Εθνικο Μετσοβιο Πολυτεχνειο

Σχολη Ηλεκτρολογων Μηχανικων και Μηχανικων Υπολογιστων

Τομεας Τεχνολογιας Πληροφορικης και Υπολογιστων

Εργαστηριο Μικροϋπολογιστων και Ψηφιακων Συστηματων

Extending RISC-V ISA for Fine-Grained Mixed-Precision in Neural Networks

Διπλωματικη Εργασια

του

Αλέξιου Μάρα

Επιβλέπων: Δημήτριος Σούντρης
Καθηγητής Ε.Μ.Π.

Εργαστηριο Μικροϋπολογιστων και Ψηφιακων Συστηματων

Αθήνα, Μάρτιος 2024

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

Extending RISC-V ISA for Fine-Grained Mixed-Precision in Neural Networks

Διπλωματικη Εργασια

του

Αλέξιου Μάρα

Επιβλέπων: Δημήτριος Σούντρης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 26 Μαρτίου 2024.

(Υπογραφή)	*(Υπογραφή)*	*(Υπογραφή)*
.....	.....	.....
Δημήτριος Σούντρης	Παναγιώτης Τσανάκας	Σωτήριος Ξύδης
Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Επίκουρος Καθηγητής Ε.Μ.Π

Αθήνα, Μάρτιος 2024

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

(Υπογραφή)

.....

Αλεξιος Μαρας
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Το αυξανόμενο ενδιαφέρον γύρω από την ανάπτυξη εφαρμογών μηχανικής μάθησης (ML) για συσκευές με περιορισμένη υπολογιστική ισχύ και χωρητικότητα ενέργειας τονίζει την ανάγκη για την εύρεση λύσεων που θα ξεχωρίζουν όχι μόνο όσον αφορά την αποδοτικότητα τους σε ζητήματα ενέργειας και μνήμης αλλά και που θα εξασφαλίζουν χαμηλό χρόνο εκτέλεσης όταν υπάρχουν αυστηροί χρονικοί περιορισμοί. Η αρχιτεκτονική RISC-V, με το ανοιχτό σετ εντολών και τις προσαρμόσιμες επεκτάσεις της, προσφέρει έναν υποσχόμενο δρόμο για την βελτιστοποίηση των αλγορίθμων αυτών, επιτρέποντας πιο εξατομικευμένες και ενεργειακά αποδοτικές λύσεις. Επιπλέον, οι πρόσφατες πρόοδοι σε τεχνικές quantization και σε μεθοδολογιές που αξιοποιούν μεταβλητές μικτής ακρίβειας, μπορούν να συνεισφέρουν στην βελτίωση του χρόνου εκτέλεσης και στην κατανάλωση ενέργειας των νευρωνικών δικτύων (NN), χωρίς να υποβαθμίζεται σημαντικά η ακρίβεια των προβλέψεων τους.

Σε αυτή την εργασία, εκμεταλλευόμαστε αυτές τις τεχνικές, έτσι ώστε να επιταχύνουμε την εκτέλεση αλγορίθμων Βαθιών Νευρωνικών Δικτύων (DNN) πάνω σε RISC-V επεξεργαστές. Για να βελτιώσουμε ακόμη περισσότερο τα αποτελέσματα μας, θα επεκτείνουμε το σετ εντολών που υποστηρίζεται από τον επεξεργαστή και θα ενσωματώσουμε μια νέα λειτουργική μονάδα εντός του pipeline του, σχεδιασμένη αποκλειστικά για την εκτέλεση αυτών των νέων εντολών. Για τον γρήγορο σχεδιασμό πρωτοτύπων, θα υλοποιήσουμε τον επεξεργαστή πάνω σε μια πλακέτα FPGA Xilinx Virtex-7, η οποία θα επιτρέψει να αξιολογήσουμε την αποτελεσματικότητα της μεθοδολογίας μας σε διάφορες αρχιτεκτονικές Νευρωνικών Δικτύων, εκπαιδευμένα πάνω σε διαφορετικά σύνολα δεδομένων. Με μία σχετικά χαμηλή αύξηση των απαιτούμενων πόρων για την υλοποίηση του, της τάξης του 34.89% στη χρήση των Lookup Tables (LUTs) και 24.28% στα Flip-Flops (FFs), η μεθοδολογία μας καταφέρνει να επιταχύνει τον χρόνο εκτέλεσης κατά **13-23x** σε κλασικές Multi-layer Perceptron αρχιτεκτονικές, **18-28x** σε τυπικά Συνελικτικά Δίκτυα και **6-7x** σε πιο σύνθετα δίκτυα, σαν τα MobileNets, με ελάχιστη μείωση της ακρίβειας τους από **1-5%**, επιδεικνύοντας μια σημαντική βελτίωση σε σχέση με τον αρχικό επεξεργαστής.

Λέξεις Κλειδιά

RISC-V, Νευρωνικά Δίκτυα, Mixed Precision Quantization, Συσχεδιασμός Υλικού-Λογισμικού, Επιταχυντής Υλικού, FPGA.

Abstract

The growing interest in deploying machine learning (ML) applications on devices with restricted processing power and energy capacity underscores the necessity for computing solutions that not only excel in power and memory efficiency but also ensure low latency for time-sensitive applications. The RISC-V architecture, with its open-source instruction set and customizable extensions, offers a promising pathway for optimizing these algorithms by enabling more tailored and energy-efficient processing capabilities. Furthermore, recent advancements in quantization and mixed precision techniques offer significant promise for improving the run-time and energy consumption of neural networks (NN), without significantly compromising their efficiency.

In this work, we propose to leverage these advancements to expedite the inference process of Deep Neural Networks (DNNs) on RISC-V processors. To push performance even further, we plan to expand the supported instruction set and incorporate a new functional unit within the processor's pipeline, specifically designed for executing these new instructions. For rapid prototyping and design exploration, we implement the processor on a Xilinx Virtex-7 FPGA board, enabling us to assess the efficacy of our methodology across diverse Neural Network architectures and datasets. With a modest overhead of 34.89% in the usage of Lookup Tables (LUTs) and 24.28% in Flip-Flops (FFs), our framework manages to accelerate the execution time by **13-23x** in classic Multi-layer Perceptron architectures, **18-28x** in typical Convolutional Networks, and **6-7x** in more complex networks, like MobileNets, with minimal reduction in their accuracy from **1-5%**, demonstrating a significant improvement compared to the original processor.

Keywords

RISC-V, Neural Networks, Mixed Precision Quantization, Hardware-Software Co-design, Hardware Accelerator, FPGA.

Ευχαριστίες

Αρχικά θα ήθελα να ευχαριστήσω θερμά τον επιβλέποντα καθηγητή μου κύριο Δημήτριο Σούντρη που μου προσέφερε την ευκαιρία να εκπονήσω την διπλωματική μου εργασία στο εργαστήριο Μικροεπεξεργαστών και Ψηφιακών Συστημάτων. Θα ήθελα επίσης να ευχαριστήσω τον υποψήφιο διδάκτορα Γιώργο Αρμενιάκο για την πολύτιμη βοήθειά του, τις ιδέες, τις συμβουλές και τις κατευθύνσεις του καθόλη τη διάρκεια της διπλωματικής μου, καθώς και τον καθηγητή Σωτήριο Ξύδη για τις παρεμβάσεις του και τη συμβολή του σε σημαντικά σημεία του έργου. Τέλος, ευχαριστώ βαθύτατα τους γονείς μου, καθώς και τους φίλους μου για την υπομονή και την συμπαράσταση που μου έχουν προσφέρει όλα αυτά τα χρόνια.

Contents

List of Figures

List of Tables

Εκτεταμένη Περίληψη

0.1 Εισαγωγή

Οι πρόσφατες εξελίξεις στον τομέα της βαθιάς μάθησης (DL) έχουν πυροδοτήσει μια εκθετική αύξηση στην ανάπτυξη εφαρμογών και υπηρεσιών που βασιζονται σε αλγορίθμους τεχνητής νοημοσύνης (AI), από προσωπικούς βοηθούς έως και συστήματα ασφαλείας μέσω χρήσης βίντεο και ήχου. Επιπλέον, η ραγδαία εξέλιξη του mobile computing και του Internet of Things (IoT) έχουν συμβάλλει στην δημιουργία ενός δικτύου με δισεκατομμύρια κινητές συσκευές συνδεδεμένες μεταξύ τους, παράγοντας τεράστιες ποσότητες δεδομένων. Εξαιτίας αυτών των εξελίξεων, έχει γίνει επιτακτική ανάγκη να επεκταθούν οι δυνατότητες του AI στην άκρη του δικτύου (Edge) [1].

Μια ευρεία γκάμα μεθοδολογιών που εμπίπτει στο ευρύ πεδίο του AI, είναι και η Μηχανική Μάθηση (ML), η οποία έχει αναδειχθεί ως η επικρατούσα λύση στον τομέα αυτόν [2]. Παραδοσιακά, η ανάπτυξη ML μοντέλων απαιτούσε μεγάλες ποσότητες ενέργειας, καθώς και σημαντικούς υπολογιστικούς πόρους, προκειμένου να επιτευχθεί το επιθυμητό επίπεδο ακρίβειας. Ωστόσο, η εξέλιξη τεχνολογιών όπως το IoT και το Edge Computing, έχουν διεγείρει το ενδιαφέρον γύρω από την προσαρμογή τεχνικών ML πάνω σε ενσωματωμένα συστήματα και συσκευές με περιορισμένους πόρους [3]. Πιο συγκεκριμένα το Tiny Machine Learning (TinyML) συνιστά την σύγκλιση του ML (ειδικότερα, του DL) με το Edge Computing. Το TinyML διευκολύνει την ανάπτυξη συμπαγών μοντέλων DL σε μικρές συσκευές με περιορισμένη υπολογιστική ισχύ (οι συχνότητες ρολογιού κυμαίνονται συνήθως στις δεκάδες megahertz), ελάχιστη χωρητικότητα μνήμης, και μόλις μερικά milliwatts (mW) ισχύος [4].

Επιπλέον, τα τελευταία χρόνια έχει παρατηρηθεί μία έκρηξη στην δημοφιλία των RISC-V αρχιτεκτονικών ως μία εναλλακτική λύση έναντι των πιο παραδοσιακών επεξεργαστών. Το ανοιχτού κώδικα σετ εντολών (ISA) του, καθώς και η δυνατότητα που προσφέρει στους χρήστες να μπορούν να προσαρμόσουν το υλικό του επεξεργαστή και τις εντολές που υποστηρίζει, σύμφωνα με τις ανάγκες του προβλήματος τους, έχουν θέσει τους RISC-V επεξεργαστές ως ισχυρούς ανταγωνιστές έναντι των αντίστοιχων λύσεων που προσφέρουν κολοσσοί, όπως η Intel, AMD και ARM.

Στην διπλωματική αυτή, εκμεταλλευόμενοι το ευέλικτο περιβάλλον του RISC-V, θα προτείνουμε ένα ολοκληρωμένο πλαίσιο για την προσαρμογή Βαθιών Νευρωνικών Δικτύων (DNN)

πάνω σε τέτοιους επεξεργαστές. Κύριος στόχος μας είναι η βελτιστοποίηση τους, όσον αφορά τον χρόνο εκτέλεσης τους, τις απαιτήσεις τους σε μνήμη και την συνολική κατανάλωση ενέργειας σε σύγκριση με τον αρχικό επεξεργαστή.

0.2 Σχετική Βιβλιογραφία

Πολλά επιτυχημένα εμπορικά εργαλεία, όπως οι βιβλιοθήκες CMSIS-NN από την ARM [5], η TensorFlow Lite (TFLite Micro) από τη Google [6], και η X-CUBE-AI από την STMicroelectronics [7], χρησιμοποιούνται ευρέως για την ενσωμάτωση αλγορίθμων μηχανικής μάθησης σε μικροελεγκτές (MCUs). Αυτά τα εργαλεία προσφέρουν μια ποικιλία τεχνικών βελτιστοποίησης (quantization, weight pruning) καθώς και βελτιστοποιημένες υλοποιήσεις σε συναρτήσεις για διάφορους τύπους Νευρωνικών, με στόχο την συμπίεση του μεγέθους των μοντέλων και την μείωση του χρόνου εκτέλεσης τους. Το MCUNet είναι ένα άλλη μία state-of-the-art εργασία, η οποία παρέχει μια σύνθετη προσέγγιση που εκτός από την αναζήτηση της καλύτερης δυνατής αρχιτεκτονικής των δικτύων, βάσει συγκεκριμένων προδιαγραφών, υλοποιεί και ακόμα πιο εξειδικευμένες συναρτήσεις για τη μεγιστοποίηση της απόδοσης και της αποδοτικότητας τους σε συστήματα IoT [8]. Ωστόσο, αυτές οι εργασίες επικεντρώνονται κυρίως σε μεθοδολογίες για την βελτιστοποίηση της ροής των δεδομένων και την καλύτερη εκμετάλλευση της ιεραρχία μνήμης.

Με στόχο την μεγιστοποίηση της απόδοσης, αρκετές μελέτες [9], [10] παρουσιάζουν αφιερωμένους επιταχυντές για DNNs. Παρόλα αυτά, η συγκεκριμένη προσέγγιση δεν εμφανίζει την ίδια ευελιξία με άλλες λύσεις, καθώς δεν παρέχεται πολλές φορές επαρκής υποστήριξη για την κατάλληλη συνεργασία με μικροελεγκτές που βρίσκονται στο Edge. Τα τελευταία χρόνια, με την άνοδο των RISC-V επεξεργαστών στο προσκήνιο, πολλοί ερευνητές έχουν στραφεί προς αυτούς για την επιτάχυνση Βαθιών Νευρωνικών Δικτύων. Κάποιες αξιόλογες εργασίες που βασίζονται στην ενσωμάτωση νέων εντολών περιλαμβάνουν το FANN-on-MCU [11], το RedMulE [12] και το Dory [13], , οι οποίες στοχεύουν στην αποτελεσματική εκτέλεση τέτοιων μοντέλων σε RISC-V επεξεργαστές της πλατφόρμας PULP. Ο κύριος περιορισμός τους είναι το ότι δεν υποστηρίζουν την χρήση μεταβλητών χαμηλής ακρίβειας (κάτω από 8-bits) και επομένως δεν αξιοποιούν τις αυξημένες υπολογιστικές δυνατότητες που προσφέρουν. Από την άλλη, έρευνες όπως το PULP-NN [14] και η προέκταση του XpulpNN [15] επιτρέπουν την χρήση παραμέτρων χαμηλότερης ακρίβειας, πετυχαίνοντας θεαματικά αποτελέσματα, χωρίς ωστόσο να προβλέπουν την υλοποίηση πράξεων μεταξύ μεταβλητών που αναπαριστώνται με διαφορετικό αριθμό από bits.

Πολλές από τις σύγχρονες αρχιτεκτονικές CPU και GPU εμφανίζουν περιορισμένες δυνατότητες για νευρωνικά δίκτυα μικτής ακρίβειας, καθώς είναι πρωτίστως διαμορφωμένες για τύπους δεδομένων των 8, 16, και 32-bits. Ως αποτέλεσμα, η πλειονότητα των επιταχυντών και των εργαλείων που χρησιμοποιούνται για την εκπαίδευση και την ανάπτυξη εφαρμογών με νευρωνικά δίκτυα είναι προσαρμοσμένα προς υλοποιήσεις που χρησιμοποιούν μορφές υψηλότερης ακρίβειας, προσφέροντας μονάχα μερικές γενικές στρατηγικές για την ενσωμάτωση

τεχνικών μικτής ακρίβειας [16], [17]. Υπάρχουν ορισμένες μελέτες, οι οποίες δίνουν περαιτέρω έμφαση στη βελτιστοποιήση DNN, χρησιμοποιώντας υπολογισμούς μικτής ακρίβειας, όπως ο BARVINN [18], ο οποιός είναι ένας επιταχυντής υλικού, υλοποιημένος πάνω σε ένα FPGA και του οποίου η λειτουργία ρυθμίζεται από έναν RISC-V επεξεργαστή, και ο Dustin [19], ο οποίος επεκτείνει το RISC-V ISA, προκειμένου να είναι δυνατή η εκτέλεση πράξεων μεταξύ τέτοιων μεταβλητών. Παρόλα αυτά, τέτοιου είδους προσεγγίσεις, σπανίως μεριμνούν για την συνολική κατανάλωση πόρων και το χώρο που απαιτείται, ώστε να υλοποιηθούν τα συστήματα τους.

0.3 Επισκόπηση Βασικών Εννοιών

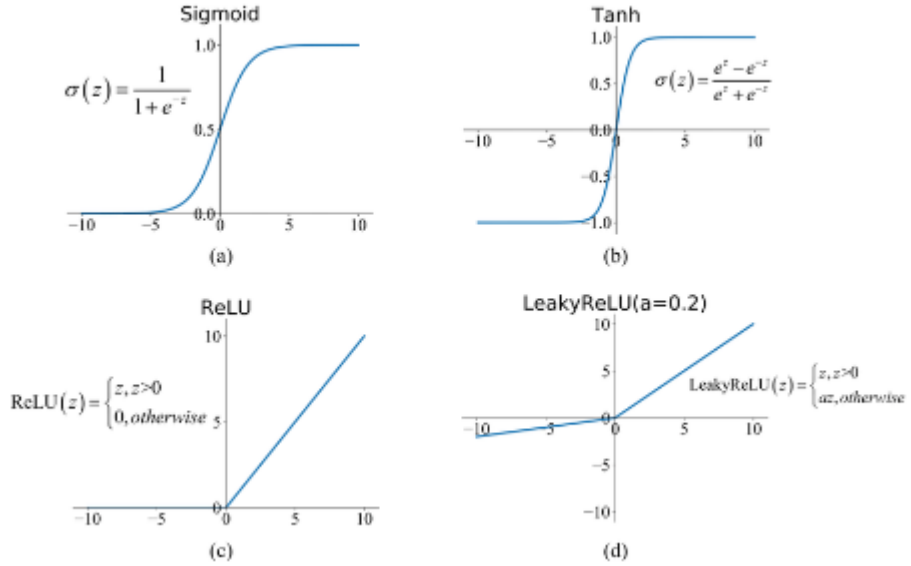
0.3.1 Βαθιά Νευρωνικά Δίκτυα

Η Μηχανική Μάθηση (ML) είναι μία υποκατηγορία της Τεχνητής Νοημοσύνης που περιλαμβάνει την ανάπτυξη αλγορίθμων και στατιστικών μοντέλων που επιτρέπουν σε υπολογιστικά συστήματα να βελτιώσουν την απόδοσή τους σε διάφορα προβλήματα μέσω της εμπειρίας. Τα μοντέλα αυτά σχεδιάζονται με στόχο να μαθαίνουν από δεδομένα και να παίρνουν αποφάσεις χωρίς να λαμβάνουν ρητές οδηγίες. Συνδυάζοντας στοιχεία που προέρχονται από τους τομείς της πληροφορικής, της στατιστικής και αξιοποιώντας πληροφορίες από το εκάστοτε πεδίο γνώσεων, δύναται να δημιουργήσει αλγόριθμους που όχι μόνο είναι ικανοί να αποκρυπτογραφούν περίπλοκα μοτίβα μέσα στα δεδομένα αλλά και να προσαρμόζονται και να βελτιώνονται με τον καιρό. Η Βαθιά Μάθηση (DL) αποτελεί έναν υποτομέα του ML που επικεντρώνεται στη χρήση τεχνητών νευρωνικών δικτύων (ANNs) που χαρακτηρίζονται από πολλαπλά στρώματα [20].

Τεχνητά Νευρωνικά Δίκτυα

Τα Τεχνητά Νευρωνικά Δίκτυα είναι υπολογιστικά μοντέλα που έχουν εμπνευστεί από τη δομή του ανθρώπινου εγκεφάλου. Το κύριο δομικό στοιχείο των ANNs ονομάζεται νευρώνας και ουσιαστικά επιχειρεί να μιμηθεί τη λειτουργικότητα του αντίστοιχου βιολογικού κυττάρου. Οι νευρώνες οργανώνονται σε στρώματα, με ένα εισαγωγικό στρώμα που λαμβάνει τα δεδομένα εισόδου, ένα ή περισσότερα κρυφά επίπεδα που τα επεξεργάζονται, και ένα στρώμα εξόδου που παράγει την τελική πρόβλεψη. Οι κόμβοι μεταξύ διαδοχικών στρωμάτων συνδέονται μεταξύ τους μέσω συνάψεων που ονομάζονται βάρη, προκειμένου να σχηματιστεί το τελικό Δίκτυο.

Μία από τις πιο θεμελιώδεις λειτουργίες των ANNs είναι η εφαρμογή συναρτήσεων ενεργοποίησης στις εξόδους του κάθε νευρώνα. Η χρήση τους αποσκοπεί στην εισαγωγή μίας μη-γραμμικότητας στον τρόπο λειτουργίας των μοντέλων, το οποίο έχει ως απόρροια την ανίχνευση πιο πολύπλοκων σχέσεων μέσα στα δεδομένα ανάλυσης. Οι πιο συνηθισμένες συναρτήσεις ενεργοποίησης φαίνονται στο Σχήμα 0.1.



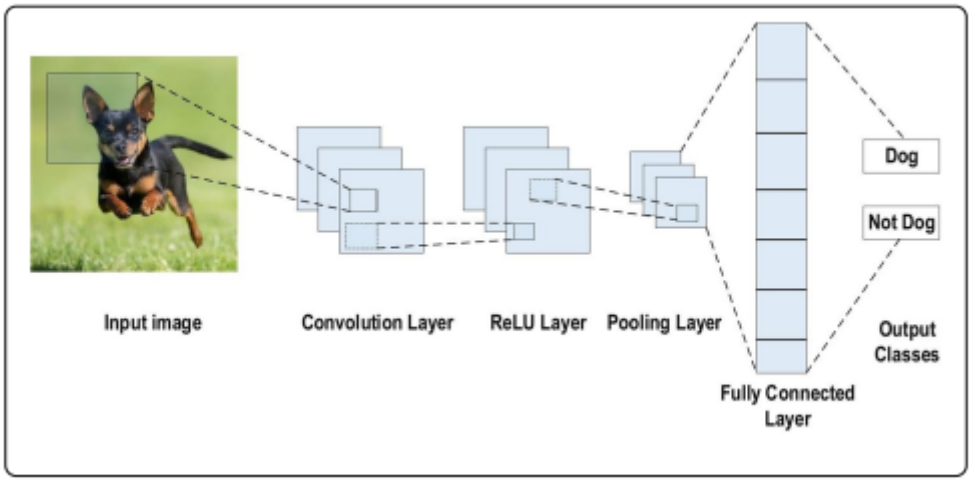
Σχήμα 0.1: Οι γραφικές παραστάσεις των συναρτήσεων ενεργοποίησης : (a) Sigmoid, (b) Tahn, (c) ReLU, και (d) Leaky ReLU.
Πηγή: [21]

Multi-layer Perceptrons

Τα Multi-layer Perceptrons (MLP) αποτελούν τη βασική αρχιτεκτονική ενός ANN και κατασκευάζονται από διαδοχικά πλήρως συνδεδεμένα στρώματα νευρώνων. Μέσα σε αυτά τα στρώματα, κάθε κόμβος είναι διασυνδεδεμένος με κάθε κόμβο του αμέσως επόμενου επιπέδου, με αποτέλεσμα την δημιουργία ένα πυκνά συνδεδεμένου δικτύου. Αυτό επιτρέπει στα MLP να αναγνωρίζουν περίπλοκες σχέσεις και μοτίβα μέσα στα δεδομένα, καθιστώντας τα ιδιαίτερα προσαρμοστικά και ευρέως εφαρμόσιμα σε διάφορους τομείς. Ωστόσο, συχνά αντιμετωπίζουν δυσκολίες κατά την επεξεργασία δεδομένων που εμφανίζουν σειριακές ή χωρικές εξαρτήσεις. Αυτός ο περιορισμός έχει ωθήσει τους ερευνητές να αναπτύξουν πιο εξειδικευμένες αρχιτεκτονικές σχεδιασμένες ειδικά για τη διαχείριση αυτών των τύπων δεδομένων [22].

Συνελικτικά Νευρωνικά Δίκτυα

Τα Συνελικτικά Νευρωνικά Δίκτυα (CNNs) είναι αλγόριθμοι εμπνευσμένοι από την οργάνωση του ανθρώπινου οπτικού συστήματος και έχουν προσαρμοστεί για την επεξεργασία οπτικών δεδομένων. Τα μοντέλα αυτά εκμεταλλεύονται αρχές από τη γραμμική άλγεβρα, και ιδιαίτερα τις ιδιότητες της συνέλιξης, για να εξάγουν χαρακτηριστικά και να αναγνωρίζουν μοτίβα μέσα στα οπτικά δεδομένα. Εξαιτίας της αποτελεσματικότητας τους στον εντοπισμό αντικειμένων, χρησιμοποιούνται κυρίως σε εφαρμογές της επιστήμης της όρασης υπολογιστών, όπως είναι η αναγνώριση εικόνας, και μπορούν να αξιοποιηθούν μέχρι και σε πολύ σύνθετα συστήματα όπως αυτά των αυτοκινήτων αυτόνομης οδήγησης και στην ανάλυση ιατρικών εικόνων [20]. Η τυπική δομή ενός CNN φαίνεται στο Σχήμα 0.2.



Σχήμα 0.2: Αρχιτεκτονική ενός τυπικού Συνελικτικού Δικτύου.
Πηγή: [20]

0.3.2 Quantization

Οι σημαντικές υπολογιστικές απαιτήσεις που είναι εγγενείς σε εφαρμογές Νευρωνικών Δικτύων, ειδικότερα όταν αυτά εκτελούνται σε ηλεκτρονικές συσκευές με περιορισμένους υπολογιστικούς πόρους. Ως απάντηση σε αυτήν την πρόκληση, η κβάντιση (quantization) των παραμέτρων τους αναδύεται ως μια στρατηγική μεθοδολογία με στόχο τη συμπίεση των μοντέλων και την μείωση του υπολογιστικού φόρτου. Ωστόσο, είναι κρίσιμο να αναγνωρίσουμε ότι η τεχνική αυτή εισάγει ορισμένους συμβιβασμούς, καθώς μπορεί να οδηγήσει σε μείωση της ακρίβειας σε σύγκριση με το αρχικό μοντέλο. Στην διαδικασία αυτή, οι παράμετροι του δικτύου, τόσο τα βάρη όσο και οι έξοδοι του κάθε επιπέδου, υπόκεινται σε μετατροπή σε μεταβλητές που χρησιμοποιούν λιγότερα bits από τα συμβατικά 16 ή 32 που χρησιμοποιούνται κατά την εκπαίδευση [23]. Τα κυριότερα πλεονεκτήματα της μετατροπής σε ακέραιες τιμές είναι η ελαχιστοποίηση του χρόνου εκτέλεσης και οι μειωμένες απαιτήσεις σε μνήμη και ενέργεια.

Mixed Precision Quantization

Η μείωση της ακρίβειας των μεταβλητών, σε τιμές που απαιτούν λιγότερο από 8 bits για την αναπαράσταση τους, μπορεί να ενισχύσει σημαντικά την απόδοση του υλικού. Ωστόσο, η κβάντιση ολόκληρου του δικτύου σε εξαιρετικά χαμηλή ακρίβεια μπορεί να οδηγήσει σε σημαντική μείωση της απόδοσης του, καθιστώντας την προσέγγιση αυτή ακατάλληλη για ορισμένες κρίσιμες εφαρμογές. Ένας κομψός τρόπος για την αντιμετώπιση αυτού του προβλήματος είναι να εφαρμοστεί ένα σχήμα Mixed Precision Quantization. Με αυτήν την φιλική προς το υλικό προσέγγιση, τα βάρη και οι έξοδοι του κάθε επιπέδου μετατρέπονται σε αριθμούς διαφορετικής ακρίβειας. Έτσι, μπορούμε να επωφεληθούμε από τα χαμηλότερα bit widths (χαμηλότερη καθυστέρηση, μειωμένη κατανάλωση ενέργειας και μικρότερο αποτύπωμα μνήμης), ελαχιστοποιώντας όσο το δυνατόν περισσότερο την μείωση της ακρίβειας των προβλέψεων του αρχικού μοντέλου.

