Εθνικο Μετσοβιο Πολυτεχνειο

Σχολη Ηλεκτρολογων Μηχανικων και Μηχανικων Υπολογιστων

τομεας τεχνολογιας πληροφορικης και υπολογιστων

Εργαστηριο Μικροϋπολογιστων και Ψηφιακων Συστηματων

Decoupled Access-Execute and Dynamic Voltage/Frequency Scaling Optimization for Energy Efficient tinyML Deployments on STM32 MCUs

Μελέτη και υλοποίηση

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

της

Ελισάβετ-Λυδία Αλβανάκη

Επιβλέπων: Δημήτριος Σούντρης
Καθηγητής Ε.Μ.Π.

Αθήνα, Οκτώβριος 2023

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Μικροϋπολογιστών και Ψηφιακών Συστημάτων

Decoupled Access-Execute and Dynamic Voltage/Frequency Scaling Optimization for Energy Efficient tinyML Deployments on STM32 MCUs

Μελέτη και υλοποίηση

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

της

Ελισάβετ-Λυδία Αλβανάκη

Επιβλέπων: Δημήτριος Σούντρης
Καθηγητής Ε.Μ.Π.

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

.....	.....	.....
Δημήτριος Σούντρης	Παναγιώτης Τσανάκας	Σωτήριος Ξύδης
Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Επίκουρος Καθηγητής Ε.Μ.Π.

Αθήνα, Οκτώβριος 2023

.....

Αλβανακη Ελισαβετ-Λυδια
Διπλωματούχος Ηλεκτρολόγος Μηχανικός
και Μηχανικός Υπολογιστών Ε.Μ.Π.

στην οικογένεια μου

Περίληψη

Τα τελευταία χρόνια, παρατηρείται σημαντική ανάπτυξη στη χρήση εφαρμογών Μηχανικής Μάθησης (ML) στο Edge. Οι συσκευές Διαδικτύου των Πραγμάτων (IoT) και μικροελεγκτών (MCUs) έχουν γίνει ολοένα και πιο δημοφιλείς στις καθημερινές δραστηριότητες. Σε αυτή την διπλωματική, επικεντρωνόμαστε στην οικογένεια των STM32 MCUs. Έχουμε υλοποιήσει τη δυνατότητα Dynamic Voltage Frequency Scaling (DVFS) για τους ARM Cortex M MCUs και την έχουμε ενσωματώσει σε ένα σύστημα πολυκριτηριακής βελτιστοποίησης, σε συνδυασμό με μια βελτιστοποίηση κώδικα με τη μέθοδο Decoupled Access Execute, που χωρίζει την εκτέλεση σε memory-bound και compute-bound κομμάτια. Παρουσιάζουμε μια πρωτοποριακή μεθοδολογία για την εφαρμογή των CNN στην οικογένεια των STM32, εστιάζοντας στην βελτιστοποίηση της ενέργειας μέσω της αποτελεσματικής εξερεύνησης του design space των ιδιοτήτων του Decoupled Access-Execute και των ρυθμίσεων του ρολογιού. Αυτή η προσέγγιση εμπλουτίζεται με τη βελτιστοποίηση της κατανάλωσης ενέργειας μέσω της τεχνικής Dynamic Voltage and Frequency Scaling (DVFS) υπό διάφορους περιορισμούς και σχηματίζει ένα NP-complete πρόβλημα βελτιστοποίησης. Συγκρίνουμε την προσέγγισή μας με το state-of-the-art σύστημα TinyEngine, καθώς και το TinyEngine σε συνδυασμό με τις λειτουργίες εξοικονόμησης ενέργειας των MCUs της STM32. Τα αποτελέσματα δείχνουν ότι μπορούμε να επιτύχουμε μείωση της κατανάλωσης ενέργειας έως και 25,2% για διάφορα επίπεδα ποιότητας υπηρεσίας (QoS).

Λέξεις Κλειδιά — Νευρωνικά Δίκτυα, Edge Computing, Dynamic Voltage Frequency Scaling, Decoupled Access-Execute

Abstract

Over the last years the rapid growth Machine Learning (ML) inference applications deployed on the Edge is rapidly increasing. Recent Internet of Things (IoT) devices and microcontrollers (MCUs), become more and more mainstream in everyday activities. In this work we focus on the family of STM32 MCUs. A DVFS capability for ARM Cortex M MCUs is implemented and integrated within a state-of-the-art inference engine, along with a Decoupled Access Execute code optimization. A novel methodology is proposed for CNN deployment on the STM32 family, focusing on power optimization through effective clocking exploration and configuration and decoupled access-execute convolution kernel execution. This approach is enhanced with optimization of the power consumption through Dynamic Voltage and Frequency Scaling (DVFS) under various latency constraints, composing an NP-complete optimization problem. We compare our approach against the state-of-the-art TinyEngine inference engine, as well as TinyEngine coupled with power-saving modes of the STM32 MCUs, indicating that we can achieve up to 25.2% less energy consumption for varying QoS levels.

Keywords — Neural Networks, Edge Computing, Dynamic Voltage Frequency Scaling, Decoupled Access-Execute

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή μου, Δημήτριο Σούντρη καθώς και τον καθηγητή Σωτήρη Ξύδη για την πολύτιμη καθοδήγηση τους κατά την διάρκεια της διπλωματικής μου. Θα ήθελα επίσης να ευχαριστήσω τους Δρ. Δημοσθένη Μασούρο και τον υποψήφιο διδάκτορα Μανώλη Κατσαραγάκη για την υποστήριξη και την προθυμία τους να βοηθήσουν όποτε ήταν απαραίτητο. Ευχαριστώ την οικογένεια μου, για την στήριξη τους καθ' όλη τη διάρκεια των σπουδών μου. Τέλος ευχαριστώ τους φίλους μου, για την υπομονή και την συμπαράσταση τους κατά την διάρκεια των φοιτητικών μου χρόνων.

Ελισάβετ-Λυδία Αλβανάκη
Οκτώβριος 2023

Contents

Figure List

Table List

Chapter 1

Εκτεταμένη Ελληνική Περίληψη

1.1 Εισαγωγή

Τα τελευταία χρόνια, το Edge AI [1] έχει εμφανιστεί ως ένα καινοτόμο υπολογιστικό παράδειγμα. Το Edge AI αναφέρεται στην υλοποίηση πολύπλοκων μοντέλων Βαθέων Νευρωνικών Δικτύων (DNN) απευθείας σε συσκευές στο Edge, όπως αισθητήρες, συσκευές Internet of Things (IoT) και άλλα ενσωματωμένα συστήματα. Επιπλέον, η αυξανόμενη ζήτηση για εφαρμογές ψηφιακής επεξεργασίας σήματος στο Edge, όπως η ανάλυση βίντεο [2], η αναγνώριση σε κινητές συσκευές [3] και άλλες εφαρμογές, έχει καθιερώσει τα Συνελικτικά Νευρωνικά Δίκτυα(CNNs) ως την κύρια τεχνολογία για την αποδοτική εκτέλεση τέτοιων εφαρμογών. Πολλές εταιρείες έχουν ανταποκριθεί σε αυτήν την μετατόπιση παραδίδοντας επιχειρηματικές λύσεις σχεδιασμένες για τη βελτιστοποίηση και την υλοποίηση μοντέλων CNN στο Edge. Αυτές οι λύσεις περιλαμβάνουν software stacks(π.χ., Amazon SageMaker Edge [4]) καθώς και ειδικά, εξειδικευμένα hardware devices (π.χ., το ASIC Edge TPU της Google [5]).

Ένα σημαντικό μέρος της τρέχουσας έρευνας επικεντρώνεται στην εξερεύνηση του πολύπλοκου σχεδιασμού DNN/CNN και της βελτιστοποίησης του dataflow scheduling [6][7] για τη βελτιστοποίηση της ενέργειας. Παρά την περίτεχνη υλοποίηση τους, αυτές οι λύσεις συχνά υποθέτουν την ύπαρξη υλικών επιταχυντών ή συσκευών στο Edge που υποστηρίζονται από λειτουργικά συστήματα και δίκτυα. Ωστόσο, καθώς η υπολογιστική συνέχεια εκτείνεται στο περιθώριο των δικτύων, συστήματα με περιορισμένους πόρους, όπως οι μικροελεγκτές (MCUs), εισάγονται στο οικοσύστημα. Αντίθετα από τους πιο ικανές, μεγαλύτερες συσκευές, τα MCUs συνήθως χρησιμοποιούν εκτέλεση εφαρμογής bare-metal, χωρίς λειτουργικό σύστημα, με προσαρμοσμένα ή ελαφρια runtime συστήματα ενώ παράλληλα βρίσκονται αποσυνδεδεμένα από το δίκτυο. Αυτή η μετατόπιση προς το Edge έχει δημιουργήσει τον όρο "tinyML" [8]. Το tinyML αφορά τη βελτιστοποίηση των μοντέλων DNN/CNN για να λειτουργούν άνετα σε αυτά τα MCUs bare-metal, επεκτείνοντας έτσι τα όρια του πού μπορεί να εφαρμοστεί η μηχανική μάθηση.

Προκειμένου να μπορέσουν να ανταποκριθούν στους αυξημένους περιορισμούς μνήμης και υπολογιστικής ισχύος, τα μοντέλα tinyML σχεδιάζονται για να προσφέρουν αποδοτικές δυνατότητες πρόβλεψης, καθιστώντας δυνατή την ενσωμάτωση τεχνητής νοημοσύνης σε συσκευές που προηγουμένως ήταν αποκλεισμένες από τον χώρο της Μηχανικής Μάθησης (ML).

Η υλοποίηση των CNNs σε MCUs επιφέρει δύο βασικές προκλήσεις. Πρώτον, η περιορισμένη μνήμη των MCUs προκαλεί δυσκολίες τόσο στην αποθήκευση όσο και στην εκτέλεση των μοντέλων CNN. Αυτό γίνεται ακόμη πιο σύνθετο, καθώς οι αναδυόμενες αρχιτεκτονικές CNN τείνουν να γίνονται όλο και πιο περίπλοκες, στοχεύοντας στην παροχή μεγαλύτερης ακρίβειας και/ή υποστήριξης πιο μεγάλων εφαρμογών [9]. Δεύτερον, δεδομένου ότι τα MCUs συχνά ενσωματώνονται σε συσκευές Edge που λειτουργούν με μπαταρίες, η διατήρηση των ενεργειακών πόρων γίνεται κρίσιμη, καθώς η εκτέλεση μεγάλων σε μέγεθος και υπολογιστικά ακριβών DNNs μπορεί να εξαντλήσει γρήγορα την μπαταρία, ιδιαίτερα σε συσκευές με αυξημένες απαιτήσεις λειτουργίας.

Ενώ οι περισσότερες προσπάθειες για τη βελτίωση της ενεργειακής αποδοτικότητας έχουν επικεντρωθεί στη μείωση της καθυστέρησης διατηρώντας την ισχύ σταθερή (δηλαδή pruning συνδέσεων, βελτιστοποίηση kernels), η εκμετάλλευση των κυκλωμάτων ρολογιού για την ενεργειακή αποδοτικότητα σε συσκευές με περιορισμένους πόρους συχνά παραμελείται.

Γνωστοί κατασκευαστές, όπως η ARM και η Atmel, δεν προσφέρουν δυναμικές λειτουργίες διαχείρισης ενέργειας για τα λιγότερο υπολογιστικά ισχυρά MCUs τους (οι τυπικές λειτουργίες εξοικονόμησης ενέργειας σε τέτοια MCUs αποσκοπούν στην αναστολή του πυρήνα CPU και, συνεπώς, στην αναστολή της λειτουργίας). Σε αυτήν τη διπλωματική, προτείνουμε ένα σχήμα δυναμικής διαχείρισης ενέργειας για τα χαμηλής κλάσης ARM Cortex M MCUs. Στη συνέχεια, βελτιστοποιούμε αυτό το σχήμα για εφαρμογές tinyML, ενσωματώνοντας Decoupled Access-Execute μετασχηματισμούς στον πηγαίο κώδικα του νευρωνικού δικτύου. Τέλος, παρουσιάζουμε έναν optimizer, που εξερευνά τον χώρο σχεδίασης για να βρει τον βέλτιστο συνδυασμό απο ρυθμίσεις ρολογιού και μετασχηματισμους κώδικα, προκειμένου να δημιουργήσει τις βέλτιστες ρυθμίσεις DVFS για έναν δεδομένο περιορισμό στην καθυστέρηση.

1.2 Σχετική Βιβλιογραφία

Εστιάζοντας στην πρόβλεψη με την βοήθεια CNNs, προηγούμενες έρευνες [10], [11] και ευρέως χρησιμοποιούμενα πλαίσια (π.χ., TFLite Micro [12], Microsoft's NNI [13], ARM's CMSIS-NN [14], κλπ.) προσφέρουν τεχνικές βελτιστοποίησης που μπορούν να δυναμώσουν την υλοποίηση μικρών μοντέλων CNN σε περιορισμένους πόρους MCUs. Αυτές οι τεχνικές περιλαμβάνουν συνήθως στατικές, μοντέλο-εξατομικευμένες, μετα-εκπαίδευσης βελτιστοποιήσεις, όπως το mixed-precision [15], weight pruning στο μοντέλο [16], [17] και quantization [18], [19], καθώς και το downsampling [20] και μεθοδολογίες αναζήτησης της αρχιτεκτονικής (NAS) [8], [21]. Άλλες διερευνήσεις εξετάζουν βελτιστοποιήσεις στο scheduling και στον πηγαίο κώδικα είτε για να επιτρέψουν αποτελεσματική αντιστοίχιση δεδομένων και υπολογισμού στο υποκείμενο υλικό [22]–[24] είτε για να διαχωρίσουν τα CNN layers σε ανεξάρτητες εργασίες που μπορούν να απεκφορτωθούν και να εκτελεστούν παράλληλα [25].

Η υλοποίηση των CNNs σε συσκευές με περιορισμένους πόρους στο Edge αυξάνει την ανάγκη για ενεργειακά αποδοτικές υλοποιήσεις που χρησιμοποιούν τεχνικές δυναμικής διαχείρισης κατανάλωσης ενέργειας. Πολλές έρευνες εστιάζουν στη χρήση του DVFS σε CPUs και GPUs [26] [27]. Συχνά, αυτή η έρευνα συνδυάζεται με μεθοδολογίες Decoupled Access-Execute, με στόχο την ελαχιστοποίηση των χρονικών επιβαρύνσεων που προκαλούνται από την αλλαγή του ρολογιού [28]. Τέτοιες μεθοδολογίες συχνά υλοποιούν λύσεις αυτοματοποίησης βασισμένες στον μεταγλωττιστή, χρησιμοποιώντας μεταγλωττιστές για να κάνουν Decoupled Access-Execute με βάση το workload profiling[29].

Πολλές έρευνες στον τομέα του design automation επικεντρώνονται στον συνδυασμό του DVFS με τα νευρωνικά δίκτυα μέσω της αναζήτησης της αρχιτεκτονικής τους. Ενώ τέτοια ερευνητικά έργα έχουν διαχειριστεί αποτελεσματικά την απομόνωση περιοχών του κώδικα όπου μπορεί να εφαρμοστεί το DVFS, πολλές από αυτές επικεντρώνονται στα δυναμικά νευρωνικά δίκτυα, όπου το DVFS μπορεί να εφαρμοστεί χωρίς σημαντικούς μετασχηματισμούς [30]. Αυτές οι δημοσιεύσεις έχουν επίσης επικεντρωθεί στην αναζήτηση της αρχιτεκτονικής του νευρωνικού δικτύου για συσκευές που δεν έχουν πολύ περιορισμένους πόρους, σε αντίθεση με τις υλοποιήσεις του tinyML που είναι ο στόχος αυτής της διπλωματικής εργασίας.

Παρόλο που έχουν ερευνηθεί πολλές βελτιστοποιήσεις για την υλοποίηση των DNN σε MCUs, επικεντρώνονται κυρίως στα δομικά χαρακτηριστικά των μοντέλων, όπως τα βάρη, οι αριθμητικές πράξεις, η αρχιτεκτονική κλπ., ενώ δίνεται λίγη προσοχή στις βελτιστοποιήσεις κατά τη διάρκεια του runtime.

Στο πλαίσιο της βελτιστοποίησης της κατανάλωσης ενέργειας και/ή της ενεργειακής απόδοσης, το Dynamic Voltage Frequency Scaling(DVFS) [31] αποτελεί ένα αποτελεσματικό μέσο για την εξισορρόπηση ανάμεσα στην απόδοση και την κατανάλωση ενέργειας, ρυθμίζοντας σωστά τη συχνότητα του ρολογιού του MCU σύμφωνα με τις υπολογιστικές απαιτήσεις του μοντέλου.

Ωστόσο, η προσαρμογή του DVFS στα συγκεκριμένα χαρακτηριστικά των DNN, καθώς και η πρακτική εφαρμογή της, δεν είναι απλή, καθώς μπορεί να επιφέρει επιπλέον χρονικές καθυστερήσεις λόγω των

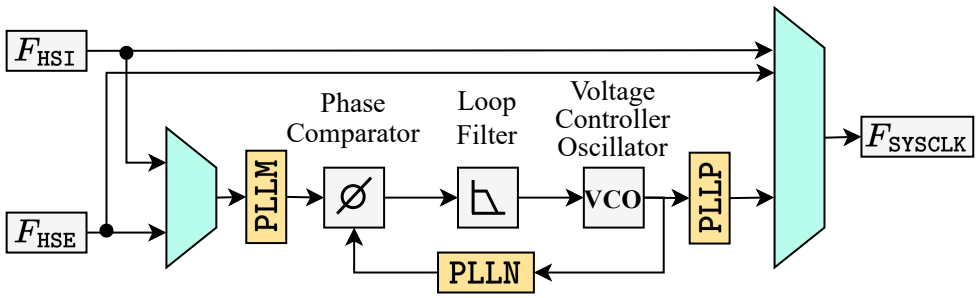

Figure 1.3.1: Απλοποιημένο διάγραμμα κυκλώματος για το clock configuration συναρτήσει των παραμέτρων HSE και PLL.

μεταβάσεων απο την μια συχνότητα στην άλλη [32] και αυξημένη διαρροή ενέργειας λόγω των μακρύτερων χρόνων εκτέλεσης [33].

1.3 Προτεινόμενη μεθοδολογία

Το σύστημα ρολογιών των μικροελεγκτών STM32 είναι διαχειρίσιμο από το Reset and Clock Control (RCC) peripheral. Το RCC παρέχει μια ευρεία γκάμα ρολογιών και πηγών ρολογιού που καλύπτουν διάφορες απαιτήσεις του συστήματος, όπως ρολόγια για τα περιφερειακά και την υποστήριξη πρωτοκόλλων επικοινωνίας [34].

Σε αυτήν την εργασία επικεντρωνόμαστε σε συγκεκριμένα ρολόγια και ρυθμίσεις, που καθορίζουν τη συχνότητα του ρολογιού του συστήματος (SYSCLK) του MCU, το οποίο είναι υπεύθυνο για τον χρονισμό του πυρήνα του CPU, της μνήμης και ορισμένων περιφερειακών μονάδων. Το Σχήμα 1.3.1 απεικονίζει το απλοποιημένο διάγραμμα κυκλώματος του πώς καθορίζεται το SYSCLK.

- **Ρολόι High-Speed Internal (HSI):** Το HSI ρολόι αποτελεί έναν εσωτερικό oscillator μέσα στον μικροελεγκτή. Το HSI λειτουργεί στα 16MHz. Το SYSCLK μπορεί να δημιουργηθεί είτε χρησιμοποιώντας απευθείας το HSI είτε μέσω του PLL χρησιμοποιώντας το HSI ως input source.
- **Ρολόι High-Speed External (HSE):** Το ρολόι HSE είναι ένας εξωτερικός oscillator. Το HSE μπορεί να προγραμματιστεί ώστε να λειτουργεί σε διαφορετικές συχνότητες, με το STM32F7 να υποστηρίζει συχνότητες μέχρι 50MHz. Όπως και με το HSI, το SYSCLK μπορεί να δημιουργηθεί είτε απο το HSE είτε μέσω του PLL όταν το HSE χρησιμοποιείται ως πηγή εισόδου.
- **Phase-Locked Loop (PLL):** Το PLL είναι ένα hardware module που επιτρέπει τον πολλαπλασιασμό της συχνότητας της επιλεγμένης πηγής εισόδου (HSI ή HSE) κατά έναν προγραμματιζόμενο παράγοντα, ώστε να δημιουργήσει μια υψηλότερη συχνότητα εξόδου. Όπως φαίνεται στο Σχήμα 1.3.1, ο παράγοντας αυτός καθορίζεται απο διαιρέτες εισόδου και εξόδου εντός του PLL κυκλώματος, οι οποίοι απλοποιούν την σχεδίαση του PLL και την επίτευξη ευστάθειας για μεγαλύτερο εύρος συχνοτήτων. Η συχνότητα εξόδου του PLL δίνεται απο την παρακάτω εξίσωση:

$$F_{\texttt{SYSCLK}} = F_{\{\texttt{HSE},\texttt{HSI}\}} * \frac{\texttt{PLLN}}{\texttt{PLLM} * \texttt{PLLP}} \tag{1.3.1}$$

όπου PLLM είναι ο διαιρέτης της συχνότητας εισόδου πριν φτάσει στον Voltage Controlled Oscillator του PLL; Το PLLN είναι ο παράγοντας πολλαπλασιασμού για τη συχνότητα εισόδου του VCO για να καθοριστεί η συχνότητα εξόδου του VCO, η οποία παρέχεται ως ανάδραση στον Phase Comparator, με στόχο να παρέχει συγχρονισμό μεταξύ εισόδου και εξόδου. Το loop filter χρησιμοποιείται για να εξασφαλίσει την σταθερότητα του συστήματος και να μειώσει τις διακυμάνσεις κατά την εκκίνηση του ρολογιού. Τέλος, το PLLP καθορίζει τον παράγοντα διαίρεσης για την απόκτηση της τελικής συχνότητας του SYSCLK.

