ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Exploring Kernel Approximations for TinyML Inference Acceleration on Microcontrollers

---

Συγγραφέας:
Γεώργιος Μέντζος

Επιβλέπων:
Δημήτριος Σούντρης
Καθηγητής

ΕΡΓΑΣΤΗΡΙΟ ΜΙΚΡΟΥΠΟΛΟΓΙΣΤΩΝ ΚΑΙ ΨΗΦΙΑΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Οκτώβριος 2023

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ΕΡΓΑΣΤΗΡΙΟ ΜΙΚΡΟΥΠΟΛΟΓΙΣΤΩΝ ΚΑΙ ΨΗΦΙΑΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Exploring Kernel Approximations for TinyML Inference Acceleration on Microcontrollers

---

Συγγραφέας:
Γεώργιος Μέντζος

Επιβλέπων:
Δημήτριος Σούντρης
Καθηγητής

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 24η Οκτωβρίου 2023.

(Υπογραφή)	*(Υπογραφή)*	*(Υπογραφή)*
Δημήτριος Σούντρης Καθηγητής	Παναγιώτης Τσανάκας Καθηγητής	Σωτήριος Ξύδης Επίκουρος Καθηγητής

Οκτώβριος 2023

Πνευματικά Δικαιώματα

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 μή 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 ερμηνευθέι ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

Υπογραφή:

Ημερομηνία:

Εθνικό Μετσόβιο Πολυτεχνείο

Περίληψη

Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Διπλωματική Εργασία

Exploring Kernel Approximations for TinyML Inference Acceleration on Microcontrollers

του Γεώργιου Μέντζου

Η ραγδαία ανάπτυξη των always-on συσκευών IoT που βασίζονται σε μικροελεγκτές έχει ανοίξει πολυάριθμες εφαρμογές, από την έξυπνη παραγωγή έως την εξατομικευμένη υγειονομική περίθαλψη. Παρά την ευρεία υιοθέτηση των ενεργειακά αποδοτικών μικροελεγκτών (MCUs) στον τομέα Tiny Maching Learning (TinyML), αντιμετωπίζουν σημαντικούς περιορισμούς όσον αφορά τις επιδόσεις και τη μνήμη (RAM, Flash), ειδικά όταν εξετάζουμε βαθιά νευρωνικά δίκτυα DNN για σύνθετες εφαρμογές ταξινόμησης. Στην παρούσα εργασία, συνδυάζουμε τον προσεγγιστικό υπολογισμό (Approximate Computing) και τον σχεδιασμό πυρήνων λογισμικού για να επιταχύνουμε την εξαγωγή συμπερασμάτων προσεγγιστικών μοντέλων CNN σε MCUs. Το προσεγγιστικό kernel-based framework μας πρώτα αποσυσκευάζει (unpacking) τους τελεστές κάθε επιπέδου συνέλιξης και στη συνέχεια εκτελεί έναν offline υπολογισμό σημαντικότητας (significance) για κάθε τελεστή. Στη συνέχεια, μέσω μιας εξερεύνησης του χώρου σχεδίασης (Design Space Exploration), χρησιμοποιεί μια στρατηγική προσέγγισης που παραλείπει υπολογισμούς με βάση την υπολογιζόμενη σημαντικότητα, προσφέροντας διάφορους συμβιβασμούς μεταξύ μειωμένων υπολογισμών και ακρίβειας ταξινόμησης. Η αξιολόγησή μας, που πραγματοποιήθηκε σε μια πλακέτα STM32-Nucleo χρησιμοποιώντας τρία δημοφιλή CNN που εκπαιδεύτηκαν στο σύνολο δεδομένων CIFAR-10, αποδεικνύει ότι οι βέλτιστες κατά Pareto λύσεις μας μπορούν να αποφέρουν σημαντικά οφέλη. Σε σύγκριση με τις πιο σύγχρονες μεθόδους ακριβούς εξαγωγής συμπερασμάτων, η προσέγγισή μας επιτυγχάνει μείωση του χρόνου εκτέλεσης κατά 9% με σχεδόν μηδενική υποβάθμιση της απώλειας ακρίβειας Top-1 ($<$1%) σε MCUs με αρχιτεκτονική με δυνατότητα caching. Επιπλέον, όταν στοχεύουμε σε MCU χωρίς κρυφή μνήμη, η μείωση της καθυστέρησης αυξάνεται σημαντικά σε 37%, και πάλι σε βάρος της απώλειας ακρίβειας Top-1 λιγότερο από 1%. Οι διάφοροι συμβιβασμοί (trade-offs) που διερευνήθηκαν σε αυτή τη διατριβή, έχουν τη δυνατότητα να επιτρέψουν περισσότερες πρακτικές εφαρμογές και την ανάπτυξη βαθύτερων δικτύων σε MCU.

Λέξεις κλειδιά: Προσεγγιστικός Υπολογισμός, Μικροελεγκτές, Μικροσκοπική Μηχανική Μάθηση, Συνελικτικό Νευρωνικό Δίκτυο, Προσαρμοσμένη Σχεδίαση.

National Technical University of Athens

Abstract

Division of Computer Science
School of Electrical And Computer Engineering

Diploma Thesis

Exploring Kernel Approximations for TinyML Inference Acceleration on Microcontrollers

by George Mentzos

The rapid growth of always-on microcontroller-based IoT devices has opened up numerous applications, from smart manufacturing to personalized healthcare. Despite the widespread adoption of energy-efficient microcontroller units (MCUs) in the Tiny Maching Learning (TinyML) domain, they face significant limitations in terms of performance and memory (RAM, Flash), especially when considering deep networks for complex classification tasks. In this work, we combine approximate computing and software kernel design to accelerate the inference of approximate CNN models on MCUs. Our kernel-based approximation framework first unpacks the operands of each convolution layer and then performs an offline significance calculation for each operand. Subsequently, through a design space exploration, it employs a computation skipping approximation strategy based on the calculated significance, offering various trade-offs between reduced computations and classification accuracy. Our evaluation, conducted on an STM32-Nucleo board using three popular CNNs trained on the CIFAR-10 dataset, demonstrates that our Pareto optimal solutions can yield significant benefits. Compared to state-of-the-art exact inference methods, our approach achieves 9% reduction in latency with almost zero degradation in Top-1 accuracy loss ($<$1%) on MCUs with cache-enabled architecture. Furthermore, when targeting non-cached MCUs, the latency reduction is highly increased to 37%, again at the expense of less than 1% Top-1 accuracy loss. The various trade-offs explored in this thesis, hold the potential to enable more practical applications and the deployment of deeper networks on compact MCUs.

Keywords: Approximate Computing, Microcontrollers, TinyML, Convolutional Neural Network, Bespoke Design.

Acknowledgements

I would like to thank Professor Dimitrios Sountris for giving me the valuable opportunity and resources to conduct research on a topic of such great interest and to complete my Thesis at Microlab at the School of Electrical and Computer Engineering of the National Technical University of Athens. This work has instilled in me the value of work ethic, collaboration, professionalism as well as the importance of pursuing excellence.

I would also like to extend my gratitude to PhD candidate Giorgos Armeniakos for his continuous support, guidance and insights over the course of this work as well as Professor Georgios Zervakis whose experience proved invaluable at critical junctures of my research.

Though this thesis marks the end of a long chapter of my academic journey, the experiences I have gained will stay with me as I continue to strive for excellence in my life. Lastly, I could never overstate the importance of my family, friends and the people closest to me. Without their continuous support and guidance I would not have been able to reach this point in my life.

Contents

Declaration of Authorship v

Acknowledgements xi

1 Εκτεταμένη Περίληψη 1

2 Introduction 15
2.1 Introduction to TinyML . . . 15
2.2 Related Work . . . 16

3 Background 19
3.1 CNN models . . . 19
3.2 Quantization . . . 20
3.3 Approximate Computing . . . 21

4 ARM Cortex-M 25
4.1 Overview . . . 25
4.2 M-Profile Architectures . . . 25
4.3 Cortex-M Cores . . . 27
4.4 Single Instruction Multiple Data (SIMD) . . . 29
4.5 Flexible second operand (Operand2) . . . 31

5 Utilized Tools, Frameworks and APIs 33
5.1 TensorFlow . . . 33
5.2 CMSIS NN . . . 34
5.2.1 Operation support . . . 34
5.2.2 Data Transformation . . . 35
5.2.3 Matrix Multiplication . . . 36
5.3 STM32 Software Development Tools . . . 38
5.3.1 STM32CubeMX . . . 38
5.3.2 STM32CubeIDE . . . 38

6 Proposed Framework 39
6.1 Initial Optimization Experiments . . . 40

6.1.1 Loop Unrolling . . . 40
6.1.2 Loop Tilling . . . 41
6.2 Layer Based Code Unpacking . . . 42
6.3 Np2 Rounding . . . 44
6.4 Significance Aware Skipping . . . 45
6.5 Exhaustive Design Space Exploration . . . 49
6.6 Deployment Technique . . . 50

7 Experimental Results **53**
7.1 Testing Environment . . . 53
7.1.1 Development Hardware . . . 53
7.1.2 Development Software . . . 54
7.2 Evaluated Models . . . 55
7.2.1 LeNet-CIFAR10 . . . 55
7.2.2 AlexNet-CIFAR10 . . . 56
7.2.3 SqueezeNet-CIFAR10 . . . 57
7.3 Baseline Characteristics for the Examined Models . . . 59
7.4 Accuracy Agnostic Skipping . . . 60
7.5 Design Space Constraint through Histograms . . . 60
7.5.1 LeNet Layer Distribution . . . 60
7.5.2 AlexNet Layer Distribution . . . 61
7.5.3 SqueezeNet Layer Distribution . . . 63
7.6 Profiling Results . . . 63
7.6.1 Accuracy-Latency Pareto Space . . . 64
7.7 Deployment Results . . . 64
7.7.1 Cache Enabled . . . 64
7.7.2 Cache Disabled . . . 66
7.8 Comparison with Relevant Frameworks . . . 66

8 Conclussion and Future Work **69**

Bibliography **71**

List of Figures

1.1 Σχηματική απεικόνιση ενός βασικού νευρωνικού δικτύου συνελίξεων (CNN) 2
1.2 Υποστήριξη συνόλου εντολών στους επεξεργαστές Cortex-M 3
1.3 Επισκόπηση της δομής του πυρήνα του νευρωνικού δικτύου 4
1.4 Λεπτομερής επισκόπηση του προτεινόμενου πλαισίου 6
1.5 Ψευδοκώδικας του εσωτερικού βρόχου της υλοποίησης του πυρήνα mat mult με βάση το CMSIS (α) και με βάση τη βελτιστοποίηση αποσυμπίεσης κώδικα (β). Για το απλότητα, θεωρήθηκε πυρήνας 1×1 και κάθε βρόχος υπολογίζει το γινόμενο 1 στήλης και 1 γραμμής, δηλαδή 1 έξοδο. 7
1.6 Κυκλωματικό διάγραμμα της MCU STM32U575ZIT6Q 10
1.7 Παρετο Σπαςες φορ τηε Εξαμινεδ NN Αρςηιτεςτυρες 12

3.1 Schematic diagram of a basic convolutional neural network (CNN) architecture 19
3.2 (a) Convolution and (b) Max pooling and average pooling examples. 20
3.3 Timeline of notable DNNs 20
3.4 Hardware&Software Approximation Techniques 22

4.1 ARM processor family 26
4.2 Instruction Set support in the Cortex-M processors 28
4.3 Harvard Architecture Diagram 29
4.4 Block Diagram for Cortex-M Cores 30
4.5 32-bit word fitting 2x8-/4x16-bit values 30

5.1 Decision tree aiding quantization method selection 34
5.2 Optimized Kernels via TFLM: CMSIS-NN and TF Lite for MCUs 35
5.3 Overview of the neural network kernel structure 35
5.4 Operand order dependant 8- to 16-bit conversion 36
5.5 The SMLAD instruction 37
5.6 The inner-loop of matrix multiplication with 2 × 2 kernel 37

6.1 Proposed Framework Detailed Overview 40
6.2 Loop Unrolling Schematic 41
6.3 Loop Tilling Schematic 42

6.4 Pseudocode of the inner-loop of mat mult kernel implementation based on CMSIS (a) and based on the code unpacking optimization (b). For simplicity, 1×1 kernel has been considered and each loop computes the dot product result of 1 column and 1 row, i.e., 1 output. 43
6.5 Power Of Two C Implementation 45
6.6 Per Layer Data Collection 47

7.1 Circuit Diagram of the STM32U575ZIT6Q MCU 54
7.2 STM32U5 Nucleo-144 board 55
7.3 Reduction in cycles w.r.t. the total cycles required for AlexNet inference versus computation reduction (in MAC units) for the 1st(a), 2nd(b), 3rd(c), and 4th(d) convolution layer. Dots represent values obtained from an STM32-U575ZI-Q execution, while the remaining values are interpolated. 61
7.4 LeNet Layer 1 and Layer 2 Distributions 61
7.5 AlexNet Layer 1 to Layer 4 Distributions 62
7.6 SqueezeNet Layer 6, Layer 9 and Layer 12 Distributions 63
7.7 Pareto Spaces for the Examined NN Architectures 65

List of Tables

1.1 Αξιολόγηση των μοντέλων αναφοράς "ΙΦΑΡ-10 ΣχυεεζεΝετ, ΑλεξΝετ, ΛεΝετ σε ένα ΣΤΜ32-Νυςλεο 2000ΚΒ ΡΟΜ, 768ΚΒ ΡΑΜ 10
1.2 ὂμπαρισον ωιτη στατε-οφ-τηε-αρτ "ΜΣΙΣ [2] ανδ Ξ-"ΥΒΕ-ΑΙ [9] ωιτη τηε ἂςηε εναβλεδ φορ τωο "ΝΝς δεπλοψεδ ον αν στμ32υ575ζι-χ βοαρδ φιττινγ 2ΜΒ Φλαση ανδ 768ΚΒ ΡΑΜ. Τηρεε αςςυρασψ λοσς τηρεσηολδς ηαε βεεν ςονσιδερεδ. 13
1.3 ὂμπαρισον ωιτη στατε-οφ-τηε-αρτ "ΜΣΙΣ [2] ωιτη τηε ἂςηε δισαβλεδ φορ τηρεε "ΝΝς δεπλοψεδ ον αν στμ32υ575ζι-χ βοαρδ φιττινγ 2ΜΒ Φλαση ανδ 768ΚΒ ΡΑΜ. Τηρεε αςςυρασψ λοσς τηρεσηολδς ηαε βεεν ςονσιδερεδ. 13

2.1 Qualitative comparison of related frameworks. 17

7.1 LeNet Parameters Per Layer 56
7.2 AlexNet Parameters Per Layer 57
7.3 SqueezeNet Parameters Per Layer 59
7.4 Evaluation of our baseline CIFAR-10 SqueezeNet, AlexNet and LeNet on an STM32-Nucleo fitting 2000KB ROM and 768KB RAM 60
7.5 Comparison with state-of-the-art CMSIS [2] and X-CUBE-AI [9] with the Cache enabled for two CNNs deployed on an stm32u575zi-q board fitting 2MB Flash and 768KB RAM. Three accuracy loss thresholds have been considered. 66
7.6 Comparison with state-of-the-art CMSIS [2] with the Cache disabled for three CNNs deployed on an stm32u575zi-q board fitting 2MB Flash and 768KB RAM. Three accuracy loss thresholds have been considered. 66

Κεφάλαιο 1

Εκτεταμένη Περίληψη

Εισαγωγή

Τα τελευταία χρόνια, έχει σημειωθεί σημαντική αύξηση του πολλαπλασιασμού των συσκευών IoT, ιδίως εκείνων που βασίζονται σε μονάδες μικροελεγκτών (MCU) χαμηλού κόστους. Ενώ οι εμπορικές MCUs υπόσχονται να ικανοποιήσουν τις απαιτήσεις της ενεργειακής απόδοσης, ο περιορισμένος προϋπολογισμός των πόρων τους, συμπεριλαμβανομένης της περιορισμένης μνήμης (SRAM, flash) και της υψηλής καθυστέρησης, δρα απαγορευτικά για την ανάπτυξη της βαθιάς μάθησης σε μικρής κλίμακας υλικό. Ως εκ τούτου, υπάρχει επιτακτική ανάγκη για νέες βελτιστοποιήσεις για τη γεφύρωση του χάσματος μεταξύ απαιτούμενων και διαθέσιμων πόρων.

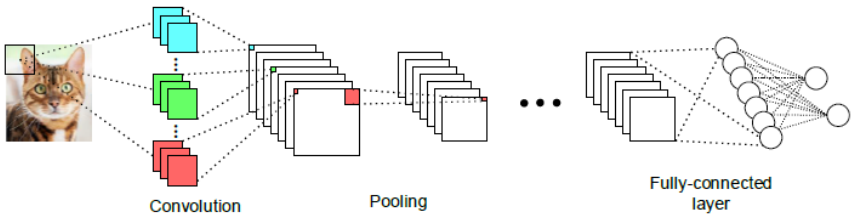
Ωστόσο, τα σύγχρονα πλαίσια δίνουν προτεραιότητα κυρίως στη μείωση του μεγέθους του μοντέλου ώστε να χωράει σε προκαθορισμένους περιορισμούς μνήμης. Δεν δεν αντιμετωπίζουν πρωτίστως τον στόχο της μείωσης της καθυστέρησης εξαγωγής συμπερασμάτων για ένα ήδη καλά προσαρμοσμένο, αν και μεγάλο, μοντέλο. Ως παράδειγμα, το TinyEngine απαιτεί σχεδόν 1,3 δευτερόλεπτα για να εκτελέσει ένα mcunet-in4 ImageNet σε MCU 160MHz.

Μοντέλα CNN

Τα [1] Convolutional Neural Networks (CNNs) αντιπροσωπεύουν μια υποκατηγορία των Τεχνητών Νευρωνικών Δικτύων (ANNs) που είναι προσαρμοσμένα για εφαρμογές που επεξεργάζονται οπτικά δεδομένα και αναμένουν μια εικόνα ως είσοδο. Μια τυπική δομή CNN Εικ. 1.1 αποτελείται από την εικόνα εισόδου που χρησιμεύει ως στρώμα εισόδου καθώς και από ένα συνδυασμό τριών πρόσθετων στρωμάτων: convolutional,pooling & dense.

Κβάντιση

Η κβάντιση [3] συνεπάγεται την εκτέλεση υπολογισμών νευρωνικών δικτύων (NN) χρησιμοποιώντας λιγότερα bits για ακρίβεια. Η πιο διαδεδομένη μέθοδος εκτελεί κβάντιση



Σχήμα 1.1: Σχηματική απεικόνιση ενός βασικού νευρωνικού δικτύου συνελίξεων (CNN)
Πηγή: [2]

στις παραμέτρους του NN μετά την εκπαίδευση μετατρέποντας τα αρχικά βάρη 32-bit κινητής υποδιαστολής σε ακέραιες τιμές 8-bit. Αυτή η τεχνική καταπολεμά μια από τις μεγαλύτερες προκλήσεις κατά την ανάπτυξη ενός NN σε μια μικροσκοπική πλατφόρμα με περιορισμένους πόρους, το μέγεθος του μοντέλου. Επιπλέον, η χρήση ακέραιων αριθμών 8-bit επιτρέπει αποδοτικούς υπολογισμούς σε συσκευές τύπου MCU, δεδομένου ότι, σε αντίθεση με τις πράξεις κινητής υποδιαστολής, συνήθως επιταχύνονται από το υλικό.

Συνολικά, αυτό μεταφράζεται σε μείωση του μεγέθους κατά 4 φορές, καθώς και σε αξιοσημείωτη βελτίωση της καθυστέρησης με ασήμαντη επίπτωση στην ακρίβεια πρόβλεψης.

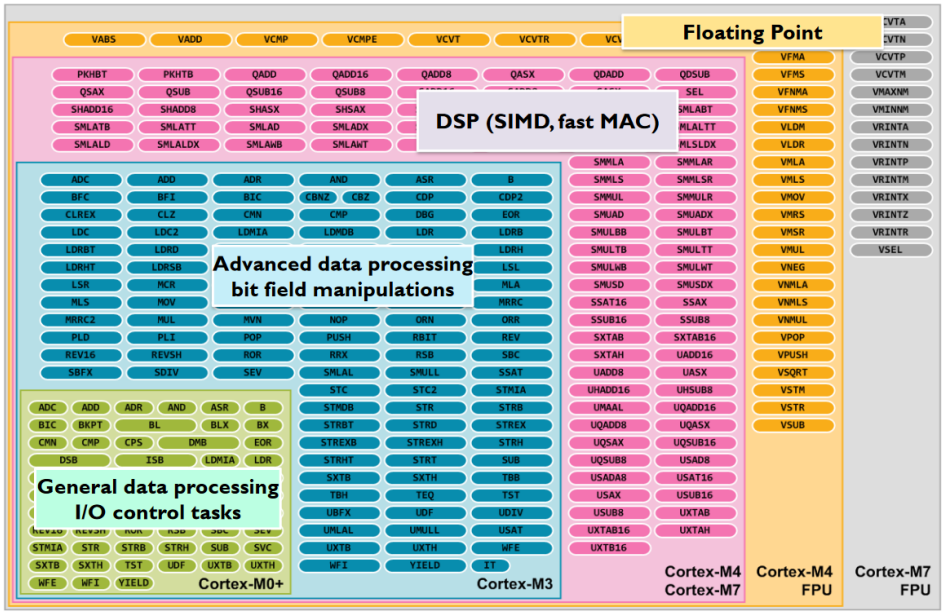
Προσεγγιστικός υπολογισμός

Ο προσεγγιστικός υπολογισμός (AC) [4] είναι μία υπολογιστική μέθοδος που εσκεμμένα ανταλλάσσει την ακρίβεια με βελτιωμένη ενεργειακή αποδοτικότητα και ταχύτητα. Η αποτελεσματικότητα αυτής της τεχνικής εξαρτάται από την εγγενή ανθεκτικότητα σε σφάλματα ορισμένων εφαρμογών (π.χ. όραση υπολογιστών). Θα πρέπει ωστόσο να σημειωθεί ότι η AC θα πρέπει να εφαρμόζεται μόνο σε μη κρίσιμες εφαρμογές δεδομένων όπου τα πιθανά σφάλματα είναι αποδεκτά.

ARM Cortex-M

Οι ARM Cortex-M είναι μια ομάδα 32-bit RISC ARM πυρήνων επεξεργαστών που είναι βελτιστοποιημένοι για χαμηλό κόστος και ενεργειακή απόδοση. Ως εκ τούτου, μπορούν να χρησιμοποιηθούν σε υπολογιστικά σενάρια με περιορισμένους πόρους είτε ενσωματωμένοι σε άλλους τύπους τσιπ (ελεγκτές συστημάτων, ελεγκτές αισθητήρων κ.λπ.) είτε συνηθέστερα ως τσιπ μικροελεγκτών.

Η αρχιτεκτονική ARM M-profile Εικ. 1.2 [5] προσφέρει ένα τυποποιημένο σύνολο εντολών και μοντέλων προγραμματισμού προσαρμοσμένων για ασφαλείς μικροεπεξεργαστές.



Σχήμα 1.2: Υποστήριξη συνόλου εντολών στους επεξεργαστές Cortex-M
Πηγή: [6]

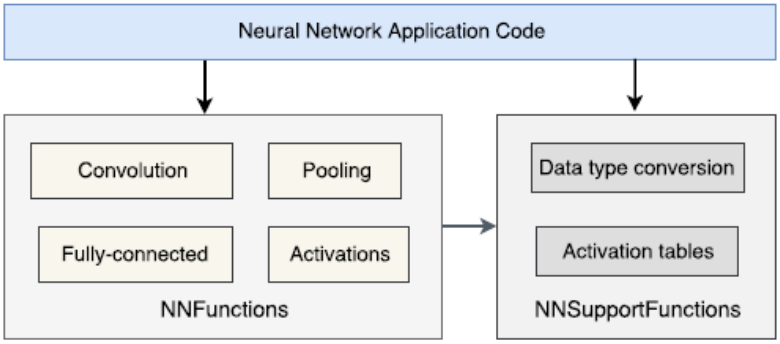
Αυτοί οι μικροεπεξεργαστές είναι ειδικά σχεδιασμένοι για να υπερέχουν σε βαθιά ενσωματωμένα συστήματα, δίνοντας προτεραιότητα στη χαμηλή κατανάλωση ενέργειας, την ελάχιστη καθυστέρηση και την εξαιρετικά προβλέψιμη λειτουργία.