Μία από τις κύριες προκλήσεις που συνδέονται με αυτήν την τεχνική είναι η καθορισμός της βέλτιστης διαμόρφωσης για την ακρίβεια των παραμέτρων σε κάθε στρώμα. Καθώς ο χώρος αναζήτησης διευρύνεται εκθετικά με τον αριθμό των επιπέδων σε ένα δίκτυο, η εξαντλητική αναζήτηση όλων των λύσεων είναι μία αρκετά χρονοβόρα διαδικασία, ιδιαίτερα σε πολυεπίπεδα μοντέλα. Ορισμένες λύσεις για το ζήτημα αυτό περιλαμβάνουν την χρήση μαθηματικών μοντέλων για να εξακριβωθεί το πόσο ευαίσθητα είναι τα επίπεδα στην διαδικασία του quantization [24]. Έτσι, επίπεδα που χαρακτηρίζονται από υψηλή ευαισθησία θα χρησιμοποιούν περισσότερα bits, ενώ αντιθέτως εκείνα με χαμηλή ευαισθησία μπορούν να αξιοποιούν παραμέτρους μικρότερης ακρίβειας. Μία εναλλακτική προσέγγιση είναι η στρατηγική μείωση των διαθέσιμων επιλογών προς ανάλυση.

Fine Tuning

Η προσαρμογή των παραμέτρων σε ένα Νευρωνικό Δίκτυο μετά την κβάντιση είναι μία συχνά απαραίτητη διαδικασία για την επαναφορά της απόδοσης του στα αρχικά επίπεδα. Για την επίτευξη αυτού του στόχου, οι 2 μεθοδολογίες που εφαρμόζονται είναι το Quantization-Aware Training (QAT) και το Post-Training Quantization (PTQ). Στην πρώτη περίπτωση, το μοντέλο υπόκειται σε μία επιπλέον εκπαίδευση, στην οποία οι παράμετροι κβάντισης λαμβάνονται υπόψη και αναπροσαρμόζονται, το οποίο έχει ως αποτέλεσμα ένα πιο αποδοτικό κβαντισμένο μοντέλο. Παρόλα αυτά, η διαδικασία αυτή διακρίνεται από τις υψηλές υπολογιστικές της απαιτήσεις, καθώς και από την μεγάλη διάρκεια που συνήθως απαιτεί [25]. Η εναλλακτική επιλογή του PTQ συνιστά μία αρκετά πιο ταχύρρυθμη διαδικασία, η οποία ρυθμίζει τις τιμές των παραμέτρων, με την βοήθεια ενός calibration σετ δεδομένων από το οποίο εκτιμά τα εύρη τιμών των εξόδων του κάθε επιπέδου, προκειμένου να επιλέξει τις κατάλληλες τιμές για την κβάντιση. Η μεθοδολογία αυτή, ωστόσο, πολλές φορές συνοδεύεται από επιδείνωση της αποτελεσματικότητας του μοντέλου [26].

0.3.3 RISC-V

Σε αντίθεση με τις υπάρχουσες αρχιτεκτονικές επεξεργαστών, ο RISC-V διακρίνεται ως ένα ανοιχτού κώδικα σετ εντολών (ISA) που έχει σχεδιαστεί για τη δημιουργία εξειδικευμένων επεξεργαστών. Αρχικά αναπτύχθηκε στο Πανεπιστήμιο της Καλιφόρνια, Berkeley και αντιπροσωπεύει την πέμπτη γενιά επεξεργαστών που έχουν κατασκευαστεί βάσει του μειωμένου συνόλου εντολών (RISC). Λόγω της ανοιχτής φύσης του, μπορεί να χρησιμοποιηθεί τόσο σε ακαδημαϊκές όσο και σε βιομηχανικές εφαρμογές [27].

Ο RISC-V χαρακτηρίζεται από το μικρό μέγεθός του, καθώς περιλαμβάνει μονάχα 47 εντολές που είναι υποχρεωτικές για υλοποίηση. Σε αντίθεση, η αρχιτεκτονική x86 διαθέτει 1.503 εντολές, ενώ οι Arm επεξεργαστές διαθέτουν περίπου 500. Ο RISC-V υιοθετεί μια απλή αρχιτεκτονική, όπου όλες οι λειτουργίες πραγματοποιούνται εντός εσωτερικών καταχωρητών, και υπάρχουν αφιερωμένες εντολές για τη μεταφορά δεδομένων μεταξύ καταχωρητών και μνήμης.

Το τυπικό User-level Integer ISA έρχεται σε δύο παραλλαγές: το RV32I και το RV64I, τα οποία προσφέρουν 32 και 64-bit χώρους διευθύνσεων αντίστοιχα. Το βασικό RISC-V ISA, στο οποίο οι εντολές έχουν σταθερό μήκος 32 bits αποτελείται από 6 βασικούς τύπους εντολών [28] που παρουσιάζονται στο Σχήμα 0.3:

- **R-type**: εντολές μεταξύ καταχωρητών.
- **I-type**: short immediates και φόρτωση δεδομένων από την μνήμη.
- **S-type**: αποθήκευση δεδομένων στην μνήμη.
- **B-type**: conditional branches.
- **U-type**: long immediates.
- **J-type**: unconditional jumps.

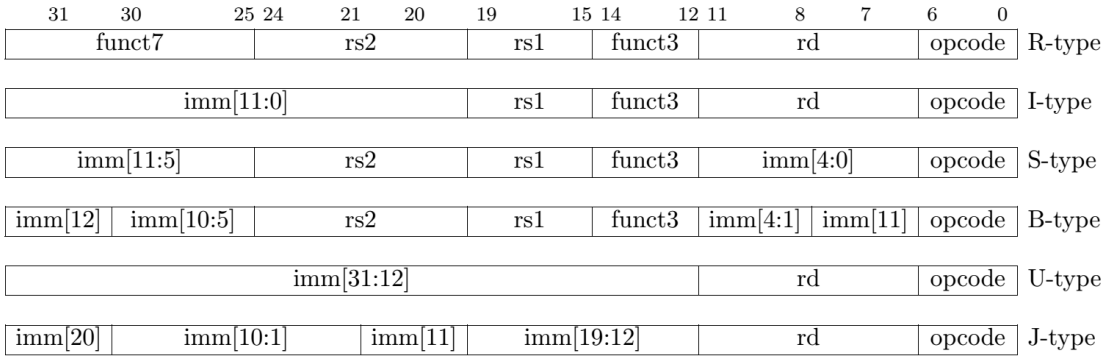


Figure 0.3: Δομή των βασικών εντολών του RISC-V ISA.
Πηγή: [28]

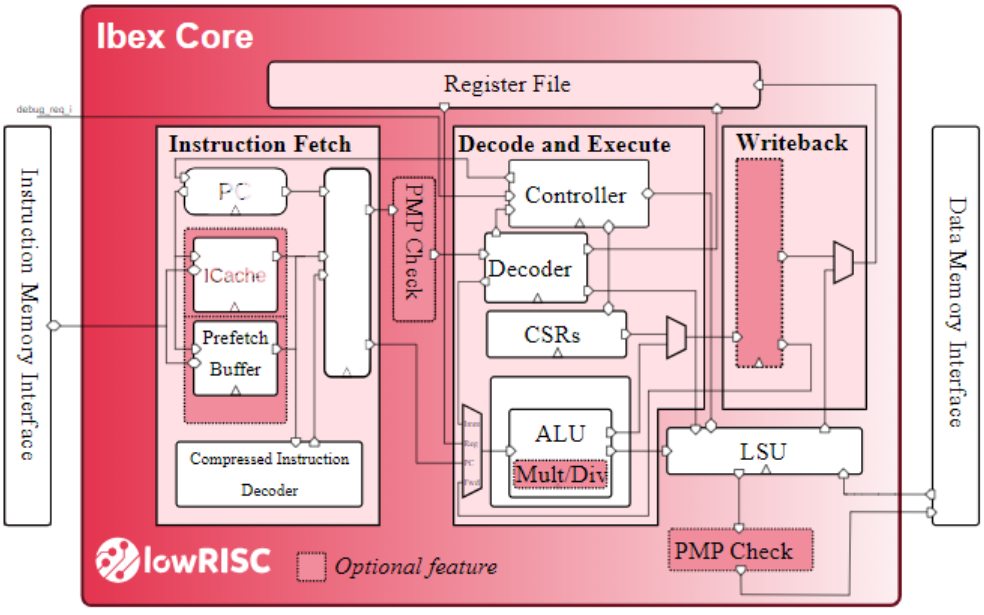
Πέρα από αυτές τις βασικές εντολές, οι προγραμματιστές, ανάλογα και με τις ανάγκες του προβλήματος τους, μπορούν να αξιοποιήσουν και μία ή και παραπάνω προαιρετικές επεκτάσεις με εντολές που υποστηρίζουν επιπλέον λειτουργίες. Ορισμένες από αυτές τις λειτουργίες αφορούν την υλοποίηση πράξεων όπως ο πολλαπλασιασμός και η διαίρεση μεταξύ ακέραιων μεταβλητών (επέκταση M), ατομικές εντολές για προσβάσεις στην μνήμη, σε περιπτώσεις που είναι αναγκαίος ο συγχρονισμός (επέκταση A), καθώς και εντολές για την διαχείριση Floating Point μεταβλητών (επεκτάσεις F, D, Q) και άλλες [29].

lowRISC/Ibex

Ο Ibex είναι ένας μικρός 32-bit RISC-V επεξεργαστής. Αρχικά, ήταν μέρος της πλατφόρμας PULP, υπό το όνομα "Zero-risky", με στόχο εφαρμογές πολύ χαμηλής κατανάλωσης ενέργειας. Σήμερα διατηρείται και αναπτύσσεται περαιτέρω από την lowRISC. Έχει υποστεί εκτενή επαλήθευση και υποστηρίζει την εκτέλεση εντολών για την διαχείριση δυαδικών αριθμών και αριθμητικές πράξεις (πρόσθεση, αφαίρεση, πολλαπλασιασμό και διαίρεση) μεταξύ

ακέραιων μεταβλητών. Η αρχιτεκτονική και η λειτουργικότητά του ορίζονται χρησιμοποιώντας τη γλώσσα περιγραφής υλικού (HDL) SystemVerilog. Διαθέτοντας ένα pipeline 2 σταδίων, εκ΄των οποίων το αρχικό στάδιο είναι αφιερωμένο στην Ανάκτηση Εντολών (IF), ακολουθούμενο από το στάδιο Αποκωδικοποίησης και Εκτέλεσης Εντολών (ID/EX). Επιπλέον, ένα προαιρετικό τρίτο στάδιο, WriteBack, είναι διαθέσιμο, το οποίο έχει ως ρόλο την αποθήκευση των αποτελεσμάτων των εκτελεσμένων εντολών πίσω στο Register File ή στη Μνήμη, διασφαλίζοντας την ομαλή λειτουργία και την ακεραιότητα των δεδομένων.

Ο πυρήνας διαθέτει διάφορες παραμέτρους για την προσαρμογή του στις απαιτήσεις συγκεκριμένων εφαρμογών. Οι επιλογές αυτές σχετίζονται με την αρχιτεκτονική της μονάδας πολλαπλασιασμού, καθώς και ζητήματα που αφορούν την ιεραρχία μνήμης, την ασφάλεια και την ενσωμάτωση λειτουργικών μονάδων που επηρεάζουν την επίδοση του (π.χ. Branch Predictor). Συγκεκριμένα, όσον αφορά τη μονάδα πολλαπλασιασμού, υπάρχουν δύο διαμορφώσεις διαθέσιμες. Η πρώτη διενεργεί τον πολλαπλασιασμό μεταξύ δύο 32-bit ακέραιων χρησιμοποιώντας μόνο έναν πολλαπλασιαστή 17×17, που αντιστοιχεί στη χρήση ενός Digital Signal Processor (DSP) κατά την υλοποίηση πάνω σε μία πλακέτα FPGA. Η πράξη ολοκληρώνεται σε 3 κύκλους, βελτιστοποιώντας τη χρήση πόρων με κόστος την αύξηση του χρόνου εκτέλεσης. Αντίθετα, η δεύτερη επιλογή επιτρέπει τον πολλαπλασιασμό να ολοκληρώνεται σε ένα μόνο κύκλο χρησιμοποιώντας τρεις πολλαπλασιαστές 17×17. Αυτή η διαμόρφωση απαιτεί την χρήση τεσσάρων DSPs—το επιπλέον DSP είναι απαραίτητο για τη συσσώρευση των μερικών γινομένων που παράγονται, προσφέροντας ένα πλεονέκτημα ταχύτητας με κόστος την αυξημένη χρήση πόρων.



Σχήμα 0.4: Σχηματικό Διάγραμμα του Ibex.
Πηγή: [30]

0.4 Configurable Mixed Precision RISC-V Architecture

Ο κύριος στόχος της παρούσας εργασίας είναι η ανάπτυξη συμπαγών, υψηλής ταχύτητας και ενεργειακά αποδοτικών DNNs σε RISC-V επεξεργαστές. Παράλληλα, δίνεται έμφαση στην ελαχιστοποίηση των επιπλέον πόρων που θα απαιτηθούν για τον τροποποιημένο πυρήνα, διασφαλίζοντας ότι οι βελτιώσεις στην απόδοση δεν θα οδηγήσουν στην υπερβολική κατανάλωση πόρων. Αυτή η μεθοδολογία είναι ευέλικτη και εφαρμόσιμη σε διάφορους τύπους DNNs. Παρ΄ όλα αυτά, θα εστιάσουμε σε αρχιτεκτονικές MLP και CNN, καθώς αυτές είναι οι δομές που χρησιμοποιούνται συχνότερα σε ML εφαρμογές για συσκευές στο Edge και μικροελεγκτές.

Η μεθοδολογία που θα ακολουθήσουμε μπορεί να χωριστεί σε τρία κύρια στάδια. Αρχικά, οι παράμετροι του μοντέλου υπόκεινται σε μια διαδικασία Mixed Precision Quantization (MPQ). Για να προσδιοριστεί η βέλτιστη επιλογή βαρών για κάθε επίπεδο, που θα ενισχύουν την ταχύτητα λειτουργίας του μοντέλου, διατηρώντας την ακρίβειά του, διενεργούμε μια ενδελεχή εξερεύνηση των διαθέσιμων επιλογών. Για περαιτέρω βελτίωση της απόδοσης του RISC-V, επιλέγουμε να επεκτείνουμε το υποστηριζόμενο σύνολο εντολών του επεξεργαστή. Οι εντολές αυτές θα αλληλεπιδρούν με έναν επιταχυντή NN, ενσωματωμένο αρμονικά μέσα στο pipeline του επεξεργαστή. Την μεθοδολογία αυτή, καθώς και την επιπλέον λειτουργική μονάδα θα τις δοκιμάσουμε πάνω στο υλικό του Ibex. Η προκύπτουσα αρχιτεκτονική θα υλοποιηθεί και θα αξιολογηθεί στην πλακέτα FPGA Virtex-7 που παρέχεται από την Xilinx.

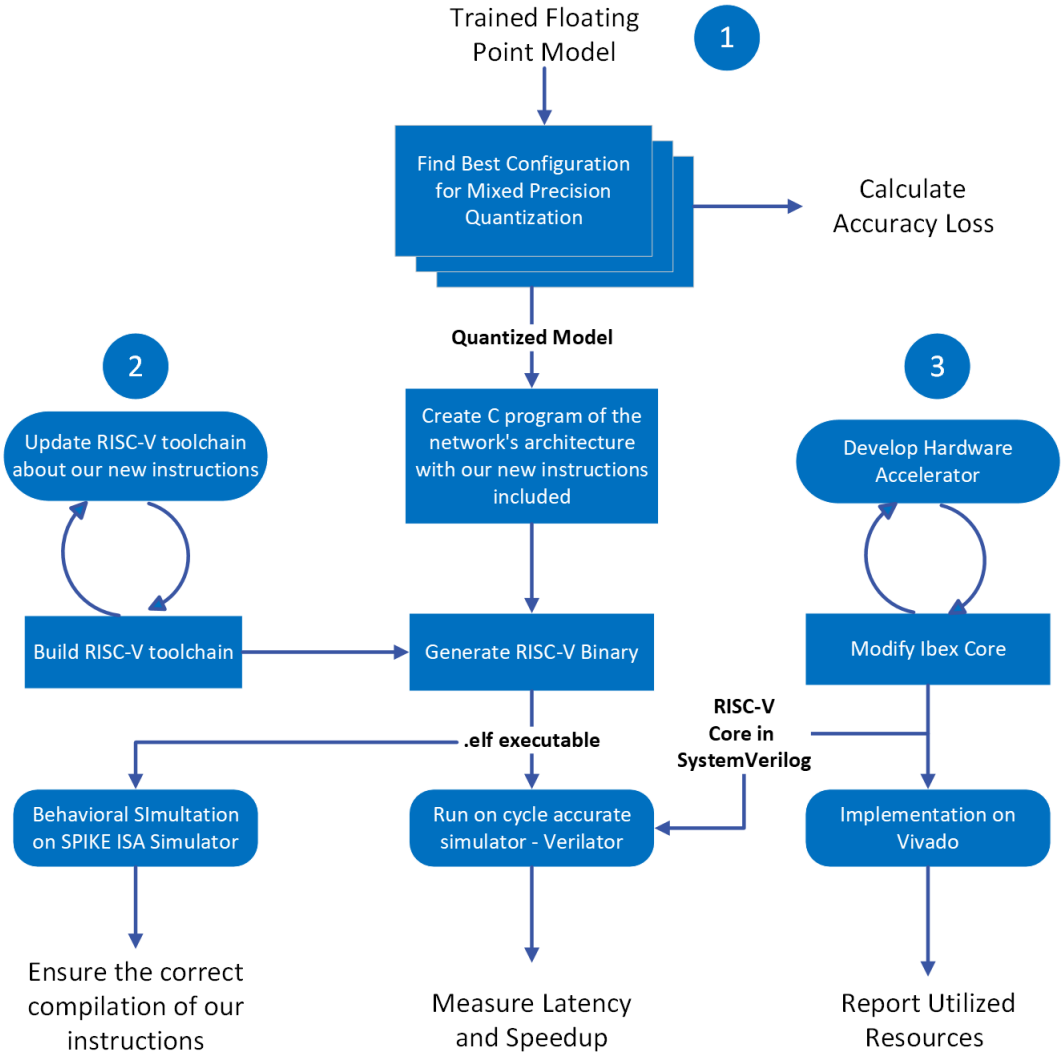
Μία σύντομη επισκόπηση της μεθοδολογίας που θα ακολουθήσουμε στην διπλωματική αυτή φαίνεται στο Σχήμα 0.5.

0.4.1 Model Quantization

Η αρχική φάση περιλαμβάνει την βαθμονόμηση της ακρίβειας ενός Νευρωνικού Δικτύου και τη λεπτομερή ρύθμιση της ακρίβειας των μεταβλητών του πριν από την ένταξή του στη συσκευή μας (❶ στο Σχήμα 0.5). Για την μείωση των συνολικών επιλογών, έχουμε επιλέξει να κωδικοποιούμε ενιαία τα δεδομένα εισόδου και τις τιμές ενεργοποίησης στις εξόδους σε όλα τα επίπεδα του δικτύου μας, χρησιμοποιώντας αποκλειστικά 8-βιτ αναπαραστάσεις. Αυτές οι τιμές μπορεί να είναι τόσο προσημασμένες (int8) όσο και μη-προσημασμένες (uint8). Όσον αφορά τα βάρη του δικτύου, αυτά μπορούν να μετατραπούν σε ακέραια μορφή με ανάλυση 2, 4, ή 8-bit.

Λαμβάνοντας υπόψη την πολυπλοκότητα της αρχιτεκτονικής ενός μοντέλου και την αποτελεσματικότητά του στο σετ δεδομένων που έχει εκπαιδευτεί, οι παράμετροι του μοντέλου αρχικά μετατρέπονται σε κβαντισμένη μορφή χρησιμοποιώντας PTQ. Εάν τα αποτελέσματα δεν είναι αρκετά ικανοποιητικά, η διαδικασία ενισχύεται περαιτέρω βελτιώνοντας τις παραμέτρους μέσω QAT μέχρι να επιτευχθεί το αποδεκτό επίπεδο απόδοσης.

Η εξαντλητική αναζήτηση αναδεικνύεται ως μια αξιόπιστη μεθοδολογία, ιδιαίτερα όταν αντιμετωπίζουμε μικρά Νευρωνικά Δίκτυα (NN) με λίγα στρώματα. Αυτή η μέθοδος εξασφαλίζει



Σχήμα 0.5: Διάγραμμα ροής της μεθοδολογίας που έχει χρησιμοποιηθεί.

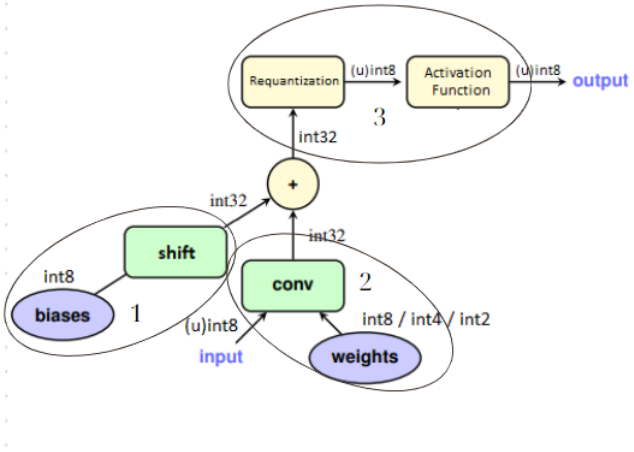
ότι η επιλεγμένη διαμόρφωση για τα βάρη του κάθε επιπέδου θα είναι η βέλτιστη εντός των καθορισμένων περιορισμών. Παρ' όλα αυτά, αυτή η στρατηγική είναι δύσκολο να εφαρμοστεί σε μεγάλα πολυεπίπεδα NNs, δεδομένου του τεράστιου όγκου των δυνατών επιλογών που πρέπει να αναλυθούν. Αυτή η πρόκληση, σε συνδυασμό με τους σημαντικούς χρόνους εκπαίδευσης, τυπικούς κατά το QAT, καθιστά επιτακτική την εξερεύνηση εναλλακτικών λύσεων για την κβάντιση μοντέλων μεγαλύτερης κλίμακας.

Μια πρακτική και αποδοτική λύση για την αντιμετώπιση αυτού του ζητήματος περιλαμβάνει την ομαδοποίηση διαδοχικών επιπέδων του μοντέλου και την αντιμετώπιση τους ως μία ενιαία οντότητα, μειώνοντας έτσι την έκταση του χώρου των πιθανών λύσεων. Εναλλακτικά μπορούμε να θέτουμε σταθερό bit-width σε επίπεδα, των οποίων το υπολογιστικό φορτίο είναι αμελητέο σε σχέση με το σύνολο των υπολογισμών ή να απορρίπτουμε λύσεις, οι οποίες θα οδηγήσουν σε πολύ μεγάλη υποβάθμιση της ακρίβειας του μοντέλου. Η χρήση τέτοιων στρα-

τηγικών είναι αναγκαία, όταν η εξαντλητική κάλυψη ολόκληρου του χώρου είναι ανέφικτη, είτε λόγω χρονικών περιορισμών είτε λόγω περιορισμένων υπολογιστικών πόρων. Ωστόσο, αυτή η μείωση μπορεί να έρθει με ένα κόστος, καθώς ενδέχεται να υιοθετηθεί μια λιγότερο αποδοτική λύση.

0.4.2 Εισαγωγή των νέων εντολών

Η διαδικασία που περιγράφει την εκτέλεση DNN μοντέλων, μονάχα με την χρήση ακέραιων μεταβλητών, αποτυπώνεται οπτικά στο Σχήμα 0.6.



Σχήμα 0.6: Inference μονάχα με τη χρήση ακέραιων μεταβλητών

Η διαδικασία μπορεί να διακριθεί σε 3 ξεχωριστές φάσεις. Το πρώτο βήμα αφορά την αρχικοποίηση των εξόδων, θέτοντας τις bias τιμές τους. Αντί για την χρήση 32-bit μεταβλητών, μία εναλλακτική και εξίσου αποδοτική λύση είναι αξιοποιήσουμε 8-bit αριθμούς και στην συνέχεια εκτελώντας αριστερή ολίσθηση να προσεγγίσουμε τις 32-bit αρχικές τιμές. Η δεύτερη φάση, η οποία είναι και η υπολογιστικά πιο απαιτητική, περιλαμβάνει την εκτέλεση όλων των απαιτούμενων πολλαπλασιασμών μεταξύ των εισόδου του εκάστοτε επιπέδου και των αντίστοιχων βαρών τους. Τα αποτελέσματα των πολλαπλασιασμών στη συνέχεια συσσωρεύονται μαζί με τα προηγουμένως ορισμένα biases. Αυτές οι διαδοχικές πράξεις πολλαπλασιασμού-συσσώρευσης (MAC) είναι καθοριστικές και αποτελούν το επίκεντρο της βελτιστοποίησης στην έρευνά μας. Το τελικό στάδιο εξασφαλίζει την μετατροπή των αποτελεσμάτων από 32-bit σε 8, ώστε να τροφοδοτειθούν στο επόμενο επίπεδο. Αυτό επιτυγχάνεται μέσω ενός πολλαπλασιασμού με μία θετική τιμή και στην συνέχεια ακολουθεί μία δεξιά ολίσθηση, ώστε η έξοδος να βρεθεί στο επιθυμητό εύρος τιμών.

Η δομή των εντολών που θα προστεθουν θα είναι ίδια με εκείνων που ανήκουν στις R-type εντολές του RISC-V ISA . Προκειμένου να διασφαλιστεί η ορθή αναγνώριση των εντολών, καθώς και για να επιτευχθεί η σωστή λειτουργία τους, θα πρέπει να ορίσουμε:

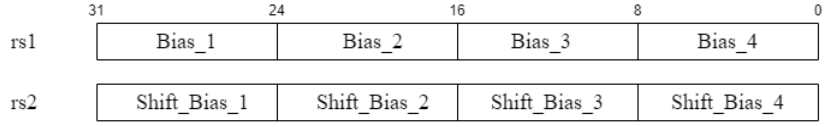
- Το όνομα της κάθε μίας, το οποιο θα πρέπει να αναδεικνύει την λειτουργικότητα της.

- Τα απαραίτητα πεδία μέσα στο 32-bit πλαίσιο της κάθε εντολής. Πιο συγκεκριμένα, για μια R-type εντολή, πρέπει να καθορίσουμε:

  1. Το opcode, το οποίο υποδηλώνει την ευρύτερη κατηγορία που ανήκει. Εφόσον όλες οι νέες εντολές θα αξιοποιούνται για την βελτιστοποίηση Νευρωνικών Δικτύων, θα έχουν όλες το ίδιο opcode.
  2. Τις τιμές των Function codes (funct3, funct7), τα οποία χρησιμοποιούνται για να προσδιορίσουν την λειτουργία ή την παραλλαγή μίας εντολής μέσα στην κατηγορία που η ίδια ανήκει.
  3. Τους καταχωρητές οι οποίοι κατά την εκτέλεση του προγράμματος, θα χρησιμοποιηθούν ως οι είσοδοι και έξοδοι της.

Οι νέες εντολές που θα προστεθούν στο τοολςηαιν (❷ στο Σχήμα 0.5) θα έχουν ως στόχο την εκτέλεση και βελτιστοποίηση καθενός από τα 3 στάδια που περιγράψαμε προηγουμένως και θα μοιράζονται το ίδιο opcode ($0x47$ or 100 0000 σε δυαδική μορφή).

Η πρώτη από αυτές θα αξιοποιείται για να αρχικοποιεί τον NN επιταχυντή που και θα αξιοποιεί 2 καταχωρητές στους οποίους θα εμπεριέχονται τέσσερις 8-bit τιμές biases , ενώ στον δεύτερο καταχωρητή, θα βρίσκονται τιμές που θα υποδηλώνουν το πλήθος των αριστερών ολισθήσεων που θα αντιστοιχούν σε κάθε ένα από αυτά τα biases (βλέπε Σχήμα 0.7).

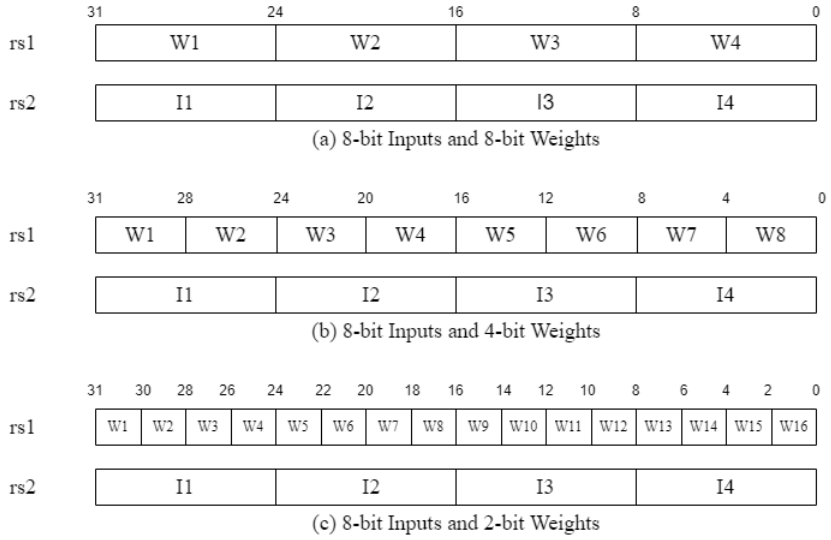


Σχήμα 0.7: Η δομή των 32-bit καταχωρητών όταν καλείται η εντολή neur_init.