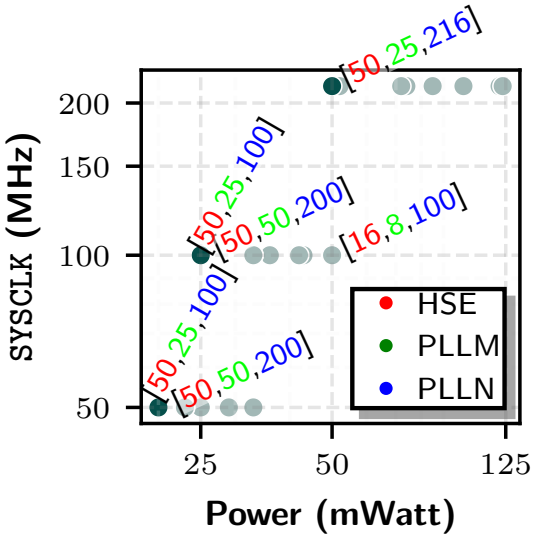

Figure 1.3.2: Συχνότητα ρολογιού και Ισχύς για διαφορετικά HSE, PLLM και PLLN configurations

Η συχνότητα εξόδου του ρολογιού του συστήματος, F_{SYSCLK}, μπορεί να επιτευχθεί με διάφορους τρόπους (π.χ. απευθείας μέσω των πηγών ρολογιού HSI/HSE ή μέσω του κυκλώματος PLL). Οι συσκευές STM32 παρέχουν την επιλογή της πηγής εισόδου του ρολογιού με πολυπλέκτη, καθώς και της πηγής εισόδου του PLL. Να σημειωθεί ότι κάθε αυτή επιλογή επηρεάζει κρίσιμες μετρικές του συστήματος για το DVFS, όπως την κατανάλωση ισχύος και τον χρόνο εκκίνησης του ρολογιού. Κάθε μια από τις επιλογές αναλύονται και τα αποτελέσματά τους εξετάζονται στις ακόλουθες ενότητες. Έχει εδραιωθεί ότι η επιλογή των πηγών του ρολογιού εισόδου τόσο για το ρολόι εξόδου όσο και για το PLL συνεπάγεται σημαντικά tradeoffs. Για να εξετάσουμε την επίδραση κάθε εναλλακτικής επιλογής στην λειτουργική απόδοση και την κατανάλωση ενέργειας του MCU, αναπτύξαμε και εκτελέσαμε ένα εξειδικευμένο microbenchmark, σχεδιασμένο να εκτελεί επαναληπτικές πράξεις πρόσθεσης μέσα σε ένα βρόγχο. Επικεντρωνόμαστε ειδικά στις παραμέτρους HSE και PLL. Η πηγή ρολογιού HSI έχει υψηλότερη κατανάλωση ισχύος σε σύγκριση με το HSE και είναι επίσης επιρρεπής σε απόκλιση και jitter, προσφέροντας λιγότερη σταθερότητα και ακρίβεια. Επίσης, έχει σημειωθεί ότι έχει μεγαλύτερο χρόνο εκκίνησης από το ρολόι υψηλής ταχύτητας. Συνεπώς, θα ήταν υποβέλτιστο όταν επιλέγεται ως πηγή ρολογιού. Επιπλέον, η πηγή ρολογιού HSI λειτουργεί σε σταθερή συχνότητα (συνήθως 8 ή 16MHz), κάτι που την καθιστά λιγότερο ευέλικτη για το DVFS. Τέλος, η επιλογή του HSI ως ενδιάμεσου ρολογιού κατά την εκτέλεση μεταβάσεων από PLL σε PLL θα ήταν επίσης υποβέλτιστη, καθώς η συχνότητα των 16MHz οδηγεί σε ανεπιθύμητη αδράνεια κατά τη διάρκεια της αλλαγής του ρολογιού.

Ορίσαμε την τιμή του PLLP ίση με 2, που είναι η ελάχιστη δυνατή τιμή για το διαιρέτη, διότι για την ίδια τιμή του F_{SYSCLK}, η επιλογή μιας υψηλότερης τιμής του PLLP οδηγεί σε υψηλότερη απαιτούμενη συχνότητα του VCO και, συνεπώς, υψηλότερη κατανάλωση ισχύος (όπως προκύπτει από το Σχήμα 1.3.1 και την Εξίσωση 5.2.1). Αξίζει να σημειωθεί ότι η ορισμένη τιμή του PLLP στην ελάχιστη συμβατή τιμή για την επιθυμητή συχνότητα εξόδου είναι προτιμότερη, καθώς η χρήση ενός διαιρέτη μετά το κύκλωμα PLL ουσιαστικά "σπαταλά" την ενέργεια του PLL, αφού οι προηγούμενες παράμετροι θα έχουν οριστεί σε υψηλότερες τιμές για τη δημιουργία υψηλότερης συχνότητας ως είσοδο του PLLP. **Κατανάλωση ισχύος σε ισο-συχνοτικές ρυθμίσεις:** Το Σχήμα 1.3.2 απεικονίζει την επίδραση διαφόρων ρυθμίσεων HSE, PLLM και PLLN στην κατανάλωση ισχύος της πλακέτας για διάφορες συχνότητες SYSCLK. Από το Σχήμα 1.3.2 προκύπτουν δύο κύριες παρατηρήσεις: (i) η ίδια συχνότητα εξόδου μπορεί να δημιουργηθεί μέσω διαφορετικών συνδυασμών HSE, PLLM και PLLN, ωστόσο (ii) η επιλεγμένη ρύθμιση επηρεάζει ισχυρά την κατανάλωση ισχύος στο STM32 MCU. Συγκεκριμένα, η τιμή της εισόδου HSE έχει αντίστροφη σχέση με την κατανάλωση ισχύος, εφόσον η συχνότητα εξόδου είναι σταθερή και ταυτόσημη σε και τις δύο ρυθμίσεις. Επιπλέον, σύμφωνα με το σύνολο δεδομένων του STM32, η τιμή του PLLN θα πρέπει να ελαχιστοποιηθεί προκειμένου να επιτευχθεί η ελάχιστη τάση του VCO. Ωστόσο, η διαφορά φαίνεται να

είναι μικρή όσον αφορά την κατανάλωση ισχύος 1.3.2. Να σημειωθεί ότι παρά τη μικρή διακύμανση στην κατανάλωση ισχύος, οι χαμηλότερες τιμές του PLLN οδηγούν επίσης σε γρηγορότερο χρόνο εκκίνησης του ρολογιού, καθώς ο χρόνος κλειδώματος του PLL μειώνεται [34]. Επομένως, επιλέγουμε τις ελάχιστες δυνατές τιμές για το PLLN όταν λαμβάνουμε υπόψη τις παραμέτρους του ρολογιού για την επιθυμητή συχνότητα DVFS εξόδου.

Επιλέγονται οι συνδυασμοί που ελαχιστοποιούν την κατανάλωση ισχύος για τον επιθυμητό στόχο SYSCLK. Διάφοροι συνδυασμοί μπορεί να δημιουργήσουν την ίδια συχνότητα εξόδου και κατανάλωση ισχύος, όπως $50, 25, 100$ και $50, 50, 200$, επομένως απαιτείται περαιτέρω έρευνα για την επιλογή του βέλτιστου συνδυασμού. Παρόμοιες παρατηρήσεις προκύπτουν για άλλες ρυθμίσεις SYSCLK.

Μετάβαση μεταξύ διαφορετικών συχνοτήτων SYSCLK: Τα περισσότερα clock trees ενσωματώνουν διαιρετές εισόδου και εξόδου στο κύκλωμα τους PLL, προκειμένου να απλοποιήσουν το σχεδιασμό του PLL και να επιτύχουν τις απαιτήσεις σταθερότητας σε ένα ευρύτερο εύρος συχνοτήτων εισόδου και εξόδου. Αυτό το εκτεταμένο κύκλωμα PLL περιλαμβάνει διαιρετές εισόδου (PLLM) και εξόδου (PLLP). Ο στόχος αυτής της διπλωματικής είναι να βελτιστοποιήσει αυτές τις παραμέτρους όσον αφορά την κατανάλωση ισχύος και τον χρόνο εκκίνησης, καθώς και να τις τροποποιήσει κατά την διάρκεια της λειτουργίας για να επιτύχει τις επιθυμητές συχνότητες εξόδου.

Η δημιουργία της συχνότητας SYSCLK χρησιμοποιώντας το κύκλωμα PLL επιφέρει μια σημαντική καθυστέρηση κατά τη μετάβαση (περίπου $200\mu sec$), διότι κατά την τροποποίηση των παραμέτρων του PLL, το κύκλωμα πρέπει να επανεκκινηθεί, με αποτέλεσμα μια σημαντική καθυστέρηση ανά μετάβαση. Το κύκλωμα PLL είναι ένα αναλογικό κύκλωμα που πρέπει να συγχρονιστεί (γνωστό και ως "κλείδωμα" του PLL) προτού μπορέσει να δημιουργηθεί η συχνότητα εξόδου και να χρησιμοποιηθεί για τη λειτουργία του πυρήνα CPU. Το PLL θεωρείται ασταθές πριν επιτευχθεί το κλείδωμα, επομένως πρέπει να πραγματοποιείται εναλλαγή σε άλλες σταθερές πηγές ρολογιού για να συνεχίσει η λειτουργία. Αυτό προκαλεί σημαντικά επιπρόσθετο κόστος σε χρόνο, καθιστώντας την αλλαγή των παραμέτρων του PLL μια δαπανηρή επιλογή σχεδιασμού για την τεχνολογία DVFS.

Από την άλλη πλευρά, η μετάβαση από τη συχνότητα του PLL στο ρολόι HSE συμβαίνει σχεδόν αμέσως, λόγω της άμεσης σύνδεσης του HSE με το SYSCLK (Σχήμα 1.3.1). Συνεπώς, για μεταβάσεις από υψηλή συχνότητα προς χαμηλή (δηλαδή $<50MHz$), η επιλογή του ρολογιού HSE αντί για την επαναβαθμονόμηση των παραμέτρων του PLL μπορεί να είναι ωφέλιμη.

Παρόλο που μια αποτελεσματική ρύθμιση του SYSCLK μπορεί να οδηγήσει σε μείωση της κατανάλωσης ενέργειας, η συχνότητα των μεταβάσεων μπορεί να επιβάλει σημαντικό κόστος. Επομένως, απαιτείται περαιτέρω έρευνα για την αποτελεσματική εφαρμογή της μεταβολής της συχνότητας, με στόχο τη μείωση της κατανάλωσης ενέργειας, χωρίς να οδηγεί σε μείωση της απόδοσης.

1.3.1 Βήμα 1: Memory Access & CPU Execution Decoupling

Αυτό το κεφάλαιο παρουσιάζει την προτεινόμενη μεθοδολογία για τη βελτιστοποίηση της κατανάλωσης ενέργειας κατά την εκτέλεση μοντέλων CNN σε έναν MCU STM32 υπό διάφορους περιορισμούς χρονικής καθυστέρησης (επίσης γνωστο ως QoS). Η λογική πίσω από αυτήν την προσέγγιση είναι ότι οι τελικοί χρήστες συχνά έχουν διακριτούς περιορισμούς χρονικής καθυστέρησης/επίδοσης για τις εφαρμογές τους και/ή χρησιμοποιούν συσκευές με περιορισμένη μπαταρία στο far edge. Συνεπώς, σε περιπτώσεις όπου οι απαιτήσεις της QoS είναι λιγότερο αυστηρές, το Dynamic Voltage and Frequency Scaling (DVFS) μπορεί να ενισχύσει την ενεργειακή απόδοση, ιδίως στο πλαίσιο των συσκευών που λειτουργούν απομακρυσμένα και με μπαταρία. Αυτό ισχύει ιδιαίτερα για συσκευές που λειτουργούν διαρκώς, όπου οι τεχνικές διαχείρισης της ενέργειας πρέπει να λαμβάνουν υπόψη μια στρατηγική ενεργειακής απόδοσης από άκρο σε άκρο. Το Σχήμα 7.0.1 δείχνει μια συνολική επισκόπηση της προτεινόμενης προσέγγισης. Η προτεινόμενη μεθοδολογία αποτελείται από τρεις διακριτικές φάσεις (που περιγράφονται στις ενότητες 7.0.1 - 7.0.3), που μπορούν να εφαρμοστούν τόσο σε μη βελτιστοποιημένα μοντέλα CNN όσο και σε βελτιστοποιημένα, π.χ. μοντέλα που εξάγονται από το MCUNet [22].

Προτείνουμε μια προσέγγιση DVFS με διαχωρισμό πρόσβασης-εκτέλεσης και λύνουμε το πρόβλημα βελτιστοποίησης της ελαχιστοποίησης της κατανάλωσης ενέργειας. Το πρόβλημά μας διατυπώνεται και

λύνεται ως πρόβλημα πολλαπλής επιλογής Knapsack. Ως είσοδο στη μεθοδολογία μας, παρέχουμε το εξεταζόμενο CNN.

Το πρώτο βήμα της προσέγγισής μας περιλαμβάνει τη διαμόρφωση παραμέτρων σε επίπεδο υλικού, με στόχο την υποστήριξη της βελτιστοποίησης της κατανάλωσης ενέργειας στα επόμενα βήματα. Η κατανάλωση ενέργειας επηρεάζεται σημαντικά από τη διαμόρφωση του ρολογιού του συστήματος. Αυτή η πρόσθετη κατανάλωση ενέργειας μπορεί να προκαλείται συχνά από χαρακτηριστικά του υλικού (π.χ. αντιστάσεις, κρίσιμα μονοπάτια μεταξύ του ρολογιού και του πυρήνα), ή από χαρακτηριστικά του κυκλώματος ελέγχου του ρολογιού (π.χ. η κατανάλωση που προκαλείται από το κύκλωμα κλειδώματος του PLL). Η συχνότητα του εξόδου του ρολογιού επηρεάζεται σημαντικά από τη διαμόρφωση του HSE ρολογιού και τη διαμόρφωση των παραμέτρων του Phase Locked Loop (PLL), δηλαδή `PLLN`,`PLLM`,`PLLP`.

Με στόχο να παράγουμε τη βέλτιστη ρύθμιση για την κατανάλωση ενέργειας, στην προτεινόμενη μας μεθοδολογία, πραγματοποιούμε: i) sensitivity analysis για τις τιμές του HSE και ii) εξερεύνηση του χώρου σχεδίασης και βελτιστοποίηση παραμέτρων του Phase Locked Loop (PLL).

Η διαμόρφωση του HSE επηρεάζει σημαντικά την κατανάλωση ενέργειας της αντίστοιχης συσκευής MCU. Πιο συγκεκριμένα, πραγματοποιούμε μια εκτενές profiling και ανάλυση της επίδρασης διαφορετικών επιπέδων συχνότητας HSE στην κατανάλωση ενέργειας. Το ρολόι HSI έχει μικρότερη ακρίβεια και μεγαλύτερο χρόνο εκκίνησης, και επομένως είναι λιγότερο επιθυμητό για εφαρμογές με αυστηρά όρια latency. Το ρολόι HSI έχει επίσης υψηλότερη κατανάλωση ενέργειας από το HSE. Στο πλαίσιο αυτής της διπλωματικής, χαμηλή επιβάρυνση επιτυγχάνεται με τη χρήση χαμηλών συχνοτήτων που δεν παράγονται από το PLL (δηλαδή συχνότητες $< 50MHz$). Επομένως, λόγω της βέλτιστης κατανάλωσης ενέργειας, το ρολόι HSE επιλέγεται ως πηγή ρολογιού για τη δημιουργία χαμηλών συχνοτήτων. Η έξοδος του πολυπλέκτη, δηλαδή το HSE, επεξεργάζεται στη συνέχεια με τις παραμέτρους του PLL: PLLM, PLLN και PLLP. Η συσχέτιση της συχνότητας εξόδου (F_{out}) με τις διαμορφώσεις HSE και PLL ορίζεται στην εξίσωση 5.2.1

Μέσω αυτής της διαδικασίας, δημιουργούμε έναν πίνακα αντιστοίχισης διάφορων επιλογών συχνότητας HSE, διαμορφώσεων παραμέτρων PLL και κατανάλωσης ενέργειας. Ο στόχος μας είναι η ελαχιστοποίηση της κατανάλωσης ενέργειας.

Πραγματοποιήθηκε ανάλυση των παραμέτρων του φάση-κλειδώματος (PLL) για τις πλακέτες STM32. Λήφθηκαν υπόψη τρεις βασικές μετρικές απόδοσης:

- Χρόνος εκκίνησης του ρολογιού (Clock startup time)
- Χρόνος κλειδώματος του PLL (PLL lock time)
- Κατανάλωση ενέργειας του PLL (PLL power consumption)

Ο χρόνος εκκίνησης του ρολογιού είναι μια μετρική που καθορίζει την καθυστέρηση που απαιτείται για την επαναρύθμιση των παραμέτρων της πλακέτας όταν πραγματοποιείται μια αλλαγή συχνότητας. Περιλαμβάνει την καθυστέρηση που προκαλείται από την αλλαγή της πηγής ρολογιού, καθώς και την επαναρύθμιση των wait states της flash και του ρυθμιστή τάσης. Όταν χρησιμοποιείται το PLL, περιλαμβάνει επίσης το χρόνο που απαιτείται για την τροποποίηση των καταχωρητών που περιέχουν τις παραμέτρους του PLL.

Ο χρόνος κλειδώματος του PLL είναι η μετρική που ποσοτικοποιεί τον χρόνο που απαιτείται για την επιπλέον καθυστέρηση που προκαλείται από τον αναλογικό κύκλωμα του PLL, προκειμένου να επιτευχθεί η συγχρονισμός. Αναφέρεται στη διάρκεια που απαιτείται για ένα κύκλωμα Phase-Locked Loop (PLL) για να συγχρονίσει τη συχνότητα εξόδου του με την επιθυμητή ή αναφοράς συχνότητα. Σε άλλα λόγια, είναι ο χρόνος που απαιτείται για το PLL να κλειδώσει στο εισερχόμενο σήμα και να εγκαθιδρύσει μια σταθερή και συνεπή συχνότητα εξόδου που αντιστοιχεί στη συχνότητα του εισερχόμενου σήματος, όταν πολλαπλασιάζεται με τις παραμέτρους του PLL. Κατά τη διάρκεια αυτής της περιόδου αλλαγής, το κύκλωμα του PLL πραγματοποιεί μια σειρά βημάτων, συμπεριλαμβανομένης της ανίχνευσης της φάσης, της σύγκρισης των συχνοτήτων και της προσαρμογής των εσωτερικών του στοιχείων, όπως του Voltage Controlled Oscillator (VCO). Ο χρόνος κλειδώματος του PLL είναι μια κρίσιμη παράμετρος σε εφαρμογές όπου είναι απαραίτητος ο ακριβής και σταθερός συγχρονισμός του ρολογιού. Συνήθως είναι προτιμητέοι μικρότεροι

χρόνοι απόκτησης, καθώς οδηγούν σε ταχύτερο συγχρονισμό του σήματος και μειωμένη περιοδική αστάθεια φάσης, εξασφαλίζοντας ακριβή και αποτελεσματική λειτουργία του συστήματος που ελέγχεται από το PLL.

Λόγω της μικρής διακύμανσης της κατανάλωσης ενέργειας, η βελτιστοποίηση επικεντρώθηκε στην ελαχιστοποίηση του χρόνου εκκίνησης του ρολογιού του MCU. Πραγματοποιήθηκε ανάλυση ευαισθησίας για διάφορες ρυθμίσεις ρολογιού, όσον αφορά τον συνολικό χρόνο εκκίνησης και την κατανάλωση ενέργειας. Παρατηρείται ότι για μια δεδομένη είσοδο και έξοδο συχνότητας, οι παράμετροι PLLN και PLLM, με χαμηλότερες τιμές πολλαπλασιαστή και υψηλότερες τιμές διαιρέτη οδηγούν σε μικρότερο χρόνο εκκίνησης του ρολογιού.

Το πρώτο στάδιο της προτεινόμενης μεθοδολογίας μας επικεντρώνεται στην αναδιάταξη του πηγαίου κώδικα, με στόχο το Decoupled Access-Execute(DAE), δημιουργώντας με αυτόν τον τρόπο υποτμήματα μνήμης και υποτμήματα υπολογισμού μέσα στη δομή του layer του νευρωνικού δικτύου.

Η αναδιάταξη αυτή αποτελεί βασικό εργαλείο για τη στρατηγική μας, καθώς παρέχει ευκολότερο έλεγχο για πότε και πόσο συχνά πραγματοποιούνται οι προσπελάσεις της μνήμης και οι υπολογισμοί (Sec. 7.0.2). Αυτό επιτρέπει την εφαρμογή διαφορετικών συχνοτήτων για εκτεταμένα χρονικά διαστήματα και με πιο ακριβή έλεγχο, προσαρμοσμένο στις συγκεκριμένες απαιτήσεις κάθε λειτουργίας, όπως η πρόσβαση στη μνήμη και οι υπολογισμοί, με αποτέλεσμα να μειώνονται τα προβλήματα που σχετίζονται με τον χρόνο εναλλαγής της συχνότητας. Για την εφαρμογή των τροποποιήσεων στο επίπεδο του πηγαίου κώδικα, πρώτα αναγνωρίζουμε τα επίπεδα του μοντέλου CNN που απαιτούν τους περισσότερους υπολογισμούς και αποτελούν το σημείο εστίασης (1A).

Επικεντρωνόμαστε και εφαρμόζουμε το DAE (1B) σε δύο συγκεκριμένους τύπους convolutional layers, δηλαδή *i)* depthwise και *ii)* pointwise.

Αυτοί οι τύποι επιπέδων αποτελούν περίπου το 80% του συνολικού αριθμού των layers που βρίσκονται σε μοντέλα CNN, όπως το Mobilenet [3], που χρησιμοποιεί την έννοια του depthwise separable convolution για να μειώσει το μέγεθος και την πολυπλοκότητα του μοντέλου.

Depthwise convolutions: Οι depthwise συνελίξεις είναι μια εξειδικευμένη λειτουργία των CNN όπου κάθε είσοδος καναλιού συνελίσσεται με έναν ξεχωριστό εκπαιδεύσιμο φίλτρο, που αντιλαμβάνεται χωρικά χαρακτηριστικά ανά κανάλι. Συνήθως, τα CNN μαθαίνουν σταδιακά όλο και πιο πολύπλοκα χαρακτηριστικά, με κάθε ένα από αυτά να αντιπροσωπεύεται από ένα διαφορετικό κανάλι. Για παράδειγμα, σε μια εικόνα, τα αρχικά 3 κανάλια εισόδου (RGB) αυξάνονται καθώς το δίκτυο επεξεργάζεται την εικόνα για να εξάγει και να αναπαριστά πιο πολύπλοκα χαρακτηριστικά, όπως υφές, συγκεκριμένα αντικείμενα κ.λπ. Πρόσφατα frameworks, όπως το CMSIS-NN [14] και το TinyEngine [22], υλοποιούν μια προσέγγιση υπολογισμού ανά κανάλι για τα depthwise convolutions.

Η προσέγγιση μας για το DAE εισάγει έναν παραμετρικό unrolling factor που ονομάζουμε "decoupling granularity", και σηματοδοτείται με το g. Αυτός ο παράγοντας καθορίζει τον αριθμό των channels που "φορτώνονται" στην cache πριν γίνει η συνέλιξη σε κάθε ένα απο αυτά. Έτσι, χωρίζουμε τη μνήμη σε memory-bound και compute-bound sections.

Το Listing 7.2 παρέχει ενα απλοποιημένο code snippet που δείχνει την υλοποίηση του DAE optimization, δείχνοντας πως το decoupling granularity δημιουργει ευνοικές συνθήκες για την αποδοτική εκτέλεση των depthwise convolutions.

Για παράδειγμα, όταν $g = 4$, τέσσερα κανάλια ανακτώνται στη μνήμη cache του MCU πριν προχωρήσουμε στον υπολογισμό. Αυτή η διαίρεση επιτρέπει την εφαρμογή διαφορετικών συχνοτήτων ρολογιού ανά τμήμα, το οποίο αναλύουμε περαιτέρω στην Ενότητα 7.0.2.

