Εθνικο Μετσοβιο Πολυτεχνειο

Σχολη Ηλεκτρολογων Μηχανικων και Μηχανικων Υπολογιστων

Τομεας Τεχνολογιας Πληροφορικης και Υπολογιστων

Εργαστηριο Μικροϋπολογιστων και Ψηφιακων Συστηματων

Utilizing Versal Architecture for Low-Latency Super Resolution Applications

Διπλωματικη Εργασια

Αφροδίτη Τζομάκα

Επιβλέπων: Δημήτριος Σούντρης
Καθηγητής Ε.Μ.Π.

Εργαστηριο Μικροϋπολογιστων και Ψηφιακων Συστηματων

Αθήνα, Οκτώβριος 2022

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

Utilizing Versal Architecture for Low-Latency Super Resolution Applications

Διπλωματικη Εργασια

Αφροδίτη Τζομάκα

Επιβλέπων: Δημήτριος Σούντρης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 17η Οκτωβρίου 2022.

(Υπογραφή)	*(Υπογραφή)*	*(Υπογραφή)*
.....	.....	.....
Δημήτριος Σούντρης	Παναγιώτης Τσανάκας	Σωτήρης Ξύδης
Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Επίκουρος Καθηγητής Χ.Π.Α

Αθήνα, Οκτώβριος 2022

(Υπογραφή)

.....

Αφροδιτη Τζομακα

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

Περίληψη

Οι εφαρμογές που κατακλύζουν τη σύγχρονη αγορά, όπως η μηχανική μάθηση, η επεξεργασία ψηφιακού σήματος και οι εφαρμογές 5G, απαιτούν ολοένα και πιο αυξημένη υπολογιστική δύναμη. Το τέλμα στο οποίο καταφτάνει ο νόμος του Moore αλλά και το τέλος της κλιμάκωσης του Dennard, έδωσαν το έναυσμα για την εξερεύνηση ετερογενών αρχιτεκτονικών και ειδικών επιταχυντών υλικού, πέραν των βαθμωτών μονάδων επεξεργασίας (CPUs), προκειμένου οι παραπάνω απαιτήσεις να ικανοποιηθούν. Οι υπάρχουσες λύσεις, όπως οι μονάδες επεξεργασίας γραφικών (GPUs), οι ψηφιακοί επεξεργαστές σήματος (DSPs) και οι συστοιχίες επιτόπιων προγραμματιζόμενων πυλών FPGAs, είτε πάσχουν σε ζητήματα μνήμης είτε η αυξημένη τεχνογνωσία που απαιτούν εμποδίζει την ευρεία ενσωμάτωσή τους στην αγορά. Στην κατεύθυνση αυτή, η Xilinx σχεδίασε και ανέπτυξε μια νέα γενιά ισχυρών, υβριδικών, υπολογιστικών πλατφορμών ονόματι Versal - Adaptive Compute Acceleration Platforms (ACAPs), που συνδυάζει τη βαθμωτή επεξεργασία, την προγραμματιζόμενη λογική και ειδικούς επιταχυντές πολύ μεγάλης παραλληλίας. Ταυτόχρονα, αναπτύχθηκαν και τα αντίστοιχα προγραμματιστικά εργαλεία για τις πλατφόρμες αυτές, τα οποία προσφέρουν πολλαπλά επίπεδα αφαίρεσης με κύριο γνώμονα την προγραμματιστική ευκολία.

Στη παρούσα διπλωματική εργασία εξερευνήσαμε τις δυνατότητες των καινοτόμων αυτών αρχιτεκτονικών και εργαλείων, με στόχο την επιτάχυνση εφαρμογών μηχανικής μάθησης που εκτελούνται στο «άκρο» (edge-computing). Πιο συγκεκριμένα, διαλέγουμε την εργασία του Super-Resolution, της αύξησης, δηλαδή, της ποιότητας μιας εικόνας με την χρήση ενός συνελικτικού νευρωνικού δικτύου, εν προκειμένω, του μοντέλου ESPCN. Στη διάθεση μας έχουμε την πλατφόρμα VCK190, της σειράς Versal AI Core, η οποία είναι ειδικά σχεδιασμένη για την επιτάχυνση νευρωνικών δικτύων και διαθέτει 400 AI Engines. Παρουσιάζουμε, λοιπόν, δύο εναλλακτικές υλοποιήσεις του προαναφερθέντος μοντέλου. Μία η οποία έχει σχεδιαστεί και αναπτυχθεί με γνώση του υλικού και στόχο την εκμετάλλευση των δυνατοτήτων όλων των υπολογιστικών πυρήνων της πλατφόρμας (hardware-specific) και μία η οποία χρησιμοποιεί το λογισμικό Vitis AI, το οποίο αποτελεί ένα αυτόματο εργαλείο για την ανάπτυξη μοντέλων AI αφαιρώντας απο τον χρήστη την ανάγκη για γνώση προγραμματισμού του υλικού (hardware-agnostic).

Συγκρίνουμε τις υλοποιήσεις μας έχοντας ως αναφορά την υλοποίηση του μοντέλου σε επίπεδο λογισμικού που εκτελείται στην CPU της πλατφόρμας VCK190. Τα αποτελέσματα που λαμβάνουμε είναι πολύ ενθαρρυντικά. Η hardware-specific υλοποίηση παρουσιάζει εξαιρετικές επιδόσεις φτάνοντας τα 518 FPS και παρουσιάζοντας μέγιστη επιτάχυνση κατά 1,87x σε σχέση με την CPU και 1,36x σε σχέση με την Vitis AI υλοποίηση, χωρίς να εξαντλούμε τους πόρους της πλατφόρμας και με ελάχιστες απώλειες στην ποιότητα της εικόνας. Η Vitis AI υλοποίηση ωστόσο, βλέπουμε ότι πάσχει σε θέματα και ποιότητας εικόνας αλλά και επιδόσεων, αναδεικνύοντας τη σημαντικότητα του διλλήματος «επιδόσεις - προγραμματιστική ευκολία».

Λέξεις Κλειδιά

Ετερογενείς Αρχιτεκτονικές, Ενσωματωμένα Συστήματα, Versal ACAPs, AI Engines, Vitis AI, Συνελικτικά Νευρωνικά Δίκτυα, Super Resolution.

Abstract

The applications flooding the modern market, such as Machine Learning (ML), digital signal processing, and 5G applications, require ever-increasing computing power. The impasse reached by Moore's law and the end of Dennard's scaling, gave the impetus for the exploration of alternative, heterogeneous architectures and hardware accelerators, beyond the conventional scalar processing units (CPUs), in order to satisfy the above requirements. Existing solutions, such as graphic processing units (GPUs), digital signal processors (DSPs), and field-programmable gate arrays (FPGAs), either suffer from memory issues or their increased hardware expertise prevents their widespread market adoption. In this direction, Xilinx developed and launched a new generation of powerful hybrid computing platforms, called Versal - Adaptive Compute Acceleration Platforms (ACAPs), which combines scalar processing (CPUs), programmable logic and dedicated accelerators of very high parallelism. At the same time, a set of programming tools to support the programmability of these platforms, was developed, with multiple levels of abstraction and the main focus on ease-of-use.

In this thesis, we explored the potential of these innovative architectures and tools, aiming to accelerate ML applications running at the "edge" (edge-computing). More specifically, we chose the task of Super-Resolution (SR), that is, increasing the quality of an image using a Convolutional Neural Network (CNN), in our case, the ESPCN model. We have at our disposal the VCK190 platform from the Versal AI Core series, which is specifically designed for accelerating deep neural networks and has 400 AI Engines. We present two alternative implementations of the aforementioned model for this platform. One that is designed and developed with knowledge of the underlying hardware and aims to exploit the capabilities of all the computing cores of the platform (hardware-specific) and one that uses the Vitis AI tool, which is an automatic tool for developing ML applications, removing from the user the need for hardware programming knowledge (hardware-agnostic).

We compare our implementations with reference to the baseline software implementation of the model, running on VCK190 device's CPU. The results we get are very encouraging. The hardware-specific implementation shows excellent performance reaching 518 FPS and presenting a maximum acceleration of 1.87x compared to the CPU and 1.36x compared to the Vitis AI implementation, even without reaching the resources' limits and with minimal losses in image quality. The Vitis AI implementation, however, suffers both in terms of image quality and performance compared to what we expected, highlighting the importance of the "performance - programming ease" trade-off.

Keywords

Heterogeneous Architectures, Embedded Systems, Versal ACAPs, AI Engines, Vitis AI, Convolutional Neural Networks, Super Resolution.

Ευχαριστίες

Η παρούσα διπλωματική σηματοδοτεί το τέλος 5 ετών διαρκούς προσπάθειας και απόκτησης γνώσης. Αισθάνομαι την ανάγκη, λοιπόν, να ευχαριστήσω τους ανθρώπους που συντέλεσαν στην περάτωσή της και με συντρόφευσαν όλα αυτά τα χρόνια. Αρχικά, είμαι ευγνώμων στον επιβλέποντα καθηγήτη κ. Δημήτριο Σούντρη για την εμπιστοσύνη του, τις συμβουλές και τις ευκαιρίες τις οποίες μου προσέφερε. Το πάθος του για την καινοτομία και η ώθηση που δίνει στους φοιτητές ώστε να εξελιχθούν, είναι συγκινητικά. Συνεχίζοντας, ευχαριστώ απεριόριστα τον υποψήφιο διδάκτορα Δημήτριο Δανόπουλο για την καθοδήγηση, τη διαρκή του υποστήριξη και παρουσία. Φυσικά, ευχαριστώ τους συναδέλφους και φίλους μου Άννα, Ευγενία, Λευτέρη και κυρίως Αγγελική και Νικόδημο για τις άψογες συνεργασίες, την αλληλοστήριξη και τις όμορφες στιγμές. Ιδιαίτερο ευχαριστώ οφείλω στον Παναγιώτη, για την βοήθεια και την αμέριστη αγάπη. Τέλος, ευχαριστώ από καρδιάς και αφιερώνω την παρούσα διπλωματική σε όλη μου την οικογένεια και ιδιαίτερα στους γονείς μου, Βαγγέλη και Ελένη, καθώς αποτελεί την κεφαλαιοποίηση των θυσιών και των κόπων που έκαναν - και κάνουν - όλα αυτά τα χρόνια για να μας προσφέρουν ένα καλύτερο μέλλον.

Contents

List of Figures

List of Tables

Εκτεταμένη Περίληψη

Εισαγωγή: Οι τεχνολογικές προκλήσεις που οδήγησαν στην δημιουργία των Versal ACAP

Η τεχνολογική εποχή στην οποία βρισκόμαστε χαρακτηρίζεται από 2 αντικρουόμενες συνθήκες, οι οποίες γεννούν προκλήσεις για την βιομηχανία των ημιαγωγών. Η πρώτη αφορά στη ραγδαία αύξηση των υπολογιστικών απαιτήσεων των σύγχρονων Machine Learning (ML) εφαρμογών που κατακλύζουν την αγορά. Η αλγοριθμικές καινοτομίες και η έκρηξη δεδομένων οδηγούν σε μια εκθετική αύξηση των υπολογιστικών απαιτήσεων, οι οποίες διπλασιάζονται περίπου ανά 3.5 μήνες τη στιγμή που ο νόμος του Moore διπλασιαζόταν "μόνο" κάθε 18 μήνες [13]. Το τέλος της αναγεννησιακής περιόδου για την βελτιστοποίηση του υλικού, που επέτρεπε η κλιμάκωση του Dennard σε συνδυασμό με τους νόμους των Moore και Amdahl, αποτελεί τη δεύτερη συνθήκη η οποία οδήγησε τους κατασκευαστές να στραφούν προς την εξερεύνηση ετερογενών, domain-specific αρχιτεκτονικών.

Στην κατεύθυνση αυτή αναπτύχθηκαν οι λεγόμενοι very large vector processing υπολογιστικοί πυρήνες, όπως τα DSPs και οι GPUs. Οι συσκευές αυτές θεωρητικά περικλύουν φοβερή υπολογιστική ισχύ, ωστόσο η ρεαλιστική τους απόδοση περιορίζεται από την τεράστια διαφορά στις επιδόσεις του επεξεργαστικού πυρήνα και της μνήμης. Τα παραδοσιακά FPGAs έρχονται ως λύση προσφέροντας προγραμματισιμότητα και εξειδίκευση. Παρότι όμως βελτιώνουν την χρονική καθυστέρηση, η ανάγκη κατανόησης του υλικού και η προγραμματιστική του δυσκολία, αποτρέπει την ευρεία ενσωμάτωσή τους στην αγορά.

Η Xilinx, προσπαθώντας να αντιμετωπίσει αποτελεσμάτικα όλα τα παραπάνω, παρουσιάζει μια νέα γενιά ετερογενών συσκευών ονόματι Versal - Adaptive Compute Acceleration Platforms (ACAPs). Οι πλατφόρμες αυτές σχεδιάστηκαν με γνώμονα τις υψηλές επιδόσεις σε υπολογιστικά απαιτητικές εφαρμογές όπως το ML ή το 5G και τη προγραμματισιμότητα. Περιέχουν ισχυρές CPU, βελτιστοποιημένη προγραμματιζόμενη λογική, DSPs και επιπλεόν ειδικούς επιταχυντές υλικού. Οι επιταχυντές αυτοί ονομάζονται AI Engines και υπόσχονται επαναστατικές επιδόσεις σε εφαρμογές όπως η βαθειά μάθηση, κρατώντας χαμηλή την κατανάλωση ισχύος. Τέλος, η απαίτηση της προγραμματισιμότητας έρχεται να καλυφθεί από μία σειρά προγραμματιστικών εργαλείων που προσφέρουν μια πληθώρα αφαιρετικών επιπέδων ανάλογα με τις ανάγκες του χρήστη.

Θεωρητικό Υπόβαθρο

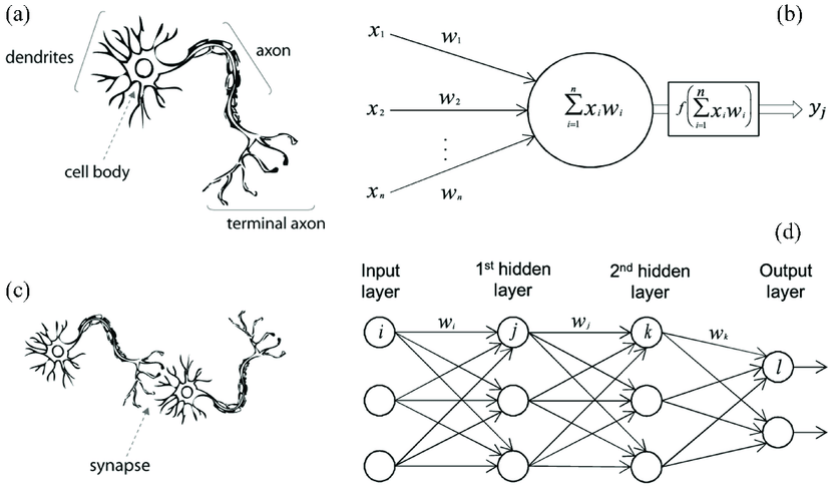
Μηχανική Μάθηση

Η Μηχανική Μάθηση αποτελεί παρακλάδι των ευρύτερων όρων Τεχνητή Νοημοσύνη και Επιστήμη των Υπολογιστών. Στόχος της Μηχανικής Μάθησης είναι η απόκτηση της ικανότητας της γενίκευσης από μία "μηχανή". Πιο συγκεκριμένα, αφορά στην ανάπτυξη αλγορίθμων ικανών να εκπαιδευτούν

σε ένα σύνολο δεδομένων με στόχο την επιτυχή πρόβλεψη αποτελεσμάτων σε ένα ευρύτερο σύνολο εργασιών χωρίς να έχουν προγραμμτιστεί εξ΄ αρχής, συγκεκριμένα, γι΄ αυτές τις εργασίες. Μπορούμε να κάνουμε λόγο για τρεις κατηγορίες συστημάτων Μηχανικής Μάθησης, την Επιβλεπόμενη Μάθηση όπου ο αλγόριθμος εκπαιδεύεται σε ένα σύνολο από γνωστά δεδομένα εισόδου-εξόδου, τη Μη-Επιβλεπόμενη Μάθηση όπου το σύστημα καλείται να αποφανθεί για δεδομένα χωρίς ετικέτες και χωρίς προκαθορισμένη δομή και την Ενισχυτική Μάθηση η οποία περιέχει ένα υποκείμενο - πράκτορα, ο οποίος δρα σε ένα περιβάλλον επιχειρώντας να μεγιστοποιήσει την "επιβράβευση" των δράσεων του.

Τεχνητά Νευρωνικά Δίκτυα

Τα τεχνητά νευρωνικά δίκτυα (ΤΝΔ) είναι υπολογιστικά συστήματα εμπνευσμένα από τον ανθρώπινο εγκέφαλο και το πιο δημοφιλές υποσύνολο της μηχανικής μάθησης. Λειτουργούν ως ένα παράλληλο και κατανεμημένο σύστημα επεξεργασίας που αποτελείται από πολλαπλούς, συνδεδεμένους, απλούς υπολογιστικούς κόμβους που έχουν τη δυνατότητα να αποκτούν εμπειρική γνώση από το περιβάλλον τους και να την αποθηκεύουν για μελλοντική χρήση[14]. Αυτοί οι υπολογιστικοί κόμβοι ονομάζονται τεχνητοί νευρώνες, ή απλά νευρώνες, και αντιστοιχούν στους νευρώνες του ανθρώπινου εγκεφάλου. Μιμούμενοι τη λειτουργικότητα του εγκεφάλου, η γνώση αποκτάται μέσω μιας διαδικασίας μάθησης και αποθηκεύεται στις συνδέσεις μεταξύ των νευρώνων - το ισοδύναμο των ανθρώπινων συνάψεων.



Σχήμα 1: Βιολογικά και τεχνητά νευρωνικά δίκτυα και νευρώνες: (α) βιολογικός νευρώνας· (β) τεχνητός νευρώνας· (γ) βιολογικές συνάψεις· και (δ) συνάψεις ΤΝΔ [1].

Το μοντέλο ενός τεχνητού νευρώνα είναι η βάση οποιουδήποτε ΤΝΔ και μοιάζει με το αντίστοιχο βιολογικό μοντέλο. Τα κύρια μέρη του είναι τα συναπτικά βάρη, ο αθροιστής και η συνάρτηση ενεργοποίησης. Τα συναπτικά βάρη αντιπροσωπεύουν τη δύναμη μιας σύνδεσης που αντιστοιχούν στις βιολογικές συνάψεις. Τα σήματα εισόδου ενός νευρώνα πολλαπλασιάζονται με τα συναπτικά βάρη του και τα αποτελέσματα του πολλαπλασιασμού προστίθενται στο σώμα του, τον αθροιστή, όπως φαίνεται στο σχήμα 1. Τότε, η συνάρτηση ενεργοποίησης είναι υπεύθυνη για τον περιορισμό του αθροίσματος. Συχνά προστίθεται μια εξωτερική πόλωση στην έξοδο του αθροιστή προκειμένου να αυξηθεί ή να μειωθεί η διέγερση της συνάρτησης ενεργοποίησης.

Πολλαπλοί νευρώνες συγκεντρώνονται σε στρώματα για να σχηματίσουν μια τοπολογία ΤΝΔ. Συνήθως, υπάρχει ένα επίπεδο εισόδου, ένα επίπεδο εξόδου και ένα ή περισσότερα κρυφά επίπεδα.