Όνομα Εντολής	funct7	funct3	rs1	rs2	Περιγραφή
neur_init	000 0000	100	4 8-bit biases	Πλήθος αριστερών ολισθήσεων για κάθε bias	Αρχικοποίηση των biases για κάθε έξοδο

Πίνακας 0.1: Εντολή αφιερωμένη για την αρχικοποίηση της λειτουργικής μονάδας που θα προσθέσουμε, θέτοντας τα biases των εξόδων.

Η δεύτερη σειρά εντολών που θα ενσωματώσουμε είναι σχεδιασμένη για να διευκολύνει τις λειτουργίες των MAC εντολών, οι οποίες είναι κρίσιμες για τον αλγόριθμό μας. Κάθε μία από αυτές θα εκτελείται σε μονάχα έναν κύκλο και θα υλοποιεί ένα διακριτό σενάριο, όπως αυτά προκύπτουν από τον συνδυασμό της ακρίβειας των βαρών (2, 4 ή 8-bit) και τη μορφή των εισόδων του εκάστοτε επιπέδου, προσημασμένη ή μη-προσημασμένη (βλέπε Σχήμα 0.8). Αυτή η σκόπιμη διάκριση υπηρετεί δύο σκοπούς. Ο πρώτος είναι για λόγους σαφήνειας και ευκολίας χρήσης, καθώς βελτιώνει την αναγνωσιμότητα και τη χρηστικότητα του πηγαίου κώδικα, ενώ ο δεύτερος έχει πιο πρακτικό χαρακτήρα, καθώς μπορούμε να επωφεληθούμε από



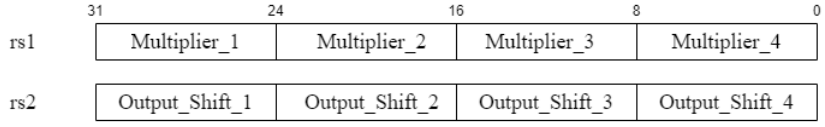
Σχήμα 0.8: Εσωτερική δομή των καταχωρητών κατά την κλήση των MAC εντολών.

αυτή την κατηγοριοποίηση προκειμένου να ενεργοποιήσουμε διαφορετικά σήματα κατά την αποκωδικοποίηση κάθε εντολής. Αυτά τα σήματα είναι ουσιώδη για τη σωστή λειτουργία του επιταχυντή υλικού, ώστε να υπολογίζει με ακρίβεια τις εξόδους σε κάθε περίπτωση.

Όνομα Εντολής	funct7	funct3	rs1	rs2	Περιγραφή
neur_mac_u_8b	000 1000	010	4 8-bit unsigned είσοδοι	4 8-bit βάρη	4 MAC λειτουργίες
neur_mac_u_4b	000 0100	010	4 8-bit unsigned είσοδοι	8 4-bit βάρη	8 MAC λειτουργίες
neur_mac_u_2b	000 0010	010	4 8-bit unsigned είσοδοι	16 2-bit βάρη	16 MAC λειτουργίες
neur_mac_s_8b	001 1000	010	4 8-bit signed είσοδοι	4 8-bit βάρη	4 MAC λειτουργίες
neur_mac_s_4b	001 0100	010	4 8-bit signed είσοδοι	8 4-bit βάρη	8 MAC λειτουργίες
neur_mac_s_2b	001 0010	010	4 8-bit signed είσοδοι	16 2-bit βάρη	16 MAC λειτουργίες

Πίνακας 0.2: Λίστα εντολών αφιερωμένων για την επιτάχυνση της εκτέλεσης των MAC ακολουθιών.

Το τελευταίο σετ εντολών θα είναι υπεύθυνο για τη μετατροπή των τιμών που είναι αποθηκευμένες στους 32-bit συσσωρευτές σε 8-bit αριθμούς. Μέσα στους πλαίσιο των καταχωρητών-εισόδων, είναι συγχωνευμένες οι απαραίτητες μεταβλητές για τον πολλαπλασιασμό και την αριστερή ολίσθηση (βλέπε Σχήμα 0.9). Αυτές είναι οι μοναδικές εντολές που απαιτούν παραπάνω από 1 κύκλο για την εκτέλεση τους (3-5 κύκλοι), ανάλογα με τον αν θα έχει ολοκληρωθεί πλήρως ο υπολογισμός των MAC εντολών.



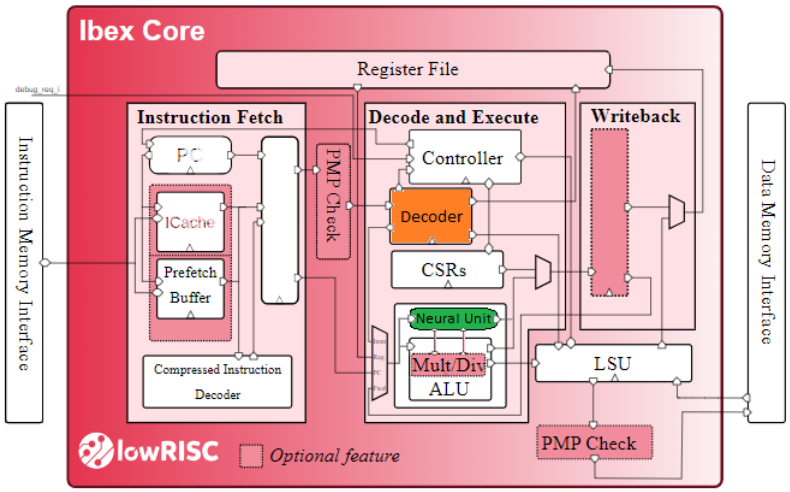
Σχήμα 0.9: Περιεχόμενο καταχωρητών κατά την εκτέλεση των εντολών που υλοποιούν το requantization βήμα.

Όνομα Εντολής	funct7	funct3	rs1	rs2	Περιγραφή
neur_res_u	000 0001	001	4 8-bit τελεστές για πολ/μο	4 8-bit τιμές για δεξιά ολίσθηση	Μετατροπή των αποτελεσμάτων σε **uint8**
neur_res_s	000 0000	001	4 8-bit τελεστές για πολ/μο	4 8-bit τιμές για δεξιά ολίσθηση	Μετατροπή των αποτελεσμάτων σε **int8**

Πίνακας 0.3: Λίστα εντολών για την μετατροπή των συσσωρευμένων τιμών σε 8-bit αριθμούς.

0.4.3 Μετατροπές στον Επεξεργαστή

Το τελευταίο βήμα αφορά την τροποποίηση του επεξεργαστή για την υποστήριξη των νέων εντολών (❸ στο Σχήμα 0.5). Αντί να αναπτύξουμε έναν ξεχωριστό co-processor, ο επιταχυντής θα τοποθετηθεί στο δεύτερο στάδιο του pipeline, όπου λαμβάνει χώρα η αποκωδικοποίηση και η εκτέλεση κάθε εντολής. Αυτή η απόφαση αποδεικνύεται όχι μονάχα πιο γρήγορη, αλλά και πιο αποδοτική ως προς την κατανάλωση ενέργειας, σε σύγκριση με την άλλη επιλογή. Το κύριο πλεονέκτημα αυτής της ένταξης κρίνεται στη σημαντική μείωση της μεταφοράς δεδομένων μέσω του διαύλου που διασυνδέει τη μνήμη, τον κύριο επεξεργαστή και αυτό που θα ήταν ένας ξεχωριστός co-processor. Μπορούμε επίσης να επωφεληθούμε από την κοινή χρήση των πόρων του επεξεργαστή, όπως είναι οι cache μνήμες και άλλα κρίσιμα στοιχεία (π.χ. πολλαπλασιαστές), προκειμένου να υλοποιήσουμε διάφορους υπολογισμούς στον επιταχυντή μας.



Σχήμα 0.10: Σχηματικό διάγραμμα του Ibex, μετά από τις τροποποιήσεις στο υλικό του. Με πορτοκαλί αναφερόμαστε στα τμήματα που έγιναν επιπλέον αλλαγές, ενώ με πράσινο τα νέα τμήματα του επεξεργαστή.

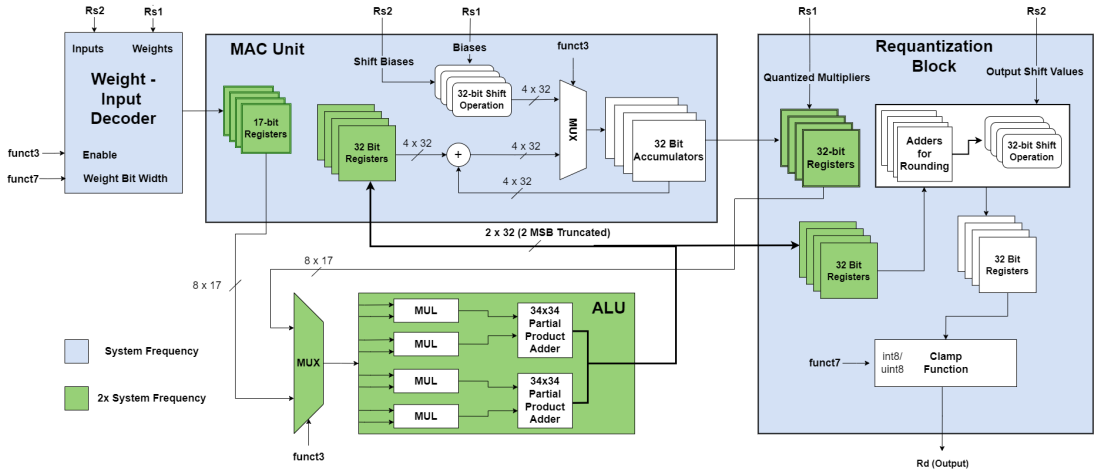
Το βασικό χαρακτηριστικό του επιταχυντή είναι η δυνατότητά του να επεξεργάζεται ταυτόχρονα τέσσερις εξόδους. Αυτή η δυνατότητα παράλληλης επεξεργασίας ενισχύει σημαντικά

την απόδοση του, επιτρέποντας στο σύστημα να διαχειρίζεται πολλαπλά δεδομένα μέσα στον ίδιο κύκλο. Τα ενδιάμεσα αποτελέσματα των υπολογισμών αποθηκεύονται σε 32-bit τοπικούς καταχωρητές εντός της μονάδας. Πέρα από τους συσσωρευτές αυτούς, τα βασικότερα στοιχεία του είναι (βλέπε Σχήμα 0.11):

- **Αποκωδικοποιητής Βαρών - Εισόδων**: Σχεδιασμένος για τον χειρισμό δύο 32-bit τελεστών, ο αποκωδικοποιητής χρησιμοποιεί το πεδίο 'funct7' για να αναγνωρίσει και να παρασκευάσει τα ορίσματα που θα εμπλακούν στον πολλαπλασιασμό. Παράγει 8 ζεύγη τιμών 16-bit που θα προωθηθούν για επεξεργασία από τις υπόλοιπες μονάδες του επιταχυντή.

- **Μονάδα Πολλαπλασιασμού-Συσσώρευσης (MAC)**: Η μονάδα MAC λαμβάνει τις αποκωδικοποιημένες τιμές και διεξάγει τους απαραίτητους πολλαπλασιασμούς. Σε περιπτώσεις που είναι απαραίτητο (περιπτώσεις (b) και (c) στο Σχήμα 0.8), μερικά αποτελέσματα στη συνέχεια αθροίζονται προτού συσσωρευτούν πίσω στους 32-bit καταχωρητές.

- **Requantization Block**: Η μονάδα αυτή λαμβάνει ως είσοδο τις τιμές από τους συσσωρευτές και εξάγει τις τιμές στην κατάλληλη μορφή, ώστε να αποθηκευτούν στη μνήμη.



Σχήμα 0.11: Η προτεινόμενη αρχιτεκτονική του επιταχυντή. Με πράσινο χρώμα αναπαρίστανται τα δομικά στοιχεία τα οποία λειτουργούν σε διπλάσια συχνότητα από ότι εκείνα του υπόλοιπου επεξεργαστή.

Ορισμένες από τις σχεδιαστικές επιλογές που λάβαμε, με κύριο στόχο την αύξηση της διεκπεραιωτικότητας του επιταχυντή και την ελαχιστοποίηση των αναγκαίων επιπλέον πόρων αναφέρονται παρακάτω.

Εκμετάλλευση των πόρων του επεξεργαστή

Όπου είναι δυνατόν, ο επιταχυντής μοιράζεται πόρους με τον κύριο επεξεργαστή. Πιο συγκεκριμένα, επιλέγουμε να επαναχρησιμοποιήσουμε τους πολλαπλασιαστές του για να υπολογίσουμε όλα τα απαραίτητα γινόμενα, κατά τη διάρκεια των MAC εντολών και για την διαδικασία του requantization. Όπως αναφέραμε και προηγουμένως, για την υλοποίηση του πολλαπλασιασμού μεταξύ δύο 32-bit ακεραίων σε έναν κύκλο, ο Ibex διαθέτει τρεις πολλαπλασιαστές 17×17 bit και στη συνέχεια προσθέτει μαζί τα μερικά γινόμενα για να δημιουργήσει το τελικό αποτέλεσμα. Για να καλύψουμε τις απαιτήσεις του επιταχυντή μας και για να ενισχύσουμε περαιτέρω την υπολογιστική του δύναμη, επεκτείνουμε αυτή τη διάταξη ενσωματώνοντας έναν επιπλέον πολλαπλασιαστή, με βάση το σχεδιασμό των υπάρχοντων τριών. Ωστόσο, κατά το implementation πάνω σε ένα FPGA, τόσο ο αρχική, όσο και η τροποποιημένη αρχιτεκτονική αξιοποιούν 4 DSPs. Αυτό οφείλεται στο οτι το επιπλέον DSP θα χρησιμοποιηθεί στην μία περίπτωση για το άθροισμα των μερικών γινομένων, ενώ στην δεύτερη, θα χρησιμοποιηθεί από τον επιταχυντή μας για την εκτέλεση ενός ακόμα πολλαπλασιασμού.

Pipelining

Το pipelining είναι μια τεχνική που χρησιμοποιείται ευρέως στην αρχιτεκτονική υπολογιστών για να αυξήσει τη ροή δεδομένων μέσα σε ένα επεξεργαστή. Αυτό το επιτυγχάνει διαιρώντας μια διεργασία σε αρκετά επιμέρους στάδια και επιτρέποντας σε κάθε στάδιο να χειρίζεται ένα διαφορετικό κομμάτι του υπολογισμού, διασφαλίζοντας έτσι την συνεχή και πιο αποδοτική εκτέλεση εντολών. Αποδεικνύεται ιδιαίτερα χρήσιμη σε προγράμματα χωρίς data hazards, τα οποία προκύπτουν όταν η εκτέλεση των εντολών εξαρτάται από τα αποτελέσματα των προηγούμενων. Στα συστήματα με pipeline, το αποτέλεσμα κάθε σταδίου προσωρινά αποθηκεύονται σε καταχωρητές προτού μεταβιβαστούν στο επόμενο στάδιο. Αν και ο συνολικός αριθμός των κύκλων που απαιτούνται για την εκτέλεση ενός μόνο υπολογισμού αυξάνεται, εν τέλει επωφελούμαστε από το αυξημένο throughput καθώς και από το μειωμένο κρίσιμο μονοπάτι, το οποίο μας επιτρέπει να αυξήσουμε τη συχνότητα του ρολογιού που χρησιμοποιείται για αυτές τις λειτουργίες. Στην περίπτωση μας, υλοποιούμε ένα pipeline στο εσωτερικό δύο διακριτών στοιχείων: την μονάδα που είναι αφιερωμένη στην εκτέλεση των MAC εντολών σε συνδυασμό με τον αποκωδικοποιητή που προηγείται, και το block που είναι υπεύθυνο για το βήμα του requantization.

Αύξηση Συχνότητας Λειτουργίας

Δεδομένου ότι η αρχιτεκτονική μας ενσωματώνει μόνο τέσσερις πολλαπλασιαστές και λαμβάνοντας υπόψη ότι ορισμένες εντολές απαιτούν την εκτέλεση 8 ή ακόμη και 16 MAC ακολουθιών, η επιβολή καθυστερήσεων στο pipeline του επεξεργαστή θα ήταν επιτακτική για την εξαγωγή των ορθών αποτελεσμάτων. Συγκεκριμένα, για τις εντολές που απαιτούν 8 MAC αλληλουχίες, θα επιβάλλαμε καθυστέρηση 1 κύκλου, ενώ για εκείνες που απαιτούν 16, καθυστέρηση 3 κύκλων θα ήταν αναπόφευκτη χωρίς επιπλέον βελτιστοποιήσεις στον επεξεργαστή.

Η εφαρμογή μιας ετερογενούς στρατηγικής όσον αφορά τα ρολόγια του συστήματος, όπου οι μονάδες με τον μεγαλύτερο υπολογιστικό φόρτο θα λειτουργούν σε υψηλότερη συχνότητα

από το υπόλοιπο σύστημα αποτελεί μια λύση σε αυτή τη πρόκληση [31]. Αυτή η προσέγγιση στοχεύει άμεσα στη βελτιστοποίηση της απόδοσης των πολλαπλασιαστών και αθροιστών που εμπλέκονται στις MAC εντολές και στη διαδικασία διαμόρφωσης της εξόδου. Αυξάνοντας τη συχνότητα λειτουργίας τους, είμαστε σε θέση να εκτελέσουμε περισσότερους υπολογισμούς ανά μονάδα χρόνου, αυξάνοντας αποτελεσματικά την συνολική απόδοση του συστήματος. Ωστόσο, με την υιοθέτηση μίας τέτοιας στρατηγικής, μπορεί να προκύψουν διάφορα ζητήματα συγχρονισμού, όπως διαφθορά ή απώλεια δεδομένων κατά τις μεταφορές μεταξύ στοιχείων που λειτουργούν σε διαφορετικές συχνότητες, ενώ επίσης παρατηρείται και αύξηση στην κατανάλωση ενέργειας καθώς και στους απαιτούμενους πόρους για την υλοποίηση του τελικού συστήματος .

Για να απλοποιήσουμε τα ζητήματα συγχρονισμού με το κύριο ρολόι του επεξεργαστή, αποφασίσαμε να αυξήσουμε τη συχνότητα του νέου ρολογιού κατά μία τιμή που είναι δύναμη του δύο. Δεδομένου ότι ο πυρήνας Ibex λειτουργεί με βασική συχνότητα 50 MHz, επιλέξαμε να ρυθμίσουμε το δεύτερο ρολόι σε διπλάσια συχνότητα στα 100 MHz. Με αυτή την προσέγγιση, μπορούμε να επιτύχουμε και την ζητούμενη αύξηση στην ταχύτητα των υπολογισμών, αφού γίνεται πλέον δυνατή η εκτέλεση των εντολών που αναλαμβάνουν 8 MAC υπολογισμούς (1 καθυστέρηση εξακολουθεί να είναι απαραίτητη για την τελευταία περίπτωση), ενώ επίσης διατηρούμε σε χαμηλά επίπεδα την κατανάλωση ενέργειας και την πολυπλοκότητα της σχεδίασης.

Soft SIMD

Η τελευταία βελτιστοποίηση στοχεύει στο επιτύχει την ολοκλήρωση των εντολών που εκτελούν 16 MAC υπολογισμούς σε έναν μόνο κύκλο. Εμπνευσμένοι από την έρευνα [32], επιδιώκουμε να επιτύχουμε αυτόν τον στόχο πραγματοποιώντας δύο πολλαπλασιασμούς μεταξύ των 2-bit βαρών και των 8-bit εισόδων σε έναν μόνο πολλαπλασιαστή.

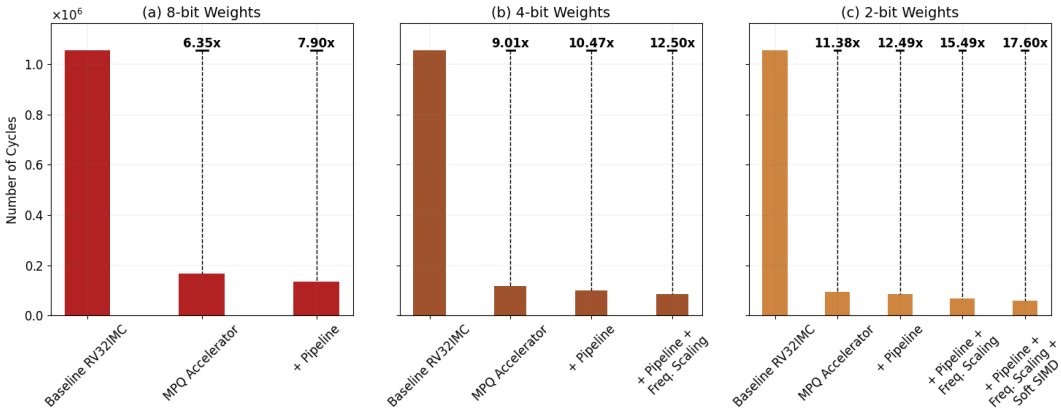
Το αποτέλεσμα του πολλαπλασιασμού μεταξύ ενός 2-bit αριθμού και ένος 9-bit αριθμού - 9, διότι το επιπλέον bit είναι αναγκαίο για το πρόσημο της εισόδου - απαιτεί ως ελάχιστο πλάτος 11 bits για ακριβή αναπαράσταση του. Σε τέτοιους πολλαπλασιασμούς, τα 22 ανώτερα bits είναι γεμάτα ομοιόμορφα είτε με 1 είτε με 0, μεταφέροντας αποτελεσματικά μόνο ένα bit πληροφορίας. Αυτό το χαρακτηριστικό επιτρέπει τη χρήση αυτών των ανώτερων bits για την εκτέλεση ενός επιπλέον υπολογισμού ταυτόχρονα, χωρίς να επηρεάζεται η ακρίβεια των αποτελεσμάτων που τοποθετούνται στα λιγότερο σημαντικά bits.

0.5 Αξιολόγηση των Αποτελεσμάτων

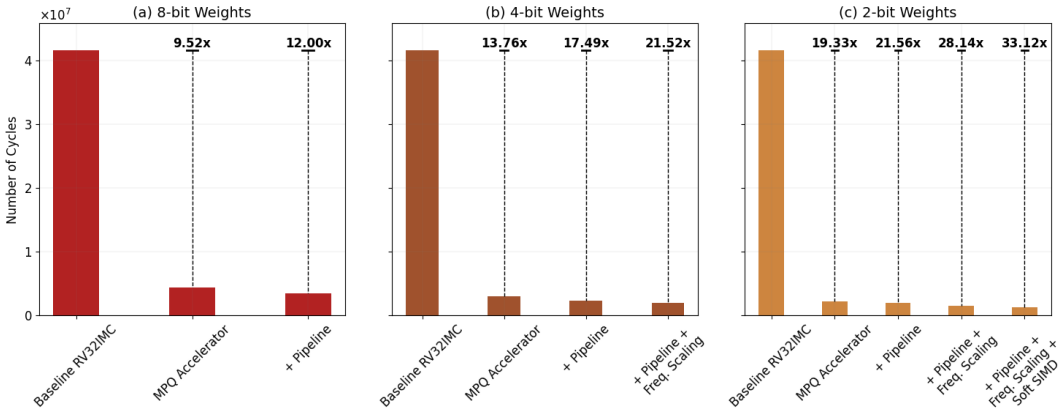
Για να αξιολογήσουμε την απόδοση του επιταχυντή μας, αρχικά θα συγκρίνοντας τα αποτελέσματα της εκτέλεσης ενός Πλήρους Συνδεδεμένους (Dense) και ενός Συνελικτικού επιπέδου έναντι των αντίστοιχων υλοποιήσεών που βασίζονται στο RV32IMC ISA. Πιο συγκεκριμένα θα εξετάσουμε:

1. ένα Fully Connected Layer με 512 εισόδους κόμβους και 256 κόμβους εξόδου και
2. ένα convolutional kernel που επεξεργάζεται μία είσοδο με διαστάσεις $16 \times 16 \times 32$ (χρησιμοποιώντας διάταξη δεδομένων Ύψος-Πλάτος-Κανάλια) χρησιμοποιώντας φίλτρα με διαστάσεις $64 \times 3 \times 3 \times 32$ (Πλήθος φίλτρων × Πλάτος Φίλτρων × Ύψος Φίλτρων × Κανάλια Εισόδου)

Σε κάθε μέτρηση, θα εξερευνήσουμε κάθε δυνατή διαμόρφωση για την ακρίβεια των βαρών ενώ επίσης θα αποτυπώσουμε πώς κάθε τεχνική βελτιστοποίησης του επιταχυντή επηρεάζει την τελική επίδοση. Τα αποτελέσματα της σύγκρισης απεικονίζονται στα Σχήματα 0.12 και 0.13. Οι τελικές μας υλοποιήσεις πετυχαίνουν επιτάχυνση **7.9 - 17.6x** στα Πλήρως Συνδεδεμένα στρώματα και **12.0 - 33.1x** σε Συνελικτικά επίπεδα.



Σχήμα 0.12: Χρόνος εκτέλεσης ενός Dense Layer στον επεξεργαστή Ibex για διαμόρφωση με: (a) 8-bit βάρη, (b) 4-bit βάρη, ανδ (c) 2-bit βάρη.



Σχήμα 0.13: Χρόνος εκτέλεσης ενός Convolutional Layer στον επεξεργαστή Ibex για διαμόρφωση με: (a) 8-bit βάρη, (b) 4-bit βάρη, ανδ (c) 2-bit βάρη.

Για να αναδείξουμε πλήρως την αποτελεσματικότητα της μεθοδολογίας μας, παρουσιάζουμε μια επισκόπηση της απόδοσης του τροποποιημένου πυρήνα Ibex πάνω σε διάφορα μοντέλα

DNN. Τα επιλεγμένα μοντέλα επιλέχθηκαν βάσει της αρχιτεκτονικής τους, της πολυπλοκότητάς τους και της εξαιρετικής απόδοσή τους σε διάφορες εφαρμογές.

Ο Πίνακας 0.4 περιλαμβάνει τα αποτελεσμάτων που έχουμε αποκτήσει από όλα τα μοντέλα που έχουμε αναλύσαμε. Εστιάζουμε κυρίως στον χρόνο εκτέλεσης που παρατηρείται στον Ibex, τόσο με την ενσωμάτωση νέων εντολών όσο και χωρίς αυτές. Επιπλέον, περιλαμβάνουμε τις κανονικοποιημένες τιμές, όπως αυτές έχουν ληφθεί από τις state-of-the-art έρευνες [5], [8] και [11], παρέχοντας ένα σημείο αναφοράς για σύγκριση και αποδεικνύοντας την αποτελεσματικότητα της προσέγγισής μας στην ενίσχυση της απόδοσης.