Ενιαία εντολή Πολλαπλά δεδομένα (SIMD)

Οι πυρήνες Cortex-M που είναι εξοπλισμένοι με την επέκταση DSP (π.χ. M4, M33, M7) προσφέρουν ισχυρή υποστήριξη για εντολές SIMD. Αυτές οι εντολές λειτουργούν αποτελεσματικά σε τελεστές που έχουν πακεταριστεί σε καταχωρητές 32-bit, είτε είναι 8- είτε 16-bit. Κατά συνέπεια, αυτές οι εντολές SIMD δίνουν στον επεξεργαστή τη δυνατότητα να εκτελεί είτε διπλούς υπολογισμούς 16-bit είτε τετραπλούς υπολογισμούς 8-bit, ανάλογα με την εκάστοτε λειτουργία.

Ευέλικτος δεύτερος τελεστής (Operand2)

Πολλές εντολές γενικής επεξεργασίας δεδομένων ARM και Thumb διαθέτουν έναν ευέλικτο δεύτερο τελεστή, ο οποίος αναφέρεται ως Operand2 [7]. Ο Operand2 μπορεί να λάβει διάφορες μορφές, όπως μια σταθερή τιμή ή ένας καταχωρητής με προαιρετική πράξη μετατόπισης. Αυτή η ευελιξία επιτρέπει σε αυτές τις εντολές να εκτελούν ένα ευρύ φάσμα πράξεων, καθιστώντας τες ιδιαίτερα ευέλικτες για αριθμητικούς και λογικούς υπολογισμούς στα σύνολα εντολών ARM και Thumb, ενισχύοντας την αποτελεσματικότητα και την προσαρμοστικότητα του προγραμματισμού χαμηλού επιπέδου για ποικίλες εργασίες.



Σχήμα 1.3: Επισκόπηση της δομής του πυρήνα του νευρωνικού δικτύου
Πηγή: [2]

TensorFlow

Το TensorFlow [8] είναι ένα πλαίσιο μηχανικής μάθησης ανοικτού κώδικα που αναπτύχθηκε από την Google, το οποίο διευκολύνει τη δημιουργία, την ανάπτυξη και τη διαχείριση διαφόρων μοντέλων βαθιάς μάθησης. Υποστηρίζει μια ποικιλία χαρακτηριστικών που κυμαίνονται από API υψηλού επιπέδου, όπως το Keras, έως API χαμηλού επιπέδου για λεπτομερή συντονισμό των μοντέλων NN.

CMSIS NN

Το πλαίσιο TFLM περιλαμβάνει τους πυρήνες αναφοράς για την εκτέλεση διαφόρων λειτουργιών ενός μοντέλου νευρωνικών δικτύων. Αυτοί οι πυρήνες δεν έχουν βελτιστοποιηθεί, αλλά μάλλον χρησιμεύουν ως σημείο εκκίνησης για βελτιστοποιήσεις συγκεκριμένων στόχων. Μια τέτοια βελτιστοποιημένη βιβλιοθήκη είναι η CMSIS NN της Arm [2], η οποία αποτελείται από ένα σύνολο εξαιρετικά αποδοτικών πυρήνων νευρωνικών δικτύων σχεδιασμένων για την επίτευξη βέλτιστης απόδοσης και την ελαχιστοποίηση της χρήσης μνήμης κατά την εκτέλεση νευρωνικών δικτύων σε επεξεργαστές Arm Cortex-M Εικ. 1.3.

Εργαλεία ανάπτυξης λογισμικού STM32

Το X-CUBE-AI [9] είναι ένα ολοκληρωμένο πακέτο λογισμικού που αναπτύχθηκε από την STMicroelectronics, προσαρμοσμένο για εφαρμογές βασισμένες σε μικροελεγκτές STM32 που επιδιώκουν να ενσωματώσουν δυνατότητες τεχνητής νοημοσύνης και μηχανικής μάθησης. Παρέχει τη δυνατότητα στους προγραμματιστές να μετατρέπουν και να αναπτύσσουν αποτελεσματικά μοντέλα AI/ML σε ενσωματωμένα συστήματα με περιορισμένους πόρους. Αυτό το πακέτο προσφέρει βελτιστοποιημένες βιβλιοθήκες εξαγωγής συμπερασμάτων, υποστηρίζει δημοφιλή πλαίσια μηχανικής μάθησης και επιτρέπει την απρόσκοπτη ενσωμάτωση μοντέλων AI/ML σε έργα που βασίζονται στο STM32. Με έμφαση στη βελτιστοποίηση

των πόρων και τη συμβατότητα με το STM32CubeIDE, το X-CUBE-AI διευκολύνει την επεξεργασία AI/ML σε πραγματικό χρόνο και με χαμηλή καθυστέρηση σε συσκευές edge, καθιστώντας το ένα πολύτιμο εργαλείο για όσους επιδιώκουν να αξιοποιήσουν τη δύναμη της AI στις ενσωματωμένες εφαρμογές τους.

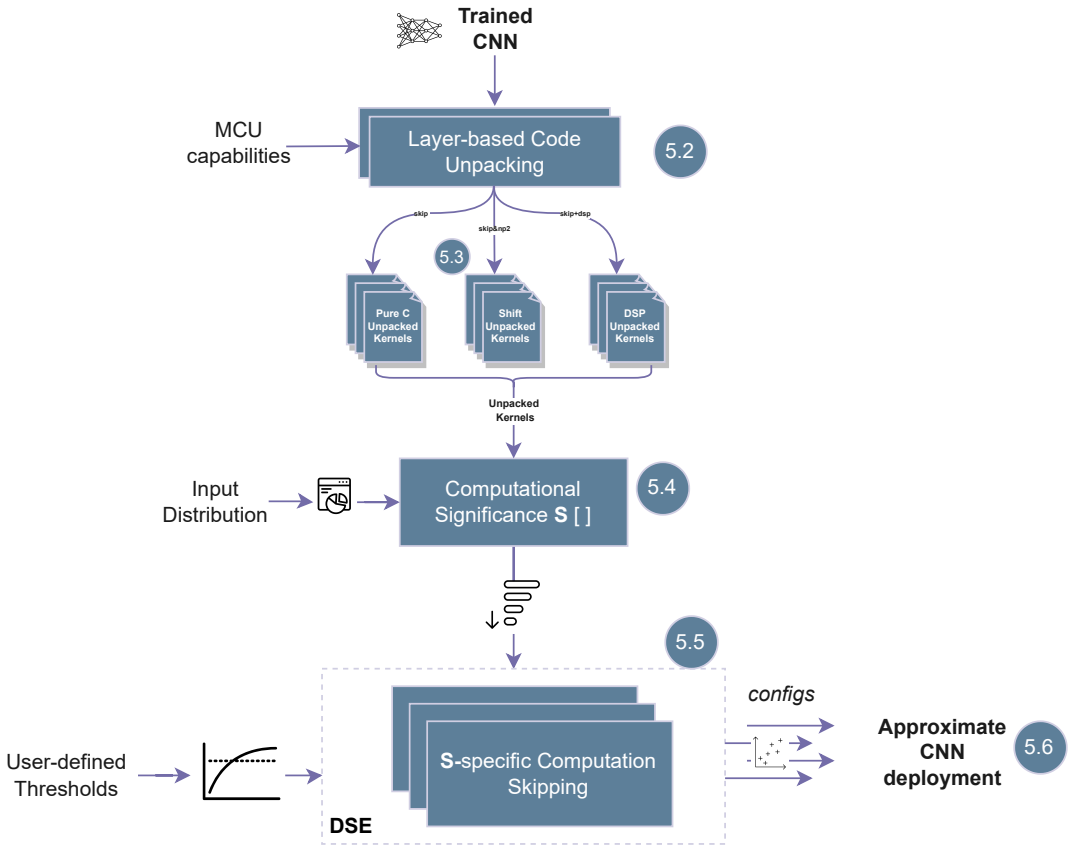
Το STM32CubeIDE [10] είναι ένα περιβάλλον ανάπτυξης C/C++ βασισμένο στο Eclipse® /CDT™, το οποίο προσφέρει λειτουργίες όπως η διαμόρφωση περιφερειακών, η δημιουργία κώδικα και η μεταγλώττιση κώδικα που ενσωματώνονται από το STM32CubeMX, καθώς και λειτουργίες εντοπισμού σφαλμάτων που βελτιώνουν τη διαδικασία εγκατάστασης και ανάπτυξης σε STM32 MCUs.

Προτεινόμενο Πλαίσιο

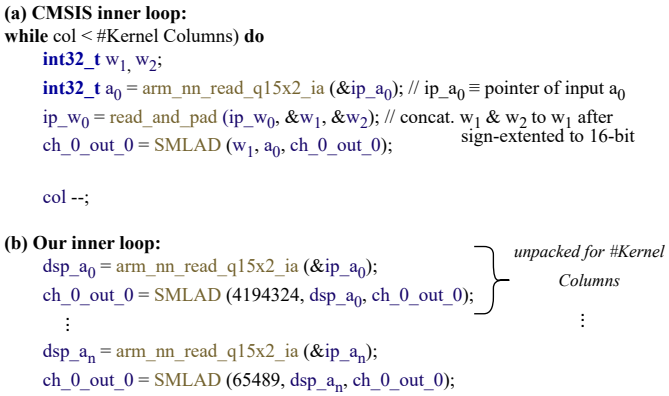
Σε όλη τη διάρκεια αυτής της διατριβής, διερευνήσαμε διάφορους τρόπους βελτιστοποίησης. Αυτές περιλαμβάνουν παραδοσιακές τεχνικές βελτιστοποίησης, όπως βελτιστοποιήσεις φωλιάσματος βρόχων, και μια λεπτομερή εξέταση του παραγόμενου κώδικα για πυρήνες εντατικής υπολογιστικής λειτουργίας. Επιπλέον, εμβαθύναμε σε νέες και προηγουμένως ανεξερεύνητες μεθόδους που περιλαμβάνουν τροποποιήσεις βασισμένες στον πυρήνα για την αξιοποίηση των αρχών του Προσεγγιστικού Υπολογισμού.

Για να ανταποκριθούμε στις μοναδικές απαιτήσεις των σεναρίων εφαρμογής μας, θεωρούμε τη CMSIS-NN ως τη βασική βιβλιοθήκη [11, 12]. Το πλαίσιο προσέγγισής μας βασίζεται στην προσαρμογή του παραγόμενου κώδικα, διασφαλίζοντας ότι υποστηρίζει μόνο τα βασικά στρώματα και λειτουργίες που απαιτούνται από το συγκεκριμένο μοντέλο. Σε αντίθεση με την CMSIS-NN και τις περισσότερες υπάρχουσες βιβλιοθήκες εξαγωγής συμπερασμάτων (π.χ. TF-Lite Micro [8]), μεταφέρουμε λειτουργίες που σχετίζονται με τις παραμέτρους της δομής του μοντέλου από τον χρόνο εκτέλεσης στον χρόνο μεταγλώττισης. Κατά συνέπεια, εκτελείται μόνο ο κώδικας που είναι απαραίτητος για το παρεχόμενο μοντέλο. Αυτό όχι μόνο βελτιώνει την αποδοτικότητα της εξαγωγής συμπερασμάτων αλλά και μειώνει τη χρήση μνήμης flash (σε ορισμένες περιπτώσεις έως και 30%). Αυτή η πρόσθετη διαθεσιμότητα μνήμης flash μας επιτρέπει να αποσυμπιέσουμε μεγαλύτερο αριθμό πυρήνων ή ακόμη και ολόκληρα στρώματα, ενισχύοντας έτσι την ακρίβεια της προσέγγισής μας.

Οι τυπικοί πυρήνες συνέλιξης στις MCUs υλοποιούνται συνήθως σε μορφή πίνακα, όπου οι είσοδοι και τα βάρη ανακτώνται από τη μνήμη χρησιμοποιώντας ένα συγκεκριμένο μοτίβο. Αυτό το μοτίβο περιλαμβάνει λεπτομέρειες όπως η σειρά με την οποία ανακτώνται τα στοιχεία δεδομένων, το μέγεθος του βήματος ή του βήματος για τη μετακίνηση μέσα στα δεδομένα και οποιαδήποτε αναγκαία συμπλήρωση ή προσαρμογή για να εξασφαλιστεί η σωστή ευθυγράμμιση των δεδομένων για τη συνέλιξη. Αντ΄ αυτού, πραγματοποιείται μια αυτοματοποιημένη αποσυμπίεση κώδικα με βάση το επίπεδο, όπου κάθε λειτουργία 'αποσυμπιέζεται' και περιλαμβάνεται ως εγγενής συνάρτηση στον τελικό παραγόμενο κώδικα. Μια απεικόνιση ενός πυρήνα πολλαπλασιασμού πινάκων παρόμοιου με την υλοποίηση του



Σχήμα 1.4: Λεπτομερής επισκόπηση του προτεινόμενου πλαισίου



Σχήμα 1.5: Ψευδοκώδικας του εσωτερικού βρόχου της υλοποίησης του πυρήνα mat mult με βάση το CMSIS (α) και με βάση τη βελτιστοποίηση αποσυμπίεσης κώδικα (β). Για το απλότητα, θεωρήθηκε πυρήνας 1×1 και κάθε βρόχος υπολογίζει το γινόμενο 1 στήλης και 1 γραμμής, δηλαδή 1 έξοδο.

CMSIS παρουσιάζεται στο Σχήμα 1.5α. Όπως φαίνεται στο Σχ. 1.5α, ο πυρήνας mat_mult εκτελεί δυναμικούς υπολογισμούς μεταξύ των εισόδων a_i και των βαρών w_i, με το *ch_0_out_0* να δηλώνει τη συσσωρευμένη έξοδο ανά κανάλι. Ο αντίστοιχος και αυτόματα παραγόμενος αποσυμπιεσμένος κώδικας απεικονίζεται στο Σχήμα 1.5β.

Τα αντίστοιχα οφέλη της διαδικασίας αποσυσκευασίας κώδικα περιλαμβάνουν τα εξής:

- Η σταθερή κατανομή βαρών σε κάθε τελεστή καταργεί την ανάγκη κατάλληλης προσαρμογής και φόρτωσης των βαρών κατά τη διαδικασία της συνέλιξης. Αυτό οδηγεί σε απλουστευμένες και προβλέψιμες, ως προς τον τύπο, πράξεις που μπορούν να προσαρμοστούν με βάση τις τιμές εισόδου για την ενίσχυση της ταχύτητας εξαγωγής συμπερασμάτων.

- Εξάλειψη της επιβάρυνσης της εντολής διακλάδωσης.

- Όπως αναφέρεται στην 5.2 Ο πυρήνας CMSIS mat mult υπολογίζει τα μερικά γινόμενα χρησιμοποιώντας την εντολή SMLAD (λογική SIMD), η οποία εκτελεί δύο 16-bit προσημασμένους πολλαπλασιασμούς, συσσωρεύοντας τα αποτελέσματα σε ένα 32-bit operand. Ως εκ τούτου, απαιτείται μια προεπεξεργασία για τη μετατροπή των δεδομένων σε τύπο δεδομένων 16 bit. Για το σκοπό αυτό, η δική μας σταθερού βάρους αντικατάσταση αποφεύγει αυτή τη χρονοβόρα λειτουργία. Εφόσον γνωρίζουμε εκ των προτέρων τις τιμές των βαρών, αυτό αποφεύγεται εύκολα με μια επεξεργασία εκτός σύνδεσης που περιλαμβάνει τη συνένωση δύο int16 (εκτεταμένο πρόσημο). int8 τιμές σε int16) βάρη. Για παράδειγμα, η τιμή ¨4194324¨ του Σχήματος 1.5β αντιπροσωπεύει w1=64 και w2=20

Η συσσώρευση κάθε καναλιού κατά τη διάρκεια του πολλαπλασιασμού του πίνακα (πυρήνας *mat_mult*) υπολογίζεται με βάση ένα σταθμισμένο άθροισμα και μια αρχικοποιημένη τιμή (bias):

$$Sum_c = b + \sum_{\forall i} a_i \cdot w_i, \tag{1.1}$$

όπου, b είναι bias, w_i είναι οι εκπαιδευμένοι συντελεστές (βάρη) και a_i είναι οι είσοδοι από το αντίστοιχο κανάλι.Διαισθητικά, όταν οι είσοδοι πολλαπλασιάζονται με μεγάλους αριθμούς, τείνουν να παράγουν σημαντικά πιο επιδραστικά προϊόντα ($a_i \cdot w_i$) στο τελικό αποτέλεσμα σε σύγκριση με τις εισόδους που πολλαπλασιάζονται με μικρές τιμές. Ωστόσο, αξίζει να σημειωθεί ότι η σημασία του προϊόντος $a_i \cdot w_i$ εξαρτάται επίσης από την τιμή του a_i. Έτσι, ορίζουμε την *σημαντικότητα* κάθε προϊόντος ως εξής:

$$S_i = |\frac{\mathrm{E}[a_i] \cdot w_i}{\sum_{\forall i} \left(\mathrm{E}[a_i] \cdot w_i\right)}|. \tag{1.2}$$

Με άλλα λόγια, η (1.2) υπολογίζει το μακροπρόθεσμο αναμενόμενο αποτέλεσμα κάθε προϊόντος $a_i \cdot w_i$ επί του συνολικού αθροίσματος Sum_c του αντίστοιχου καναλιού. Εάν το άθροισμα ισούται με μηδέν, θεωρούμε ότι το αντίστοιχο S_i είναι μεγάλο και, συνεπώς, το προϊόν διατηρείται. Για κάθε κανάλι και Sum_c, ο υπολογισμός του $S_i, \forall i$, είναι απλός και περιλαμβάνει τη λήψη της κατανομής εισόδου από ένα τμήμα του συνόλου δεδομένων. Εκμεταλλευόμενοι αυτή την πληροφορία υψηλού επιπέδου, ελαχιστοποιούμε τους συνολικούς υπολογισμούς που απαιτούνται για κάθε πράξη άθροισης Sum_c σε χρόνο μεταγλώττισης, και έτσι η άθροιση προσεγγίζεται ανάλογα. Συγκεκριμένα, για κάθε γινόμενο $a_i \cdot w_i$, εάν το S_i είναι μικρότερο ή ίσο από ένα δεδομένο κατώφλι τ, ενσωματώνεται στον παραγόμενο κώδικα, ενώ άλλα παραλείπονται. Έτσι, το προσεγγιστικό άθροισμα ανά κανάλι αντιπροσωπεύεται τώρα από:

$$Sum'_c = b + \sum_{\forall i} (a_i \cdot w_i) - \sum_{\forall i : S_i < \tau} (a_i \cdot w_i) \tag{1.3}$$

Εφαρμογή Μεθόδου

Για την αξιοποίηση των πλεονεκτημάτων που αναλύονται σε αυτό το κεφάλαιο, απαιτείται συνδυασμός τεχνικών. Ενώ η απόκτηση ενός βέλτιστου μοντέλου TensorFlow Lite με βάση τους περιορισμούς σχεδιασμού που ορίζει ο χρήστης περιλαμβάνει τον μηδενισμό των βαρών, αυτό το βήμα από μόνο του δεν θα αποφέρει σημαντικές βελτιώσεις στην καθυστέρηση. Για την περαιτέρω βελτίωση των επιδόσεων, χρησιμοποιείται η διαδικασία αποσυσκευασίας με βάση τα στρώματα. Το βέλτιστο μοντέλο αποσυμπιέζεται και αποκλείονται όλα τα βάρη με μηδενικές τιμές. Αυτό όχι μόνο μειώνει τον αριθμό των πράξεων πολλαπλασιασμού-συσσώρευσης (MAC) στον πυρήνα πολλαπλασιασμού πινάκων, αλλά μετριάζει επίσης ορισμένα από τα μειονεκτήματα που σχετίζονται με τη διαδικασία αποσυμπίεσης. Λιγότερες γραμμές κώδικα οδηγούν σε μειωμένη χρήση flash και ενδεχομένως βελτιώνουν την απόδοση της κρυφής μνήμης εντολών.

Στο τέλος της διαδικασίας αποσυμπίεσης, οι παραγόμενοι πυρήνες πολλαπλασιασμού πινάκων για κάθε επίπεδο μπορούν να υλοποιηθούν με διάφορους τρόπους. Εάν ο επεξεργαστής υποστηρίζει την επέκταση DSP, οι πυρήνες θα παραχθούν με την αποσυσκευασία των βαρών σε ομάδες των 2 ανά εντολή SMLAD. Διαφορετικά, θα χρησιμοποιηθεί η υλοποίηση σε Pure C και ο τελικός χρήστης θα έχει τη δυνατότητα να ενεργοποιήσει τη βελτιστοποίηση np2 αν το νευρωνικό δίκτυο έχει εκπαιδευτεί κατάλληλα. Αυτή η βελτιστοποίηση αντικαθιστά τις δαπανηρές εντολές MAC με απλές προσθήκες με δυνατότητα μετατόπισης, βελτιώνοντας περαιτέρω την υλοποίηση για βελτιωμένη απόδοση όταν πρόκειται για low-end MCUs.

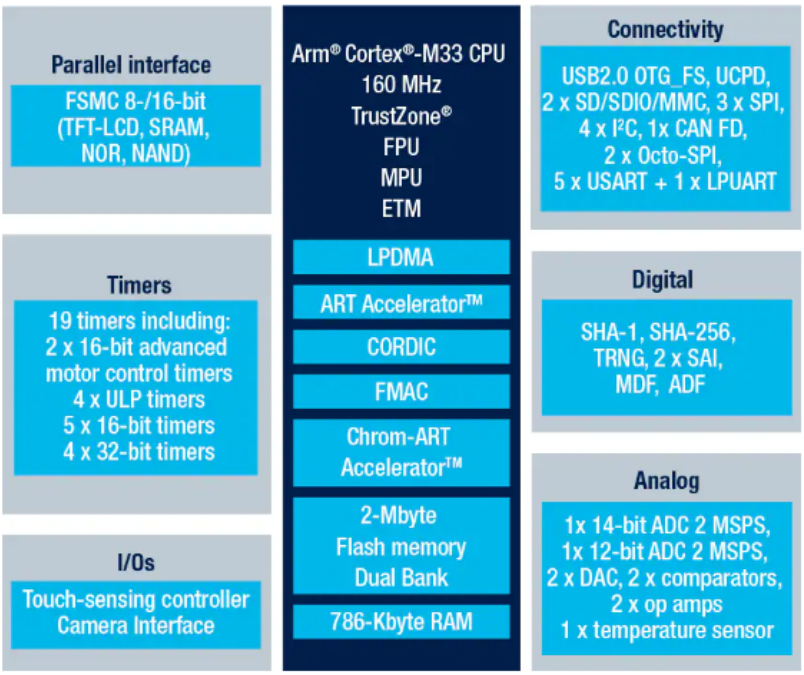
Υλικό Ανάπτυξης

Στο πλαίσιο των πειραμάτων που αναλύονται στις επόμενες ενότητες, αξιολογήσαμε διάφορες αναπτυξιακές πλακέτες που διαθέτουν πυρήνες ARM Cortex M. Συγκεκριμένα, επικεντρωθήκαμε στις υλοποιήσεις STM32 των Cortex M Cores λόγω της εκτεταμένης προσβασιμότητας στο σύνολο εργαλείων τους, η οποία απλοποιεί σημαντικά όλες τις πτυχές της διαδικασίας ανάπτυξης, από την αποσφαλμάτωση έως την τελική ανάπτυξη. Οι πλακέτες Nucleo, οι οποίες επιλέχθηκαν για την παρούσα αξιολόγηση, προσφέρουν τα πλεονεκτήματα ότι είναι οικονομικά αποδοτικές, καλά τεκμηριωμένες και πλήρως συμβατές με τα προαναφερθέντα εργαλεία ανάπτυξης. Επιπλέον, σε ευθυγράμμιση με παρόμοιες εργασίες [11, 12], οι πλακέτες Nucleo έχουν καθιερωθεί ως το βιομηχανικό πρότυπο για την αξιολόγηση νευρωνικών δικτύων σε συσκευές κατηγορίας μικροελεγκτών.

