthesizable C/C++ kernels for Xilinx FPGAs through High-Level Synthesis. This optimization scheme identifies points of interest, i.e., loops and arrays, applies directives (e.g., loop unrolling, array partition), and performs synthesis to get the latency and resource utilization. By applying different combinations of directives, the optimizer proposes an approximation to the Pareto-optimal designs with respect to the underline architecture of the target device.
Due to the large design space, the DSE is performed using the algorithm NSGA-II. The exploration phase consists of the following steps: a) the configuration population is initialized, b) each configuration of the current population is applied to the source code using a source-to-source compiler and the output is synthesized using the Xilinx Vitis tool chain, and c) the synthesis outputs of the population are passed to NSGA-II to build the next generation configurations. Steps b) and c) are executed iteratively until the termination criterion is reached.

Automatic CUDA Optimization: A CUDA auto-tuning framework for kernel source codes targeting Nvidia GPUs has also been developed. This framework is based on block coarsening, a kernel transformation that merges the workload of 2 or more thread blocks while keeping the number of threads per block the same. Consequently, multiple adjacent blocks are merged to deal with the issues associated with extensive fine-grained parallelism. The proposed framework is based on two components: a) a regression model trained on a representative source code dataset, and b) the source-to-source compiler. The regressor predicts the optimal block coarsening factors for different applications, workload inputs, and GPU architectures, while the source-to-source compiler applies the predicted source code transformations.

Dynamic Memory Management in HLS: A tool has been developed that allows multiple FPGA-implemented HLS accelerators to share and reuse on-chip memory resources at execution time with minimal external fragmentation \cite{DMM}, \cite{Defrag}. This tool groups portions of on-chip memory into structures comparable to those of traditional computer architectures, forming a shared memory area consisting of the local memories of the reconfigurable platform. Dynamic memory allocation is performed by the HLS accelerators through a first-fit allocator implemented on-chip.
To minimize external fragmentation of the heaps and thus increase memory efficiency, an HLS on-chip garbage collector is implemented to compact the fragmented memory regions of the heaps. To reduce the performance overhead of executing the garbage collector, an offline stochastic analysis of the accelerators' memory patterns is performed to determine when (i.e., at the time when the heaps' fragmentation percentage exceeds a user-defined fragmentation threshold called Θ the compaction algorithm should be executed. This analysis is based on a Monte-Carlo model that pseudo-randomly emulates the memory patterns created by running many accelerators in parallel for different Θ thresholds.

The AI orchestration services of SERRANO follow a hierarchical architecture that provides end-to-end closed-loop orchestration of running workloads. At the top is the AI-assisted service orchestrator (AISO), which works in conjunction with the resource orchestrators responsible for the application deployments.

AI-enhanced service orchestrator: The AISO is the central orchestrator of the platform and is responsible for translating the parameters specified by the end user into the appropriate deployment constraints. The output of this component is a list of possible deployment scenarios (edge, cloud). The input of the AISO is a JSON file containing various constraints related to the application and its execution. The main component of the AISO is the Requests Manager, which is responsible for processing the data provided by the end user regarding the execution of an application, as well as the telemetry data collected by the telemetry framework. The other components of the AISO are the translation mechanism and the prediction mechanism. These two mechanisms make decisions about deployment scenarios after analyzing the telemetry data (e.g., compute, memory, disk, network, and hardware information) and user parameters based on machine learning models (ML).

SERRANO contributes to application deployment providing services that target trusted execution, workload isolation, and lightweight virtualization.

vAccel framework: To achieve isolated and private execution of the hardware accelerators on the platform's disaggregated infrastructure, the vAccel framework is used to virtualize and deploy the hardware accelerators in multi-tenant environments. vAccel decouples the user application from the HLS and CUDA kernels. The actual hardware-specific code that implements these functions for a given hardware device is provided in the form of plugins that are loaded at runtime. Consequently, the deployed applications can be migrated from one host to another without the need to modify or recompile the code. At the same time, its modular nature prevents user code from running on shared accelerators. Only the code contained in the plugin is executed on the hardware accelerator, which enables secure hardware execution.

A.4. Experimental Results

Automatic HLS Optimization: The automatic HLS optimization scheme was used to optimize the Kalman filter algorithm provided by the high-performance fintech analysis use case. The proposed methodology is compared to a) the HLS optimizations performed by the Vitis Unified Software platform (version 2021.1), b) a human expert, and c) an automated optimizer that randomly navigates the design space of directives. The proposed NSGA-II based (GenOpt) and the random (RandOpt) optimizers operate for 12 hours. The optimization process targets the Xilinx MPSoC ZCU104 FPGA available on the SERRANO platform. We synthesized the input source code using the HLS optimizations performed by Vitis and determined the latency and resources of the output design. This optimization scheme results in an infeasible design (BRAM>100%), showing that it cannot always account for the architecture of the target device. GenOpt achieves 40% and 27% lower latency compared to RandOpt (1.023 ms) and human expert (1.009 ms), respectively, with no significant difference in resource usage. Consequently, the proposed optimization method outperforms both the naive automatic DSE approach and the human expert in terms of latency and yields designs that take into account the available resources without human intervention.

Automatic CUDA Optimization: The automatic CUDA optimization method was evaluated using PolyBench-ACC, an open-source benchmark suite that includes CUDA kernels from data mining, linear algebra, and stencils. The proposed scheme also targets various edge-cloud GPUs from Nvidia (e.g., Xavier NX, T4) available on the SERRANO platform. To determine the best predictor for the block coarsening factor, several regression models were evaluated using MSE and the R^2 metric. For the model evaluation phase, the 10% of PolyBench-ACC suite was used. XGBoost has the highest prediction accuracy with an MSE of 0.02 and R^2 metric of 0.88. Figure below shows the speedup of the optimized source code targeted to the Nvidia T4 GPU for the evaluation dataset applications. As shown, the automatic CUDA optimizer achieves an average speedup of 3.1x compared to the native implementations.

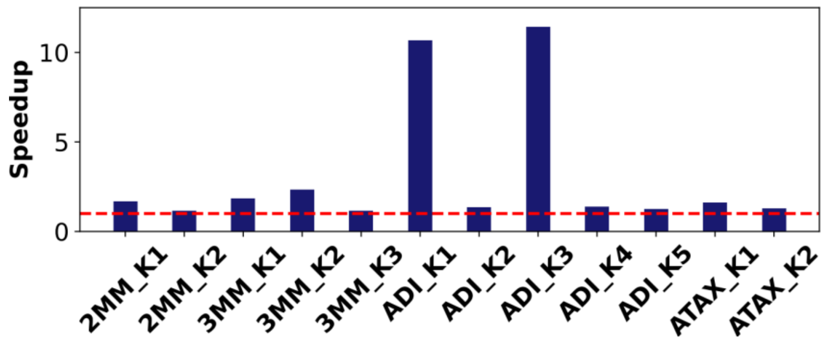


Figure A.4-1. CUDA automatic optimizer evaluation

Dynamic Memory Management in HLS: The implemented garbage collector and defragmentation methodology were evaluated on the Alveo U200 FPGA of the SERRANO platform to highlight their effectiveness in improving the memory efficiency of the platform under different threshold Θ and for different number of running accelerators. Figure below shows the percentage of fragmentation-induced memory allocation errors at different thresholds Θ when a different number (from 1 to 10) of accelerators run in parallel on the same platform and share a single heap. Note that the HLS accelerators K-means and moving average were run 10,000 times and statistical analysis was performed. The black reference lines correspond to HLS accelerators that do not use the garbage collector. At lower Θ values, the garbage collector is activated more frequently, leading to a significant reduction in fragmentation levels and, consequently, fragmentation-related allocation failures. However, frequent execution of the garbage collector introduces latency in the execution of the accelerators.