Pointwise Convolutions: Τα depthwise convolutions συνήθως ακολουθούνται από τα pointwise convolutions για να πραγματοποιηθεί μείωση της διαστατικότητας ανά κανάλι, μειώνοντας έτσι το μέγεθος του μοντέλου και την υπολογιστική πολυπλοκότητα ενώ διατηρείται η απόδοση. Οι pointwise convolutions περιλαμβάνουν πυρήνες μεγέθους `1x1` και εφαρμόζονται σε κάθε στοιχείο εντός των καναλιών εισόδου. Η CMSIS-NN [14] και το TinyEngine [22] εφαρμόζουν συνελικτικές αποτυπώσεις σημείων με τρόπο ανά

στήλη. Κάθε στήλη αποτελείται από ένα στοιχείο ανά κανάλι εισόδου. Η προσέγγισή μας πραγματοποιεί decoupling στις προσβάσεις μνήμης ανά κανάλι, διαιρώντας έτσι το τμήμα κώδικα σε περιοχές μνήμης και υπολογισμού. Όπως και στις depthwise convolutions, εισάγουμε το έννοια της αποσυζευκτικής ακρίβειας, συμβολίζεται ως g, για υποστήριξη της μονάδας προσωρινής αποθήκευσης ως προς τον αριθμό των στηλών που ανακτώνται από τη μνήμη. Για παράδειγμα, για μια εικόνα εισόδου 8x8x3 και έναν πυρήνα 1x1x3, g στήλες φορτώνονται στην μνήμη πριν πραγματοποιηθεί ο υπολογισμός για καθέναν, αντίθετα με το TinyEngine και το CMSIS-NN, που φορτώνουν μία μόνο στήλη εικόνας 1x1x3 κάθε φορά.

Συνολικά, η ενσωμάτωση πολλαπλών buffers οδηγεί στη δημιουργία μεγαλύτερων περιοχών μνήμης/υπολογισμού, με αποτέλεσμα την ελαχιστοποίηση των επιβαρύνσεων από την αλλαγή συχνότητας, αποφεύγοντας ταυτόχρονα την υψηλή κατανάλωση ενέργειας. Ωστόσο, η πολύ έντονη χρήση buffers μπορεί να οδηγήσει σε αυξημένα cache misses, με αποτέλεσμα την υποβάθμιση της απόδοσης.

Το Παράρτημα 7.1 παρέχει μια απλοποιημένη επισκόπηση της αποσυζευγμένης πρόσβασης-εκτέλεσης για pointwise convolutions.

Μετά την ολοκλήρωση της φάσης DAE, τα DAE-enabled CNN περνάνε απο profiling και design space exploration. Το τροποποιημένο DAE-enabled μοντέλο CNN προωθείται στο επόμενο στάδιο της προτεινόμενης μεθοδολογίας μας (Βήμα 2) για την αποτελεσματική εξερεύνηση και ρύθμιση του παράγοντα διακριτοποίησης DAE, μαζί με την εξερεύνηση των παραμέτρων DVFS.

1.3.2 Βήμα 2: Ταυτόχρονη Εξερεύνηση DAE και Clocking

Σε αυτό το βήμα, αναλύουμε την απόδοση και την κατανάλωση ενέργειας με ανα layer(2A), λαμβάνοντας υπόψη τις επιδράσεις της ταυτόχρονης διαμόρφωσης DAE και clocking. Για τη μέτρηση της κατανάλωσης ενέργειας και της απόδοσης κάθε layer, έχουμε αναπτύξει και ενσωματώσει έναν προσαρμοσμένο μηχανισμό παρακολούθησης κατά τη διάρκεια της εκτέλεσης για την υποστήριξη της παρακολούθησης και του profiling με per layer τρόπο. Ο μηχανισμός μας χρησιμοποιεί τους ενσωματωμένους timers του κάθε MCU, οι οποίοι ενεργοποιούνται μεταξύ των τμημάτων κώδικα των layers. Επιπλέον, εκμεταλλευόμαστε την ενσωματωμένη υποστήριξη δειγματοληψίας ισχύος των STM32 MCU και παρακολουθούμε την κατανάλωση ισχύος πριν και μετά την ενσωμάτωση DVFS σε κάθε επίπεδο CNN. Οι μετρήσεις της κατανάλωσης ισχύος και της απόδοσης για την DVFS σε κάθε επίπεδο συγκεντρώνονται και χρησιμοποιούνται για την εξερεύνηση του χώρου σχεδίασης για την διαμόρφωση DAE και clocking.

Πιο συγκεκριμένα, εξερευνούμε τον σχεδιαστικό χώρο που ορίζεται από τις ακόλουθες τρεις βασικές παραμέτρους: *i)* ο παράγοντας αποσύζευξης λεπτομερειών g που περιγράφεται στο Τμήμα 7.0.1; *ii)* η συχνότητα ρολογιού του ρολογιού `SYSCLK`; και *iii)* η επιλογή παραμέτρων για το περιοδικό σύστημα `PLL` (δηλαδή `PLLM` και `PLLN`), και οι δύο περιγράφονται στο Τμήμα **??**. Όσον αφορά το decoupling granularity, ο καθορισμός της πιο κατάλληλης τιμής ανά layer εξαρτάται τόσο από προδιαγραφές που σχετίζονται με την πλακέτα (π.χ., μέγεθος cache) όσο και από χαρακτηριστικά που σχετίζονται με τον κώδικα (π.χ., αριθμός καναλιών εξόδου και kernel size). Στην περίπτωσή μας, εξετάζουμε έξι διαφορετικές τιμές, δηλαδή $g \in 0, 2, 4, 8, 12, 16$, όπου $g = 0$ υποδηλώνει καμία βελτιστοποίηση DAE και αντιστοιχεί στο προκαθορισμένο μοντέλο εισόδου.

Όσον αφορά τις διαφορετικές εναλλακτικές ρολογιών, ορίζουμε δύο παραμετρικούς τρόπους λειτουργίας, αντίστοιχα την *Low Frequency Operation* (LFO) και την *High Frequency Operation* (HFO). Η LFO χρησιμοποιεί αποκλειστικά την πηγή ρολογιού `HSE` σε μια προκαθορισμένη συχνότητα (50MHz) και στοχεύει στη μείωση της κατανάλωσης ισχύος στην πλακέτα. Αντίθετα, η HFO ρυθμίζει το ρολόι του συστήματος χρησιμοποιώντας το κύκλωμα `PLL`, όπου η τελική τιμή `SYSCLK` καθορίζεται από διάφορους συνδυασμούς των παραμέτρων `PLLN` και `PLLM`, ειδικότερα $\texttt{PLLN} \in 75, 100, 150, 168, 216, 336, 432$ και $\texttt{PLLM} \in 25, 50$. Αυτή η διάκριση μεταξύ των δύο τρόπων λειτουργίας μας επιτρέπει να μεταβαίνουμε γρήγορα ανάμεσα σε διαφορετικές συχνότητες `SYSCLK`, ελαττώνοντας έτσι την ευαισθησία στην υπερκείμενη καθυστέρηση που συνδέεται με το `PLL`, όπως αναλύεται στο Τμήμα 5.2.2.

Συνολικά, η εναλλαγή DVFS πραγματοποιείται μεταξύ των δεσμευμένων περιοχών μνήμης (Lst. 7.2:3) και υπολογισμού (Lst. 7.2:7) , προκειμένου να αξιοποιηθεί βέλτιστα ο μετασχηματισμός αποσυνδεδεμένης

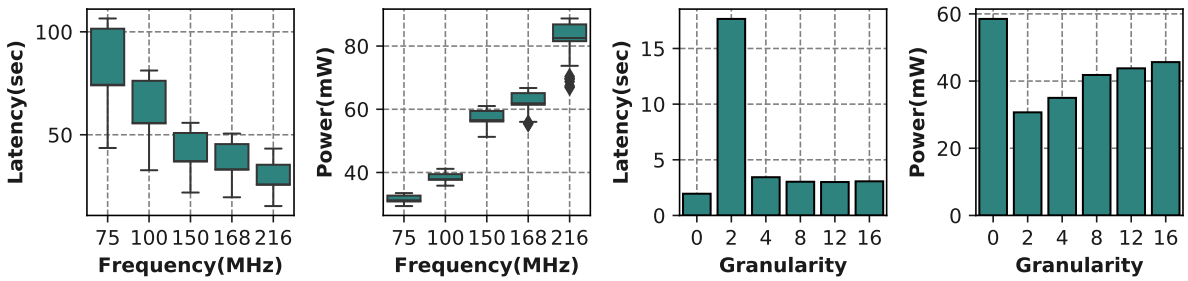

Figure 1.3.3: Επίδραση διαφορετικών ρυθμίσεων DAE και χρονισμού στην καθυστέρηση και την ισχύ των pointwise και depthwise layers.

πρόσβασης-εκτέλεσης, με το LFO να εφαρμόζεται στα δεσμευμένα στη μνήμη και τα HFO στα υποτμήματα που συνδέονται με υπολογισμό αντίστοιχα. Μια παρόμοια προσέγγιση ακολουθείται στα pointwise convolution layers.

DSE Insights: Το Σχήμα 1.3.3 δείχνει την επίδραση των μεταβαλλόμενων συχνοτήτων λειτουργίας (αριστερά) και των παραγόντων ευαισθησίας g (δεξιά) στην καθυστέρηση του layer και στην κατανάλωση ενέργειας. Πρώτον, παρατηρούμε ότι όσο αυξάνεται το μέτρο των συχνοτήτων λειτουργίας, η κατανάλωση ενέργειας ανταλλάσσεται για καλύτερη απόδοση, συνθέτοντας έτσι τον χώρο σχεδιασμού. Επιπλέον, η αλλαγή των τιμών του συντελεστή granularity μπορεί να προσφέρει σημαντική διακύμανση στην καθυστέρηση και την κατανάλωση ενέργειας. Για παράδειγμα, η κατανάλωση ενέργειας μπορεί να μειωθεί στο 54,2% σε σύγκριση με την αρχική εκτέλεση. Έτσι, η αποτελεσματική από κοινού εξερεύνηση της ευαισθησίας g και της συχνότητας οδηγεί σε συμβιβασμούς ισχύος/latency. Το αποτέλεσμα του DSE είναι ένας χώρος λύσης ανά layer, όπου κάθε λύση ανταποκρίνεται μεταξύ της απόδοσης και της κατανάλωσης ενέργειας (2B). Σε αυτό το διάστημα, επιλέγουμε τα βέλτιστα σημεία Pareto, που θα διαδοθούν στο Βήμα 3.

1.3.3 Βήμα 3: Ενεργειακή βελτιστοποίηση με επίγνωση QoS

Στο τελικό στάδιο, προσδιορίζουμε τις βέλτιστες συχνότητες για κάθε επίπεδο εντός του CNN, με στόχο να ελαχιστοποιήσουμε τη συνολική κατανάλωση ενέργειας του μοντέλου, ικανοποιώντας παράλληλα έναν προκαθορισμένο προϋπολογισμό καθυστέρησης (QoS). Συμβολίζουμε το σύνολο όλων των πιθανών συχνοτήτων που δημιουργούνται είτε μέσω του ρολογιού `PLL` ή του ρολογιού `HSE` ως F και το σύνολο όλων των πιθανών παραγόντων ευαισθησίας ως G. Έστω n ο συνολικός αριθμός επιπέδων του μοντέλου CNN και $P_k = \{..., p_j^k = \{t_j^k, E_j^k\}, ...\}, k \in \{1, ..., n\}, j \in \{1, ..., |P_k|\}$ είναι το σύνολο των βέλτιστων λύσεων Pareto (από το Βήμα 2) του επιπέδου k, όπου τα t_j^k και E_j^k υποδηλώνουν την καθυστέρηση και την κατανάλωση ενέργειας του j^{th} pareto βέλτιστη λύση για το επίπεδο k όταν λειτουργεί με DVFS ενεργοποιημένο με συχνότητα HFO $f \in F$ και ευαισθησία $g \in G$. Θεωρούμε την ελαχιστοποίηση της συνολικής ενέργειας E του CNN που αναπτύσσεται στο στόχο STM32 MCU, έτσι ώστε ο συνολικός χρόνος εκτέλεσης T να μην υπερβαίνει ένα QoS που ορίζεται από τον χρήστη. Στη συνέχεια, το πρόβλημα βελτιστοποίησης μπορεί να διατυπωθεί ως εξής:

$$\text{minimize} \quad E = \sum_{k=1}^{n} \sum_{j \in P_k} E_j^k x_{kj} \tag{1.3.2}$$

$$\text{s.t.} \quad T = \sum_{k=1}^{n} \sum_{j \in P_k} t_j^k x_{kj} \leq QoS \tag{1.3.3}$$

$$\sum_{j \in P_k} x_{kj} = 1, \quad k = 1, ..., n \tag{1.3.4}$$

$$x_{kj} \in 0, 1, \qquad k = 1, ..., n, j \in P_k. \tag{1.3.5}$$

Μοντελοποιούμε το πρόβλημά μας σύμφωνα με το Πρόβλημα Σακιδίου Πολλαπλής Επιλογής (MCKP) [35], το οποίο επεκτείνει το κλασικό πρόβλημα του σακιδίου κατηγοριοποιώντας τα στοιχεία

σε διακριτές κλάσεις(3A). Σε αυτή τη διατύπωση, η δυαδική απόφαση συμπερίληψης ενός αντικειμένου αντικαθίσταται από την επιλογή ακριβώς ενός αντικειμένου από κάθε κατηγορία και ο στόχος είναι να μεγιστοποιηθεί η αξία των αντικειμένων που περιλαμβάνονται στο σακίδιο χωρίς να υπερβαίνει το μέγεθός του. Στην περίπτωσή μας, κάθε μεμονωμένη κλάση αντιπροσωπεύει τις διάφορες βέλτιστες λύσεις Pareto p_j^k ανά layer k. Κάθε στοιχείο στην κατηγορία χαρακτηρίζεται από τη δική του αξία (δηλαδή, κατανάλωση ενέργειας E_j^k) και το μέγεθος (δηλαδή, λανθάνουσα κατάσταση t_j^k). Στόχος μας είναι να ελαχιστοποιήσουμε τη συνολική κατανάλωση ενέργειας (E) ενώ τηρούμε τον περιορισμό ποιότητας υπηρεσιών ($T < QoS$). Μετατρέπουμε τον στόχο ελαχιστοποίησης μας σε στόχο μεγιστοποίησης χρησιμοποιώντας τον μετασχηματισμό που βρίσκεται στο [35]. Τέλος, λύνουμε το πρόβλημα βελτιστοποίησης χρησιμοποιώντας μια ψευδο-πολυωνυμική λύση χρόνου που βασίζεται σε Προσέγγιση δυναμικού προγραμματισμού (DP).

1.4 Αξιολόγηση

1.4.1 Πειραματική Διάταξη

Η πειραματική μας αξιολόγηση πραγματοποιείται σε μια πλακέτα STM32F767ZI Nucleo, εξοπλισμένη με CPU ARM Cortex M7 που διαθέτει cache μνήμη L1 16 KB. Η πλακέτα ενσωματώνει ένα εξωτερικό ρολόι υψηλής ταχύτητας (`HSE`), που κυμαίνεται από 1MHz έως 50MHz. Για την παρακολούθηση της κατανάλωσης ενέργειας, χρησιμοποιήσαμε τον αισθητήρα ισχύος INA219. Για να μετριαστούν οι πιθανές διακυμάνσεις που προκύπτουν από τις διακυμάνσεις ισχύος που προκαλούνται από τη θερμοκρασία, συγκρίναμε συστηματικά κάθε μέτρηση ισχύος με την κατανάλωση ισχύος του βασικού μοντέλου εισόδου στην αντίστοιχη χρονική σήμανση. Η προτεινόμενη μεθοδολογία μας αξιολογείται σε τρία προεκπαιδευμένα μοντέλα CNN, συγκεκριμένα Visual Wake Words (`VWW`), Person Detection (`PD`) και Mobilenet-V2 (`MBV2`), που προέρχονται από το model library του MCUNet [22].

Πραγματοποιούμε μια συγκριτική ανάλυση μεταξύ της προσέγγισής μας και του state-of-the-art framework `TinyEngine` [22], το οποίο χρησιμεύει ως βάση για την αξιολόγηση. Τα πειράματά μας διεξάγονται σε ένα σενάριο εκτέλεσης ισόχρονης καθυστέρησης, όπου μετράμε την κατανάλωση ενέργειας για μια συγκεκριμένη περίοδο που καθορίζεται από έναν περιορισμό QoS. Στην περίπτωση του TinyEngine, αυτό σημαίνει ότι η πλακέτα παραμένει σε κατάσταση αδράνειας με σταθερή συχνότητα 216 MHz μετά από το inference, έως ότου επιτευχθεί το όριο QoS. Θεωρούμε επίσης το `TinyEngine` βελτιωμένο με clock gating, μια τεχνική που έχει σχεδιαστεί για τη βελτιστοποίηση της κατανάλωσης ενέργειας απενεργοποιώντας επιλεκτικά τα μη χρησιμοποιημένα ρολόγια της πλακέτας και τον ρυθμιστή τάσης, ελαχιστοποιώντας έτσι τη διαρροή ισχύος κατά τη διάρκεια του inference του CNN.

1.4.2 Αποτελέσματα

Απόδοση και Κατανάλωση ενέργειας

Το σχήμα 1.4.1 παρουσιάζει μια ενδεικτική σύγκριση της κατανάλωσης ενέργειας μεταξύ της προτεινόμενης προσέγγισής μας και των δύο διαμορφώσεων του `TinyEngine`: η μία χωρίς καμία βελτιστοποίηση και η άλλη με clock gating. Αυτή η αξιολόγηση περιλαμβάνει διάφορα μοντέλα Συνελικτικών Νευρωνικών Δικτύων (CNN), καθένα από τα οποία υπόκειται σε διακριτούς περιορισμούς ποιότητας υπηρεσίας (QoS) που ορίζονται σε 10% (σφιχτό), 30% (μέτριο) και 50% (χαλαρό). Ο άξονας X απεικονίζει τους περιορισμούς QoS, ενώ ο άξονας Υ αντιπροσωπεύει την κανονικοποιημένη κατανάλωση ενέργειας. Η προτεινόμενη προσέγγισή μας ξεπερνά και τις δύο περιπτώσεις του `TinyEngine`, παρουσιάζοντας μείωση στην κατανάλωση ενέργειας έως και 25,2%. Επιπλέον, σε σύγκριση με το `TinyEngine` με clock gating επιτυγχάνουμε 7,2% λιγότερη κατανάλωση ενέργειας. Επιπλέον, οι παρατηρήσεις μας υποδεικνύουν ότι η χαλάρωση των περιορισμών QoS μπορεί να οδηγήσει σε αξιοσημείωτη μείωση της κατανάλωσης ενέργειας, αν και με κόστος ορισμένων αντισταθμίσεων απόδοσης. Για παράδειγμα, κατά την εξέταση του μοντέλου Mobilenet-V2, η κατανάλωση ενέργειας της προσέγγισής μας υπό έναν χαλαρό περιορισμό QoS 50% μειώνεται στο 20,4% σε σύγκριση με τον αυστηρό περιορισμό 10%.

Ανάλυση frequency scaling: Το σχήμα 1.4.2 απεικονίζει το HFO για κάθε εξεταζόμενο CNN. Ο

άξονας X υποδεικνύει τον αντίστοιχο τύπο layer καθώς προχωρά η εκτέλεση του CNN, και τα επιλεγμένα granularitities για τον περιορισμό QoS 10% και 50%, αντίστοιχα, ενώ ο άξονας Y δείχνει τη συχνότητα λειτουργίας ανά επίπεδο. Η διαμόρφωση LFO παραμένει σταθερή στα 50 MHz για λόγους απλότητας. Οι παρατηρήσεις που προέκυψαν είναι οι ακόλουθες. Πρώτον, η λειτουργική συχνότητα διαμορφώνεται στο μέγιστο (216MHz) κυρίως για την εκτέλεση pointwise convolutions, δηλαδή 58,8% έναντι 21,4% για depthwise convolutions. Τα τελευταία είναι λιγότερο υπολογιστικά, επομένως η μείωση της λειτουργικής συχνότητας δεν θα οδηγήσει σε σημαντική υποβάθμιση της απόδοσης. Επιπλέον, το 46,1% των pointwise convolutions και το 43,4% των depthwise convolutions εκτελούνται στις χαμηλότερες συχνότητες, δηλαδή 75MHz και 100MHz, με στόχο την ενίσχυση του στόχου βελτιστοποίησης της ελαχιστοποίησης ισχύος. Τέλος, διερευνούμε την επίδραση των περιορισμών QoS στη συχνότητα λειτουργίας. Τα πειράματά μας δείχνουν ότι 18,6% περισσότερα layers λειτουργούν στα 216 MHz για αυστηρούς περιορισμούς (10%). Όσον αφορά την ανάλυση granularity, για το χαλαρό QoS(50%), λειτουργούν 22,3% περισσότερα layers με συντελεστή granularity 16, σε σύγκριση με τον περιορισμό 10%. Αυτό οφείλεται στο γεγονός ότι υπάρχει μεγαλύτερος χώρος για latency trading, επομένως computation-bound κομμάτια χωρίζονται σε μεγαλύτερα τμήματα, με στόχο την ελαχιστοποίηση του switching overhead και την παροχή μείωσης ισχύος.

1.5 Συμπεράσματα και Μελλοντική δουλειά

Σε αυτή την εργασία, παρουσιάζουμε μια νέα μεθοδολογία από άκρο σε άκρο που εκμεταλλεύεται το DVFS για τη βελτιστοποίηση της κατανάλωσης ενέργειας από το CNN inference σε low-end STM32 MCUs. Η προσέγγισή μας αξιοποιεί σε μεγάλο βαθμό τις τεχνικές Decoupled Access Execute για τη διακριτοποίηση τμημάτων του layer που χαρακτηρίζονται ως memory-bound και compute-bound. Αυτή η προσέγγιση χρησιμοποιεί επίσης τεχνικές βελτιστοποίησης για να αποφέρει εφικτά αποτελέσματα για αυστηρά QoS constraints. Ξεπερνά τις υπάρχουσες προσεγγίσεις επιτυγχάνοντας έως και 25,2% λιγότερη κατανάλωση ενέργειας.

1.5.1 Μελλοντική δουλειά

Αυτή η εργασία θα μπορούσε να βελτιστοποιηθεί περαιτέρω για να βελτιωθεί το scalability, η ευελιξία και η αποτελεσματικότητά της. Μέσω αυτών των επεκτάσεων, στοχεύουμε να κάνουμε το DVFS προσβάσιμο για ετερογενείς MCU χαμηλού επιπέδου, καθώς και να εξερευνήσουμε τη συν-σχεδίαση μοντέλου-υλισμικού για tinyML με δυνατότητες dynamic voltage frequency scaling.

Ορισμένες πιθανές βελτιστοποιήσεις περιλαμβάνουν:

- DVFS με fine-grained granularity και μη-ομοιόμορφη επιλογή συχνότητας εντός του kernel.
- Ενσωμάτωση διαφορετικών ευριστικών για την επιλογή σε layers και DVFS settings.
- Επιλογή στρατηγικής DVFS on-device, για μείωση της ανάγκης για εξωτερικό runtime monitoring.
- Ο αυτοματισμός ανάπτυξης για διάφορα MCU και η χρήση compiler για την αυτοματοποίηση του decoupled access execute.
- DVFS και Decoupled Access Execute aware NAS, για την δημιουργία βέλτιστων μοντέλων για DVFS.