Οι πλακέτες Nucleo προσφέρουν μια σειρά επιλογών προσαρμοσμένων για ποικίλες εφαρμογές. Για τα πειράματά μας, επιλέξαμε την πλακέτα NUCLEO-U575ZI-Q, η οποία διαθέτει μια μονάδα μικροελεγκτή (MCU) STM32U575ZIT6Q που βασίζεται στον πυρήνα ARM Cortex-M33. Η επιλογή αυτή επιτυγχάνει μια ισορροπία, τοποθετούμενη μεταξύ των παραλλαγών Cortex-M7 υψηλής απόδοσης και των πιο βασικών MCU που χρησιμοποιούν τον πυρήνα Cortex-M3.

΄Οπως απεικονίζεται στο Σχήμα 1.6, η επιλεγμένη M"Y λειτουργεί σε συχνότητα ρολογιού 160 MHz και ενσωματώνει τον επιταχυντή STM32 ART Accelerator, παρέχοντας λειτουργικότητα κρυφής μνήμης με τη μορφή 8KB ICache και 4KB DCache, που βρίσκονται εκτός του πυρήνα ARM. Επιπλέον, προσφέρει 2 megabytes μνήμης Flash και 768kilobytes μνήμης RAM.

Αξίζει να σημειωθεί ότι η πλακέτα NUCLEO-U575ZI-Q είναι εξοπλισμένη με έναν ενσωματωμένο αποσφαλματωτή και προγραμματιστή κυκλώματος STLINK, ο οποίος απλοποιεί σημαντικά τη διαδικασία αποσφαλμάτωσης και προγραμματισμού. Η λειτουργία αυτή διευκολύνεται απρόσκοπτα μέσω μιας διεπαφής USB, εξαλείφοντας την ανάγκη για εξωτερικό υλικό.



Σχήμα 1.6: Κυκλωματικό διάγραμμα της MCU STM32U575ZIT6Q
Source: ST

CNN	Acc	# MAC Ops	Latency (ms)	Flash Usage (%)	RAM (KB)
SqueezeNet	75.0	14.2 M	551.5	16	298.64
AlexNet	71.9	11.5 M	179.9	13	212.16
LeNet	72.0	4.6 M	82.8	12	183.5

Table 1.1: Αξιολόγηση των μοντέλων αναφοράς CIFAR-10 SqueezeNet, AlexNet, LeNet σε ένα STM32-Nucleo 2000KB ROM, 768KB RAM

Βασικά Χαρακτηριστικά Αξιολογούμενων Μοντέλων

Ο πίνακας 1.1 αναφέρει τα χαρακτηριστικά (π.χ. αρχική ακρίβεια, καθυστέρηση) για τα βασικά μας μοντέλα που αναπτύσσονται σε ένα STM32-Nucleo-U575ZI-Q με συχνότητα ρολογιού στα 160MHz. Τα αντίστοιχα μοντέλα εκπαιδεύτηκαν σε CIFAR- 10 σύνολο δεδομένων, χρησιμοποιώντας TensorFlow με επακόλουθη εφαρμογή κβαντισμού 8-bit μετά την εκπαίδευση. Οι είσοδοι έχουν διαστάσεις 32x32 ανάλυση και κανονικοποιούνται στο [0,1] σε αναπαράσταση 32-bit. και η εκπαίδευση/δοκιμή χρησιμοποιεί τυχαίο διαχωρισμό 70%/30%. Όπως φαίνεται στον πίνακα 1.1, η καθυστέρηση για ένα μικρό μοντέλο με λιγότερο από 5M παραμέτρους είναι πάνω από 80ms, ενώ είναι αξιοσημείωτο ότι το 86%, κατά μέσο όρο, της διαθέσιμης μνήμης flash παραμένει ανεκμετάλλευτο.

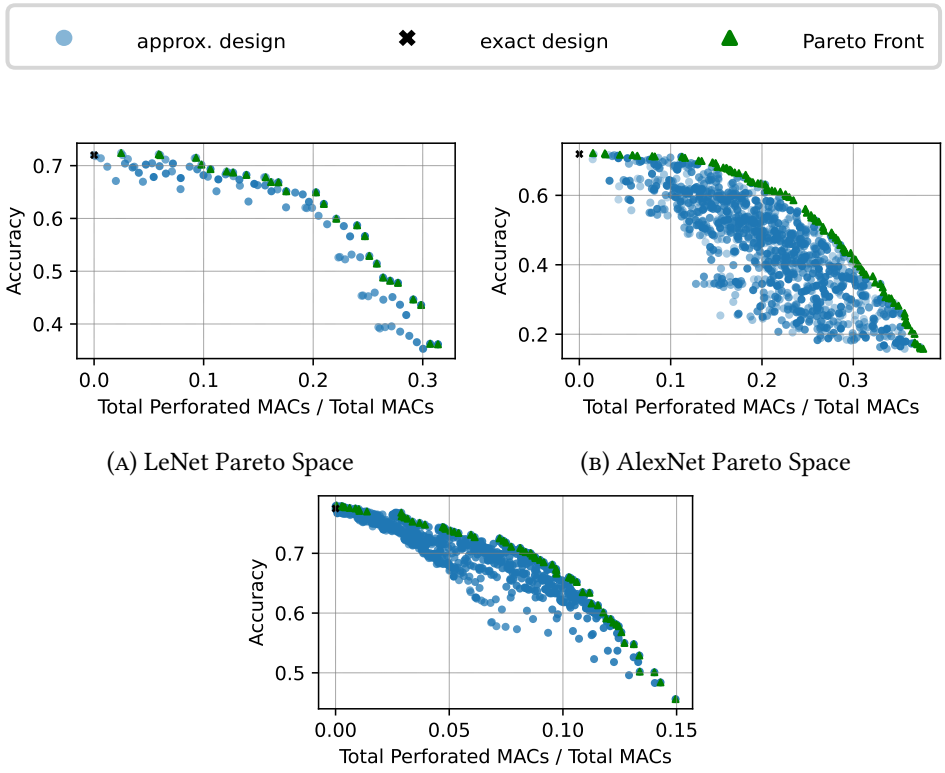
Χώρος Pareto ακρίβειας-καθυστέρησης

Στο Σχήμα 1.7 παρουσιάζεται ο χώρος Pareto μεταξύ ακρίβειας και καθυστέρησης για το τρία εξεταζόμενα CNN. Σε αυτό το σημείο, η χρονική καθυστέρηση αφορά μόνο στρώματα συνέλιξης και αναφέρεται ως κανονικοποιημένη τιμή σε σχέση με το. την καθυστέρηση της βελτιστοποίησής μας ""unpacked only"". Η τελευταία, αν και επιτρέπει στο πλαίσιό μας να εκτελεί μια επιλεκτική χαμηλού επιπέδου παράλειψη υπολογισμών, σε ορισμένες περιπτώσεις εισάγει μια ελάχιστη επιβάρυνση λόγω υπερφόρτωσης της μνήμης κρυφής μνήμης. Στο Σχήμα 1.7 το μαύρο "x" είναι ο ακριβής βασικός μας σχεδιασμός. Οι μπλε κουκκίδες στο γράφημα αντιστοιχούν σε προσεγγιστικές διαμορφώσεις. Το πράσινα τρίγωνα, από την άλλη πλευρά, σχηματίζουν τη γραμμή του μετώπου Pareto. Αυτές οι διαμορφώσεις αντιπροσωπεύουν ειδικότερα το ποσοστό λειτουργιών που παραλείπονται και τους δείκτες αυτών των λειτουργιών στον τελικό παραγόμενο κώδικα. Για τη δημιουργία αυτού του σχήματος, η καθυστέρηση υπολογίστηκε με βάση τους μετρητές κύκλων μας, καθώς η ανάπτυξη πάνω από χίλια σχέδια σε MCUs είναι ανέφικτη, ακόμη και όταν είναι αυτοματοποιημένη. Πρόκειται για μια αρχική ανάλυση που βοηθά στην εξαγωγή προσεγγιστικά σχέδια με βάση την καθορισμένη απώλεια ακρίβειας όριο και στη συνέχεια την ανάπτυξή τους σε MCUs. Σημείωση ότι ο αριθμός των εξεταζόμενων διαμορφώσεων/σχεδίων είναι μοντέλο εξαρτάται από το μοντέλο. Η σημαντικότητα Si κυμαίνεται από [0-0,009] για όλα τα CNN, αλλά τα εξεταζόμενα στρώματα στα οποία Si διερευνάται, εξαρτώνται επίσης από το μοντέλο. Συνολικά αξιολογήσαμε περισσότερα από 100 προσεγγιστικά σχέδια για το LeNet και περισσότερα από 1000 σχέδια για το AlexNet και το SqueezeNet.

Συνολικά, παρατηρείται ότι ο προσεγγιστικός υπολογισμός μπορεί να να χρησιμοποιηθεί αποτελεσματικά για την επιτάχυνση της εξαγωγής συμπερασμάτων CNN μοντέλων, καθώς όλα τα προσεγγιστικά σχέδια διαθέτουν χαμηλότερη καθυστέρηση από τον ακριβή (δηλαδή, μαύρο "x"). Όπως φαίνεται, την ¨μόνο skipping¨ προσέγγιση επιτυγχάνει κατά μέσο όρο 11% λανθάνουσα κατάσταση μείωση για σχεδόν ίδια ακρίβεια (<1%) με το ακριβές βασική γραμμή. Επιπλέον, για τα δύο πρώτα μοντέλα, παρέχει κατά μέσο όρο περισσότερο από 15%, 20% αύξηση της ταχύτητας για λιγότερο από 3% και 7% απώλεια ακρίβειας, αντίστοιχα. Στην περίπτωση του SqueezeNet, τα αποτελέσματα φαίνονται λιγότερο ελπιδοφόρα, ενδεχομένως λόγω της ιδιαίτερα συμπιεσμένης αρχιτεκτονικής του.

Deployment Results

Όπως απεικονίζεται στον Πίνακα 1.2 για cached υλοποίηση, η cooperative προσέγγιση, η οποία περιλαμβάνει τόσο την αποσυμπίεση κώδικα όσο και την προσέγγιση με επίγνωση της σημαντικότητας, επιτυγχάνει μέση επιτάχυνση 11%, ενώ έχει απώλεια ακρίβειας μόνο 3% σε σύγκριση με την ακριβή βασική προσέγγιση [2]. Επιπλέον, τα κέρδη αυτά αυξάνονται σε 21% όταν δεχόμαστε απώλεια ακρίβειας 9%.



(A) LeNet Pareto Space

(B) AlexNet Pareto Space

(C) SqueezeNet Pareto Space

FIGURE 1.7: Pareto Spaces for the Examined NN Architectures

Επίσης, για την cacheless υλοποίηση φαίνεται στον Πίνακα 1.3, η cooperative μας προσέγγιση, η οποία περιλαμβάνει τόσο την αποσυμπίεση κώδικα όσο και την προσέγγιση με επίγνωση της σημαντικότητας, έχει επιτάχυνση κατά 32% με απώλεια ακρίβειας μόνο 3% σε σύγκριση με το ακριβές baseline [2]. Επιπλέον, τα κέρδη αυτά αυξάνονται σε 34% όταν δεχόμαστε απώλεια ακρίβειας 6%.

Σύγκριση με συναφή framework

Τέλος, πραγματοποιούμε μια ποιοτική αξιολόγηση, συγκρίνοντας το πλαίσιο προσέγγισης που διαθέτουμε με άλλες σύγχρονες μεθοδολογίες. Σε αυτή την περίπτωση είναι σημαντικό να σημειωθεί ότι, δεδομένου ότι αυτά τα πλαίσια έχουν υλοποιηθεί με τη χρήση πλακετών που φέρουν κάποια μορφή κρυφής μνήμης, οι ακόλουθες αποδόσεις έχουν μετρηθεί σε σχέση με τον πίνακα Cached Results. 7.5. Κατά τη σύγκριση με το TinyEngine [12], η άμεση αξιολόγηση καθίσταται δύσκολη λόγω της περιορισμένης υποστήριξής του για τα απαραίτητα στρώματα στα CNN που μελετήσαμε. Επιπλέον, η μεθοδολογία συν-σχεδιασμού της είναι προσαρμοσμένη κυρίως για συγκεκριμένα μοντέλα. Ωστόσο, αξίζει να σημειωθεί ότι στο [12], αναφέρουν μείωση της καθυστέρησης κατά 10.5% σε σύγκριση με το [9] όταν χρησιμοποιούν το μοντέλο τους mcunet-vww1, το οποίο χρησιμοποιεί βελτιστοποιήσεις τόσο πριν όσο και μετά την εκπαίδευση. Αντίθετα, για ένα προ-εκπαιδευμένο μοντέλο με

	CMSIS-NN		X-CUBE-AI		Proposed (ours)					
Network	**LeNet**	**AlexNet**	**LeNet**	**AlexNet**	**LeNet 5%**	**LeNet 10%**	**LeNet 15%**	**AlexNet 5%**	**AlexNet 10%**	**AlexNet 15%**
Top-1 Accuracy (%)	**72.0**	**71.9**	72.0	71.9	68.7	65.0	62.8	68.4	64.8	62.6
Latency (ms)	82.8	179.9	**63.5**	150.7	76.4	72.0	70.1	153.9	**148.5**	**133.2**
Flash (KB)	239	267	154	178	659	591	585	954	902	999
#MAC Ops.	4.6M	11.5M	4.6M	11.5M	**4.2M**	**4.0M**	**3.9M**	**9.6M**	**9.5M**	**8.2M**
Energy (mJ)	2.73	5.94	**2.10**	4.97	2.52	2.38	2.31	5.08	**4.90**	**4.39**

Table 1.2: Comparison with state-of-the-art CMSIS [2] and X-CUBE-AI [9] with the Cache enabled for two CNNs deployed on an stm32u575zi-q board fitting 2MB Flash and 768KB RAM. Three accuracy loss thresholds have been considered.

	CMSIS-NN			Proposed (ours)								
Network	**LeNet**	**AlexNet**	**SqueezeNet**	**LeNet 1%**	**LeNet 5%**	**LeNet 10%**	**AlexNet 1%**	**AlexNet 5%**	**AlexNet 10%**	**SqueezeNet 1%**	**SqueezeNet 5%**	**SqueezeNet 10%**
Top-1 Acc (%)	**72.0**	**71.9**	**75**	71.5	68.7	65.0	71.2	69.3	64.8	75	72	71
Latency (ms)	172.9	407.7	972.9	114.5	106.2	100.9	249.8	227.5	218.8	855.3	852.1	849.8
Flash (KB)	232	261	313	1015	962	832	1690	1420	1370	1440	1380	1360
#MAC Ops.	4.6M	11.5M	14.2 M	**4.5 M**	**4.2M**	**4.0 M**	**11.3M**	**10.0 M**	**9.5M**	**14.0 M**	**13.9 M**	**13.8 M**
Energy (mJ)	6.74	15.90	37.94	**4.46**	**4.14**	**3.93**	**9.74**	**8.87**	**8.53**	**33.36**	**33.23**	**33.14**

Table 1.3: Comparison with state-of-the-art CMSIS [2] with the Cache disabled for three CNNs deployed on an stm32u575zi-q board fitting 2MB Flash and 768KB RAM. Three accuracy loss thresholds have been considered.

παρόμοιες MAC, το πλαίσιο μας επιτυγχάνει μείωση της καθυστέρησης κατά 12% με κόστος 9% απώλεια ακρίβειας Top-1. Επιπλέον, σε σύγκριση με το CMix-NN [11] χρησιμοποιώντας ένα μοντέλο με 13.8M MAC operations, το πλαίσιό μας επιτυγχάνει καθυστέρηση 164ms σε μια MCU 160MHz. Αυτό σημαίνει ότι, σε σύγκριση με το CMix-NN [11], το πλαίσιό μας επιτυγχάνει αξιοσημείωτη μείωση της καθυστέρησης κατά 50%, με αμελητέα υποβάθμιση της ακρίβειας μικρότερη από 1%. Τέλος, το uTVM [13], ένα ολοκληρωμένο πλαίσιο μεταγλωττιστή ML προσαρμοσμένο για bare metal MCUs, αναφέρει 13% επιβάρυνση καθυστέρησης σε σύγκριση με το CMSIS όταν χρησιμοποιεί παρόμοια αρχιτεκτονική μοντέλου LeNet όπως εξετάστηκε στην εργασία μας. Για το ίδιο μοντέλο, η προσέγγισή μας υπερτερεί έναντι του uTVM, επιτυγχάνοντας επιπρόσθετη επιτάχυνση 26% με απώλεια ακρίβειας μικρότερη από 7%. Αυτή η προσαρμογή στην καθυστέρηση εξαγωγής συμπερασμάτων έχει τη δυνατότητα να επιτρέψει πιο πρακτικές και ρεαλιστικές εφαρμογές, ανοίγοντας το δρόμο για την υλοποίηση βαθύτερων δικτύων σε μικροσκοπικές MCU.

Συμπεράσματα και μελλοντικές εργασίες

Παρά την ευρεία υιοθέτησή τους στη μικροσκοπική μηχανή μάθησης, οι ενεργειακά αποδοτικές μονάδες μικροελεγκτών αντιμετωπίζουν αξιοσημείωτους περιορισμούς. Οι περιορισμοί αυτοί αφορούν κυρίως την περιορισμένη υπολογιστική τους χωρητικότητα και τους σφιχτούς ενεργειακούς προϋπολογισμούς, και κατά συνέπεια, η ανάπτυξη σύνθετων μοντέλων βαθιάς μάθησης σε τέτοιες πλατφόρμες γίνεται ένα δύσκολο εγχείρημα. Σε

αυτή την εργασία, για την αντιμετώπιση αυτής της πρόκλησης, εισάγουμε ένα cooperative framework προσέγγισης που συνδυάζει τις αρχές του προσεγγιστικού υπολογισμού με βελτιστοποιήσεις πυρήνων λογισμικού.

Χρησιμοποιώντας μια συστηματική μέθοδο που βασίζεται στην παράλειψη υπολογισμού πυρήνων, το πλαίσιό μας εξαλείφει αποτελεσματικά λειτουργίες που θεωρούνται ασήμαντες για τη διαδικασία εξαγωγής συμπερασμάτων του μοντέλου. Αυτό έχει ως αποτέλεσμα καλύτερες ταχύτητες εξαγωγής συμπερασμάτων, αν και με μικρή μείωση της ακρίβειας. Επιπλέον, η προσέγγισή μας επιτρέπει την αποτελεσματική εκτέλεση μοντέλων βαθιών νευρωνικών δικτύων σε σχέδια μονάδων μικροελεγκτών που δεν διαθέτουν κρυφή μνήμη. Το επιτυγχάνουμε αυτό επιτρέποντας την επιλεκτική αποσυμπίεση κώδικα με βάση το επίπεδο, η οποία αξιοποιεί πλήρως τη μνήμη flash μιας MCU και επιτρέπει βελτιστοποιήσεις ειδικά για τον πυρήνα. Σε ορισμένες περιπτώσεις, αυτή η προσέγγιση μπορεί να προσεγγίσει το επίπεδο επιδόσεων των MCU με μνήμη cache. Ο συνδυασμός της αξιοποίησης της μνήμης flash και του συμβιβασμού μεταξύ ακρίβειας και καθυστέρησης μπορεί να ανοίξει νέες δυνατότητες για εφαρμογές τεχνητής νοημοσύνης και να καταστήσει εφικτή την εκτέλεση πιο σύνθετων βαθιών νευρωνικών δικτύων σε μικρές MCU.

Ως μελλοντικό έργο, θα μπορούσαμε να εξετάσουμε το ενδεχόμενο επέκτασης της υποστήριξής μας για πρόσθετους τύπους συνελικτικών layer, όπως η συνέλιξη κατά βάθος depthwise και η διαχωρίσιμη συνέλιξη κατά βάθος depthwise seperable. Αυτή η επέκταση θα επέτρεπε τη συμβατότητα με ένα ευρύτερο φάσμα μοντέλων βαθιάς μάθησης, συμπεριλαμβανομένων αρχιτεκτονικών όπως το MobileNet. Θα διευκόλυνε επίσης τις άμεσες συγκρίσεις με γνωστά πλαίσια τελευταίας τεχνολογίας όπως το TinyEngine [12]. Επιπλέον, υπάρχει η δυνατότητα να διερευνηθεί και να βελτιστοποιηθεί η διαδικασία αποσυμπίεσης ώστε να γίνει πιο φιλική προς την κρυφή μνήμη. Αυτή η βελτιστοποίηση θα μπορούσε να βοηθήσει στην εξάλειψη κάθε πιθανής επιβάρυνσης που εισάγεται κατά την εκτέλεση, ενισχύοντας τη συνολική αποδοτικότητα του πλαισίου μας.

Chapter 2

Introduction

2.1 Introduction to TinyML

In recent years, there has been a significant growth in the proliferation of IoT devices, particularly those built upon low-cost microcontroller units (MCUs). This growth has been characterized by the widespread adoption of energy-efficient microcontrollers, expanding the horizons of the Tiny Machine Learning (TinyML) domain [14, 15]. This emerging field has unlocked a novel avenue for deploying deep learning models on tiny devices, allowing for real-time and near sensor data processing, which extends the realm of AI applications [16, 17].

While commercial MCUs hold the promise of meeting the demands of energy efficiency, their constrained resource budget, including limited memory (SRAM, Flash) and high latency, acts prohibitively for the deployment of deep learning on small-scale hardware. The latter include implementations of ML classification algorithms that are required by a large number of AI applications in the aforementioned domains [16]. Hence, there is a pressing need for new optimizations to bridge the gap between required and available resources.

In this effort, ARM developed a software library CMSIS-NN [2], providing efficient and optimized implementations of neural network operations for MCUs running on Arm Cortex-M CPUs. While this library results in a marginal increase in flash usage, it delivers nearly an 11x improvement in latency, on average, compared to TensorFlow Lite Micro [8] for several ImageNet models [12] deployed on a commercial STM32H743 board. TinyEngine [12], a system-model co-design framework, employs jointly a neural architecture search (NAS) and memory-based optimized inference library. This combination results in notable improvements with average reductions of $2.1\times$ in latency and $2.4\times$ in SRAM usage compared to CMSIS-NN [2]. However, the state-of-the-art frameworks mentioned earlier mainly prioritize reducing the size of the model to fit within predefined memory constraints. They do not primarily address the goal of reducing inference latency for an already well-fitted, albeit large, model. As a case in point, TinyEngine requires nearly 1.3s to execute an mcunet-in4 ImageNet model on a 160MHz MCU. This highlights the existing latency challenges, particularly in real-time applications, where fast execution is of utmost importance.

In this work, we investigate the feasibility of utilizing the complete resources of MCUs, which are typically constrained by predefined board limits, to substantially enhance the performance of deep neural networks on microcontrollers. To this end, we bring together principles from Approximate Computing (AC) [18] with optimized software kernels, designed for deploying neural networks on MCUs. Our approach leads to the development of an automated cooperative framework that generates specialized approximate code tailored to a specific Convolutional Neural Network (CNN). Firstly, our framework exploits the accessible flash memory to unpack the kernel code within convolution layers, effectively eliminating any associated instruction overheads. Subsequently, by leveraging the unpacked operations and the fact that each computation contributes uniquely to the final output, we employ an offline significance-aware computation skipping approach. Finally, through a full search design space exploration (DSE), our framework extracts Pareto-optimal solutions, each providing distinct accuracy-latency trade-offs, tailored to individual requirements. Overall, compared to the exact state-of-the-art CMSIS-NN, we achieve 9% latency reduction with almost identical (<1%) Top-1 accuracy, over 2 CNN models trained on the popular CIFAR-10 dataset and 37% latency reduction for less than 1% loss of Top-1 accuracy over 3 CNN models trained on the CIFAR-10 dataset. On the other hand, for lower accuracy requirements our approach outpeforms even commercial frameworks.