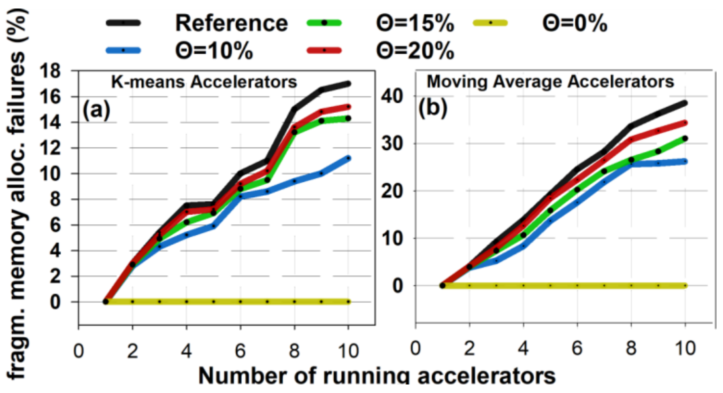


Figure A.4-2. Percentage of the fragmentation induced allocation failures of the overall 10,000 parallel executions of the (a) K-means HLS accelerators and (b) moving average HLS accelerators.

The vAccel framework was evaluated using two applications from the high-performance fintech analysis use-case: (i) the Savitzky-Golay filter and (ii) the Black-Scholes algorithm. These applications were virtualized via the vAccel framework, ported via the *vsock* socket, and executed on the Alveo U50 FPGA. Figure below shows the energy consumption and execution time speedup of the two HLS accelerators when running on the selected platform (patterned in green) and when running the same designs after they are virtualized by vAccel (blue). Virtualizing the designs results in negligible degradation in the energy consumption and speedup of the accelerators, ranging between 2% and 4% compared to the non-virtualized designs.

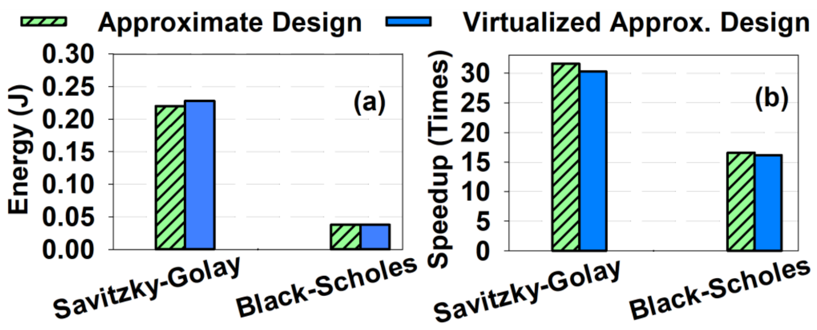


Figure A.4-3. Accelerator energy consumption (a) and execution time speedup (b) on the Alveo U50 for the virtualized/non-virtualized designs.

In this paper, we introduce the SERRANO platform and its software, orchestration, and deployment services, focusing on its methods for automated GPU/FPGA acceleration and efficient, isolated, and secure deployments. By evaluating these services against representative use cases, we highlight SERRANO 's ability to simplify the development and deployment process without sacrificing performance, underscoring its importance for heterogeneous resource adoption in the edge-cloud computing continuum.

Publications

Journals

[1] Dimitrios Danopoulos, Georgios Zervakis, Kostas Siozios, Dimitrios Soudris, and Jörg Henkel, “Adapt: Fast emulation of approximate dnn accelerators in pytorch,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2022.

[2] Dimitrios Danopoulos, Christoforos Kachris, and Dimitrios Soudris, “Utilizing cloud fpgas towards the open neural network standard,” Sustainable Computing: Informatics and Systems, vol. 30, p. 100520, 2021.

Book Chapters

[1] Dimitrios Danopoulos & Christoforos Kachris & Dimitrios Soudris. (2019). A Quantitative Comparison for Image Recognition on Accelerated Heterogeneous Cloud Infrastructures. 10.1201/9780429399602-8.

Conferences

[1] Vellas, Simon & Psomas, Bill & Karadima, Kalliopi & Danopoulos, Dimitrios & Paterakis, Alexandros & Lentaris, George & Soudris, Dimitrios & Karantzalos, Konstantinos. (2024). Evaluation of Resource-Efficient Crater Detectors on Embedded Systems.

[2] Dimitrios Danopoulos, Georgios Zervakis, and Dimitrios Soudris, “Simulating inexact dnns for use in custom hardware architectures,” in 2024 Panhellenic Conference on Electronics & Telecommunications (PACET). IEEE, 2024, pp. 1–4.

[3] A. Nanos, A. Kretsis, C. Mainas, G. Ntouskos, A. Ferikoglou, D. Danopoulos, A. Kokkinis, D. Masouros, K. Siozios, P. Soumplis et al., “Hardware-accelerated faas for the edge-cloud continuum,” in 2023 IEEE 31st International Conference on Network Protocols (ICNP). IEEE, 2023, pp. 1–6.

[4] A. Ferikoglou, E. Varvarigos, K. Siozios, P. Kokkinos, I. Oroutzoglou, A. Kokkinis, D. Danopoulos, A. Kretsis, A. F. Gomez, D. Masouros´ et al., “Towards efficient hw acceleration in edge-cloud infrastructures,” Embedded Computer Systems: Architectures, Modeling, and Simulation, vol. 13227, no. IKEEXREF-348314, pp. 354–367, 2022.

[5] A. Leftheriotis, A. Tzomaka, D. Danopoulos, G. Lentaris, G. Theodoridis, and D. Soudris, “Evaluating versal acap and conventional fpga platforms for ai inference,” in 2023 12th International Conference on Modern Circuits and Systems Technologies (MOCAST). IEEE, 2023, pp. 1–6.

[6] A. Ferikoglou, A. Kokkinis, D. Danopoulos, I. Oroutzoglou, A. Nanos, S. Karanastasis, M. Sipos, J. F. Ghotbi, J. J. V. Olmos, D. Masouros et al., “The serrano platform: Stepping towards seamless application development & deployment in the heterogeneous edge-cloud continuum,” in 2023 Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE, 2023, pp. 1–4.

[7] D. Danopoulos, I. Stamoulias, G. Lentaris, D. Masouros, I. Kanaropoulos, A. K. Kakolyris, and D. Soudris, “Lstm acceleration with fpga and gpu devices for edge computing applications in b5g mec,” in International Conference on Embedded Computer Systems. Springer International Publishing Cham, 2022, pp. 406–419.