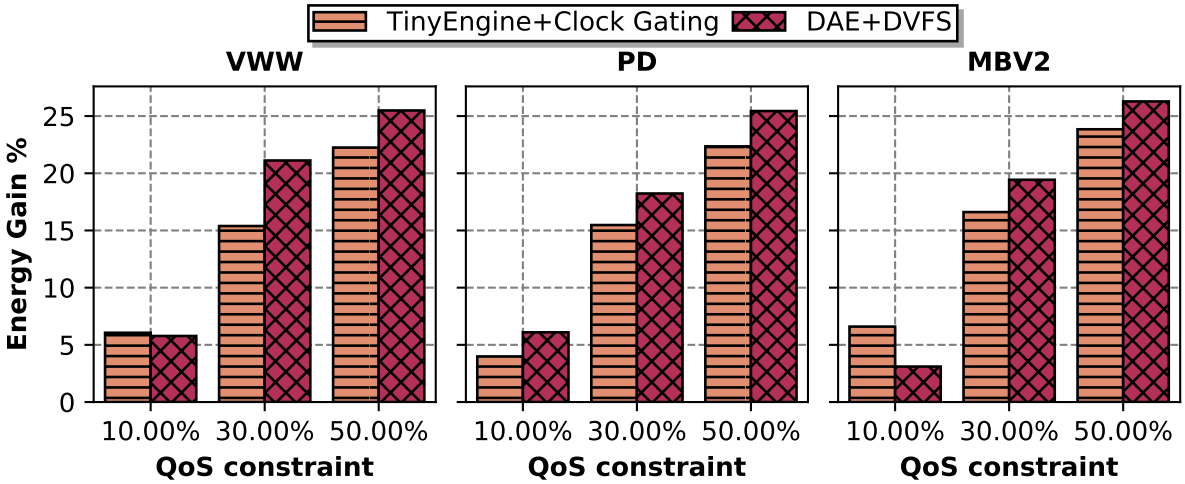

Figure 1.4.1: Ενεργειακή κατανάλωση της υλοποίησης μας σε σύγκριση με το TinyEngine [22].Συγκρίνουμε τα αποτελέσματα μας τόσο με το TinyEngine όσο και με το TinyEngine σε συνδυασμό με Clock Gating.

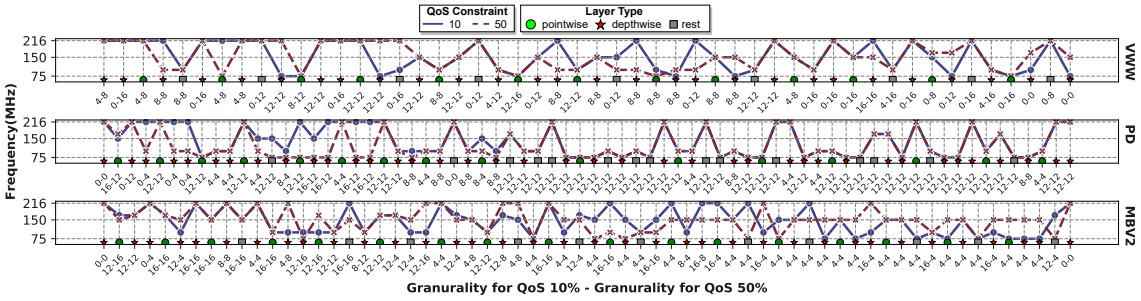

Figure 1.4.2: Κατανομή συχνοτήτων ανα layer για τα CNNs υπο εξέταση για 10% και 50%.

Chapter 2

Introduction

Over the last years, *Edge AI* [1] has appeared to the forefront as a novel computing paradigm. Edge AI refers to the deployment of complex Deep Neural Network (DNN) models directly on edge devices, such as sensors, Internet of Things (IoT) devices, and other embedded systems. Furthermore, the increasing demand for edge-centric digital signal processing applications, such as video analytics [2], mobile visual tasks [3] and others has established Convolutional Neural Networks (CNNs) as the go-to technology for efficiently handling these tasks at the edge. Notably, major technology providers have responded to this paradigm shift by offering enterprise-grade solutions designed to optimize and deploy CNN models at the edge. These solutions include software-based stacks (e.g., Amazon SageMaker Edge [4]) as well as custom, purpose-built hardware chips (e.g., Google's Edge TPU ASIC [5]).

A significant amount of research efforts are focusing on exploring the complex DNN/CNN design space of dataflow schedules [6], [7] to optimize energy. Despite their sophistication, these solutions often assume the presence of hardware accelerators or OS-supported and network-connected edge devices. However, as the computing continuum extends to the far edge of networks, arrays of resource-constrained devices, typified by microcontrollers (MCUs), are introduced into the ecosystem. Unlike their more capable counterparts, MCUs typically employ bare-metal application execution with custom or lightweight run-times and, at times, disconnected from the network grid. This paradigm shift to the far edge has highlighted the concept of "tinyML" [8]. TinyML involves the optimization of DNN/CNN models to operate seamlessly on these bare-metal MCUs, thus, extending the boundaries of where ML can be applied. In the face of limited processing power and memory, tinyML models are designed to offer efficient inference capabilities, making it possible to bring intelligence to devices previously excluded from the ML landscape.

Deploying CNNs on MCUs introduces two major challenges. First, the limited memory capacity of MCUs poses difficulties in both storing and executing CNN models. This becomes even more complex, since emerging CNN architectures tend to become more and more deeper, aiming to provide better accuracy and/or support more complex applications [9]. Second, given that MCUs are frequently embedded in battery-operated edge devices, preserving energy resources becomes crucial, since the execution of resource-intensive and computationally hungry DNNs can rapidly deplete the battery, particularly concerning devices with extended operational requirements.

While most attempts at improving energy efficiency have been focused at reducing latency while keeping power constant (i.e. pruning, kernel optimization), the explotation of clocking circuits for energy efficiency in resource constrained devices has often been neglected. Well known vendors, such as ARM and Atmel, offer no dynamic power management modes for their low end MCUs (typical power-saving modes in such MCUs suspend the CPU core thus suspending operation). In this thesis, we propose a dynamic power management scheme for low-end ARM Cortex M MCUs. We then optimize this scheme for tinyML applications, by integrating decoupled access-execute based code transformations to the inference engine. Finally we introduce a QoS-aware optimizer that performs design space exploration

on the possible configurations, in order to generate the optimal DVFS configurations for a given latency constraint.

Chapter 3

Related Work

Focusing on CNN inference, prior research [10], [11] and widely-used frameworks (e.g., TFLite Micro [12], Microsoft's NNI [13], ARM's CMSIS-NN [14], etc.) offer optimization techniques that can enable the deployment of tiny CNN models on resource-constrained MCUs. These techniques typically include static, model-specific, post-training optimizations, such as mixed precision arithmetic [15], model's weight pruning [16], [17] and quantization [18], [19], as well as downsampling [20] and Neural Architecture Search (NAS) methodologies [8], [21]. Other works examine schedule and source code optimizations either to enable efficient data and computation mapping on the underlying hardware [22]–[24] or to divide CNN layers into independently distributable tasks that can be offloaded and executed in parallel [25].

Deploying CNNs on far-edge, resource constrained devices, increases the need for energy efficient deployments, that utilize power consumption techniques. Many works have investigated the use of DVFS on CPUs and GPUs [26] [27]. This research is often coupled with decoupled access execution methodologies, aiming to minimize the overhead that is induced by clock switching [28]. Such methodologies often implement compiler-based automation solution, using just-in-time compilers for profiling-based decoupled access-execute region generation [29].

Many design automation works focus on the co-design of DVFS with neural networks, through neural architecture search. While such works have effectively managed to isolate regions of the code where DVFS can be implemented, many of them focus on dynamic neural network decision regions, where DVFS can be implemented without major code restructuring [30]. Such works have also focused on Neural Architecture search for non resource-constrained devices, in contrast to the tinyML deployments that are the goal of this thesis work.

Although a vast amount of optimizations have been investigated for DNN deployment on MCUs, they mostly focus on models' structural features, e.g., weights, arithmetic, architecture etc., while little attention is given to system-level runtime optimization knobs. In the context of power consumption and/or energy efficiency optimization, Dynamic Voltage and Frequency Scaling (DVFS) [31] forms an efficient tuning knob to balance performance and energy consumption by right-adjusting the clock frequency of the MCU according to the computational demands of the deployed model. Still, the customization of DVFS policy to DNN specific features as well as the materialization of DVFS in practice is not straightforward, as it may impose switching overheads [32] not straightforwardly analyzable and increased leakage power due to longer execution times [33].

Chapter 4

Theoretical background

The convergence of two transformative technologies, Convolutional Neural Networks (CNNs) and Tiny Machine Learning (TinyML), has ushered in a new era of intelligent computing at the edge. As the demand for deploying CNN-based models on resource-constrained devices like microcontrollers continues to surge, the quest for achieving the delicate balance between computational efficiency and model accuracy becomes increasingly paramount. Dynamic Voltage and Frequency Scaling (DVFS), a long-standing technique in embedded systems for managing power and performance, presents itself as a promising tool to address this challenge. By dynamically adjusting the voltage and clock frequency of microcontrollers, DVFS can optimize power consumption while adhering to QoS constraints for CNN-based tasks. In this background section, we delve into the foundational principles of CNNs, explore the emerging field of TinyML, and provide a comprehensive overview of DVFS, setting the stage for an in-depth investigation into their application for efficient CNN inference on resource-constrained edge devices.

4.1 Machine Learning - Introduction

Machine learning (ML) constitutes one of the most important advancements in the fields of computer science and data analysis in recent years. It represents a paradigm shift in problem-solving, where computers are not explicitly programmed to perform a task but instead learn from data and adapt their behavior autonomously. This departure from traditional rule-based programming and optimization techniques has opened up a realm of possibilities across various domains. In this section, we delve into the fundamental principles of machine learning, explore its diverse types, and highlight its critical distinctions from traditional optimization methods.

4.1.1 Types of Machine Learning

Machine learning encompasses a spectrum of techniques, each designed to address specific problem types and learning paradigms. Three primary categories are prominently recognized:

- **Supervised Learning**: Supervised learning is perhaps the most familiar type of machine learning. In this paradigm, algorithms learn to map input data to predefined output labels based on a labeled training dataset. Common applications include image classification, natural language processing, and regression tasks. Supervised learning models aim to generalize patterns from training data to make accurate predictions on unseen data. Notable examples include decision trees and support vector machines [36].
- **Unsupervised Learning**: Unsupervised learning focuses on discovering hidden patterns or structures within data without explicit labels [37]. Clustering and dimensionality reduction are typical tasks in this category. Clustering algorithms group data points based on similarity, while

dimensionality reduction methods seek to capture essential features of the data while reducing complexity. Unsupervised learning is vital for tasks like anomaly detection and data exploration. The most popular example of an unsupervised learning algorithm is the k-means algorithm, where items are often classified according to their spatial properties.

- **Reinforcement Learning**: Reinforcement learning models interact with an environment and learn to make sequences of decisions that maximize a cumulative reward signal. This type of learning has found success in applications such as game-playing, robotics control, and autonomous systems. Reinforcement learning algorithms explore various strategies and adapt their behavior based on feedback from the environment [38].

4.1.2 Supervised learning: Types and characteristics

Supervised learning is a foundational category of machine learning where algorithms learn to map input data to corresponding output labels based on a labeled training dataset. Within this domain, several types of supervised learning have emerged, each tailored to specific data types and problem domains. In this section, we delve into three primary types: Artificial Neural Networks (ANNs), Convolutional Neural Networks (CNNs), and Recurrent Neural Networks (RNNs), while highlighting their distinctive characteristics.

Artificial Neural Networks

Artificial Neural Networks, represent the most popular basis for many machine learning models. ANNs consist of interconnected layers of nodes organized into an input layer, one or more hidden layers, and an output layer. These nodes are interconnected through weighted synapses (which are often referred to as weights). These weights shape the importance of each feature in the neural network. The nodes process input data through weighted connections and activation functions to produce predictions.

ANNs, especially shallow architectures with fewer layers and neurons, tend to have a lower computational cost compared to deeper networks. The cost primarily depends on the number of parameters (weights and biases) to be learned and the number of operations required for inference.

Each neuron with a binary threshold in an artificial neural network can be described by the equation, as proved in [39]:

$$y = \theta(\sum_{j=1}^{n} w_j x_j - u) \tag{4.1.1}$$

where theta is the step function (used as an activation for the neuron), w represents the synapse weight matrix, x represents the input matrix and u is the activation threshold. In modern literature, u is often referred to as the bias of the neuron.

In modern works, the threshold function was replaced by other activation function in order to achieve the desired asymptotic properties. Some notable activation functions include:

- The sigmoid function [40],
$$g(x) = \frac{1}{1 + exp\{-\beta x\}} \tag{4.1.2}$$
where β is the slope that the sigmoid function has.
- The rectified linear unit function (ReLU) [41],
$$g(x) = max(0, x) \tag{4.1.3}$$
- Leaky rectified linear unit [42],
$$g(x) = \begin{cases} \beta x & \text{for } x < 0 \\ x & \text{otherwise} \end{cases} \tag{4.1.4}$$

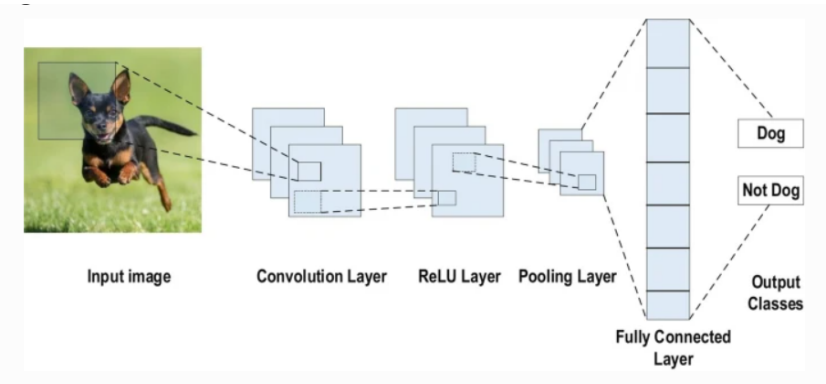

Figure 4.1.1: Example of a CNN architecture.

where β is the slope of the function.

- Hyperbolic tangent function (often referred to as *tanh*) [43],

$$g(x) = \frac{(e^x - e^{-x})}{(e^x + e^{-x})} \tag{4.1.5}$$

Convolutional Neural Networks

Convolutional Neural Networks are a specialized type of ANN designed to excel in processing grid-like data, such as images and videos. They introduce convolutional layers to efficiently capture spatial hierarchies and patterns within the input data. In contrast to fully connected layers, these architectures utilize local connections to extract image properties, which are spatially correlated.

At their core, CNNs employ convolutional layers that scan input data with learnable filters, enabling the detection of local patterns. This property makes CNNs ideal for tasks like image classification, object detection, facial recognition, and image generation. Furthermore, CNNs exhibit spatial invariance, allowing them to recognize patterns in different regions of an image, making them indispensable in applications where understanding complex visual data is essential. As models get bigger, deploying them in resource constrained devices, optimizing and deploying CNN variants plays an important role in enabling efficient and accurate image-related tasks on resource-constrained embedded devices. CNNs are computationally more intensive than simple ANNs, primarily due to the convolutional layers, which involve a significant number of multiplicative operations for feature extraction. An example of a CNN is shown in Fig.4.1.1.

Many CNNs can be employed in TinyML applications, particularly for image-related tasks, when optimized implementations, quantization, and model pruning are utilized to reduce computational requirements. Efficient architectures like MobileNets are designed to be more lightweight, making them suitable for resource-constrained devices.

Most CNNs consist of the following components [43]:

- **Convolutional layers** apply learnable filters to input data, detecting local patterns and hierarchies within the data. Each convolutional layer receives a filter (often referred to as kernel), whose utility is to extract specific features out of the input image. Each element in the kernel matrix holds a different value, and those values are adjusted during training depending on the loss value (or the value of other metrics as discussed below). In order to extract border features, padding elements are used. When output dimensions need to be reduced, a larger stride value is utilized.
- **Pooling layers** reduce spatial dimensions and enhance translation invariance by downsampling feature maps. They help to retain important information while reducing computational complexity.

- **Fully Connected layers** make predictions based on the high-level features learned by previous layers. They serve as the output layer of the network, producing the final results.
- **Activation functions** such as ReLU (Rectified Linear Unit), introduce non-linearity into the model, enhancing its capacity to capture complex relationships within the data. They help make the network more expressive and adaptable.

This work mainly focuses on the efficient deployment of convolutional neural networks for detection tasks.

In order to understand the optimization domain of the current research issue, one needs to examine the optimization domain. This requires some background on the building blocks of convolutional neural networks.

Each CNN layer receives an input that has been coded in order to support optimal feature extraction. The input layer typically receives input in 3 channels, red green and black (RGB) and has 3 dimensions, height, width and depth (also indicated by the number of channels). The objective is to extract feature maps, used to make the model's predictions. Each feature map is calculated as follows:

$$h^k = f(W^k * x + b^k) \tag{4.1.6}$$

where h is the k-dimensional feature matrix, f is the activation function, W is the weight matrix, x is the input and b is the bias matrix.

These outputs are then sub-sampled using pooling functions.

In order to train the network, a loss function is employed in order to quantify the impact of each weight change on the network performance.

Loss functions are employed in the output layer of the CNN, in order to calculate the output error, which is representative of the difference between the correct output (ofter referred to as the data label) and the predicted output. Some of the most popular loss functions include:

- The **Cross-Entropy** or **Softmax Loss Function** (also known as the log loss function) yields an output in the form of a probability (p) within the range of [0,1]. In the output layer, this function leverages softmax activations to generate a probability distribution. The probability distribution is described by [44]:

$$p_i = \frac{e^{a_i}}{\sum_{k=1}^{N} e_k^a} \tag{4.1.7}$$

  The loss function is given by:

$$H(p, y) = -\sum_i y_i log(p_i) \text{ where } i \in [1, N] \tag{4.1.8}$$

- **Euclidean Loss Function**, often used in regression related problems:

$$H(p, y) = \frac{1}{2N} \sum_{i=1}^{N} (p_i - y_i)^2 \tag{4.1.9}$$

- **Hinge loss function**, most commonly used in binary classification:

$$H(p, y) = \sum_{i=1}^{N} max(0, m - (2y_i - 1) * p_i) \tag{4.1.10}$$

Recurrent Neural Networks

Recurrent Neural Networks are designed for sequential data, where the order of elements matters, such as time series data, natural language, and speech. RNNs introduce recurrent connections to capture dependencies across time steps. RNNs can be computationally demanding, especially when processing long sequences or using deep RNN architectures. The recurrent nature of RNNs makes them less efficient in terms of parallelization compared to feedforward networks. Deploying RNNs on microcontrollers for tasks like real-time natural language processing or speech recognition can be challenging due to their computational demands. Lightweight variants like GRU (Gated Recurrent Unit) and LSTM (Long Short-Term Memory) can be more suitable for TinyML when computational efficiency is a priority.

4.1.3 Building blocks of Convolutional Kernels

Convolutional kernels consist of various different types of convolutions, which have been derived from the original convolution operation, in order to achieve better performance in different metrics (computational cost, accuracy, etc.)

The convolution operation

In the context of neural network training, a convolution refers to a process that reduces the number of model parameters needed for learning by condensing image information into fewer pixels. Convolutions involve aggregating data from nearby pixels, effectively summarizing them into a more compact representation. This summarization is achieved by moving a kernel or filter across an image, producing an output for each position it covers.

A convolutional kernel is represented as an n x n matrix of numerical values. Each element within the kernel matrix is multiplied with the corresponding pixel value it aligns with in an image. These products are then summed up to produce a single output value. Subsequently, the kernel is shifted to a new position in the image, and this process is iterated. The ideal values for a kernel matrix are acquired through model training. For optimal performance, a model learns values for a kernel that capture essential information within an image, even though this information may not be directly interpretable by humans. Examples of such information that kernels in Convolutional Neural Networks (CNNs) can extract include specific objects, structural patterns, or prominent outlines within an image.

Types of convolution

There are 3 main variants of the convolutional kernel:

- **Regular convolutions**: Standard convolutional kernels are the foundation of Convolutional Neural Networks (CNNs). They consist of learnable filters that slide across the entire input volume. Each filter captures spatial hierarchies of features by considering all input channels simultaneously. These kernels are effective at detecting complex patterns and interactions between features in the data. However, they come at a computational cost, as they involve a large number of parameters and operations. Standard convolutions are commonly used in the early layers of CNNs to capture low-level features like edges and textures.
- **Pointwise Convolutions (1x1 Convolutions)**: Pointwise convolutions, often referred to as 1x1 convolutions, are a crucial component in optimizing the computational efficiency of CNN architectures. These convolutions use 1x1-sized filters to process each element within a feature map independently. While they may seem trivial due to their size, pointwise convolutions play a significant role in adjusting the dimensionality of feature maps. They are employed to reduce the number of channels while preserving essential features. This reduction in channel dimensions not only reduces computational complexity but also facilitates model compression and acceleration, making them particularly valuable in scenarios with limited computational resources.

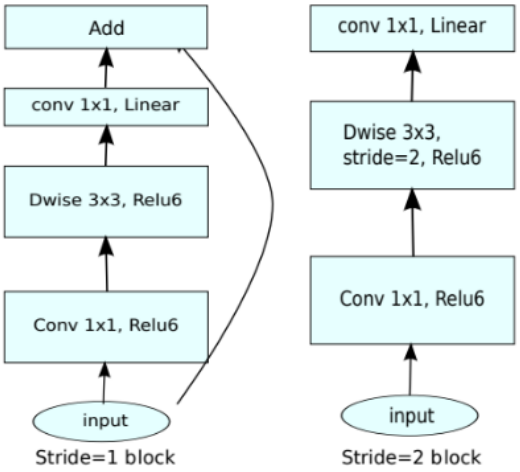

Figure 4.1.2: MobilenetV2 architecture.

- **Depthwise Convolutions**: Depthwise convolutions represent another key optimization technique for CNNs, especially in resource-constrained environments like mobile and embedded devices. Unlike standard convolutions, depthwise convolutions perform spatial convolution separately for each input channel. This means that for each input channel, a separate set of filters is applied, leading to a significant reduction in both parameters and computational operations. Depthwise convolutions are commonly followed by pointwise convolutions (known as depthwise separable convolutions) to capture inter-channel interactions, further reducing the computational load. These convolutions are particularly well-suited for tasks that require high computational efficiency, such as real-time image processing and object detection on edge devices.

4.1.4 MobilenetV2 Architecture

One of the most popular CNN architectures, the MobileNet network, as well as the improved MobilenetV2 network, exploit pointwise and depthwise convolutions in order to achieve efficient inference for vision-related tasks. The architecture and computational cost of such networks is described below.