Our novel contributions within this work are as follows:

1. This is the first work that evaluates the impact of approximate computing on the optimized inference library of CMSIS-NN specifically when targeting MCUs.

2. We propose an automated cooperative approximation framework for accelerating CNNs inference on MCUs

3. Using our framework, we demonstrate that, in many cases approximate computing is able to realize large, but fast networks on tiny devices.

2.2 Related Work

Research in the deployment of deep inference networks in microcontrollers is increasingly growing with numerous frameworks targeting the widely used ARM Cortex-M MCUs and trying to bridge the gap posed by computational and memory limitations.

DNN inference on Microcontrollers. STMicroelectronics has introduced X-CUBE-AI [9], an expansion pack designed to streamline the deployment of high-level deep learning models, including 32-bit floating point and 8-bit quantized models, onto low-end STM32 microcontrollers. TinyEngine [12] which builds upon the CMSIS-NN [2] and its software stack tailored for inference on ARM Cortex-M devices, integrates numerous optimization strategies. These optimizations are aimed at enhancing inference speed and minimizing memory footprint, resulting in superior performance compared to aforementioned frameworks across

TABLE 2.1: Qualitative comparison of related frameworks.

Framework	Orthog.[1]	Customized[2]	AC	Resource Utilization[3]	Acc - Latency Trade-off
CMSIS-NN	✓	✗	✗	✗	✗
X-CUBE-AI	✗	✓	✗	≈	≈
TinyEngine	✓	✓	✗	✗	✓
CMix-NN	✓	✗	✓	✓	✓
MicroTVM	✗	✓	✗	✗	✗
Ours	✓	✓	✓	✓	✓

[1]Orthogonal with other frameworks/techniques. [2]Customized code for specific models. [3]Efficient utilization of resources

several benchmarks. Notably, the operations used within this framework are model-specific and thus the generated code is fully specialized to accommodate the models obtained through their NAS approach. On the other hand, CMiX-NN [11], a mixed low-precision inference library for low bit-width Quantized Networks [19], tries to minimize memory requirements, by enabling aggressive and arbitrary quantization to effectively deploy CNNs on tiny MCUs. Finally, MicroTVM [13] is an open-source deep learning compiler stack (TVM) that provides a set of tools and libraries to optimize and deploy similarly DNNs across MCUs.

Neural Network Model Optimization. An active research field explores ways to create efficient network models capable of running on small, resource-limited IoT devices. Some approaches focus on approximating the model's architecture by prunning trainable parameters of the network [20, 21, 22] and in some cases they also totally remove less crucial channels from convolutional layers [23], or replace the entire layers with smaller sub-layers [24]. On the other hand, others employ NAS [25, 26] as a method to lower complexity and effectively design neural networks specifically for edge devices.

Our work distinguishes from existing state-of-the-art works and can be classified alongside the frameworks listed in Table 2.1, since these frameworks share a common focus on optimizing the inferenece of trained DNNs on microcontrollers and do not target solely on the modification of the model's architecture or pre-training optimization/approximation approaches. This Table provides a qualitative comparison of the related frameworks in the field of resource-constrained DNN inference, highlighting the key differences left unexplored in these works.

Chapter 3

Background

3.1 CNN models

Convolutional Neural Networks (CNNs) [1] represent a subcategory of Artificial Neural Networks (ANNs) that are tailored for applications that process visual data and expect an image as their input. A typical CNN structure Fig. 3.1 is comprised of the input image which serves as the input layer as well as a combination of three additional layers: convolutional,pooling and dense.

Convolutional Layer is a layer that utilizes a convolutional filter (i.e. a matrix with a smaller dimensions but the same rank as the input matrix) in order to produce a feature map, which is a matrix containing the resulting convolutional operations. Additional convolution parameters such as padding exist in order to avoid information loss on the border of the kernel as well stride to control the convolution density.

Pooling Layer is a layer whose purpose is the reduction of the feature maps produced by previous convolutional layers by selecting either the maximum or average value within the defined pooling region. This spatial reduction serves to reduce the trainable parameter count by eliminating redundant attributes as well as alleviating potential over-fitting due to the large number of features.

A comprehensive example of the convolution operation along with average and max pooling can be seen in Fig. 3.2

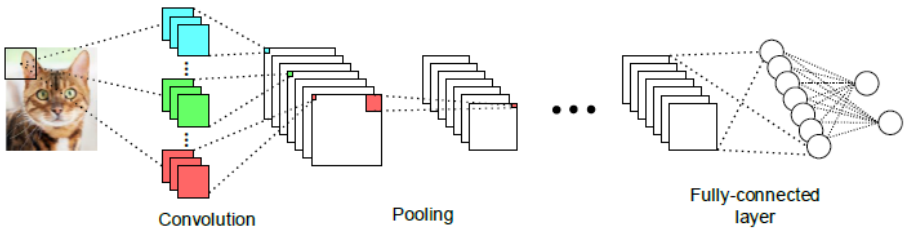


FIGURE 3.1: Schematic diagram of a basic convolutional neural network (CNN) architecture
Source: [2]

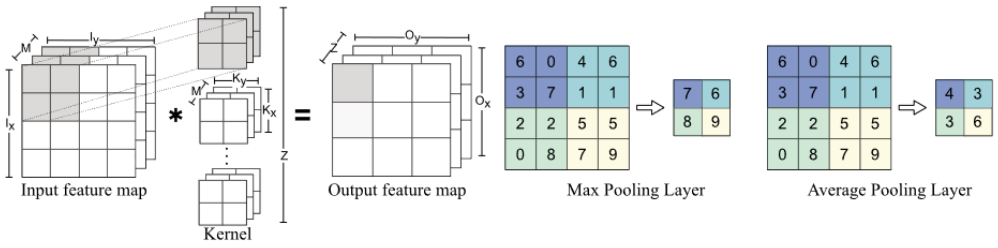


FIGURE 3.2: (a) Convolution and (b) Max pooling and average pooling examples.
Source: [18]

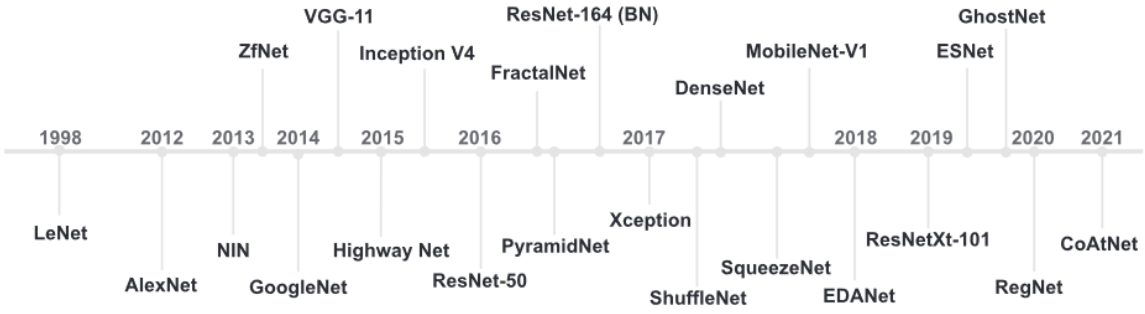


FIGURE 3.3: Timeline of notable DNNs
Source: [18]

Dense Layer or fully-connected layer performs the consolidation of high-level features extracted from previous layer into a final classification using an activation function. This type of layer has each neuron of the previous and current layers connected.

As highlighted in [18], a series of Convolutional Neural Networks have emerged as effective solutions for various computer vision tasks, beginning with LeNet in the 1980s. Since then, these CNNs have played a pivotal role in achieving significant breakthroughs in classification tasks. The complexity of these Deep Neural Networks has continually increased in response to the growing demand for higher classification accuracy. Figure 3.3 provides a visual representation of prominent CNN architectures that have been introduced over the years.

3.2 Quantization

Quantization [3] entails executing neural network (NN) computations using fewer bits for precision. The most prevalent method performs quantization on the NN parameters post-training converting the original 32-bit floating point weights to 8-bit integer values. This technique combats one of the greatest challenges when deploying a NN on a tiny resource

constrained platform, model size. Additionally, the use of 8-bit integers allows efficient computation on MCU style devices since, in contrast to floating point operations, they are typically hardware accelerated.

All in all, this translates to a 4 time size reduction as well as notable latency improvement with inconsequential impact on prediction accuracy.

3.3 Approximate Computing

Approximate Computing (AC) [4] is a computational paradigm that intentionally trades off accuracy for improved efficiency and performance. The efficacy of this ektechnique hinges on the inherent error resilience of certain applications (e.g. computer vision). It should however be noted that AC should only be applied in non-critical data applications where potential errors are acceptable.

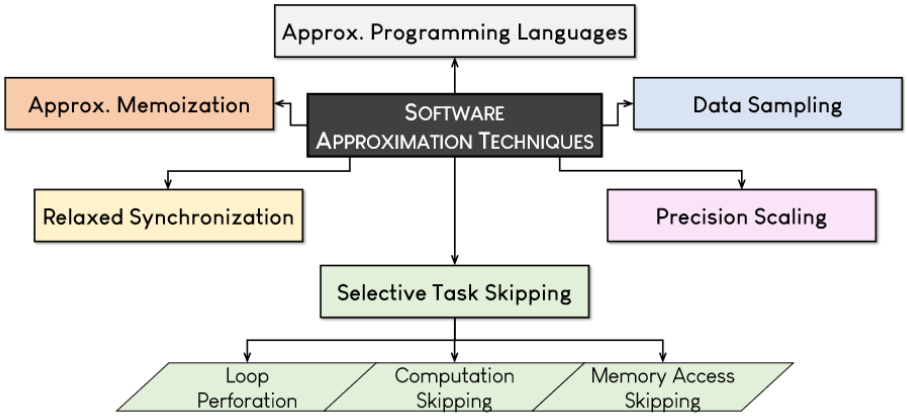
Several approaches exist for the implementation of AC either on the software Fig. 3.4a or the hardware side Fig. 3.4b, (e.g. approximate arithmetic) but for the purposes of this thesis we will focus mainly on software approximation techniques.

Overview of Selective Task Skipping

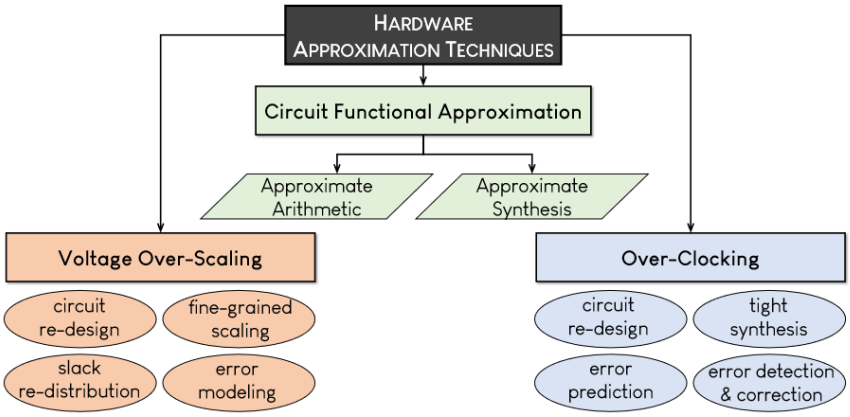
A popular approximation approach is to skip a certain percentage of the total computations in the form of selective task skipping as mentioned in [4]. This can be done by skipping loop iterations, skipping entire code blocks or even by omitting memory accesses.

Loop perforation. Loop perforation techniques involve selectively skipping certain iterations within loops in software programs. This approach aims to achieve performance and energy efficiency improvements by trading off computational accuracy or Quality of Service (QoS). Various techniques and tools have been developed to explore the design space of loop perforation, identify non-critical code segments, and optimize perforation configurations based on predefined QoS metrics. These techniques can vary in granularity, from skipping entire loop iterations to dynamically skipping specific loop instructions that do not significantly impact output accuracy. Loop perforation has found applications in diverse domains, including compilers, profiling tools, approximation frameworks for multi-core systems, and input-aware schemes for specific algorithms like graph algorithms.

Computation Skipping. Computation Skipping (CS) is a technique that selectively omits the execution of code blocks based on acceptable Quality of Service (QoS) loss, programmer-defined constraints, or runtime predictions regarding output accuracy. In contrast to loop perforation, CS goes beyond just skipping loop iterations; it also targets higher-level computations or tasks, including entire operations like convolutions. Many state-of-the-art approaches focus on application-specific CS strategies. For instance, some techniques offer strategies like convergence-based computation pruning and early termination of iterations. Others adapt CS for specific algorithms, omitting less impactful computations or tasks based



(A) Classification of software approximation techniques in six main classes: Selective Task Skipping, Approximate Memoization, Relaxed Synchronization, Precision Scaling, Data Sampling, and Approximate Programming Languages



(B) Classification of hardware approximation techniques in three main classes: Circuit Functional Approximation, Voltage Over-Scaling and Over-Clocking

FIGURE 3.4: Hardware&Software Approximation Techniques
Source: [4]

on various criteria. These techniques aim to strike a balance between performance optimization and maintaining acceptable QoS levels in software programs.

Memory Access Skipping

Memory Access Skipping (MAS) encompasses various software-level techniques designed to optimize execution time and reduce energy consumption by avoiding high-latency memory operations while simultaneously reducing computational workload. These techniques, which include approximate data locality, value prediction, cache miss skipping, memory optimization for GPUs, slicing-based data skipping, and neuron criticality-based skipping, offer versatile solutions to enhance software performance. By intelligently skipping memory accesses based on programmer-defined constraints, error predictions, or neural network criticality, these MAS techniques strike a balance between computational accuracy and resource efficiency, resulting in more efficient software execution.

Chapter 4

ARM Cortex-M

4.1 Overview

ARM Cortex-M is a group of 32-bit RISC ARM processor cores that are optimized for low cost and energy efficiency. As such they can be deployed in resource constrained computing scenarios either embedded into other types of chips (system controllers, sensor controllers etc.) or more commonly as microcontroller chips.

Higher end offerings exists from ARM, such as the Cortex-A lineup that target high performance systems designed to run rich operating systems on devices such as smartphones and tablets. Moreover, in some aspects similar to a high end MCU but targeting larger systems, the Cortex-R series is designed for use in high-performance real-time applications (e.g. hard disk controllers, networking equipment and printers).

Where the Cortex-M family differs from more sophisticated cores such as the Cortex-A is the lack of a memory management unit and the inability to run a regular operating system. In figure Fig. 4.1 an extensive overview of the differences between the different ARM solutions according to the intended application can be seen.

4.2 M-Profile Architectures

The ARM M-profile architecture [5] offers a standardized set of instructions and programming models tailored for secure microprocessors. These microprocessors are specifically designed to excel in deeply embedded systems by prioritizing low power consumption, minimal latency, and highly predictable operation.

Armv8.1-M

Armv8.1-M is an ARM architecture that introduces several significant enhancements, making it well-suited for signal processing, machine learning, and secure embedded systems. It features the M-Profile Vector Extension (MVE) called Helium, which provides efficient vector processing capabilities for accelerating signal processing and machine learning algorithms. This extension also brings support for additional data types in vectors, such as half-precision

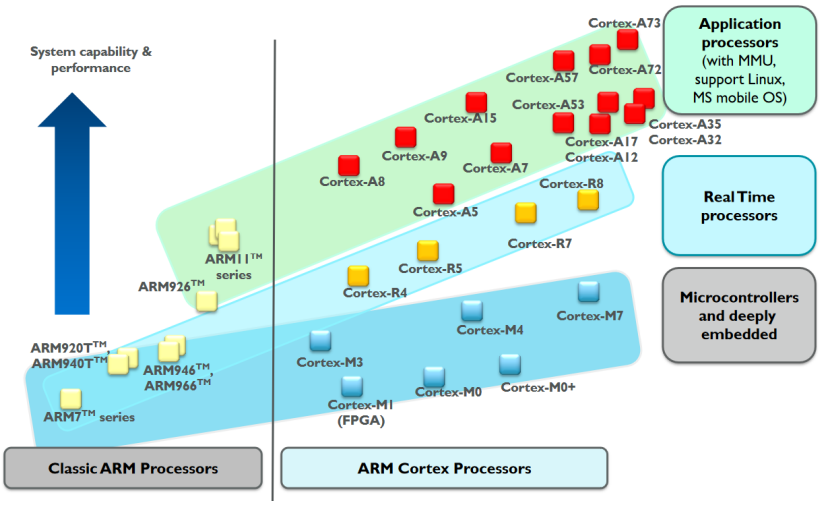


FIGURE 4.1: ARM processor family
Source: Arm Whitepaper [6]

floating point (FP16) and 8-bit integer (INT8), enhancing computational flexibility. Armv8.1-M offers low-overhead loops and advanced memory access capabilities like gather load and scatter store. It introduces various debug features, including a Performance Monitoring Unit and support for multiple security domains in debug, enhancing development and system analysis. The Pointer Authentication and Branch Target Identification (PACBTI) extension strengthens security and provides new tools for software developers. Additionally, Armv8.1-M incorporates extensions like Low Overhead Branch, Privileged eXecute Never (PXN) for Memory Protection Unit (MPU), and Reliability, Availability, and Serviceability (RAS), further expanding its capabilities in the embedded and secure systems domain.

Armv8-M

Armv8-M is an ARM architecture known for its versatility and adaptability in embedded systems. It introduces a new programmer's model with optional Memory Protection Unit (MPU) support, a streamlined subset of the T32 instruction set for efficient operation, and various architectural extensions to cater to diverse system design needs. Notably, it offers the flexibility of adding Arm Custom Instructions for customization without compromising access to the Arm software ecosystem. Key extensions include the Main Extension for backward compatibility with Armv7-M, the Security Extension (TrustZone) for enhanced security, the Floating-point Extension for precise floating-point arithmetic, the Debug Extension for debugging support, and the DSP Extension for digital signal processing tasks. These features make Armv8-M a powerful choice for building secure and flexible embedded systems.

Armv7-M

Armv7-M is an ARM architecture optimized for deeply embedded systems, emphasizing simplicity, low cycle count execution, minimal interrupt latency, and cacheless operation. It excels in scenarios where determinism and compact size are prioritized over absolute performance. This architecture includes support for a variant of the T32 instruction set to achieve these goals efficiently. Additionally, Armv7-M offers the DSP and Floating-point Extension, enhancing its capabilities for digital signal processing and floating-point arithmetic in embedded applications across various markets and use cases.

Armv6-M

Armv6-M is a lightweight version of the ARM architecture tailored for microcontroller applications. It incorporates features like the Debug Extension for debugging support, support for the T32 instruction set for enhanced performance, and upward compatibility with Armv7-M, allowing unmodified execution of software across versions. The addition of the Unprivileged/Privileged Extension enables varying privilege levels, facilitating more sophisticated operating systems, while the PMSA Extension, which requires the Unprivileged/Privileged Extension, enhances memory protection and security.

4.3 Cortex-M Cores

Based on the aforementioned M-Profile Architectures (mentioned in Section 4.2) the Cortex-M lineup [27] offers processors that according to the target application, offer a variety of features, providing low-latency operation for deeply embedded systems.

All Cortex-M processors universally support the Thumb instruction set, but the extent to which they incorporate instructions from the expanded Thumb-2 Technology varies. Different Cortex-M processors may offer different subsets of the instructions available in the Thumb ISA, providing flexibility in terms of instruction support while maintaining compatibility with the fundamental Thumb instruction set.

Cortex M0/M0+

The Cortex-M0 and Cortex-M0+ are energy-efficient microcontroller cores designed for low-power and cost-sensitive embedded applications. While both cores share similarities, such as their focus on ultra-low-power operation and use of the Armv6-M architecture, the Cortex-M0+ offers slightly improved performance over the Cortex-M0, making it an attractive choice when a bit more processing power is required without compromising energy efficiency. Both cores lack advanced features like digital signal processing (DSP) extensions and hardware division instructions, but they excel in scenarios where compact size, low power consumption,

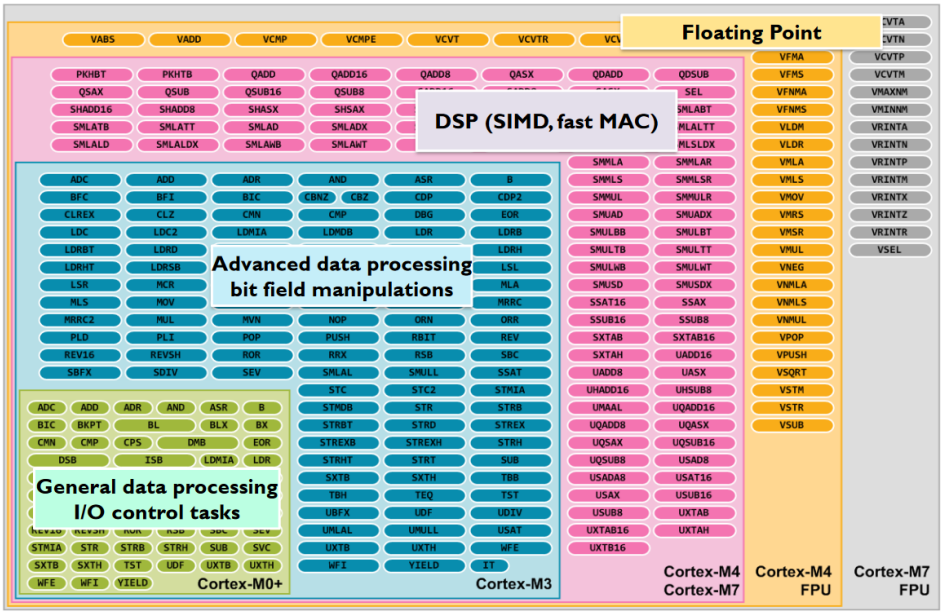


FIGURE 4.2: Instruction Set support in the Cortex-M processors
Source: Arm Whitepaper [6]

and cost-effectiveness are paramount, making them well-suited for applications such as IoT devices, basic industrial controllers, and various consumer electronics.

Cortex M4

The Cortex-M4 is renowned for its capability to deliver high-performance processing in energy-efficient microcontroller applications. It is based on the Armv7-M architecture and is equipped with digital signal processing (DSP) extensions, which enable efficient handling of DSP tasks, making it particularly well-suited for applications demanding advanced signal processing, while also supporting hardware division instructions. The Cortex-M4 finds widespread usage in applications such as automotive control systems, industrial automation, and multimedia processing, where a balance between high performance and power efficiency is critical.

Cortex M33

The Cortex-M33 is an Armv8-M Mainline based microcontroller core equipped with TrustZone technology for hardware-based security partitioning. With support for digital signal processing (DSP) instructions and hardware division, it offers robust performance, making it suitable for a wide range of embedded and IoT applications. Optional features like a co-processor interface as well as the capability to include custom instructions provide flexibility for customization, while its adaptability to handle various external interrupts and Memory Protection Unit (MPU) support makes it ideal for systems requiring both performance and security.

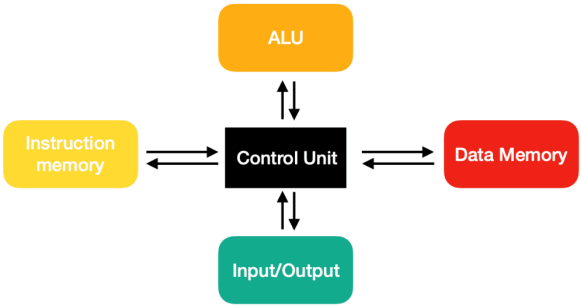


FIGURE 4.3: Harvard Architecture Diagram
Source: Teach Computer Science

Cortex M7

The Cortex-M7 is a high-performance MCU core with digital signal processing (DSP) extensions, hardware division support as well as instruction and data caches, the Cortex-M7 excels in computational tasks and is based on the Armv7-M architecture. Its adaptability and processing speed make it a preferred choice for applications where both performance and low-latency execution are critical.