Κάθε κόμβος ενός επιπέδου συνδέεται με έναν κόμβο ενός άλλου επιπέδου, μεταδίδοντας πληροφορίες μεταξύ τους και σε όλο το δίκτυο. Ένας νευρώνας λαμβάνει ένα σήμα, το επεξεργάζεται όπως περιγράφεται παραπάνω και στη συνέχεια ανάλογα με το αποτέλεσμα της συνάρτησης ενεργοποίησης δίνει σήμα στους συνδεδεμένους σε αυτόν νευρώνες ώστε να εκτελέσουν και εκείνοι την ίδια εργασία.

Η διαδικασία μάθησης, ή συνήθως η εκπαίδευση (training), ενός ΤΝΔ μπορεί να είναι επιβλεπόμενη, μη επιβλεπόμενη ή να χρησιμοποιεί το μοντέλο ενισχυτικής μάθησης. Σε αυτή τη διατριβή χρησιμοποιείται η διαδικασία της επιβλεπόμενης μάθησης, γι΄ αυτό θα εστιάσουμε στη συνέχεια εκεί.

Η εποπτευόμενη διαδικασία εκμάθησης χρησιμοποιεί εκπαίδευση σε παραδείγματα δεδομένων για να αυξήσει την ακρίβεια πρόβλεψης του ΤΝΔ με την πάροδο του χρόνου. Τα δεδομένα έχουν τη μορφή γνωστών ζευγών εισόδου-εξόδου. Το πρώτο βήμα της εκπαίδευσης είναι το λεγόμενο forward-step, το οποίο, βασιζόμενο στον υπολογισμό του σταθμισμένου αθροίσματος όπως αναλύθηκε προηγουμένως, πραγματοποιεί τη πρόβλεψη. Στη συνέχεια, η πρόβλεψη συγκρίνεται με την έξοδο στόχο και το σφάλμα πρόβλεψης ορίζεται μέσω μιας συνάρτησης απώλειας. Αυτό το σφάλμα διαδίδεται σε όλο το δίκτυο χρησιμοποιώντας έναν αλγόριθμο back-propagation και προκαλεί την προσαρμογή των συναπτικών βαρών του δικτύου. Μετά από έναν πεπερασμένο αριθμό επαναλήψεων, η πρόβλεψη γίνεται πολύ ακριβής και η διαδικασία εκπαίδευσης σταματά. Η γνώση που αποθηκεύεται στο δίκτυο είναι στην πραγματικότητα τα προσαρμοσμένα συναπτικά βάρη και το forward-step για την πραγματοποίηση προβλέψεων μετά την εκπαίδευση ονομάζεται inference.

Συνελικτικά Νευρωνικά Δίκτυα

Τα συνελικτικά νευρωνικά δίκτυα (ΣΝΔ) είναι ένας τύπος ΤΝΔ εμπνευσμένος από το οπτικό σύστημα των ζώων και χρησιμοποιούνται κυρίως για εργασίες ταξινόμησης και όρασης υπολογιστών. Τα ΣΝΔ ακολουθούν την ίδια ακριβώς διαδικασία εκπαίδευσης και εξαγωγής προβλέψεων που περιγράφεται παραπάνω για τα κανονικά ΤΝΔ. Η διαφορά έγκειται στο γεγονός ότι τα ΣΝΔ γνωρίζουν η είσοδος τους θα είναι εικόνα, το οποίο τους επιτρέπει να κωδικοποιήσουν ορισμένες ιδιότητες στην αρχιτεκτονική τους. Αυτές οι ιδιότητες λοιπόν, όχι μόνο κάνουν την υλοποίηση του forward-step πιο αποτελεσματική αλλά επίσης οδηγούν σε ακραία μείωση του αριθμού των εκπαιδεύσιμων παραμέτρων στο δίκτυο, δηλαδή των συναπτικών βαρών και προκαταλήψεων [15].

Μια αρχιτεκτονική ΣΝΔ αποτελείται από τρεις κύριους τύπους επιπέδων: Συνελικτικά (Convolutional - Conv) επίπεδα, Pooling (Pool) επίπεδα και Πλήρως Συνδεδεμένα (Fully-Connected - FC) επίπεδα. Κατά κανόνα, η ίδια η εικόνα εισόδου ονομάζεται στρώμα εισόδου ενός ΣΝΔ. Τα άλλα τρία επίπεδα μπορούν να στοιβάζονται με διαφορετικούς τρόπους για να διαμορφώσουν διαφορετικές αρχιτεκτονικές ΣΝΔ. Ας αναλύσουμε εν συντομία τα επίπεδα του ΄ΝΝ:

- **Στρώμα εισόδου:** Συγκρατεί τις τιμές των pixel της εικόνας εισόδου. Οι διαστάσεις του στρώματος ταυτίζονται με τις διαστάσεις της εικόνας εισόδου.

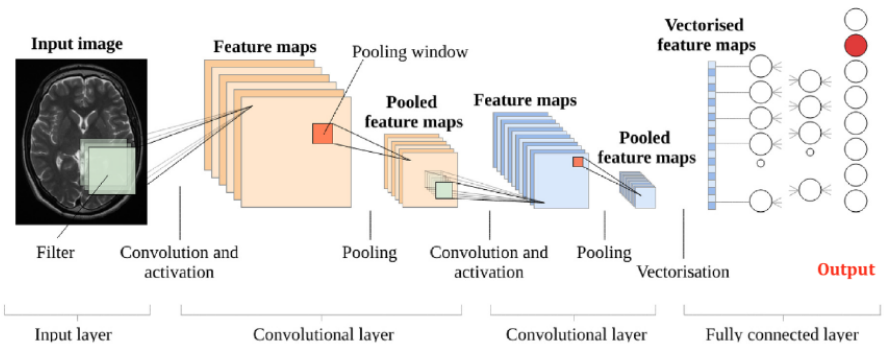
- **Συνελικτικό στρώμα (Conv):** Το βασικό δομικό στοιχείο ενός ΣΝΔ. Στον υπολογισμό του περιλαμβάνει τρεις κύριες έννοιες: ένα επίπεδο εισόδου όπως εξηγήθηκε προηγουμένως, ένα φίλτρο, γνωστό και ως kernel, και έναν χάρτη χαρακτηριστικών (feature map). Το φίλτρο είναι βασικά το παράθυρο της συνέλιξης, ένας διδιάστατος πίνακας βαρών που αντιστοιχίζεται σε μια τοπική περιοχή της εικόνας σύμφωνα με τις διαστάσεις της, το λεγόμενο πεδίο υποδοχής (receptive field). Αυτός ο πυρήνας ολισθαίνει στο επίπεδο εισόδου από αριστερά προς τα δεξιά και από πάνω προς τα κάτω υπολογίζοντας τα αντίστοιχα γινόμενα. Αυτή η διαδικασία, γνωστή ως συνέλιξη, χρησιμοποιείται στα ΣΝΔ για εξαγωγή χαρακτηριστικών και η έξοδος του ονομάζεται χάρτης χαρακτηριστικών. Ο αριθμός και οι διαστάσεις των χαρτών χαρακτηριστικών εξόδου εξαρτώνται από το μέγεθος και τον αριθμό των συνελικτικών πυρήνων, το βήμα της

συνέλιξης και το πιθανό padding. Τέλος, κάθε συνελικτικό στρώμα εφαρμόζει μια συνάρτηση ενεργοποίησης όπως ReLu, Sigmoid ή Tanh που μπορεί να θεωρηθεί ως ένα επιπλέον στρώμα, το στρώμα ενεργοποίησης.

- **Στρώμα Pooling:** Μια μορφή μη γραμμικής υποειγματοληψίας. Παρόμοια με τον συνελικτικό πυρήνα που συζητήθηκε παραπάνω, το pooling στρώμα ολισθαίνει έναν πυρήνα κατά μήκος της εισόδου, που δεν έχει βάρη και εφαρμόζει μόνο μια συνάρτηση συνάθροισης στο πεδίο υποδοχής. Το pooling στρώμα συμβάλλει στη σταδιακή μείωση του χωρικού μεγέθους της αναπαράστασης, στη μείωση του αριθμού των εκπαιδεύσιμων παραμέτρων, στη βελτίωση της απόδοσης και, συνεπώς, στον περιορισμό του κινδύνου υπερπροσαρμογής (over-fitting). Υπάρχουν αρκετές μη γραμμικές συναρτήσεις για την υλοποίηση του στρώματος αυτού, με πιο κοινή το max pooling.

- **Πλήρως συνδεδεμένο στρώμα (Fully-Connected - FC):** Στα συνελικτικά στρώματα που περιγράφονται παραπάνω, οι νευρώνες συνδέονται μόνο με ένα μέρος του όγκου εισόδου, το receptive field. Αντίθετα, κάθε νευρώνας σε ένα FC στρώμα συνδέεται απευθείας με κάθε έναν κόμβο του προηγούμενου επιπέδου, ακριβώς όπως τα κανονικά ΤΝΔ. Αυτό το επίπεδο εκτελεί την τελική ταξινόμηση με βάση τα χαρακτηριστικά που εξήχθησαν σε προηγούμενα επίπεδα. Τα επίπεδα αυτά συνήθως χρησιμοποιούν μια συνάρτηση ενεργοποίησης τύπου softmax για να παράγουν μια πιθανότητα από 0 έως 1 για να ταιριάζουν με τις ανάγκες ταξινόμησης.

Ένα παράδειγμα αρχιτεκτονικής ΣΝΔ με όλα τα προαναφερθέντα κύρια δομικά στοιχεία απεικονίζεται στην παρακάτω εικόνα:



Σχήμα 2: Αρχιτεκτονική ΣΝΔ[2].

Super Resolution

Η εργασία του Image Super-Resolution (SR) αφορά στην ανακατασκευή εικόνων υψηλής ανάλυσης (HR) από δεδομένες εικόνες χαμηλής ανάλυσης (LR) και είναι ένας από τους σημαντικότερους τομείς της όρασης υπολογιστών και της επεξεργασίας εικόνας. Εκτός από το ακαδημαϊκό και ερευνητικό ενδιαφέρον, έχει επίσης πολυάριθμες εφαρμογές στον πραγματικό κόσμο, όπως η δορυφορική απεικόνιση, η ιατρική απεικόνιση, η επιτήρηση και η ασφάλεια. Ωστόσο, το SR είναι ένα αρκετά απαιτητικό πρόβλημα, καθώς υπάρχουν πάντα πολλές εικόνες HR που αντιστοιχούν σε μία μόνο εικόνα LR. Στη βιβλιογραφία, υπάρχει πληθώρα κλασικών τεχνικών SR, όπως αυτές που βασίζονται στην πρόβλεψη ή στη στατιστική. Ωστόσο, με τη διάδοση της μηχανικής μάθησης, έχει προκύψει μια σημαντική και πολλά υποσχόμενη έρευνα σε προσεγγίσεις βαθιάς μάθησης για την αντιμετώπιση του SR, η οποία συχνά αποφέρει επιδόσεις αιχμής. Μια ποικιλία αρχιτεκτονικών βαθιάς μάθησης, συμπεριλαμβανομένου των ΣΝΔ έχουν προταθεί καθώς και τα καινοτόμα μοντέλα ανταγωνιστικής μάθησης (Generative Adversarial Nets - GANs).

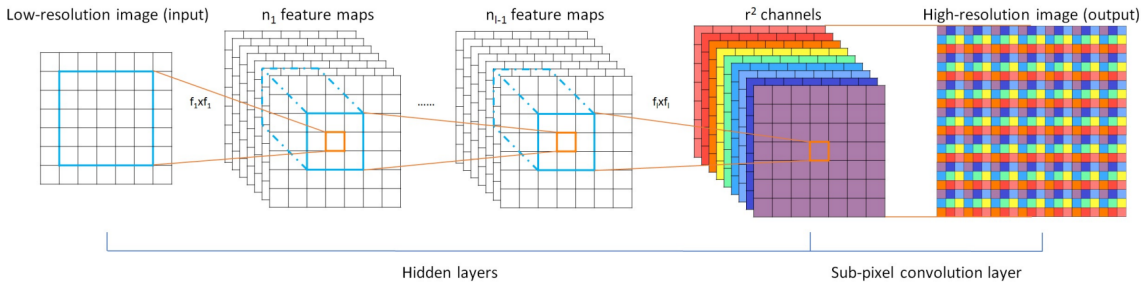
Η αρχιτεκτονική ESPCN

Προκειμένου να αντιμετωπιστούν περιορισμοί στην υπολογιστική πολυπλοκότητα και στην ποιότητα του SR, προτάθηκε το δίκτυο ESPCN (Efficient Sub-Pixel Convolutional Neural Network). Το δίκτυο αυτό εισάγει στην αρχιτεκτονική του ΣΝΔ ένα αποτελεσματικό συνελικτικό επίπεδο που βασίζεζεται στην τεχνική του sub-pixel. Στο δίκτυο ESPCN η αύξηση της ποιότητας και ταυτόχρονα των διαστάσεων γίνεται στο τέλος του δικτύου, το οποίο σημαίνει ότι δεν απαιτείται κάποιοι είδους interpolation για την είσοδο και έτσι το δίκτυο είναι σε θέση να μάθει μια καλύτερη αντιστοίχιση μεταξύ των ζευγών LR-HR εικόνων. Επίσης, ο μειωμένος όγκος εισόδου απαιτεί μικρότερα μεγέθη φίλτρων, μειώνει την πολυπλοκότητα των υπολογισμών και επομένως το δίκτυο γίνεται ελαφρύτερο. Αυτή η βελτιωμένη απόδοση καθιστά το ESPCN ιδανική επιλογή για εργασίες SR ακόμη και σε βίντεο HD σε πραγματικό χρόνο.

Περιγραφή Αρχιτεκτονικής

Το μοντέλο ESPCN αναμένει ως είσοδο μια εικόνα LR και έναν ακέραιο επί τον οποίο θα πολλαπλασιαστούν οι διαστάσεις της εισόδου (upscale factor). Η εικόνα LR παράγεται με υπό-δειγματοληψία των εικόνων HR του συνόλου δεδομένων σύμφωνα με το upscale factor. Στην πραγματικότητα, η είσοδος του δικτύου θεωρείται ότι είναι μόνο το κανάλι φωτεινότητας στον χρωματικό χώρο YCbCr. Η έξοδος θα είναι η ανακατασκευασμένη, SR εικόνα.

Η αρχιτεκτονική του ESPCN απεικονίζεται στο Σχήμα 3. Για ένα δίκτυο που αποτελείται από L επίπεδα, τα πρώτα $L-1$ είναι κλασικά επίπεδα Conv για εξαγωγή χαρακτηριστικών και το τελευταίο είναι υπεύθυνο για την αντιστοίχιση στον χώρο υψηλών διαστάσεων, σύμφωνα με το upscale factor, έστω r. Αυτό το τελικό επίπεδο είναι το αποτελεσματικό sub-pixel συνελικτικό επίπεδο.



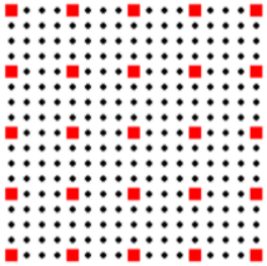
Σχήμα 3: Η αρχιτεκτονική του μοντέλου ESPCN[3].

Κατά κύριο λόγο για το ESPCN ισχύει $L = 3$ και αναλυτικά τα επίπεδα του δικτύου έχουν τις ακόλουθες διαστάσεις και παραμέτρους:

- Επίπεδο εισόδου: Υποθέτουμε ότι έχουμε ένα στρώμα εισόδου με εικόνα διαστάσεων $[B, C, N, N]$.
- Πρώτο στρώμα Conv: 64 φίλτρα με μέγεθος πυρήνα 5×5, βήμα 1×1 και padding 2×2, ακολουθούμενα από ένα στρώμα ενεργοποίησης Tanh.
- Δεύτερο στρώμα Conv: 32 φίλτρα με μέγεθος πυρήνα 3×3, βήμα 1×1 και padding 1×1, ακολουθούμενα από ένα στρώμα ενεργοποίησης Tanh.
- Τρίτο στρώμα Conv: Φίλτρα $C \times r \times r$, με μέγεθος πυρήνα 3×3, βήμα 1×1 και padding 1×1.
- Στρώμα PixelShuffle: Η λειτουργία τυχαίας ανακατανομής των sub-pixel που δίνει στην εικόνα εξόδου τις επιθυμητές διαστάσεις, δηλαδή $[B, C, r \times r, r \times r]$.

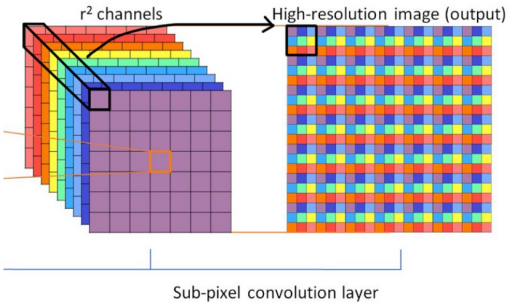
Η συνέλιξη Sub-Pixel

Στο σύστημα απεικόνισης μιας κάμερας, λόγω των περιορισμών του οπτικού αισθητήρα, οι εικόνες περιορίζονται στην αρχική ανάλυση pixel. Επομένως, τα αντικείμενα του πραγματικού κόσμου σε μια εικόνα κβαντίζονται χωρικά σε αυτή τη σταθερή ανάλυση. Ωστόσο, στον μικροσκοπικό κόσμο υπάρχουν μικροσκοπικά pixel μεταξύ δύο γειτονικών φυσικών pixel. Αυτά τα μικροσκοπικά εικονοστοιχεία ονομάζονται sub-pixels και απεικονίζονται στο σχήμα 4. Κάθε pixel του συστήματος απεικόνισης είναι στην πραγματικότητα κάθε τετράγωνη περιοχή που ορίζεται από τέσσερα κόκκινα τετράγωνα, όπου οι μαύρες κουκκίδες είναι τα sub-pixels. Η ακρίβεια των sub-pixels μπορεί να ρυθμιστεί ανάλογα με το interpolation μεταξύ των γειτονικών pixel. Με αυτόν τον τρόπο, η αντιστοίχιση από μικρές τετράγωνες περιοχές σε μεγάλες τετραγωνικές περιοχές μπορεί να υλοποιηθεί μέσω interpolation των sub-pixels. [4]



Σχήμα 4: Αναπαράσταση των sub-pixels [4].

Η συνέλιξη sub-pixel, επίσης γνωστή ως PixelShuffle, είναι μια από τις πιο αξιοσημείωτες μεθόδους που εισήχθησαν μέσω του ESPCN. Περιλαμβάνει μια γενική λειτουργία συνέλιξης που ακολουθείται από μια αναδιάταξη των πιξελς χρησιμοποιώντας τη θεωρία των sub-pixels. Το κανάλι εξόδου του τελευταίου επιπέδου πρέπει να είναι $C \times r \times r$, έτσι ώστε ο συνολικός αριθμός των pixel να είναι συνεπής με την HR εικόνα του συνόλου δεδομένων. Στη συνέχεια, η συνάρτηση PixelShuffle συνδυάζει κάθε sub-pixel σε χάρτες χαρακτηριστικών πολλαπλών καναλιών σε μια τετραγωνική περιοχή $r \times r$ στην εικόνα εξόδου. Έτσι, κάθε pixel στους χάρτες χαρακτηριστικών είναι ισοδύναμο με το δευτερεύον sub-pixel στην παραγόμενη εικόνα εξόδου:



Σχήμα 5: Η διαδικασία PixelShuffle[4].