[8] A. Kokkinis, A. Ferikoglou, I. Oroutzoglou, D. Danopoulos, D. Masouros, and K. Siozios, “Hw/sw acceleration of multiple workloads within the serrano’s computing continuum,” in International Conference on Embedded Computer Systems. Springer International Publishing Cham, 2022, pp. 394–405.

[9] I. Oroutzoglou, A. Kokkinis, A. Ferikoglou, D. Danopoulos, D. Masouros, and K. Siozios, “Optimizing savitzky-golay filter on gpu and fpga accelerators for financial applications,” in 2022 11th International Conference on Modern Circuits and Systems Technologies (MOCAST). IEEE, 2022, pp. 1–4.

[10] A. Ferikoglou, I. Oroutzoglou, A. Kokkinis, D. Danopoulos, D. Masouros, E. Chondrogiannis, A. F. Gomez, A. Kretsis, P. Kokkinos,´ E. Varvarigos et al., “Towards efficient hw acceleration in edge-cloud infrastructures: the serrano approach,” in International Conference on Embedded Computer Systems. Springer International Publishing Cham, 2021, pp. 354–367.

[11] I. Stratakos, E. A. Papatheofanous, D. Danopoulos, G. Lentaris, D. Reisis, and D. Soudris, “Towards sharing one fpga soc for both low-level phy and high-level ai/ml computing at the edge,” in 2021 IEEE International Mediterranean Conference on Communications and Networking (MeditCom). IEEE, 2021, pp. 76–81.

[12] A. Kokkinis, A. Ferikoglou, D. Danopoulos, D. Masouros, and K. Siozios, “Leveraging hw approximation for exploiting performance energy trade-offs within the edge-cloud computing continuum,” in High Performance Computing: ISC High Performance Digital 2021 International Workshops, Frankfurt am Main, Germany, June 24–July 2, 2021, Revised Selected Papers 36. Springer International Publishing, 2021, pp. 406–415.

[13] D. Danopoulos, K. Anagnostopoulos, C. Kachris, and D. Soudris, “Fpga acceleration of generative adversarial networks for image reconstruction,” in 2021 10th International Conference on Modern Circuits and Systems Technologies (MOCAST). IEEE, 2021, pp. 1–5.

[14] Danopoulos, D., Kachris, C., Soudris, D. (2021). Covid4HPC: A Fast and Accurate Solution for Covid Detection in the Cloud Using X-Rays. In: Derrien, S., Hannig, F., Diniz, P.C., Chillet, D. (eds) Applied Reconfigurable Computing. Architectures, Tools, and Applications. ARC 2021. Lecture Notes in Computer Science(), vol 12700. Springer, Cham. https://doi.org/10.1007/978-3-030-79025-7_25

[15] J. Violos, E. Psomakelis, D. Danopoulos, S. Tsanakas, and T. Varvarigou, “Using lstm neural networks as resource utilization predictors: The case of training deep learning models on the edge,” in Economics of Grids, Clouds, Systems, and Services: 17th International Conference, GECON 2020, Izola, Slovenia, September 15–17, 2020, Revised Selected Papers 17. Springer International Publishing, 2020, pp. 67–74.

[16] D. Danopoulos, C. Kachris and D. Soudris, "Automatic Generation of FPGA Kernels From Open Format CNN Models," 2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), Fayetteville, AR, USA, 2020, pp. 237-237, doi: 10.1109/FCCM48280.2020.00070.

[17] Danopoulos, Dimitrios & Kachris, Christoforos & Soudris, Dimitrios. (2019). FPGA Acceleration of Approximate KNN Indexing on High- Dimensional Vectors. 59-65. 10.1109/ReCoSoC48741.2019.9034938.

[18] D. Danopoulos, C. Kachris, and D. Soudris, “Approximate similarity search with faiss framework using fpgas on the cloud,” in International Conference on Embedded Computer Systems. Springer International Publishing Cham, 2019, pp. 373–386.

[19] D. Danopoulos, C. Kachris, and D. Soudris, “Acceleration of image classification with caffe framework using fpga,” in 2018 7th International Conference on Modern Circuits and Systems Technologies (MOCAST). IEEE, 2018, pp. 1–4.

Bibliography

[1] K. Rupp, *50 Years of Microprocessor Trend Data.*

[2] J. Molina, L. Heim, A. Ho, T. Besiroglu, M. Hobbhahn and P. Villalobos, *Compute Trends Across Three Eras of Machine Learning,* 2022.

[3] J. Duarte and others, "Fast inference of deep neural networks in FPGAs for particle physics," *JINST,* vol. 13, p. P07027, 2018.

[4] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *Bulletin of Mathematical Biophysics,* vol. 5, p. 115–133, December 1943.

[5] D. H. Hubel and T. N. Wiesel, *Brain and visual perception: The story of a 25-year collaboration,* Oxford University Press, 2005.

[6] P. Werbos, "Applications of advances in nonlinear sensitivity analysis," *System modeling and optimization,* p. 762–770, 1982.

[7] D. E. Rumelhart, G. E. Hinton and R. J. Williams, "Learning Internal Representations by Error Propagation," in *Parallel distributed processing: Explorations in the microstructure of cognition, Volume 1: Foundation*, D. E. Rumelhart, J. L. McClelland and the PDP research group, Eds., MIT Press, 1986.

[8] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard and L. D. Jackel, "Backpropagation Applied to Handwritten Zip Code Recognition," *Neural Computation,* vol. 1, pp. 541-551, 1989.

[9] A. Krizhevsky, I. Sutskever and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, Red Hook, NY, USA, 2012.

[10] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville and Y. Bengio, "Generative Adversarial Networks," *Advances in Neural Information Processing Systems*, vol. 3, June 2014.

[11] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. Gomez, L. Kaiser and I. Polosukhin, "Attention Is All You Need," June 2017.

[12] Computer History Museum, *13 Sextillion & Counting: The Long & Winding Road to the Most Frequently Manufactured Human Artifact in History,* 2018.

[13] C. Åleskog, H. Grahn and A. Borg, "Recent Developments in Low-Power AI Accelerators: A Survey," *Algorithms*, vol. 15, 2022.

[14] S. Bianco, R. Cadene, L. Celona and P. Napoletano, *Benchmark Analysis of Representative Deep Neural Network Architectures*, 2018.

[15] L. Xiao, Y. Bahri, J. N. Sohl-Dickstein, S. S. Schoenholz and J. Pennington, "Dynamical Isometry and a Mean Field Theory of CNNs: How to Train 10, 000-Layer Vanilla Convolutional Neural Networks," in *International Conference on Machine Learning*, 2018.

[16] M. Horowitz, "Energy table for 45nm process," *Stanford VLSI wiki.*

[17] T. P. Morgan, *Diving Deep Into The Nvidia Ampere GPU Architecture*, 2020.

[18] NVIDIA, *TensorRT*, Accessed: July 14, 2023.