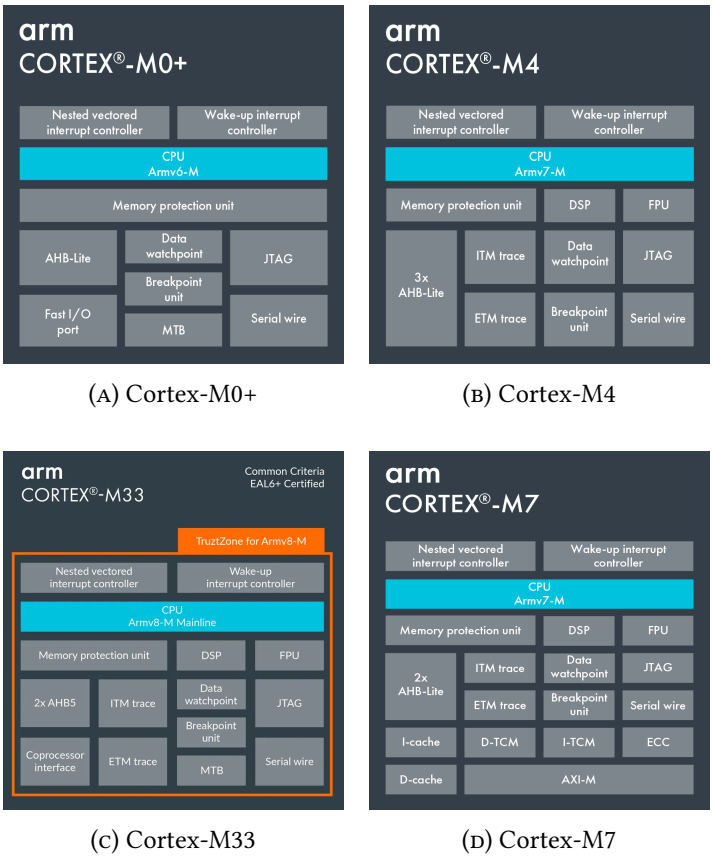
Caches

As mentioned in the ARM® Cortex®-M7 Processor Technical Reference Manual [28] the architecture for the Cortex-M7 is Harvard, meaning only instructions can be fetched from instruction cache and only data can be read from and written to the data cache as can be seen in Fig. 4.3.

4.4 Single Instruction Multiple Data (SIMD)

Cortex-M cores equipped with the DSP extension (e.g., M4, M33, M7) offer robust support for SIMD instructions. These instructions efficiently operate on operands that have been packed into 32-bit registers, whether they are 8- or 16-bit in nature. Consequently, these SIMD instructions empower the processor to perform either dual 16-bit or quad 8-bit calculations, depending on the specific operation at hand.

SIMD instructions play a pivotal role in dramatically boosting the performance of DSP algorithms, particularly in the realm of digital filters. These algorithms often entail a multitude of multiply and sum calculations within a data pipeline. SIMD instructions excel in optimizing these operations, resulting in substantial gains in computational efficiency for such tasks [29].



(A) Cortex-M0+ (B) Cortex-M4

(C) Cortex-M33 (D) Cortex-M7

FIGURE 4.4: Block Diagram for Cortex-M Cores
Source: Arm

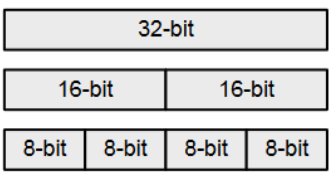


FIGURE 4.5: 32-bit word fitting 2x8-/4x16-bit values

4.5 Flexible second operand (Operand2)

Many ARM and Thumb general data processing instructions feature a flexible second operand, referred to as Operand2 in their syntax descriptions [7]. Operand2 can take on various forms, including a constant value or a register with an optional shift operation. This flexibility enables these instructions to perform a wide range of operations, making them highly versatile for arithmetic and logical computations in ARM and Thumb instruction sets, enhancing the efficiency and adaptability of low-level programming for diverse tasks.

Operand2 as a constant

In ARM instructions, constants can encompass values achievable by right-rotating an 8-bit value rightward by any even number of bits within a 32-bit word.

Operand2 as a register with optional shift

In ARM instructions, Operand2 for data processing is defined in the format "Rm , shift," where "Rm" represents the register containing the data for the second operand. Additionally, "shift" is an optional parameter that allows for various types of bitwise shifts and rotations to be applied to the value in Rm. These shift operations can be:

- ASR #n: Signifies an arithmetic shift right by n bits ($1 \leq n \leq 32$).
- LSL #n: Denotes a logical shift left by n bits ($1 \leq n \leq 31$).
- LSR #n: Indicates a logical shift right by n bits ($1 \leq n \leq 32$).
- ROR #n: Represents a rotate right by n bits ($1 \leq n \leq 31$).
- RRX: Stands for a rotate right by one bit, with extend.

These shift operations are facilitated by the barrel shifter, a hardware component in ARM processors. The barrel shifter can rapidly perform these shifts and rotations on binary data, providing efficient and flexible data manipulation capabilities within ARM instructions. Importantly, specifying a register with a shift operation can also impact the carry flag in certain instructions, allowing for precise control over data processing operations while optimizing performance.

Chapter 5

Utilized Tools, Frameworks and APIs

5.1 TensorFlow

TensorFlow [8] is an open source machine learning framework developed be Google, that facilitates the creation, deployment and management of various deep learning models. It supports a variety of features ranging from high level APIs such as Keras to low level APIs for fine-grained tuning of NN models.

TensorFlow Lite

TensorFlow Lite is an optimized lightweight implementation of the TensorFlow framework that targets resource constrained devices such as mobile phone, embedded and IoT devices. Pretrained models can be converted to the efficient flatbuffer format (.tflite) and optimizations such as Quantization (mentioned in Section 3.2) can then be applied. Furthermore, custom operations that are specifically tailored to the the targeted hardware are used allowing for more efficient execution.

TensorFlow Lite offers a variety of quantization based optimizations that vary in implementation depending on the targeted hardware and access to a Representative Dataset. The process can be over-viewed by observing the decision tree Fig. 5.1. The quantization specification applied to all pretrained TensorFlow Lite models in this work will be done by limiting the model to 8-bit integer operations using a representative dataset.

The floating point values are approximated using the formula as described in Equation 5.1.

$$realvalue = (int8value - zeropoint) \times scale \tag{5.1}$$

TensorFlow Lite for Microcontrollers

The TensorFlow Lite for Microcontrollers (TFLM) framework takes the optimization a step further and has been designed and tested for the Arm Cortex-M Series architecture, discussed in Section 4.2. Designed for memory constrained on device inference, it doesn't require an operating system, any typical C or C++ libraries, or the allocation of dynamic memory.

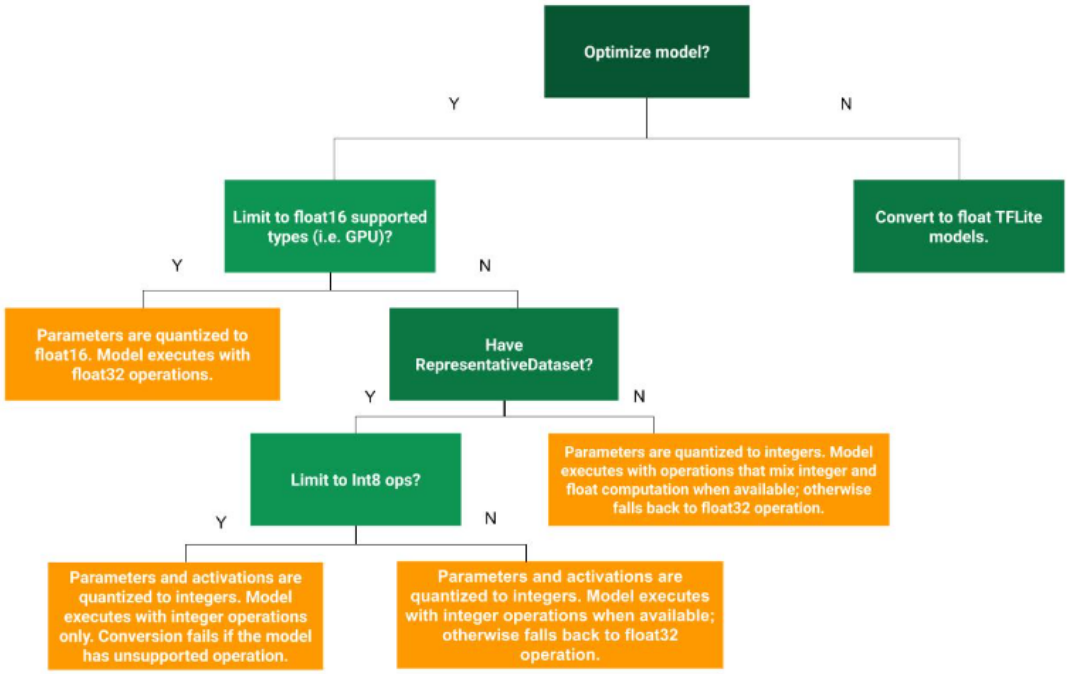


FIGURE 5.1: Decision tree aiding quantization method selection
Source: TensorFlow

Deployment of models using TFLM requires a TensorFlow model to be converted to TensorFlow Lite and then to a C byte array, the NN is then deployed using the TFLM C++ library. The TensorFlow model has to follow the space constraints of the target device as well as including operations supported by the TFLM reference kernels.

5.2 CMSIS NN

The TFLM framework includes the reference kernels for executing the various functions of a NN model. These kernels have not been optimized ,but rather serve as a collaborative starting point for target specific optimizations. One such optimized library is CMSIS NN by Arm [2] ,which comprises a set of highly efficient neural network kernels designed to achieve optimal performance and minimize memory usage when running neural networks on Arm Cortex-M processors.

5.2.1 Operation support

The optimized kernels are written with a specific architecture in mind, each operator supporting different implementations based on the processors available hardware extensions. The operators have been implemented for 3 architecture features :

- **Pure C** implementation for processors that do not support hardware extensions.

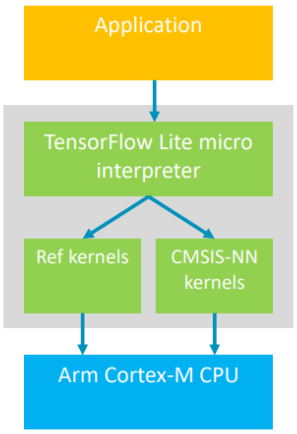


FIGURE 5.2: Optimized Kernels via TFLM: CMSIS-NN and TF Lite for MCUs
Source: TensorFlow Website

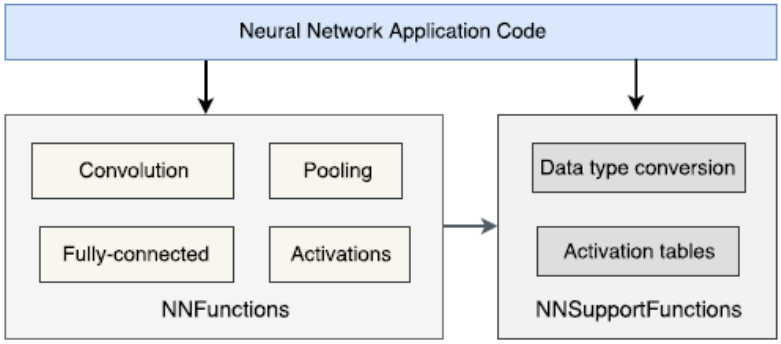


FIGURE 5.3: Overview of the neural network kernel structure
Source:[2]

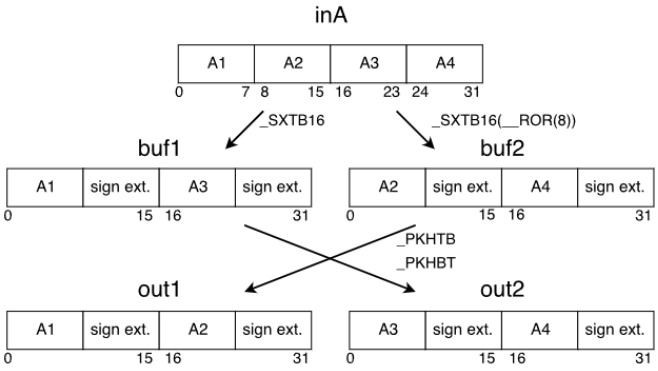
- **DSP** implementation that utilizes SIMD instructions.
- **MVE** implemenntaion for processors with Arm Helium Technology.

The neural network kernels Fig. 5.3 consist of the main NN Functions, developed to support 8- and 16-bit data, as well as leveraging SIMD instructions (analyzed in Section 4.4) on supported Cortex-M MCUs Sec. 4.3. These kernels implement NN typologies, comprised of common layer types (eg. convolution). Additionally, the NN support functions execute utility tasks (e.g. data conversion) and can also be used to implement more sophisticated NN modules.

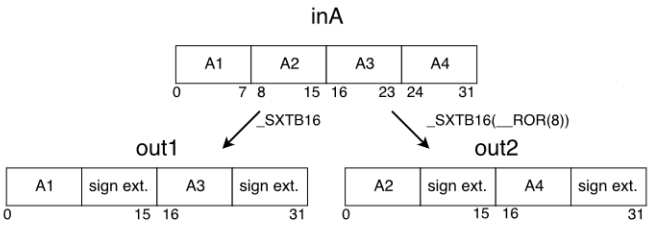
Similar to the aforementioned operator implementations different kernel implementations are used according to the NN parameters that include optimizations tied to specific NN parameters, while basic versions also exist for compatibility.

5.2.2 Data Transformation

The majority of NNFunctions use 16-bit MAC instructions thus, data conversion from 8- to 16-bit data type is required. Depending on operand ordering the conversion implementation



(A) Sign Extension without Reordering



(B) Sign Extension

FIGURE 5.4: Operand order dependant 8- to 16-bit conversion
Source: [2]

process can be illustrated by Fig. 5.4. Since, the data transformation is required in the innermost loop of the convolution operation care must be taken to optimize the computational overhead it introduces. In the case of same operand ordering Fig. 5.4a only the data conversion, implemented by the dedicated SIMD SXTB16 instruction, is needed. However, when the operands do not follow the same ordering Fig. 5.4b a second rearrangement step follows, using the dedicated SIMD PKHTB and PKHTT instructions.

5.2.3 Matrix Multiplication

In the case of CNN models discussed in Section 3.1 the most computationally intensive part of the convolution operation is matrix multiplication, which is implemented by the `mat_mult` kernels in CMSIS with 2x2 kernels Fig. 5.6. This kernel size not only reduces the total instruction count but also promotes data reuse. The calculation of individual partial products is performed using the dedicated SMLAD Fig. 5.5 SIMD instruction, with a 32-bit accumulator and 16-bit operands. It is noteworthy that the matrix multiplication operation is done by using partial products across 4 channels as can be seen in the pseudo-code provided by ARM in Fig. 5.5.

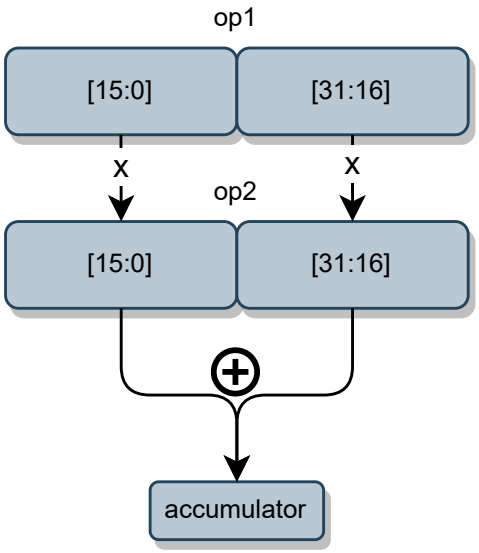


FIGURE 5.5: The SMLAD instruction

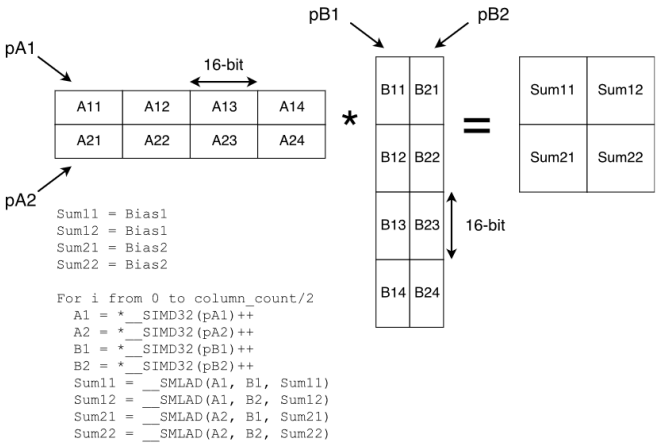


FIGURE 5.6: The inner-loop of matrix multiplication with 2 × 2 kernel
Source: [2]

5.3 STM32 Software Development Tools

STMicroelectronics provides a suite of powerful development tools for STM32 microcontrollers, offering simplified configuration and advanced debugging, to aid engineers and developers in creating innovative embedded solutions.

5.3.1 STM32CubeMX

STM32CubeMX [30], is an essential tool in the STM32 development ecosystem provided by STMicroelectronics. It simplifies the initial setup and configuration process for STM32 microcontroller-based projects. With a user-friendly graphical interface, developers can effortlessly select the appropriate STM32 microcontroller model, configure pin assignments, establish clock settings, and set up various on-chip peripherals like UART, SPI, I2C, and timers.

One of its standout features is its ability to automatically generate initialization code and drivers based on the configured settings, significantly reducing development time and minimizing the risk of configuration errors. Additionally, STM32CubeMX allows for easy project export to popular integrated development environments (IDEs) like STM32CubeIDE facilitating a seamless transition from configuration to coding.

X-CUBE-AI [9], is a comprehensive software package developed by STMicroelectronics, tailored for STM32 microcontroller-based applications seeking to incorporate artificial intelligence and machine learning capabilities. It empowers developers to efficiently convert and deploy AI/ML models on resource-constrained embedded systems. This package offers optimized inference libraries, supports popular machine learning frameworks, and enables seamless integration of AI/ML models into STM32-based projects. With a focus on resource optimization and compatibility with STM32CubeIDE, X-CUBE-AI facilitates real-time and low-latency AI/ML processing on edge devices, making it a valuable tool for those seeking to leverage the power of AI in their embedded applications.

5.3.2 STM32CubeIDE

STM32CubeIDE [10] is an Eclipse®/CDT™ based C/C++ development environment, offering functionalities such as peripheral configuration, code generation and code compilation integrated from STM32CubeMX, as well as debug features that streamline the installation and development procedure on STM32 MCUs.

More notably, it includes tools like build and stack analyzers, which provide essential insights into project status and memory usage. Furthermore, it offers advanced debugging capabilities, giving developers access to CPU core registers, memory details, peripheral registers, live variable monitoring, Serial Wire Viewer interface support, and a fault analyzer.

Chapter 6

Proposed Framework

Throughout this thesis, we have explored various optimization avenues. These include traditional optimization techniques, such as loop nest optimizations, and a detailed examination of the generated assembly code for computationally intensive kernels. Additionally, we have delved into novel and previously unexplored methods involving kernel-based modifications to harness the principles of Approximate Computing, as discussed in Section 3.3.

The foundation of our framework is the CMSIS_NN library [31], as discussed in Section 5.2. This library serves as an open-source, highly optimized, and standardized starting point for deploying neural networks on a wide range of ARM-based Cortex-M MCUs, as detailed in Section 4.3.

To meet the unique requirements of our deployment scenarios, we consider CMSIS-NN as our baseline inference library [11, 12]. Our approximation framework is built upon the customization of the generated code, ensuring it supports only the essential layers and functions required by the given model. Unlike CMSIS-NN and most existing inference libraries (e.g., TF-Lite Micro [8]), we offload operations related to model structure parameters from runtime to compile time. Consequently, only the code necessary for the provided model is executed. This not only enhances inference efficiency but also reduces flash memory usage (in some cases up to 30%). This extra availability of flash memory enables us to unpack a greater number of kernels or even entire layers, thereby enhancing the granularity of our approximation.

As most of the cycles in CNN models, similar to those examined in this work, are consumed by convolutional layers [32], our primary focus lies on optimizing this specific operator. A convolution operation, as implemented in CMSIS-NN, involves computing a dot product between filter weights and a small receptive field within the input feature map. This is accomplished through an initial input reordering and expansion step, followed by a matrix multiplication operation. Software kernels adopt a power-of-two scaling quantization of *int8* data type format for the weights, while for activations it increases to int16. This work extends these kernels including cycle counters that can be used to profile parts of the C code for individual operators. This allows the user to profile the model and gain insights into its baseline performance, highlighting areas that necessitate further acceleration. Though, since these counters are intended for profiling purposes, they are deactivated during runtime. A

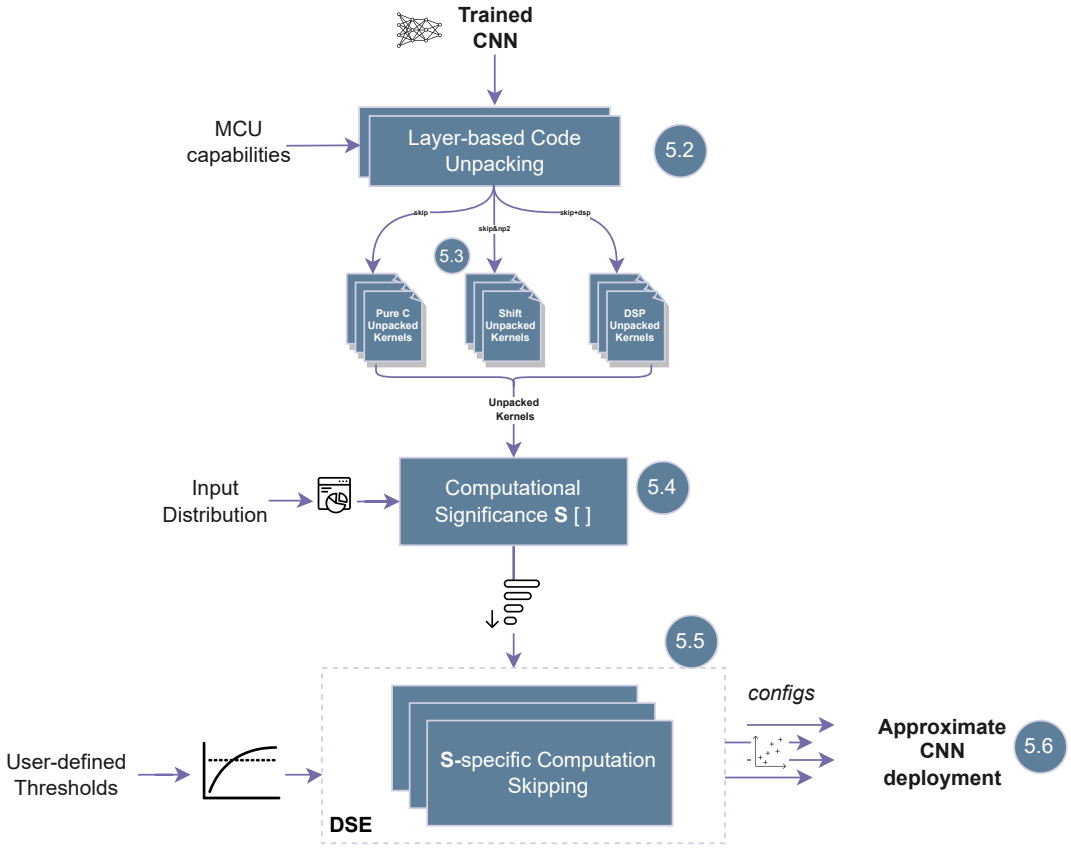


FIGURE 6.1: Proposed Framework Detailed Overview

detailed outline of this process is provided in Fig. 6.1.