Μοντέλο	**RV32IMC**	**State of the art**	**QNN με μείωση ακρίβειας**		
			<1%	**<2%**	**<5%**
MLP από FANN-on-MCU	23.13 msec	1.60 msec	1.77 msec	**1.41 msec**	**1.01 msec**
CNN από CMSIS-NN	3480 msec	428 msec	**195 msec**	**145 msec**	**123 msec**
mcunet-vww-1	3697 msec	552 msec	619 msec	**548 msec**	**525 msec**

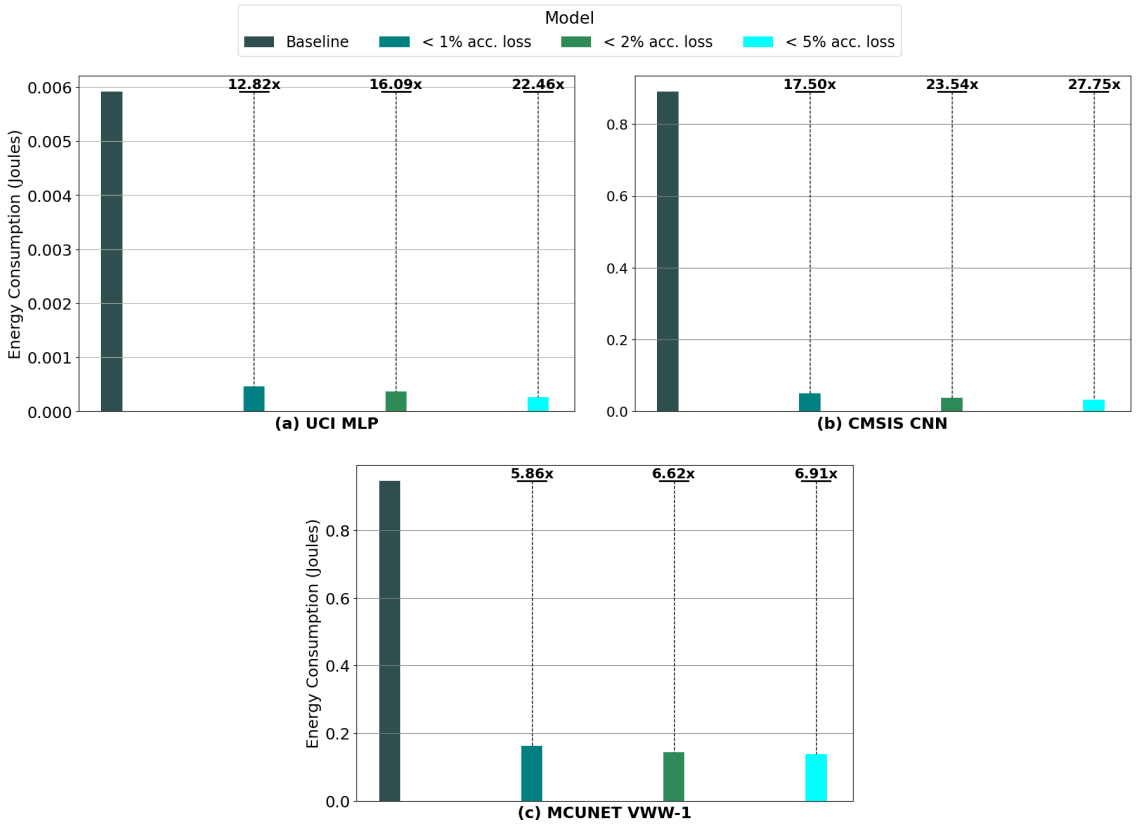
Πίνακας 0.4: Χρόνος εκτέλεσης που καταγράφηκε (msec) για κάθε μοντέλο που εξετάσαμε. Οι τιμές που αντιστοιχούν στις λύσεις από το state-of-the-art έχουν ληφθεί από τα αντίστοιχα paper, ενώ οι υπόλοιπες μετρήθηκαν στον Ibex core.

Θα αναλύσουμε το πρόσθετο φορτίο που εισήχθη στον αρχικό RISC-V επεξεργαστή, το οποίο συνιστά βασικό κριτήριο καθόλη τη διάρκεια της εργασίας μας, ιδιαίτερα κατά την ανάπτυξη του επιταχυντή. Ο Πίνακας 0.5 περιέχει συγκεντρωτικά τους πόρους που καταγράφηκαν κατά την φάση της υλοποίησης πάνω σε FPGA. Η μεθοδολογία μας καταφέρνει να ενισχύσει την απόδοση του Ibex, διατηρώντας την αύξηση των πόρων (34.89% αύξηση στην χρήση Lookup Tables και 24.28% αύξηση στην χρήση Flip-Flops) και την κατανάλωση ενέργειας (5mW αύξηση) σε χαμηλά επίπεδα.

Ibex Processor	**Αρχικός**	**Προτεινόμενος**
LUTs	5479	7391
FFs	5122	6366
DSPs	4	4
Κατανάλωση Ισχύος (Watt)	0.256	0.261
Συχνότητα Ρολογιού	50 MHz	50 MHz/100 MHz (Dual Clock Conf.)

Table 0.5: Σύγκριση μεταξύ του αρχικού και του τροποποιημένου επεξεργαστή όσον αφορά τους απαιτούμενους πόρους σε μία πλακέτα FPGA Virtex-7, την κατανάλωση ενέργειας τους και την ταχύτητα των ρολογιών που χρησιμοποιούνται.

Οι πιο ενεργειακά αποδοτικές λύσεις είναι εκείνες με την υψηλότερη απώλεια ακρίβειας. Αυτό οφείλεται στην επιτάχυνση που πετυχαίνουμε, παρά την οριακή αύξηση στην κατανάλωση ισχύος της προτεινόμενης αρχιτεκτονικής. Συγκεκριμένα, αυτές οι βελτιστοποιήσεις οδηγούν σε μια μείωση **22.46x** στις επεξεργαστικές απαιτήσεις για το MLP, μια μείωση **27.75x** για το CNN, και μια μείωση **6.91x** για το mcunet-vww1. Το Σχήμα 0.14 παρουσιάζει τα σημαντικά κέρδη στην κατανάλωση ενέργειας.



Σχήμα 0.14: Κατανάλωση Ενέργειας κάθε εκδοχής των μοντέλων που εξετάσαμε: (a) MLP από το "FANN-on-MCU", (b) CNN από το "CMSIS-NN", και (c) mcunet-vww1 από το "MCUNet".

0.6 Συμπεράσματα και Μελλοντικές Προεκτάσεις

Στην εργασία μας, παρουσιάσαμε ένα ολοκληρωμένο πλαίσιο για την ανάπτυξη και βελτιστοποίηση μοντέλων TinyML σε έναν επεξεργαστή RISC-V. Αποδείξαμε ότι μέσω της εφαρμογής ενός σχήματος Mixed Precision Quantization και της μεθοδικής εξερεύνησης όλων των σχεδιαστικών επιλογών, δύναται να βρούμε λύσεις που ισορροπούν επιτάχυνση των αλγορίθμων ελαχιστοποιώντας τις απώλειες στην ακρίβεια των μοντέλων. Για να ενισχύσουμε την εκτέλεση τους στον επιλεγμένο επεξεργαστή, επεκτείναμε το RV32IMC ISA με εξειδικευμένες εντολές και ενσωματώσαμε μια επιπλέον λειτουργική μονάδα μέσα στο pipeline του επεξεργαστή για να τις υποστηρίξει. Αυτή η προσέγγιση καταφέρνει να συναγωνιστεί και σε πολλές περιπτώσεις να ξεπεράσει τις αντίστοιχες επιδόσεις εργαλείων και μεθοδολογιών που στοχεύουν αρχιτεκτονικές RISC-V, καθώς και άλλες IoT συσκευές.

Ορισμένες προτάσεις, ώστε να βελτιωθούν περαιτέρω τα αποτελέσματα μας μελλοντικά αφορούν την:

- Πιο αποτελεσματική εξερεύνηση του χώρου λύσεων μέσω της ανάπτυξης μιας ταχύτερης στρατηγικής, βασισμένης σε στατιστικές τιμές για την εκτίμηση της ευαισθησίας κάθε

στρώματος και την αποτελεσματική ταυτοποίηση της βέλτιστης διαμόρφωσης για την ακρίβεια των βαρών.

- Ενσωμάτωση post-load increment εντολών για την απλούστευση της διαχείρισης δεδομένων εντός βρόχων και επαναλαμβανόμενων λειτουργιών.
- Υλοποίηση του επεξεργαστή σε Application-Specific Integrated Circuit (ASIC) τεχνολογίες, για ακόμα καλύτερες επιδόσεις όσον αφορά την ταχύτητα και την ενέργεια.
- Υλοποίηση πολλαπλών πυρήνων της RISC-V αρχιτεκτονικής που κατασκευάσαμε πάνω στην ίδια FPGA πλακέτα, για να αυξήσουμε την παραλληλοποίηση των αλγορίθμων μας.
- Εφαρμογή της μεθοδολογίας μας σε διαφορετικούς RISC-V επεξεργαστές, για να αποδείξουμε την εγκυρότητα και ευρύτητα της.

Chapter 1

Introduction

Recent advancements in deep learning (DL) have fueled an exponential growth in artificial intelligence (AI) applications and services, ranging from personal assistants to recommendation engines to video and audio surveillance systems. Furthermore, the expansion of mobile computing and the Internet of Things (IoT) has led to billions of mobile πbeing interconnected, producing vast amounts of data at the network's edge. Motivated by this evolution, there is a pressing need to extend AI capabilities to the edge of the network. Doing so is essential for tapping into the vast potential of big data generated at the edge, enabling real-time data processing, analysis, and decision-making closer to where data is created. This shift aims to enhance efficiency, reduce latency, and support the development of more responsive and intelligent edge-based applications [1].

A wide array of methodologies falls within the expansive scope of AI, among which Machine Learning (ML) has emerged as a particularly prominent technique [2]. Traditionally, the deployment of ML has been power-intensive, necessitating substantial computational resources to achieve the desired level of accuracy. This requirement has historically restricted the application of ML to high-capacity devices, such as network nodes, equipped to handle such demands. However, the evolution of technologies like the IoT and edge computing has sparked a keen interest in adapting ML techniques for use in resource-constrained embedded devices [3].

The advent of Tiny Machine Learning (TinyML) marks a revolutionary convergence of ML (specifically, DL) with edge computing technology. TinyML facilitates the deployment of compact DL models onto small edge devices that are subject to resource limitations, including restricted computational power (with clock speeds typically in the tens of megahertz), minimal memory capacity, and a mere few milliwatts (mW) of power consumption. This innovation empowers devices to perform data analysis and interpretation locally, enabling them to act in real-time based on the insights gained. Moreover, TinyML has made it feasible to deploy pre-trained DL models onto these edge devices. This is achieved through the application of techniques such as quantization and pruning, which significantly reduce the size of DL models and optimize them for efficient infer-

ence, thus bridging the gap between advanced AI capabilities and resource-constrained environments [4].

Recently, RISC-V has emerged as a notable open-source alternative to traditional Control Process Unit (CPU) architectures. This development has positioned RISC-V as a formidable competitor of Intel, AMD, and ARM CPUs, across both 32-bit and 64-bit variants, primarily due to its royalty-free nature [33]. The defining and most significant attribute of RISC-V is its open-source Instruction Set Architecture (ISA). This feature enables individuals to customize it by adding their unique instructions and functionalities. Such flexibility is pivotal as it facilitates the creation of highly efficient, low-latency co-processors, functional units, and accelerators. Unlike traditional approaches, these enhancements can be integrated directly within the RISC-V architecture, eliminating the complexities associated with treating them as external devices reliant on memory mapping and interrupts for communication. This inherent adaptability of RISC-V not only democratizes processor design but also significantly accelerates innovation in customized computing solutions.

1.1 Contributions

In this work, we utilize the flexible RISC-V eco-system to provide an end-to-end framework designed to deploy Deep Neural Networks (DNNs) on RISC-V cores. Our primary objective is to optimize the performance of these models by reducing their latency, memory footprint and power consumption with respect to the initial RISC-V processor setup. The first step involves the quantization of the Network's floating point (FP) parameters. We apply a Mixed Precision Quantization scheme, wherein the weights of each layer are converted into integers, each with a distinct bit-width resolution. Through an exhaustive search of the design space, we identify configurations that significantly enhance the model's speed without compromising its original accuracy. To maximize performance on the RISC-V core, we decide to extend the processor's supported instruction set. The instructions we introduce are targeting a custom functional unit that is integrated in the processor's pipeline and is specialized in executing Deep Neural Network (DNN) algorithms. Our approach was evaluated in a variety of DNN architectures and dataset and it manages to compete and in many cases outperform state-of-the-art methodologies and other popular frameworks dedicated to incorporating Neural Networks (NNs) in microprocessors and RISC-V cores.

1.2 Thesis Outline

This document consists of 6 main chapters. In chapter 2, we are briefly going to present some state-of-the-art methodologies that aim at optimizing the inference of DNNs on edge devices. Some of these works will later be used as the baseline for our final results. In

chapter 3 we will dive into the theoretical background needed in order to gain a better understanding of the concepts, techniques and structures that are used throughout this work. Following, in chapter 4 we focus on the frameworks and tools that will be used in our approach, as well as the Ibex processor that is the "heart" of our work, since this is the selected RISC-V core that we are going to modify and test. Chapter 5 is the main section of this work, in which we thoroughly describe the entirety of our methodology and the optimization steps that resulted in the final architecture. In chapter 6, we are going to present all the experimental results that were conducted and compare them to related works. Finally, chapter 7 evaluates the performance of our design and provides some suggestions in order to further optimize the results in the future.

Chapter 2

Related Work

As NNs continue to garner increasing interest, several frameworks like PyTorch [34], TensorFlow [35], and Caffe2 [36] have emerged, primarily designed for training NNs and deploying them at scale in GPU-accelerated data centers. Recently, however, there has been a significant shift towards optimizing neural network (NN) inference for low-power edge devices, particularly microcontrollers (MCUs) and RISC-V cores. In this context, we will provide an overview of some widely-used frameworks and state-of-the-art methods that aim at deploying NN models on these resource-constrained platforms.

Several leading frameworks, such as CMSIS-NN from ARM [5], TensorFlow Lite for microcontrollers (TFLite Micro) from Google [6], and X-CUBE-AI from STMicroelectronics [7] are widely used commercially in order to integrate ML algorithms into MCUs. These frameworks offer a variety of optimization techniques (quantization and weight pruning) and optimized kernels that allow practitioners to compress the size of the models and minimize their latency when running on edge devices. MCUNet is another state-of-the-art work that provides a sophisticated approach that not only searches for the optimal architecture of the networks, but also implements specialized functions to maximize performance and efficiency on IoT systems [8]. Nonetheless, these works confine their Design Space Exploration primarily to dataflow optimizations, focusing either on the memory hierarchy or on strategies for loop unrolling and tiling.

To further push performance, several studies [9], [10] introduce dedicated DNN accelerators. However, this approach often compromises on providing comprehensive support for instruction-driven MCUs on ultra-edge platforms, affecting versatility across a wide range of IoT nodes. With the increasing prominence of RISC-V cores in recent years, there has been a notable surge in research focusing on exploring RISC-V architectures for NN acceleration. It has been observed that enhancing the throughput of RISC-V processors for quantized DNNs demonstrates significant potential over the use of dedicated DNN accelerators. These strategies are centered around expanding the RISC-V ISA by introducing new instructions specifically designed to expedite the inference process of Quantized Neural Networks (QNNs) on RISC-V cores. Some notable works include FANN-on-MCU [11],

RedMulE [12] and Dory [13] that enable the inference of lightweight and energy-efficient NNs on the RISC-V-based PULP platform processors. These works enable only 8-bit or higher precision, which can hardly leverage the computational reduction potential of lower-bit quantized DNNs. RISC-V processors, such as the ones introduced by PULP-NN [14], achieve notable speed improvements by facilitating lower-bit DNN inference, through the use of instructions designed to pack and extract vectors of smaller data sizes, such as 4-bit and 2-bit, while utilizing the same SIMD MAC units. However, these casting instructions incur overheads for computations based on 4-bit and 2-bit sizes, which in turn reduce the performance gains for these lower bit widths. On the other hand, its extension XpulpNN [15] supports 2, 4, and 8-bit SIMD operations but requires that both operands are of equal precision.

Many of the current CPU and GPU architectures exhibit limited capabilities for mixed-precision neural networks, as these systems are primarily optimized for 8, 16, and 32-bit data types. As a result, the vast majority of accelerators and established software frameworks for neural network training and deployment are tailored towards implementations that utilize higher-precision formats, offering only broad strategies for integrating mixed-precision techniques [16], [17]. There exist some works that emphasize on optimizing coarse-grained (per-network or per-layer) mixed-precision operations with sub-byte parameters, such as BARVINN [18], which is an FPGA based neural network accelerator that communicates with a RISC-V core and Dustin [19] that extends the RV32IC to perform mixed-precision operations. However, these approaches often overlook critical factors like area and resource utilization, which are essential for more efficient implementations.

Chapter 3

Theoretical Background

3.1 Deep Neural Networks

3.1.1 Machine Learning - Introduction

In the dynamic landscape of technological advancement, ML stands at the forefront, driving innovation and transforming the way we perceive and interact with the world. Whether it's predicting stock market trends, recognizing speech, or recommending personalized content, the applications of ML are vast and diverse, permeating through various industries such as healthcare, finance, marketing, and more.

ML is a subfield of AI that involves the development of algorithms and statistical models that enable computers to improve their performance in tasks through experience. These algorithms and models are designed to learn from data and make predictions or decisions without explicit instructions. It combines elements stemming from the fields of computer science, statistics as well as domain-specific knowledge to create algorithms that not only decipher complex patterns within data but also adapt and improve over time.

ML can be broadly categorized into four types based on the learning styles and approaches [37]:

- **Supervised Learning**: the algorithm is trained on a labeled dataset, where the input data is paired with corresponding output labels. After the training is completed, the model is usually tested in new unseen data, in order to estimate its performance. It is most commonly applied in classification tasks, such as object detection, and regression problems.

- **Unsupervised Learning**: in this case the algorithm is dealing with unlabeled data and is trying to discover hidden patterns and structures within the training data, without any human interference. This method is preferred in clustering problems, dimensionality reduction tasks or anomaly detection.

- **Semi-supervised Learning**: this is a paradigm that falls between supervised and unsupervised learning. The ML model is trained on a dataset that contains both labeled and unlabeled data. The labeled data allow the model to understand the relationships between the input data and the expected outputs, while the unlabeled inputs help the model generalize and discover underlying structures. Semi-supervised training is suitable in cases in which obtaining labeled data is a time-consuming task and unlabeled data are numerous.

- **Reinforcement Learning**: this type of algorithm enables software agents to interact with the environment around them, make decisions and learn through trial-and-error. Based on the outcome of its decisions, the model receives a penalty or a reward and adjusts its parameters accordingly, in order to improve its efficiency. This type of strategy has been applied in the gaming industry or even in more sophisticated systems, such as robotics or autonomous driving.

3.1.2 Deep Learning

Over the recent years, the ML community has widely recognized the DL computing paradigm as the gold standard and it has emerged as the predominant approach in the field. Exceptional outcomes on demanding tasks have been attained by developers and researchers, and in certain instances, they have succeeded in generating models that equal or surpass human capabilities in these specific tasks. DL constitutes a subfield of ML (see Figure 3.1), concentrating on the utilization of artificial neural networks (ANNs) characterized by multiple layers [20].

Artificial Neural Networks

ANNs are computational models that have drawn inspiration from the structure of a human's brain. The main building block of ANNs is called a neuron and it mimics the functionality of its biological counterpart. Neurons are organized into layers, with an input layer receiving data, one or more hidden layers processing them, and an output layer producing the final result. Nodes between consecutive layers are connected to each other via synapses called weights, in order to form the Network.

Each neuron receives multiple input data and produces an output value, which can be calculated by applying a nonlinear function on the weighted sum of its inputs plus a learnable bias term, as described in Equation 3.1[38].

$$y = f\left(\sum_{i=1}^{n} W_i u_i + b\right) = f\left(W^T u + b\right) \tag{3.1}$$

The function f on the equation is called activation function and allows the model to discover more complex relationships within the input data. These functions decide if the neuron

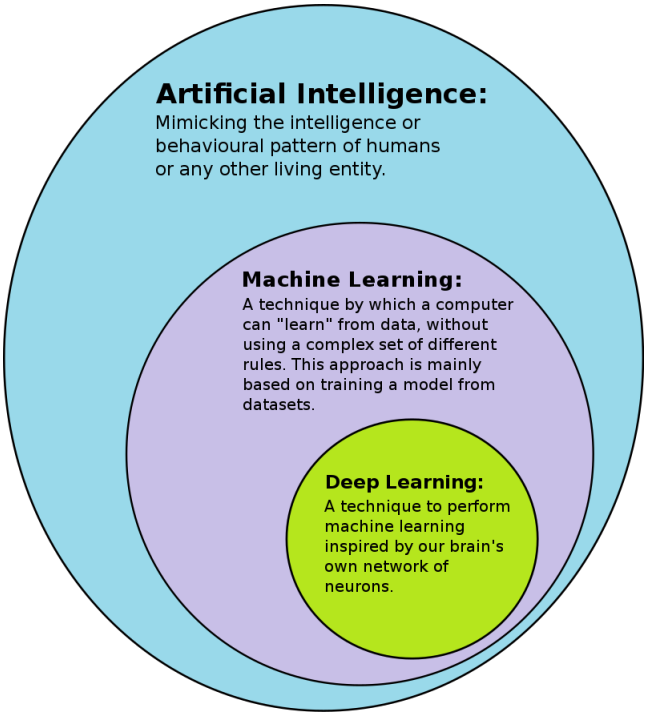


Figure 3.1: Deep Learning Family

should be activated or not, based on the magnitude of the signal they receive. Some of the most commonly used activation functions in modern NNs are:

- Sigmoid (logistic) function
- Hyperbolic Tangent (Tanh)
- Rectified Linear Unit (ReLU)
- Leaky ReLU

The mathematical formulas representing these activation functions, along with their graphical representations, are depicted in Figure 3.2.

Multi-layer Perceptrons

A Multi-layer Perceptron (MLP) represents the foundational architecture of an Artificial Neural Network (ANN) and is composed of several Dense or Fully Connected layers. Within these layers, every node is interconnected with every node in the following layer, establishing a densely connected network. Such comprehensive connectivity enables MLPs to discern intricate relationships and patterns within the data, rendering them highly adaptable and broadly applicable across various fields. The output of node i in the k-th layer can be calculated using the formula:

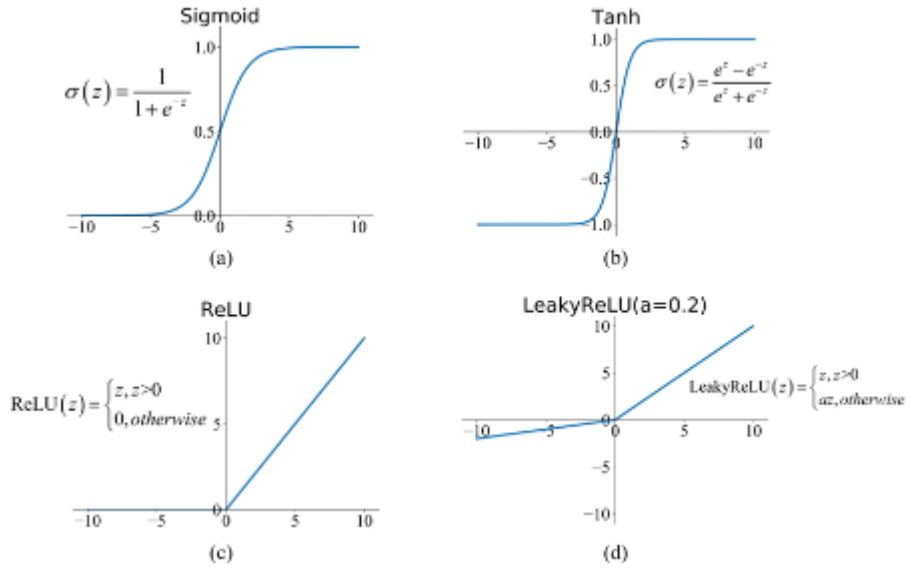


Figure 3.2: The graphical curves of the most common activation functions : (a) Sigmoid, (b) Tahn, (c) ReLU, and (d) Leaky ReLU.
Source: [21]

$$a_i^k = f\left(\sum_{j=1}^{N_{k-1}} w_{ij}^k \alpha_j^{k-1} + w_{i0}^k\right) \tag{3.2}$$

While Multi-layer Perceptrons (MLPs) are recognized as universal approximators capable of being trained to replicate any given nonlinear input-output mapping, they often encounter challenges when processing sequential or spatial data. This limitation has prompted researchers to develop specialized architectures designed specifically for handling these types of data [22].

Convolutional Neural Networks

Convolutional Neural Networks (CNNs) stand as a transformative advancement in the realm of DL. Inspired by the organization of the human visual system, these algorithms have been tailored for the processing of visual data. CNNs leverage principles from linear algebra, particularly convolution operations, to extract features and recognize patterns within visual information. Because Convolutional Networks are so effective at identifying objects, they are primarily deployed in computer vision tasks, such as image recognition and object detection, with common use cases including self-driving cars, facial recognition and medical image analysis. However, their adaptability extends to handling audio and other forms of signal data as well.

CNNs mainly consist of 3 components: convolutional, pooling and fully connected layers. The typical architecture of a CNN is illustrated in Figure 3.3.

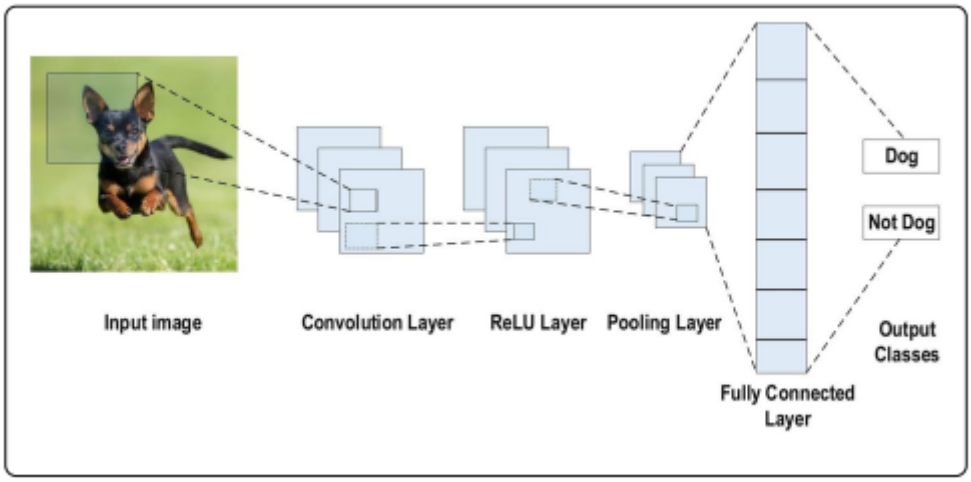


Figure 3.3: Structure of a typical Convolutional Neural Network.
Source: [20]

The convolutional layer is the fundamental building block of a CNN and is where the majority of computations occur. This layer performs the dot product between a matrix of learnable parameters otherwise known as a kernel and a restricted portion of the receptive field of an input image. The output feature map that will be produced, contains information about the presence of specific features [20]. These features could vary from basic image characteristics such as edges, textures and colors, especially in the earlier convolutional layers, to more complex structures, such as objects and shapes in later stages.

Pooling layers serve as crucial components strategically positioned after Convolutional Layers. Their primary purpose is to downsample the spatial dimensions of the feature maps, contributing to a reduction in the computational complexity of the network. This downsampling, in turn, offers advantages such as faster training and more efficient inference times, making pooling layers integral for enhancing the overall performance and efficiency of CNN architectures. Furthermore, it is useful for extracting dominant features which are rotationally and positionally invariant, thus maintaining the process of effectively training the model. There are two types of pooling layers that are mainly used in modern CNNs:

- **Max Pooling**, that returns the maximum value from the portion of the image covered by the kernel and
- **Average Pooling**, that returns the average of all the values from the portion of the image covered by the kernel.

A detailed illustration of how the two pooling functions operate is provided in Figure 3.4.

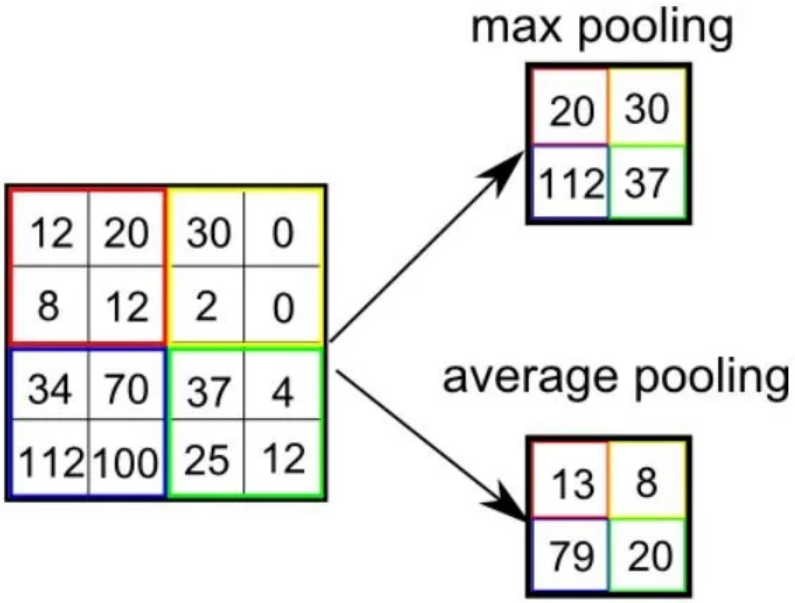


Figure 3.4: Comprehensive example of Max and Average Pooling.