[19] S. Sharma, *Speed up TensorFlow Inference on GPUs with TensorRT*, 2018.

[20] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu and X. Zheng, *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*, 2015.

[21] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai and S. Chintala, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, Red Hook, NY, USA, Curran Associates Inc., 2019.

[22] m. n. md isa and S. A. Zainol Murad, "Field Programmable Gate Array (FPGA): From Conventional to Modern Architectures," 2015, p. 53.

[23] Xilinx, *Vitis AI: Xilinx's development stack for AI inference on Xilinx hardware platforms.*

[24] Google Inc., *Tensor Processing Unit*, Year.

[25] 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, pp. 3105-3117, 2016.

[26] S. Lie, "Cerebras Architecture Deep Dive: First Look Inside the HW/SW Co-Design for Deep Learning : Cerebras Systems," in *2022 IEEE Hot Chips 34 Symposium (HCS)*, 2022.

[27] E. Dagan, "Habana Labs Purpose-Built AI Inference and Training Processor Architectures: Scaling AI Training Systems Using Standard Ethernet With Gaudi Processor," *IEEE Micro,* vol. PP, pp. 1-1, February 2020.

[28] D. Abts, J. Ross, J. Sparling, M. Wong-VanHaren, M. Baker, T. Hawkins, A. Bell, J. Thompson, T. Kahsai, G. Kimmell, J. Hwang, R. Leslie-Hurd, M. Bye, E. R. Creswick, M. Boyd, M. Venigalla, E. Laforge, J. Purdy, P. Kamath, D. Maheshwari, M. Beidler, G. Rosseel, O. Ahmad, G. Gagarin, R. Czekalski, A. Rane, S. Parmar, J. Werner, J. Sproch, A. Macias and B. Kurtz, "Think Fast: A Tensor Streaming Processor (TSP) for Accelerating Deep Learning Workloads," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020.

[29] J. Henkel, H. Khdr, S. Pagani and M. Shafique, "New trends in dark silicon," 2015.

[30] A. Reuther, P. Michaleas, M. Jones, V. Gadepally, S. Samsi and J. Kepner, "AI and ML Accelerator Survey and Trends," in *2022 IEEE High Performance Extreme Computing Conference (HPEC)*, 2022.

[31] M. M. Research, *Nvidia vs. AMD - who will win AI in the short run?,* 2023.

[32] S. Ren, K. He, R. Girshick and J. Sun, "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks," in *Advances in Neural Information Processing Systems*, 2015.

[33] J. Redmon, S. Divvala, R. Girshick and A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection," 2016.

[34] M. Aamir, N. Goli and T. Aamodt, "Modeling Deep Learning Accelerator Enabled GPUs," 2019.

[35] A. Boutros, E. Nurvitadhi, R. Ma, S. Gribok, Z. Zhao, J. C. Hoe, V. Betz and M. Langhammer, "Beyond Peak Performance: Comparing the Real Performance of AI-Optimized FPGAs and GPUs," in *2020 International Conference on Field-Programmable Technology (ICFPT),* 2020.

[36] D. Danopoulos, I. Stamoulias, G. Lentaris, D. Masouros, I. Kanaropoulos, A. K. Kakolyris and D. Soudris, "LSTM Acceleration with FPGA and GPU Devices for Edge Computing Applications in B5G MEC," in *Embedded Computer Systems: Architectures, Modeling, and Simulation*, Cham, 2022.

[37] X.-Y. Liu, Z. Zhang, Z. Wang, H. Lu, X. Wang and A. Walid, "High-Performance Tensor Learning Primitives Using GPU Tensor Cores," *IEEE Transactions on Computers,* vol. 72, pp. 1733-1746, 2023.

[38] S. Venkataramani, V. Srinivasan, W. Wang, S. Sen, J. Zhang, A. Agrawal, M. Kar, S. Jain, A. Mannari, H. Tran, Y. Li, E. Ogawa, K. Ishizaki, H. Inoue, M. Schaal, M. Serrano, J. Choi, X. Sun, N. Wang, C.-Y. Chen, A. Allain, J. Bonano, N. Cao, R. Casatuta, M. Cohen, B. Fleischer, M. Guillorn, H. Haynie, J. Jung, M. Kang, K.-h. Kim, S. Koswatta, S. Lee, M. Lutz, S. Mueller, J. Oh, A. Ranjan, Z. Ren, S. Rider, K. Schelm, M. Scheuermann, J. Silberman, J. Yang, V. Zalani, X. Zhang, C. Zhou, M. Ziegler, V. Shah, M. Ohara, P.-F. Lu, B. Curran, S. Shukla, L. Chang and K. Gopalakrishnan, "RaPiD: AI Accelerator for Ultra-low Precision Training and Inference," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021.

[39] F. Sunny, A. Mirza, M. Nikdast and S. Pasricha, *ROBIN: A Robust Optical Binary Neural Network Accelerator,* 2021.

[40] J. Turley, "GAP9 for ML at the Edge – EEJourna," 2020.

[41] Y. Li, S. Xu, B. Zhang, X. Cao, P. Gao and G. Guo, "Q-ViT: Accurate and Fully Quantized Low-bit Vision Transformer," *Advances in Neural Information Processing Systems,* 2022.

[42] Z. Liu, Y. Wang, K. Han, W. Zhang, S. Ma and W. Gao, "Post-Training Quantization for Vision Transformer," *Advances in Neural Information Processing Systems,* 2021.

[43] S. Kim, A. Gholami, Z. Yao, M. W. Mahoney and K. Keutzer, "I-BERT: Integer-only BERT Quantization," *International Conference on Machine Learning (Accepted),* 2021.

[44] Y. Li, L. Zichuan, K. Xu, H. Yu and F. Ren, "A 7.663-TOPS 8.2-W Energy-efficient FPGA Accelerator for Binary Convolutional Neural Networks (Abstract Only)," 2017.

[45] M. Xia, Z. Huang, L. Tian, H. Wang, V. Chang, Y. Zhu and S. Feng, "SparkNoC: An Energy-efficiency FPGA-based Accelerator Using Optimized Lightweight CNN for Edge Computing," *Journal of Systems Architecture,* vol. 115, p. 101991, January 2021.

[46] A. Podili, C. Zhang and V. Prasanna, "Fast and efficient implementation of Convolutional Neural Networks on FPGA," 2017.

[47] Y. Guan, H. Liang, N. Xu, W. Wang, S. Shi, X. Chen, G. Sun, W. Zhang and J. Cong, "FP-DNN: An Automated Framework for Mapping Deep Neural Networks onto FPGAs with RTL-HLS Hybrid Templates," *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM),* pp. 152-159, 2017.

[48] H. Li, X. Fan, L. Jiao, W. Cao, X. Zhou and L. Wang, "A high performance FPGA-based accelerator for large-scale convolutional neural

networks," 2016.

[49] C. Zhang, Z. Fang, P. Zhou, P. Pan and J. Cong, "Caffeine: Towards Uniformed Representation and Acceleration for Deep Convolutional Neural Networks," 2016.