Standard convolution

The standard convolution process involves taking an input tensor Li with dimensions hi $\times$ wi $\times$ di and applying a convolutional kernel K, which is of size k $\times$ k $\times$ di $\times$ dj, resulting in an output tensor Lj with dimensions hi $\times$ wi $\times$ dj. Standard convolutional layers come with a computational cost determined by hi $\times$ wi $\times$ di $\times$ dj $\times$ k $\times$ k.

Depthwise separable convolutions

Depthwise Separable Convolutions are a fundamental component in the construction of highly efficient neural network architectures.

The core concept involves replacing a traditional convolutional operation with a more efficient alternative that divides the convolution into two separate stages. Firstly, the depthwise convolution layer applies lightweight filtering to the input data, accomplishing this by individually applying a single convolutional filter to each input channel. Secondly, the pointwise convolution layer, with a 1 $\times$ 1 kernel, is responsible for creating fresh features by performing linear combinations across the input channels.

Depthwise separable convolutions illustrate similar accuracy to standard convolutions, with reduced computational cost. More specifically, a depthwise separable convolution has a cost of:

$h_i * w_i * d_i(k^2 + d_j)$

It is worth noting that these convolutions reduce the computational cost by a factor of k^2 [45].

Linear Bottlenecks

Consider the context of a deep neural network comprising n layers, each characterized by an activation tensor of dimensions $h_i w_i d_i$. These are considered to be groups of $h_i w_i$ pixels, each with d_i dimensions.

The concept of a "manifold of interest" in a given layer implies that the set of activations at that particular layer can be construed as a structured entity. These manifolds can be projected onto lower-dimensional spaces. When examining the "d-channel pixels," of a deep convolutional layer, the information in these channels can be simplifies by projection into lower dimensional spaces [46].

While this efficiency optimization is utilized in previous works, such as MobilenetV1 [46], it is subject to various constraints.. Notably, deep convolutional neural networks incorporate nonlinear transformations, such as the Rectified Linear Unit (ReLU). In contrast to linear transformations that don't have dimensionality constraints, ReLU introduces various complexities. For instance, applying ReLU to a one-dimensional line yields a "ray," whereas its application in an R^n space (an n-dimensional space) yields a non linear solution[45].

Essentially, it can be observed that deep networks can act similarly to linear classifiers within the domain of non-zero volume in the output space [45].

Conversely, when ReLU reduces the dimensionality of a channel, which results in setting certain values to zero, this unavoidably results in information loss within that channel. However, when multiple channels exist, and there exists a structure (as described above) within the activation manifold, information preservation can potentially be conserved within other channels.

Two notable conclusions can be reduced from the definitions above:

- When the manifold of interest retains a non-zero volume following a ReLU transformation, it corresponds to a linear transformation.
- ReLU possesses the capability to preserve comprehensive information pertaining to the input manifold, but only under the condition that the input manifold resides within a low-dimensional subspace of the input space.

These observations indicate that neural network architecture could be improved by integrating linear bottleneck layers into convolutional blocks.

Henceforth in this thesis, the "expansion ratio" is defined as the ratio between the size of the input bottleneck and the inner size.

For a block with dimensions hw, an expansion factor of t, and a kernel size of k, having d_0 input channels and d'_0 output channels, the total number of multiply-add operations required can be expressed as $h * w * d_0 * t * (d_0 + k^2 + d'_0)$. While this expression includes an additional term compared to the standard approach due to an extra 11 convolution, the nature of the MobilenetV2 networks typically utilizes smaller input and output dimensions.

The MobilenetV2 architecture is customized to fit various performance requirements. This is achieved by employing adjustable hyperparameters (the input image resolution and width multiplier). These hyperparameters can be fine-tuned based on the desired trade-offs between accuracy and performance[45].

4.2 Dynamic Voltage Frequency Scaling

The burgeoning field of edge artificial intelligence (AI) has sparked many research endeavors aimed at investigating and optimizing power efficiency in edge devices. Over the past several decades, research has prominently featured techniques such as idling exploitation, low-power operating systems [47], methodologies dedicated to low-power design for reconfigurable devices [48], and dynamic voltage frequency scaling. These methodologies collectively represent a multifaceted approach to mitigating the power consumption challenges associated with edge AI deployment.

In particular, idling exploitation strategies involve the useful utilization of idle states in hardware components, allowing devices to temporarily enter low-power modes when computational demands are minimal. Low-power operating systems, tailored to the constraints of edge devices, facilitate energy-efficient task scheduling and resource management. Concurrently, methodologies for low-power design in reconfigurable devices seek to optimize the energy footprint of programmable logic, making them amenable to AI workloads at the edge. Furthermore, dynamic voltage frequency scaling (DVFS) techniques enable real-time adjustments to the operating voltage and clock frequencies of processors, striking an equilibrium between performance requirements and energy conservation.

At the heart of DVFS lies a fundamental relationship between voltage (V) and clock frequency (F) in microelectronic devices. This relationship can be expressed by the following equation:

$P = C * V^2 * F$

Where:

- P is the power consumption.
- C denotes the capacitance of the device.
- V is the supply voltage.
- F represents the clock frequency.

Therefore, by reducing either the voltage or frequency, considerable power savings can be achieved.

When DVFS is applied in the context of a CPU, the CPU clock frequency is scaled, in order to reduce power consumption. In tandem to the frequency modification, the voltage regulator and the flash memory wait states are modified in order to maximize the energy gain and ensure consistent operation.

DVFS approaches often exploit processor idling in order to select regions where the frequency is scaled down. More specifically, regions where the processor is idle and waiting for the external memories are exploited, in order to obtain optimal energy consumption with negligible performance degradation. This methodology is often referred to as **Decoupled Access-Execute** [28].

4.3 Optimization - Pareto optimality

Multiobjective optimization problems are characterized by increased difficulty, as there is no unique solution; rather, there is a set of acceptable trade-off optimal solutions. This set is called Pareto front [49]. Examples of pareto frontiers are illustrated in Fig. 4.3.1.

Pareto Dominance

A solution is considered to have dominance over another solution when it fulfills the following two criteria:

- The objective values of the first solution are at least as good as those of the second solution in all objectives. In simpler terms, for this two-objective scenario, Solution A performs as well as or better than Solution B in every objective, denoted as $f(A) \leq f(B)$ for all objectives.

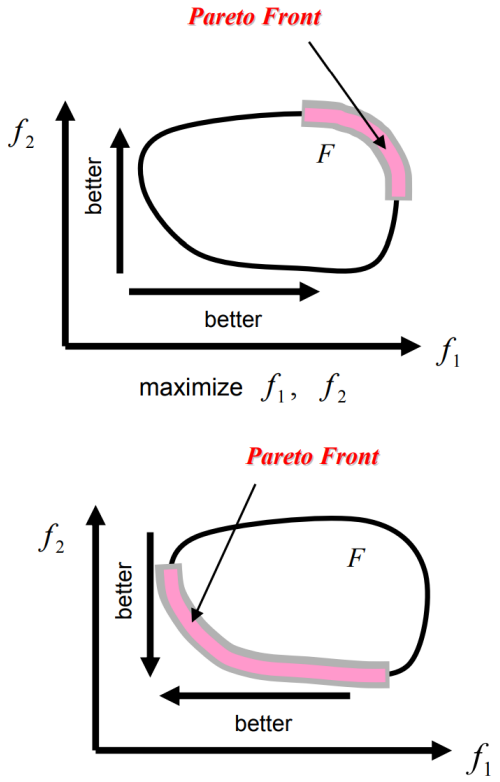

Figure 4.3.1: Examples of pareto frontiers for different optimization use-cases[49].

- The objective values of the first solution are clearly superior to the objective values of the second solution in at least one objective. In this context, for this two-objective situation, Solution A outperforms Solution B in at least one objective, denoted as f(A) < f(B) for at least one objective.

If either of these two conditions is not met, it cannot be stated that Solution A dominates Solution B. However, when both criteria are satisfied, it can be asserted that Solution A holds dominance over Solution B. The pareto frontier is formed by solutions that are non-dominated across the design space. An example of pareto domination concepts is illustrated in Fig.4.3.2

4.4 Optimization algorithms - Knapsack

The Knapsack Problem is a classic optimization problem that can be encountered in various real-world scenarios. At its core, it involves making choices under constraints to maximize a certain objective. Each item in the knapsack optimization problem is characterized by two main metrics: the cost metric (often referred to as the item's weight) and the value, which indicates the reward of including the item. In the context of the Knapsack Problem, the algorithm gets input as a set of items, each with its own weight and value, and a knapsack with a limited capacity. The goal is to decide which items to include in the knapsack to maximize the total value of the selected items, while ensuring that the combined weight of the items does not exceed the knapsack's capacity.

The objective function of the knapsack problem can be formed as follows:

$maximize \ \sum_{i=1}^{n} u_i x_i$

$subject \ to \ \sum_{i=1}^{n} w_i x_i \leq W$

where w_i indicates the weight of item i, u_i indicates the item value, x_i indicates the number number of items i to include and W is the maximum weight capacity.

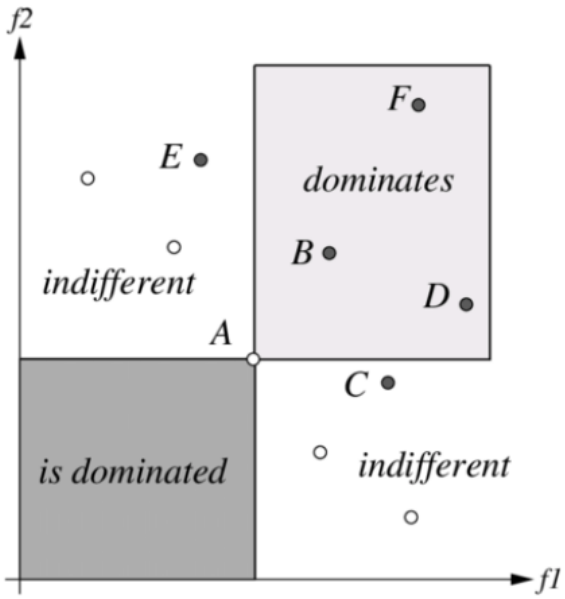

Figure 4.3.2: Pareto domination example.

When the knapsack variation only enables selection of one item i, the value x_i is constrained to {0,1}

It is worth noting that the knapsack optimization problem is NP-complete, meaning that there is no polynomial time algorithm for the general case of the algorithm. There are many approaches to solving the knapsack optimisation problem more efficiently, such as dynamic programming, branch and bound and optimization algorithms.

In the dynamic programming approach, an adjacency matrix is used to store the optimal solutions. The matrix equation can be expressed as:

$c[i, w] = max\{c[i-1, w-w[i]] + P[i]\}$

where $w[i]$ is the weight of item i and P[i] is the value of item i.

4.4.1 Multiple Choice Knapsack

The Multiple-Choice Knapsack Problem (MCKP) [35], extends the classical knapsack problem by categorizing items into distinct classes. In this formulation, the binary decision of including an item is replaced by the selection of precisely one item from each class and the goal is to maximize the value of items included in the knapsack while not exceeding its size. The problem can be formulated as follows:

$$\text{minimize} \quad E = \sum_{k=1}^{n} \sum_{j \in P_k} E_j^k x_{kj} \tag{4.4.1}$$

$$\text{s.t.} \quad T = \sum_{k=1}^{n} \sum_{j \in P_k} t_j^k x_{kj} \leq QoS \tag{4.4.2}$$

$$\sum_{j \in P_k} x_{kj} = 1, \quad k = 1, ..., n \tag{4.4.3}$$

$$x_{kj} \in 0, 1, \qquad k = 1, ..., n, j \in P_k. \tag{4.4.4}$$

4.5 TinyML

TinyML, short for Tiny Machine Learning, represents a groundbreaking intersection of two technologies: machine learning and resource-constrained embedded systems. It refers to the deployment of machine learning models on tiny, resource-constrained devices such as microcontrollers, sensors, and other constrained hardware. TinyML is expected to revolutionize various industries by enabling intelligence at the edge, where data is generated, without relying on a constant internet connection or powerful cloud servers.

The main components that need to be taken into account when designing tinyML systems are the following:

- **Hardware**: TinyML is deployed on microcontrollers, sensors, and other embedded hardware. These devices have limited processing power, memory, and energy resources, making it challenging to deploy traditional machine learning models. Many tinyML works also explore solutions that are architecture and hardware-specific, making the hardware an essential parameter to consider for tinyML.
- **Inference Engine**: The inference engine is responsible for running machine learning models on edge devices. It optimizes model execution for minimal computational and power consumption. Current inference engines consist of hardware-efficient kernels that exploit the underlying hardware in order to push the state-of-the-art boundaries on latency.
- **Machine Learning Models**: TinyML employs compact, optimized machine learning models. These models are specifically designed to run efficiently on resource-constrained hardware. The models are either optimized post-training using techniques like quantization, pruning, and model distillation or before they are trained, with methodologies such as Neural Architecture Search, depending on specific problem constraints.
- **Data Acquisition**: Edge devices gather data from sensors or other sources. This data serves as input for the machine learning model. This involves optimizing the latency and energy consumption of the peripherals as well as the data buses, often in conjunction with the main machine learning workloads.

The elements mentioned above constitute the main focus point for tinyML research. It is worth discussing the main constraints that are encountered when these elements need to be optimized. The most notable ones include:

- Resource Constraints: Developing models that can operate within the limited memory and processing power of edge devices requires careful optimization and trade-offs. Modern MCUs are extremely constrained in terms of RAM and Flash memories, whereas state-of-the-art machine learning models have been exponentially scaling in size. This calls for a different approach to machine learning deployments in tinyML devices.
- Variance in sensor readings: TinyML sensors are typically lower resolution, making their performance degrade when the model has been trained on different, less constrained sensors. Training neural networks online in such devices has proved to be extremely challenging, with few works attempting limited, on-device training that fine-tunes the network **hanlabtraining**.
- Energy Efficiency: Managing power consumption is critical for battery-operated devices, making efficient model design and inference essential.
- Security: Protecting TinyML models from tampering or reverse engineering is essential, particularly in safety-critical applications.

While tinyML has many challenges, it also presents significant benefits that cloud computing often fails to deliver. These advantages include:

- Privacy and Data Localization: With TinyML, data is preserved on the edge device, enhancing privacy and reducing the risk of data breaches. The lack of network connection also makes tinyML

robust to numerous network-based cyber attacks. This is especially critical in applications where data security and compliance are paramount such as biomedical use-cases.

- Low Latency: TinyML enables real-time processing of data, making it ideal for applications that require quick decision-making. This includes robotics, autonomous systems, and applications in industrial automation.
- Offline Functionality: Many TinyML applications continue to function even when there is no internet connection, providing uninterrupted service in remote or disconnected environments.

4.5.1 CMSIS-NN

Deploying neural networks on resource-constrained platforms, such as Arm Cortex-M CPUs, presents unique challenges due to limited computational resources and memory. Many works have focused on architecture specific code optimizations. The most notable ones include ARM's CMSIS-NN [50], microTVM [51], Microsoft's NNI [13]. CMSIS-NN focuses on the most popular neural network kernels, as well as utility functions and performs optimizations while taking into consideration the underlying ARM architecture. The framework consists of two main components: The first component, named NNFunctions, where most neural network kernels are implemented. These kernels are fully parametric, making them easy to integrate with various frameworks, such as TFLM[52] or Tiny Engine [50] The most notable ones include convolution, depthwise separable convolution, fully-connected, pooling, and activation. NNSupportFunctions, where utility functions such as data conversion and activation function tables are implemented. These utility functions unlock important functionalities such as the capability to deploy more complex neural network models (Long Short Term Memory, Gate Recurrent Units). At last, the kernel generation process is optimized. For a given neural network, the optimal kernels are selected and used to perform inference.

Quantization

Traditionally, neural network models are trained using 32-bit floating-point data representation, which demands high precision. However, during inference, the high computational cost of floating-point operations and the memory overhead for storing weights and activations can be prohibitive. Most tinyML-optimized kernels utilize quantization in order to counter this problem, thus reducing model size and improving efficiency. This approach is preferred for ARM Cortex M CPUs, as it also enables faster computation through Single Instruction Multiple Data (SIMD) operations.

The quantization granularity is determined by the ARM instruction set, optimized in the CMSIS framework, which supports 8-bit and 16-bit quantized datatypes. These datatypes are declared through separate datatype instructions, namely `q7_t (int8)`, `q15_t (int16)`, and `q31_t (int32)`. Quantization is performed in a fixed-point format with power-of-two scaling, represented as $A * 2^n$. This quantization format also enables the use of lookup tables for implementing activation functions and makes dequantization between layers redundant.

Efficient kernel implementation in CMSIS-NN

As mentioned previously, CMSIS-NN utilizes SIMD instructions to perform efficient computation. More specifically 16-bit Multiply and Accumulate operations are performed using specialised instructions such as `SMLAD`. These instructions are the building block of multiplication operations, which are the main bottleneck for neural network computation.

The support functions in CMSIS-NN mostly implement data reordering and in-register operations in order to implement efficient basic matrix multiplication and transformations between quantized datatypes.

The matrix multiplication functions, implemented in NNFunctions, utilize the basic multiplication implemented in NNSupportFunctions, in order to optimize the in terms of data reuse. The implementation, based on CMSIS `mat_mult` kernels, employs 2×2 kernels to maximize data reuse and reduce

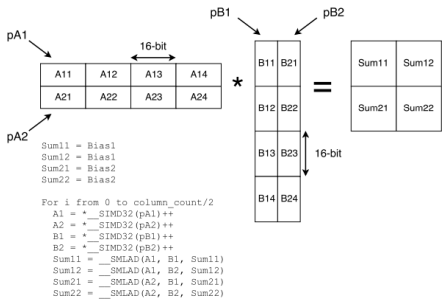

Figure 4.5.1: Matrix multiplication example with CMSIS-NN [50].

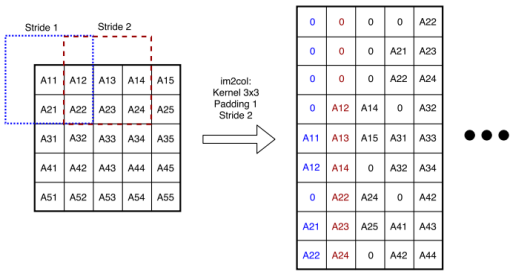

Figure 4.5.2: Example of an im2col transformation [50].

load instructions. These functions also include optimized kernels for matrix vector multiplications. An example of CMSIS matrix multiplication is illustrated in 4.5.1

The convolution operations are based on the image-to-column (im2col) transformation (Fig. 4.5.2). This transformation performs input reordering and expansion, in order to utilize Single Instruction Multiple Data (SIMD) instructions to perform efficient convolution. Since input expansion utilizes memory resources which are limited in microcontrollers, CMSIS-NN performs partial `im2col`, only expanding two columns at a time. In addition, data format optimality is utilized (Channel-Width-Height (CHW) and Height-Width-Channel (HWC)), and efficient HWC convolutional kernels are implemented.

4.5.2 TinyEngine

Many works have based their implementations on the CMSIS-NN kernels, and added additional optimization in order to improve performance. The TinyEngine framework combines an optimized, CMSIS-NN based inference engine with a Neural Architecture Search Framework, that generates optimal model architectures for the given engine. TinyEngine aims to make large scale detection models deployable on resource constrained MCUs.

TinyNAS employs a two-stage neural architecture search (NAS) approach capable of accommodating the diverse memory constraints of various microcontrollers. It automates the search space optimization process by adjusting input resolution and model width to fit specific resource constraints. TinyNAS also performs neural architecture search within the optimized space, resulting in improved model accuracy.

TinyEngine, the memory-efficient inference library, eliminates unnecessary memory overhead and allows for a larger model capacity within the given memory constraints. While most existing inference libraries rely on interpreters, TinyEngine adopts code generator-based compilation to eliminate the interpretation overhead and free up memory for larger models. This approach reduces memory usage and improves inference efficiency. Furthermore, TinyEngine employs model-adaptive memory scheduling, optimizing memory allocation based on model-level statistics. This enhances input data reuse and re-

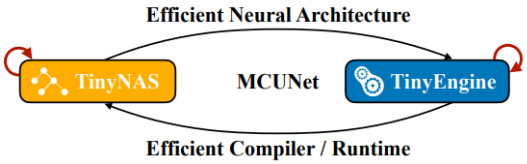

Figure 4.5.3: TinyEngine and MCUnet architecture[22].

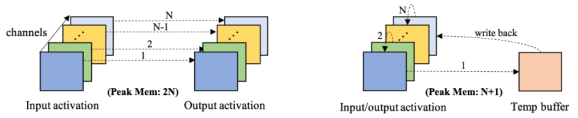

Figure 4.5.4: In place depthwise convolution [22].

duces runtime overheads, such as memory fragmentation and data movement. Specialized computation kernel optimizations further improve inference efficiency.

An overview of the framework's architecture is illustrated in Fig 4.5.3. Some of the main code optimizations implemented in TinyEngine include:

1. **In-place depth-wise convolution**: This technique replaces input data with intermediate or output data, thus performing operations without additional memory overhead. It's used to reduce the maximum SRAM memory requirement during processing.

2. **Operator fusion**: Performance improvements are achieved by combining different operators in a way that they can be executed together without the need to access memory repeatedly. This utilizes data reuse, thus improving efficiency.

3. **Patch-based inference**: TinyEngine adopts a strategy where it focuses on smaller sections of the feature map during inference, processing data in a patch-by-patch manner. The input data is processed in patches and the features for each patch are extracted with less memory overhead. The patches are then processed to extract correlation feature between them. This approach effectively reduces the amount of memory needed for large layers.

4. **HWC to CHW weight format transformation**: This technique transforms the format of weights in a way that enhances the efficiency of cache usage (useful for in-place depth-wise convolution) as well as SIMD operations in convolution, which require a specific data format.

5. **SIMD (Single instruction, multiple data) instructions**.

6. **GEMM operations**: It's a method for computing convolution operations using general matrix multiplication (GEMM) operations, optimizing the process.

7. **Loop reordering**: This technique involves rearranging the sequence of loops within a program to enhance its execution speed, by facilitating cache and register file hits.

8. **Loop unrolling**: It's a method that speeds up program execution but increases binary size, representing a trade-off between space and time efficiency.

9. **Loop tiling**: Loop tiling partitions a loop's iteration space into smaller blocks or chunks, reducing memory access latency and ensuring that data used in a loop remains in the cache until it's needed again.

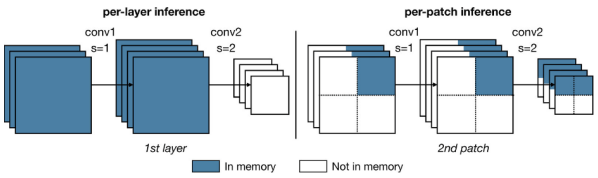

Figure 4.5.5: Example of patch-based inference [22].