In recent years, the evolution of CNNs has been driven by a multitude of challenges and a growing demand for enhanced performance in various fields. The increasing complexity of tasks, especially in the realms of computer vision and pattern recognition, has necessitated advancements in the architecture and capabilities of CNNs. These challenges include the need for improved accuracy, efficiency, and adaptability to diverse data types and modalities.

While on the one hand, deeper and more complex networks can often perform better, training them efficiently is a very time consuming and challenging task, due to problems like vanishing or exploding gradients. Architectures like ResNet [39] have introduced the concept of residual connections as an elegant solution to this problem. These shortcut connections provide a direct path for the input to bypass the block's internal transformations and reach the output (see Figure 3.5). The residual learning concept that involves learning the difference between the input and output has had a profound impact on the training of DNNs, enabling the development of deeper and more expressive architectures that can effectively learn complex representations from data.

Expanding the size of NNs is a typical tactic to boost their performance. However, such scaling up requires more computational power. When deploying on resource-limited platforms, such as mobile or embedded devices, the increased demands of larger models pose a significant challenge. Furthermore, time-critical applications, like autonomous vehicles or augmented reality, struggle to satisfy their stringent temporal constraints when operating on devices with constrained computing resources. An effective strategy employed in models such as MobileNets [40] involves the use of Depthwise Separable Convolutions, which serve as a more efficient alternative to standard convolutional layers.

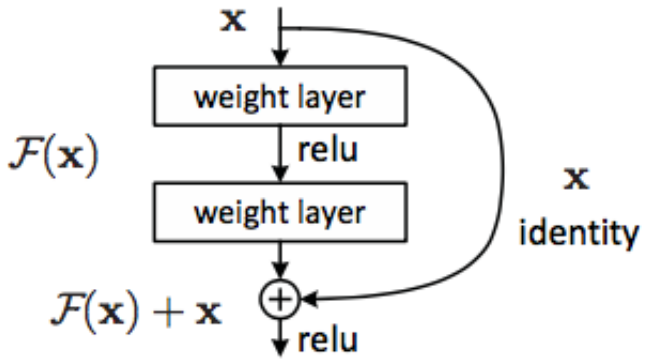


Figure 3.5: ResNet Block.
Source: [39]

A conventional convolutional layer accepts a feature map F with dimensions DF × DF × M, where DF represents the spatial width and height of the square input feature map and M denotes the number of input channels (input depth). This layer produces an output feature map G of dimensions DF × DF × N, with N being the number of output channels (output depth). At the same time, the operation of these layers is defined by a convolutional kernel K, which is sized at DK × DK × M × N. Here, DK is the square spatial dimension of the kernel, and M and N are the numbers of input and output channels, respectively, as previously mentioned.

Assuming that stride is equal to one and that padding is applied, the output feature map can be computed as:

$$G_{k,l,n} = \sum_{i,j,m} K_{i,j,m,n} \cdot F_{k+i-1,l+j-1,m} \tag{3.3}$$

The computational cost of the standard convolutions is:

$$DK \cdot DK \cdot M \cdot N \cdot DF \cdot DF \tag{3.4}$$

Depthwise Separable convolutional layers are a variant of convolutional layers that simplify the computation and reduce the number of parameters, while maintaining the model's accuracy at the same level. This approach factorizes the standard convolution into:

- a depthwise convolution and
- a pointwise convolution (1x1 convolution).

Unlike standard convolutional layers that apply a filter across all input channels and combine the results, depthwise convolutions perform a single convolution for each input channel. Depthwise convolution with one filter per input channel can be written as:

$$\hat{G}_{k,l,m} = \sum_{i,j} \hat{K}_{i,j,m} \cdot F_{k+i-1,l+j-1,m} \tag{3.5}$$

where $\hat{K}$ is the depthwise convolutional kernel of size DK × DK × M where the m-th filter in $\hat{K}$ is applied to the m-th channel in F to produce the mth channel of the filtered output feature map $\hat{G}$. The depthwise convolution has a computational cost of:

$$DK \cdot DK \cdot M \cdot DF \cdot DF \tag{3.6}$$

Depthwise convolution is notably efficient when compared to standard convolution. Nevertheless, it exclusively filters input channels and it does not combine them into creating new features. So an additional layer, which performs pointwise convolution is needed, in order to generate these new features. In other words, this layer calculates a linear combination of the outputs of the depthwise convolution via applying a 1×1 convolution. The total cost of the Depthwise separable convolutions is the sum of the depthwise and 1×1 pointwise convolutions:

$$DK \cdot DK \cdot M \cdot DF \cdot DF + M \cdot N \cdot DF \cdot DF \tag{3.7}$$

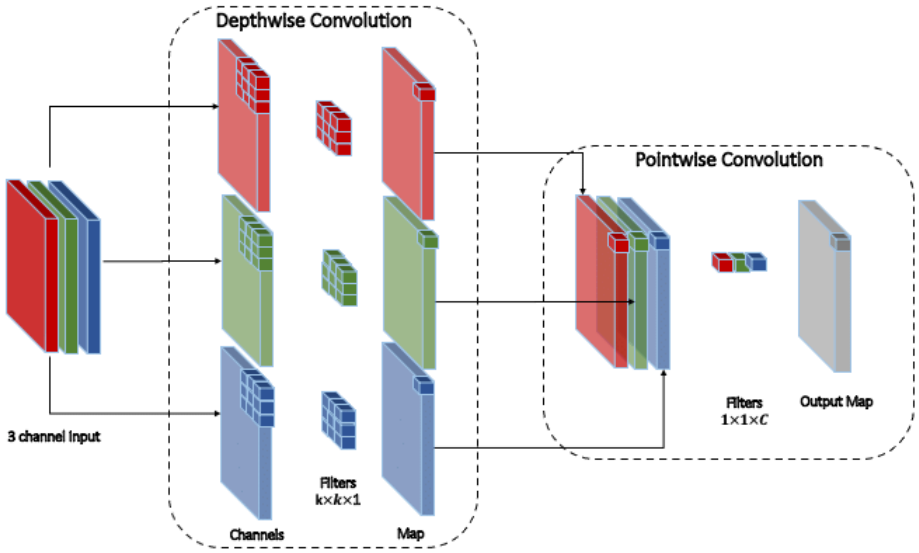


Figure 3.6: Depthwise Separate Convolution.

which is significantly lower than the one in standard convolutions.

Recurrent Neural Networks

Recurrent Neural Networks (RNNs) stand at the forefront of sequential data processing within the realm of AI. Such data can be handwriting, genomes, text or numerical time series which are often produced in industry settings. Nonetheless, they can also be employed

on images when they are decomposed into a sequence of patches and processed accordingly [41]. What separates them from other ANNs is how the information flows through them. RNNs have the unique ability to retain information from previous time steps, creating some type of memory, hence making them exceptional when handling sequential inputs.

There are other sub-classes of RNNs, most notably the Long Short Term Memory (LSTM) and the Gated Recurrent Unit (GRU) architectures, which have gained preference over traditional RNNs, primarily due to their ability to address certain limitations associated with RNNs, such as the vanishing gradient problem and the challenge of retaining information over long sequences of data.

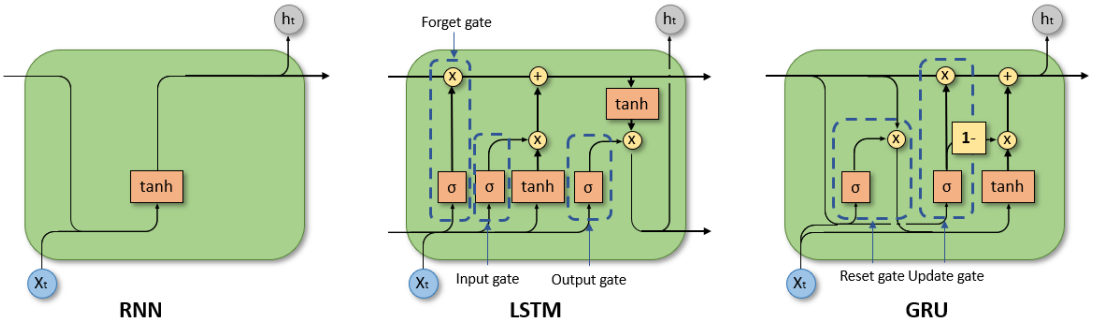


Figure 3.7: RNN, LSTM and GRU Cell Structure.

3.2 Neural Network Quantization

The substantial computational demands inherent in deploying NNs on resource constrained embedded electronic devices present a significant challenge. In response to this, NN quantization surfaces as a strategic methodology aimed at compressing models and alleviating computational burdens. However, it is crucial to acknowledge that quantization introduces trade-offs, as it may result in accuracy degradation compared to the original model. This delicate balance between computational efficiency and model precision underscores the nuanced considerations inherent in the adoption of quantization for real-world deployment on constrained devices.

In the process of NN quantization, the parameters of the network, both weights and activations, undergo storage in a bit precision lower than the conventional 16 or 32 bits used during training [23]. Utilizing lower-precision formats during inference yields several advantages:

1. **Decreased Latency**: Lower-precision formats result in lower latency, leveraging processors that exhibit higher throughput for arithmetic operations with fewer bits. This acceleration is particularly beneficial for intensive mathematical operations like matrix multiplications and convolutions.

2. **Diminished Memory Footprint**: Storing the network's parameters in lower bit precision contributes to a reduced memory footprint. This not only conserves memory space but also enhances the efficient utilization of the system's caches.

3. **Energy Efficiency**: Lower precision facilitates faster algorithm execution and reduces memory accesses, leading to decreased power consumption. This energy efficiency is a valuable outcome of employing lower-precision formats during inference.

3.2.1 Quantization Fundamentals

Initially, it is essential to establish a systematic method to transform real values, which are expressed as FP numbers, into a narrower precision range. Our emphasis will be on Uniform Quantization, the most widely used quantization technique known for facilitating efficient fixed-point implementation arithmetics.

Uniform Quantization is characterized by three parameters: the scale factor (S), the zero point (Z), and the bit width (b). The scale factor and zero point play a pivotal role in mapping real values onto an integer grid, the size of which is determined by the bit width. Typically represented as a FP number, the scale factor dictates the quantizer's step size. On the other hand, the zero point, an integer value, guarantees accurate quantization of the real zero, a crucial aspect for operations such as zero padding or ReLU to avoid introducing quantization errors.

There are 2 types of Uniform Quantization: Asymmetric and Symmetric. The main difference between the 2 is that in the latter case, the zero point value is equal to 0. First we map the real-valued vector $\mathbf{x}$ to the unsigned integer grid $\{-2^{b-1}, \ldots, 2^{b-1}-1\}$:

$$x_{\text{int}} = \text{clamp}\left(\left[\frac{x}{S}\right] + Z; -2^{b-1}, 2^{b-1}-1\right) \tag{3.8}$$

where $[\cdot]$ is the round-to-nearest operator and clamping is defined as:

$$\text{clamp}(x; a, c) = \begin{cases} a, & x < a \\ x, & a \leq x \leq c \\ c, & x > c \end{cases} \tag{3.9}$$

For the case of unsigned integers, the quantization scheme is described by the following equation:

$$x_{\text{uint}} = \text{clamp}\left(\left[\frac{x}{S}\right] + Z; 0, 2^{b}-1\right) \tag{3.10}$$

Unsigned symmetric quantization is particularly appropriate for distributions that exhibit a single tail, as observed in ReLU activations. Conversely, signed symmetric quantization is a suitable choice for distributions that demonstrate approximate symmetry around zero, a common scenario when quantizing the weights and biases of a network [42].

The values of the scale factors and zero points are highly dependent on the clipping range $[\alpha, \beta]$ of the FP values and are computed as described in the following equations:

$$S = \frac{\beta - \alpha}{2^b - 1} \tag{3.11}$$

$$Z = -[\beta \cdot S] - 2^{b-1} \tag{3.12}$$

As we can see, it is essential to determine the clipping range before calculating the scale and zero-point values. This procedure is called calibration. In asymmetric quantization, a very simple but effective approach is to use the minimum and maximum values of the signal, i.e., $\alpha = r_{\text{min}}$ and $\beta = r_{\text{max}}$. In the case of symmetric quantization, we choose a symmetric clipping range of $\beta = -\alpha = \max(|r_{\text{min}}|, |r_{\text{max}}|)$.

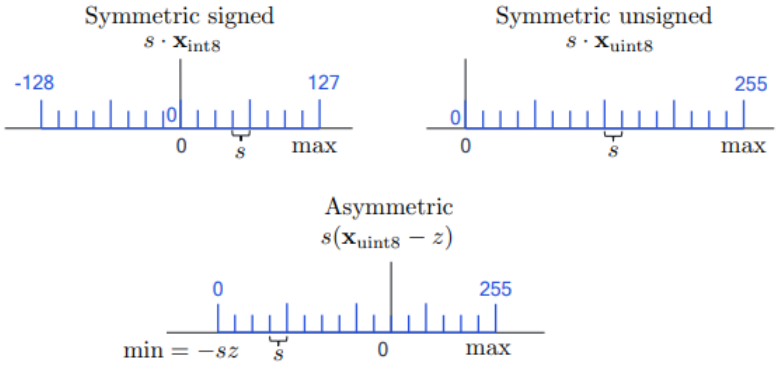


Figure 3.8: 8-bit Mapping across the different types of Uniform Quantization.
Source: [42]

In CNNs, the convolutional filters can have a different range of values. Consequently, a distinguishing factor among quantization methods lies in the granularity with which the clipping range $[\alpha, \beta]$ is determined for the weights. The first method is called **per-tensor quantization**, in which only a single set of quantization parameters (scale factor and zero point) is selected. Even though this technique is associated with sub-optimal accuracy, it is often preferred, as it can easily be mapped into hardware. The second approach is referred to as **per-channel quantization** and is currently the established method for quantizing convolutional kernels. In this scenario, each channel is assigned a distinct scaling factor, ensuring enhanced quantization resolution and improved overall performance (see Figure 3.9).

3.2.2 Full Integer Quantization and Inference

In addition to the previously outlined benefits, Full Integer Quantization is put into action when deploying a ML model on a microcontroller or device, lacking support for FP and fixed-point arithmetic operations.

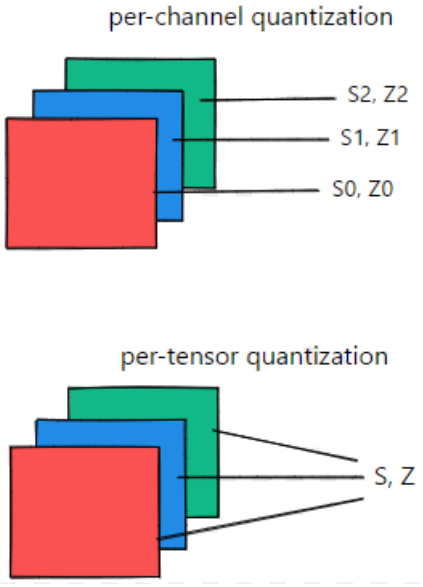


Figure 3.9: Per-Layer vs Per-Channel Quantization in CNNs.

The products that result from the matrix multiplications between the input and weight vectors are stored in an accumulator with higher bit width, typically 32-bits or more. This approach mitigates the risk of errors arising from overflow. The activations stored in the 32-bit accumulators need to be written to memory before they can be used by the next layer. To minimize data transfer and simplify the operations of the next layer, these activations undergo quantization back to a lower bit width. This step is called requantization. Figure 3.10 provides a simplified schematic of this whole procedure.

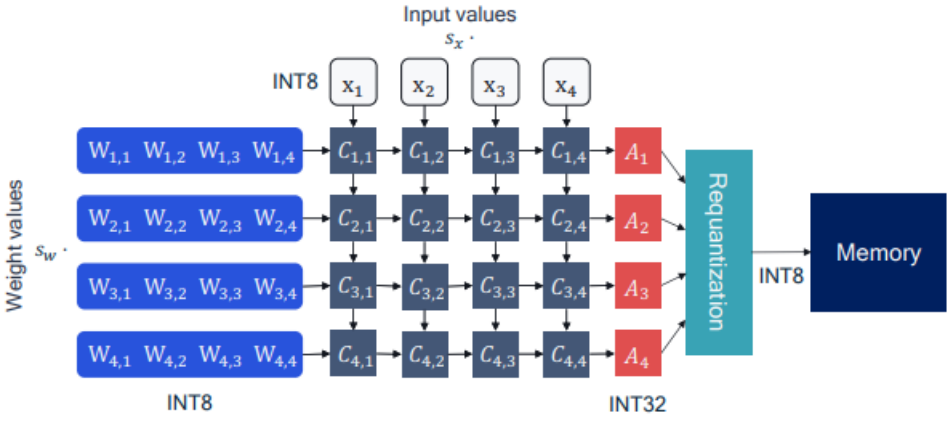


Figure 3.10: A schematic of matrix - vector multiplication and requantization back to 8 bits. Source: [42]

Consider the multiplication of two square $N \times N$ matrices of real numbers, r_1 and r_2, with their product represented as $r_3 = r_1 \cdot r_2$. We denote the entries of each of these matrices r_α (where $\alpha = 1, 2$ or 3) as $r_\alpha^{(i,j)}$ for $1 \leq i, j \leq N$, and the quantization parameters

with which they are quantized as (S_α, Z_α). We denote the quantized entries by $q_\alpha^{(i,j)}$. The relation between the real and the quantized values based on equation 3.8, can be expressed as:

$$r_\alpha^{(i,j)} = S_\alpha \cdot (q_\alpha^{(i,j)} - Z_\alpha) \tag{3.13}$$

The results from matrix multiplication can be given by:

$$S_3 \cdot (q_3^{(i,k)} - Z_3) = \sum_{j=1}^{N} S_1 \cdot (q_1^{(i,j)} - Z_1) \cdot S_2 \cdot (q_2^{(j,k)} - Z_2) \tag{3.14}$$

This equation can also be written as:

$$q_3^{(i,j)} = M \cdot \sum_{j=1}^{N} \left(q_1^{(i,j)} - Z_1\right) \cdot \left(q_2^{(j,k)} - Z_2\right) \tag{3.15}$$

where M is defined as:

$$M = \frac{S_1 \cdot S_2}{S_3} \tag{3.16}$$

In equation 3.15, the only FP value is the multiplier M, which is derived by the scaling values S_1, S_2, and S_3 as shown in equation 3.16. Instead of this formula, we can use the following approximate expression [43]:

$$M = M_0 \cdot 2^{-n} \tag{3.17}$$

where M_0 and n are both non negative integers. As a result, rather than performing a multiplication with a FP number, we will replace it with a multiplication with an integer value followed by a rounding bit shift operation. After scaling down the accumulated result, the last step is to cast it down to an unsigned integer value (uint8), if ReLU is the applied activation function, or to a signed 8-bit (int8) format otherwise.

3.2.3 Mixed Precision Quantization

Reducing the quantization to even lower bit widths (below 8 bits) can substantially enhance hardware performance. However, quantizing the entire network to extremely low precision may lead to a considerable loss in accuracy, making it unsuitable for critical applications. An elegant way to tackle this problem is to apply Mixed Precision Quantization.

In this hardware friendly approach, weights and activations in each layer are quantized with different bit widths. This way we can benefit from the lower bit width (lower latency, decreased power consumption and lower memory footprint), while minimizing accuracy degradation as much as possible.

One of the primary challenges associated with this technique is determining the optimal configuration for the bit widths of the weights in each layer. The search space expands exponentially with the number of layers in a network, making exhaustive search, particularly in large models like MobileNets, a time-consuming task.

Suppose we have an L-layers network, each layer has n optional bit-widths for weights and activations, the resulting search space is:

$$n^{2L} \tag{3.18}$$

Many different methods have been proposed in order to tackle this problem. Their main objective is to identify how sensitive each layer is to quantization. More sensitive layers are kept at higher precision, while more aggressive quantization is applied on layers with lower sensitivity [24]. Other solutions involve reducing the design space, in order to speedup the whole process.

3.2.4 Fine-Tuning Methods

Adjusting the parameters in the NN after quantization is frequently necessary. This adjustment can be accomplished through either retraining the model, known as Quantization-Aware Training (QAT), or without retraining, a process often referred to as Post-Training Quantization (PTQ).

QAT involves refining the model through additional training, with quantization in mind. During QAT, the quantization parameters (scaling, clipping, and rounding) are integrated into the training procedure. This incorporation allows the model to learn and adapt to maintain its accuracy even after quantization, resulting in a more efficient quantized model. During training, the forward and backward pass are performed in FP format, which is necessary in order to avoid zero gradient values. However the weights are quantized after each iteration to imitate the testing or inference phase [25]. Nonetheless, a significant drawback of QAT is the computational cost associated with retraining the NN model. This retraining process may necessitate several hundred epochs to regain accuracy, particularly when dealing with low-bit precision quantization.

PTQ is a quantization technique where the model is quantized after it has been trained. The Quantization parameters are determined with the use of a calibration dataset (representative of the data that model has been trained at), without any further fine tuning. As such, PTQ stands out as a fast method for quantizing NNs [26]. However, this process often entails a trade-off with lower accuracy when compared to QAT.

3.3 RISC-V

3.3.1 RISC-V Overview

In contrast to proprietary processor architectures, RISC-V distinguishes itself as an open-source ISA tailored for crafting customized processors designed for a variety of end applications. Originating from the University of California, Berkeley, the RISC-V ISA

represents the fifth generation of processors built on the principles of the reduced instruction set computer (RISC). Due to its open-source nature, RISC-V can be utilized both in academic and industrial applications [27].

RISC-V is touted for being a very small and efficient architecture, and at the same time has been defined to be easily extensible. In recent years, it has gained popularity because the architecture provides simplified instructions to the processor to accomplish various tasks. It also enables designers to create thousands of potential custom processors that specialize on a specific task, e.g NN Optimization.

3.3.2 Base User-Level ISA

The RISC-V foundation is characterized by its modest size, comprising merely 47 instructions that are obligatory for implementation. In contrast, the x86 architecture encompasses 1.503 instructions, and Arm features around 500. RISC-V adopts a straightforward load/store architecture, where all operations take place within internal registers, and there are dedicated instructions for transferring data between registers and memory.

The standard user-level integer ISA in RISC-V comes in two variations: RV32I and RV64I, offering 32-bit and 64-bit address spaces, respectively. In the base RISC-V ISA, instructions are of a fixed length at 32 bits, and they must align naturally on 32-bit boundaries [28]. There are 6 basic instruction formats that are shown in Figure 3.11:

- **R-type**: register to register
- **I-type**: short immediates and loads
- **S-type**: stores
- **B-type**: conditional branches, a variation of S-type
- **U-type**: long immediates
- **J-type**: unconditional jumps, a variation of U-type

3.3.3 RISC-V ISA Extensions

The RISC-V ISA is extendable, which means that developers, based on the requirements of their project, can evaluate if they need to use more instructions than the ones that base integer ISA offers. RV32I uses one-eighth of the encoding space. This means there's plenty of room for extensions.

Each base integer ISA can be extended with one or more of the standard optional instruction-set extensions defined by the RISC-V Foundation. Some of the most important ones are noted below [29]:

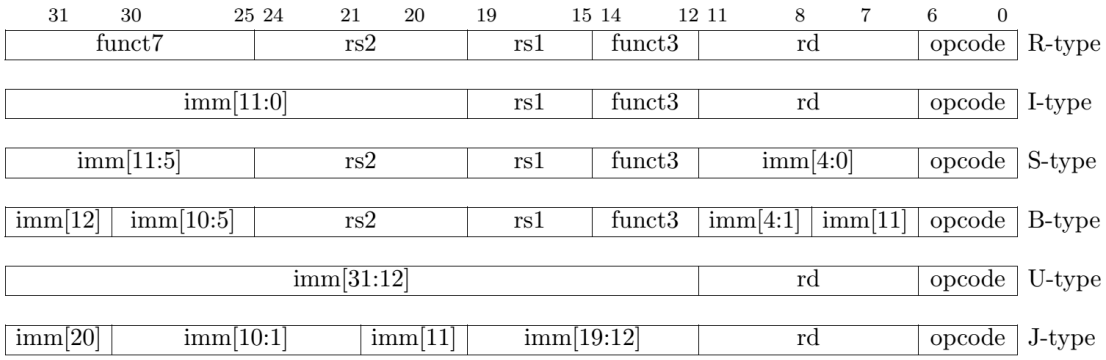


Figure 3.11: Basic RV32I ISA Instructions Format.
Source: [28]

- **M-Extension**: Instructions for Integer (signed and unsigned) multiplication and division.
- **A-Extension**: Atomic instructions that read, modify, and write memory atomically to provide synchronization across several RISC-V harts in the same memory space.
- **F-Extension**: Offers Single-precision FP instructions that are compliant with IEEE 754-2008 and adds another extra 32 registers for FP operations (each 32-bits in length).
- **D-Extension**: Includes instructions for double-precision FP compliant with IEEE 754-2008 and extends the width of the FP registers to 64-bits.
- **Q-Extension**: Adds support for instructions that implement Quad-precision FP operations compliant with IEEE 754-2008. Furthermore, the width of the registers is widened to 128-bits.
- **C-Extension**: Compressed instructions (16-bit instructions) to produce reduced code size.
- **Counters-Extension**: Provides up to 32 64-bit counters that can be accessed through specific read-only Control and Status registers (CSR) and can be very useful for counting the elapsed number of cycles or programmable event counting.
- **B-Extension**: It is a bit manipulation instruction-set extension.

Chapter 4

Utilized Tools and Frameworks

4.1 PyTorch

PyTorch, built upon the Torch library, is an open-source ML framework utilized in domains such as reinforcement learning, computer vision and natural language processing. Initially developed by Meta AI, it is currently under the Linux Foundation's umbrella. Although the Python interface is more polished and the primary focus of development, PyTorch also has a C++ interface.

Written in Python, it's relatively easy for most ML developers to learn and use. PyTorch stands out due to its outstanding support for GPUs and its utilization of reverse-mode auto-differentiation, allowing for dynamic modification of computation graphs. This makes it a popular choice for fast experimentation and prototyping.

PyTorch provides two high-level features [34]:

- Tensor computation (like NumPy) with strong GPU acceleration
- DNNs built on a tape-based autograd system

PyTorch, similarly to other popular ML frameworks, such as TensorFlow, use tensors and graphs as their core components. Tensors are a core PyTorch data type, similar to a multidimensional array, serving the purpose of storing and manipulating both the inputs and outputs of a model, along with the model's parameters. While sharing similarities with NumPy's data structures (namely *ndarrays*), tensors possess the added capability of running on GPUs, facilitating accelerated computing. Graphs are data structures consisting of connected nodes (called vertices) and edges. Every modern framework for DL is based on the concept of graphs, where NNs are depicted as a graph structure of computations (see Figure 4.1). While numerous frameworks, like TensorFlow, adopt static computation graphs, PyTorch distinguishes itself by employing dynamic computation graphs. In PyTorch, the computation graph is constructed and reconstructed during runtime. The same code responsible for executing computations in the forward pass is simultaneously engaged in creating the data structure necessary for backpropagation.

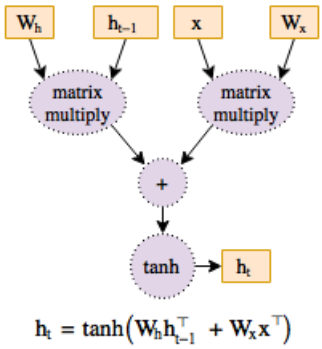


Figure 4.1: Example of a Neural Network Graph.

4.2 Brevitas

Brevitas is a PyTorch library that allows us to perform quantization to NN models and can support both PTQ and QAT. It has found successful application in numerous research initiatives and commercial deployments aimed at CPUs, GPUs, and custom accelerators operating on Xilinx FPGAs. The general quantization technique employed is affine quantization, with focus on uniform quantization. As of now, out-of-the-box support for non-uniform quantization is not available.

Brevitas provides the necessary tools and functions in order to implement and fine tune quantized models. More specifically, it offers quantized implementations of the most commonly used DNN layers, while at the same time it allows developers to configure their models with a variety of quantization schemes, e.g. power of two quantization, and different bit-widths for the weights and activations of each layer of the network. Lastly, the library has the built-in tools for researchers to create new QAT techniques and test their efficiency [44].

4.3 lowRISC/Ibex

Ibex is a small production-quality open source 32-bit RISC-V processor. Ibex was originally part of the PULP platform, under the name "Zero-risky" aimed at very low power applications. Nowadays it is maintained and further developed by lowRISC [30].