[50] K. Guo, L. Sui, J. Qiu, J. Yu, J. Wang, S. Yao, S. Han, Y. Wang and H. Yang, "Angel-Eye: A complete design flow for mapping CNN onto embedded FPGA," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. PP, pp. 1-1, May 2017.

[51] Y. Zhang, J. Pan, X. Liu, H. Chen, D. Chen and Z. Zhang, "FracBNN: Accurate and FPGA-Efficient Binary Neural Networks with Fractional Activations," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays,* New York, NY, USA, 2021.

[52] Y. Li, T. Geng, A. Li and H. Yu, "BCNN: Binary complex neural network," *Microprocessors and Microsystems,* vol. 87, p. 104359, October 2021.

[53] Y. Wang, Y. Yang, F. Sun and A. Yao, "Sub-bit Neural Networks: Learning to Compress and Accelerate Binary Neural Networks," in *International Conference on Computer Vision (ICCV),* 2021.

[54] X. Sui, L. Qunbo, L. Zhi, B. Zhu, Y. Yang, Y. Zhang and Z. Tan, "A Hardware-Friendly High-Precision CNN Pruning Method and Its FPGA Implementation," *Sensors,* vol. 23, p. 824, January 2023.

[55] Q. Zhang, J. Cao, Y. Zhang, S. Zhang, Q. Zhang and D. Yu, "FPGA Implementation of Quantized Convolutional Neural Networks," in *2019 IEEE 19th International Conference on Communication Technology (ICCT),* 2019.

[56] C. Zhang and V. Prasanna, "Frequency Domain Acceleration of Convolutional Neural Networks on CPU-FPGA Shared Memory System," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays,* New York, NY, USA, 2017.

[57] C. Ding, S. Liao, Y. Wang, Z. Li, N. Liu, Y. Zhuo, C. Wang, X. Qian, Y. Bai, G. Yuan, X. Ma, Y. Zhang, J. Tang, Q. Qiu, X. Lin and B. Yuan, "CirCNN: Accelerating and Compressing Deep Neural Networks Using Block-Circulant Weight Matrices," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture,* New York, NY, USA, 2017.

[58] X. Wang, C. Wang, J. Cao, L. Gong and X. Zhou, "WinoNN: Optimizing FPGA-Based Convolutional Neural Network Accelerators Using Sparse Winograd Algorithm," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. 39, pp. 4290-4302, 2020.

[59] Y. Huang, J. Shen, Z. Wang, M. Wen and C. Zhang, "A High-efficiency

FPGA-based Accelerator for Convolutional Neural Networks using Winograd Algorithm," *Journal of Physics: Conference Series,* vol. 1026, p. 012019, May 2018.

[60] S. Basalama, A. Sohrabizadeh, J. Wang, L. Guo and J. Cong, "FlexCNN: An End-to-end Framework for Composing CNN Accelerators on FPGA," *ACM Transactions on Reconfigurable Technology and Systems,* vol. 16, December 2022.

[61] 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 *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO),* 2016.

[62] M. Blott, T. B. Preußer, N. J. Fraser, G. Gambardella, K. O'brien, Y. Umuroglu, M. Leeser and K. Vissers, "FINN-R: An End-to-End Deep-Learning Framework for Fast Exploration of Quantized Neural Networks," *ACM Trans. Reconfigurable Technol. Syst.,* vol. 11, December 2018.

[63] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam and D. Kalenichenko, "Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018.

[64] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv and Y. Bengio, "Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations," *J. Mach. Learn. Res.,* vol. 18, p. 6869–6898, January 2017.

[65] I. Chelombiev, D. Justus, D. Orr, A. Dietrich, F. Gressmann, A. Koliousis and C. Luschi, *GroupBERT: Enhanced Transformer Architecture with Efficient Grouped Structures*, 2021.

[66] V. Mrazek, S. Sarwar, L. Sekanina, Z. Vasicek and K. Roy, "Design of Power-Efficient Approximate Multipliers for Approximate Artificial Neural Networks," 2016.

[67] O. Spantidi, G. Zervakis, I. Anagnostopoulos, H. Amrouch and J. Henkel, "Positive/Negative Approximate Multipliers for DNN Accelerators," in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD),* Munich, 2021.

[68] V. Mrazek, Z. Vasicek, L. Sekanina, M. Hanif and M. Shafique, "ALWANN: Automatic Layer-Wise Approximation of Deep Neural Network Accelerators without Retraining," 2019.

[69] M. Pinos, V. Mrazek, F. Vaverka, Z. Vasicek and L. Sekanina, "Acceleration Techniques for Automated Design of Approximate Convolutional Neural Networks," *IEEE Journal on Emerging and Selected*

Topics in Circuits and Systems, vol. 13, pp. 212-224, 2023.

[70] F. Vaverka, V. Mrazek, Z. Vasicek and L. Sekanina, "TFApprox: Towards a Fast Emulation of DNN Approximate Hardware Accelerators on GPU," 2020.

[71] M. Taheri, M. Riazati, M. H. Ahmadilivani, M. Jenihhin, M. Daneshtalab, J. Raik, M. Sjodin and B. Lisper, "DeepAxe: A Framework for Exploration of Approximation and Reliability Trade-offs in DNN Accelerators," *2023 24th International Symposium on Quality Electronic Design (ISQED),* pp. 1-8, 2023.

[72] A. Kuzmin, M. van Baalen, Y. Ren, M. Nagel, J. W. T. Peters and T. Blankevoort, "FP8 Quantization: The Power of the Exponent," *ArXiv,* vol. abs/2208.09225, 2022.

[73] Y. Lecun, J. Denker and S. Solla, "Optimal Brain Damage," 1989.

[74] S. Han, J. Pool, J. Tran and W. J. Dally, "Learning Both Weights and Connections for Efficient Neural Networks," in *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1,* Cambridge, 2015.

[75] N. Megiddo and V. Sarkar, "Optimal Weighted Loop Fusion for Parallel Programs," in *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures*, New York, NY, USA, 1997.

[76] J. Johnson, M. Douze and H. Jégou, "Billion-scale similarity search with GPUs," *IEEE Transactions on Big Data,* vol. 7, p. 535–547, 2019.

[77] S. Mavridis, M. Pavlidakis, I. Stamoulias, C. Kozanitis, N. Chrysos, C. Kachris, D. Soudris and A. Bilas, "VineTalk: Simplifying software access and sharing of FPGAs in datacenters," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, 2017.

[78] C. Kachris and D. Soudris, "A survey on reconfigurable accelerators for cloud computing," in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, 2016.

[79] N. Kouiroukidis and G. Evangelidis, "The Effects of Dimensionality Curse in High Dimensional kNN Search," in *2011 15th Panhellenic Conference on Informatics*, 2011.

[80] A. Andoni, P. Indyk and I. P. Razenshteyn, "Approximate Nearest Neighbor Search in High Dimensions," *CoRR*, vol. abs/1806.09823, 2018.