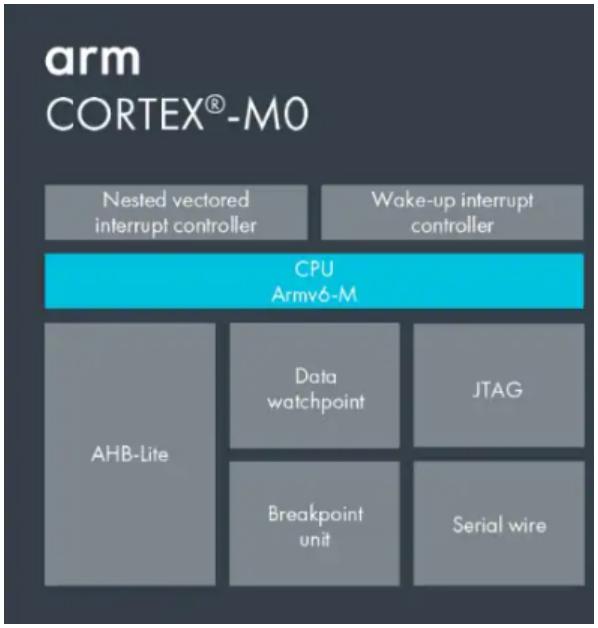

Figure 4.6.1: ARM Cortex M0 architecture overview

4.6 ARM Cortex M Architecture

The deployments in this thesis are performed on the ARM Cortex M series of CPUs which are integrated in the STM32 family.

4.6.1 ARM Cortex M0

The Arm Cortex-M0 is the smallest Arm processor available. It is characterized by a very small silicon area, low hardware capabilities, low power and minimal code footprint. Suitable for analog and mixed signal devices, it features 8-bit and 16-bit precision.

M0-based microcontrollers are widely adopted and offer significant advantages in entry-level applications. They meet the computing performance needs, and their fundamental design allows them to achieve exceptionally low power consumption, particularly in applications where minimizing the number of switching gates is essential. The Cortex®M0 core helps reduce electromagnetic interference (EMI) and meets performance requirements by optimizing clock speed.

- Versatile Core Usage: The compact size of the core allows it to serve as either a standalone core in small devices or as an additional embedded companion core when specific hardware isolation or task separation is needed. The core size in these CPUs as small as $0.03mm^2$ in 90nm lithography.
- Dynamic Power Range: The power consumption of the core varies from 5 to 50µW per MHz, depending on the manufacturing technology used. However, it's crucial to note that the core's power consumption is just one part of the overall device power profile.
- Thumb Instruction Set: The Thumb instruction set, which is a subset of the Cortex-M family,

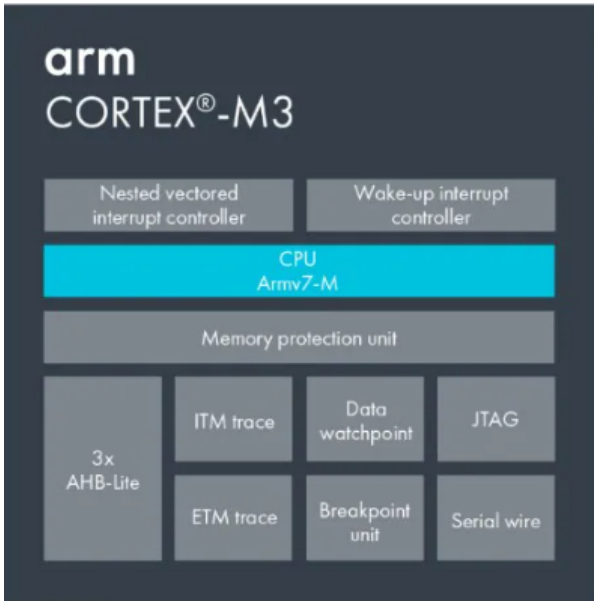

Figure 4.6.2: An overview of the ARM Cortex M3 architecture.

simplifies the scalability of the product portfolio by enabling the reuse of validated software components across various Cortex-M products.

The ARM Cortex M0 CPUs are typically integrated into the STM32F0 family of MCUs.

4.6.2 ARM Cortex M3

The 32-bit Arm Cortex-M3 core processor is suitable for high-performance, real-time processing and can handle complex tasks. Any Arm Cortex-M3 microcontroller offers scalability combined with an optimal trade-off between performance and cost.

The main features of the M3 series include:

- Compact Size: The compact size of the core enables its utilization either as a sole core in diminutive devices or as an extra embedded companion core in situations necessitating precise hardware isolation or task segregation.
- Dynamic power consumption (10 to 150µW/MHz)
- Memory Protection Unit (MPU): The Memory Protection Unit (MPU) manages the CPU's access to the memory. It ensures that a task does not accidentally corrupt the memory or the resources used by other active tasks. The MPU is usually controlled by a Real-Time Operating System (RTOS).

The ARM Cortex M3 cores are integrated in STM32L1, STM32F1 and STM32F2 MCU families.

4.6.3 ARM Cortex M33

The Arm Cortex-M33 core processor is tailored for IoT and embedded applications demanding effective security or digital signal management. This processor includes a digital signal processing extension (DSP), TrustZone security for robust hardware-based isolation, memory-protection units (MPUs), and a floating-point unit (FPU).

In terms of performance, the Cortex-M33 outpaces the Cortex-M4 by approximately 20%, boasting a remarkable 1.5 DMIPS/MHz and 4.09 CoreMark/MHz.

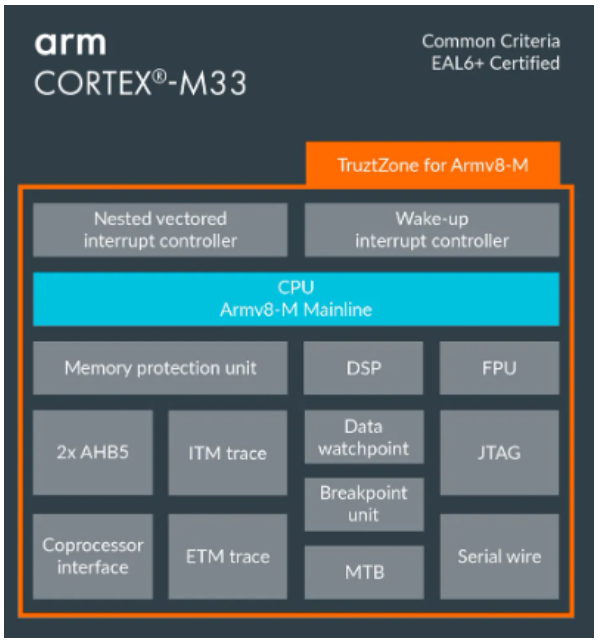

Figure 4.6.3: An overview of the ARM Cortex M33 architecture

The main benefits of the M33 series include:

- Armv8-M architecture: The Cortex-M33 leverages the Armv8-M architecture, which incorporates programmer models optimized for swift processing with minimal latency. Additionally, it offers the flexibility to integrate a memory protection unit (MPU) based on the protected memory system architecture (PMSA).
- Lower design costs and easier system design: The Cortex-M33 includes digital signal processing (DSP), single instruction on multiple data (SIMD) and MAC instructions. It also included CMSIS support, simplifying interaction with the peripherals.
- Large scope of applications: Cortex-M33 core includes low-latency interrupt handling, integrated sleep modes, debug and trace capabilities

The Arm Cortex M33 CPUs are included in the STM32L5, STM32U5, STM32H5 and STM32WBA boards.

4.6.4 ARM Cortex M4

The Cortex-M4 core achieves 1.25 DMIPS/MHz and 3.42 CoreMark/MHz thread performance.

The main highlights of this CPU include:

- Armv7E-M architecture
- Digital Signal Processing
- Scalability and power efficiency
- Dual core processing options in big.LITTLE architectures, either as the main or as the secondary core

4.6.5 ARM Cortex M7

The Cortex-M7 core achieves 2.14 DMIPS/MHz and a 5.01 CoreMark/MHz thread performance.

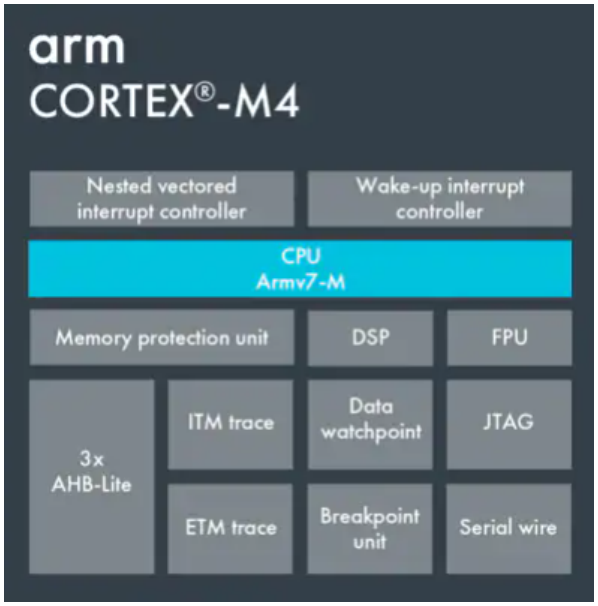

Figure 4.6.4: An overview of the ARM Cortex M4 architecture.

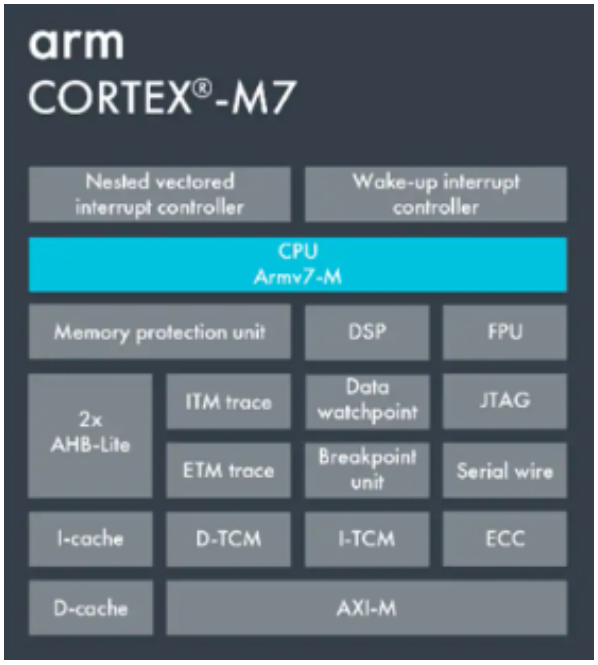

Figure 4.6.5: An overview of the M7 architecture

The main advantages of the M7 series include:

- A 6-stage superscalar pipeline with branch prediction, combined with instruction and data caches
- High CPU frequency up to 216MHz
- Instruction and data Tightly Coupled Memories (TCM) allowing 0-wait execution
- Double precision FPU

Chapter 5

Clocking Scheme of STM32 Microcontrollers

5.1 Properties of STM32 MCUs

The STM32 Nucleo boards are equipped with 32-bit ARM Cortex-M microcontrollers (MCUs) along with Instruction and Data caches, as well as integrated Flash and SRAM memories. For the purposes of this thesis, emphasis will be placed on the STM32F7 series, and as such, the following specifications will pertain primarily to these boards.

5.1.1 F7 Architecture

The STM32F7 Nucleo family of MCUs have the following defining characteristics:

- 32-bit MCU
- Floating point unit
- 462 DMIPS computing capabilities
- up to 2 MB flash memory
- 512 KB RAM memory
- 16 KB instruction TCM RAM
- 4 KB backup
- DSP instruction support

5.1.2 Power efficiency

The STM32 boards include native support for various low power modes. Each mode utilizes voltage regulation or circuit gating. The STM32 devices support three low power modes:

- **Sleep mode**
  When sleep mode is enabled, the CPU is turned off, while the rest of the peripherals continue to operate. The CPU wakes up when an interrupt is issued.
- **Stop mode**
  The Stop mode achieves the lowest power consumption while retaining the contents of SRAM and registers. All clocks in the 1.2 V domain are stopped, the PLL, the HSI RC and the HSE

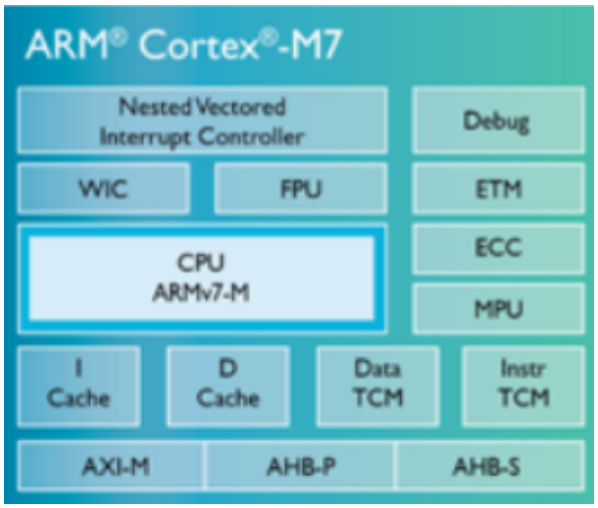

Figure 5.1.1: Arm Cortex M7 CPU architecture.

crystal oscillators are disabled. The voltage regulator can be put either in main regulator mode (MR) or in low-power mode (LPR).

- **Standby mode**
  The Standby mode is used to achieve the lowest power consumption. The internal voltage regulator is switched off so that the entire 1.2 V domain is powered off. The PLL, the HSI RC and the HSE crystal oscillators are also switched off. After entering Standby mode, the SRAM and register contents are lost except for registers in the backup domain and the backup SRAM when selected.

5.2 Clocking scheme

The clocking system of STM32 microcontrollers is managed by the Reset and Clock Control (RCC) peripheral. The RCC provides a wide range of clocks and clock sources which cater to various system requirements, e.g., peripheral Bus and UART clocks [34]. In this work, we focus on specific clocks and settings, which determine the frequency of the system clock (`SYSCLK`) of the MCU, responsible for driving the CPU core, memory, and some peripheral modules. Figure 5.2.1 illustrates the simplified circuit diagram of how the `SYSCLK` is determined. Specifically, the output frequency of the clock can be configured by the following clock sources and settings:

- **High-Speed Internal (`HSI`) Clock:** The `HSI` clock source is an internal oscillator within the MCU. The `HSI` clock operates at 16MHz by default. `SYSCLK` can be derived directly from `HSI` or through the `PLL` when `HSI` is selected as the `PLL` input source.
- **High-Speed External (`HSE`) Clock:** The `HSE` clock source is an external clock provided by an external crystal oscillator or clock generator. Depending on the MCU, the `HSE` clock can be configured to run at various frequencies, with our examined board supporting a range from 1 to 50MHz. Similar to the `HSI`, the `SYSCLK` can be derived directly from `HSE` or through the `PLL` when `HSE` is selected as the `PLL` input.
- **Phase-Locked Loop (`PLL`):** The `PLL` is a hardware module which allows to multiply the frequency of the selected input clock source (either `HSI` or `HSE`) by a programmable factor to generate a higher-frequency output. As shown in Fig. 5.2.1, this programmable factor is determined by input and output dividers within the `PLL` circuit, which simplify the `PLL` design and achieve stability requirements within a wider range of input and output frequencies [34]. Specifically, the frequency of the system clock can be calculated by the following equation:

Acronym	Short Description
HSI	**H**igh **S**peed **I**nternal clock. Used as an intermediate clock to continue operation when the PLL is off
HSE	**H**igh **S**peed **E**xternal clock
PLL	**P**hase **L**ocked **L**oop. Consists of PLLM, PLLN, PLLP, PLLQ parameters used to adjust the input frequency
DAE	**D**ecoupled **A**ccess **E**xecution
Granularity N	Number of buffers (N) to be decoupled

Table 5.1: Acronym description

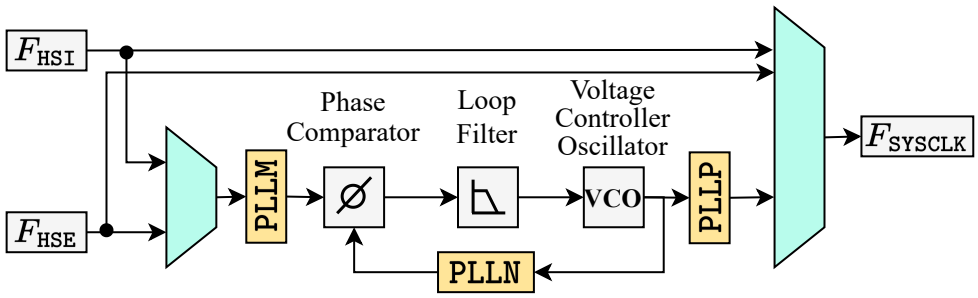

Figure 5.2.1: Simplified circuit diagram for clock configuration through `HSE` and `PLL` parameters.

$$F_{\texttt{SYSCLK}} = F_{\{\texttt{HSE,HSI}\}} * \frac{\texttt{PLLN}}{\texttt{PLLM} * \texttt{PLLP}} \tag{5.2.1}$$

where, `PLLM` is a factor that determines how much the input frequency is multiplied before it reaches the Voltage-Controlled Oscillator (`VCO`) in the `PLL`; `PLLN` is the multiplication factor for the `VCO` input frequency to determine the `VCO` output frequency, which is then provided as feedback to the Phase Comparator, aiming to provide input/output synchronization. The loop filter is utilized in order to ensure system stability and mitigate ripple effects during clock startup; Last, `PLLP` determines the division factor to obtain the final `SYSCLK` frequency.

An overview of the acronyms described above is provided in Table 5.1.

5.2.1 Clock switching methodology

In order to perform Dynamic Voltage Frequency Scaling, the register configuration of the STM32 MCUs needs to be examined.

Reset and clock control registers

The Reset and Clock control peripheral consists of registers used by the STM32 Hardware Abstraction Layer and the STM32 Low Layer libraries, to configure the clock properties at board startup time. The board starts using the default HSI configuration at 16MHz. The HAL auto-generated code performs alterations to the RCC registers in order to configure the clock source, parameters and output frequency according to the user selection (usually performed through the STM32 Cube IDE clock configuration GUI).

The RCC registers can be exploited to alter the clock properties at runtime, using software commands. The outline of the clock switching methodology is as follows

1. `PLL` to `HSE` switching
   When the low frequency is $\leq$ 50 MHz, the High Speed External Clock can be utilized as a direct

output clock source, in order to reduce switching overhead induced by the PLL startup time

- The `RCC->CFGR` clock configuration register is modified by clearing the previous `RCC_CFGR_SW` mask and updating it with the `RCC_CFGR_SW_HSE` flag. This changes the clock source to the High Speed External clock.
- The `RCC->CFGR` value is read until the necessary flags have been set, in order to avoid out-of-order execution due to compiler optimizations.
- The `RCC->CR` clock control register is modified, in order to disable the PLL.
- Overdrive mode and overdrive switching, used to increase performance at high frequencies, are disabled by modifying the `PWR->CR1` register flags.
- The voltage scale is adjusted by modifying the `PWR->CR1` register.
- The flash wait states are adjusted according to the output frequency, using the `FLASH->ACR` register flags.
- The SysTick (which relates to the timer synchronization) and SysClk (which records the frequency of the system clock) variables are updated accordingly

2. `PLL` to `HSI` switching
   - The `RCC->CFGR` register is modified by clearing the previous `RCC_CFGR_SW` mask and updating it with the `RCC_CFGR_SW_HSI` flag. This changes the clock source to the High Speed Internal clock.
   - The `RCC->CFGR` value is read until the switch is complete.
   - The `PLL` is disabled.
   - Overdrive mode and overdrive switching are disabled.
   - The `SysTick`, flash wait states and voltage scale are adjusted as previously.
3. `PLL` to `PLL` switching
   When the high and low frequency are both $>$ 50MHz, the phase locked loop needs to be utilized for both output frequencies. The clock switch can be described as follows:
   - The clock source is switched to the `High Speed External` clock, in order to continue operation when disabling the `PLL`.
   - The `PLL` is disabled.
   - The flash wait states are modified.
   - The new `PLL` parameters are selected and passed as flags in the `RCC- >PLLCFGR`
   - The `PLL` is enabled.
   - Due to the high startup time of the `PLL` circuit, the processor polls the dedicated `PLL_RDY` flag, to determine whether the clock source can be changed to the `PLL`.
   - The clock source is changed to the PLL, as described above.
   - The `SysTick`, flash wait states and voltage scale are adjusted.

5.2.2 Considerations of `SYSCLK` Frequency Scaling

Determining the frequency of the system's clock, $F_{\texttt{SYSCLK}}$, can be achieved in different ways (e.g., directly through the `HSI/HSE` clock sources or through the `PLL` circuit). The STM32 boards provide multiplexer selection of the input clock source, as well as the PLL input clock source. It should be noted that each of these selection affect crucial system metrics for DVFS, such as the power consumption and

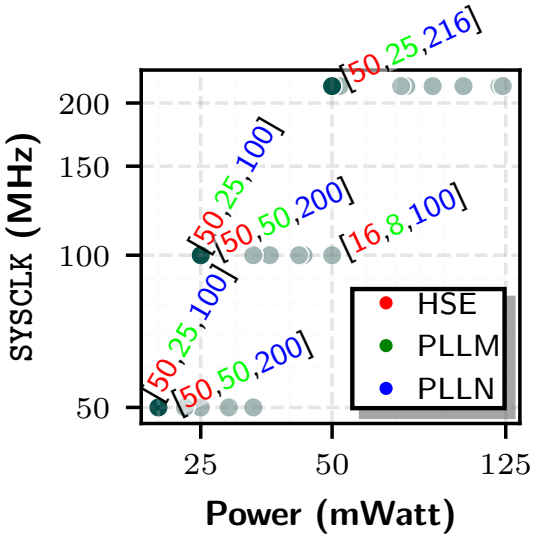

Figure 5.2.2: Clock Frequency and Power for different HSE, PLLM and PLLN configurations

clock startup time. Each of the option are examined and their results are investigated in the following sections.

It has been established that choosing input clock sources for both the output clock and the PLL involves important tradeoffs. To examine the impact of each alternative on the operational efficiency and power consumption of the MCU, we develop and execute a specialized microbenchmark, designed to execute repetitive addition operations within a loop. We focus our exploration specifically on the HSE and PLL parameters. The HSI clock source yields higher power consumption compared to the HSE and is also prone to drift and jitter, thus, providing less stability and precision [34]. It has also been noted [53] has a higher startup time than the High Speed External clock. Therefore, it would be suboptimal when selected as a clock source. Furthermore the HSI clock operates at a constant frequency (typically 8 or 16MHz), making it less versatile for DVFS. At last, selecting the HSI as intermediate clock when performing PLL-to-PLL switches would also be suboptimal, as the 16MHz frequency leads to unneccessary idling during clock switching. We set the value of the PLLP equal to 2, which is the minimum possible value for the divider, since for the same F_{SYSCLK}, selecting a higher PLLP value leads to a higher required VCO frequency and, thus, higher power consumption (as evident from Fig. 5.2.1 and Eq. 5.2.1). It is worth noting that setting the PLLP value to the minimum compatible value for the desired output frequency is preferred, as using a divider after the PLL circuit essentially "wastes" PLL energy, as the previous parameters have been set to higher values in order to generate a higher frequency as the PLLP input.