Η αρχιτεκτονική των Versal ACAPs

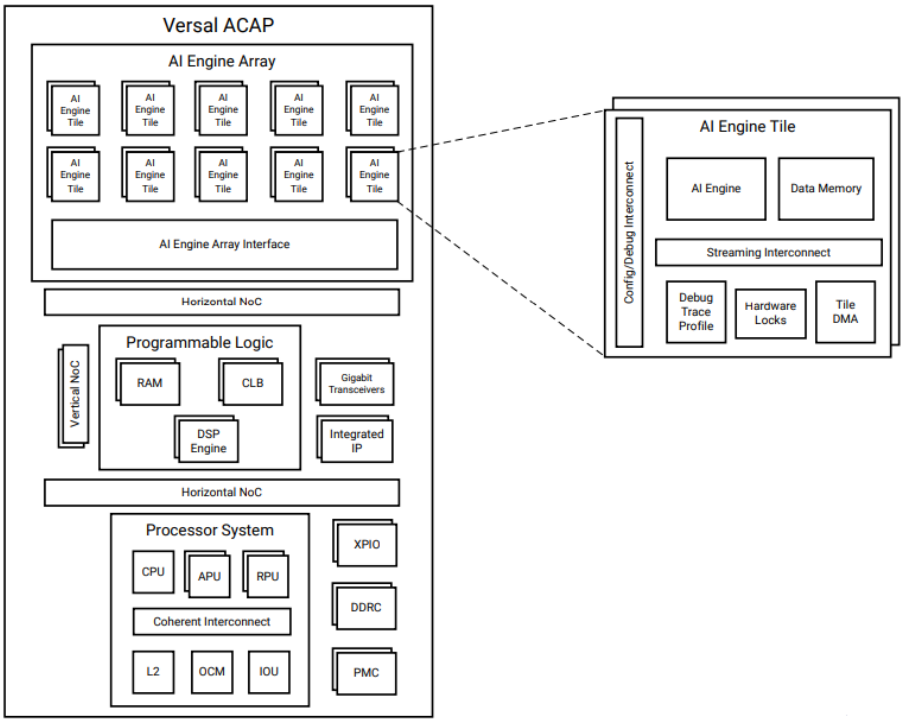
Η σειρά Versal περιλαμβάνει έξι κατηγορίες συσκευών που βασίζονται στην τεχνολογία 7nm FinFET και έχουν σχεδιαστεί μοναδικά για να παρέχουν κορυφαία ετερογενή επιτάχυνση για ένα ευρύ φάσμα εφαρμογών από το edge έως το cloud. Αυτές οι προσαρμοστικές πλατφόρμες ονόματι Adap-

tive Compute Acceleration Platforms (ACAPs) συνδυάζουν Scalar Engines, Adaptable Engines και Intelligent Engines με μια δυνατότητα απρόσκοπτης επικοινωνίας μέσω ενός προγραμματιζόμενου Network on a Chip (NoC). Επίσης, μαζί με αυτές τις συσκευές κυκλοφόρησε και μια πληθώρα εργαλείων, βιβλιοθηκών, IP και framework για να υποστηρίξουν την προγραμματισιμότητα τους και τις απαιτήσεις για χαμηλό time-to-market.

Σε αυτή τη διπλωματική εργασία θα δουλέψουμε με τη σειρά Versal AI Core και συγκεκριμένα το VCK190 Evaluation Kit. Αυτή η σειρά παρέχει την υψηλότερη απόδοση για AI Inference και επεξεργασίας σήματος για εφαρμογές cloud, network και edge [16]. Το VCK190 System on Chip (SoC) περιέχει πλήθος ισχυρών υπολογιστικών πόρων και διασυνδέσεων. Στις επόμενες ενότητες, θα περιγραφούν διεξοδικά οι κυριότεροι από αυτούς.

Τα AI Engines

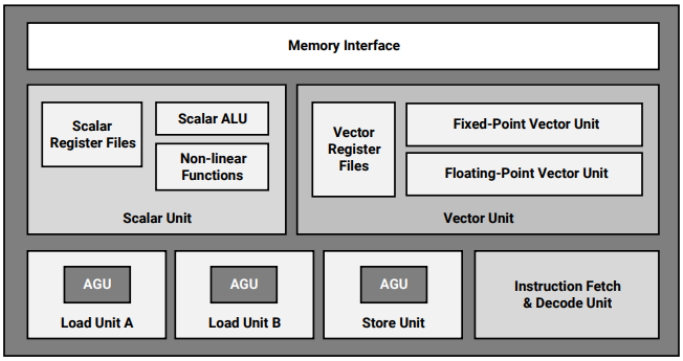
Για να ξεκινήσουμε την ανάλυση αυτής της εξελιγμένης αρχιτεκτονικής, ας δούμε ένα high-level μπλοκ διάγραμμα ενός Versal ACAP με ένα array από AI Engines (AIE) μαζί με τους άλλους κύριους τομείς, το σύστημα επεξεργαστή (PS) και την προγραμματιζόμενη λογική (PL):



Σχήμα 6: Versal Device Top-Level Block Diagram [5].

Όπως μπορούμε να δούμε, η αρχιτεκτονική AIE περιλαμβάνει τρία ιεραρχικά επίπεδα. Το πρώτο επίπεδο σε αυτήν την ιεραρχία είναι το AI Engine Array. Είναι βασικά μια συστοιχία 2 διαστάσεων από AI Engine Tiles, τα οποία αποτελούν το δεύτερο ιεραρχικό επίπεδο. Αυτά τα tiles ενσωματώνουν μνήμες, διασυνδέσεις και το θεμελιώδες μπλοκ της AI Engine αρχιτεκτονικής, το λεγόμενο AI Engine. Πρόκειται ουσιαστικά για έναν εξαιρετικά βελτιστοποιημένο, single instruction - multiple data (SIMD) και very large instruction word (VLIW) επεξεργαστή. Η συστοιχία AIE χρησιμοποιεί πολυάριθμες διεπαφές που καθιστούν τα AIE ικανά να επικοινωνούν μεταξύ τους αλλά και με το υπόλοιπο Versal SoC.

Ας αναλύσουμε τώρα κάθε ένα από τα προαναφερθέντα δομικά στοιχεία, ξεκινώντας από το θεμελιώδες, το **AI Engine**. Όπως αναφέρθηκε προηγουμένως, μιλάμε για έναν καινοτόμο VLIW και SIMD πυρήνα επεξεργασίας που υποστηρίζει ακρίβεια σταθερής και κινητής υποδιαστολής. Στο παρακάτω διάγραμμα μπορούμε να δούμε όλα τα χαρακτηριστικά του AIE:



Σχήμα 7: AI Engine [6].

- **Register Files:** Τα AIE υποστηρίζουν βαθμωτούς καταχωρητές, ειδικούς καταχωρητές, διανυσματικούς καταχωρητές πλάτους 128 - 1024 bit για να επιτρέπονται οι εντολές SIMD και αςσυμυλατορ καταχωρητές 384/786 bit για την αποθήκευση των αποτελεσμάτων της διανυσματικής διαδρομής δεδομένων.

- **Instruction Fetch & Decode:** Μέχρι επτά λειτουργίες μπορούν να εκδοθούν παράλληλα χρησιμοποιώντας μία VLIW λέξη όλες τις λειτουργικές μονάδες. Το μέγεθος της μνήμης προγράμματος είναι 16 KB.

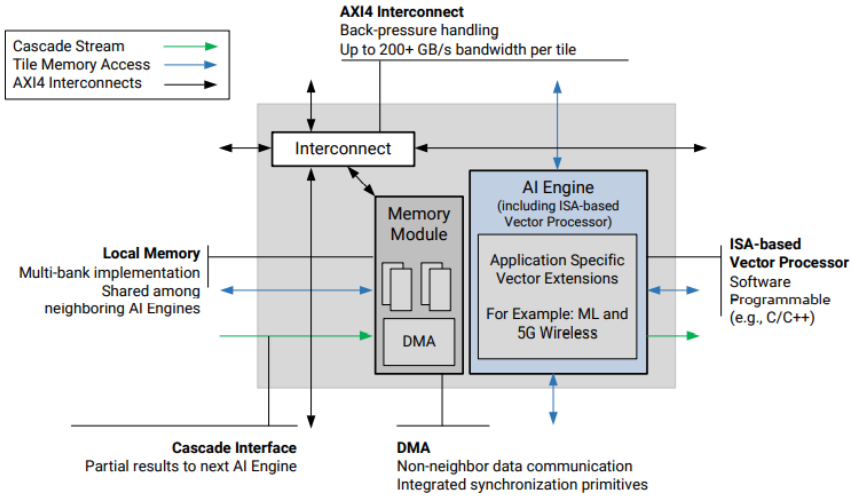
- **Load & Store Unit:** Υπάρχουν τρια data memory ports, δύο για φόρτωση και μία για αποθήκευση που μπορούν να λειτουργούν ταυτόχρονα. Οι αντίστοιχες μονάδες παραγωγής διευθύνσεων (AGU) υποστηρίζουν τη δημιουργία διευθύνσεων Fast Fourier Transform (FFT) για πολλαπλούς τρόπους διευθυνσιοδότησης (fixed, indirect, post-incremental, ή cyclic). Τα δεδομένα φορτώνονται ή αποθηκεύονται στη μνήμη δεδομένων για την οποία θα μιλήσουμε αργότερα.

- **Scalar Unit:** 32-bit RISC επεξεργαστής με general purpose pointer και configuration register files. Υποστηρίζει μη γραμμικές συναρτήσεις και μετατροπή τύπων δεδομένων μεταξύ βαθμωτών fixed-point αριθμών και βαθμωτών αριθμών κινητής υποδιαστολής. Ενσωματώνει επίσης μια βαθμωτή ALU με βαθμωτό πολλαπλασιαστή 32×32 bit.

- **Vector Fixed-Point Unit:** Περιλαμβάνει τρεις ξεχωριστές και σε μεγάλο βαθμό ανεξάρτητες διαδρομές δεδομένων, τη διαδρομή multiply - accumulate (MAC), τη διαδρομή upshift και τη διαδρομή shift-round saturate (SRS). Υποστηρίζει ταυτόχρονες λειτουργίες σε πολλαπλές διανυσματικές λωρίδες και διαθέτει πλήρη μονάδα permutation με 32 bit granularity. Μπορεί να εκτελέσει έως και 128 MAC/κύκλο για πραγματικούς τελεστές 8-bit και 8 MAC/κύκλο για σύνθετους τελεστές 16-bit.

- **Vector Floating-Point Unit:** Ίδιο permutation και concurrency με το Vector Fixed-Point Unit. Υποστηρίζει 8 floating-point MAC/κύκλο.

Τα AIE οργανώνται σε AI Engine Tiles. Τα tiles είναι το μεσαίο επίπεδο στην ιεραρχία της αρχιτεκτονικής AIE και αποτελείται από τα ακόλουθα:



Σχήμα 8: Ένα AI Engine Tile

- **Tile Interconnect Module:** Υποστηρίζει AXI4-Stream και memory-mapped AXI4 κίνηση. Το AXI4-Stream επιτρέπει τη μετακίνηση δεδομένων μεταξύ μη γειτονικών AIE tiles ή μεταξύ του AIE και του PL ή του NoC. Το memory-mapped AXI4 μπορεί να οδηγηθεί εκτός του AIE array από οποιοδήποτε AXI4 master που μπορεί να συνδεθεί στο NoC.

- **AIE Memory Module:** Διαθέτει 32 KB μνήμης δεδομένων, μια διεπαφή μνήμης, DMA, κλειδώματα και μονάδες ελέγχου, trace και debugging. Όταν η επικοινωνία πραγματοποιείται μέσα σε ένα AIE tile ή μεταξύ απευθείας γειτονικών AIE tile, το shared memory module χρησιμοποιείται με υποστήριξη ping-pong buffer και κλειδωμάτων για συγχρονισμό. Ωστόσο, για μη γειτονικά AIE tiles, παρόμοια, αλλά με αυξημένη καθυστέρηση και πόρους, επικοινωνία μπορεί να δημιουργηθεί μέσω του DMA σε κάθε μονάδα μνήμης.

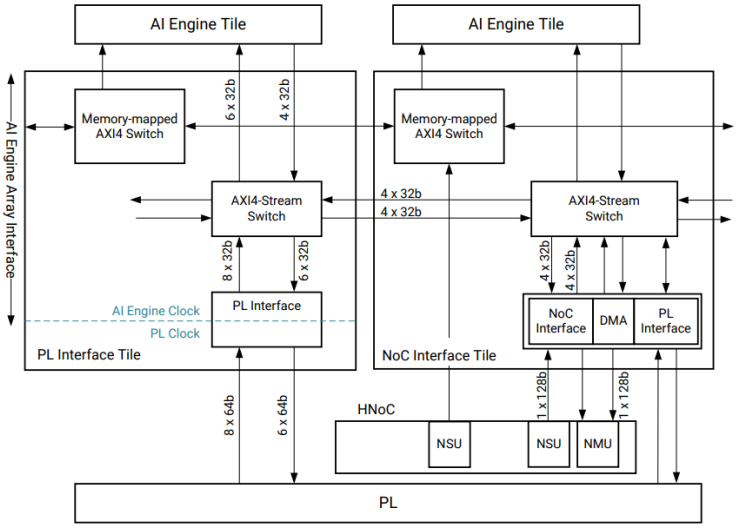
- **Το AI Engine:** Όπως περιγράφεται παραπάνω.

Όπως φαίνεται στο σχήμα 6 τα AIE tiles είναι οργανωμένα σε έναν διδιάστατο πίνακα, το AIE array. Ο αριθμός των tiles και κατά συνέπεια ο αριθμός των σειρών και των στηλών εξαρτάται από τη συσκευή. Για την πλακέτα VCK190 διατίθενται το μέγιστο των 400 tiles οργανωμένο σε μια διάταξη 2 διαστάσεων, 8×50. Επίσης, όπως μπορούμε να δούμε στο μπλοκ διάγραμμα 6, η τελευταία σειρά του AIE array φιλοξενεί τα interface tiles, τα οποία παρέχουν την απαραίτητη λειτουργικότητα για τη διασύνδεση του AIE array με την υπόλοιπη συσκευή. Υπάρχουν τρεις τύποι interface tile:

- **AI Engine configuration interface tile:** Υπάρχει ακριβώς ένα ανά AIE array. Περιέχει ένα PLL για τη δημιουργία του ρολογιού του AI Ενγινε και άλλες λειτουργίες καθολικού ελέγχου, όπως interrupt controllers, global reset control και τη λογική DFx.

- **AI Engine - PL interface tile:** Περιέχει μια μονάδα PL που φιλοξενεί ένα memory-mapped AXI4 και έναν διακόπτη AXI4-Stream, μια stream διεπαφή μεταξύ AIE και PL και μια μονάδα ελέγχου, trace και debug. Παρέχονταιεπίσης ασύγχρονες FIFO για τη διαχείριση του clock domain crossing.

- **AI Engine - NoC interface tile:** Περιέχει μια μονάδα PL όπως περιγράφεται παραπάνω και επίσης μια μονάδα NoC με διασυνδέσεις προς την master μονάδα NoC (NMU) και την slave μονάδα NoC (NSU). Επειδή το NMU και το NSU βρίσκονται σε διαφορετικό τομέα ισχύος από τα AIE απαιτείται level shifting.



Σχήμα 9: Η τοπολογία των PL και NoC Interface Tile

Η προγραμματιζόμενη λογική (PL)

Η προγραμματιζόμενη λογική (PL) είναι το 'FPGA' μέρος των Versal ACAP, για την αποτελεσματική υλοποίηση οποιασδήποτε λειτουργίας ειδικού σκοπού στο hardware. Το Versal PL περιέχει μια εντελώς επανασχεδιασμένη γενιά απο configurable logic blocks (CLBs) με τέσσερις φορές περισσότερη λογική χωρητικότητα, δηλαδή 32LUTs/64 slice flip-flop σε αντίθεση με τα 8 LUTs/16 slice flip-flop στην προηγούμενη γενιά των UltraScale συσκευών. Επίσης διαθέτει μια πληθώρα εσωτερικών μνημών (CLB 64-bit RAM, Block RAMs, UltraRAMs), μηχανές DSP και διεπαφές με οποιοδήποτε άλλο ενσωματωμένο υλικό της πλακέτας, όπως το AIE και το σύστημα επεξεργασίας (PS).

Το υπολογιστικό σύστημα (PS)

Το σύστημα επεξεργασίας (PS) αποτελείται από δυο ARM Cortex επεξεργαστές, τη μονάδα επεξεργασίας εφαρμογών (Application Processing Unit - APU) στον τομέα πλήρους ισχύος (Full Power Domain - FPD) και τη μονάδα επεξεργασίας πραγματικού χρόνου (Real-Time Processing Unit - RPU) στον τομέα χαμηλής κατανάλωσης (Low-Power Domain - LPD) και διάφορα περιφερειακά I/O. Το PS, μαζί με τον ελεγκτή διαχείρισης πλατφόρμας (Platform Management Controller - PMC) και το Coherent PCIe Unit (CPM) ομαδοποιούνται στον Control, Interface and Processing System (CIPS) IP πυρήνα.

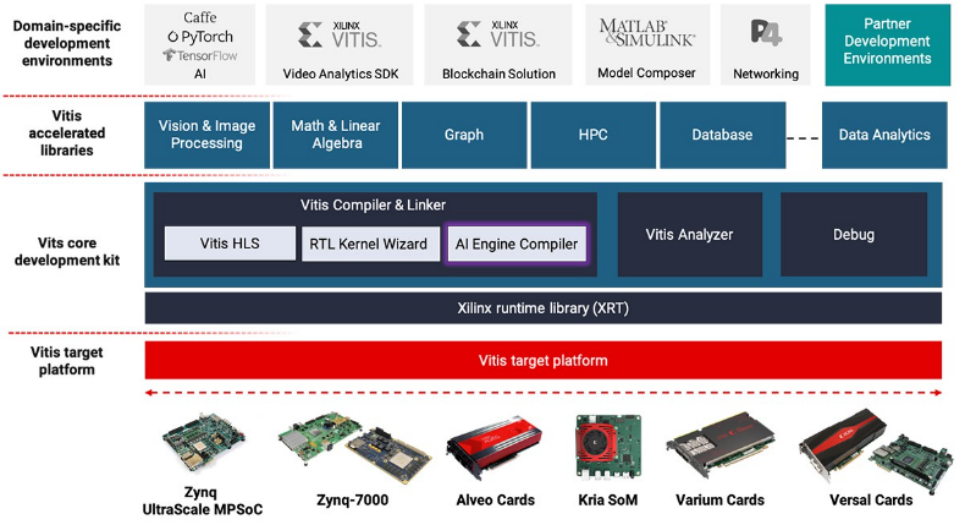
Το Network-on-Chip (NoC)

Το Network-on-Chip (NoC) εμπλουτίζει την παραδοσιακή διασύνδεση μέσω του PL fabric και επιτρέπει επικοινωνία υψηλής ταχύτητας σε επίπεδο συστήματος μεταξύ των πολυάριθμων ετερογενών χαρακτηριστικών, συμπεριλαμβανομένων των AIE, του PS, του PL και του ελεγκτή μνήμης DDR. Μπορεί να διαμορφωθεί χρησιμοποιώντας τα AXI3, AXI4 ή AXI4-Stream για τις διεπαφές master και slave και εκτείνεται τόσο σε οριζόντιες όσο και σε κάθετες κατευθύνσεις μέχρι τις άκρες της πλακέτας. Η διαμόρφωση ή ο προγραμματισμός του NoC γίνεται κατά την εκκίνηση μέσω της διεπαφής προγραμματισμού NoC (NPI) και είναι πλήρως αυτοματοποιημένος και εκτελούμενος από τον PMC.