[81] M. Norouzi and D. J. Fleet, "Cartesian K-Means," in *Proceedings of the 2013 IEEE Conference on Computer Vision and Pattern Recognition*, Washington, 2013.

[82] Y. Pu, J. Peng, L. Huang and J. Chen, "An Efficient KNN Algorithm

Implemented on FPGA Based Heterogeneous Computing System Using OpenCL," in *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, 2015.

[83] J. Zhang, J. Li and S. Khoram, "Efficient Large-Scale Approximate Nearest Neighbor Search on OpenCL FPGA," 2018.

[84] H. M. Hussain, K. Benkrid, A. T. Erdogan and H. Seker, "Highly Parameterized K-means Clustering on FPGAs: Comparative Results with GPPs and GPUs," in *2011 International Conference on Reconfigurable Computing and FPGAs*, 2011.

[85] Q. Chen, D. Li and C. Tang, "KNN matting," in *2012 IEEE Conference on Computer Vision and Pattern Recognition*, 2012.

[86] S. Liu, X. Liang, L. Liu, X. Shen, J. Yang, C. Xu, L. Lin and S. Yan, "Matching-CNN meets KNN: Quasi-parametric human parsing," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015.

[87] K. Fukunage and P. M. Narendra, "A Branch and Bound Algorithm for Computing k-Nearest Neighbors," *IEEE Trans. Comput.*, vol. 24, p. 750–753, July 1975.

[88] J. L. Bentley, "Multidimensional Binary Search Trees Used for Associative Searching," *Commun. ACM*, vol. 18, p. 509–517, September 1975.

[89] A. Gionis, P. Indyk and R. Motwani, "Similarity Search in High Dimensions via Hashing," in *Proceedings of the 25th International Conference on Very Large Data Bases*, San Francisco, CA, USA, 1999.

[90] M. Sharifzadehand and C. Shahabi, "Approximate Voronoi Cell Computation on Geometric Data Streams," March 2019.

[91] K. Li, W. Li, Z. Chen and Y. Liu, Computational Intelligence and Intelligent Systems, First ed., Springer Singapor, 2018, pp. 379-380.

[92] J. Kybic and I. Vnučko, "Approximate Best Bin First kd Tree All Nearest Neighbor Search with Incremental Updates," vol. 10, pp. 420-2, August 2010.

[93] Y. Chen, T. Guan and C. Wang, "Approximate Nearest Neighbor Search by Residual Vector Quantization," in *Sensors*, 2010.

[94] Xilinx, Inc, *SDAccel Development Environment*, 2018.

[95] H. Alqahtani, M. Kavakli-Thorne and D. G. Kumar Ahuja, "Applications of Generative Adversarial Networks (GANs): An Updated Review," *Archives of Computational Methods in Engineering*, December 2019.

[96] M. Ghasemzadeh, M. Samragh and F. Koushanfar, "ReBNet: Residual Binarized Neural Network," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2018.

[97] C. Farabet, B. Martini, P. Akselrod, S. Talay, Y. Lecun and E. Culurciello, "Hardware Accelerated Convolutional Neural Networks for Synthetic Vision Systems," 2010.

[98] D. Danopoulos, C. Kachris and D. Soudris, "Automatic Generation of FPGA Kernels From Open Format CNN Models," in *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2020.

[99] H. Alemdar, V. Leroy, A. Prost-Boucle and F. Pétrot, "Ternary neural networks for resource-efficient AI applications," 2017.

[100] D. Danopoulos, C. Kachris and D. Soudris, "FPGA Acceleration of Approximate KNN Indexing on High- Dimensional Vectors," in *2019 14th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, 2019.

[101] S. Kazeminia, C. Baur, A. Kuijper, B. Ginneken, N. Navab, S. Albarqouni and A. Mukhopadhyay, "GANs for Medical Image Analysis," *Artificial Intelligence in Medicine*, p. 101938, August 2020.

[102] S. O. Memik, A. K. Katsaggelos and M. Sarrafzadeh, "Analysis and FPGA implementation of image restoration under resource constraints," *IEEE Transactions on Computers,* vol. 52, pp. 390-399, 2003.

[103] S. Kumar and R. K. Jha, "An FPGA-Based Design for a Real-Time Image Denoising Using Fractional Integrator," *Multidimensional Systems and Signal Processing,* February 2020.

[104] S. Liu, C. Zeng, H. Fan, H. Ng, J. Meng, Z. Que, X. Niu and W. Luk, "Memory-Efficient Architecture for Accelerating Generative Networks on FPGA," in *2018 International Conference on Field-Programmable Technology (FPT)*, 2018.

[105] A. Yazdanbakhsh, M. Brzozowski, B. Khaleghi, S. Ghodrati, K. Samadi, N. Kim and H. Esmaeilzadeh, "FlexiGAN: An End-to-End Solution for FPGA Acceleration of Generative Adversarial Networks," 2018.

[106] M. I. Neuman, E. Y. Lee, S. Bixby, S. Diperna, J. Hellinger, R. Markowitz, S. Servaes, M. C. Monuteaux and S. S. Shah, "Variability in the interpretation of chest radiographs for the diagnosis of pneumonia in children," *Journal of Hospital Medicine,* vol. 7, pp. 294-298, 2012.

[107] H. DAVIES, M. S. C. DELE MD and E. L. A. I. N. E. E.-L., "Reliability of the chest radiograph in the diagnosis of lower respiratory infections in young children, The Pediatric Infectious Disease Journal," *The Pediatric Infectious Disease Journal,* vol. 15, pp. 600-604, 1996.

[108] R. Hopstaken, T. Witbraad, J. van Engelshoven and G. Dinant, "Inter-

observer variation in the interpretation of chest radiographs for pneumonia in community-acquired lower respiratory infections," *Clinical radiology,* vol. 59, pp. 743-52, September 2004.

[109] A. Mangal, S. Kalia, H. Rajgopal, K. Rangarajan, V. Namboodiri, S. Banerjee and C. Arora, *CovidAID: COVID-19 Detection Using Chest X-Ray,* 2020.

[110] L. Wang and A. Wong, *COVID-Net: A Tailored Deep Convolutional Neural Network Design for Detection of COVID-19 Cases from Chest Radiography Images*, 2020.

[111] J. Zhang, Y. Xie, G. Pang, Z. Liao, J. Verjans, W. Li, Z. Sun, J. He, Y. Li, C. Shen and Y. Xia, "Viral Pneumonia Screening on Chest X-rays Using Confidence-Aware Anomaly Detection," *IEEE Transactions on Medical Imaging,* vol. PP, pp. 1-1, November 2020.

[112] A. Channa, N. Popescu and N. U. R. Malik, "Robust Technique to Detect COVID-19 using Chest X-ray Images," 2020.

[113] R. Miranda Pereira, D. Bertolini, L. Teixeira, C. Silla and Y. Costa, "COVID-19 identification in chest X-ray images on flat and hierarchical classification scenarios," *Computer Methods and Programs in Biomedicine,* vol. 194, p. 105532, May 2020.