Ibex is being extensively verified and supports the Integer (I) or Embedded (E), Integer Multiplication and Division (M), Compressed (C), and B (Bit Manipulation) extensions. Its architecture and functionality are meticulously defined using SystemVerilog. Featuring a 2-stage pipeline, the core's initial stage is dedicated to Instruction Fetch (IF), followed by the Instruction Decode and Execution (ID/EX) stage. Additionally, an optional third stage, WriteBack, is available, tasked with writing the results of executed instructions back to the Register File or Memory, ensuring seamless operation and data integrity.

The core possesses various parameters that can be adjusted to align with the requirements of specific applications, enhancing its adaptability to diverse scenarios. This characteristic renders the core highly parameterizable. The options include different choices for the architecture of the multiplier unit, as well as a range of performance and security features. Specifically, regarding the multiplier unit, there are two distinct configurations available. The first setup conducts the multiplication between two 32-bit integers using only one 17×17 multiplier, which corresponds to the utilization of one Digital Signal Processor (DSP) in a FPGA design. The operation completes in 3 cycles, optimizing resource usage at the cost of increased operation time. In contrast, the second alternative allows for single-cycle multiplication by employing three 17×17 multipliers. This setup requires four DSPs—the additional DSP is essential for accumulating the partial products. This configuration offers a speed advantage at the expense of increased resource utilization.

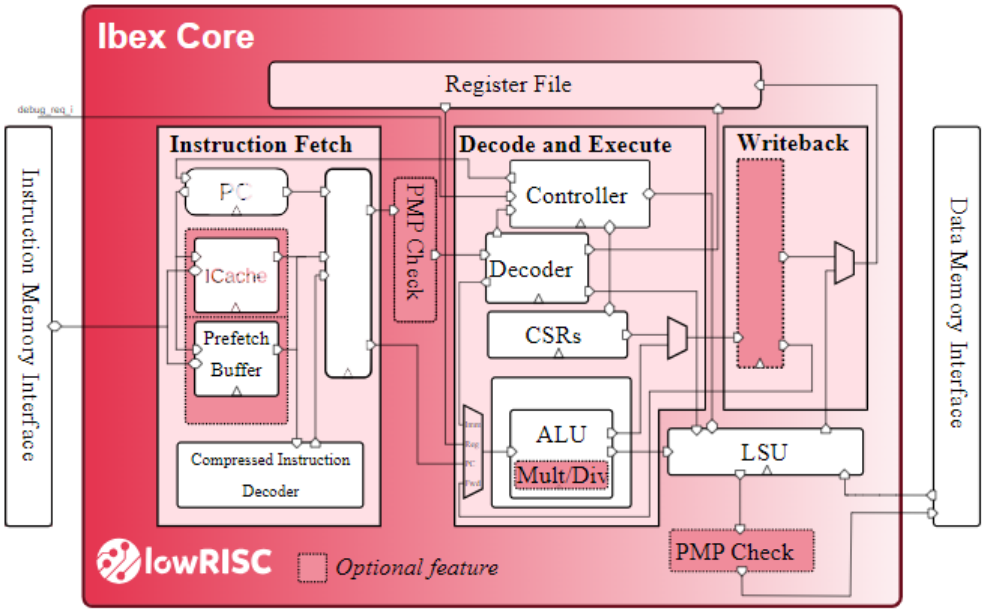


Figure 4.2: Schematic Diagram of the Ibex RISC-V Core.
Source: [30]

SPIKE RISC-V ISA Simulator

SPIKE is an open-source simulator that emulates the behavior of RISC-V processors. It is primarily designed for software development, testing, and debugging, providing a convenient environment for RISC-V software developers. It can simulate different variants of the RISC-V ISA, including RV32I, RV64I and various instruction set extensions [45]. Spike proves to be especially valuable for developers when introducing new instructions to the RISC-V ISA, as it provides confirmation that the RISC-V toolchain has been effectively updated to incorporate the new instructions and ensures that the correct executable has been generated.

FuseSoC

FuseSoC is an award-winning open-source tool and framework designed for managing and building Hardware Description Language (HDL) codes. It provides a flexible and scalable environment for developing, sharing, and integrating Intellectual Property (IP) cores and can be an aid for creating, building and simulating System-on-Chip (SoC) solutions [46]. Ibex flows utilize FuseSoC to collect the necessary Register-Transfer Level (RTL) files and perform builds, automating the process of constructing the Ibex core for both simulation and deployment on FPGA or ASIC platforms. Furthermore, by specifying a set of parameters, different configurations of the RISC-V core can be built and evaluated.

Verilator

Verilator is an open-source tool used for the simulation and synthesis of digital designs described in Verilog or SystemVerilog. It operates as a cycle-accurate simulator and synthesizer, offering designers a powerful platform for modeling and analyzing digital circuits, by allowing them to simulate the behavior of their components at a detailed level, tracking changes from one clock cycle to the next. Verilator is known for its speed and efficiency, as it can outperform other closed-source commercial simulators.

Verilator is invoked using parameters similar to GCC or Synopsys's VCS. Verilator reads the HDL code, conducts lint checks, and optionally incorporates assertion checks and coverage-analysis points. The resultant Verilated C++/SystemC files are then compiled by a C++ compiler (such as gcc/clang/MSVC++), potentially alongside a user's custom C++/SystemC wrapper file, to instantiate the Verilated model. Executing the resulting executable facilitates the simulation of the design [47].

Verilator can be used in compliance with FuseSoc to simulate the functionality of the Ibex core when running an executable file (in the formats of .elf or .vmem). Following the simulation, we can verify the accurate execution of the program and obtain valuable insights into the program's latency, the count of retired instructions, and other relevant metrics.

Chapter 5

Configurable Mixed Precision RISC-V Architecture

In this section, we are going to thoroughly discuss the workflow used in this thesis. The primary goal centers on deploying compact, high-speed, and energy-efficient DNNs on RISC-V cores. Concurrently, there's a focus on minimizing the additional resource utilization required on the modified core, ensuring that the enhancements in performance and efficiency do not come at the expense of excessive resource consumption. This methodology is versatile and applicable across various types of DNNs. Nevertheless, our primary focus and analysis will be on MLPs and CNN architectures, as these are the structures that are most commonly utilized in ML applications, deployed on edge devices and microcontrollers, making them highly relevant for our exploration of efficient NN development.

Our methodology unfolds across three primary stages. Initially, the model parameters undergo a process of Mixed Precision Quantization (MPQ). To identify the optimal configurations that boost the model's operational speed while preserving its accuracy, we conduct a thorough exploration of the design space. To further enhance performance on the RISC-V core, we opt to extend the processor's supported instruction set. The newly introduced instructions are specifically designed to interact with a NN accelerator, seamlessly integrated within the processor's pipeline. As a demonstration of our approach's viability, we apply this methodology on the Ibex core. The resulting architecture will be mapped and evaluated on the Virtex-7 FPGA board provided by Xilinx.

The proposed methodology can be summarized in Figure 5.1

5.1 Model Quantization

The initial phase involves meticulously calibrating the precision of a designated NN and fine-tuning its variables prior to its integration into our device (❶ in Figure 5.1). As delineated in Section 3.2.3, the strategy of MPQ stands out as particularly advantageous. To streamline the range of design options, we have chosen to uniformly encode the input

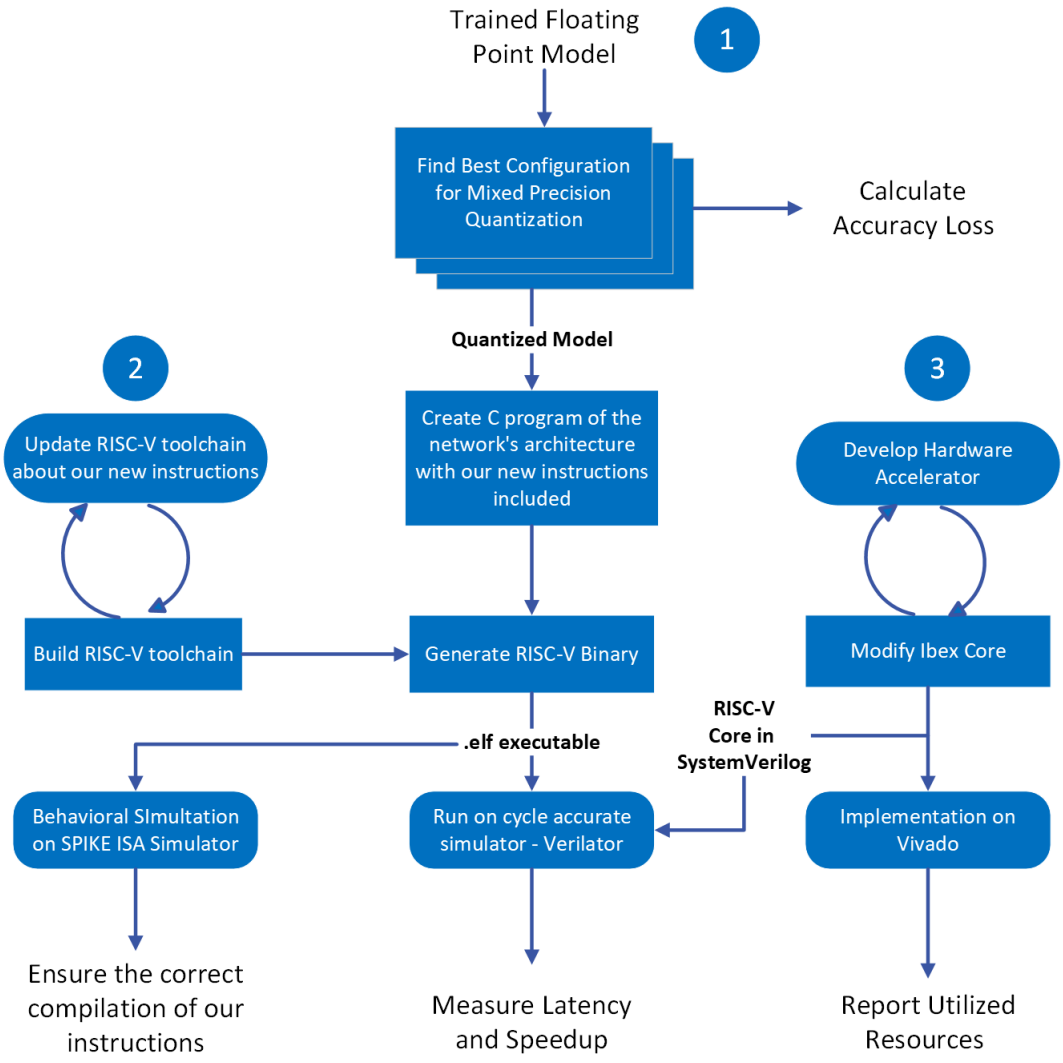


Figure 5.1: Flowchart of the proposed framework.

data and the activation values across all layers of our network using only 8-bit representations. These representations can accommodate both signed (int8) and unsigned (uint8) values. Regarding the network's weights, they can be quantified using signed numerical values with either 2-bit, 4-bit, or 8-bit resolution.

Considering the complexity of a given model and its efficacy on the dataset it's been trained at, the model's parameters are initially transformed into a quantized format using PTQ. Should the results from PTQ not meet the expected standards of performance, the process is further enhanced by refining the parameters through QAT until achieving the targeted level of performance. To streamline the QAT phase and ensure an efficient training process, several strategies are employed, including early stopping mechanisms. These mechanisms are activated when further training yields diminishing returns, thereby optimizing the training duration without sacrificing the model's accuracy.

5.1.1 Design Space Exploration

Since the bit width of the weights is the only parameter that we need to determine in each layer, the total number of configurations for the entirety of the model that we need to analyze, according to equation 3.18, is equal to 3^L, where L is the total number of parameterizable layers in the network. Exhaustive search spanning all conceivable configurations proves to be a robust methodology, especially when dealing with small NNs with a few number of layers. This method ensures that the selected precision configuration is not just a feasible solution but the optimal one within the defined constraints. Nevertheless, this strategy becomes impractical for large NNs, given the sheer volume of possible configurations that need scrutiny. This challenge, in combination with the substantial training times, typical of QAT, necessitates the exploration of alternative strategies for the quantization of larger-scale models.

A practical and efficient solution to address this challenge involves grouping sequential layers into blocks and assessing them as a single entity, thereby diminishing the range of the exploration space. Alternatively, we can set a fixed bit-width at levels whose computational load is negligible compared to the total computations, or we can discard solutions that would lead to a significant degradation in the model's accuracy. However, this reduction of the design space may come at a cost, as it risks bypassing certain configurations, potentially resulting in the adoption of a suboptimal precision setting for the model.

After a comprehensive evaluation of all potential configurations, the concluding phase involves selecting a Pareto-optimal solution. This solution is meticulously calibrated to strike a strategic balance between computational cost and model accuracy. Such an approach guarantees that enhancements in any one objective inevitably necessitate concessions in the other. Pareto-optimal solutions are distinct configurations positioned on the Pareto frontier, identifiable through graphical representation. These solutions are interconnected, forming a curve on the Pareto frontier that traverses through a multi-dimensional space, visually representing the optimal configurations between conflicting objectives (see Figure 5.2).

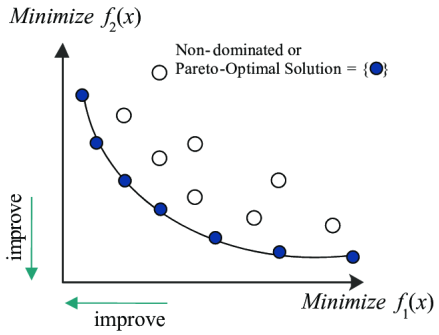


Figure 5.2: Pareto optimal solution of a multivariable problem.

Upon completion of this procedure, all essential parameters of the network, including

weights, biases, and quantization parameters (scales, zero points and the multiply and shift values for the requantization step), will be precisely defined and established.

5.2 Integrating new Instructions on the RISC-V ISA

The algorithm for the inference of a NN that we will implement and optimize can schematically be described in Figure 5.3.

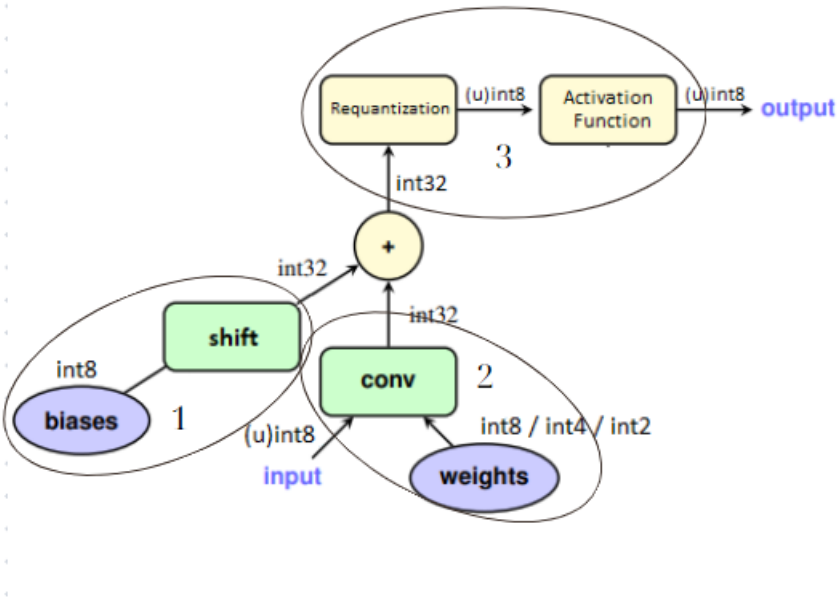


Figure 5.3: Integer only Inference

The computation of each layer's output in a NN is a process that can be divided into three distinct phases. The initial phase revolves around setting the biases for each neuron, or in the case of a convolutional layer, for each activation map. Typically, biases are stored in a higher bit width (32 bits), with their scale values derived from the product of scale factors between input and weight vectors. An efficient approximation can be achieved by representing these biases as 8-bit variables, followed by a left shift operation, a technique that effectively maintains the network's overall efficiency.

The second phase, which is computationally the most demanding, involves carrying out all the required multiplications between the input values and their corresponding weights. The multiplication results are then collectively accumulated and added to the previously set biases. These successive multiply-accumulate (MAC) operations are pivotal and represent the focal point of optimization in our research.

The final phase entails the requantization of the 32-bit accumulated result back into an 8-bit numerical value, which can be either unsigned or signed. As outlined in section 3.2.2 (equation 3.17), this requantization is accomplished through a multiplication followed by

a right shift operation. Subsequently, an activation function is applied to this 8-bit value, yielding the final output that will be passed on to the subsequent layer.

5.2.1 Custom Instruction Compilation

The RISC-V eco-system offers a flexible environment, which by enabling the augmentation of the processor's instruction set, allows us to create a bespoke system that can optimize the application of DNNs.

The initial action required in this process is to update the RISC-V toolchain, which contains the C (and C++) cross-compiler. This updated compiler will be responsible for generating the executable file, which will incorporate the newly introduced instructions. Since modifying the GCC compiler itself would prove too complex, we will instead modify the RISC-V GNU/GCC Binutils. By updating the Binutils, which are a collection of tools for handling binary files, we can add the new instructions required for our purposes. This approach, however, implies that the GCC compiler will not have full awareness or direct support for these newly added instructions [48].

As a result, to effectively utilize these instructions within our code, we will need to resort to using inline assembly. Inline assembly provides a way to embed assembly code within high-level language programs, allowing direct access to the processor's instruction set, including the custom instructions that we intend to add. To declare inline assembly functions the keyword "**asm volatile**" has to be used. By marking the instruction as "volatile" we ensure that the instruction will be utilized exactly as intended in the context of our application and will not be affected by the compiler's optimizations.

The new instructions to be added to the ISA will mirror the structure of the R-type arithmetic instructions found in the I-extension. To ensure distinct identification and proper functionality of each instruction, we will need to:

- **Provide a unique name**, which is crucial for clarity and ease-of-use. The name should preferably be indicative of its functionality.
- Define the necessary instruction fields. For the case of R-type instructions, we will need to specify:

    1. the opcode of the instruction: this field identifies the overall category of the instruction. Since our instructions will only be applied in NN algorithms, we will provide a new unique opcode to all of them.
    2. Function codes (`funct3`, `funct7`): these fields further specify the operation or variation of the instruction within its opcode category.
    3. Source and destination registers: these fields specify the registers that the inputs and outputs will be stored, during the runtime.

All pertinent information regarding the new instructions should be meticulously incorporated into the relevant files within the RISC-V toolchain (❷ in Figure 5.1). For instance, if we aim to introduce a novel R-type instruction named "*my_new_instruction*", then the files that will require updating are the following:

1. *path/to/riscv-gnu-toolchain/riscv-binutils-gdb/include/opcode/riscv-opc.h*

   where we declare the instruction and its structure, by defining the **MATCH** and **MASK** variables. MATCH is essentially a 32-bit hexadecimal number, incorporating the values for the opcode and the function codes. Meanwhile, MASK is another 32-bit hex value that indicates which bits in the instruction should be compared against the MATCH value to determine if the fetched instruction is the one in question. It's important to note that for custom instructions the two least significant bits of the MATCH value must be set to 1. The following piece of code showcases the changes that need to be made in the file.

```
#ifndef RISCV_ENCODING_H
#define RISCV_ENCODING_H

/* Instruction opcode macros.     */
/* New opcode =  0x27 (010 0111) */
/* funct3 = 010, funct7 = 0x0     */

#define MATCH_NEW 0x6027
#define MASK_NEW 0xfe00707f
...
#endif /* RISCV_ENCODING_H */

#ifdef DECLARE_INSN
DECLARE_INSN(my_new_instruction, MATCH_NEW, MASK_NEW)
...
#endif /* DECLARE_INSN */
```

2. *path/to/riscv-gnu-toolchain/riscv-binutils-gdb/opcodes/riscv-opc.c*

   In this scope we declare additional information, such as the name of the instruction, whether the instruction is targeted for only 32 or 64-bit RISCV variants, the type of the new instruction and its operands.

```
...
const struct riscv_opcode riscv_opcodes[] =
{
/* name, xlen, isa, operands, match, mask, match_func, pinfo.*/
{"my_new_instruction", 0,     INSN_CLASS_I, "d,s,t",
    MATCH_NEW, MASK_NEW, match_opcode, 0},
...
```

5.2.2 Description of the new instructions

The new instructions to be integrated into the RISC-V ISA can be classified in three different categories, each tailored to address one of the three phases we have analyzed respectively (Figure 5.3). These instructions will be embedded within the source codes that implement the various types of NN layers, thereby enhancing their operational efficiency and effectiveness. All of them will share the same opcode ($0x47$ or 100 0000 in binary form) and will be uniquely identified by the `funct3` and `funct7` fields.

The first instruction we will introduce to the ISA is designed to execute the initial phase of our outlined procedure, primarily focusing on establishing the biases for each output. This single cycle instruction will utilize two input registers, each holding 32-bit values. The first register is particularly structured to concatenate four 8-bit weights, forming a cohesive 32-bit value. The second register specifies the number of left bit shifts required in order to transform the original 8-bit weights into an approximated 32-bit value of the bias number that was produced during the QAT process (see Figure 5.4).

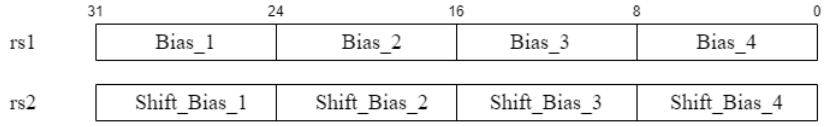


Figure 5.4: The format for the 32-bit registers when invoking the neur_init instruction.

Instruction Name	funct7	funct3	rs1	rs2	Description
neur_init	000 0000	100	4 8-bit biases	Amount of left shifts for each bias	Setting the biases for each output

Table 5.1: Instruction dedicated for the initialization of our custom component that set the biases of the outputs in each layer.

The second suite of instructions we plan to integrate into the RISC-V ISA is meticulously designed to facilitate the MAC operations pivotal to our algorithm. Within this phase, we intend to introduce a diverse array of instructions, each tailored to address distinct scenarios that arise from the combination of the weight's resolution (2-bit, 4-bit, or 8-bit) and the format of the inputs for a particular layer – either unsigned 8-bit or signed 8-bit (see Figure 5.5). Every instruction in this set will also be scheduled to be executed in just one cycle. This deliberate differentiation serves two purposes. The first one is for clarity reasons and ease-of-use as it enhances the readability and usability of the source code, while the second is a more practical one, as we can benefit from this categorization in order to trigger different signals during the decoding of each instruction. These signals are essential for the proper functioning of the specialized hardware accelerator, which will be incorporated into the processor's pipeline. The accelerator, in turn, utilizes these signals to accurately compute the outputs for each scenario.

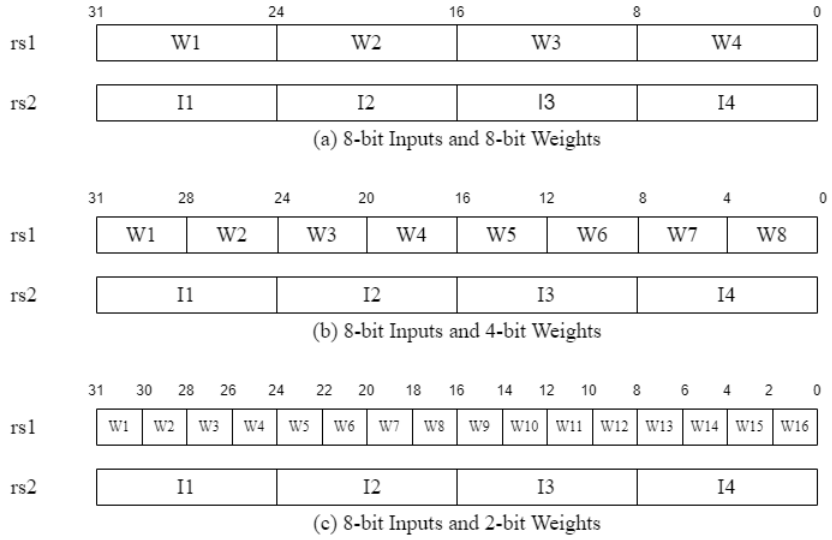


Figure 5.5: Structure of the register's content when calling the MAC instructions.

Instruction Name	funct7	funct3	rs1	rs2	Description
neur_mac_u_8b	000 1000	010	4 8-bit unsigned inputs	4 8-bit weights	4 MAC operations
neur_mac_u_4b	000 0100	010	4 8-bit unsigned inputs	8 4-bit weights	8 MAC operations
neur_mac_u_2b	000 0010	010	4 8-bit unsigned inputs	16 2-bit weights	16 MAC operations
neur_mac_s_8b	001 1000	010	4 8-bit signed inputs	4 8-bit weights	4 MAC operations
neur_mac_s_4b	001 0100	010	4 8-bit signed inputs	8 4-bit weights	8 MAC operations
neur_mac_s_2b	001 0010	010	4 8-bit signed inputs	16 2-bit weights	16 MAC operations

Table 5.2: List of Instructions dedicated for the acceleration of MAC operations.

The final collection of instructions we plan to introduce to the ISA will be used during the requantization step of the algorithm, where we transform the 32-bit accumulated results back to an 8-bit values. Each instruction in this set will operate on 32-bit words stored in the source registers. These 32-bit words are structured to encapsulate both the 8-bit values necessary for the multiplication process and the parameters for the right-shift operation that will be applied on every output of the respective layer (see Figure 5.6). Upon decoding, these instructions will interact with the custom unit in order to cast the outputs to the appropriate 8-bit format (signed or unsigned). These are the only multi-cycle instructions that will be added in the ISA, since their execution lasts 3-5 clock cycles (depending on if the execution of the MAC instructions has been successfully completed) and their output will be stored back in memory.

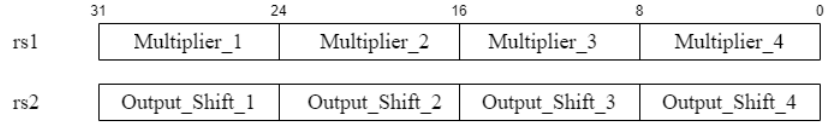


Figure 5.6: Structure of the register's content when performing the instructions for the requantization step.

Instruction Name	funct7	funct3	rs1	rs2	Description
neur_res_u	000 0001	001	4 8-bit multiplication values	4 8-bit values for right shift	Requantization and conversion to **uint8**
neur_res_s	000 0000	001	4 8-bit multiplication values	4 8-bit values for right shift	Requantization and conversion to **int8**

Table 5.3: List of instructions dedicated for the implementation of the requantization step.

5.2.3 A Practical Example

In the upcoming section, we'll present a brief example illustrating the integration of the newly introduced instructions within the source code for a straightforward Dense (Fully Connected) layer. This instance presumes that the layer's weights are quantized to an 8-bit precision and that both the inputs fed into the layer and the outputs generated will utilize an unsigned 8-bit (uint8) format.