6.1 Initial Optimization Experiments

6.1.1 Loop Unrolling

Loop unrolling aims to improve loop performance through loop iteration reduction [33]. The primary idea behind this method is to perform multiple loop body statements within a single iteration. The key takeaways from the unrolling optimizations are the following:

- **Loop Overhead Reduction** is achieved by reducing branch instructions and counter iteration.
- **Instruction Parallelism** can take advantage of a processors ability to execute multiple instructions at once.
- **Cache Efficiency** can also be improved by encouraging data reuse through multiple iterations.

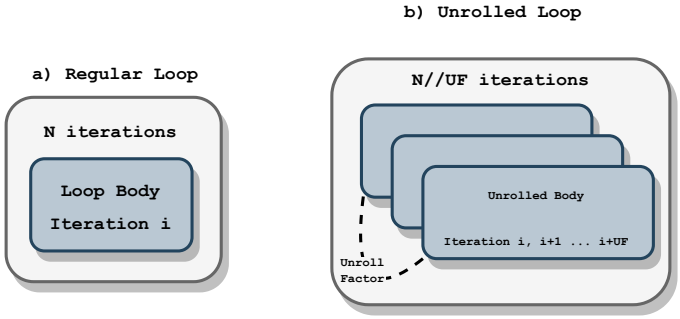


FIGURE 6.2: Loop Unrolling Schematic

Unrolling Strategy

The CMSIS_NN library performs `im2col` convolution using multiple nested loops. Loop Unrolling was applied to the loop iterating over `kernel_y` in `arm_convolve_s8` and using a switch statement the appropriate unroll factor was applied to each individual layer.

The entire process can be effortlessly automated through the application of code injecting python scripts on the target convolution function. In order to improve code readability and reduce the script complexity , while observing no measurable performance degradation, an inline helper function that encompasses the loop body was used.

Another, way of implementing the Loop Unrolling optimization is through the use of `#pragma`, a directive providing additional information to the compiler concerning the compilation and optimizations of specific sections of code. As mentioned in the Arm Compiler armclang Reference Guide [34] the directive `#pragma unroll n` will unroll the specified loop by a factor of n if it has not been automatically unrolled assuming -O3 optimization level.

Trade-offs

Although, Loop Unrolling can potentially provide significant performance improvement both of the aforementioned implementation methods have drawbacks. On one hand the use of `#pragmas` can be easier to implement and maintain but it relies on the compiler performing the optimizations meaning that the user lacks granular control. On the other hand, manual unrolling can be fine-tuned by the user but it requires deep understanding of the target architecture and is much more difficult to port and maintain. Moreover, both lead to increased code size and register usage often leading to performance degradation rather than improvement.

6.1.2 Loop Tilling

Loop tiling [33] , also known as loop blocking, is a compiler and code optimization technique used to improve the cache locality and overall performance of nested loops. It is particularly

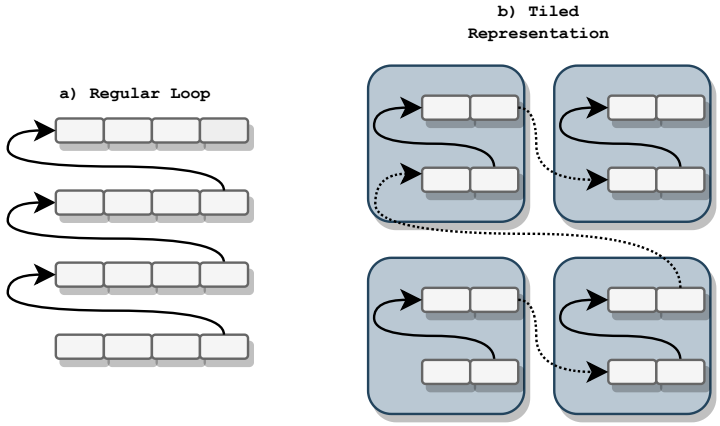


FIGURE 6.3: Loop Tilling Schematic

beneficial in situations where memory access patterns can be a bottleneck, such as when dealing with large datasets or multi-dimensional arrays. Loop tiling achieves this optimization by dividing a multi-dimensional loop into smaller, regularly sized blocks or tiles, and processing these blocks sequentially.

Advantages

- Improved Cache Efficiency through data processing using appropriately sized blocks that fit into cache. This leads to a cache miss reduction and subsequent performance improvement.
- Improved Parallelization though potential independent execution of different blocks by the processor.

Tilling Strategy

The `output_y` loop of `arm_convolve_s8` was targeted for Loop Tiling and a range of block sizes are tested. This process can be implemented simply by performing the standard tilling transformation and adjusting block size between different runs.

However, Loop tilling leads to increased loop complexity as additional loop and index calculations are introduces. Furthermore, it may not always lead to performance improvements since the performance benefits are tied to specific memory access patterns, cache sizes and hardware architecture.

6.2 Layer Based Code Unpacking

Typical convolution kernels on MCUs are usually implemented in a matrix format, where inputs and weights are retrieved from memory using a specific pattern. This pattern includes details such as the order in which data elements are fetched, the stride or step size for moving

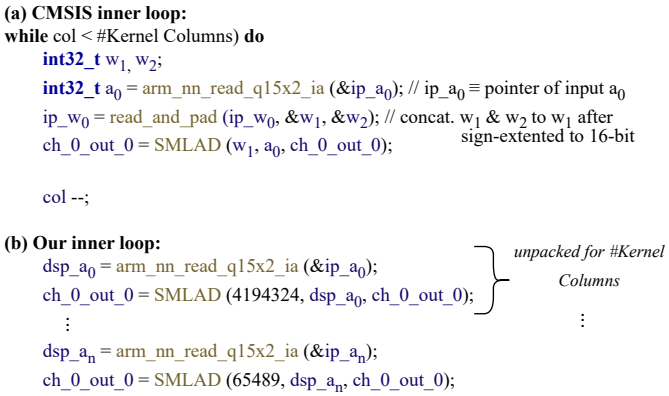


FIGURE 6.4: Pseudocode of the inner-loop of mat mult kernel implementation based on CMSIS (a) and based on the code unpacking optimization (b). For simplicity, 1×1 kernel has been considered and each loop computes the dot product result of 1 column and 1 row, i.e., 1 output.

through the data and any necessary padding or adjustments to ensure the correct alignment of the data for convolution. Instead, an automated layer-based code unpacking is performed, where each operation is "unpacked" and included as an intrinsic function in the final generated code. An illustration of a matrix multiplication kernel similar to CMSIS implementation is shown in Fig. 6.4a. As shown in Fig. 6.4a, the mat_mult kernel performs dynamic computations between inputs a_i and weights w_i, with *ch_0_out_0* denoting the accumulated output per channel. The corresponding and automatically generated unpacked code is illustrated in Fig. 6.4b.

Code is automatically generated using a python script, the original matrix multiplication C header file is modified with appropriate flags allowing easy code injection by the end user. A python function has been created with a number of options ranging from control over the unpacking range to options to modify C code lines to either perform shift based MACs further discussed in Section 6.3 or even omit entire code segments (Section 6.4). The DSP capabilities of modern ARM MCUs have also been taken into consideration allowing for alternative DSP code injection.

The respective benefits of the code unpacking process include the followings:

- Fixed weight allocation to each operand removes the necessity to adapt and load the weights properly during the convolution process. This leads to simplified and predictable, in terms of type, operations that can be adjusted based on input values to enhance inference speed.
- Branch instruction overhead elimination.
- As mentioned in 5.2 CMSIS mat mult kernel calculates the partial products using the SMLAD instruction (SIMD logic), which performs two 16-bit signed multiplications, accumulating the results into a 32-bit operand. Hence, a pre-processing is required

to convert the data to the 16-bit data type. To this end, our fixed-weight replacement avoids this time-consuming operation. Since we know apriori the values of weights, this is easily avoided by an offline processing that involves concatenating two int16 (sign extended int8 values to int16) weights. As an instance, the value '4194324' of Fig. 6.4b represents w1=64 and w2=20

Trade-Offs and Takeways

The main consideration of code Unpacking is the inherent code-size increase that it introduces, thus leading to increased flash memory usage. Special care must therefore be taken in order fully utilize the entire available flash storage of the target MCU while also adhering to the tight memory budget of resource constrained devices.

The length of the unpacked code is considered w.r.t. the available unused flash memory, creating an interesting trade-off between these two metrics. The commonly unexploited flash usage appears promising for storing customized and pre-defined constant operations that will enable faster overall CNN execution. Given that, in the majority of cases, works like [12] and [17] report an average flash memory utilization of less than 25% (specifically, an average of 41% among the examined models in [12] and 8% of those in [17]), it becomes evident that a fully unpacked fixed-weight convolution can effortlessly be enabled. For instance, even in the worst case (i.e., AlexNet with 5 convolution layers) our framework fitted the whole kernel instructions utilizing less than 60% of the available flash memory.

The key takeaways from the code unpacking are twofolds. First, the new modified code triggers the execution of lowlevel processor-specific instructions, taking full advantage of the processor's capabilities and achieve better performance than using generic, high-level language constructs. Second, the layer-based code unpacking, offers a high level of customization at the lowest level, allowing for tailored modifications/approximations to specific operands. This level of customization is typically unattainable in generic kernel implementations or would involve substantial overheads from branch instructions. At this point it is crucial to note that unpacking the entire kernel for all layers of a network would not only introduce a considerable latency overhead, but would also risk overloading the cache memory.

6.3 Np2 Rounding

Nearest Power of Two (Np2) Rounding is proposed as an alternative quantization strategy for deep neural networks (DNNs), where the weights of the DNN are quantized to a representation determined by Equation 6.1.

$$w_i = 2^n \tag{6.1}$$

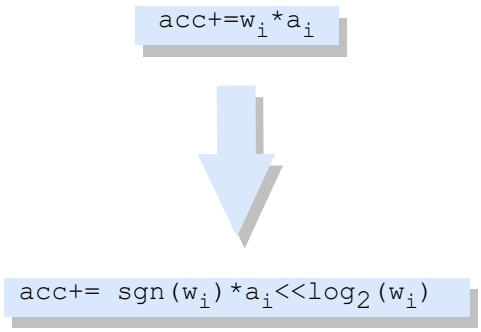


FIGURE 6.5: Power Of Two C Implementation

The main advantage of this method is the ability to perform multiple accumulate operations through the combination of the ADD instruction for the accumulation and a logical shift left #LSL facilitated by the inline barrel shifter to handle multiplications (mentioned in Section 4.5) and executed with a single instruction.

Np2 Rounding can be used in scenarios where a low-end, low-power processor that lacks DSP capabilities needs to perform a large amount of MAC operations as is the case when running DNN inference.

Deployment Method

Hardware support for this type of quantization can be easily implemented by modifying the CMSIS-NN matrix multiplication kernel as seen in Fig. 6.5. Writing code in this manner results in optimized assembly code generation by the compiler when using the Ofast optimization strategy, specifically depending on the sign of the current weight the MAC equivalent instruction will be implemented with a single ADD or SUB.

Moreover, the aforementioned unpacking strategy enables all required calculations (i.e determination of the weight sign and shift value) to be done offline and hard-coded into the unpacked code thus reducing costly operations in the innermost loop of the convolution operation.

6.4 Significance Aware Skipping

Layer based code unpacking can leveraged to systematically omit certain operations that are considered insignificant [35] or choose to retain as needed some others. Other approaches do exist as mentioned in Section 2.2 where entire channels or even layers are skipped but this approach operates at a finer granularity analyzing operations at the kernel level as its highest resolution. The significance-driven analysis is motivated by two facts:

- Each computation within the convolution makes a unique contribution to the final output, meaning that certain computations could potentially be skipped without compromising classification accuracy

- Effectively reducing the total number of computations could provide a valuable tradeoff between accuracy and latency

Data Handling

An integral part of the Computation Skipping process is the collection of the necessary data samples that will be used to calculate the each layers mean values.

Data collection can be achieved through two primary methods: on-device collection using the Serial Communication Protocol or through the Pure C implementation of the CMSIS NN library in a suitable development environment. On-device logging provides real-time information about the code's execution, but it can become cumbersome to implement, potentially leading to synchronization errors when high baud rates are used. Furthermore, it may have limitations in terms of logging capabilities, especially when using an appropriate baud rate. On the other hand, data logging becomes a much simpler task when simulating runtime on a desktop PC. In this scenario, data files are logged locally, eliminating the need for Serial Communication and simplifying the data collection process.

For the scope of this work, our focus will exclusively revolve around the data within the CMSIS NN matrix multiplication kernel. It's worth noting that the matrix multiplication kernel in each layer is iteratively applied multiple times during the convolution process, adhering to the pattern described in Equation 6.2, which is the expected behavior when employing im2col convolution techniques.

$$mat\_mult\_counter = \frac{output\_dims\_x \times output\_dims\_y}{2} \tag{6.2}$$

While the sample dataset, which serves as the foundation for generating all other data and neural network (NN) weights, is accessible to the end user, it's important to note that this data is solely adequate for computing partial products in the first convolution layer. Consequently, the subsequent samples, derived from the outputs of each layer and subsequently passed as inputs to the next layer, must be collected on a per-layer basis, as illustrated in Fig. 6.6. The data logging code is strategically integrated within the matrix multiplication kernel to meticulously record all layer input data in pairs that are one row apart.

This procedure streamlines the logging of all layer data into a single file, making it exceptionally user-friendly for subsequent calculations. Our proposed method for data grouping is elegantly straightforward: it arranges all data in a sequential order while simultaneously marking their origins with indexes that correspond to the data segment, layer, and the iteration of the matrix multiplication kernel for each layer.

This process corresponds to the creation of an array for each index group, and each of these arrays contains the input data necessary for calculating the partial products within the corresponding matrix multiplication kernel. Each of these arrays is structured with dimensions of $rows \times cols$, mirroring the MAC (Multiply-Accumulate) operations performed by that specific kernel.

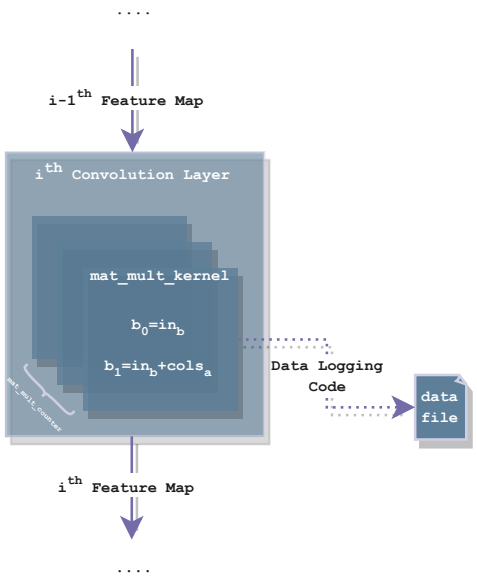


FIGURE 6.6: Per Layer Data Collection

Given that all the data is consolidated into a single file, some pre-processing is necessary to generate appropriate NumPy arrays that can be utilized in the subsequent Python scripts.

In the initial stage of data pre-processing, we embark on the task of partitioning the substantial file that houses input data for all data segments into appropriately sized header files, each containing multiple NumPy arrays. Within this framework, files are partitioned at intervals of 100 data segments, resulting in the creation of 10 header files for datasets comprising 1000 samples. This partitioning strategy assumes paramount importance due to the inherent limitations of NumPy in efficiently managing large header files containing multiple NumPy arrays

For each data segment, the partitioned and logged kernel data is then further divided into distinct NumPy arrays. These arrays are organized into folders, with each folder corresponding to a specific input type, denoted as b_0 and b_1. Within each of these folders, a collection of NumPy arrays, precisely matching the $mat\_mult\_counter$ value for that particular layer is found.

Mean Value Based Significance Calculation

Once all the data has been systematically organized, a straightforward Python script can be developed to reconstruct all the partial products generated by the corresponding convolutional kernel for each layer within the sample dataset. These partial products can then be aggregated across all matrix multiplication iterations and data segments to compute the mean value arrays for each layer efficiently.

Each of the mentioned mean value arrays is calculated across the four partial product channels, as detailed in Section 5.2. Separate mean value arrays are generated for each layer

and channel, ensuring a comprehensive analysis of the data.

The accumulation of each channel during the matrix multiplication (*mat_mult* kernel) is calculated based on a weighted sum and an initialized bias:

$$Sum_c = b + \sum_{\forall i} a_i \cdot w_i, \tag{6.3}$$

where, b is the initialized bias, w_i are the trained coefficients (weights) and a_i are the inputs from the respective channel. Intuitively, when inputs are multiplied by large numbers, they tend to produce significantly more impactful products ($a_i \cdot w_i$) in the final result compared to inputs multiplied by small values. However, it is worth noting that the significance of the product $a_i \cdot w_i$ also depends on the value of a_i. Thus, we define the *significance* of each product as follows:

$$S_i = |\frac{\mathrm{E}[a_i] \cdot w_i}{\sum_{\forall i} \left(\mathrm{E}[a_i] \cdot w_i\right)}|. \tag{6.4}$$

In other words, (6.4) calculates the long-term expected outcome of each product $a_i \cdot w_i$ over the total sum Sum_c of the respective channel. If the sum equals zero, we consider the corresponding S_i to be large, and thus, the product is retained. For each channel and Sum_c, the calculation of $S_i, \forall i$, is straightforward and involves capturing the input distribution from a portion of the dataset. By exploiting this high-level information, we minimize the total computations required for each summation operation Sum_c at compile time, and thus, summation is approximate accordingly. Specifically, for each product $a_i \cdot w_i$, if S_i is less or equal to a given threshold τ, it is incorporated into the generated code, while others are omitted. Thus, our approximate summation per channel is now represented by:

$$Sum'_c = b + \sum_{\forall i} (a_i \cdot w_i) - \sum_{\forall i : S_i < \tau} (a_i \cdot w_i) \tag{6.5}$$

A Python function has been crafted to manage the execution of the aforementioned process. It's worth emphasizing that while four channels are employed for the computation of partial products, at the conclusion of each row calculation, the ultimate outcomes undergo re-quantization. Specifically, channels ch_{00} and ch_{10} are re-quantized and summed into out_0, while channels ch_{01} and ch_{11} are re-quantized and accumulated into out_1.

The specific approach of managing channels by the matrix multiplication kernel necessitates meticulous attention to ensure that the corresponding outputs maintain their balance once the skipping procedure is completed. In this study, we focus on the groups $[ch_{00}, ch_{01}]$ and $[ch_{10}, ch_{11}]$, which implies that only the shared insignificant values within each group will be skipped to uphold this balance.

After the computation skipping process is completed, files are generated for specific threshold values associated with each channel and layer. These files include shared skip indexes for each group, organized in rows and columns, making them suitable for TensorFlow

Lite model Generation. Additionally, these files contain essential information regarding the final percentage of skipped calculations, which will be employed later in the Design Space Exploration stage.

6.5 Exhaustive Design Space Exploration

Exhaustive Design Space Exploration (DSE) is a critical process in optimization and algorithm design, involving the thorough examination of all potential configurations, parameters, or options within a given design space. This exploration can be quantified as the product of the actual number of thresholds to be tested and the number of layers under consideration. The resulting number of possible configurations can be immense. For instance, consider the example of AlexNet, a relatively straightforward Convolutional Neural Network (CNN) architecture with 5 layers. If each layer must be exhaustively tested against 100 different thresholds, the total number of unique combinations to explore would be a staggering 100^5, equivalent to 10,000,000,000 (ten billion) configurations. This overwhelming complexity underscores the formidable challenge of navigating and optimizing the entire design space, emphasizing the need for effective strategies to manage and prioritize configurations within practical computational constraints.

Selection of Design Space

Given the constraints mentioned earlier, particularly in the case of exploring more intricate CNN architectures where comprehensively covering the entire design space is infeasible, it becomes imperative to consider strategies for limiting the design space. This can be achieved by either restricting the number of layers under examination, thereby reducing problem complexity, or by narrowing down the range of threshold values to be explored. These design space limitation strategies are essential for striking a balance between thorough exploration and practical feasibility, allowing for more efficient and targeted optimization processes.

During the model import step, one can readily extract the parameter count of each layer and compare it against the total neural network parameter count. This initial analysis provides a rough estimation of the potential latency improvements that each layer can contribute. Consequently, layers with a relatively small contribution to the overall neural network parameter count can be selectively omitted from the exploration process. By doing so, the total exploration complexity is reduced, enabling a more focused and efficient optimization effort.

Another approach to mitigate the computational overhead of Exhaustive Design Space Exploration (DSE) is by constraining the number of thresholds to be explored. This can be accomplished by condensing histograms for each layer based on their mean values. The resulting distributions can be visualized, and from these, a preliminary estimate of the optimal threshold values can be deduced. By strategically narrowing down the range of threshold

values under consideration, the exploration process becomes more efficient, focusing on configurations with the most potential for optimization.

Automated Approximate Model Generation and Evaluation

In the evaluation step of the procedure, a series of Python header files is generated, each corresponding to a specific combination of significance thresholds for each layer. These significance indexes are then leveraged to create files in the TensorFlow Lite format, where the corresponding insignificant weights are set to zero. These modified models can be utilized not only during the evaluation phase of the Design Space Exploration (DSE) procedure but also in tandem with the unpacking step described in Section 6.2. This collaborative usage streamlines the model deployment process, offering a simplified and optimized approach.

Once the candidate models have been generated, they undergo evaluation on a distinct subset of the test dataset to gauge the extent of accuracy loss in comparison to the baseline model. This evaluation process can be time-consuming since multiple models must be assessed using a substantial number of samples. This underscores the significance of effectively constraining the design space. Simultaneously, the amount of skipped calculations is recorded for each configuration, forming two axes for assessment: one for accuracy loss and another for the percentage of skipped computations. This multidimensional analysis aids in identifying optimal configurations that strike a balance between computational efficiency and model accuracy.

The concluding phase of the Design Space Exploration (DSE) process involves the visualization of the previously conducted multidimensional analysis. All the configurations generated throughout the exploration are represented as data points scattered across the analysis space. In this visualization, the Pareto optimal values are superimposed, providing a clear delineation of the trade-offs between accuracy loss and the percentage of skipped computations. This visual representation aids in identifying configurations that reside on the Pareto frontier, showcasing the optimal solutions that strike an equilibrium between computational efficiency and model accuracy. It serves as a valuable tool for the end user to make informed choices and select the most suitable configurations for deployment or further refinement providing granularity in the decision-making process between accuracy and performance.

6.6 Deployment Technique

To harness the benefits discussed in this chapter, a combination of techniques is required. While obtaining an optimal TensorFlow Lite model based on user-defined design constraints involves setting the weights to zero, this step alone won't yield significant latency improvements. To further enhance performance, the layer-based unpacking process is employed. The optimal model is unpacked, and all weights with zero values are excluded. This not

only reduces the number of Multiply-Accumulate (MAC) operations in the matrix multiplication kernel but also mitigates some of the drawbacks associated with the unpacking process. Fewer lines of code result in reduced flash usage and potentially improve the performance of the instruction cache.

At the end of the unpacking process, the generated matrix multiplication kernels for each layer can be implemented in various ways. If the processor supports the DSP extension, the kernels will be generated by unpacking the weights in groups of 2 per SMLAD instruction. Otherwise, the Pure C implementation will be used, and the end user will have the option to enable the np2 optimization (Sec. 6.3) if the neural network has been appropriately trained. This optimization replaces costly MAC instructions with simple shift-enabled additions, further refining the implementation for improved performance when dealing with low-end MCUs.

Chapter 7

Experimental Results

7.1 Testing Environment

In order to accurately evaluate the experimental results discussed in this section, it is essential to provide a comprehensive description of the equipment utilized for the subsequent assessments, as well as the specific hardware and software development tools employed throughout each experiment.

7.1.1 Development Hardware

In the context of the experiments discussed in the following sections, we evaluated various development boards featuring ARM Cortex M Cores (as elaborated in Section 4.2). Specifically, we focused on STM32 implementations of the Cortex M Cores due to their extensive toolset accessibility (as detailed in Section 5.3), which greatly streamlines all aspects of the development process, ranging from debugging to the final deployment. Nucleo boards, chosen for this evaluation, offer the advantages of being cost-effective, well-documented, and fully compatible with the aforementioned development tools. Furthermore, in alignment with similar works [11, 12], Nucleo boards have established themselves as the industry standard for evaluating neural networks on microcontroller-class devices.