[114] R. Jain, M. Gupta, S. Taneja and D. Jude, "Deep learning based detection and analysis of COVID-19 on chest X-ray images," *Applied Intelligence,* pp. 1-11, October 2020.

[115] T. Pham, "A comprehensive study on classification of COVID-19 on computed tomography with pretrained convolutional neural networks," *Scientific Reports,* vol. 10, October 2020.

[116] D. Dansana, R. Kumar, A. Bhattacharjee, D. Jude, D. Gupta, A. Khanna and O. Castillo, "Early diagnosis of COVID-19-affected patients based on X-ray and computed tomography images using deep learning algorithm," *Soft Computing,* August 2020.

[117] M. Azemin, R. Hassan, M. I. Mohd Tamrin and M. Ali, "COVID-19 Deep Learning Prediction Model Using Publicly Available Radiologist-Adjudicated Chest X-Ray Images as Training Data: Preliminary Findings," *International Journal of Biomedical Imaging,* vol. 2020, pp. 1-7, August 2020.

[118] M. Chowdhury, T. Rahman, A. Khandakar, R. Mazhar, M. Kadir, Z. Mahbub, K. Islam, M. S. Khan, A. Iqbal, N. Al-Emadi, M. B. I. Reaz and M. Islam, "Can AI help in screening Viral and COVID-19 pneumonia?," *IEEE Access,* vol. 8, pp. 132665-132676, July 2020.

[119] A. R. Omondi and J. C. Rajapakse, FPGA Implementations of Neural

Networks, Berlin, Heidelberg: Springer-Verlag, 2006.

[120] D. Danopoulos, C. Kachris and D. Soudris, "Approximate Similarity Search with FAISS Framework Using FPGAs on the Cloud," 2019, pp. 373-386.

[121] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao and D. Burger, "A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services," *IEEE Micro,* vol. 35, pp. 10-22, May 2015.

[122] V. Sze, Y.-H. Chen, J. Einer, A. Suleiman and Z. Zhang, "Hardware for machine learning: Challenges and opportunities," 2017.

[123] J. Cong, Z. Fang, M. Lo, H. Wang, J. Xu and S. Zhang, "Understanding Performance Differences of FPGAs and GPUs," 2018.

[124] K. Abdelouahab, M. Pelcat, J. Sérot and F. Berry, "Accelerating CNN inference on FPGAs: A Survey," *CoRR,* vol. abs/1806.01683, 2018.

[125] B. Kreis, N. Tran and J. Duarte, *Machine learning in FPGAs using HLS,* GitHub, 2019.

[126] W. You, D. Chen and C. Wu, "A Flexible DNN Accelerator Design with Layer Pipeline for FPGAs," in *2019 6th International Conference on Information Science and Control Engineering (ICISCE)*, 2019.

[127] C. Hao, X. Zhang, Y. Li, S. Huang, J. Xiong, K. Rupnow, W.-m. Hwu and D. Chen, "FPGA/DNN Co-Design: An Efficient Design Methodology for IoT Intelligence on the Edge," 2019.

[128] S. Huang, C. Pearson, R. Nagi, J. Xiong, D. Chen and W. Hwu, "Accelerating Sparse Deep Neural Networks on FPGAs," in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, 2019.

[129] J. Si and S. L. Harris, "Handwritten digit recognition system on an FPGA," in *2018 IEEE 8th Annual Computing and Communication Workshop and Conference (CCWC)*, 2018.

[130] H. M. Makrani and H. Homayoun, "MeNa: A memory navigator for modern hardware in a scale-out environment," in *2017 IEEE International Symposium on Workload Characterization (IISWC),* 2017.

[131] P. Gysel, M. Motamedi and S. Ghiasi, "Hardware-oriented Approximation of Convolutional Neural Networks," *CoRR,* vol. abs/1604.03168, 2016.

[132] S. Han, H. Mao and W. Dally, "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding," 2016.

[133] D. Danopoulos, C. Kachris and D. Soudris, "Utilizing cloud FPGAs towards the open neural network standard," *Sustainable Computing: Informatics and Systems,* vol. 30, p. 100520, 2021.

[134] D. T. Nguyen, T. N. Nguyen, H. Kim and H.-J. Lee, "A High-Throughput and Power-Efficient FPGA Implementation of YOLO CNN for Object Detection," *IEEE Trans. VLSI Syst.,* vol. 27, pp. 1861-1873, 2019.

[135] 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. Killebrew, 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 Processing Unit," in *Int. Symp. Computer Architecture,* 2017.

[136] G. Armeniakos, G. Zervakis, D. Soudris and J. Henkel, "Hardware Approximate Techniques for Deep Neural Network Accelerators: A Survey," *ACM Comput. Surv.*, March 2022.

[137] 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 Exhibition*, 2017.

[138] Z. G. Tasoulas, G. Zervakis, I. Anagnostopoulos, H. Amrouch and J. Henkel, "Weight-Oriented Approximation for Energy-Efficient Neural Network Inference Accelerators," *IEEE Trans. Circuits Syst. I, Reg. Papers,* vol. 67, pp. 4670-4683, December 2020.

[139] N. Benaich and I. Hogarth, *State of AI Report 2020,* 2020.

[140] Y. Fan, X. Wu, J. Dong and Z. Qi, "AxDNN: Towards the Cross-Layer Design of Approximate DNNs," in *Asia and South Pacific Design Automation Conference,* 2019.

[141] P. Rek and L. Sekanina, "TypeCNN: CNN Development Framework With Flexible Data Types," in *Design, Automation Test in Europe Conference Exhibition*, 2019.

[142] P. Gysel, "Ristretto: Hardware-Oriented Approximation of Convolutional Neural Networks," *ArXiv,* vol. abs/1605.06402, 2016.

[143] C. De la Parra, A. Guntoro and A. Kumar, "ProxSim: GPU-based Simulation Framework for Cross-Layer Approximate DNN Optimization," in *Design, Automation Test in Europe Conference Exhibition*, 2020.

[144] NVIDIA, *Pytorch-quantization's documentation*, Nvidia, 2021.

[145] R. Krishnamoorthi, "Quantizing deep convolutional networks for efficient inference: A whitepaper," *ArXiv*, vol. abs/1806.08342, 2018.

[146] P. K. Gadosey, Y. Li and P. T. Yamak, "On Pruned, Quantized and Compact CNN Architectures for Vision Applications: An Empirical Study," in *Int. Conf. Artificial Intelligence, Information Processing and Cloud Computing*, 2019.

[147] D. Danopoulos, G. Zervakis, K. Siozios, D. Soudris and J. Henkel, "AdaPT: Fast Emulation of Approximate DNN Accelerators in PyTorch," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. PP, pp. 1-1, January 2022.

[148] X. He, L. Ke, W. Lu, G. Yan and X. Zhang, "AxTrain: Hardware-Oriented Neural Network Training for Approximate Inference," in *Proceedings of the International Symposium on Low Power Electronics and Design*, New York, NY, USA, 2018.