The code snippet presented below is a customized version of the function that describes the behavior of this layer with the use of the basic RV32IMC ISA and is inspired by the implementation from the Tensorflow Lite Library [6]:

```
/* Requantization Function */
int requantize_relu(int accumulated_value, int multiplier,
        int right_shift){
    int32_t quantized_value = accumulated_value * multiplier;
    // Rounding Operation
    quantized_value += (1 << (right_shift - 1));
    quantized_value = quantized_value >> right_shift;
    // Apply ReLU activation function
    // and clamp the value to [0,255]
    quantized_value < 0 ? 0:(quantized_value > 255 ? 255:quantized_value);
    return quantized_value;
}

void mlp_layer(int input[],int output[],int num_inputs,int num_outputs,
    const int weights[][num_inputs],const int bias[], const int b_shift[],
    const int quantized_multiplier, const int out_shift){

    int z1, z2, z3, z4;
    for (int i = 0; i < num_outputs; i+=4) {
    /* Step 1: Set up the biases for each neuron */
        z1 = bias[i] << b_shift[i];
        z2 = bias[i+1] << b_shift[i+1];
        z3 = bias[i+2] << b_shift[i+2];
        z4 = bias[i+3] << b_shift[i+3];
        /* Step 2: Execute MAC operations */
        for (int j = 0; j < num_inputs; j++) {
            z1 += input[j] * weights[j][i];
            z2 += input[j] * weights[j][i+1];
            z3 += input[j] * weights[j][i+2];
```

```
            z4 += input[j] * weights[j][i+3];
        }
        /* Step 3: Requantize and apply activation function */
        z1 = requantize_relu(z1, quantized_multiplier, out_shift);
        z2 = requantize_relu(z2, quantized_multiplier, out_shift);
        z3 = requantize_relu(z3, quantized_multiplier, out_shift);
        z4 = requantize_relu(z4, quantized_multiplier, out_shift);
        output[i] = z1;
        output[i+1] = z2;
        output[i+2] = z3;
        output[i+3] = z4;
    }
}
```

We notice that we have applied the Loop Unrolling transformation technique on the outer loop. This optimization method unfolds the loop, executing multiple iterations within a single loop cycle, thus reducing the number of iterations and the associated control overhead. Furthermore, it can enhance the performance of critical sections within applications, due to the improved cache memory utilization. When compilers are invoked with specific optimization flags, they can automatically apply loop unrolling among other optimization techniques to enhance code execution efficiency. However, in this scenario, we are manually applying loop unrolling to gain a clearer understanding of the impact of the new instructions. These instructions are designed to operate on at least 4 data points in parallel, significantly optimizing computational processes. With the introduction of the new instructions, the code can be reconstructed as shown below:

```
void mlp_layer_8bits_relu(int input[], int output[], int num_inputs,
    int num_outputs, const int weights[][num_inputs], const int bias[],
    const int b_shifts[], const int quantized_multiplier,
    const int out_shift_rl){
    int z, w, inp, temp, quant_mul, shift;
    for (int i = 0; i < num_outputs >> 2; i++) {
        /* Step 1: Set up the biases for each neuron */
        asm volatile("neur_init %0, %1, %2\n":"=r"(z):"r"(bias[i]),
            "r"(b_shifts[i]):);
        for (int j = 0; j < num_inputs >> 2; j++) {
            /* Step 2: Execute MAC operations */
            w = weights[i][4*j];
            inp = input[j];
            asm volatile("neur_macc_u_8b %0, %1, %2\n":"=r"(temp):"r"(w),
                "r"(inp):);
            w = weights[i][4*j+1];
            asm volatile("neur_macc_u_8b %0, %1, %2\n":"=r"(temp):"r"(w),
                "r"(inp):);
            w = weights[i][4*j+2];
            asm volatile("neur_macc_u_8b %0, %1, %2\n":"=r"(temp):"r"(w),
                "r"(inp):);
            w = weights[i][4*j+3];
            asm volatile("neur_macc_u_8b %0, %1, %2\n":"=r"(temp):"r"(w),
```

```
                "r"(inp):);
        }
        /* Step 3: Requantize and apply activation function */
        asm volatile("neur_res_u %0, %1, %2\n":"=r"(output[i]):
            "r"(quantized_multiplier),"r"(out_shift_rl):);
    }
}
```

Comparing the original and optimized algorithms, it's evident that the latter demonstrates superior performance due to several critical factors:

1. Reduced number of iterations in the nested loops.
2. Decreased number of total executed instructions (e.g. in the original code operations like MAC require separate cycles for the execution of the multiplication and addition, while on the optimized code, this can be achieved with the use of just one instruction).
3. Parallel execution of every individual step on multiple input data.
4. Significant reduction in memory loads and stores.

5.3 Hardware Modifications

5.3.1 Core

The co-design of hardware and software is a pivotal element in our optimization strategy. By integrating new instructions into the ISA, we're able to generate specific signals that activate a new component, functioning as a NN accelerator (❸ in Figure 5.1). This synergy between hardware enhancements and software instructions is central to boosting the efficiency and performance of NN computations.

Rather than developing a separate coprocessor, this new component will be positioned on the second stage of the pipeline, where the decoding and execution of each instruction takes place (see Figure 5.7). This decision underpins the creation of a solution that is not only faster and more robust but also more power-efficient, when compared to the other option. The key advantage of this integration lies in the significant reduction of data transfer across the bus system that interconnects memory, the main processor, and what would have been a separate NN coprocessor. This approach minimizes latency and energy consumption associated with data movement, thereby enhancing the overall system performance and efficiency. We can also benefit from sharing the processor's resources, such as the cache memory and other crucial components (e.g. multipliers), in order to implement various computations in our accelerator.

Given that the system's decoder is initially not configured to recognize the newly introduced instructions, modifications to the decoder are imperative, alongside the integration

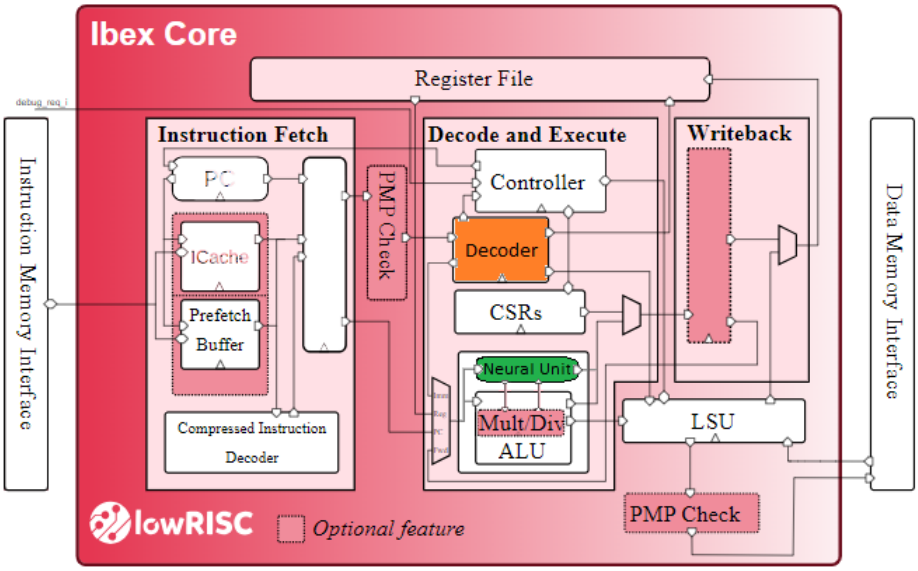


Figure 5.7: Schematic Diagram of the Modified Ibex Core. The components that have been modified or added are highlighted with orange (decoder) and green (hardware accelerator) respectively.

of the accelerator. By updating the decoder, we ensure it can accurately interpret the opcode and function code fields of these new instructions. This modification enables the decoder to correctly identify and differentiate the new instructions, extract the necessary values from the source registers, and subsequently feed forward this information to the newly added unit.

5.3.2 Hardware Accelerator

In the upcoming section, we will delve into the architecture of the newly integrated component, exploring its functionalities and the strategic design decisions undertaken to enhance its performance. A simplified block diagram that illustrates its internal structure can be seen in Figure 5.8.

Functionality

The core attribute of this accelerator is its capability to simultaneously compute four outputs, whether they are neurons in the context of a MLP or activation map features within CNNs. This parallel processing feature significantly enhances the computational efficiency, enabling the system to handle multiple data within the same operational cycle.

The intermediate results of the computations are held in additional 32-bit registers, serving as accumulators within the new component. The operational sequence begins with the initial instruction, which primarily sets up these registers by initializing them

with the shifted bias values. Upon decoding the subsequent series of instructions, the component is tasked with performing the multiplications between the input vectors and their corresponding weights, with the resulting products being aggregated in the 4 registers - accumulators. As the process advances to the concluding phase, triggered by the third and final set of instructions, the component undertakes the task of converting the activations to 8-bit values.

Architecture

Apart from the 32-bit registers, the accelerator is composed of several other key components, each playing a vital role in the processing pipeline (see Figure 5.8):

- **Weights - Inputs Decoder**: Equipped to handle two 32-bit operands, the decoder utilizes the 'funct7' field to effectively identify and segregate the operands that will be involved in multiplication. This component produces 8 pairs of 16-bit values that will be forwarded for processing by the rest of the accelerator's units.
- **Multiply-and-Accumulate (MAC) Unit**: Following the decoder, the MAC unit receives the decoded operands and conducts the necessary multiplications. In instances where it is necessary (cases (b) and (c) in Figure 5.5), some pairs of these products are then summed together before being accumulated back into the 32-bit registers.
- **Requantization Block:** this specialized block implements the requantization step. This component takes as input the values from the accumulators and its outputs are the ones that will be stored in the memory, in order to be advanced on the next layer of the network.

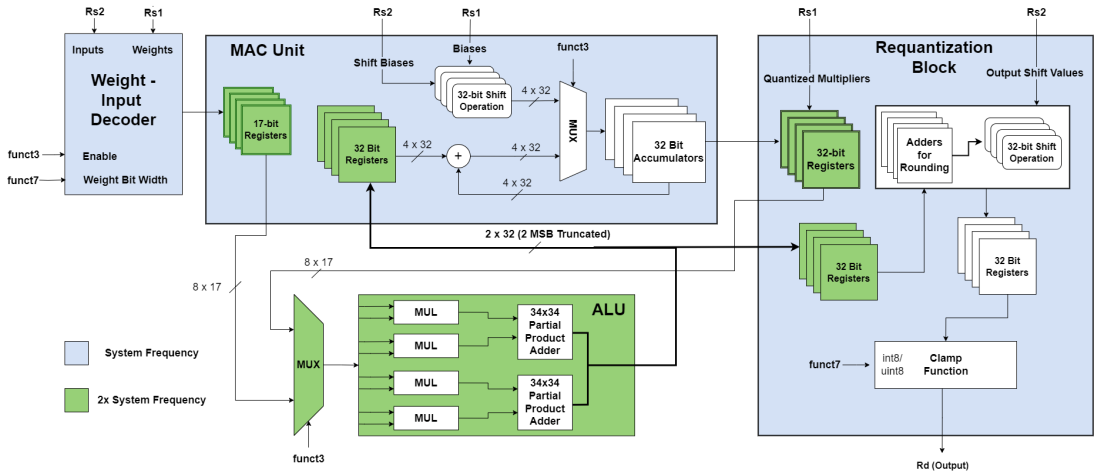


Figure 5.8: Proposed Neural Network Accelerator schematic diagram. The blocks with green color are operating with double the frequency, than the rest of the system.

5.3.3 Hardware Accelerator Optimizations

In this section, we describe the design choices made to elevate the performance of the accelerator, with a primary focus on optimizing efficiency. The overarching objective is to amplify the throughput of MAC operations per cycle, a critical metric for computational efficiency in NN processing, while simultaneously minimizing resource utilization and power consumption. These optimizations will enable us to activate the system's prefetcher and utilize the advanced optimization flags of the GCC compiler resulting in the generation of faster and more efficient code, without having to stall the processor's pipeline.

Exploiting Processor Resources

Where possible, the accelerator shares resources with the main processor. More specifically we opt to use the processor's multipliers in order to compute all the necessary products, during the MAC operations and the requantization procedure. For the implementation of the multiplication between two 32-bit integers in a single cycle, Ibex employs three 17x17 bit multipliers and then adds together the partial products to create the final result. To meet the specific demands of our accelerator and to further boost its computational capacity, we plan to extend this setup by incorporating an additional multiplier, mirroring the design of the existing three. This inclusion is intended to scale up the system's ability to handle parallel computations effectively.

Upon implementing both the original and the enhanced core designs on an FPGA device, we observe the utilization of 4 DSP blocks. This allocation stems from the operational requirements where, in the base multiplication scenario, 3 DSP blocks are dedicated to computing the partial products, and the fourth block is tasked with their aggregation. In contrast, with the modified core architecture, the "introduction" of the additional unit permits the use of these DSPs to perform four independent multiplications across each one of them.

Pipelining

Pipelining is a computer architecture technique used to increase the throughput or execution speed of a CPU by overlapping the execution of multiple instructions. It allows us to fetch and process new data in every single cycle and can particularly prove useful in scenarios devoid of data hazards, which occur when instructions rely on the outcomes of preceding ones. By dividing the processing tasks into several smaller stages and allowing each stage to handle a different part, pipelining ensures a more continuous and efficient flow of instruction execution.

In the pipelined architecture of our design, the outcomes of each stage within the processing blocks are temporarily held in registers before being transitioned to the subsequent

stage. Even though the total amount of cycles needed in order to execute a single computation increases, we benefit from the higher throughput of the pipelined design as well as the decreased critical path of the particular block (this allows us to potentially increase the frequency of the clock used for these operations). In our design, we implement pipelining within the internal architecture of two distinct blocks: the component dedicated to executing the MAC operations, which operates in conjunction with a preceding decoder, and the unit responsible for the requantization step.

In the first case, we have divided the execution of the MAC instruction into three different stages. In the initial cycle, upon receiving the two operands (weights and inputs), the system decodes their values according to the *funct7* value of the instruction and stores them before using them for our computations. The second stage is dedicated to performing the actual multiplication operations between the decoded operands and the extra additions between the multiplication products when needed, while in the final stage, the results computed in the previous step are added to the accumulators. In the second occasion, the requantization procedure unfolds over three distinct cycles: the first cycle is dedicated to the multiplication process. The second cycle encompasses a rounding operation (which is performed by an addition with a specified value) to refine the results followed by a right shift to scale down the values, and the third cycle involves the clamping of the outputs to ensure they fall within the designated range, depending on whether the output is asked to be in *int8* or *uint8* format.

Since we only have access to 17x17 multipliers, it is not feasible to compute the result between a 32-bit value and an 8-bit number directly. For this exact reason we are going to divide the 32-bit number into two 16-bit ones and calculate the partial products between all the formed pairs. This divide-and-conquer approach can be seen in Figure 5.9.

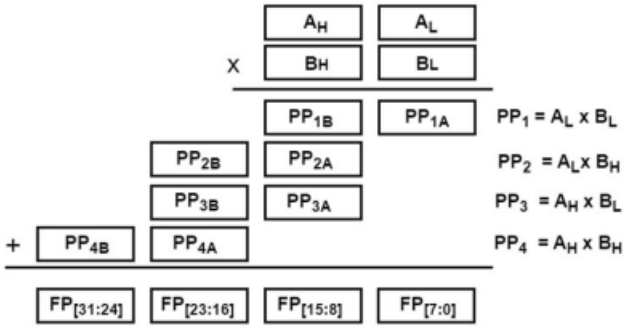


Figure 5.9: Divide-and-conquer multiplication between two 16-bit numbers. The same methodology can be applied in our case too.
Source: [49]

We can further simplify this strategy in our implementation, because the second operant that corresponds to the 8-bit quantized multiplier (from equation 3.17) is always a positive number and that means that its 16-higher bits will always be grounded to zero. This allows

us to perform this calculation by just computing two out of the four partial products and then adding them together after properly aligning.

Frequency Scaling

So far we have mentioned that the MAC instructions will be executed, ideally, in just one cycle. However, as we can see in Table 5.2 the different types of instructions that are involved in these operations, execute a varying number of multiplications and additions. Given that our architecture incorporates only four multipliers and considering that certain instructions demand the execution of 8 or even 16 MAC sequences, a straightforward execution approach would inherently lead to pipeline stalls. Specifically in a pipelined architecture, to accommodate instructions requiring 8 MAC sequences, we would encounter a stall of 1 cycle, and for those necessitating 16 sequences, a stall of 3 cycles would be unavoidable without additional optimizations to the processor.

Adopting a heterogeneous clocking strategy, where key computational components operate at a higher frequency than the rest of the system, presents an elegant solution to the challenge [31]. This approach directly targets the optimization of those units that bear the majority of the computational workload, specifically the multipliers and adders involved in MAC operations and the scaling down process. By increasing the clock frequency of these critical components, we are able to execute more operations per unit of time, effectively increasing their throughput. However, by using a higher frequency clock, new difficulties arise. Notably, various synchronization issues may occur, such as data corruption or loss during transfers between components operating at different frequencies. Furthermore, as clock speed escalates, resources that are utilized will also increase due to the additional circuitry needed in order to maintain signal integrity over longer distances. Another critical aspect to consider is the surge in power consumption, as dynamic power consumption is proportional to the operating frequency.

In order to simplify synchronization with the processor's main clock, we decided to upscale the frequency of the new clock by a power-of-two. Since the Ibex core operates at a baseline frequency of 50 MHz, we chose to configure the second clock with **double** the frequency at 100 MHz. By adopting this approach we can achieve increased computational speed, since the single cycle execution of the instructions that compute 8 MAC operations is possible (1 stall will still be required for the last case), while also balancing power consumption, resource utilization and design complexity.

Soft SIMD

The last optimization focuses on implementing the instructions that execute 16 MAC operations in a single cycle. Inspired by [32] we are planning on achieving this goal by packing two multiplications between the 2-bit weights and the 8-bit inputs into a single multiplier.

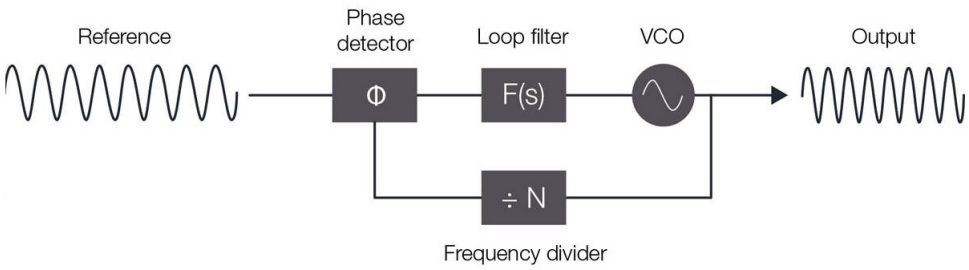


Figure 5.10: The Phase Locked Loop (PLL) control system is used to generate the 2 different clock signals from a single oscillator. It can also guarantee synchronization of the initial rising edges of the clocks.

The outcome of multiplying a 2-bit number by a 9-bit number - accounting for both unsigned and signed inputs, which necessitates an additional bit for the sign - requires a minimum width of 11 bits for precise representation. In such multiplications, the top 22 bits are uniformly filled with either 1s or 0s, effectively conveying only a single bit of information. This characteristic allows for the utilization of these upper bits to perform an additional computation concurrently, without impacting the results of the lower-order inputs. To effectively leverage the unused upper bits for an additional computation, two essential guidelines must be adhered to:

1. The computation involving the upper bits must not influence the outcome of the lower bits.

2. It should be possible to identify and correct any interference of the upper bits caused by the computation involving the lower bits.

Specifically in our case, to satisfy the first rule, the least significant bit of the upper product results must not fall into the lower 11-bits. Thus, the upper bits that correspond to the second weight must commence from at least the 12th bit onwards. For safety reasons, we are going to add a guard bit between the results of the 2 multiplications and start from the 13th bit. The equation that expresses the result is the following:

$$(W_2 \cdot 2^{13} + W_1) \cdot I_1 = W_2 \cdot I_1 \cdot 2^{13} + W_1 \cdot I_1$$

To obtain the outcomes of the two simultaneous multiplications, we can extract the bits located in positions 23 to 13 for the upper multiplication result, and positions 10 to 0 for the lower multiplication result.

Regarding the second rule, a thorough examination of all possible outcomes from the combinations of numbers used in our method reveals that errors in the results of the multiplication involving the higher bits occur exclusively when the weight associated with the lower bits is negative. Notably, in every instance of such discrepancies, the actual result differs from the expected outcome by only 1. Given the predictable nature of

these scenarios, the simplest solution to this minor issue is to adjust the result of this multiplication by adding a value of 1. This adjustment effectively corrects the error, ensuring the accuracy of the computation.

5.3.4 Implementation on FPGA

The final step involves gathering comprehensive details regarding the required resources, the anticipated power consumption of our system and the latency of the critical paths in our design. The latter is crucial because we must make sure that we meet all the timing constraints in our design, especially because we are using a dual clock configuration. The mapping on the FPGA device is done with the assistance of the Xilinx Vivado tool. For our implementation we use the XC7VX485T part of the Xilinx Virtex-7 FPGA family, which has sufficient memory for the evaluation of all of the benchmarks that we will examine in the following chapter.

Chapter 6

Experimental Results

In this chapter, we will showcase the results of our implementation, with a particular emphasis on key performance metrics, including:

- the accuracy of the quantized models,
- the latency experienced during the execution of our models on the Ibex RISC-V core, operating at 50 MHz and
- the resource utilization and power consumption observed while deploying the models on the FPGA device

All models featured in this chapter underwent compilation with the -O3 optimization flag activated. This option ensures that more aggressive optimization steps will be taken by the compiler. As a result, compilation time becomes slower, but in return we can achieve the best results in terms of latency.

6.1 Comparison with Baseline RV32IMC ISA

To assess the performance of our design, we will initially compare the execution results of a Dense and a Convolutional layer against their respective implementations based on the RV32IMC ISA. The complexity and computational intensity of these common computational patterns make them ideal for revealing the advantages of our optimized hardware architecture over standard RV32IMC ISA implementations, especially in the context of efficiency and speed. More specifically, we are going to benchmark:

1. a fully connected layer configured with 512 input nodes and 256 output nodes and
2. a convolutional kernel processing a $16 \times 16 \times 32$ input tensor (utilizing a Height-Width-Channel data layout) with a filter dimension of $64 \times 3 \times 3 \times 32$ (Channels $\times$ Kernel Width $\times$ Kernel Height $\times$ Multiplier).

In each benchmark, we will explore every potential configuration for the bit widths employed in representing the weights. Additionally, we are going to depict how each accelerator optimization technique affects the final performance in every instance. The results of the comparison in terms of latency measured in clock cycles, are presented in Figures 6.1 and 6.2 for the Fully Connected Layer and the CNN kernel respectively. Our final implementation can improve inference on the RISC-V core by **7.9 - 17.6x** on a single Dense layer and **12.0 - 33.1x** on the much more computationally demanding convolutional layer.

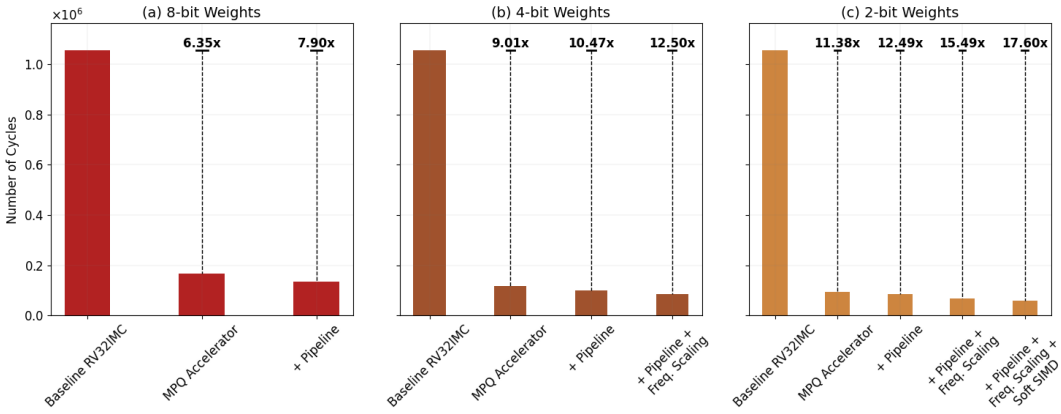


Figure 6.1: Latency of Dense Layer implementation on Ibex Core for Configurations: (a) 8-bit weights, (b) 4-bit weights, and (c) 2-bit weights.

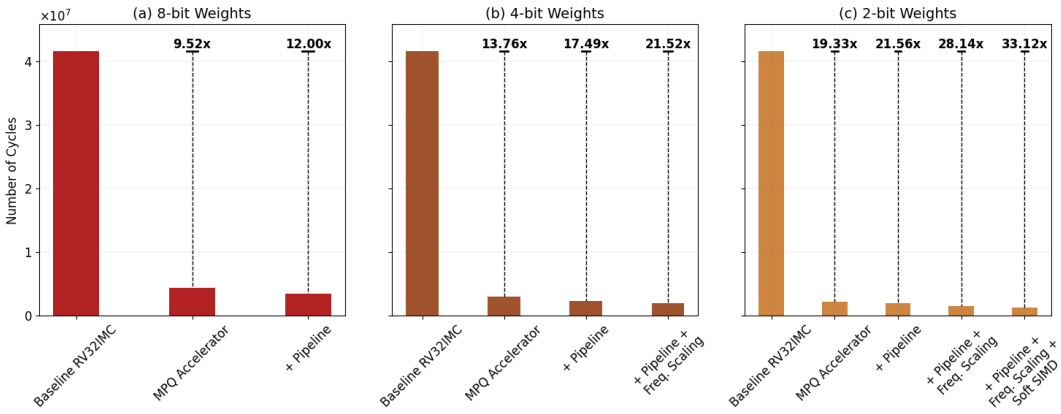


Figure 6.2: Latency of Convolutional Layer implementation on Ibex Core for Configurations: (a) 8-bit weights, (b) 4-bit weights, and (c) 2-bit weights.

Our approach heavily favors the more extreme quantization scenarios, evidenced by both benchmarks showing that the most significant speedup improvements are achieved when employing 2-bit parameters. This behavior is attributed primarily to two factors. Firstly, the use of 2-bit parameters allows for a denser packing of weights within the instructions, which significantly reduces the amount memory accesses and the number of MAC instructions required. The second reason is that our accelerator is meticulously

designed to process each MAC instruction within a single cycle, thereby amplifying the efficiency gains for configurations utilizing 2-bit weights compared to those with higher bit widths.

6.2 Comparison with State-of-the-Art

To fully illustrate the efficacy of our methodology, we plan to present a detailed overview of the modified Ibex core's performance across a diverse array of NN models. The selected models for evaluation are prominent in cutting-edge research, as discussed in Chapter 2, and were chosen based on their complexity and exceptional performance in various applications. For every case-study we are going to:

1. Describe their architecture, the dataset they were trained at and their full precision performance.

2. Identify the pareto optimal solutions, by assessing the trade-offs between the quantized model's accuracy and the total number of MAC instructions executed.

3. Select from the pareto frontier the best solution that demonstrates accuracy degradation of less than **1%, 2% and 5%**, measure the execution time of the optimized algorithm (with the new instructions included) and compare it with the initial model.

It is important to note that the reported times from the state-of-the-art works [5], [8] and [11], are normalized based on the frequency of the Ibex core.

6.2.1 FANN-on-MCU

The first model to be evaluated is a MLP with a relatively simple structure, which is illustrated in Figure 6.3 (with the use of the Netron App). The architecture has been replicated from [11] and it has 76 inputs, 10 outputs and 3 hidden layers with 300, 200 and 100 nodes. However, the dataset it has been trained at is not public, we will use one from the UCI database [50] that has a similar number of input features and expected outputs/classes. The baseline model with FP parameters reaches an accuracy of **98.14%**.

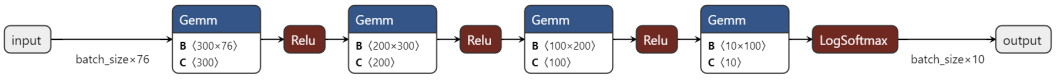


Figure 6.3: Architecture of MLP under examination.

Given that the model is composed of four consecutive Fully Connected Layers, we are presented with a total of 81 potential configurations to evaluate. Our task is to meticulously examine each configuration to identify which ones deliver optimal results that align with our specific requirements. Since measuring the latency for each individual configuration on the Ibex core is an impractical and time-intensive task, we will utilize

the count of MAC instructions executed as a proxy to estimate which solutions are likely to exhibit the lowest latency.

Figure 6.4 illustrates the Pareto Space, mapping the trade-off between accuracy (depicted on the y-axis) and the quantity of MAC operations performed (shown on the x-axis) for this specific model. On the same graph, apart from the solutions that are located on the Pareto frontier (with the green square symbol), we will also include the full precision model (represented by a star symbol) to visually demonstrate the accuracy degradation and the reduction in the number of MAC operations achieved through our optimization approach.

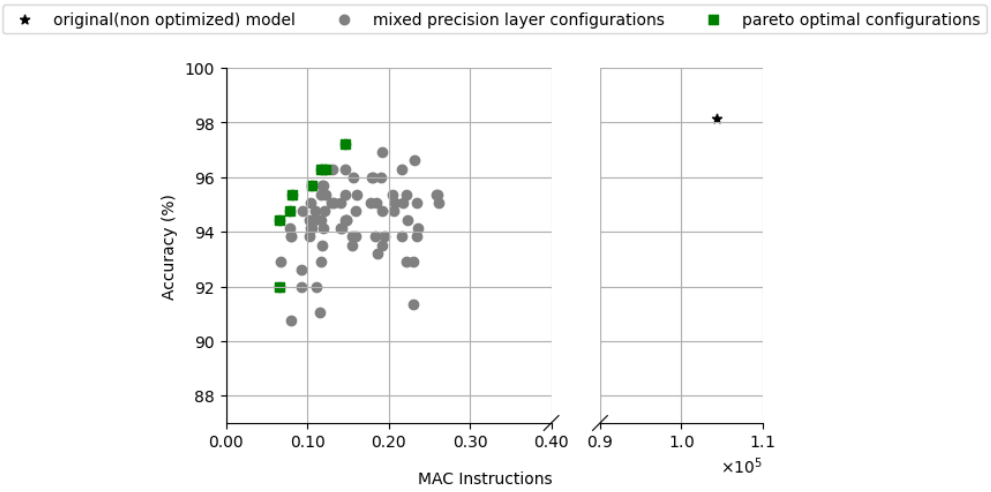


Figure 6.4: Pareto Space of the MLP under examination.