Εργαλεία

Vitis Unified Software Platform

Το λογισμικό Vitis Unified Software Platform είναι ένα εργαλείο που συνδυάζει όλες τις πτυχές της ανάπτυξης λογισμικού για προϊόντα της Xilinx σε ένα ολοκληρωμένο, ενοποιημένο περιβάλλον για την επιτάχυνση edge, cloud και hybrid εφαρμογών. Επιτρέπει την ανάπτυξη ενσωματωμένου λογισμικού και επιταχυνόμενων εφαρμογών σε ετερογενείς πλατφόρμες της Xilinx, συμπεριλαμβανομένων των FPGA, SoC και Versal ACAP. Το Vitis προσφέρει μια software-centric προσέγγιση για την ανάπτυξη τόσο υλικού όσο και λογισμικού παρέχοντας στον χρήστη τη δυνατότητα να επιλέξει το επίπεδο αφαίρεσης που χρειάζεται [12].

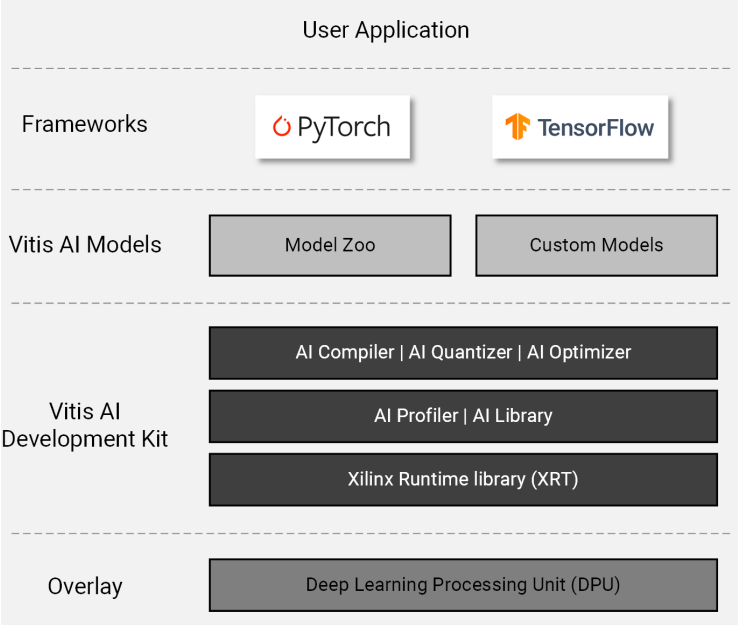


Σχήμα 10: Η στοίβα του Vitis Unified Software Platform [7].

Το Vitis ενσωματώνει επίσης στην ροή ανάπτυξης accelerated εφαρμογών και το Vitis High-Level Synthesis (HLS), ένα εργαλείο σύνθεσης υψηλού επιπέδου. Το Vitis HLS είναι υπεύθυνο για τη μεταγλώττιση των hardware πυρήνων για το εργαλείο Vitis εκτελώντας σύνθεση υψηλού επιπέδου. Αυτοί οι πυρήνες θα επιταχυνθούν στην περιοχή του PL των συσκευών της Xilinx οι οποίοι μπορούν να γραφτούν σε C/C++ και OpenCL.

Vitis AI

Το Vitis AI είναι μια πλατφόρμα ανάπτυξης εφαρμογών σχεδιασμένη από τη Xilinx για να προσφέρει εύκολα hardware-accelerated AI inference σε Xilinx συσκευές, συμπεριλαμβανομένων και των συσκευών edge αλλά και των data center καρτών επιτάχυνσης Alveo. Αποτελείται από βελτιστοποιημένα IP, τις deep-learning επεξεργαστικές μονάδες (DPU), εργαλεία, βιβλιοθήκες, frameworks, έτοιμα AI μοντέλα και πολλά παραδείγματα. Η βασική του ιδέα είναι να παρέχει υψηλή απόδοση και ευκολία στη χρήση, αξιοποιώντας στο μέγιστο τις δυνατότητες επιτάχυνσης εφαρμογών AI σε Xilinx FPGA και ACAP συσκευές. Προς αυτή την κατεύθυνση, η στοίβα του Vitis AI υποστηρίζει mainstream AI frameworks όπως το PyTorch και το TensorFlow και μια εύχρηστη εργαλειοθήκη:



Σχήμα 11: Η στοίβα του Vitis AI [8]

Προτεινόμενη Εφαρμογή και Υλοποίηση

Σχεδιασμός του Συστήματος και Προετοιμασία του δικτύου

Όπως περιγράφεται στην Ενότητα , το μοντέλο ESPCN χρησιμοποιεί τρεις τύπους επιπέδων: L Συνελικτικά επίπεδα, $L-1$ Tanh επίπεδα ενεργοποίησης και 1 επίπεδο PixelShuffle. Για την παρούσα διατριβή ακολουθήθηκε η παραδοσιακή προσέγγιση του ESPCN 3 επιπέδων, οπότε ορίσαμε $L = 3$. Η υλοποίηση του μοντέλου στο Versal SoC θα πρέπει να γίνει σύμφωνα με τις υπολογιστικές ανάγκες του κάθε επιπέδου αλλά και τις υπολογιστικές δυνατότητες του διαθέσιμου υλικού.

Είναι περιττό να δείξουμε ότι η λειτουργία της συνέλιξης είναι η πιο απαιτητική υπολογιστικά και ότι είναι αυτή που χρειάζεται την πιο ισχυρή επιτάχυνση μέσω του υλικού. Για το λόγο αυτό και προκειμένου να αξιοποιήσουμε τις υπολογιστικές δυνατότητες των καινοτόμων AI Engines (AIE), θα υλοποιήσουμε τα επίπεδα Conv του μοντέλου στο AIE τμήμα της πλακέτας. Με την κατάλληλη αναδιάταξη των δεδομένων, η συνέλιξη μπορεί να μετατραπεί σε πολλαπλασιασμό πινάκων και να βελτιστοποιηθεί ώστε να εφαρμοστεί αποτελεσματικά στο AIE array.

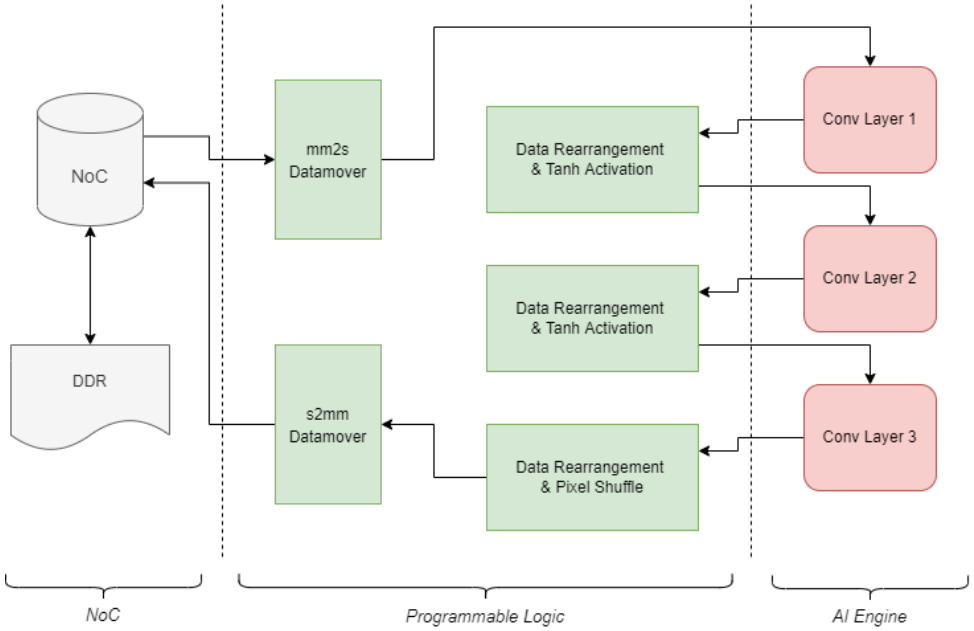
Δύο επιλογές εξετάστηκαν για την υλοποίηση της συνάρτησης ενεργοποίησης - Tanh: α) μια διανυσματική υλοποίηση στο AIE χρησιμοποιώντας την πολυωνυμική προσέγγιση:

$$tanh(x) = \frac{x(27 + x^2)}{(27 + 9x^2)}$$

και β) μια υλοποίηση με πίνακα αναζήτησης (look-up table - LUT) που βασίζεται στο τμήμα της προγραμματιζόμενης λογικής της συσκευής (PL). Η αξιολόγηση και των δύο υλοποιήσεων έδειξε ότι οι υπολογιστικές ανάγκες του πολυωνύμου δεν συγκαλύφθηκαν από τη διανυσματοποίηση των πράξεων στα AIE. Επομένως, η υλοποίηση με LUT ξεπέρασε την κατά προσέγγιση υλοποίηση τόσο στην ακρίβεια, όσο και στην ταχύτητα. Επίσης, η συνάρτηση PixelShuffle και η αναδιάταξη των δεδομένων που απαιτείται για τη μετατροπή της συνέλιξης σε πολλαπλασιασμό πινάκων περιλαμβάνουν scalar byte λειτουργίες και αλληλεπιδρούν με read/write μνήμες. Επομένως και αυτό το σύνολο λειτουργιών είναι

κατάλληλο για εφαρμογή στο PL και όχι στο AIE array. Τέλος, απαιτείται η υλοποίηση πυρήνων μεταφοράς δεδομένων στο PL για την επικοινωνία με την DDR μέσω του NoC.

Η προτεινόμενη αντιστοίχιση του δικτύου στη Versal συσκευή και ο σχεδιασμός του συστήματος παρουσιάζονται στο ακόλουθο μπλοκ διάγραμμα:



Σχήμα 12: Μπλοκ διάγραμμα του προτεινόμενου συστήματος για την HW-specific υλοποίηση.

Η software υλοποίηση του μοντέλου ESPCN, δηλαδή οι κώδικες για την εκπαίδευση και την αξιολόγηση παρέχονται από τα επίσημα Github repositories της PyTorch [17]. Το σύνολο δεδομένων που χρησιμοποιήθηκε για την εκπαίδευση ήταν το BSD300, το οποίο περιέχει 300 εικόνες ανθρώπινων υποκειμένων. Οι εικόνες χωρίζονται σε ένα σετ εκπαίδευσης 200 εικόνων και σε ένα δοκιμαστικό σετ 100 εικόνων. Το μοντέλο, και στην εκπαίδευση και στην αξιολόγηση, λειτουργεί με το κανάλι φωτεινότητας της αναπαράστασης YCbCr, δηλαδή το κανάλι Y.

Χρησιμοποιώντας τον κώδικα εκπαίδευσης, εκπαιδεύσαμε το δίκτυο μέσα σε ένα Google Colaboratory με μια NVIDIA Tesla T4 GPU. Για αυτή τη διατριβή, λαμβάνεται υπόψη ένας upscale factor ίσος με 2, για εικόνες εισόδου μεγέθους 16×16. Επίσης, η εκπαίδευση έγινε για 350 εποχές με ποσοστό μάθησης ίσο με 0,001. Μετά την εκπαίδευση, τα βάρη και οι πολώσεις εξήχθησαν χρησιμοποιώντας την διαδικτυακή εφαρμογή Netron [18].

Hardware-Specific Custom Υλοποίηση

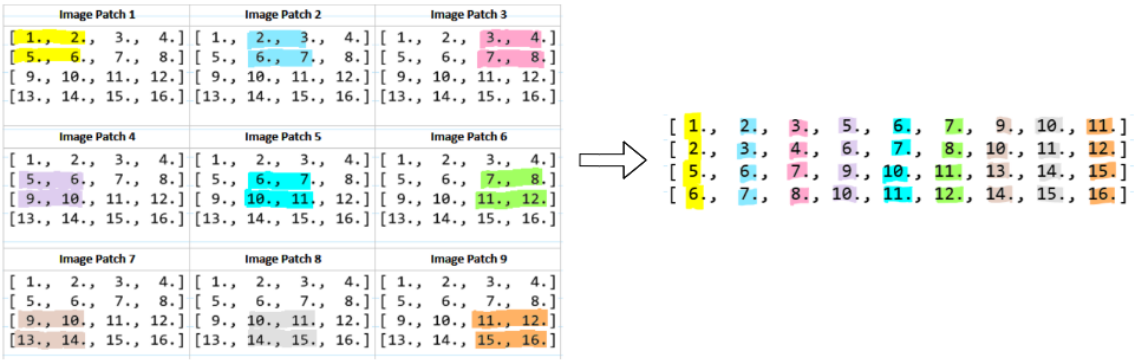
AI Engines: Το Συνελικτικό Επίπεδο

Όπως αναφέρθηκε και στην προηγούμενη υποενότητα, η λειτουργία της συνέλιξης είναι αυτή που θα αντιστοιχιστεί στην περιοχή των AIE για επιτάχυνση. Σύμφωνα και με το παράδειγμα του LeNet που παρέχεται από τη Xilinx [19], η λειτουργία της συνέλιξης αποφασίζεται να μετατραπεί σε πολλαπλασιασμού πινάκων (MMul) για περαιτέρω αξιοποίηση των δυνατοτήτων των AIE. Ο προτεινόμενος σχεδιασμός χρησιμοποιεί επίσης ένα σχήμα διπλού tiling των πινάκων για την εξαγωγή του μέγιστου παραλληλισμού μέσω των AIE array και του AIE API. Το AIE API είναι μια φορητή προγραμματιστική διεπαφή, που υλοποιείται ως C++ header-only βιβλιοθήκες και παρέχει τύπους και λειτουργίες που

μεταφράζονται αποτελεσματικά σε low-level εντολές (intrinsics) και αφαιρέσεις υψηλότερου επιπέδου, όπως iterators και πολυδιάστατους πίνακες [20].

Η μέθοδος Im2Col

Το Im2Col σημαίνει Image to Column και είναι μια τεχνική που μετατρέπει τη λειτουργία της συνέλιξης σε λειτουργία MMul. Ας υποθέσουμε ότι έχουμε έναν όγκο εικόνας εισόδου $1 \times 4 \times 4$ και έναν συνελικτικό πυρήνα μεγέθους 2×2. Ο πίνακας της εικόνας εισόδου θα μετατραπεί σε έναν πίνακα οπού οι στήλες θα αποτελούν patches της εικόνας εισόδου, σύμφωνα με τις διαστάσεις του φίλτρου της συνέλιξης, που εν προκειμένω είναι 2×2, το βήμα της συνέλιξης και το πιθανό padding, όπως φαίνεται στο παρακάτω σχήμα:

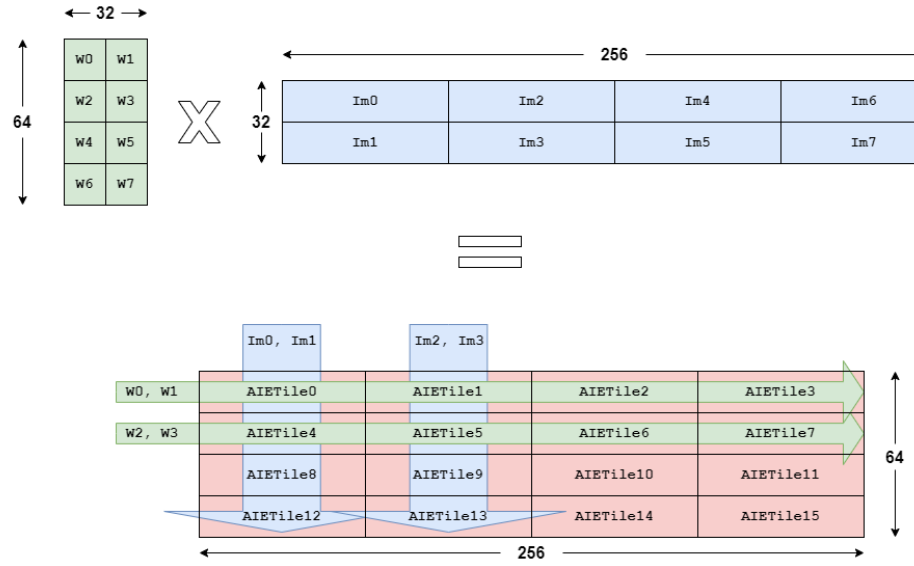


Σχήμα 13: Η μετατροπή της εικόνας εισόδου σε έναν πίνακα από patches [9].

Το αποτέλεσμα είναι ένας πίνακας από patches διαστάσεων 4×9. Επίσης, το 2×2 φίλρο θα μετασχηματιστεί σε έναν πίνακα 1×4 σύμφωνα με την τεχνική im2col και τελικά η λειτουργία της συνέλιξης μπορεί να υλοποιηθεί ως ενας $1 \times 4 \times 9$ MMul.

Προτεινόμενο σχήμα Tiling

Ας πάρουμε για παράδειγμα το πρώτο επίπεδο Conv του δικτύου. Εργαζόμαστε με εικόνες μεγέθους 16×16, διατηρώντας μόνο το κανάλι Y της YCrCb αναπαράστασής τους, όπως περιγράφεται στην ενότητα . Έτσι, μια εικόνα εισόδου μεγέθους $1 \times 16 \times 16$ με πυρήνα συνέλιξης 5×5, 64 κανάλια εξόδου, padding 2×2 και stride 1×1 παράγει έναν πίνακα εισόδου μεγέθους 25×256 σύμφωνα με τη μέθοδο Im2Col. Στη συνέχεια, ο πίνακας των βαρών έχει μέγεθος 64×25. Εκτελούμε zero-padding και για τους δύο πίνακες και έχουμε τον ακόλουθο $64 \times 32 \times 256$ MMul:



Σχήμα 14: Προτεινόμενο σχήμα Tiling για το πρώτο συνελικτικό επίπεδο.

Όπως μπορούμε να δούμε στο σχήμα 14, οι πίνακες χωρίζονται σε tiles μεγέθους 16×16 και 16×64 για τα βάρη και την είσοδο αντίστοιχα. Στη συνέχεια, αντιστοιχίζουμε κάθε tile του πίνακα εξόδου σε ένα AIE tile. Έτσι, για το πρώτο στρώμα χρειαζόμαστε 16 AIE tiles που θα εκτελούν από έναν $16 \times 16 \times 64$ MMul.

Για την υλοποίηση του MMul χρησιμοποιούμε το class template aie::mmul που παρέχεται από το AIE API. Αυτό το template παραμετροποιείται ανάλογα με το σχήμα, $M \times K \times N$, του MMul, τους τύπους δεδομένων και, προαιρετικά, την απαιτούμενη ακρίβεια για το accumulation. Για το σχέδιό μας, χρησιμοποιούνται floating-point αριθμοί, επομένως επιλέγουμε σχήμα MMul από τα επιτρεπόμενα ίσο με $4 \times 2 \times 4$.