[149] Y. Li, S. Xu, B. Zhang, X. Cao, P. Gao and G. Guo, "Q-ViT: Accurate and Fully Quantized Low-bit Vision Transformer," in *Advances in Neural Information Processing Systems*, 2022.

[150] K. Wang, Z. Liu, Y. Lin, J. Lin and S. Han, "HAQ: Hardware-Aware Automated Quantization With Mixed Precision," in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019.

[151] Q. Lou, F. Guo, L. Liu, M. Kim and L. Jiang, "AutoQ: Automated Kernel-Wise Neural Network Quantization," *arXiv: Learning*, 2019.

[152] Z. Yao, Z. Dong, Z. Zheng, A. Gholami, J. Yu, E. Tan, L. Wang, Q. Huang, Y. Wang, M. W. Mahoney and K. Keutzer, "HAWQ-V3: Dyadic neural network quantization," in *ICML 2021*, 2021.

[153] Y. Ding, H. Qin, Q. Yan, Z. Chai, J. Liu, X. Wei and X. Liu, "Towards Accurate Post-Training Quantization for Vision Transformer," in *Proceedings of the 30th ACM International Conference on Multimedia*, New York, NY, USA, 2022.

[154] J. Gong, H. Saadat, H. Gamaarachchi, H. Javaid, X. S. Hu and S. Parameswaran, "ApproxTrain: Fast Simulation of Approximate Multipliers for DNN Training and Inference," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1-1, 2023.

[155] J. Wang, W. Wang, R. Wang and W. Gao, "Beyond Monte Carlo Tree

Search: Playing Go with Deep Alternative Neural Network and Long-Term Evaluation," *Proceedings of the AAAI Conference on Artificial Intelligence,* vol. 31, June 2017.

[156] H. Rakotoarison, M. Schoenauer and M. Sebag, "Automated Machine Learning with Monte-Carlo Tree Search (Extended Version)," in *International Joint Conference on Artificial Intelligence*, 2019.

[157] A. Raina, J. Cagan and C. McComb, "Learning to Design Without Prior Data: Discovering Generalizable Design Strategies Using Deep Learning and Tree Search," *Journal of Mechanical Design,* vol. 145, p. 031402, December 2022.

[158] Y. He, H. Li, T. Jin and F. S. Bao, "Circuit Routing Using Monte Carlo Tree Search and Deep Reinforcement Learning," in *2022 International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, 2022.

[159] P.-Y. Chen, B.-T. Ke, T.-C. Lee, I.-C. Tsai, T.-W. Kung, L.-Y. Lin, E.-C. Liu, Y.-C. Chang, Y.-L. Li and M. C.-T. Chao, "A Reinforcement Learning Agent for Obstacle-Avoiding Rectilinear Steiner Tree Construction," in *Proceedings of the 2022 International Symposium on Physical Design*, New York, NY, USA, 2022.

[160] C. Tang, K. Ouyang, Z. Wang, Y. Zhu, Y. Wang, W. Ji and W. Zhu, *Mixed-Precision Neural Network Quantization via Learned Layer-wise Importance*, 2022.

[161] Y. Bengio, P. Lamblin, D. Popovici, H. Larochelle and U. Montreal, "Greedy layer-wise training of deep networks," 2007.

[162] M. A. Javadi, H. Ghomashi, M. Taherinezhad, M. Nazarahari and R. Ghasemiasl, "Comparison of Monte Carlo Simulation and Genetic Algorithm in Optimal Wind Farm Layout Design in Manjil Site Based on Jensen Model," in *7th Iran Wind Energy Conference (IWEC2021)*, 2021.

[163] A. Uriarte and S. Ontanon, "Improving Monte Carlo Tree Search Policies in StarCraft via Probabilistic Models Learned from Replay Data," 2016.

[164] C. Dann, T. Lattimore and E. Brunskill, "Unifying PAC and Regret: Uniform PAC Bounds for Episodic Reinforcement Learning," in *Proceedings of the 31st International Conference on Neural Information Processing Systems,* Red Hook, NY, USA, 2017.

[165] E. J. Powley, D. Whitehouse and P. I. Cowling, "Bandits all the way down: UCB1 as a simulation policy in Monte Carlo Tree Search," in *2013 IEEE Conference on Computational Inteligence in Games (CIG),* 2013.

[166] C. Guo, G. Pleiss, Y. Sun and K. Q. Weinberger, *On Calibration of Modern Neural Networks,* arXiv, 2017.

[167] A. Komatsuzaki, "One Epoch Is All You Need," *CoRR,* vol. abs/1906.06669, 2019.

[168] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit and N. Houlsby, "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale," in *International Conference on Learning Representations,* 2021.

[169] H. Touvron, M. Cord, M. Douze, F. Massa, A. Sablayrolles and H. Jegou, "Training data-efficient image transformers & distillation through attention," in *International Conference on Machine Learning*, 2021.

[170] Z. Liu, Y. Lin, Y. Cao, H. Hu, Y. Wei, Z. Zhang, S. Lin and B. Guo, "Swin Transformer: Hierarchical Vision Transformer using Shifted Windows," *2021 IEEE/CVF International Conference on Computer Vision (ICCV),* pp. 9992-10002, 2021.

[171] A. Hatamizadeh, H. Yin, J. Kautz and P. Molchanov, "Global Context Vision Transformers," *arXiv preprint arXiv:2206.09959*, 2022.

[172] M. Nagel, M. Fournarakis, R. A. Amjad, Y. Bondarenko, M. van Baalen and T. Blankevoort, *A White Paper on Neural Network Quantization*, 2021.

[173] W. Shi and others, "Edge computing: Vision and challenges," *IEEE internet of things journal,* vol. 3, p. 637–646, 2016.

[174] D. Diamantopoulos and others, "Plug&chip: A framework for supporting rapid prototyping of 3d hybrid virtual socs," *ACM Transactions on Embedded Computing Systems (TECS),* vol. 13, p. 1–25, 2014.

[175] S. B. Imandoust and M. Bolandraftar, "Application of K-nearest neighbor (KNN) approach for predicting economic events theoretical background," *Int J Eng Res Appl*, vol. 3, pp. 605-610, January 2013.

[176] D. Danopoulos, C. Kachris and D. Soudris, "Acceleration of image classification with Caffe framework using FPGA," in *2018 7th International Conference on Modern Circuits and Systems Technologies (MOCAST)*, 2018.

[177] S. Singh, A. Khamparia, D. Gupta, P. Tiwari, C. Moreira, R. Damasevicius and V. Albuquerque, "A Novel Transfer Learning Based Approach for Pneumonia Detection in Chest X-ray Images," *Applied Sciences*, vol. 10, p. 559, January 2020.

[178] G. Zervakis, O. Spantidi, I. Anagnostopoulos, H. Amrouch and J. Henkel, "Control Variate Approximation for DNN Accelerators," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, 2021.

[179] R. Garcia, F. Asgarinejad, B. Khaleghi, T. Rosing and M. Imani, "TruLook: A Framework for Configurable GPU Approximation," in *2021*

Design, Automation & Test in Europe Conference & Exhibition (DATE), 2021.