Table 6.1 provides a comprehensive summary of each selected configuration that was executed on the Ibex core, as well as a comparative analysis the speedup metrics in relation to the original RV32IMC implementation and the findings reported in the "FANN-on-MCU" paper. More specifically, with the use of the original ISA, the algorithm is executed in **23.13 milliseconds**. In contrast, [11] demonstrates that the same model, when run on a single Ibex core, completes in **11.4 milliseconds** and when deployed on a multicore system, the execution time significantly reduces to just **1.6 milliseconds**.

Accuracy Degradation	Weight Configuration	Accuracy	Speedup w.r.t RV32IMC	Speedup w.r.t [11]
<1% acc. drop	(4, 4, 4, 2)	97.22 %	13.0x	0.89x
<2% acc. drop	(2, 8, 4, 8)	96.29 %	16.3x	**1.12x**
<5% acc. drop	(2, 2, 2, 4)	94.44 %	22.9x	**1.59x**

Table 6.1: Performance of the selected configurations for the MLP model under examination. The accuracy of the baseline model with FP weights and activations stands at 98.14%. We have highlighted the solutions that achieve better results than the state-of-the-art.

6.2.2 CMSIS-NN

The second model under our examination originates from reference [5]. The CMSIS-NN library, renowned for its widespread use in deploying neural networks on ARM processors, serves as a benchmark in numerous studies, providing a foundational comparison for their outcomes. The architecture of this model is depicted in Figure 6.5, showcasing its structural details and design considerations. It is a more intricate and computationally demanding network compared to the previous one we evaluated. It comprises three consecutive blocks of Convolutional layers, each succeeded by a Max Pooling layer, culminating in a Fully Connected layer that functions as a classifier. We will assess this model's performance using the widely recognized Cifar10 dataset, offering a comprehensive evaluation of its capabilities in handling complex image recognition tasks. Since this network also consists of 4 layers in total, we are going to thoroughly investigate every possible combination of bit-width configurations for each layer.

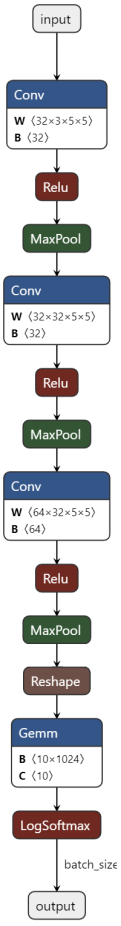


Figure 6.5: Architecture of the CNN under examination.

The outcomes of our detailed analysis are presented in Figure 6.6 and Table 6.2. Figure 6.6 graphically depicts the Pareto space that is produced after performing QAT to meticulously fine-tune the model's parameters, while Table 6.2 lists all the necessary information

about the speedup we achieve when deploying the selected configurations on the Ibex core. For a single inference of the NN, the baseline RV32IMC needs up to **3480 milliseconds**, while the CMSIS-NN library can execute the same algorithm in just **428 milliseconds**, when operating at 50 MHz with an accuracy of **78.89%**.

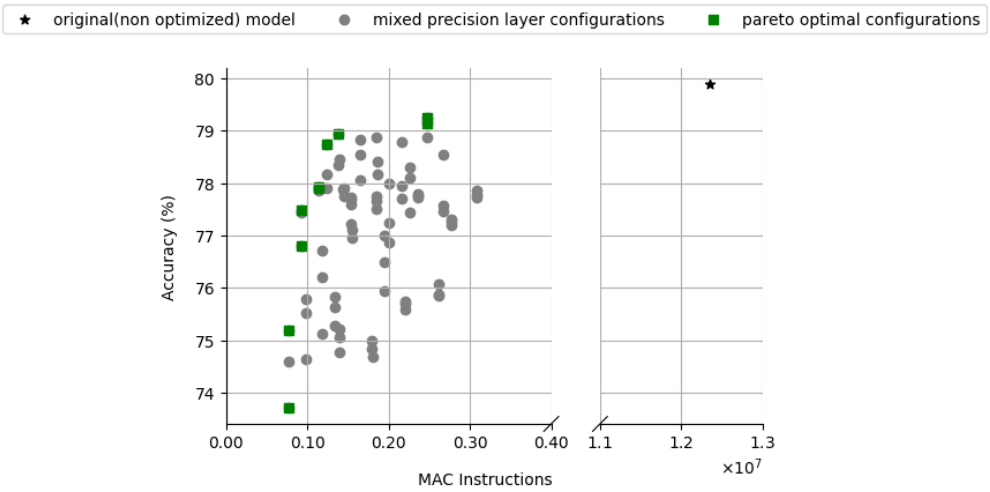


Figure 6.6: Pareto Space of the CNN under examination.

Accuracy Degradation	Weight Configuration	Accuracy	Speedup w.r.t RV32IMC	Speedup w.r.t [5]
<1% acc. drop	(4, 8, 2, 2)	77.95 %	17.8x	**2.19x**
<2% acc. drop	(4, 2, 4, 2)	76.90 %	24.0x	**2.95x**
<5% acc. drop	(2, 2, 2, 4)	74.19 %	28.3x	**3.47x**

Table 6.2: Performance of the selected configurations for the CNN model under examination. The accuracy of the baseline model with FP weights and activations stands at 78.89%. We have highlighted the solutions that achieve better results than the state-of-the-art.

6.2.3 MCUNet

The last model we are going to analyze comes from the MCUNet paper [8]. Specifically, we will deploy the mcunet-vww1 model, which presents a higher level of complexity compared to the two previously analyzed models due to its integration of various layer types. This network is a more compact version of MobileNetV2 [51] and is composed of:

- An initial conventional convolutional layer that processes the input data,
- Fifteen composite blocks of layers, each consisting of depthwise and pointwise convolutions. Some of these blocks also feature residual connections that facilitate the flow of information by linking the inputs directly to the outputs of these blocks (Figure 6.7).

- An average pooling layer followed by a fully connected (Dense) layer, to produce the final output predictions of the network.

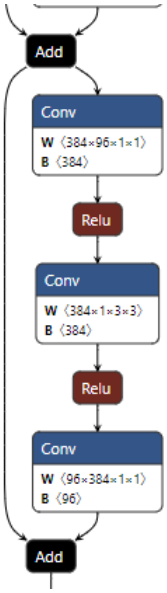


Figure 6.7: Main building block of the mcunet-vww1 model.

This model is trained and evaluated on the Visual Wake Words (VWW) Dataset [52], a collection of images designed to facilitate the development of models capable of detecting a person within the device's field of view. Owing to the extensive number of layers that constitute this NN, an exhaustive examination of every possible weight configuration is impractical.

To address this challenge, we have opted to aggregate sets of three consecutive blocks, applying a consistent bit-width across them for quantization purposes. Additionally, just for the last layer of the network we are going to manually set its bit-width to 8-bit, as it does not contribute a lot to the network's total workload. This leads us to search through 736 possible configurations. Although this approach may lead to a suboptimal solution, it allows us to expedite the analysis process considerably. Figure 6.8 displays the Pareto space, highlighting the trade-off between accuracy and the estimated latency, based on the number of MAC instructions for the mcunet-vww1 model.

The baseline model accuracy on the Visual Wake Words (VWW) dataset is **88.9%**. In terms of performance, executing a single iteration of the feed-forward algorithm using the MCUNet framework takes up to **552 milliseconds**. In contrast, employing the original RV32IMC ISA for the same task requires approximately **3697 milliseconds**. The MCUNet framework offers a markedly more efficient solution in terms of both execution time and memory usage, especially when compared to the performance of the unoptimized RISC-V core, as well as the CMSIS-NN library (**1162 milliseconds**) and the X-CUBE-AI

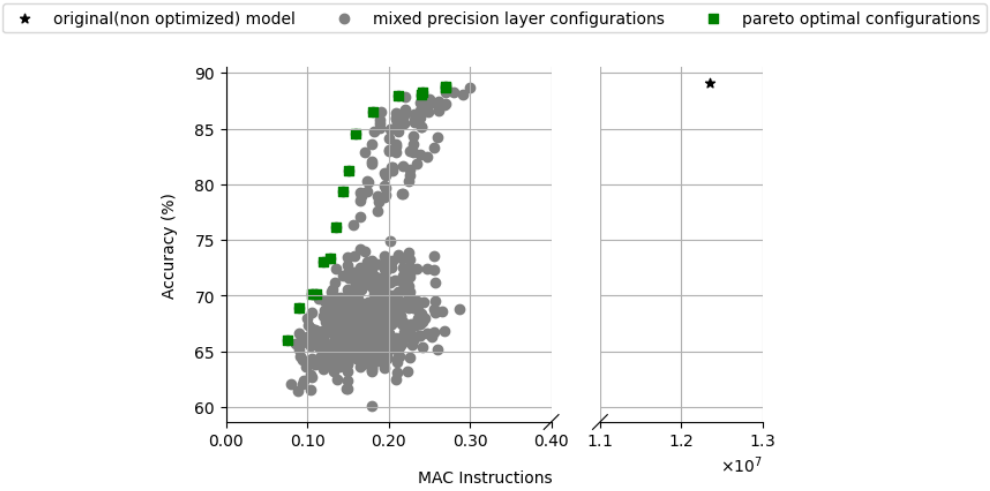


Figure 6.8: Pareto Space for the mcunet-vww1 model.

package (**591 milliseconds**). Table 6.3 compiles all essential metrics for the optimal configurations found on the Pareto frontier. These configurations meet the established criteria for minimal accuracy loss post-quantization and they are presented alongside the results from the RV32IMC ISA and MCUNet to facilitate a thorough comparative analysis.

Accuracy Degradation	Weight Configuration	Accuracy	Speedup w.r.t RV32IMC	Speedup w.r.t [5]
<1% acc. drop	(8, 8, 8, 4, 8, 4)	88.15 %	5.97x	0.89x
<2% acc. drop	(8, 4, 4, 4, 8, 8)	87.02 %	6.74x	**1.007x**
<5% acc. drop	(8, 4, 4, 4, 4, 4)	84.50 %	7.04x	**1.05x**

Table 6.3: Performance of each selected configuration for the mcunet-vww1 model. The accuracy of the baseline model with FP weights and activations stands at 88.9%. We have highlighted the solutions that achieve better results than the state-of-the-art.

It is crucial to acknowledge that employing 2-bit quantization for this particular challenge, fails to yield satisfactory outcomes, as it leads to significant accuracy loss in this complex model, absent extensive fine-tuning of its parameters. Given the constraints in available resources and time, conducting QAT for extended durations across every configuration is impractical in this context, though such an approach might have potentially enhanced performance.

It's worth mentioning that while our method (slightly) surpasses the performance of the state-of-the-art solution, along with the CMSIS-NN and X-CUBE-AI implementations, achieving only minimal accuracy loss, the enhancement over the RV32IMC ISA does not match the impressive gains observed in the two previously evaluated models. This discrepancy primarily stems from the fact that the speedup realized during the execution of Depthwise convolutions is approximately **4.5-5 times** across all weight resolutions. Such

layers do not permit the same degree of input reuse as seen in standard convolutional layers (utilized for the pointwise convolution operations), where we noted a latency improvement exceeding 13 times for this specific model. The inherent architectural differences in Depthwise convolutions limit the extent to which performance can be optimized compared to standard convolutions.

Table 6.4 contains an overview of the results that we acquired from all the models that we have analyzed thus far. Our focus predominantly lies on the latency observed during the execution of each model on the Ibex core, both with the incorporation of new instructions and without them. Additionally, we include the normalized latency figures as measured in the state-of-the-art solutions, providing a benchmark for comparison and demonstrating the effectiveness of our approach in enhancing computational performance.

Model	**RV32IMC**	**State of the art**	**QNN with accuracy degradation**		
			<1%	**<2%**	**<5%**
MLP from FANN-on-MCU	23.13 msec	1.60 msec	1.77 msec	**1.41 msec**	**1.01 msec**
CNN from CMSIS-NN	3480 msec	428 msec	**195 msec**	**145 msec**	**123 msec**
mcunet-vww-1	3697 msec	552 msec	619 msec	**548 msec**	**525 msec**

Table 6.4: Latency reported in milliseconds (msec) for every model we examined. The numbers that correspond to the state-of-the-art solutions are gathered from their respective paper, while the rest are measured on the Ibex RISC-V core.

6.3 Resource Utilization & Energy Consumption

In this last section, we will detail the overhead introduced to the initial RISC-V processor, a critical aspect of our consideration throughout this entire project, particularly in the development of the hardware accelerator. Table 6.5 compiles data from the implementation phase, reflecting the processor's mapping onto the FPGA device. Our methodology has successfully enhanced the performance of the Ibex core while keeping the increase in resource usage and power consumption to a minimum. More specifically, we report:

- A 34.89% rise in the usage of Lookup Tables (LUTs),
- A 24.28% increase in the employment of registers, specifically Flip-Flops (FFs),
- The retention of the original count of DSP blocks, with no additional units required,
- A modest 5 mW surge in the design's power consumption,which mainly stems from the incorporation of a secondary clock operating at a higher frequency.

These metrics underscore our commitment to optimizing the Ibex core's performance in a resource-efficient manner, balancing the dual objectives of enhancing computational speed and maintaining low power usage.

Ibex Processor	Initial	Proposed
LUTs	5479	7391
FFs	5122	6366
DSPs	4	4
Power Consumption (Watt)	0.256	0.261
Clock(s) Frequency	50 MHz	50 MHz/100 MHz (Dual Clock Conf.)

Table 6.5: Comparison between the original and the modified Ibex core regarding the utilized resources on a Virtex-7 FPGA board, their power consumption and the speed of the clocks used for the entirety of the system

Our most energy-efficient solutions, despite experiencing the highest accuracy loss, achieve the most significant reductions in processing demands. This efficiency is due to the speed enhancements realized, notwithstanding a slight increase in the power consumption of the proposed architecture. Specifically, these optimizations result in a **22.46x** reduction in processing demands for the first model, a **27.75x** decrease for the second case study, and a **6.91x** reduction for the mcunet-vww1 model. Figure 6.9 showcases the considerable efficiency gains per inference for each model, demonstrating the substantial improvements our approach offers, even when trading off a degree of accuracy.

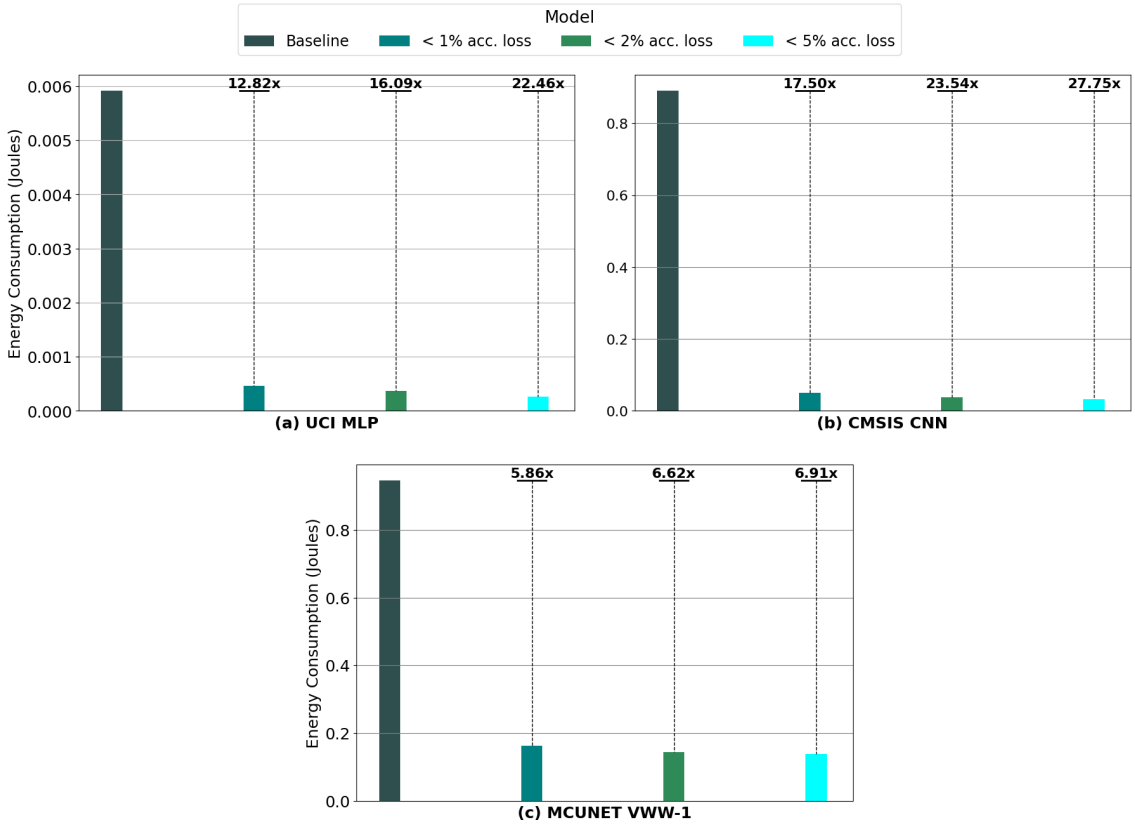


Figure 6.9: Energy Consumption of each configuration we selected for the analysis of: (a) MLP from "FANN-on-MCU", (b) CNN from "CMSIS-NN", and (c) mcunet-vww1 from "MCUNet".

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this work, we have presented an end-to-end workflow designed for the deployment and optimization of TinyML models on a RISC-V core. We demonstrated that through the application of a MPQ scheme and a methodical exploration of the Design Space, it is possible to identify solutions that adeptly balance the trade-off between latency and accuracy loss. To enhance the execution of those Quantized Neural Networks on the chosen processor, we extended the RV32IMC ISA with specialized instructions and incorporated a functional unit into the processor's pipeline to support them. This approach manages to compete and in many cases outperform state-of-the-art works targeting RISC-V architectures, as well as popular frameworks employed by IoT devices.

7.2 Future Work

Some potential optimizations that could elevate the efficiency of our work and expand its applicability include:

- Refining Design Space Exploration by developing a faster and more robust strategy, based on statistical metrics to estimate the sensitivity of each layer and efficiently identify the optimal configuration for the bit-width resolution of the weights.
- Incorporating post-load increment instructions that automatically increment the address after a load operation to streamline data handling within loops and repetitive operations.
- Mapping the processor onto an Application-Specific Integrated Circuit (ASIC) design for enhanced performance in terms of speed and power efficiency.
- Deploying multiple instances of the proposed architecture on a single FPGA board presents an opportunity to increase parallel processing, which can significantly speed up model inference by distributing computational tasks across multiple cores.

- Applying the workflow to a broader range of RISC-V cores, which in turn will help us assess the generalizability and performance scalability of the approach.

Bibliography

[1] Zhi Zhou, Xu Chen, En Li, Liekang Zeng, Ke Luo, and Junshan Zhang. Edge intelligence: Paving the last mile of artificial intelligence with edge computing. *Proceedings of the IEEE*, 107(8):1738–1762, 2019.

[2] Maurizio Capra, Beatrice Bussolino, Alberto Marchisio, Guido Masera, Maurizio Martina, and Muhammad Shafique. Hardware and software optimizations for accelerating deep neural networks: Survey of current trends, challenges, and the road ahead. *IEEE Access*, 8:225134–225180, 2020.

[3] Rakhee Kallimani, Krishna Pai, Prasoon Raghuwanshi, Sridhar Iyer, and Onel L. A. López. Tinyml: Tools, applications, challenges, and future research directions. *Multimedia Tools and Applications*, September 2023.

[4] Norah N. Alajlan and Dina M. Ibrahim. Tinyml: Enabling of inference deep learning models on ultra-low-power iot edge devices for ai applications. *Micromachines*, 13(6), 2022.

[5] Liangzhen Lai, Naveen Suda, and Vikas Chandra. Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus, 2018.

[6] Robert David, Jared Duke, Advait Jain, Vijay Janapa Reddi, Nat Jeffries, Jian Li, Nick Kreeger, Ian Nappier, Meghna Natraj, Shlomi Regev, Rocky Rhodes, Tiezhen Wang, and Pete Warden. Tensorflow lite micro: Embedded machine learning on tinyml systems, 2021.

[7] STMicroelectronics. X-CUBE-AI - Artificial Intelligence Expansion Package. `https://www.st.com/en/embedded-software/x-cube-ai.html`, 2023.

[8] Ji Lin, Wei-Ming Chen, Yujun Lin, John Cohn, Chuang Gan, and Song Han. Mcunet: Tiny deep learning on iot devices, 2020.

[9] Jiaxiang Wu, Cong Leng, Yuhang Wang, Qinghao Hu, and Jian Cheng. Quantized convolutional neural networks for mobile devices, 2016.

[10] Ruizhou Ding, Zeye Liu, R. D. Shawn Blanton, and Diana Marculescu. Quantized deep neural networks for energy efficient hardware-based inference. In *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 1–8, 2018.

[11] Xiaying Wang, Michele Magno, Lukas Cavigelli, and Luca Benini. Fann-on-mcu: An open-source toolkit for energy-efficient neural network inference at the edge of the internet of things, 2022.

[12] Yvan Tortorella, Luca Bertaccini, Luca Benini, Davide Rossi, and Francesco Conti. Redmule: A mixed-precision matrix-matrix operation engine for flexible and energy-efficient on-chip linear algebra and tinyml training acceleration, 2023.

[13] Alessio Burrello, Angelo Garofalo, Nazareno Bruschi, Giuseppe Tagliavini, Davide Rossi, and Francesco Conti. Dory: Automatic end-to-end deployment of real-world dnns on low-cost iot mcus. *IEEE Transactions on Computers*, 70(8):1253–1268, August 2021.

[14] Angelo Garofalo, Manuele Rusci, Francesco Conti, Davide Rossi, and Luca Benini. Pulp-nn: accelerating quantized neural networks on parallel ultra-low-power risc-v processors. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 378(2164):20190155, December 2019.

[15] Angelo Garofalo, Giuseppe Tagliavini, Francesco Conti, Luca Benini, and Davide Rossi. Xpulpnn: Enabling energy efficient and flexible inference of quantized neural network on risc-v based iot end nodes, 2020.

[16] Philip Colangelo, Nasibeh Nasiri, Eriko Nurvitadhi, Asit K. Mishra, Martin Margala, and Kevin Nealis. Exploration of low numeric precision deep learning inference using intel® fpgas. *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 73–80, 2018.

[17] Hao Zhang, Dongdong Chen, and Seok-Bum Ko. New flexible multiple-precision multiply-accumulate unit for deep neural network training and inference. *IEEE Transactions on Computers*, 69(1):26–38, 2020.

[18] Mohammadhossein Askarihemmat, Sean Wagner, Olexa Bilaniuk, Yassine Hariri, Yvon Savaria, and Jean-Pierre David. Barvinn: Arbitrary precision dnn accelerator controlled by a risc-v cpu. In *Proceedings of the 28th Asia and South Pacific Design Automation Conference*, ASPDAC '23. ACM, January 2023.

[19] Angelo Garofalo, Gianmarco Ottavi, Alfio di Mauro, Francesco Conti, Giuseppe Tagliavini, Luca Benini, and Davide Rossi. A 1.15 tops/w, 16-cores parallel ultra-low power cluster with 2b-to-32b fully flexible bit-precision and vector lockstep execution mode. In *ESSCIRC 2021 - IEEE 47th European Solid State Circuits Conference (ESSCIRC)*, pages 267–270, 2021.

[20] L. Alzubaidi, J. Zhang, A.J. Humaidi, A. Al-Dujaili, Y. Duan, O. Al-Shamma, J. Santamaría, M.A. Fadhel, M. Al-Amidie, and L. Farhan. Review of deep learning: concepts, cnn architectures, challenges, applications, future directions. *Journal of Big Data*, 8(1):53, 2021.

[21] Junxi Feng, Xiaohai He, Qizhi Teng, Chao Ren, Honggang Chen, and Yang Li. Reconstruction of porous media from extremely limited information using conditional generative adversarial networks. *Phys. Rev. E*, 100:033308, Sep 2019.

[22] Anke Meyer-Bäse. X - specialized neural networks relevant to bioimaging. In Anke Meyer-Bäse, editor, *Pattern Recognition in Medical Imaging*, pages 318–345. Academic Press, San Diego, 2004.

[23] Hao Wu, Patrick Judd, Xiaojie Zhang, Mikhail Isaev, and Paulius Micikevicius. Integer quantization for deep learning inference: Principles and empirical evaluation, 2020.

[24] Nilesh Prasad Pandey, Markus Nagel, Mart van Baalen, Yin Huang, Chirag Patel, and Tijmen Blankevoort. A practical mixed precision algorithm for post-training quantization, 2023.

[25] Giorgos Armeniakos, Georgios Zervakis, Dimitrios Soudris, and Jörg Henkel. Hardware approximate techniques for deep neural network accelerators: A survey. *ACM Computing Surveys*, 55(4):1–36, November 2022.

[26] Jiawei Liu, Lin Niu, Zhihang Yuan, Dawei Yang, Xinggang Wang, and Wenyu Liu. Pd-quant: Post-training quantization based on prediction difference metric, 2023.

[27] RISC-V Foundation. RISC-V Foundation — Instruction Set Architecture (ISA). `https://riscv.org/`, 2019. Accessed: 2023-02-10.

[28] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović. The risc-v instruction set manual, volume 1: User-level isa, version 2.0. Technical report, RISC-V Foundation, May 2014. Version 2.0.

[29] Enfang Cui, Tianzheng Li, and Qian Wei. Risc-v instruction set architecture extensions: A survey. *IEEE Access*, 11:24696–24711, 2023.

[30] lowRISC. Ibex: A small and efficient RISC-V core. `https://github.com/lowRISC/ibex`, 2023.

[31] Vasileios Leon, Muhammad Abdullah Hanif, Giorgos Armeniakos, Xun Jiao, Muhammad Shafique, Kiamal Pekmestzi, and Dimitrios Soudris. Approximate computing survey, part i: Terminology and software & hardware approximation techniques, 2023.

[32] Convolutional neural network with int4 optimization on xilinx devices white paper. Online, 2014. `https://api.semanticscholar.org/CorpusID:225061851`.

[33] Marco Cococcioni, Federico Rossi, Emanuele Ruffaldi, and Sergio Saponara. A lightweight posit processing unit for risc-v processors in deep neural network applications. *IEEE Transactions on Emerging Topics in Computing*, 10(4):1898–1908, 2022.

[34] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019.

[35] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning, 2016.

[36] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding, 2014.

[37] I. H. Sarker. Machine learning: Algorithms, real-world applications and research directions. *SN Computer Science*, 2(3):160, 2021.

[38] Evelyn Herberg. Lecture notes: Neural network architectures, 2023.

[39] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[40] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications, 2017.

[41] Robin M. Schmidt. Recurrent neural networks (rnns): A gentle introduction and overview, 2019.

[42] Markus Nagel, Marios Fournarakis, Rana Ali Amjad, Yelysei Bondarenko, Mart van Baalen, and Tijmen Blankevoort. A white paper on neural network quantization, 2021.

[43] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference, 2017.

[44] Alessandro Pappalardo. Xilinx/brevitas. `https://github.com/Xilinx/brevitas`, 2023.

[45] Riscv-Software-Src. RISCV-software-src/RISCV-isa-SIM: Spike, a RISC-V ISA simulator. `https://github.com/riscv-software-src/riscv-isa-sim`, 2023.

[46] Olofk. fusesoc: Package manager and build abstraction tool for fpga/asic development. `https://github.com/olofk/fusesoc`, 2023.

[47] Verilator. Verilator/Verilator: Verilator open-source SystemVerilog Simulator and Lint System. https://github.com/verilator/verilator, 2023.

[48] Adding custom instructions to the risc-v gnu-gcc toolchain. https://hsandid.github.io/posts/risc-v-custom-instruction/.

[49] M. Abrar, H. Elahi, B.A. Ahmad, et al. An area-optimized n-bit multiplication technique using n/2-bit multiplication algorithm. *SN Applied Sciences*, 1:1348, 2019.

[50] Clara Higuera, Katheleen Gardiner, and Krzysztof Cios. Mice protein expression. https://archive.ics.uci.edu/ml/datasets/Mice+Protein+Expression, 2015. DOI: 10.24432/C50S3Z.

[51] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks, 2019.

[52] Aakanksha Chowdhery, Pete Warden, Jonathon Shlens, Andrew Howard, and Rocky Rhodes. Visual wake words dataset, 2019.