Nucleo boards offer a range of options tailored for diverse applications. For our experiments, we opted for the NUCLEO-U575ZI-Q board, which features an STM32U575ZIT6Q microcontroller unit (MCU) based on the ARM Cortex-M33 core (as discussed in Section 4.3). This choice strikes a balance, positioning itself between the high-performance Cortex-M7 variants and the more basic MCUs using the Cortex-M3 core.

As illustrated in Fig. 7.1, the chosen MCU operates at a clock frequency of 160 MHz and incorporates the STM32 ART Accelerator, providing cache functionality in the form of 8KB ICache and 4KB DCache, located outside the ARM Core. Additionally, it offers 2 megabytes of Flash memory and 768 kilobytes of RAM.

Notably, the NUCLEO-U575ZI-Q board comes equipped with a built-in STLINK in-circuit debugger and programmer, which significantly simplifies the debugging and programming

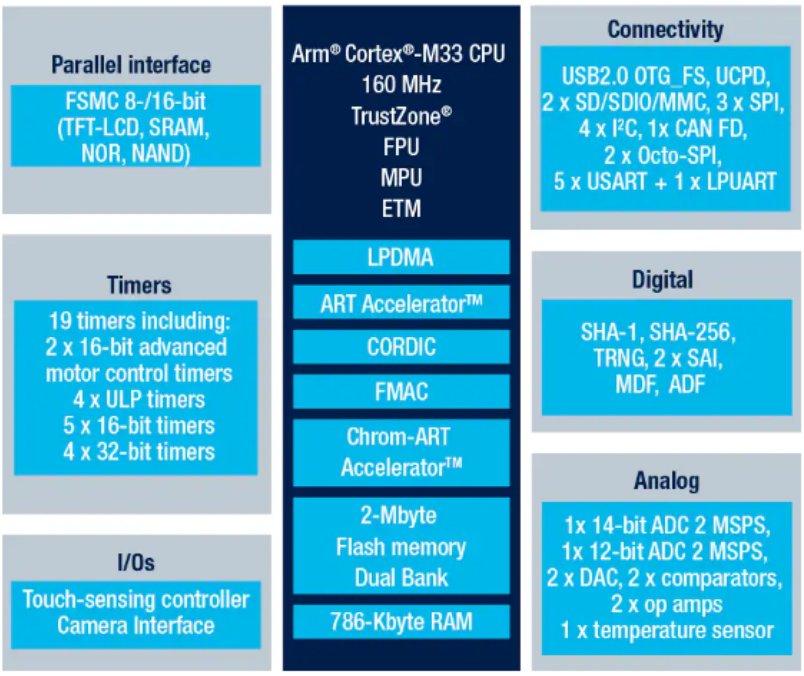


FIGURE 7.1: Circuit Diagram of the STM32U575ZIT6Q MCU
Source: ST

process. This functionality is seamlessly facilitated via a USB interface, eliminating the need for external hardware.

7.1.2 Development Software

The development process for this experimentation consists of distinct stages, each tailored to specific requirements.

1. Data Collection: In the first stage, we gather the necessary data for each neural network (NN) configuration to be tested. This is achieved by utilizing a C/C++ development environment to run the Pure C version of CMSIS NN. In our case, we employed the Eclipse IDE due to its similarity with Cube IDE.

2. Model Optimization: The second stage, involves a combination of Python scripts predominantly based on the TensorFlow library for evaluating and optimizing the neural network models.

3. Model Deployment: The final step employs Cube IDE for the seamless deployment of optimized models onto the target hardware.

Throughout this process, the results are collected from the board's Serial Port using the ST-Link interface and monitored using a Serial Port Monitor program, such as Putty.

FIGURE 7.2: STM32U5 Nucleo-144 board
Source:ST

7.2 Evaluated Models

The models described in this section have been selected from a variety of designs as mentioned in 3.1. The choice of the models is made based on their popularity and varying complexity meaning that they serve as adequate representation for the efficacy of the Framework.

It should at this point be noted that the evaluation is done using basic models with no tuning during the training process as a baseline, therefore while the baseline accuracy of each model may not reflect the best possible results achievable using each architecture they serve as a demonstration showcasing the capabilities of our Framework.

7.2.1 LeNet-CIFAR10

The first model to be evaluated using this design is LeNet whose training has been based on a Jupyter Notebook provided by TensorFlow as a simple tutorial for the introduction to CNNs [36]. The model has been trained using 10 epochs on the CIFAR-10 dataset reaching a baseline accuracy of 71%. The design of this NN can be viewed graphically in the generated TensorFlow summary.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 30, 30, 32)        896

max_pooling2d (MaxPooling2   (None, 15, 15, 32)        0
D)

conv2d_1 (Conv2D)            (None, 13, 13, 64)        18496

max_pooling2d_1 (MaxPoolin   (None, 6, 6, 64)          0
g2D)

conv2d_2 (Conv2D)            (None, 4, 4, 64)          36928

flatten (Flatten)            (None, 1024)              0

dense (Dense)                (None, 64)                65600
```

$Layer_i$	Parameter Count	% $Layer_i$ Contribution w.r.t Total
0	896	0.73
1	18496	**15.09**
2	36928	**30.03**
Total Layers	122570	100.00

TABLE 7.1: LeNet Parameters Per Layer

```
 dense_1 (Dense)             (None, 10)                650

=================================================================
Total params: 122570 (478.79 KB)
Trainable params: 122570 (478.79 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

By examining the summary generated, an initial assessment can be conducted to gauge the contribution of each layer to the overall parameter count, as detailed in Table 7.1. This preliminary evaluation indicates that the substantial proportion of the total neural network parameter count is primarily attributed to Convolutional Layers 1 and 2. Consequently, the design space for the LeNet exploration will be constrained to focus exclusively on these two influential layers.

7.2.2 AlexNet-CIFAR10

The AlexNet model implementation follows a generic implementation for the architecture in this case achieving a baseline accuracy of 70%.

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_5 (Conv2D)           (None, 30, 30, 32)        896

 max_pooling2d_2 (MaxPoolin  (None, 15, 15, 32)        0
 g2D)

 conv2d_6 (Conv2D)           (None, 13, 13, 64)        18496

 conv2d_7 (Conv2D)           (None, 11, 11, 64)        36928

 conv2d_8 (Conv2D)           (None, 9, 9, 64)          36928

 max_pooling2d_3 (MaxPoolin  (None, 4, 4, 64)          0
 g2D)

 conv2d_9 (Conv2D)           (None, 2, 2, 64)          36928

 flatten_1 (Flatten)         (None, 256)               0

 dense_2 (Dense)             (None, 64)                16448

 dense_3 (Dense)             (None, 10)                650

=================================================================
Total params: 147274 (575.29 KB)
Trainable params: 147274 (575.29 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

$Layer_i$	Parameter Count	% $Layer_i$ Contribution w.r.t Total
0	896	0.61
1	18496	**12.56**
2	36928	**25.07**
3	36928	**25.07**
4	36928	**25.07**
Total Layers	147274	100.00

TABLE 7.2: AlexNet Parameters Per Layer

As mentioned earlier, when reviewing Table 7.2, an initial assessment can be conducted. Once again, it is evident that the first layer should be excluded from the exploration due to its negligible parameter count in comparison to the total neural network (NN) parameters.

Furthermore, it becomes apparent that, in contrast to the LeNet model, the AlexNet model holds promise for potential latency benefits through the implementation of our Computation Skipping strategy. This is because the total parameter count, particularly within the convolutional layers targeted by our framework, has significantly increased. Additionally, the contribution of the dense layers to the overall parameter count is comparatively lower in this case representing just 11.61% of the total parameters as opposed to 54.05% in the LeNet model.

7.2.3 SqueezeNet-CIFAR10

The final model selected for evaluation is based on the SqueezeNet architecture, as outlined in the 'Modern Convnets' tutorial [37], further inspired by the work of [38] and adapted to the CIFAR-10 dataset. This architecture is notably more intricate compared to the previously mentioned ones, incorporating multiple layer types. However, it's essential to clarify that this work concentrates exclusively on the Convolutional Layer type within this complex architecture.

Significantly, this architecture achieves a baseline accuracy of 75%, and the model structure can be examined in detail in the TensorFlow model summary.

```
Model: "model_2"
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
input_3 (InputLayer)            [(None, 32, 32, 3)]  0           []

conv2d_32 (Conv2D)              (None, 32, 32, 32)   896         ['input_3[0][0]']

batch_normalization_32 (BatchN  (None, 32, 32, 32)  128         ['conv2d_32[0][0]']
ormalization)

conv2d_33 (Conv2D)              (None, 32, 32, 24)   792         ['batch_normalization_32[0][0]']

batch_normalization_33 (BatchN  (None, 32, 32, 24)  96          ['conv2d_33[0][0]']
ormalization)

conv2d_34 (Conv2D)              (None, 32, 32, 24)   600         ['batch_normalization_33[0][0]']

conv2d_35 (Conv2D)              (None, 32, 32, 24)   5208        ['batch_normalization_33[0][0]']

batch_normalization_34 (BatchN  (None, 32, 32, 24)  96          ['conv2d_34[0][0]']
```

```
ormalization)

batch_normalization_35 (BatchN  (None, 32, 32, 24)  96          ['conv2d_35[0][0]']
ormalization)

concatenate_10 (Concatenate)   (None, 32, 32, 48)   0           ['batch_normalization_34[0][0]',
                                                                 'batch_normalization_35[0][0]']

max_pooling2d_8 (MaxPooling2D)  (None, 16, 16, 48)  0           ['concatenate_10[0][0]']

conv2d_36 (Conv2D)             (None, 16, 16, 48)   2352        ['max_pooling2d_8[0][0]']

batch_normalization_36 (BatchN  (None, 16, 16, 48)  192         ['conv2d_36[0][0]']
ormalization)

conv2d_37 (Conv2D)             (None, 16, 16, 48)   2352        ['batch_normalization_36[0][0]']

conv2d_38 (Conv2D)             (None, 16, 16, 48)   20784       ['batch_normalization_36[0][0]']

batch_normalization_37 (BatchN  (None, 16, 16, 48)  192         ['conv2d_37[0][0]']
ormalization)

batch_normalization_38 (BatchN  (None, 16, 16, 48)  192         ['conv2d_38[0][0]']
ormalization)

concatenate_11 (Concatenate)   (None, 16, 16, 96)   0           ['batch_normalization_37[0][0]',
                                                                 'batch_normalization_38[0][0]']

max_pooling2d_9 (MaxPooling2D)  (None, 8, 8, 96)    0           ['concatenate_11[0][0]']

conv2d_39 (Conv2D)             (None, 8, 8, 64)     6208        ['max_pooling2d_9[0][0]']

batch_normalization_39 (BatchN  (None, 8, 8, 64)    256         ['conv2d_39[0][0]']
ormalization)

conv2d_40 (Conv2D)             (None, 8, 8, 64)     4160        ['batch_normalization_39[0][0]']

conv2d_41 (Conv2D)             (None, 8, 8, 64)     36928       ['batch_normalization_39[0][0]']

batch_normalization_40 (BatchN  (None, 8, 8, 64)    256         ['conv2d_40[0][0]']
ormalization)

batch_normalization_41 (BatchN  (None, 8, 8, 64)    256         ['conv2d_41[0][0]']
ormalization)

concatenate_12 (Concatenate)   (None, 8, 8, 128)    0           ['batch_normalization_40[0][0]',
                                                                 'batch_normalization_41[0][0]']

max_pooling2d_10 (MaxPooling2D  (None, 4, 4, 128)   0           ['concatenate_12[0][0]']
)

conv2d_42 (Conv2D)             (None, 4, 4, 48)     6192        ['max_pooling2d_10[0][0]']

batch_normalization_42 (BatchN  (None, 4, 4, 48)    192         ['conv2d_42[0][0]']
ormalization)

conv2d_43 (Conv2D)             (None, 4, 4, 48)     2352        ['batch_normalization_42[0][0]']

conv2d_44 (Conv2D)             (None, 4, 4, 48)     20784       ['batch_normalization_42[0][0]']

batch_normalization_43 (BatchN  (None, 4, 4, 48)    192         ['conv2d_43[0][0]']
ormalization)

batch_normalization_44 (BatchN  (None, 4, 4, 48)    192         ['conv2d_44[0][0]']
ormalization)

concatenate_13 (Concatenate)   (None, 4, 4, 96)     0           ['batch_normalization_43[0][0]',
                                                                 'batch_normalization_44[0][0]']

max_pooling2d_11 (MaxPooling2D  (None, 2, 2, 96)    0           ['concatenate_13[0][0]']
)

conv2d_45 (Conv2D)             (None, 2, 2, 24)     2328        ['max_pooling2d_11[0][0]']

batch_normalization_45 (BatchN  (None, 2, 2, 24)    96          ['conv2d_45[0][0]']
ormalization)

conv2d_46 (Conv2D)             (None, 2, 2, 24)     600         ['batch_normalization_45[0][0]']

conv2d_47 (Conv2D)             (None, 2, 2, 24)     5208        ['batch_normalization_45[0][0]']
```

$Layer_i$	Parameter Count	% $Layer_i$ Contribution w.r.t Total
0	896	0.74
1	5208	4.31
2	20784	**17.20**
3	36928	**30.55**
4	20784	**17.20**
5	5208	4.31
Total Layers	120858	100.00

TABLE 7.3: SqueezeNet Parameters Per Layer

```
batch_normalization_46 (BatchN  (None, 2, 2, 24)    96          ['conv2d_46[0][0]']
ormalization)

batch_normalization_47 (BatchN  (None, 2, 2, 24)    96          ['conv2d_47[0][0]']
ormalization)

concatenate_14 (Concatenate)    (None, 2, 2, 48)    0           ['batch_normalization_46[0][0]',
                                                                 'batch_normalization_47[0][0]']

global_average_pooling2d_2 (Gl  (None, 48)          0           ['concatenate_14[0][0]']
obalAveragePooling2D)

dense_2 (Dense)                 (None, 10)          490         ['global_average_pooling2d_2[0][0
                                                                ]']

==================================================================================================
Total params: 120,858
Trainable params: 119,546
Non-trainable params: 1,312
__________________________________________________________________________________________________
```

Given that our optimization procedure exclusively targets classic Convolutional Layers, as indicated in Table 7.3, our evaluation will focus solely on these layers.

It is important to note that while only 6 layers are considered for optimization they account for a substantial 74.31% of the total NN parameters. Similar to the previous approaches we will target the primary parameter contributors which in this case are layers 2,3 and 4.

7.3 Baseline Characteristics for the Examined Models

Table 7.4 reports the characteristics (e.g., baseline accuracy, latency) for our baseline models deployed on an STM32-Nucleo-U575ZI-Q board with a clock frequency at 160MHz. The respective models were trained on CIFAR- 10 dataset, using TensorFlow with a subsequent application of 8-bit post-training quantization. The inputs have 32x32 resolution and are normalized to [0,1] in 32-bit representation and training/testing uses a random 70%/30% split. As shown in Table 7.4, the latency for a small model with less than 5M parameters is more than 80ms, while it is noteworthy that 86%, on average, of the available flash memory remains unexploited.

CNN	Acc	# MAC Ops	Latency (ms)	Flash Usage (%)	RAM (KB)
SqueezeNet	75.0	14.2 M	551.5	16	298.64
AlexNet	71.9	11.5 M	179.9	13	212.16
LeNet	72.0	4.6 M	82.8	12	183.5

TABLE 7.4: Evaluation of our baseline CIFAR-10 SqueezeNet, AlexNet and LeNet on an STM32-Nucleo fitting 2000KB ROM and 768KB RAM

7.4 Accuracy Agnostic Skipping

Now that we've identified the primary latency contributors, we can proceed with a straightforward experiment to assess the potential of our significance-aware computation skipping optimization. For this experiment, we've chosen AlexNet, the most promising neural network (NN) candidate. Our focus will be on optimizing Convolutional Layers 1 to 4 using our accuracy-agnostic computation skipping technique, facilitated by our layer-based unpacking method. It's worth mentioning that, for this specific experiment, we've enabled ICache and implemented unpacking using the DSP option.

Fig. 7.3 shows how the total cycles required for AlexNet inference fluctuate when we gradually reducing computations in MAC units increasingly. To generate this figure and profile the baseline performance, cycle counters are used to profile parts of the C code, while percentages of computations can be easily skipped due to our unpacking approach. Notably, each layer contributes distinctively to the total number of cycles (28.8 M). For instance, when skipping 20% MAC operations from a single layer, the respective gains compared to the total execution time vary from 2% (layer 1) up to 6% (layer 3). This further highlights how the profiling information obtained from our offline analysis can identify performance-critical areas and effectively eliminate unnecessary computations within each channel and layer.

7.5 Design Space Constraint through Histograms

After initially narrowing down the design space through an overview of the models' parameters and identifying the potential latency improvements of the framework, the next step in our evaluation process, as described in Section 6.4, involves the quest for the optimal design space for the neural network (NN) in use. This is achieved by generating and examining histograms that correspond to the calculated mean values for each layer and each model.

7.5.1 LeNet Layer Distribution

The distributions of the layers that will be modified by our framework for LeNet, namely layer 1 and 2 can be seen in Fig. 7.4. The mean values for this configuration have been calculated

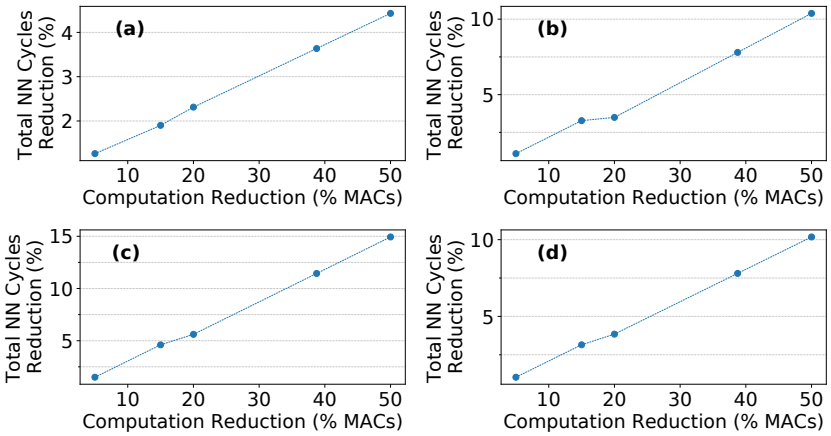
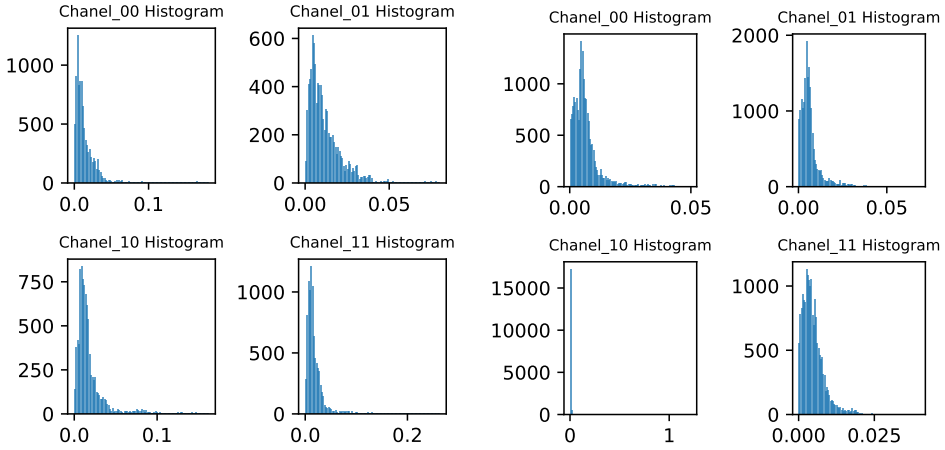


FIGURE 7.3: Reduction in cycles w.r.t. the total cycles required for AlexNet inference versus computation reduction (in MAC units) for the 1st(a), 2nd(b), 3rd(c), and 4th(d) convolution layer. Dots represent values obtained from an STM32-U575ZI-Q execution, while the remaining values are interpolated.



(A) LeNet Layer 1 Distribution (B) LeNet Layer 2 Distribution

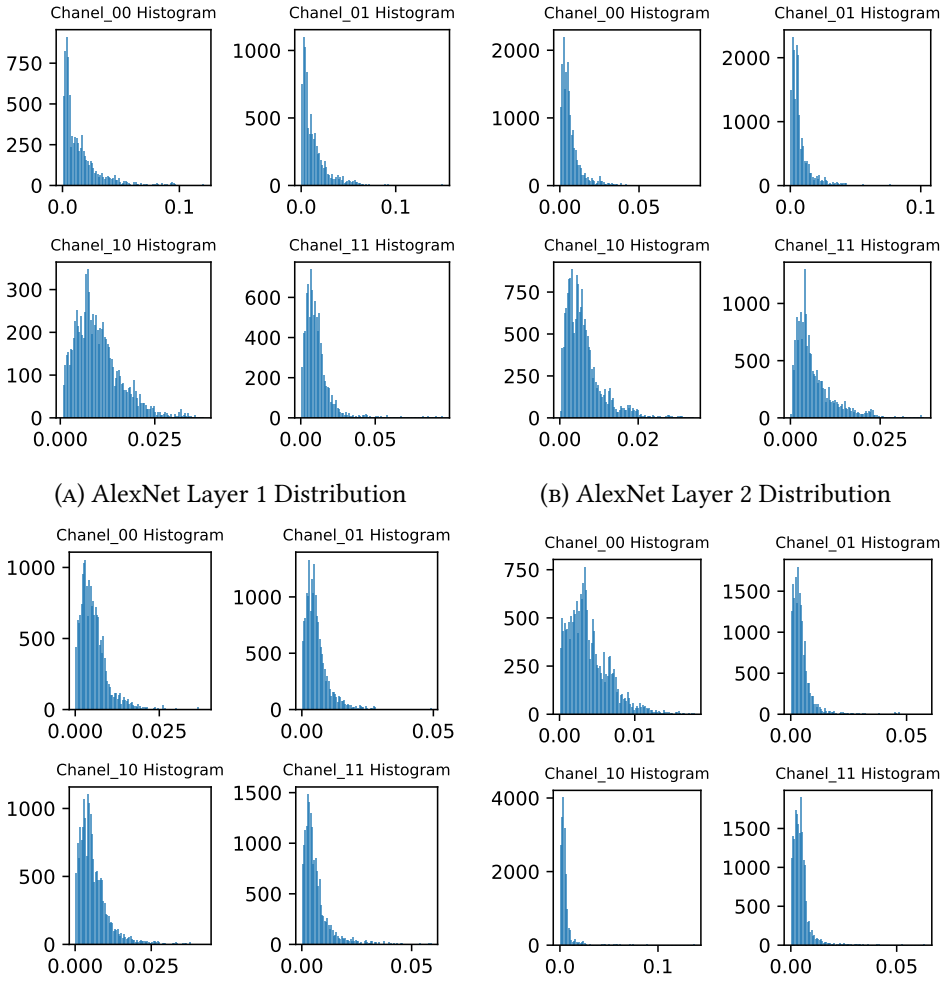
FIGURE 7.4: LeNet Layer 1 and Layer 2 Distributions

using 1000 samples of the dataset.

Through this analysis step a reasonable constraint of [0 to 0.009] can be initially placed upon the design space of the LeNet model. This is an initial constraint and ca be later amended if the DSE does not yield results according to the users constraints.

7.5.2 AlexNet Layer Distribution

Similarly, we can analyze layers 1 to 4 of the AlexNet implementation for which the mean values have been calculated using 3000 samples of the dataset. As can be clearly seen in Fig 7.5 the design space can also be constrained to [0,0.009].