Power consumption of iso-frequency configurations: Figure 5.2.2 illustrates the impact of different HSE, PLLM and PLLN configurations on the power consumption of the board for different SYSCLK frequencies. Two major observations are derived from Fig. 5.2.2: (i) the same output frequency can be generated through different HSE, PLLM and PLLN combinations, however (ii) the selected configuration strongly affects the power consumption on the STM32 MCU. More specifically, the HSE input value has an inverse relationship with the power consumption, provided that the output frequency is constant and identical in both of the configurations. Furthermore, according to the STM32 dataset, the value of PLLN should be minimized in order to achieve the minimal VCO voltage. However, the difference seems to be minor in terms of power consumption 5.2.2. It should be noted that despite the minor variance in power consumption, lower PLLN values also yield faster clock startup time, as the PLL lock time is reduced **??**. Thus, we select the smallest possible values for the PLLN when considering clock parameters for a desired DVFS output frequency. For instance, generating a set SYSCLK output frequency with different PLL parameters leads to 50% power gap. The combinations that minimize the power consumption are selected for the target SYSCLK. Different combinations may generate the

same output frequency and power consumption, e.g. $\{50, 25, 100\}$ and $\{50, 50, 200\}$, thus further investigation is required for selecting the optimal combination. Similar observations are derived for other SYSCLK configurations.

Switching between different SYSCLK frequencies: Most clock trees integrate input and output dividers within their PLL circuit, in order to simplify the PLL design and achieve stability requirements within a wider range of input and output frequencies. This extended PLL circuit includes input (PLLM) and output (PLLP) dividers. The objective of this framework is to optimize these parameters in regards to power consumption and startup time, as well as modify them at runtime to achieve the goal output frequencies.

Generating the SYSCLK frequency using the PLL module introduces a notable switching overhead ($\approx 200\mu sec$), since, when modifying the PLL parameters, the circuit has to be restarted, resulting in a substantial delay per switch. The PLL circuit is an analog circuit that needs to be synchronized (also referred do as the PLL "lock"). Before the output frequency can be generated and used to operate the CPU core. The PLL is considered unstable before the lock has been achieved, therefore intermittent switching to other stable clock sources needs to be performed in order to continue operation. This causes significant overhead, making the change of PLL parameters a costly design option for DVFS.

On the other hand, switching from the PLL frequency to the HSE clock occurs almost instantly, due to the direct wiring of the HSE with the SYSCLK (Fig. 5.2.1). Thus, for high-to-low (i.e. $< 50MHz$)) frequency switches, opting for the HSE clock over re-calibrating the PLL parameters can be beneficial.

Chapter 6

Decoupled access execute

One way to address the substantial overhead associated with clock switching is to implement Dynamic Voltage and Frequency Scaling (DVFS) at a level of granularity that is more detailed than the entire program but coarser than just a few individual instructions. By splitting program execution into distinct phases characterized by consistent memory behavior, it becomes possible to predict and set the optimal operating frequency for each phase separately, as demonstrated in reference [16]. The finer the segmentation, the more effectively DVFS can be utilized (although there is a practical limit due to the latency involved in DVFS transitions, which adds overhead). Consequently, in any execution model where memory operations closely interact with arithmetic computations, we can anticipate harnessing only a portion of the potential benefits offered by DVFS.

To address the intricacies of this challenge, existing works [28] introduce a novel solution named Decoupled Access Execute. This solution is coupled with DVFS in order to maximize its' performance gain. Within this framework, the execution of programs is organized into a sequence of tasks, with each task partitioned into two distinct phases: the access phase, which entails data prefetching, and the execute phase, responsible for the core computational workload. In multi-core systems, the execute phase is scheduled for immediate execution on the same processing core following the conclusion of the access phase, in order to avoid cache misses. In this thesis, we focus on single core MCUs.

The central concept underlying this approach is the deliberate decoupling of data access from computational processing, affording us the capability to make independent voltage and frequency decisions for each phase—distinguishing between the access and execute phases. This significantly reduces the switching overhead induced by naive DVFS. An outline of this methodology is illustrated in 6.0.1 During the access phase, the primary objective is data prefetching, a process that effectively mitigates the occurrence of cache misses during the subsequent execute phase, thereby substantially amplifying overall performance. The access phase predominantly involves waiting for data retrieval from memory, with a relatively minor portion of its time allocated to address computation. Consequently, it remains relatively impervious to fluctuations in core frequency, allowing for its operation at the lower frequency levels —a strategic power-saving measure that does not compromise performance integrity.

Conversely, the execute phase gets substantial benefits from the prefetching operations executed during the access phase. This translates into a notable reduction in the frequency of cache misses during the execute phase. The minimization of execution stalls during this phase aligns with the utilization of the highest frequency setting, making it the optimal choice from the perspective of the Energy-Delay Product (EDP).

In any execution model the main source of power inefficiency is induced by the stalls during the memory accesses, because the processor is spending power waiting for memory. Stalls occur when computations are unable to conceal prolonged miss penalties. By lowering the operating frequency, the computational processes (during the memory access phase in the case of decoupled access-execute)

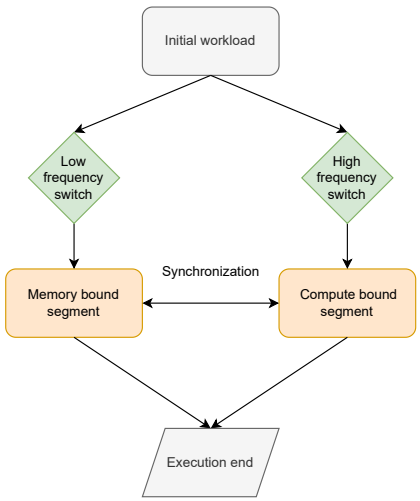

Figure 6.0.1: Example of Decoupled access execution methodology.

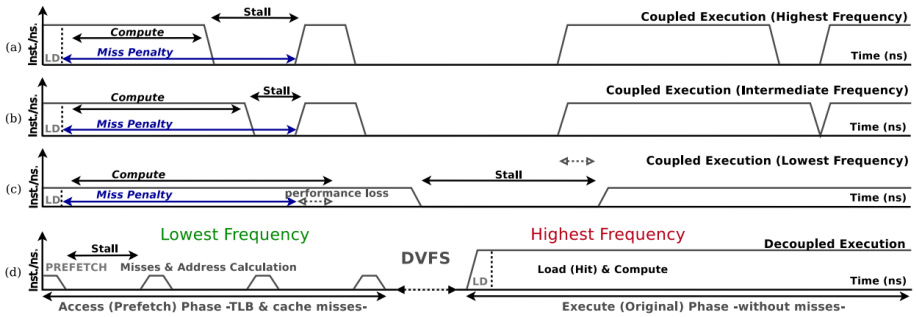

Figure 6.0.2: Example of DVFS stalling with and without Decoupled Access Execute[28].

are deliberately slowed down. This deliberate slowing down allows computations to overlap more with the miss penalty, resulting in a reduction in the duration of stalls. Although execution at the highest frequency is optimal when it comes to execution time, it is suboptimal in terms of power efficiency due to the processor idling at high speed while waiting for the memory elements from the RAM memory. The goal of co-designing DAE with DVFS is to match the miss penalty to the new, lower-frequency induced execution time, thus exploiting processor idling without causing additional execution time overhead.

In this scenario, data prefetching is conducted into the cache during the access phase, effectively mitigating the majority of cache misses during the subsequent execute phase due to the previously completed prefetching. The process of decoupling enables the transformation and division of an instruction interval into two distinct phases: an entirely memory-bound phase (referred to as "access") succeeded by a purely compute-bound phase (named "execute"). This separation enables the fine-tuning of program behavior in alignment with DVFS granularity. By doing so, we can optimize the utilization of available memory resources more effectively, adjusting the operating frequency downward when the system is waiting for memory operations.

While current works focus their approach on parallel workloads, this thesis implements Decoupled Access Execute on single core, single-thread sequential workloads, such as the ones encountered in tinyML applications. In this case decoupling is performed with code transformations and no inter-core communication takes place. The cache hierarchy discussed is also simplified compared to complex CPUs. The boards that were evaluated contain at most L1-caches, as well as instruction and data caches that have not been integrated as tightly near the CPU core.

Chapter 7

Proposed methodology

This chapter introduces the proposed methodology for optimizing energy consumption of CNN model inference on a STM32 MCU under various latency constraints (a.k.a. QoS).The rationale behind this approach is that end-users often have distinct inference latency/throughput constraints for their applications and/or even operating in the context of battery-operated far-edge MCUs. Consequently, in cases where QoS requirements are less strict, Dynamic Voltage and Frequency Scaling (DVFS) can be exploited to amplify energy efficiency, particularly in the context of battery-operated far-edge MCUs. This is particularly true for always-on devices, where energy management techniques need to consider an end-to-end energy efficiency strategy. Figure 7.0.1 shows an end-to-end overview of the proposed approach. The proposed methodology consists of three distinct phases (described in Sections 7.0.1 - 7.0.3), applicable to both unoptimized CNN models and optimized ones, e.g., models exported from MCUNet [22]. We propose a decoupled-access execute DVFS approach and tackle the NP-complete optimization problem of power consumption minimization, while obtaining the maximum possible performance. Our problem is formulated and solved as a Multiple Choice Knapsack optimization problem. As input to our methodology, we provide the examined CNN.

The first step of our approach consists of the configuration of hardware-level parameters, aiming to provide support for energy consumption optimization on the next steps. Power consumption is strongly affected by the configuration of the system clock. This overhead can often be induced by hardware characteristics (e.g. resistances, critical paths between the clock and the core), or clock control circuit characteristics (e.g. the PLL consumption caused by the lock circuit). The of the output clock is highly affected by the configuration of the High Speed External (HSE) clock and the configuration of the Phase Locked Loop (PLL) parameters, namely `PLLN,PLLM,PLLP`. Aiming to derive the optimal power consumption configuration, in our proposed methodology we perform: i) sensitivity analysis for the configuration of the HSE and ii) exploration and optimization of the Phase Locked Loop (PLL).

The configuration of the HSE significantly affects the power consumption of corresponding MCU device. More specifically, we perform an extensive profiling and analysis of the impact of different HSE frequency levels on the power consumption. HSI clocks tend to have jitter, and are therefore less preferable for latency-critical applications. The HSI clock also yields higher power consumption than HSE. In the context of this framework, low overhead switching is implemented by utilizing non-PLL generated low frequencies (i.e frequencies $< 50MHz$). Therefore, due to its optimal power consumption, the HSE clock is selected as a clock source for low frequency generation. The output of the multiplexer, i.e. the HSE, is then processed with the PLL paremeters: PLLM, PLLN and PLLP. The correlation of output frequency (F_{out}) with the HSE and PLL configurations is defined in equation 5.2.1.

Through this process, we generate a table mapping of different HSE frequency options, PLL parameter configurations and power consumption. Our optimization objective is to minimize the power consumption, thus the combination that minimizes the power is selected.

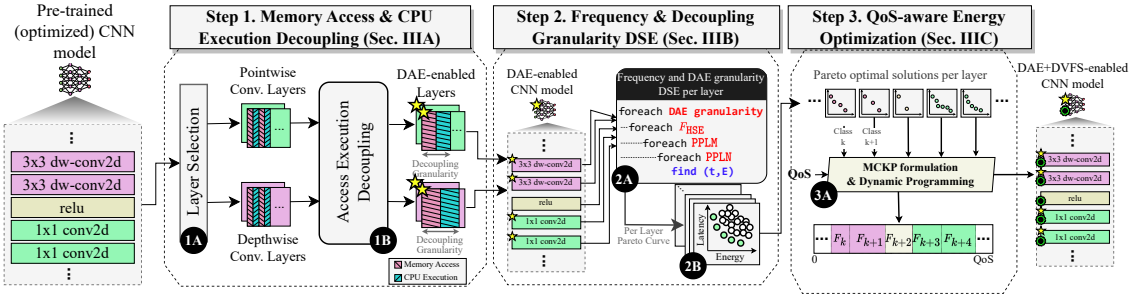

Figure 7.0.1: Overview of the proposed methodology.

Symbol	Parameter	Conditions	Min	Max	Unit
t_{lock}	PLL lock time	VCO freq = 192 MHz	75	200	μs
		VCO freq = 432 MHz	100	300	

Table 7.1: Clock startup time values depending on VCO - Datasheet reference

An analysis was performed on phase-locked loop parameters for the STM32 boards. Three main performance metrics were considered:

- Clock startup time
- PLL lock time
- PLL power consumption

The clock startup time is a metric defines as the latency required to recalibrate the board parameters when frequency switching occurs. It encompasses the clock source switching latencyas well as the recalibration of the flash wait states and voltage regulator. When the PLL is utilized, it also includes the time to modify the registers that contain the PLL parameters.

The PLL lock time is the metric that quantifies the overhead induced by the analog PLL circuit, in order to achieve synchronization. It refers to the duration it takes for a Phase-Locked Loop (PLL) circuit to synchronize its output frequency with the desired or reference frequency. In other words, it is the time required for the PLL to lock onto the input signal and establish a stable and consistent output frequency that matches the input signal's frequency,when multiplied by the PLL parameters. During this acquisition period, the PLL circuit goes through a series of steps, including phase detection, frequency comparison, and adjustment of its internal components, such as the voltage-controlled oscillator (VCO). The PLL acquisition time is a critical parameter in applications where precise and stable clock synchronization is essential. Shorter acquisition times are generally desirable as they lead to quicker signal synchronization and reduced phase jitter, ensuring accurate and efficient operation of the PLL-controlled system.

Due to minor variance of power consumption, the optimization focused on minimizing the clock startup time of the MCU. Sensitivity analysis was performed for different clock configurations, in regards to total startup time and power consumption. It can be observed that for a given input and output frequency, the parameters PLLN and PLLM, lower multiplier values and higher divider values yield lower clock startup time.

These observations can also be partially validated through the STM32 datasheet, as referenced in Tables 7.1, 7.2

Symbol	Parameter	Conditions	Min	Max	Unit
$I_{DD(PLL)}$	PLL power consumption on VDD	VCO freq = 192 MHz	0.15	0.4	mA
		VCO freq = 432 MHz	0.45	0.75	

Table 7.2: Current consumption depending on VCO value - Datasheet reference[53]

```
for (element = 0; element < num_elements/g; element+=g) {
    //LFO for memory bound operations (Sec.IIIB)
    ClockSwitchHSE();
    //Memory Bound Segment: Load g channels from the feature maps and pack them into 16-bit buffers
    q15_t buf1,buf2,...,bufg = getColumns(ch1,ch2,...,chg);
    //HFO for computation (Sec. IIIB)
    ClockSwitchPLL(PLLM,PLLN);
    // Compute Bound Segment: Perform matrix multiplication for each column
    out = matmul(kernel, buf1, **args);
    out = matmul(kernel, buf2, **args);
    ...
    out = matmul(kernel, bufg, **args);
}
```

Listing 7.1: Simplified source code modification for enabling decoupled access-execution (DAE) in pointwise convolutions

7.0.1 Step 1: Memory Access & CPU Execution Decoupling

The first step of our proposed methodology focuses on source-code level restructuring, with the objective to **D**ecouple memory **A**ccesses from CPU **E**xecution (DAE), thus, creating memory-bound and compute-bound sub-segments within the layer's structure. DAE restructuring is a key-enabler for our strategy, as it provides more fine-grained control over when and how frequently memory accesses and computations occur (Sec. 7.0.2). This enables the application of different frequencies for extended durations and with finer control, tailored to the specific requirements of each operation, such as memory access and computation, thus, resulting in the mitigation of clock switching overhead issues. To apply the source-code level modifications, we first identify the CNN model's most computationally-intensive and time-consuming layers (1A). We focus and apply DAE (1B) on two specific layer types, i.e., *i)* depthwise and *ii)* pointwise convolutions. These layer types make up over 80% of the total number of layers found in deep lightweight CNN models, e.g. Mobilenet [3], which employ the concept of depthwise separable convolution to reduce model's size and complexity.

For instance,

Convolutional layers are known to be the most power and resource-hungry layers, thus we focus on convolutional layer optimization. Furthermore, for state-of-the-art models, such as mobilenet-like

```
for (channel = 0; channel < in_channels/g; channel+=g) {
    //LFO for memory bound operations (Sec.IIIB)
    ClockSwitchHSE(hse);
    //Memory Bound Segment: Load g channels from the feature maps
    q15_t buf1,...,bufg = getChannels(ch1,...,chg);
    //HFO for computation (Sec. IIIB)
    ClockSwitchPLL(pllm,plln,hse);
    // Compute Bound Segment: Perform depthwise convolution for each channel
    convolve_depthwise(kernel, buf1, **args);
    convolve_depthwise(kernel, buf2, **args);
    ...
    convolve_depthwise(kernel, bufg, **args);
}
```

Listing 7.2: Simplified source code modification for enabling decoupled access-execution (DAE) in depthwise convolutions

models, the computation overhead mostly occurs on (i) pointwise and (ii) depthwise convolutions [22]. In order to achieve the maximum exploitation of the convolutional kernels for DVFS with minimal switching overhead, decoupled access execution is performed with configurable granularities.

Depthwise Convolutions: Depthwise convolution is a specialized CNN operation where each input channel is convolved with a separate learnable filter, capturing spatial features per channel. Typically, CNNs gradually learn increasingly complex features, each one represented by a different channel. For instance, in an image, the initial 3 input channels (RGB) increase as the network processes the image to extract and represent more complex features, such as textures, specific objects and others. State-of-the-art frameworks, such as CMSIS-NN [14] and TinyEngine [22] implement a per-channel computation approach for depthwise convolutions. In contrast, our DAE approach introduces a parametric unrolling factor called the "decoupling granularity", denoted by g. This factor determines the number of channels that are buffered in cache memory before the convolution operation is executed on each of them, thus, separating the code into memory-bound and compute-bound subsegments. Listing 7.2 provides a simplified code snippet that illustrates the practical implementation of DAE optimization, showing how the decoupling granularity enables the efficient execution of depthwise convolutions. For example, when $g = 4$, four channels are fetched in the MCU's cache memory before proceeding to the actual computation. This division enables the application of different clock frequencies per segment, which we further discuss in Sec. 7.0.2.

Pointwise Convolutions: Depthwise convolution is often followed by pointwise convolution to perform channel-wise mixing and dimensionality reduction, thus, reducing model size and computational complexity while maintaining performance. Pointwise convolutions involve `1x1` kernel sizes and are applied to each element within input channels. CMSIS-NN[14] and TinyEngine [22] implement pointwise convolutions in a per-column manner. Each column consists of one element per input channel. Our approach performs decoupling on the per channel memory accesses, thus splitting the code segment into memory and compute-bound regions. Just as in the depthwise convolution, we introduce the concept of the decoupling granularity, denoted as g, for modular buffering support w.r.t. the number of columns fetched from memory. For instance, for a 8x8x3 input image and a 1x1x3 kernel, g columns are loaded into the cache before performing computation for each one, in contrast with TinyEngine and CMSIS-NN, which load a single 1x1x3 image column at a time. In general, integrating multiple buffers leads to the generation of larger memory/compute-bound regions, thus we can minimize the frequency switching overhead while avoiding high power consumption. However, very high buffer size can lead the cache misses to skyrocket resulting in performance degradation.

Listing 7.1 provides a simplified overview of decoupled access-execute for pointwise convolutions.

After the DAE phase is performed, the DAE-enabled CNN layers are encapsulated with a configurable decoupling granularity factor for the layer unrolling. The modified DAE-enabled CNN model is propagated to the next step of our proposed methodology (Step 2.) for effective exploration and configuration of the DAE granularity factor, alongside with the DVFS parameters exploration.

7.0.2 Step 2: DAE and Clocking Co-exploration

At this step, we analyze the performance and energy consumption in a per-layer manner(2A) considering the interplay effects of DAE and clocking scheme configurations. To measure the energy consumption and performance of each layer, we have developed and integrated a custom run-time monitoring mechanism for supporting per-layer monitoring and profiling. Our mechanism relies on the on-board timers of the target MCU, which are triggered in-between the layers' code segments. Furthermore, we take advantage of STM32 MCUs integrated support for power sampling and we monitor the power consumption prior and after the DVFS integration on every CNN layer. The power and performance metrics for the DVFS in each layer are aggregated and utilized for the design space exploration for DAE and clocking configuration.

More specifically, we co-explore and gain insights on the design space defined by the following three key parameters: *i)* the decoupling granularity factor g; *ii)* the clock frequency of the `SYSCLK` clock; and *iii)* the selection of parameters for the `PLL` module (namely `PLLM` and `PLLN`), both described in Sec. **??**.

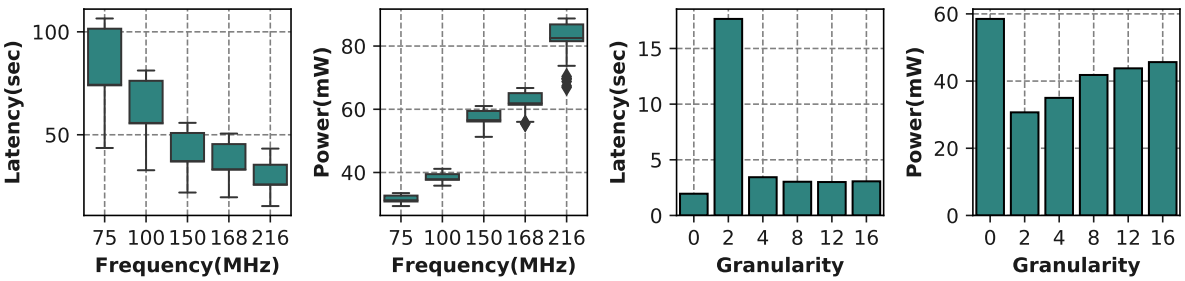

Figure 7.0.2: Impact of different DAE and clocking configurations on latency and power of depthwise and pointwise layers.

Regarding the decoupling granularity, determining the most suitable value per layer depends on both board-related specifications (e.g., cache size) as well as code-related characteristics (e.g., number of output channels and kernel size). In our case, we examine six different values, i.e., $g \in \{0, 2, 4, 8, 12, 16\}$, where $g = 0$ indicates no DAE optimization and corresponds to the default input model .

Regarding the different clock alternatives, we define two parametric operating modes, namely the *Low Frequency Operation* (LFO) and the *High Frequency Operation* (HFO). LFO exclusively employs the `HSE` clock source at a predefined frequency (50MHz) and aims to reduce power consumption on the board. In contrast, HFO configures the system's clock using the `PLL` circuit, where the final `SYSCLK` value is determined by varying combinations of the `PLLN` and `PLLM` parameters, specifically `PLLN` $\in \{75, 100, 150, 168, 216, 336, 432\}$ and `PLLM` $\in \{25, 50\}$. This distinction between the two operating modes allows us to quickly transition between different `SYSCLK` frequencies, thus, mitigates susceptibility to the switching overhead associated with the `PLL`, as elaborated in Sec. 5.2.2. Overall, DVFS switching is performed between the memory (Lst. 7.2:3) and compute (Lst. 7.2:7) bound regions , in order to exploit the decoupled access-execution transformation optimally, with LFO applied to the memory-bound and HFO to the compute-bound subsegments respectively. A similar approach is followed in pointwise convolution layers.