Επομένως, έχουμε 16 AIE tiles, καθένα από τα οποία εκτελεί ένα $16 \times 16 \times 64$ MMul σε μπλοκ των $4 \times 2 \times 4$. Αυτό σημαίνει ότι χρησιμοποιήσαμε ένα σχήμα διπλού Tiling: Inter-Tiling με επιμερισμό του υπολογισμού σε πολλαπλά AIE tiles και Intra-Tiling εκτελώντας blocked MMul μέσα σε κάθε AIE tile. Έτσι, συνολικά, το μέγεθος του MMul, το επιλεγμένο σχήμα tiling και ο αριθμός των AIE tiles που απαιτούνται σε κάθε στρώμα παρουσιάζονται στον ακόλουθο πίνακα:

Πίνακας 1: Σχήμα Tiling για κάθε Conv στρώμα

Στρώμα	**Διαστάσεις MMul**	**Διαστάσεις Tiling**	**Πλήθος AIE Tiles**
Conv1	$64 \times 32 \times 256$	$16 \times 16 \times 64$	16
Conv2	$32 \times 576 \times 256$	$8 \times 576 \times 8$	128
Conv3	$4 \times 288 \times 256$	$4 \times 288 \times 16$	16

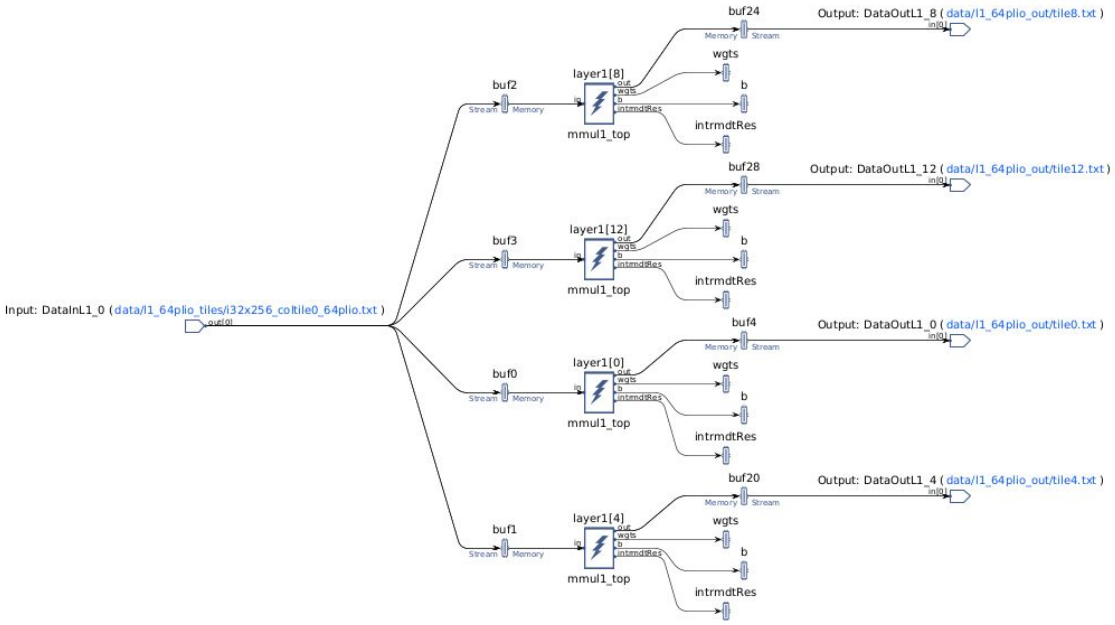
Προγραμματισμός των AI Engines

Ο προγραμματισμός των AI Engine απαιτεί δύο τύπους αρχείων κώδικα:

- Τους **kernels** (πχ kernel.cc), που είναι οι κύριες συναρτήσεις υπολογισμού που υλοποιούν την λογική της εφαρμογής. Είναι γραμμένοι στην γλώσσα προγραμματισμού C++, επιστρέφουν void και χρησιμοποιούν εξειδικευμένα intrinsic calls που στοχεύουν τον διανυσματικό επεξεργαστή VLIW και λειτουργούν σε ροές δεδομένων. Εν προκειμένω, εδώ υλοποιούμε την λογική του

MMul μέσω των συναρτήσεων που παρέχει το AIE API, χρησιμοποιώντας μια window-based επικοινωνία για την είσοδο και έξοδο των δεδομένων, δηλαδή ο πυρήνας βλέπει δεδομένα εισόδου και παράγει δεδομένα εξόδου σε μπλοκ συγκεκριμένου μεγέθους. Επιπλέον, χρησιμοποιούμε το C++ Kernel Class Support που υποστηρίζει η πλατφόρμα ώστε να αντιστοιχίσουμε κάθε kernel object με το απαραίτητο σύνολο βαρών και πολώσεων. Η δήλωση των κλάσεων και των kernel functions τοποθετείται σε ένα ξεχωριστό header-file (πχ kernel.h).

- Τον **adaptable data flow γράφο** (ADF) ο οποίος απαιτεί ένα top-level application αρχείο (πχ graph.cpp) και ένα ξεχωριστό header-file (πχ graph.h). Ένας ADF γράφος αποτελείται από κόμβους και ακμές όπου, οι κόμβοι είναι τα kernel functions που εκτελούν τους υπολογισμούς και οι ακμές αντιπροσωπεύουν τις συνδέσεις δεδομένων μεταξύ τους. Το header-file πρέπει να περιέχει την κεφαλίδα της βιβλιοθήκης ADF (adf.h) και τις δηλώσεις των kernel functions (kernel.h). Χρησιμοποιώντας την κλάση adf::graph ορίζουμε το γράφο της εφαρμογής, δηλώνουμε τους κόμβους/πυρήνες και τις εισόδους/εξόδους του γράφου. Στο παρόν σύστημα δεν υπάρχει επικοινωνία μεταξύ των κόμβων οπότε πρόκειται ουσιαστικά και σύμφωνα με τον πίνακα 1 για $16 + 128 + 16$ AIE tiles που αλληλεπιδρούν με το PL μέσω των πλιο θυρών εισόδου/εξόδου:



Σχήμα 15: Παράδειγμα AIE πυρήνων από το 1ο στρώμα και οι συνδέσεις τους προς το PL

Στη συνέχεια ρυθμίζουμε μια πληθώρα παραμέτρων όπως τα ορίσματα των kernel functions, τις συνδέσεις με τις εισόδους/εξόδους και το μέγεθος των αντίστοιχων παραθύρων, το ποσοστό χρησιμοποίησης ενός AIE από ένα kernel function (runtime ratio) και επιπρόσθετα μπορούμε να προσδιορίσουμε τη θέση στο AIE array των πυρήνων αλλά και των διάφορων buffer και παραμετρών που χρησιμοποιούν. Καθορίζοντας την ακριβή θέση κάθε πυρήνα και των αντίστοιχων buffer ή παραμέτρων του, μπορούμε να ελέγξουμε και να καθοδηγήσουμε τον AIE mapper για να βρει μια εφικτή λύση για τη σχεδίαση. Αυτό είναι εξαιρετικά χρήσιμο για τη δημιουργία μεγάλων γραφημάτων, όπως το δικό μας, γιατί χωρίς αυτούς τους περιορισμούς, ο mapper μπορεί να αποτύχει να βρει μια εφικτή λύση. Κάτι που πράγματι ισχύει και για το παρόν design. Τέλος, το αρχείο της top-level εφαρμογής περιέχει ένα instance της κλάσης του ADF γράφου και μία main. Η main αυτή αποτελεί ουσιαστικά τον driver του γράφου. Χρησιμοποιείται για

τη φόρτωση, την προετοιμασία, την εκτέλεση, την αναμονή και τον τερματισμό του γραφήματος χρησιμοποιώντας συγκεκριμένες κλήσεις μεθόδων που ορίζονται από το API του γραφήματος.

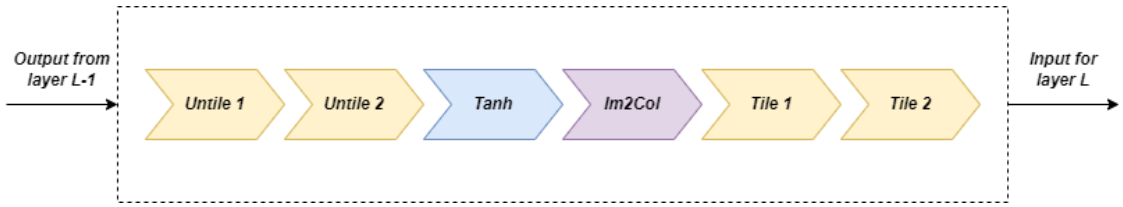
Οι πυρήνες του PL

Datamovers

Οι Datamovers αποτελούνται από δύο πυρήνες στο PL: τον πυρήνα mm2s και τον πυρήνα s2mm. Ο πυρήνας mm2s λαμβάνει τις εικόνες εισόδου από τη μνήμη DDR μέσω του NoC και τις στέλνει κατάλληλα στα AIE tiles του πρώτου Conv επιπέδου μέσω των plio θυρών εισόδου τους. Ο πυρήνας s2mm λαμβάνει τις εικόνες εξόδου από το επίπεδο PixelShuffle και μεταδίδει τα δεδομένα στη μνήμη DDR μέσω του NoC. Η διεπαφή για τις συνδέσεις με τη μνήμη DDR είναι μια memory-mapped AXI4 διεπαφή και η διεπαφή μεταξύ των πυρήνων PL και των AIE είναι μια διεπαφή AXI4-Stream, συμβατή με τη διεπαφή του AIE array.

Επίπεδο Ενεργοποίησης

Αυτό το επίπεδο περιέχει τη λειτουργία της συνάρτησης ενεργοποίησης Tanh και όλη την απαιτούμενη λογική αναδιάταξης των δεδομένων για να μεταβούμε από το ένα Conv επίπεδο στο άλλο. Πιο συγκεκριμένα, όπως περιγράφει η ενότητα , έχουμε ένα σχήμα διπλού tiling για το MMul. Επίσης, απαιτείται η τεχνική Im2Col όπως εξηγήσαμε παραπάνω. Ωστόσο, για τη σωστή χρήση της, οι πίνακες δεν πρέπει να είναι tiled. Επομένως, ένα επίπεδο ενεργοποίησης μεταξύ δύο επιπέδων Conv, υλοποιεί το pipeline που απεικονίζεται στο παρακάτω μπλοκ διάγραμμα:



Σχήμα 16: Μπλοκ διάγραμμα επιπέδου ενεργοποίησης.

Το Στάδιο του Hardware Linking

Η μεταγλώττιση του AI Engine γράφου καταλήγει στο libadf.a binary ενώ οι C/C++ πυρήνες του PL μεταγλωττίζονται σε Xilinx Object (XO) αρχεία. Μετά από αυτό, ο linker του Vitis έχει σειρά. Χρησιμοποιώντας την εντολή σύνδεσης v++ τα προαναφερθέντα μεταγλωττισμένα δυαδικά αρχεία συνδέονται με την πλατφόρμα για τη δημιουργία του αρχείου πλατφόρμας (XSA), που χρησιμοποιείται για το packaging του συστήματος. Όταν το σύστημα περιέχει και AIE γράφο, ο linker του Vitis απαιτεί κάποιες οδηγίες σχετικά με τον τρόπο σύνδεσης των PL πυρήνων στον AIE γράφο. Για αυτόν τον λόγο, παρέχεται και ένα configuration αρχείο με την ενότητα [connectivity]. Ακόμα, εκεί ορίζονται και οι συχνότητες στις οποίες θα λειτουργήσουν οι πυρήνες του PL.

PS: Η Host Εφαρμογή

Το σύστημά μας χρησιμοποιεί το ενσωματωμένο σύστημα επεξεργασίας (PS) ως εξωτερικό ελεγκτή για να ενορχηστρώσει τις κινήσεις δεδομένων μεταξύ του AIE γράφου και των PL πυρήνων. Η PS host εφαρμογή είναι γραμμένη σε C/C++, χρησιμοποιώντας κλήσεις API για τον έλεγχο της αρχικοποίησης, εκτέλεσης και τερματισμού του AIE γράφου και των PL πυρήνων. Στα λειτουργικά

συστήματα Linux, το ADF API ελέγχει τον AIE γράφο ενώ το Xilinx Runtime (XRT) API χρησιμοποιείται για τον έλεγχο PL πυρήνων. Ωστόσο, η Xilinx παρέχει ένα OpenSource XRT C/C++ API που μπορεί επίσης να χρησιμοποιηθεί για τον έλεγχο της εκτέλεσης του AIE γράφου κατά τον προγραμματισμό τηςLinux και θα το χρησιμοποιήσουμε και εμείς.

Έτσι, η host PS εφαρμογή (host_xrt.cpp) κάνει τα εξής:

- Περιλαμβάνει όλα τα απαραίτητα αρχεία κεφαλίδας για τη ροή ελέγχου XRT C API.
- Περιλαμβάνει το αρχείο graph.h με την δήλωση της κλάσης του AIE γράφου και δημιουργεί το αντίστοιχο αντικείμενο.
- Ανοίγει τη συσκευή και φορτώνει το δυαδικό αρχείο XCLBIN.
- Εκχωρεί buffers για τα δεδομένα εισόδου και τα αποτελέσματα στην καθολική μνήμη και αρχικοποιεί τα δεδομένα εισόδου.
- Ανοίγει τους PL datamovers και λαμβάνει τα αντίστοιχα kernel handles. Στη συνέχεια, αποκτά και τα αντίστοιχα run handles, ορίζει τα ορίσματα των πυρήνων και τους ξεκινά.
- Ανοίγει το γράφο, αποκτάει τα graph handles, επαναφέρει και εκτελεί το γράφο.
- Περιμένει να ολοκληρωθεί η εκτέλεση των datamovers και κλείνει τα run και kernel handles.
- Επαληθεύει τα αποτελέσματα εξόδου και απελευθερώνει τους κατανεμημένους πόρους.

Hardware-Agnostic Vitis AI Υλοποίηση

Το μοντέλο ESPCN υλοποιήθηκε και κάνοντας χρήση του λογισμικό Vitis AI, το οποίο μπορεί να μεταγλωττίσει αυτόματα πολλά διαφορετικά μοντέλα νευρωνικών δικτύων για εκτέλεση στην DPU της Xilinx. Καθώς η βασική CPU υλοποίηση του δικτύου αναπτύχθηκε στο PyTorch, θα ακολουθήσουμε τη PyTorch ροή ανάπτυξης που προσφέρει το Vitis AI. Το εργαλείο αυτό βρίσκεται σε μια εικόνα docker και τα βήματα που ακολουθήθηκαν είναι τα εξής:

1. **Εκπαίδευση:** Πρώτο βήμα, είναι η εκπαίδευση και η αξιολόγηση του μοντέλου. Εκπαιδεύουμε ξανά το μοντέλο ESPCN με τις ίδιες παραμέτρους όπως η εκπαίδευση του Google Colab που έγινε για την υλοποίηση HW και περιγράφεται παραπάνω. Το μοντέλο εξόδου αποθηκεύεται ως ένα 'f model.pth' αρχείο. Αυτό θα είναι το μοντέλο κινητής υποδιαστολής.

2. **Κβαντοποίηση:** Οι Xilinx DPUs εκτελούν μοντέλα που έχουν τις παραμέτρους τους σε ακέραια μορφή, επομένως, το επόμενο βήμα θα είναι η κβαντοποίηση (quantization) του εκπαιδευμένου μοντέλου κινητής υποδιαστολής. Την διαδικασία αυτή την αναλαμβάνει ο quantizer του Vitis AI, ο οποίος λαμβάνει αυτό το μοντέλο ως είσοδο, εκτελεί προεπεξεργασία και στη συνέχεια κβαντίζει τα βάρη, τις πολώσεις και τις ενεργοποιήσεις στο δεδομένο πλάτος bit. Αυτό το εργαλείο περιλαμβάνεται στο πψτηον πακέτο vai_q_pytorch.

3. **Μεταγλώττιση:** Η Xilinx Intermediate Representation (XIR) είναι μια ενδιάμεση αναπαράσταση βασισμένη σε γράφους AI αλγορίθμων, η οποία έχει σχεδιαστεί για μεταγλώττιση και αποτελεσματική ανάπτυξη στη DPU. Ο μεταγλωττιστής Vitis AI, που βασίζεται στην XIR (vai_c_xir), παίρνει το κβαντισμένο μοντέλο ως είσοδο και αφού το μετατρέψει σε XIR μορφή, εφαρμόζει διάφορες βελτιστοποιήσεις. Στη συνέχεια, χωρίζει το γράφο σε πολλούς υπογράφους με βάση το εάν η λειτουργία μπορεί να εκτελεστεί στη DPU ή όχι. Για τους υπογράφους DPU,

ο μεταγλωττιστής δημιουργεί τη ροή εντολών και προσαρτάται σε αυτήν. Τέλος, ο βελτιστοποιημένος γράφος με τις απαραίτητες πληροφορίες και οδηγίες για το Vitis AI Runtime (VART) συγκεντρώνεται σε ένα μεταγλωττισμένο αρχείο XMODEL.

4. **Εκτέλεση:**Για να τρέξουμε την εφαρμογή στην target πλακέτα, VCK190, πρέπει να αντιγράψουμε εκεί τις εικόνες εισόδου, το XMODEL και τον κώδικα της εφαρμογής. Ο κώδικας της εφαρμογής είναι γραμμένος σε Python και υλοποιεί τις ακόλουθες λειτουργίες:

   - Ανοίγει το XMODEL, το κάνει deserialize σε ένα XIR graph object και λαμβάνει μια λίστα με τα υπογραφήματα.
   - Δημιουργεί τα αντικείμενα DPURunner που ελέγχουν την εκτέλεση των υπογράφων που θα εκτελεστούν στη DPU.
   - Προεπεξεργάζεται τις εικόνες εισόδου, δηλαδή μετατρέπει τις εικόνες στην αναπαράσταση YCbCr και διατηρεί μόνο το κανάλι Y.
   - Λαμβάνει τους τανυστές εισόδου και εξόδου από τα αντικείμενα DPURunner και αρχικοποιεί κατάλληλα το μέγεθος του batch που υποστηρίζεται από την εκάστοτε DPU και τους buffer εξόδου.
   - Αρχικοποιεί με μηδενικά τους buffer εισόδου και εκτελεί τους υπογράφους. Σημειώνουμε εδώ ότι η λειτουργία Tanh υλοποιείται σαν μια συνάρτηση χωριστά στον Python κώδικα της εφαρμογής για να εκτελεστεί στην CPU.
   - Επεξεργάζεται εκ των υστέρων την έξοδο και υπολογίζει το επιτευχθέν PSNR.