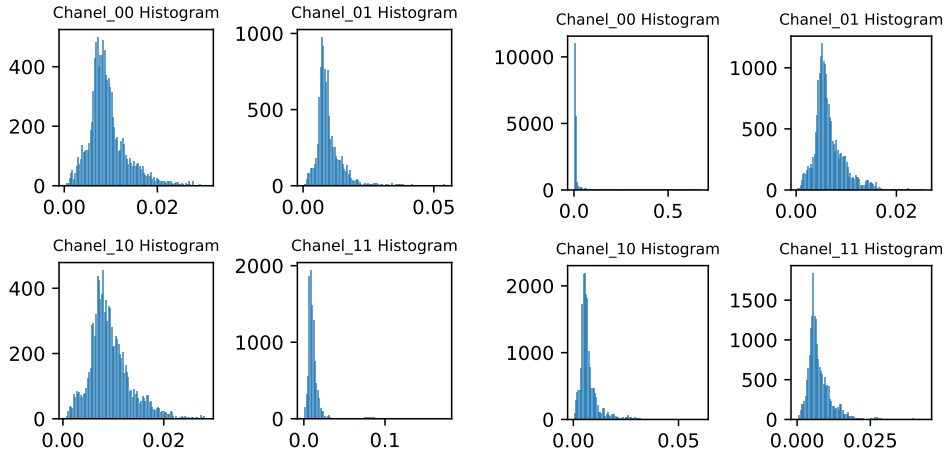
(A) AlexNet Layer 1 Distribution

(B) AlexNet Layer 2 Distribution

(C) AlexNet Layer 3 Distribution

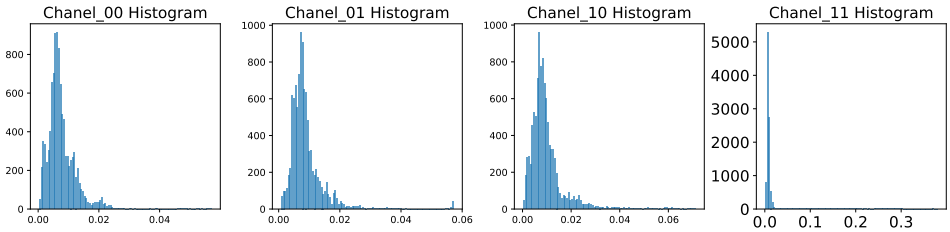
(D) AlexNet Layer 4 Distribution

FIGURE 7.5: AlexNet Layer 1 to Layer 4 Distributions



(A) SqueezeNet Layer 6 Distribution

(B) SqueezeNet Layer 9 Distribution



(C) SqueezeNet Layer 12 Distribution

FIGURE 7.6: SqueezeNet Layer 6, Layer 9 and Layer 12 Distributions

7.5.3 SqueezeNet Layer Distribution

The final distributions to be analyzed will be those concerning the SqueezeNet architecture. As seen in Fig. 7.6 the situation is different from the previous 2 architectures with more differences presenting themselves between the distributions of each channel in each layer. This characteristic will lead to a slightly different approach in the choice of the significance thresholds for this configuration.

7.6 Profiling Results

Finally, we perform an exhaustive DSE w.r.t. the values of ranging from [0, 0.009] with a step of 0.001. This explotration is performed offline and only once. Every approximate configuration, denoting which layers and computations are approximated, undergoes simulation to calculate the classification accuracy. Subsequently, a Pareto analysis is conducted to determine the Accuracy-Speedup Pareto Front. Note that in this work, we exclusively concentrate on the convolution layers, and therefore, the model's behavior, when considering the rest of the functions, becomes rather predictable. Consequently, the clock cycles reported by our counters during our simulations closely align with the actual model deployment. On average, DSE required less than 2 hours using 6 threads. The aforementioned execution times refer to an Intel-i7-8750H with 32GB RAM. Following the DSE analysis, we extract the suitable

approximate configuration based on the user's specified accuracy loss threshold and desired latency. Subsequently, our framework automatically generates the approximate code, which is then compiled and deployed to the MCU.

To ensure the robustness of our analysis, all configurations have undergone evaluation through inference on 1000 diverse samples from the CIFAR-10 dataset. This rigorous evaluation process will provide valuable insights into the performance characteristics of each NN architecture and guide further refinements in our model designs.

7.6.1 Accuracy-Latency Pareto Space

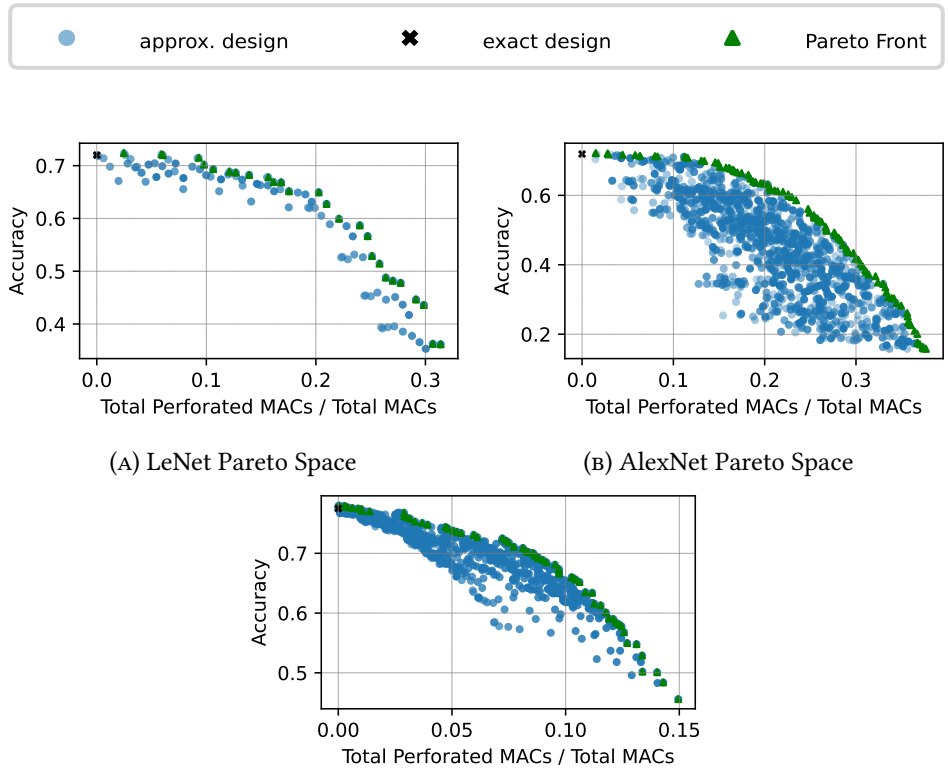
Fig. 7.7 presents the Accuracy-Latency Pareto space for the three examined CNNs. At this point, latency concerns only convolution layers and is reported as a normalized value w.r.t. the latency of our "unpacked only" optimization. The latter, although enables our framework to perform a low-level selective computation skipping, in some cases introduces a slightest latency overhead due to cache memory overload. In Fig. 7.7 the black 'x' is our exact baseline design. The blue dots on the graph correspond to approximate configurations. The green triangles, on the other hand, form the Pareto Front line. These configurations particularly represent the percentage of operations that are skipped and the indexes of these operations in the final generated code. To generate this figure, latency was computed based on our cycle counters, as deploying over a thousand designs on MCUs is impractical, even when automated. This is an initial analysis that assists in extracting approximate designs based on the specified accuracy loss threshold and subsequently deploying them on MCUs. Note that the number of the explored configuration/designs is model dependent. As aforementioned, the significance Si ranges from [0-0.009] for all CNNs, but the examined layers in which Si is explored, are also model dependent. In total we evaluated more than 100 approximate designs for LeNet and more than 1000 designs for AlexNet and SqueezeNet.

Overall, it is observed that approximate computing can effectively be employed to speedup the inference of the CNN models since all the approximate designs feature lower latency than the exact one (i.e., black 'x'). As shown, our "only skipping" approximation achieves on average 11% latency reduction for almost identical accuracy (<1%) with the exact baseline. In addition, for the first two models, it averagely delivers more than 15%, 20% speedup gains for less than 3% and 7% accuracy loss, respectively. In the case of SqueezeNet, the results appear to be less promising, possibly owing to its highly compressed architecture.

7.7 Deployment Results

7.7.1 Cache Enabled

In this section, we have decided to exclude SqueezeNet from our analysis. An initial assessment of the best-case accuracy-latency trade-off for a 10% accuracy threshold revealed only a



(A) LeNet Pareto Space

(B) AlexNet Pareto Space

(C) SqueezeNet Pareto Space

FIGURE 7.7: Pareto Spaces for the Examined NN Architectures

disappointing 1.4% improvement. Therefore, we conclude that further experimentation with SqueezeNet is unnecessary.

In Table 7.5 we report some important metrics of our framework. To generate this table we considered three conservative accuracy loss thresholds (i.e., 5%, 10% and 15%) and we report the latency, Top-1 accuracy, flash, and energy metrics for the latency-optimized approximate designs after deployment on the examined MCU. As aforementioned, due to the nature of target AI applications, such as real-time processing, a fast inference is one of the foremost requirements when targeting DNNs on MCUs and so prioritizing it over strict accuracy constraints is a typical procedure [12, 39, 40]. As depicted in Table 7.5, our cooperative approximation approach, which includes both code unpacking and significance-aware approximation, achieves an average speedup of 11% while incurring only a 3% accuracy loss compared to the exact baseline [2]. Moreover, these gains increase to 21% when accepting a 9% accuracy loss. In Table 7.5, we also provide a comparison of our models with the state-of-the-art homogeneous inference library, X-CUBE-AI. While X-CUBE-AI excels when our framework doesn't sacrifice accuracy, it's worth noting that for the more complex CNN of AlexNet, our approach outperforms X-CUBE-AI. Specifically, we achieve a speedup of 12% at the cost of 9% accuracy loss. In conclusion, our framework, despite a significant accuracy loss, can surpass even commercial tools like X-CUBE-AI, providing an accuracy-latency trade-off that was previously unattainable for optimized libraries like CMSIS.

	CMSIS-NN		X-CUBE-AI		Proposed (ours)					
Network	**LeNet**	**AlexNet**	**LeNet**	**AlexNet**	**LeNet 5%**	**LeNet 10%**	**LeNet 15%**	**AlexNet 5%**	**AlexNet 10%**	**AlexNet 15%**
Top-1 Accuracy (%)	**72.0**	**71.9**	72.0	71.9	68.7	65.0	62.8	68.4	64.8	62.6
Latency (ms)	82.8	179.9	**63.5**	150.7	76.4	72.0	70.1	153.9	**148.5**	**133.2**
Flash (KB)	239	267	154	178	659	591	585	954	902	999
#MAC Ops.	4.6M	11.5M	4.6M	11.5M	**4.2M**	**4.0M**	**3.9M**	**9.6M**	**9.5M**	**8.2M**
Energy (mJ)	2.73	5.94	**2.10**	4.97	2.52	2.38	2.31	5.08	**4.90**	**4.39**

TABLE 7.5: Comparison with state-of-the-art CMSIS [2] and X-CUBE-AI [9] with the Cache enabled for two CNNs deployed on an stm32u575zi-q board fitting 2MB Flash and 768KB RAM. Three accuracy loss thresholds have been considered.

	CMSIS-NN			Proposed (ours)								
Network	**LeNet**	**AlexNet**	**SqueezeNet**	**LeNet 1%**	**LeNet 5%**	**LeNet 10%**	**AlexNet 1%**	**AlexNet 5%**	**AlexNet 10%**	**SqueezeNet 1%**	**SqueezeNet 5%**	**SqueezeNet 10%**
Top-1 Acc (%)	**72.0**	**71.9**	**75**	71.5	68.7	65.0	71.2	69.3	64.8	75	72	71
Latency (ms)	172.9	407.7	972.9	114.5	106.2	100.9	249.8	227.5	218.8	855.3	852.1	849.8
Flash (KB)	232	261	313	1015	962	832	1690	1420	1370	1440	1380	1360
#MAC Ops.	4.6M	11.5M	14.2 M	**4.5 M**	**4.2M**	**4.0 M**	**11.3M**	**10.0 M**	**9.5M**	**14.0 M**	**13.9 M**	**13.8 M**
Energy (mJ)	6.74	15.90	37.94	**4.46**	**4.14**	**3.93**	**9.74**	**8.87**	**8.53**	**33.36**	**33.23**	**33.14**

TABLE 7.6: Comparison with state-of-the-art CMSIS [2] with the Cache disabled for three CNNs deployed on an stm32u575zi-q board fitting 2MB Flash and 768KB RAM. Three accuracy loss thresholds have been considered.

7.7.2 Cache Disabled

In Table 7.6, we present crucial metrics for our framework. To compile this table, we considered three conservative accuracy loss thresholds: 1%, 5%, and 10%. The table reports latency, Top-1 accuracy, flash usage, and energy metrics for latency-optimized approximate designs deployed on the examined MCU.

We selected lower accuracy loss thresholds for the cacheless implementation to address cache overhead stemming from increased code size. Thus, more aggressive approximation is unnecessary to mitigate this overhead.

As shown in Table 7.6, our cooperative approximation approach, which includes both code unpacking and significance-aware approximation, achieves an average speedup of 32% with only a 3% accuracy loss compared to the exact baseline [2]. Furthermore, these gains increase to 34% when accepting a 6% accuracy loss.

7.8 Comparison with Relevant Frameworks

Lastly, we undertake a qualitative evaluation, comparing our approximation framework with other state-of-the-art methodologies. In this case it is important to note that since these frameworks have been implemented using boards carrying some form of Cache the following reuslts have been meassured against our Cached Results Tab. 7.5. When comparing with TinyEngine [12], a direct evaluation becomes challenging due to its limited support for the

necessary layers in our studied CNNs. Additionally, its co-design methodology is primarily tailored for specific models. However, it's worth noting that in [12], they report a 10.5% reduction in latency compared to [9] when using their mcunet-vww1 model, which employs both pre-training and post-training optimizations. In contrast, for a pre-trained model with similar MACs, our framework achieves a 12% reduction in latency at the cost of a 9% Top-1 accuracy loss. Furthermore, when compared to CMix-NN [11] using a model with 13.8M MAC operations, our framework achieves a latency of 164ms on a 160MHz MCU. This means that, compared to CMix-NN [11], our framework achieves a remarkable 50% reduction in latency, with a negligible accuracy degradation of less than 1%. Finally, uTVM [13], an end-to-end ML compiler framework tailored for bare-metal MCUs, reports a 13% latency overhead compared to CMSIS when using a similar LeNet model architecture as examined in our work. For the same model, our approach outperforms uTVM, achieving an additional 26% speedup with an accuracy loss of less than 7%. This adjustment in inference latency holds the potential to enable more practical and realistic applications, paving the way for the implementation of deeper networks on tiny MCUs.

Chapter 8

Conclussion and Future Work

Despite their widespread adoption in the Tiny Machine Learning domain, energy-efficient microcontroller units face notable limitations. These restrictions primarily pertain to their limited computational capacity and tight energy budgets, and consequently, deploying complex deep learning models on such platforms becomes a challenging endeavor. In this work, to address this challenge, we introduce a cooperative approximation framework that combines the principles of approximate computing with software kernel optimizations.

Using a systematic method based on kernel computation skipping, our framework effectively eliminates operations that are considered unimportant for the model's inference process. This results in faster inference speeds, albeit with a slight reduction in accuracy. Furthermore, our approach enables efficient execution of deep neural network models on microcontroller unit (MCU) designs that lack cache memory. We achieve this by allowing selective layer-based code unpacking, which fully utilizes an MCU's flash memory and enables kernel-specific optimizations. In some cases, this approach can approach the performance level of MCUs with cache memory. The combination of flash memory utilization and the trade-off between accuracy and latency has the potential to open up new possibilities for AI applications and make it feasible to run more complex deep neural networks on small MCUs.

As a future project, we could consider expanding our support for additional Convolutional Layer types, such as Depth-wise convolution and Depth-wise Separable Convolution. This expansion would enable compatibility with a broader range of deep learning models, including architectures like MobileNet. It would also facilitate direct comparisons with well-known state-of-the-art frameworks like TinyEngine [12]. Additionally, there is an opportunity to explore and optimize the unpacking procedure to make it more cache-friendly. This optimization could help eliminate any potential overhead introduced during the execution, enhancing the overall efficiency of our framework.

Bibliography

[1] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2015.

[2] Liangzhen Lai, Naveen Suda, and Vikas Chandra. "CMSIS-NN: Efficient Neural Network Kernels for Arm Cortex-M CPUs". In: (Jan. 2018).

[3] P. Warden and D. Situnayake. *TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers*. O'Reilly Media, 2019. ISBN: 978-1-4920-5199-2. URL: https://books.google.gr/books?id=tH3EDwAAQBAJ.

[4] Vasileios Leon et al. *Approximate Computing Survey, Part I: Terminology and Software & Hardware Approximation Techniques*. July 20, 2023. arXiv: 2307.11124[cs]. URL: http://arxiv.org/abs/2307.11124 (visited on 09/22/2023).

[5] *M-Profile Architecture*. URL: https://developer.arm.com/Architectures/M-Profile%20Architecture (visited on 09/21/2023).

[6] Joseph Yiu. "Cortex-M for beginners". In: (2016).

[7] ARM. "Arm® Compiler toolchain Assembler Reference". In: (2013).

[8] Martín Abadi et al. "TensorFlow: A System for Large-Scale Machine Learning". In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. OSDI'16. USENIX Association, 2016, 265–283. ISBN: 9781931971331.

[9] STMicroelectronics. *X-Cube-AI: AI Expansion Pack for STM32CubeMX*. 2019. URL: https://www.st.com/en/embedded-software/x-cube-ai.html.

[10] *STM32CubeIDE - Integrated Development Environment for STM32 - STMicroelectronics*. URL: https://www.st.com/en/development-tools/stm32cubeide.html (visited on 09/20/2023).

[11] Alessandro Capotondi et al. "CMix-NN: Mixed Low-Precision CNN Library for Memory-Constrained Edge Devices". In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 67.5 (2020), pp. 871–875. DOI: 10.1109/TCSII.2020.2983648.

[12] Ji Lin et al. "MCUNet: Tiny Deep Learning on IoT Devices". In: *Proceedings of the 34th International Conference on Neural Information Processing Systems*. NIPS'20. Curran Associates Inc., 2020. ISBN: 9781713829546.

[13] Tianqi Chen et al. "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning". In: *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*. OSDI'18. 2018, 579–594. ISBN: 9781931971478.

[14] Muhammad Shafique et al. “TinyML: Current Progress, Research Challenges, and Future Roadmap”. In: *2021 58th ACM/IEEE Design Automation Conference (DAC)*. 2021, pp. 1303–1306. DOI: 10.1109/DAC18074.2021.9586232.

[15] Visal Rajapakse, Ishan Karunanayake, and Nadeem Ahmed. “Intelligence at the Extreme Edge: A Survey on Reformable TinyML”. In: *ACM Comput. Surv.* 55.13s (2023). DOI: 10.1145/3583683.

[16] Kunran Xu et al. “An Ultra-Low Power TinyML System for Real-Time Visual Processing at Edge”. In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 70.7 (2023), pp. 2640–2644. DOI: 10.1109/TCSII.2023.3239044.

[17] Zhenge Jia et al. “TinyML Design Contest for Life-Threatening Ventricular Arrhythmia Detection”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2023), pp. 1–1. DOI: 10.1109/TCAD.2023.3309744.

[18] Giorgos Armeniakos et al. “Hardware Approximate Techniques for Deep Neural Network Accelerators: A Survey”. In: *ACM Comput. Surv.* 55.4 (2022). DOI: 10.1145/3527156.

[19] Qianyun Lu and Boris Murmann. “Enhancing the Energy Efficiency and Robustness of TinyML Computer Vision Using Coarsely-Quantized Log-Gradient Input Images”. In: *ACM Trans. Embed. Comput. Syst.* (2023). ISSN: 1539-9087. DOI: 10.1145/3591466.

[20] Han Cai et al. “Enable Deep Learning on Mobile Devices: Methods, Systems, and Applications”. In: *ACM Trans. Des. Autom. Electron. Syst.* 27.3 (2022). DOI: 10.1145/3486618.

[21] Riade Benbaki et al. *Fast as CHITA: Neural Network Pruning with Combinatorial Optimization*. 2023. arXiv: 2302.14623 [cs.LG].

[22] Yuri Arbeletche et al. “MAxPy: A Framework for Bridging Approximate Computing Circuits to its Applications”. In: *IEEE Transactions on Circuits and Systems II: Express Briefs* (2023), pp. 1–1. DOI: 10.1109/TCSII.2023.3240897.

[23] Yihui He, Xiangyu Zhang, and Jian Sun. “Channel Pruning for Accelerating Very Deep Neural Networks”. In: *2017 IEEE International Conference on Computer Vision (ICCV)*. 2017, pp. 1398–1406. DOI: 10.1109/ICCV.2017.155.

[24] Hongfu Liu et al. “Tiny Machine Learning (Tiny-ML) for Efficient Channel Estimation and Signal Detection”. In: *IEEE Transactions on Vehicular Technology* 71.6 (2022), pp. 6795–6800. DOI: 10.1109/TVT.2022.3163786.

[25] Matteo Risso et al. “Lightweight Neural Architecture Search for Temporal Convolutional Networks at the Edge”. In: *IEEE Transactions on Computers* 72.3 (2023), pp. 744–758. DOI: 10.1109/TC.2022.3177955.

[26] Krishna Teja Chitty-Venkata and Arun K. Somani. "Neural Architecture Search Survey: A Hardware Perspective". In: *ACM Comput. Surv.* 55.4 (2022). DOI: 10.1145/3524500.

[27] ARM. *ARM Processors*. 2023. URL: https://developer.arm.com/Processors.

[28] "ARM Cortex-M7 Processor Technical Reference Manual". In: (2014).

[29] Thomas Lorenser. "The DSP capabilities of ARM® Cortex®-M4 and Cortex-M7 Processors". In: (2016).

[30] *STM32CubeMX - STM32Cube initialization code generator - STMicroelectronics*. URL: https://www.st.com/en/development-tools/stm32cubemx.html (visited on 09/21/2023).

[31] ARM Software. *CMSIS-NN: Efficient Neural Network Kernels*. https://github.com/ARM-software/CMSIS-NN. 2023.

[32] Shvetank Prakash et al. "CFU Playground: Full-Stack Open-Source Framework for Tiny Machine Learning (TinyML) Acceleration on FPGAs". In: *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2023, pp. 157–167. DOI: 10.1109/ISPASS57527.2023.00024.

[33] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. "Compiler transformations for high-performance computing". In: *ACM Computing Surveys* 26.4 (Dec. 1994), pp. 345–420. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/197405.197406. URL: https://dl.acm.org/doi/10.1145/197405.197406 (visited on 10/05/2023).

[34] ARM. "Arm® Compiler Reference Guide". In: (2021).

[35] Giorgos Armeniakos et al. "Co-Design of Approximate Multilayer Perceptron for Ultra-Resource Constrained Printed Circuits". In: *IEEE Transactions on Computers* 72.9 (2023), pp. 2717–2725. DOI: 10.1109/TC.2023.3251863.

[36] TensorFlow. *Convolutional Neural Network (CNN) | TensorFlow Core*. 2023. URL: https://github.com/tensorflow/docs/blob/master/site/en/tutorials/images/cnn.ipynb (visited on 2023).

[37] Google Cloud Platform. *Keras Flowers TPU Squeezenet Notebook*. 2023. URL: https://github.com/GoogleCloudPlatform/training-data-analyst/blob/master/courses/fast-and-lean-data-science/07_Keras_Flowers_TPU_squeezenet.ipynb (visited on 2023).

[38] Forrest N. Iandola et al. *SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size*. Nov. 4, 2016. DOI: 10.48550/arXiv.1602.07360. arXiv: 1602.07360[cs]. URL: http://arxiv.org/abs/1602.07360 (visited on 10/06/2023).

[39] Giorgos Armeniakos et al. “Model-to-Circuit Cross-Approximation For Printed Machine Learning Classifiers”. In: *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.* (2023).

[40] Giorgos Armeniakos et al. “Cross-Layer Approximation For Printed Machine Learning Circuits”. In: *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2022, pp. 190–195. DOI: 10.23919/DATE54114.2022.9774689.