DSE Insights: Figure 7.0.2 shows the impact of varying operating frequencies (left) and granularity factors g (right) on the layer latency and power consumption. First, we observe that as the number of operating frequency increases, the power consumption is traded for better performance, thus composing the design space. Moreover, changing values of the granularity factor can provide significant variation on the latency and the power consumption. For instance, power consumption can drop down to 54.2% compared to the initial execution. Thus, the effective co-exploration of granularity g and frequency leads to power/latency trade-offs. The result of the DSE is a solution space per layer, where each solution trade-offs between performance and power consumption (2B). In this space, we select the Pareto optimal points, to be propagated to Step 3.

7.0.3 Step 3: QoS-aware Energy Optimization

In the final stage, we determine the optimal frequencies for each layer within the CNN, aiming to minimize the model's total energy consumption while satisfying a predefined latency budget (QoS). We denote the set of all possible frequencies generated either through the `PLL` or the `HSE` clock as F and the set of all possible granularity factors as G. Let n be the total number of layers of the CNN model and $P_k = \{..., p_j^k = \{t_j^k, E_j^k\}, ...\}, k \in \{1, ..., n\}, j \in \{1, ..., |P_k|\}$ be the set of Pareto optimal solutions (from Step 2) of layer k, where t_j^k and E_j^k denote the latency and the energy consumption of the j^{th} pareto optimal solution for layer k when operating with DVFS enabled with an HFO frequency $f \in F$ and granularity $g \in G$. We consider the minimization of the overall energy E of the CNN deployed on the target STM32 MCU, so that the overall execution time T does not go beyond a user-defined QoS. Then, the target optimization problem can be formulated as follows:

$$\text{minimize} \quad E = \sum_{k=1}^{n} \sum_{j \in P_k} E_j^k x_{kj} \tag{7.0.1}$$

$$\text{s.t.} \quad T = \sum_{k=1}^{n} \sum_{j \in P_k} t_j^k x_{kj} \leq QoS \tag{7.0.2}$$

$$\sum_{j \in P_k} x_{kj} = 1, \quad k = 1, ..., n \tag{7.0.3}$$

$$x_{kj} \in 0, 1, \qquad k = 1, ..., n, j \in P_k. \tag{7.0.4}$$

We model our problem according to the Multiple-Choice Knapsack Problem (MCKP) [35], which extends the classical knapsack problem by categorizing items into distinct classes(3A). In this formulation, the binary decision of including an item is replaced by the selection of precisely one item from each class and the goal is to maximize the value of items included in the knapsack while not exceeding its size. In our case, each individual class represents the various Pareto optimal solutions p_j^k per layer k. Each item in the class is characterized by its own value (i.e., energy consumption E_j^k) and size (i.e., latency t_j^k). Our goal is to minimize the overall energy consumption (E) while adhering to the size constraint ($T < QoS$). We convert our minimization objective to a maximization one using the transformation found in [35]. Last, we solve the optimization problem using a pseudo-polynomial time solution based on a dynamic programming (DP) approach.

Chapter 8

Experimental Evaluation

8.1 Experimental Setup and Evaluation

Experimental Setup: Our experimental evaluation is conducted on an STM32F767ZI Nucleo board, equipped with an ARM Cortex M7 CPU featuring a 16KB L1-cache. The board incorporates a High-Speed External (`HSE`) clock, ranging from 1MHz to 50MHz. For power consumption monitoring, we employed the INA219 power sensor. To mitigate potential variations arising from temperature-induced power fluctuations, we systematically compared each power measurement with the power consumption of the baseline input model at the corresponding timestamp. Our proposed methodology is evaluated over three pre-trained CNN models, namely Visual Wake Words (`VWW`), Person Detection (`PD`), and Mobilenet-V2 (`MBV2`), derived from the MCUNet [22] inference library. We conduct a comparative analysis between our approach and the state-of-the-art `TinyEngine` [22], which serves as the baseline for evaluation. Our experiments are conducted in an iso-latency execution scenario, where we measure energy consumption for a certain period specified by a QoS constraint. In the case of TinyEngine, this entails the board remaining in an idle state with a constant frequency of 216MHz after an inference, until the QoS threshold is met. We also consider `TinyEngine` enhanced with clock gating, a technique designed to optimize power consumption by selectively deactivating non-utilized board clocks and the voltage regulator, thus minimizing power leakage throughout the CNN inference.

Energy Comparison and Analysis: Figure 8.1.1 presents an illustrative comparison of energy consumption between our proposed approach, and the two configurations of the `TinyEngine`: one without any optimization and the other with clock gating applied. This evaluation encompasses various Convolutional Neural Network (CNN) inference models, each subject to discrete Quality of Service (QoS) constraints set at 10% (tight), 30% (moderate), and 50% (relaxed). X-axis illustrates the QoS constraints, while Y-axis represents the normalized energy consumption. Our proposed approach surpasses both instances of the `TinyEngine`, exhibiting a reduction in energy consumption up to 25.2%. Additionally, compared to the `TinyEngine` equipped with clock gating we achieve down to 7.2% less energy consumption. Furthermore, our observations indicate that relaxing the QoS constraints can lead to a notable reduction in energy consumption, albeit at the cost of some performance trade-offs. For instance, when examining the Mobilenet-V2 model, the energy consumption of our approach under a relaxed 50% QoS constraint decreases to 20.4% compared to the stringent 10% constraint.

Frequency Scaling Analysis: Figure 8.1.2 illustrates the HFO for each examined CNN. X axis indicates the corresponding layer type as the CNN execution proceeds and the granularities selected for 10% and 50% QoS constraint, respectively, while Y axis shows the operating frequency per layer. The LFO configuration at 50MHz of the memory-bound segment of each layer is excluded for simplicity. The observations derived are the following. Firstly, the operational frequency is configured to maximum (216MHz) mostly for performing pointwise convolutions, i.e. 58.8% against 21.4% for depthwise convolutions. The latter are less compute-intensive, thus decreasing the operational frequency will

not lead to significant performance degradation. Furthermore, the 46.1% of the pointwise convolutions and 43.4% of the depthwise convolutions are executed over the lowest operating frequencies, i.e. 75MHz and 100MHz, aiming to boost the optimization objective of power minimization. Last but not least, we investigate the impact of QoS constraints on the operating frequency. Our experiments indicate that 18.6% more layers are operating at 216MHz for tight constraints (10%). Regarding the granularity analysis, for the relaxed QoS(50%), 22.3% more layers operate with granularity factor 16, compared to 10% constraint. This is due to the fact that there is higher space to trade latency, thus the computation-bound parts are split to bigger segments, aiming to minimize switching overhead and provide power reduction.

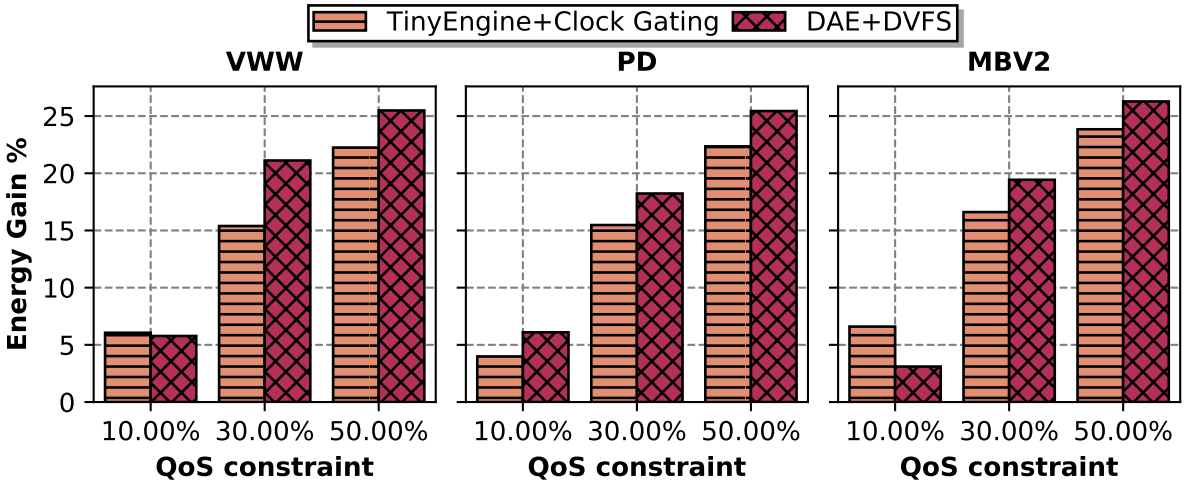

Figure 8.1.1: Energy consumption gains of our approach over the TinyEngine [22] baseline. We compare against TinyEngine with Clock Gating over the examined CNN models

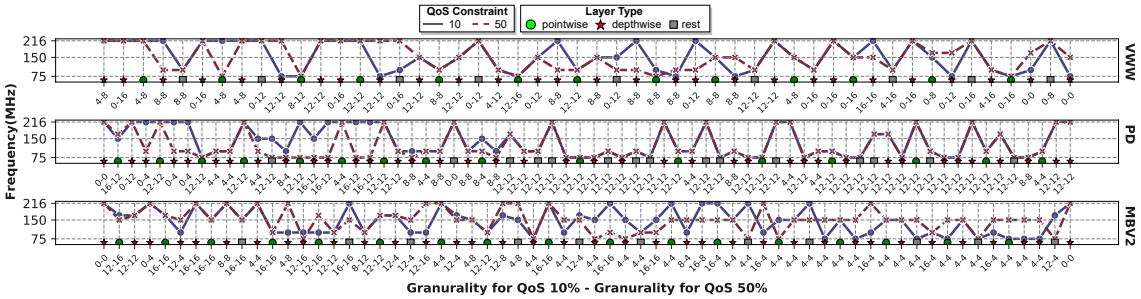

Figure 8.1.2: Frequency distribution throughout layer progression over the examined CNN models for 10% and 50% QoS constraints.

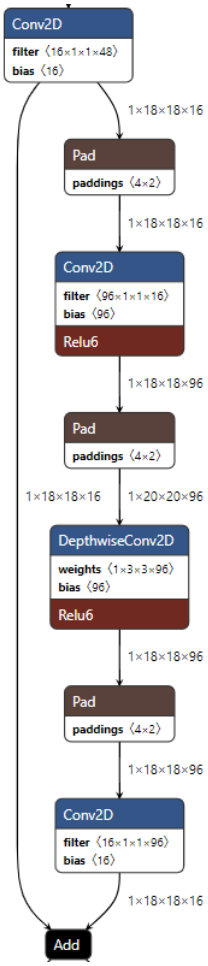

Figure 8.1.3: Example building block of the MobilenetV2 baseline model.

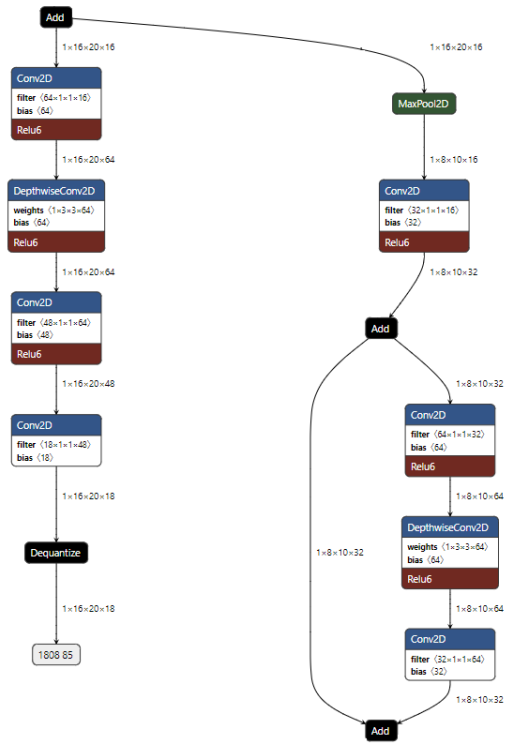

Figure 8.1.4: Example of the MCUNet-generated VWW architecture

Chapter 9

Conclusion and Future Work

9.1 Conclusion

In this work, we present a novel end-to-end methodology that exploits DVFS for optimizing the energy consumption of CNN inference on low-end STM32 MCUs. Our approach strongly leverages Decoupled Access Execute techniques to discretize memory-bound and compute-bound layer segments. This approach also utilizes optimization techniques to yield feasible results for tight latency constraints.It outperforms existing state-of-the-art approaches by achieving up to 25.2% less energy consumption.

9.2 Future Work

This work could be further optimized to improve its scalability, versatility and efficiency. Through these extensions, we aim to make DVFS accessible for heterogeneous low-end MCUs, as well as explore model-hardware co-design for tinyML with dynamic voltage frequency scaling capabilities. Some potential optimizations include:

- Finer DVFS granularity, including non-uniform in kernel frequency selection and adaptive DVFS selection.
- Integration of diverse heuristics for layer selection.
- On-device DVFS strategy selection, to reduce the need for an external runtime monitoring framework.
- Deployment automation for diverse MCUs and integration of the decoupled access execute optimization into a compiler pass.
- DVFS and Decoupled Access Execute aware Neural Architecture search, to generate optimal models for DVFS.

Bibliography

[1] E. Li, Z. Zhou, and X. Chen, "Edge intelligence: On-demand deep learning model co-inference with device-edge synergy," in *Proceedings of the 2018 Workshop on Mobile Edge Communications*, 2018, pp. 31–36.

[2] G. Ananthanarayanan, P. Bahl, P. Bodík, *et al.*, "Real-time video analytics: The killer app for edge computing," *computer*, vol. 50, no. 10, pp. 58–67, 2017.

[3] A. G. Howard, M. Zhu, B. Chen, *et al.*, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint*, 2017.

[4] *Amazon sagemaker edge*,

[5] *Google edge tpu*,

[6] H. Kwon, P. Chatarasi, V. Sarkar, T. Krishna, M. Pellauer, and A. Parashar, "Maestro: A data-centric approach to understand reuse, performance, and hardware cost of dnn mappings," *IEEE micro*, 2020.

[7] L. Mei, P. Houshmand, V. Jain, S. Giraldo, and M. Verhelst, "Zigzag: Enlarging joint architecture-mapping design space exploration for dnn accelerators," *IEEE Transactions on Computers*, 2021.

[8] C. Banbury, C. Zhou, I. Fedorov, *et al.*, "Micronets: Neural network architectures for deploying tinyml applications on commodity microcontrollers," *Proceedings of Machine Learning and Systems*, 2021.

[9] A. K. Kakolyris, M. Katsaragakis, D. Masouros, and D. Soudris, "Road-runner: Collaborative dnn partitioning and offloading on heterogeneous edge systems," in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, 2023, pp. 1–6.

[10] H. Cai, J. Lin, Y. Lin, *et al.*, "Enable deep learning on mobile devices: Methods, systems, and applications," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 2022.

[11] K. T. Chitty-Venkata and A. K. Somani, "Neural architecture search survey: A hardware perspective," *ACM Computing Surveys*, 2022.

[12] R. David, J. Duke, A. Jain, *et al.*, "Tensorflow lite micro: Embedded machine learning for tinyml systems," *Proceedings of Machine Learning and Systems*, vol. 3, pp. 800–811, 2021.

[13] *Microsoft nni*,

[14] L. Lai, N. Suda, and V. Chandra, "Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus," *arXiv preprint arXiv:1801.06601*, 2018.

[15] A. Capotondi, M. Rusci, M. Fariselli, and L. Benini, "Cmix-nn: Mixed low-precision cnn library for memory-constrained edge devices," *IEEE Transactions on Circuits and Systems II: Express Briefs*, 2020.

[16] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," *Advances in neural information processing systems*, vol. 28, 2015.

[17] J. D. De Leon and R. Atienza, "Depth pruning with auxiliary networks for tinyml," in *ICASSP 2022-2022 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, IEEE, 2022.

[18] J. Moosmann, H. Mueller, N. Zimmerman, G. Rutishauser, L. Benini, and M. Magno, "Flexible and fully quantized ultra-lightweight tinyissimoyolo for ultra-low-power edge systems," *arXiv preprint:2307.05999*, 2023.

[19] O. Spantidi and I. Anagnostopoulos, "Automated energy-efficient dnn compression under fine-grain accuracy constraints," in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2023.

[20] O. Saha, A. Kusupati, H. V. Simhadri, M. Varma, and P. Jain, "Rnnpool: Efficient non-linear pooling for ram constrained inference," *Advances in Neural Information Processing Systems*, vol. 33, pp. 20 473–20 484, 2020.

[21] E. Liberis, Ł. Dudziak, and N. D. Lane, "μNas: Constrained neural architecture search for microcontrollers," in *Proceedings of the 1st Workshop on Machine Learning and Systems*, 2021, pp. 70–79.

[22] J. Lin, W.-M. Chen, Y. Lin, C. Gan, S. Han, *et al.*, "Mcunet: Tiny deep learning on iot devices," *Advances in Neural Information Processing Systems*, vol. 33, pp. 11 711–11 722, 2020.

[23] S. Gopinath, N. Ghanathe, V. Seshadri, and R. Sharma, "Compiling kb-sized machine learning models to tiny iot devices," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 79–95.

[24] S. Jaiswal, R. K. K. Goli, A. Kumar, V. Seshadri, and R. Sharma, "Minun: Accurate ml inference on microcontrollers," in *Proceedings of the 24th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, 2023, pp. 26–39.

[25] Z. Zhao, K. M. Barijough, and A. Gerstlauer, "Deepthings: Distributed adaptive deep learning inference on resource-constrained iot edge clusters," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2348–2359, 2018.

[26] Z. Tang, Y. Wang, Q. Wang, and X. Chu, *The impact of gpu dvfs on the energy and performance of deep learning: An empirical study*, 2019. arXiv: 1905.11012 [cs.PF].

[27] E. Le Sueur and G. Heiser, "Dynamic voltage and frequency scaling: The laws of diminishing returns," in *Proceedings of the 2010 International Conference on Power Aware Computing and Systems*, ser. HotPower'10, Vancouver, BC, Canada: USENIX Association, 2010, pp. 1–8.

[28] K. Koukos, D. Black-Schaffer, V. Spiliopoulos, and S. Kaxiras, "Towards more efficient execution: A decoupled access-execute approach," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ser. ICS '13, Eugene, Oregon, USA: Association for Computing Machinery, 2013, pp. 253–262, ISBN: 9781450321303. DOI: 10.1145/2464996.2465012. [Online]. Available:

[29] A. Jimborean, K. Koukos, V. Spiliopoulos, D. Black-Schaffer, and S. Kaxiras, "Fix the code. don't tweak the hardware: A new compiler approach to voltage-frequency scaling," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '14, Orlando, FL, USA: Association for Computing Machinery, 2014, pp. 262–272, ISBN: 9781450326704. DOI: 10.1145/2581122.2544161. [Online]. Available:

[30] H. Bouzidi, M. Odema, H. Ouarnoughi, M. A. Al Faruque, and S. Niar, "Hadas: Hardware-aware dynamic neural architecture search for edge performance scaling," in *2023 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2023, pp. 1–6. DOI: 10.23919/DATE56975.2023.10137095.

[31] D. C. Snowdon12, E. Le Sueur, S. M. Petters, and G. Heiser, "A platform for os-level power management," *The European Professional Society on Computer Systems 2009*, 2009.

[32] B. Acun, K. Chandrasekar, and L. V. Kale, "Fine-grained energy efficiency using per-core dvfs with an adaptive runtime system," in *2019 Tenth International Green and Sustainable Computing Conference (IGSC)*, IEEE, 2019, pp. 1–8.

[33] R. Jejurikar, C. Pereira, and R. Gupta, "Leakage aware dynamic voltage scaling for real-time embedded systems," in *Proceedings of the 41st annual Design Automation Conference*, 2004, pp. 275–280.

[34] W. Gay, "Beginning stm32," *Beginning STM32*, 2018.

[35] H. Kellerer, U. Pferschy, D. Pisinger, H. Kellerer, U. Pferschy, and D. Pisinger, "The multiple-choice knapsack problem," *Knapsack Problems*, pp. 317–347, 2004.

[36] P. Cunningham, M. Cord, and S. J. Delany, "Supervised learning," in *Machine Learning Techniques for Multimedia: Case Studies on Organization and Retrieval*, M. Cord and P. Cunningham, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 21–49, ISBN: 978-3-540-75171-7. DOI: 10.1007/978-3-540-75171-7_2. [Online]. Available:

[37] T. Hastie, R. Tibshirani, J. Friedman, T. Hastie, R. Tibshirani, and J. Friedman, "Unsupervised learning," *The elements of statistical learning: Data mining, inference, and prediction*, pp. 485–585, 2009.

[38] S. J. Russell and P. Norvig, *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited, 2016.

[39] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, pp. 115–133, 1943.

[40] A. Jain, J. Mao, and K. Mohiuddin, "Artificial neural networks: A tutorial," *Computer*, vol. 29, no. 3, pp. 31–44, 1996. DOI: 10.1109/2.485891.

[41] A. F. Agarap, "Deep learning using rectified linear units (relu)," *arXiv preprint arXiv:1803.08375*, 2018.

[42] B. Xu, N. Wang, T. Chen, and M. Li, "Empirical evaluation of rectified activations in convolutional network," *CoRR*, vol. abs/1505.00853, 2015. arXiv: 1505.00853. [Online]. Available:

[43] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[44] L. Alzubaidi, J. Zhang, A. J. Humaidi, *et al.*, "Review of deep learning: Concepts, CNN architectures, challenges, applications, future directions," *Journal of Big Data*, vol. 8, no. 1, Mar. 2021. DOI: 10.1186/s40537-021-00444-8. [Online]. Available:

[45] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, *Mobilenetv2: Inverted residuals and linear bottlenecks*, 2019. arXiv: 1801.04381 [cs.CV].

[46] A. G. Howard, M. Zhu, B. Chen, *et al.*, *Mobilenets: Efficient convolutional neural networks for mobile vision applications*, 2017. arXiv: 1704.04861 [cs.CV].

[47] A. Levy, B. Campbell, B. Ghena, *et al.*, "The tock embedded operating system," in *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, ser. SenSys '17, Delft, Netherlands: Association for Computing Machinery, 2017, ISBN: 9781450354592. DOI: 10.1145/3131672.3136988. [Online]. Available:

[48] E. Aras, S. Delbruel, F. Yang, W. Joosen, and D. Hughes, "Chimera: A low-power reconfigurable platform for internet of things," *ACM Trans. Internet Things*, vol. 2, no. 2, Mar. 2021. DOI: 10.1145/3440995. [Online]. Available:

[49] P. Ngatchou, A. Zarei, and A. El-Sharkawi, "Pareto multi objective optimization," in *Proceedings of the 13th International Conference on, Intelligent Systems Application to Power Systems*, 2005, pp. 84–91. DOI: 10.1109/ISAP.2005.1599245.

[50] L. Lai, N. Suda, and V. Chandra, *Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus*, 2018. arXiv: 1801.06601 [cs.NE].

[51] *Microtvm*,

[52] R. David, J. Duke, A. Jain, *et al.*, *Tensorflow lite micro: Embedded machine learning on tinyml systems*, 2021. arXiv: 2010.08678 [cs.LG].

[53] *Stm32f767zi datasheet*,