Αξιολόγηση των Προτεινόμενων Συστημάτων

Ποιότητα Ανακατασκευασμένης Εικόνας

Το μοντέλο ESPCN αξιολογήθηκε χρησιμοποιώντας τη μετρική του PSNR.Για την εργασία του Super Resolution (SR), το PSNR ορίζεται ως εξής:

$$PSNR = 10 \cdot \log_{10}(\frac{L^2}{\frac{1}{N}\sum_{i=1}^{N}(I(i) - \hat{I}(i))^2}),$$

όπου I είναι η ground truth εικόνα με N pixels, $\hat{I}$ η ανακατασκευασμένη εικόνα και L η μέγιστη τιμή pixel. Το σύνολο δεδομένων αξιολόγησης ήταν το σύνολο δεδομένων Set5, αποτελούμενο από 5 εικόνες («baby», «bird», «butterfly», «head», «woman») που χρησιμοποιούνται συνήθως για τη δοκιμή της απόδοσης μοντέλων SR. Ο παρακάτω πίνακας συνοψίζει τα αποτελέσματα PSNR σε dB για τις τρεις διαθέσιμες υλοποιήσεις:

Πίνακας 2: Αποτελέσματα PSNR

Εικόνα	**Υλοποίηση σε CPU (dB)**	**Υλοποίηση σε Vitis AI (dB)**	**Hardware-Specific Υλοποίηση (dB)**
baby	24.58	17.11	24.56
bird	27.62	15.49	27.55
butterfly	22.78	12.42	22.77
head	30.74	14.37	30.61
woman	26.14	15.94	26.10

Όπως μπορούμε να δούμε, η hw-specific εφαρμογή επιτυγχάνει εξαιρετικά αποτελέσματα, πολύ κοντά στο baseline της CPU, με μέγιστη απόκλιση περίπου **0,4%**. Αυτή η απόκλιση οφείλεται στην υλοποίηση της λειτουργίας ενεργοποίησης Tanh βάσει LUT. Η εφαρμογή Vitis AI από την άλλη πλευρά, έχει υπερβολικά κακή απόδοση. Αν και περιμέναμε κάποιες απώλειες στο PSNR λόγω του κβαντισμού, η απόκλιση εδώ είναι τεράστια. Η Xilinx παρέχει μια δήλωση αποποίησης ευθυνών που δηλώνει ότι «Για ορισμένα δίκτυα όπως το Mobilenet, η απώλεια ακρίβειας μπορεί να είναι μεγάλη». Ωστόσο, συνιστώνται κάποιο fine-tuning αλγόριθμοι ή εκπαίδευση με επίγνωση κβαντισμού (Quantization Aware Training) για τη βελτίωση της ακρίβειας των κβαντισμένων μοντέλων.

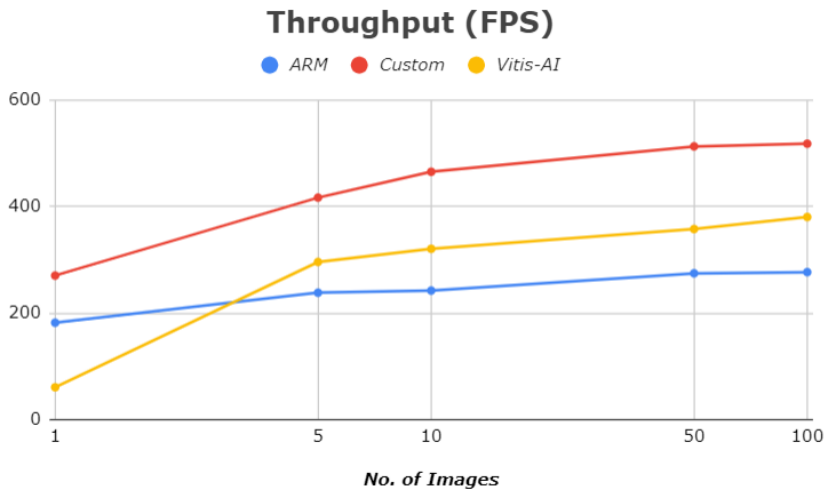
Χρόνοι Εκτέλεσης

Για να συγκρίνουμε την απόδοση των υλοποιήσεων που παρουσιάστηκαν σε αυτή τη διατριβή ως προς την ταχύτητα, λαμβάνουμε υπόψη τον χρόνο του AI inference. Για τη CPU baseline, εκτελούμε το σενάριο αξιολόγησης στον dual-core επεξεργαστή Arm Cortex-A72 που υπάρχει στο PS τμήμα της πλακέτας VKC190. Αρχικά θα ξεκινήσουμε με την μέτρηση του inference time για μία εικόνα ώτε να προσδιορίσουμε την καθυστέρηση (latency) των συστημάτων:

Πίνακας 3: Αποτελέσματα Καθυστέρησης

Υλοποίηση	Καθυστέρηση (ms)	Frames per Second (FPS)
CPU	5,5	181,82
Vitis AI	16,5	60,61
HW-Specific	3,7	270,27

Στη συνέχεια εκτελούμε τις εφαρμογές για περισσότερες από μία εικόνες ώστε να μετρήσουμε τη ρυθμαπόδοση (throughput) των συστημάτων και παίρνουμε το παρακάτω διάγραμμα:



Σχήμα 17: Ρυθμαπόδοση των Προτεινόμενων Υλοποιήσεων

Το πρώτο πράγμα που παρατηρούμε εδώ, είναι ότι η εφαρμογή του Vitis AI έχει σημαντικά χαμηλότερη απόδοση σε σύγκριση τόσο με τη CPU baseline όσο και με την HW-specific υλοποίηση όσον αφορά το λατενςψ του συστήματος. Αυτή είναι μια αναμενόμενη συμπεριφορά λόγω της ασυμβατότητας Tanh με τη DPU. Πιο συγκεκριμένα, για την υλοποίηση του δικτύου η λειτουργία Tanh ανατέθηκε στην CPU. Έτσι, το μπρος-πίσω μεταξύ της DPU και της CPU προσέθεσε επιπλέον κόστος στην

απόδοση του συστήματος. Ωστόσο, κλιμακώνει καλύτερα από τη CPU baseline, χωρίς φυσικά να υπερβαίνει την HW-specific υλοποίηση.

Αντίθετα, η HW-specific υλοποίηση αποδίδει πολύ καλά τόσο από άποψη καθυστέρησης όσο και από άποψη απόδοσης. Κερδίζουμε μια επιτάχυνση περίπου **1,5x** σε σύγκριση με τη CPU baseline και **4,5x** σε σύγκριση με το Vitis AI όταν μιλάμε για καθυστέρηση. Επιπλέον, όπως δείχνει το σχήμα 17, η υλοποίησή αυτή ξεπερνά σε απόδοση τόσο τη CPU όσο και το Vitis AI, φτάνοντας τα **518 FPS**.

Κατανάλωση Ισχύος

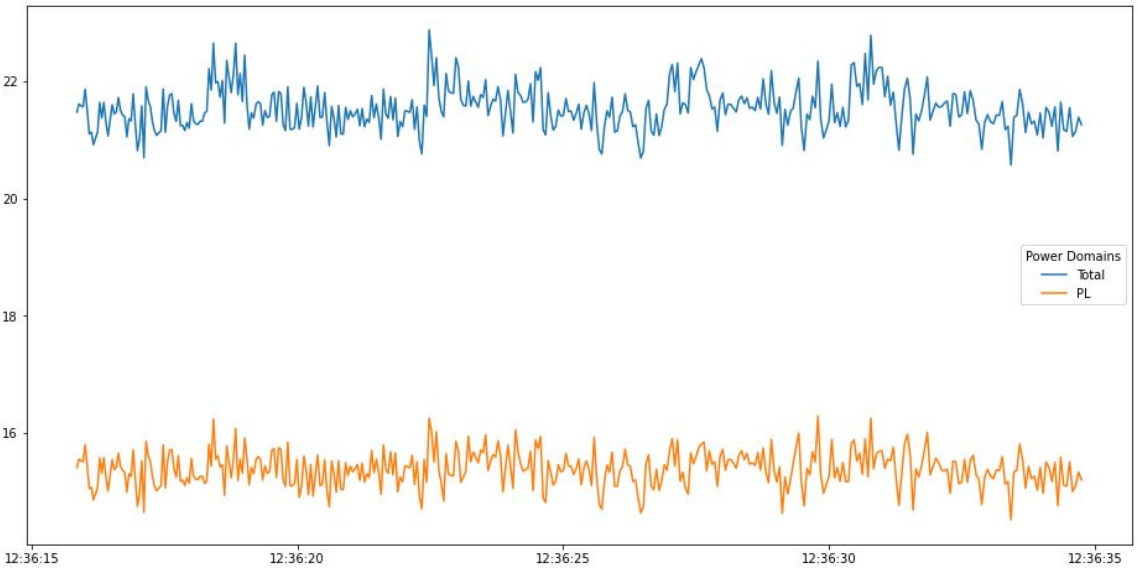
Η απόδοση ισχύος είναι επίσης μια σημαντική πτυχή, ειδικά για edge εφαρμογές. Η Xilinx κυκλοφόρησε ένα power tool που έχει σχεδιαστεί για να αναδείξει τις δυνατότητες ισχύος των Versal ACAP συσκευών. Ονομάζεται Power Advantage Tool (PAT) και αποτελείται από:

1. Την Python βιβλιοθήκη poweradvantage.py που παρέχει ένα φορητό σύστημα μέτρησης ισχύος

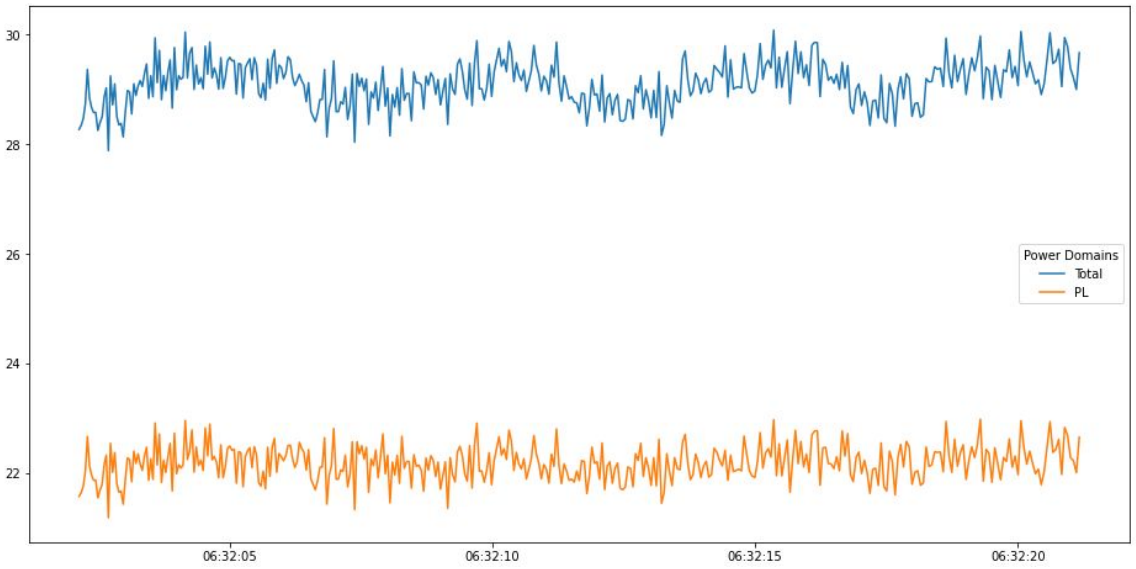
2. Ένα Python Jupyter Notebook με όνομα power_advantage_tool., που παρέχει επεκτάσιμα παραδείγματα χρήσης της βιβλιοθήκης poweradvantage.py.

Το PAT είναι προεγκατεστημένο ως μέρος του εργαλείου Board Evaluation and Management (BEAM). Το BEAM αποτελεί μια διαδικτυακή εφαρμογή GUI συνδεδεμένη με τον ωεβ-σερερ του ελεγκτή συστήματος (System Controller) που επιτρέπει στους χρήστες να παρακολουθούν και να ελέγχουν την πλακέτα λαμβάνοντας μετρήσεις ή τροποποιώντας παραμέτρους όπως ρολόγια, τάσεις ή ισχύ.

Για να ελέγξουμε την κατανάλωση ενέργειας της εφαρμογής μας χρησιμοποιούμε το PAT μέσω του κώδικα του Jupyter Notebook. Μπορούμε να κάνουμε μετρήσεις από διαφορετικούς τομείς ισχύος και νησίδες ισχύος της πλατφόρμας. Επιλέγουμε να μετρήσουμε τη συνολική κατανάλωση ενέργειας για να κάνουμε μια συνολική αξιολόγηση και την κατανάλωση ενέργειας του τομέα ισχύος PL, που περιλαμβάνει την προγραμματιζόμενη λογική, τα PL GT, τη coherent μονάδα PCIe (CPM4) και τα AI Engines. Τροποποιούμε την εφαρμογή ώστε να εκτελείται για 500 εικόνες προκειμένου να έχουμε επαρκή χρόνο εκτέλεσης στα AIE για τη μέτρηση της ισχύος. Τροποποιούμε επίσης τον δεδομένο κώδικα του Notebook για να αυξήσουμε τη συχνότητα της δειγματοληψίας και να σχεδιάσουμε το γράφημα της μετρούμενης ισχύος:



(α′) HW-specific Υλοποίηση



(β′) Vitis AI Υλοποίηση

Σχήμα 18: Μετρήσεις Ισχύος.

Στο διάγραμμα 18α′ μπορούμε να δούμε 4 αιχμές που αντιστοιχούν στις 4 εκτελέσεις της εφαρμογής μας για τη HW-Specific υλοποίηση. Αντίστοιχα, οι 4 εκτελέσεις της εφαρμογής για την Vitis AI υλοποίηση αντιστοιχούν στις 4 καμπύλες που παρατηρούνται στο διάγραμμα 5.3β.

Πρώτα απ ′όλα, μπορούμε να παρατηρήσουμε ότι δεν υπάρχουν τεχνικές διαχείρισης ενέργειας (όπως το clock-gating) από προεπιλογή στην πλακέτα. Και στις δύο περιπτώσεις δεν έχουμε ακραίες αποκλίσεις από την ισχύ της κατάστασης αδράνειας, ειδικά στον τομέα AI Engines (PL domain). Αξιοσημείωτο είναι επίσης το γεγονός ότι όταν υπάρχει κάρτα DPU SD στην πλακέτα η συνολική ισχύς αυξάνεται ελαφρώς.

Chapter 1

Introduction

1.1 State of the Industry

Challenges in the Semiconductor Industry

The rapid scaling of a technology node alone, despite its benefits, often introduces great challenges for the others. Speaking of computer science, this can be witnessed by the great challenges that AI - and especially Deep Learning - revolution has imposed to the semiconductor industry. The algorithmic innovation and the explosion of data drive an exponential increase in computational needs, as the following figure shows.

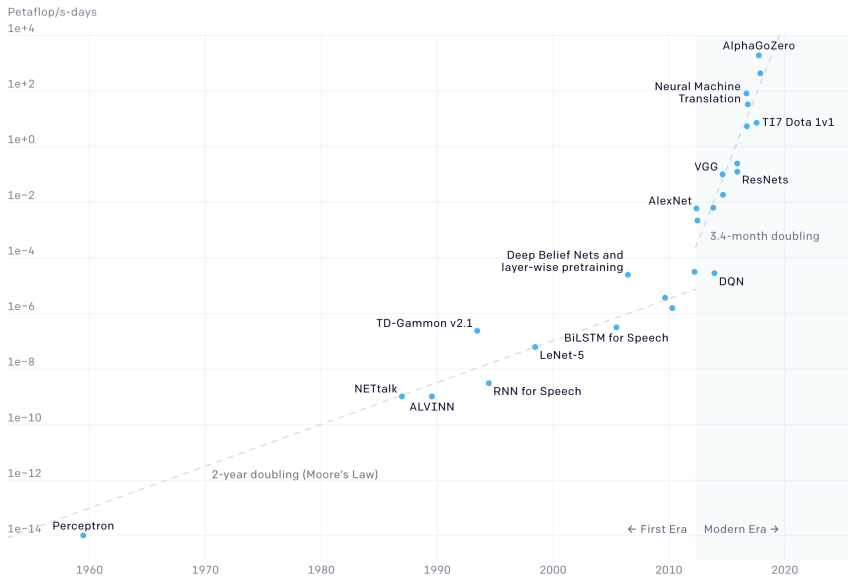


Figure 1.1: The total amount of compute, in petaflop/s-days used to train selected network architectures. [10]

AI training compute demands double every 3.5 months, whereas Moore's Law doubled "only" every 18 months [13].

This demand is further inflamed by the end of the Renaissance period in hardware optimization

that Dennard's scaling and the laws of Moore and Amdahl permitted, forcing the semiconductor industry to seek solutions beyond the stereotypical "one size fits all" CPU, by exploring domain-specific architectures including DSPs, GPUs and FPGAs:

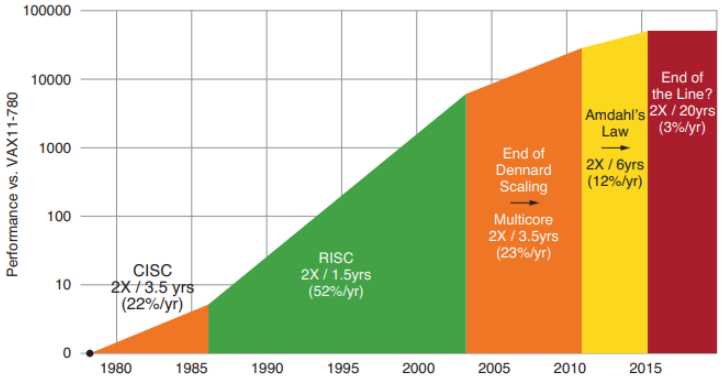


Figure 1.2: 40 Years of Processor Performance vs. Time[11]

Very large vector processing cores (DSPs, GPUs) employ a theoretically tremendous computing power, but their realistic performance is often limited by the huge performance gap between the processors and the memory system. Traditional FPGA solutions on the other hand, provide a programmable memory hierarchy [21] and customization to improve latency, but the requirement for expertise remains, putting a barrier to their high-volume adoption in various application domains and markets.

In this direction, Xilinx is introducing a revolutionary, new, heterogeneous compute architecture, the Adaptive Compute Acceleration Platform (ACAP) that solves the economic, technological, and ease of use challenges mentioned above. Versal is the first generation of ACAP devices that employs numerous standout features, such as the newly introduced AI Engines, to provide leading edge performance. Also, a portfolio of software-abstracted toolchains were also launched to enable the quick development of optimized applications by eliminating the hardware expertise required by the traditional hardware development flow.

1.2 Thesis Motivation and Contribution

Machine learning (ML) and artificial intelligence (AI) govern the majority of today's real-world edge applications. However, these applications suffer from latency limitations imposed by the use of common CPU or GPU solutions. Programmable logic (e.g., FPGAs) provides efficient customization in order to support latency-critical real-time edge applications but the traditional hardware development flow prevents them from being a widely adopted solution as mentioned above. The Versal ACAP portfolio launched by Xilinx, promise a breakthrough ML acceleration and, alongside with its new design tools, the lowest barrier in hardware expertise. Therefore, the primary **motivation** of this thesis was to explore the Versal innovative architecture and also give an insight about the degree in which the aforementioned promises were kept.

More specifically, a custom, hardware-aware implementation is going to be compared with software implementations that either target traditional ARM CPUs or leverage the capabilities of the Vitis AI development platform. Vitis AI is an automated end-to-end toolflow promoted by Xilinx for AI inference tasks. As the use case application, we consider the deep learning image super-resolution (SR) task. SR is the process of increasing the resolution of an image,

with numerous real-world applications. The ESPCN (Real-Time Single Image and Video Super-Resolution Using an Efficient Sub-Pixel Convolutional Neural Network) model was chosen as the network to be implemented, due to its efficiency and performance. The target device will be the VCK190 evaluation kit from the Versal AI Core series.

The **contribution** of this thesis extends in two axes. The first one regards the implementation of an SR model on the VCK190 board with two different approaches: a custom, hardware-specific implementation and one that uses the Vitis AI stack. The custom implementation, gives significant latency and power gains compared both to the Vitis AI and the CPU implementations, without losses on accuracy. On the other hand, the Vitis AI implementation suffers from accuracy losses and performance limitations as not all model's operations are yet supported by the tool. The second axis of contribution lays in the direction of the insights given for these newly introduced architectures and tools. The aforementioned implementations provide a detailed walk-through of the programming process for both cutting-edge heterogeneous hardware and high-level software-specific designs. Along with their performance comparison we can evaluate the ease-of-use versus optimization trade-off that Xilinx aspires to eliminate.

1.3 Theoretical Background

1.3.1 Machine Learning

Machine Learning (ML) is a discipline of Artificial Intelligence (AI) and a prominent part of Computer Science (CS), aiming to provide machines with the ability to learn, imitating the human learning process. More specifically, ML systems employ algorithms that try to derive past experiential knowledge from large volumes of data in order to be able to predict future outcomes autonomously, without explicit programming. Based on the methods of implementing the learning/training process of a ML algorithm, ML is broadly categorized into three main types:

- **Supervised Learning:** In Supervised Learning labeled datasets are used for training, where the desired output for a specific input is provided. Thus, the ML system learns the mapping pattern of the input - output pairs of data and adjusts its parameters to make accurate predictions of unseen, future inputs.

- **Unsupervised Learning:** On the other hand, Unsupervised Learning forces the ML system to predict the output without any supervision, using an unlabeled dataset. This type of learning is ideal for exploration, that is, clustering of data by identifying hidden patterns, similarities or differences.

- **Reinforcement learning:** Reinforcement ML models use a feedback-based learning method. The knowledge is obtained by interacting with the environment. The subject takes actions and gets rewards for the "good" ones and penalties for the "bad" ones. The goal is to maximize its performance. This "reward" does not imply a desired output, but it is only an estimation of how good an action was, so it should not be confused with the supervised learning paradigm.

1.3.2 Artificial Neural Networks

Artificial Neural Networks (ANNs) are computing systems inspired by the human brain and the most popular subset of ML. They act as a parallel and distributed processing system consisting of

multiple, connected, simple computing nodes that have the ability to acquire empirical knowledge from their environment and store it for future use [14]. These computing nodes are named artificial neurons, or simply neurons, and can be perceived as the neurons of the human brain. Mimicking the functionality of the biological brain the knowledge is acquired through a learning process and it is stored at the connections between the neurons - the equivalent of the human synapses.

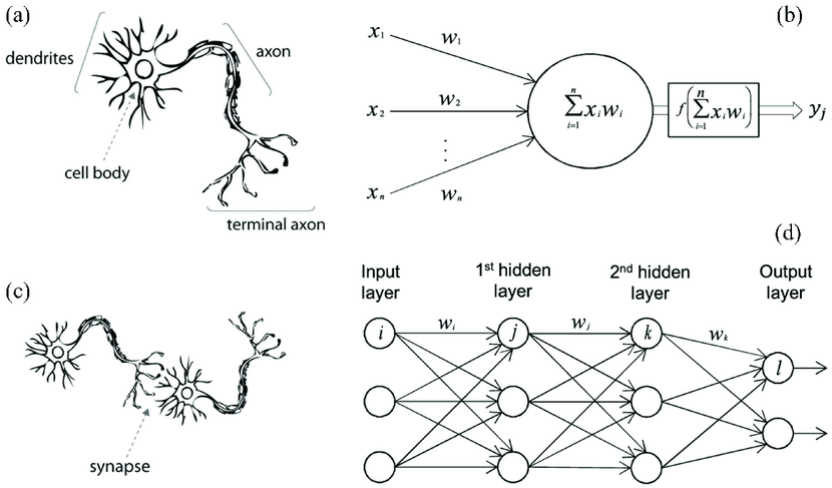


Figure 1.3: A biological neuron in comparison to an artificial neural network: (a) human neuron; (b) artificial neuron; (c) biological synapse; and (d) ANN synapses [1].

Neuron Model

The model of an artificial neuron is the basis of any ANN and resembles the biological one. Its major parts are the synaptic weights, the adder and the activation function. The synaptic weights represent the strength of a connection that reenacts the biological synapses. The input signals of a neuron are multiplied by its synaptic weights and the resulted products are added at its body, the adder, as shown in figure 1.3 (b). Then, the activation function is responsible for bounding the sum. Frequently an external bias is added to the output of the adder in order to increase or decrease the stimulus of the activation function.

Multiple neurons are aggregated into layers to form an ANN topology. Typically, there is an input layer, an output layer and one or more hidden layers. Each node of one layer is connected to a node of another layer, transmitting information between them and throughout the network. A neuron receives a signal, processes it as described above and then signals, or not, neurons connected to it do the same.

Train & Inference

The learning process, or commonly training, of an ANN can be Supervised, Unsupervised or Reinforced in an analogy with the corresponding techniques analyzed in ML Section 1.3.1. In this thesis, the Supervised learning process is employed, so we will hereinafter focus there.

The supervised learning process, employs training on example data to increase the ANN's prediction accuracy over time. The example data have the form of known input-output pairs. First step of training is the so called *forward-step* where based on the weighted sum computing process analyzed before, the prediction takes place. Then, the predicted output is compared to the target output and the prediction error is defined via a *loss function*. This error is propagated

throughout the network using a *back-propagation* algorithm and causes the adjustment of the network's synaptic weights. After a finite number of iterations the prediction becomes highly accurate and the training process stops. The knowledge stored into the network is in fact the adjusted synaptic weights and the feed-forward step for making predictions after training is called inference.

Convolutional Neural Networks (CNNs)

Convolutional Neural Networks (CNN) are a type of ANNs inspired from the animal visual cortex and mainly used for classification and computer vision tasks. CNNs follow the exact same training and inference process described above for the regular ANNs. The difference lays in the fact that the CNNs are aware that an image will be their input which permits certain properties to be encoded into their architecture. These properties then, not only make the implementation of the forward-step more efficient but also lead to extreme reduction of the number of trainable parameters in the network, that is the synaptic weights and biases. [15]

A CNN architecture consists of three main types of layers: Convolutional (Conv) layers, Pooling (Pool) layers and Fully-connected (FC) layers. As a convention the input image itself is called the input layer of a CNN. The other three layers can be stacked interchangeably to formulate different CNN architectures. Let's briefly analyze the CNN layers:

- **Input Layer:** Holds the raw pixel values of the input image. The layer's dimensions match the dimensions of the input image.

- **Conv Layer:** The core building block of a CNN. In its computation involves three major concepts: an input layer as explained before, a filter, also known as kernel and a feature map. The filter is basically the convolution window, a 2-D kernel of weights that maps to a local region of the image according to its dimensions, the so called receptive field. This kernel slides across the input volume from left to right and from top to bottom computing the corresponding dot products. This process, known as a convolution, is used in CNNs for feature extraction and its output is called a feature map. The number and the dimensions of the output feature maps are depended on the size and the number of the convolutional kernels, the stride of the convolution and the possible padding. Finally, each Conv layer applies an elementwise activation function such as ReLU, Sigmoid or Tanh that may be considered as another layer type, the activation layer.

- **Pool Layer:** A form of non-linear down-sampling. Similar to the convolutional kernel discussed above, the pooling layer slides a kernel across the input volume that does not have weights and applies only an aggregation function to the receptive field. The pooling layer contributes to progressively reduce the spatial size of the representation, to reduce the number of trainable parameters, improve efficiency, and thus to limit the risk of over-fitting. There are several non-linear functions to implement pooling, where max pooling is the most common.

- **FC Layer:** In Conv layers described above, neurons are connected only to a part of the input volume, the receptive field. In contrast, each neuron in a FC layer connects directly to a node of the previous level, just like the regular ANNs. This layer performs the final classification based on the features extracted in previous layers. FC layers usually leverage a softmax activation function to produce a probability from 0 to 1 to match the classification needs.

An example CNN architecture with all the aforementioned major building blocks is illustrated in the following image:

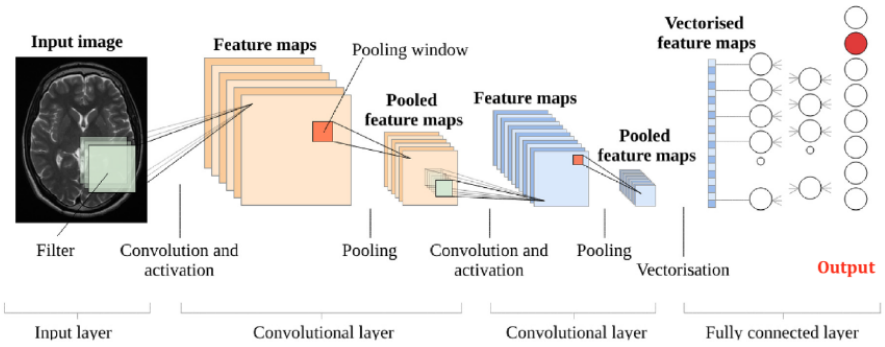


Figure 1.4: CNN Architecture[2].

1.3.3 Image Super-Resolution

Image Super-Resolution (SR) refers to the process of reconstructing high-resolution (HR) images from given low-resolution (LR) ones and it is a task of great importance in the domains of computer vision and image processing. Apart from the academic and research interest it also has numerous real-world applications such as satellite imaging, medical imaging, surveillance and security. However, SR is a rather challenging and ill-posed problem, since there are always multiple HR images corresponding to a single LR image. [22]

In literature, there is plethora of classical SR techniques like the prediction-based or the statistical ones. However, with the outburst of ML, a significant and promising research in deep learning approaches for tackling the challenging SR task has arise, often bringing off state-of-the-art performance. A variety of deep learning architectures have been proposed including Convolutional Neural Networks (CNNs) as well as the present-day Generative Adversarial Nets (GANs).

There is a wide gamut of deep learning SR algorithms as there are major aspects of their architecture that have many different available approaches. To begin with, the majority of SR models focus on supervised methods, i.e., training with both HR and the corresponding LR images that are produced via a predefined degradation on the HR ones. However, the real-world scenarios forced the development of unsupervised SR models too, which only use unpaired LR-HR images for training. In addition, there are different SR frameworks that define where the upsampling should take place in a model and, of course, different upsampling methods either interpolation-based or learning-based ones. Researchers, on top of the SR frameworks, apply different model architectures like residual, recursive or attention learning. Finally, the learning strategies of these models, beside the traditional L2 loss, usually employ pixel-wise, content and adversarial loss functions.

To examine and evaluate the performance of SR models, several image quality assessment methods are used. The most commonly used is the Peak Signal-to-Noise Ratio (PSNR), the most popular reconstruction quality metric of lossy transformations. For the image SR task, PSNR is defined as follows:

$$PSNR = 10 \cdot \log_{10}\left(\frac{L^2}{\frac{1}{N}\sum_{i=1}^{N}(I(i) - \hat{I}(i))^2}\right),$$

where I is the ground truth image with N pixels, $\hat{I}$ the reconstruction and L the maximum pixel value. The denominator of the above fraction is the pixel-level mean squared error (MSE) and takes into account only the pixel differences, that is not exactly accurate with the visual perception

and often leads to poor performance. However, due to the necessity to compare with literature works and the lack of completely accurate perceptual metrics, PSNR is still currently the most widely used evaluation criteria for SR models [22].

1.3.3.1 ESPCN: Real-Time Single Image and Video Super-Resolution Using an Efficient Sub-Pixel Convolutional Neural Network

Overview

The selected SR model for this thesis is the Efficient Sub-Pixel Convolutional Neural Network (ESPCN) [3]. Shi et al. proposed ESPCN to address the limitations of the previous CNN based approaches. Those used the pre-upsampling SR framework, where the LR image is firstly upsampled to a coarse HR image and then deep CNNs are applied to refine the details. The fact that the CNN computations were performed in the higher dimensional space increased the complexity and memory cost of these implementations. Furthermore, the upsampling were based on simple interpolation methods, such as the bicubic one.

In order to address these limitations, ESPCN was proposed introducing an efficient sub-pixel convolutional layer to the CNN architecture. ESPCN uses the post-upsampling SR framework in which the upscaling is done at the end of the network. That means that no interpolation is needed for the input and thus the network is capable of learning a better mapping between the LR-HR image pairs. Also, the reduced input volume requires smaller filter sizes, reduces the complexity of the computations and therefore the network becomes lighter. This enhanced efficiency makes ESPCN an ideal choice for super-resolution tasks even in real-time HD videos.

Design & Architecture

The ESPCN model expects an LR image and an upscale factor as inputs. The LR image is produced by downsampling the HR images of the dataset according to the specified upscale factor. In fact, the actual input of the network is considered to be only the luminance channel in the YCbCr colour space. The output will be the reconstructed super-resolved (SR) image.

The architecture of ESPCN is illustrated in Figure 1.5. For a network composed of L layers, the first $L-1$ are classical Conv layers for feature extraction and the last one is responsible for the mapping to the high dimensional space, according to the upscale factor r. This final layer is the efficient sub-pixel convolutional layer.

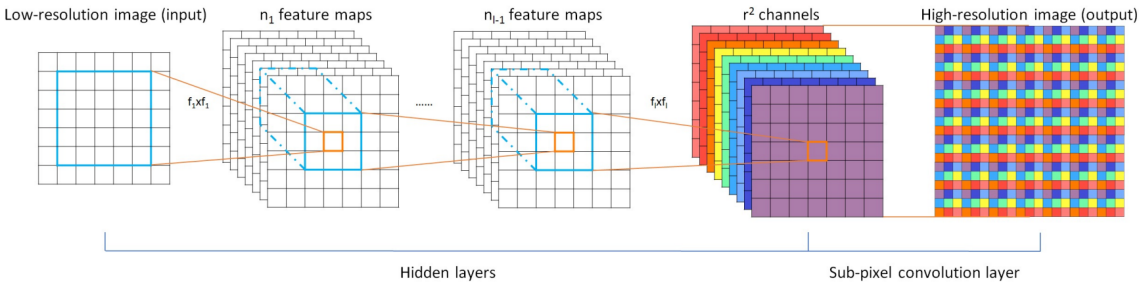


Figure 1.5: ESPCN model architecture[3].

The most common ESPCN design sets $L = 3$ and in detail the layers of the network have the following shapes and parameters:

1. Input layer: Suppose we have an input layer with an image of shape $[B, C, N, N]$.

2. First Conv Layer: 64 filters with kernel size of 5×5, stride of 1×1 and padding 2×2, followed by a tanh activation layer.

3. Second Conv Layer: 32 filters with kernel size of 3×3, stride of 1×1 and padding 1×1, followed by a tanh activation layer.

4. Third Conv Layer: $C \times r \times r$ filters, where r is the upscale factor, with kernel size of 3×3, stride of 1×1 and padding 1×1.

5. PixelShuffle Layer: The sub-pixel shuffle function that gives to the output SR image the desired dimensions, that is, $[B, C, r \times N, r \times N]$.

Sub-pixel Convolution

In the camera imaging system, due to the limitations of the light sensor, images are technically limited to the original pixel resolution. Therefore, real-world objects in an image are spatially quantized at this fixed resolution. However, in the microscopic world there are tiny pixels between two adjacent physical pixels. Those tiny pixels are called sub-pixels and are illustrated in Figure 1.6. Each pixel of the imaging system is in fact each square area defined by four red squares, where the black dots are the sub-pixels. The accuracy of sub-pixels can be adjusted depending on the interpolation between the adjacent pixels. In this way, the mapping from small square areas to big square areas can be implemented through sub-pixel interpolation. [4]

Figure 1.6: Sup-pixels' illustration [4].

Sub-pixel convolution, also known as pixel shuffle, is one of the most notable concepts introduced by Shi et al. It involves a general convolution operation followed by a rearrangement of pixels using the sub-pixel theory. The output channel of the last layer has to be $C \times r \times r$ so that the total number of pixels is consistent with the HR image to be obtained. Then the pixel shuffle function combines each pixel on multiple-channel feature maps into one $r \times r$ square area in the output image. Thus, each pixel on feature maps is equivalent to the sub-pixel on the generated output image:

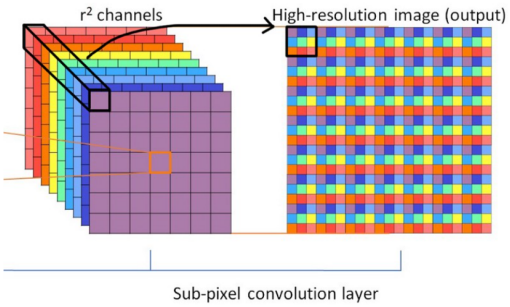


Figure 1.7: Pixel shuffle operation [4].

In the ESPCN network, the interpolation method is implicitly contained in the convolutional layers and it can be learned automatically by the network. This, comes to replace traditional deconvolution operations used for obtaining the HR images, like the bicubic interpolation, improving the network's speed and performance.

1.3.4 High-Level Synthesis

FPGA programming, traditionally, required a design in a hardware description language (HDL) to define its behavior. The most common HDLs are VHDL and Verilog. However writing an HDL code for a hardware design is a rather laborious and error-prone process that requires hardware expertise. More specifically, most designs are described at a register transfer level (RTL), where you have to describe combinational logic, basic arithmetic operations and registers, all driven by clock signals. Also, many designing decisions have to be made before code writing, making future changes difficult and costly.

High-level synthesis (HLS) is an automated design process that provides optimized hardware synthesis from high-level programming language specifications such as C/C++ and System C. HLS tries to lower the barriers of HDLs by abstracting away many implementation details. The designer typically develops the algorithmic functionality and the interconnection protocol. The high-level synthesis tools handle the micro-architecture and transform untimed or partially timed functional code into fully timed RTL implementations. The HLS abstraction capabilities allow faster design cycles for FPGAs and let designers to exploit hardware advantages without building up hardware expertise.

Chapter 2

Xilinx Versal ACAPs

2.1 Overview

The Versal portfolio includes six series of devices built on the TSMC 7nm FinFET process technology and uniquely designed to deliver leading-edge heterogeneous acceleration for a wide range of applications from edge to cloud. These adaptive compute acceleration platforms (ACAPs) combine Scalar Engines, Adaptable Engines and Intelligent Engines with a seamless network on chip (NoC) enabled communication capability. Also, a wealth of tools, libraries, IPs and frameworks were launched along with these devices to support the software programmability and low time-to-market requirements.

In this thesis, we will work with the Versal AI Core series and specifically, the VCK190 Evaluation Kit. This series provides the portfolio's highest AI inference and signal processing throughput for cloud, network, and edge applications [16]. The VCK190 system on chip (SoC) contains a host of resources. In the following sections, its major resource blocks will be thoroughly described.

2.2 Design and Architecture

2.2.1 AI Engine (AIE)

2.2.1.1 Overview

To start analyzing this sophisticated architecture let's see a high-level block diagram of a Versal ACAP with an AI Engine (AIE) array in it along with the other two major domains, the processor system (PS) and the programmable logic (PL):

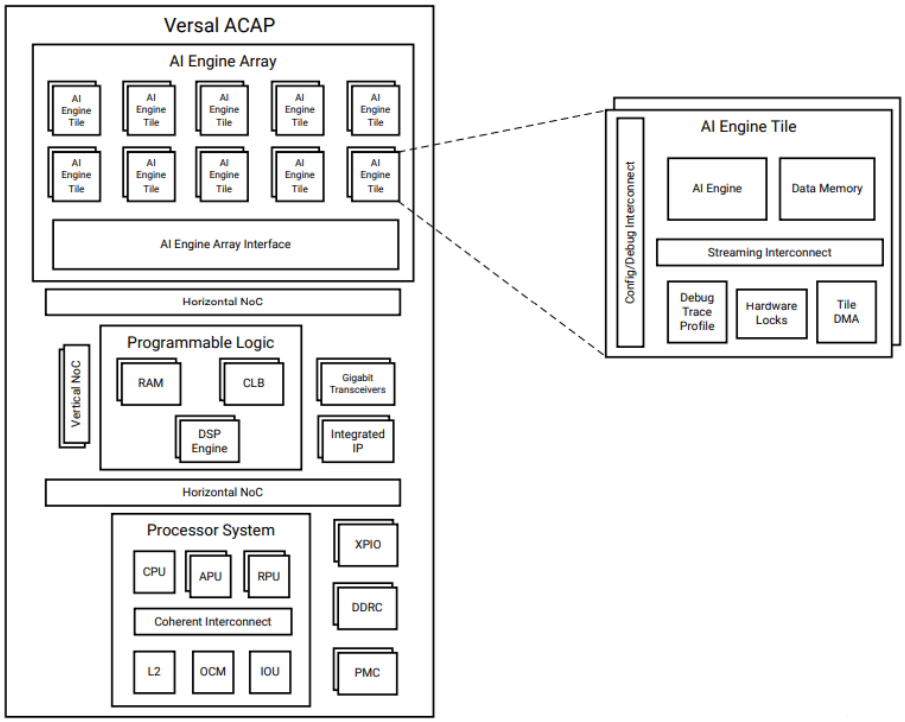


Figure 2.1: Versal Device Top-Level Block Diagram [5].

As we can see, the AIE architecture involves three hierarchical levels. The first level in this hierarchy is the **AI Engine Array**. It is basically a 2D array of **AI Engine Tiles**, that is, the second level. These tiles integrate memory and interconnects, along with the fundamental block of the AI Engine architecture and the last level of the hierarchy, the so called **AI Engine**, a highly-optimized processor featuring single-instruction multiple-data (SIMD) and very-long instruction word (VLIW) processor. The AIE array employs numerous interfaces that make the AI Engines capable of communicating with the rest of the Versal SoC.

AI Engine Architecture

Let's now dive into each one of the aforementioned building blocks, starting with the fundamental one, the AI Engine. As previously said, we are talking about an innovative VLIW and SIMD processing core that supports both fixed-point and floating-point precision. In the diagram below we can see all AIE features:

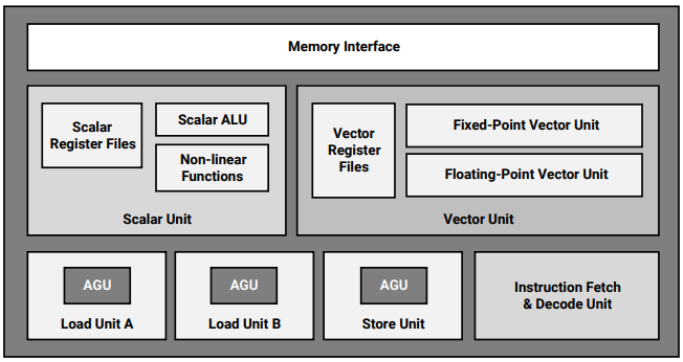


Figure 2.2: AI Engine [6].

- **Register Files:** AIE supports scalar registers, special registers, 128 - 1024-bit wide vector registers to allow SIMD instructions and 384/786-bit accumulator registers to store the results of the vector data path.
- **Instruction Fetch & Decode:** Up to seven operations can be issued in parallel using one VLIW word with support for concurrent issuing of operation to all functional units. The program memory size is 16 KB.
- **Load & Store Unit:** There are three Data Memory Ports, two for load and one for store operations that can operate concurrently. The respective address generator units (AGU) support Fast Fourier Transform (FFT) address generation for multiple addressing modes (fixed, indirect, post-incremental, or cyclic). Data is loaded or stored in data memory that we will talk about later.
- **Scalar Unit:** A 32-bit RISC processor with general purpose pointer and configuration register files. It supports non-linear functions and data type conversion between scalar fixed point and scalar floating point numbers. It also incorporates a scalar ALU with a 32x32-bit scalar multiplier.
- **Vector Fixed-Point Unit:** Contains three separate and largely independent data paths, the Multiply Accumulator (MAC) Path, the Upshift Path and the Shift-round Saturate (SRS) Path. Supports concurrent operations on multiple vector lanes and has a full permute unit with 32-bit granularity. Can perform up to 128 MACs/cycle for 8-bit real operands and 8 MACs/cycle for 16-bit complex operands.
- **Vector Floating-Point Unit:** Same permute and concurrency as the fixed-point vector unit. 8 single-precision MACs/cycle.

AI Engine Tile Architecture

The AI Engine tile is the mid level in the AIE architecture hierarchy and consists of the following modules: Tile interconnect, AI Engine, AI Engine memory module:

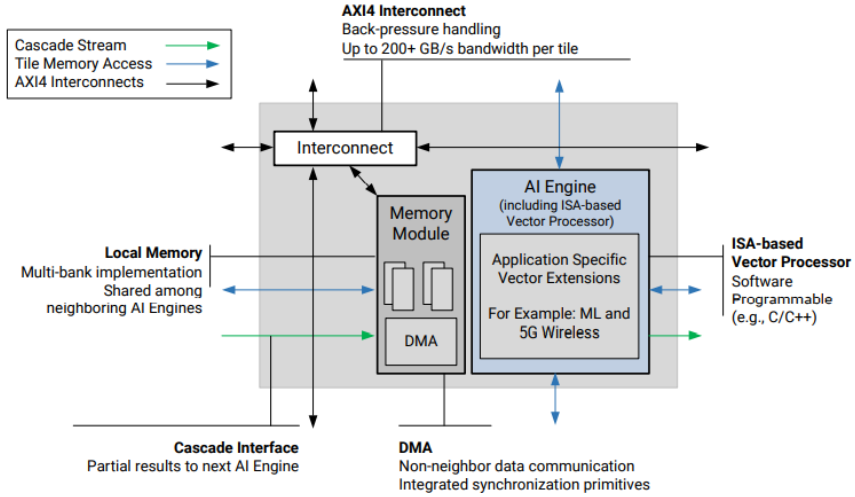


Figure 2.3: AI Engine Tile

- **Tile interconnect module:** Supports AXI4-Stream and memory mapped AXI4 input/output traffic. AXI4-Stream enables data movement between non-neighbouring AIE tiles or

between the AIE and the PL or the NoC. The memory-mapped AXI4 interconnect can be driven from outside of the AIE array by any AXI4 master that can connect to the NoC.

- **AIE memory module:** Has 32 KB of data memory, a memory interface, DMA, locks and control, debug and trace units. When the communication takes place inside an AIE tile or between direct neighboring AIE tiles the shared memory module is used with ping-pong buffer support and all the appropriate locking for synchronization. However, for non-neighbouring AIE tiles a similar, but with increased latency and resources, communication can be established through the DMA in each memory module.

- **The AI Engine:** As described in the previous section.

AI Engine Array Interface Architecture

As shown in figure 2.1, the AIE tiles are organized into a 2-dimensional array, the AIE array. The number of tiles and consequently the number of rows and columns are device specific. For the VCK190 board the maximum of 400 AI Engines is available organized in a 8x50 2D array. Also, as we can see in the block diagram 2.1, the last row of the AIE array hosts the interface tiles that provide the necessary functionality for the AIE array to interface with the rest of the device. There are three types of AIE interface tiles:

- **AI Engine configuration interface tile:** There is exactly one per AIE array. Contains a PLL for AI Engine clock generation and other global control functions such as interrupt controllers, global reset control, and DFx logic.

- **AI Engine - PL interface tile:** Contains a PL module that hosts a Memory-mapped AXI4 and a AXI4-Stream switch, an AIE to PL stream interface and a control, debug and trace unit. Asynchronous FIFOs are provided to handle clock domain crossing.

- **AI Engine - NoC interface tile:** Contains a PL module as described above and also a NoC module with interfaces to the NoC master unit (NMU) and NoC slave unit (NSU). Level shifting is performed because the NMU and NSU are in a different power domain from the AI Engine.

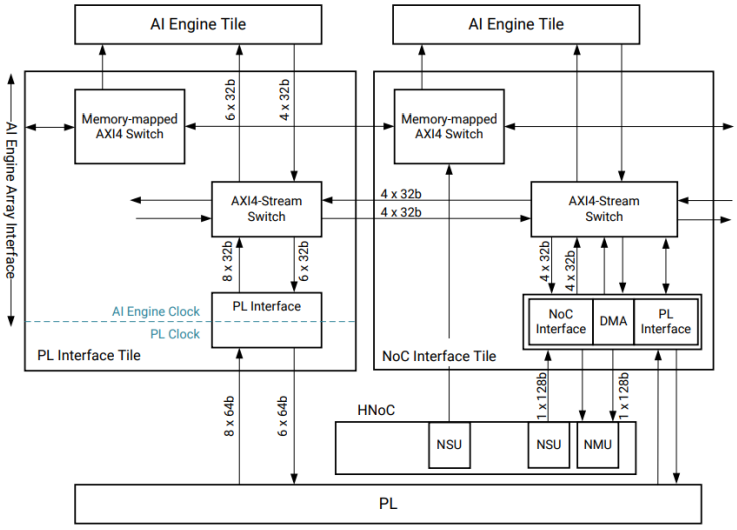


Figure 2.4: PL and NoC Interface Topology

2.2.2 Programmable Logic (PL)

The programmable logic (PL) is the "FPGA" part of Versal ACAPs, a scalable fabric for the efficient hardware implementation of any special purpose function. The Versal PL contains refined configurable logic blocks (CLBs), internal memory, DSP engines and interfaces to any other integrated hardware like the AIE and the processing system (PS).

Configurable Logic Blocks

CLBs contain flexible Look-Up Tables (LUTs) that implement progrmmable logic plus storage elements used as flip-flops or latches. The Versal generation CLB tile is completely redesigned compared to the previous generation UltraScale devices. It has four times more logic capacity, that means 32LUTs/64 slice flip-flops as opposed to 8 LUTs/16 slice flip-flops in UltraScale devices. Also, new CLB features such as the redesigned carry lookahead logic for implementing arithmetic functions or wide logic functions and the dedicated, internal connections to create fast LUT cascades without external routing were introduced to increase total device capacity by reducing area per utilized logic function.

Memory Resources

In addition to the distributed CLB 64-bit RAM, Versal devices feature two types of RAM arrays: Block RAMs and UltraRAMs. Each dual-port block RAM offer a total of 36 Kb with error correction coding (ECC) protection and an option to be configured as two independent 18 Kb RAMs. Also single port mode configurations are available. Furthermore, 288 Kb UltraRAMs are included. They are cascade-able for even larger memories, with ECC and sleep power saving features.

Digital Signal Processing Engine

The DSP Engines are a powerful combination of high speed and small size that preserves the system design flexibility. Versal devices integrate into the PL many dedicated low-power DSP resources with new functional modes that increase the speed and efficiency of many applications beyond digital signal processing such as wide dynamic bus shifters and memory address generators. The DSP engine is defined using the Xilinx DSP58 primitive and can be configured in various modes to better match the application needs.

Each engine includes a dedicated 27×24 bit multiplier and a 58-bit accumulator. The multiplier can be dynamically bypassed, and two 58-bit inputs can feed a single-instruction multiple-data (SIMD) arithmetic unit (dual 24-bit or quad 12-bit add/subtract/accumulate), or a logic unit that can generate any one of ten different logic functions on the two operands.

2.2.3 Processing System (PS)

The processing system (PS) constitutes of the application processing unit (APU) in the full-power domain (FPD), the real-time processing unit (RPU) in the low-power domain (LPD), and various I/O peripherals. The PS, along with the Platform Management Controller (PMC) and the Coherent PCIe Module (CPM) are clustered to form the Control, Interface and Processing System (CIPS) IP core.

APU

The application processing unit (APU) features a dual-core Arm Cortex-A72 processor with an increased L1 instruction cache size (32 KB to 48 KB) and an 1 MB unified L2 cache attached to a Cache Coherent Interconnect (CCI). It is designed for the system control part and for applications that are compute-intensive with no real-time demands.

RPU

The real-time processing unit (RPU) is an Arm Cortex-R5F processor targeting real-time applications. Its main features include the lock-step and the dual processor modes as well as the fact that it integrates tightly coupled memories for predictive execution times.

I/O Peripherals

The I/O peripherals reside in the PMC subsystem for the initial boot and control of the board and in the LPD. They include GPIO controllers, I2C controllers, CAN FD controllers, GEM Gigabit Ethernet MACs, SPI controllers, UART controllers and USB 2.0 controllers.

2.2.4 Network on Chip (NoC)

The network-on-chip (NoC) enriches the traditional fabric interconnect and enables high speed, system level communication between the numerous heterogeneous features, including the AI Engines, the PS, the PL and the DDR Memory Controller. It can be configured using the AXI3, AXI4 or AXI4-Stream for his master and slave interfaces and extends in both horizontal and vertical directions to the edges of the board. The configuration or the programming of the NoC is done through the NoC programming interface (NPI) at boot time and it is fully automated and executed by the PMC.

2.3 AIE Programming

An AI Engine application consists of an adaptable data flow graph (ADF) specification and kernel functions that are written in C++. An ADF graph is made of nodes and edges where nodes are the compute kernel functions and edges represent the data connections between them.

2.3.1 AIE Kernels

Kernels are the main computation functions. They implement the algorithmic logic of the application as C/C++ functions that return void. These kernels use specialized intrinsic calls that target the VLIW vector processor and operate on data streams. They consume input blocks of data and produce output blocks of data either on a window-based or stream-based manner. Also, kernels can have static data or run-time parameter arguments that can be either synchronous or asynchronous. The AIE kernel code is compiled using the AIE compiler (aiecompiler) that is included in the Vitis core development kit (described in Section 3.3).

Each kernel must be defined in its own source file. It is recommended that a header file should declare the function prototypes for all kernels used in a graph. Furthermore, the kernel source files should include all the necessary header files for independent compilation. This organization scheme is recommended for reusability and faster compilation.

2.3.2 ADF Graph

An ADF graph specification consists of a top-level application file and a separate header file. The header file must include the ADF library header (adf.h) and the kernel function prototypes. There, the graph class is defined by declaring the nodes, that is the kernels and some top-level input/output objects like input_plio and output_plio. Then, these nodes are instantiated with the corresponding kernel source files and the ratio of the function run time compared to the cycle budget (known as the run-time ratio), the IO objects are configured using specified PLIO width and IO files and some connectivity information is added to describe the data connections between the kernels.

Finally, a top-level application file is required. This file contains an instance of the configured graph class and a main program. The main program is the driver for the graph. It is used to load, initialize, run, wait and terminate the graph using specific method calls from the graph API.

Chapter 3

Tools, Frameworks and Libraries

3.1 Google Colab

Google Colaboratory, or "Colab" for short, as its name implies, is a product from Google Research. Colab is a Jupyter-like notebook service, entirely hosted on the cloud that requires no resources or setup to use, while providing free of charge access to computing resources like GPUs and TPUs. In Colab, you can write and execute python code through the browser, as well as add rich text and document your code. It also easily integrates frameworks like PyTorch, TensorFlow, Keras, OpenCV, making it especially well suited for machine learning, data analysis and education. Colab notebooks are stored in Google Drive and can be easily shared and edited from multiple team members and co-workers.

3.2 PyTorch

PyTorch is an open-source machine learning framework used for developing and training deep learning models, primarily developed by Meta AI. It is based on the Torch library and can be naturally used with Python but also with C++ in a less polished interface. The two main features of PyTorch are:

- Tensor Computation with strong GPU acceleration support
- Automatic Differentiation system for deep neural networks

The Pythonic nature and the dynamic computation of PyTorch makes it more and more popular among other prominent frameworks, like TensorFlow and Keras.

For this thesis, we used PyTorch for the creation, training and validation of the ESPCN network analyzed in Section 1.3.3.1. The training process was hosted in Google Colab using its native GPU. This is the baseline software implementation with which we will evaluate the performance of the proposed hardware implementation.

3.3 Vitis Unified Software Platform

3.3.1 Introduction

The Vitis Unified Software Platform is a tool that combines all aspects of Xilinx software development into one comprehensive, unified environment for accelerating Edge, Cloud, and Hybrid

computing applications. It enables the development of embedded software and accelerated applications on heterogeneous Xilinx platforms including FPGAs, SoCs, and Versal ACAPs. Vitis offers a software-centric approach to developing both hardware and software, providing the user with the option to choose the level of abstraction he needs [12].

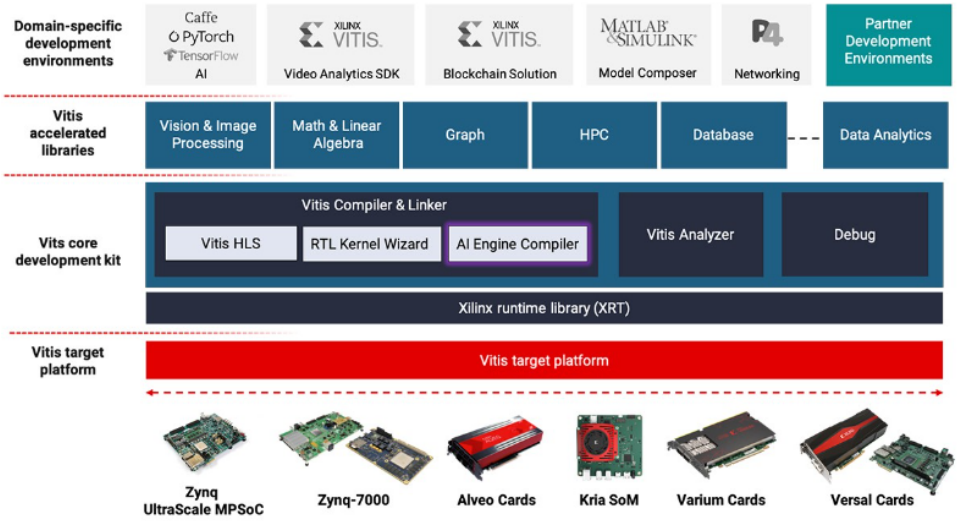


Figure 3.1: Vitis Unified Software Platform stack [7].

As shown in the figure above, the Vitis unified software platform consists of the following features and elements (bottom-up):

- Vitis technology targets acceleration hardware platforms, such as the Alveo Data Center accelerator cards, and Versal or Zynq UltraScale+ MPSoC-based embedded processor platforms.

- Xilinx Runtime (XRT) provides an API and drivers to connect the host program with the target platform and also orchestrate the communication between with the host program and the accelerated kernels.

- Vitis core development kit provides the software development tool stack, such as compilers, linkers, debuggers and analyzers to help you efficiently build and analyze the performance of your application.

- Vitis accelerated libraries, a rich set of hardware-accelerated open-source libraries optimized for Xilinx FPGA and Versal ACAP hardware platforms.

- Plug-in domain-specific development environments enabling development directly in familiar, higher-level frameworks for domain specific applications, like vision and image processing, quantitative finance, database, data analytics, and data compression.

The Vitis software platform consists of an integrated design environment (IDE) for interactive project development, and command-line tools for scripted or manual application development. The Vitis software platform also includes the Vivado Design Suite for implementing the kernel on the target device, and for developing custom hardware platforms.

3.3.2 Execution Model & Embedded Processor Application Acceleration Development Flow

A host application and hardware accelerated kernels are the two main components of a traditional application program in the Vitis core development kit. These two components communicate through channels between them that are either PCIe bus or an AXI bus for embedded platforms. The host program is written in C/C++, uses API abstractions like OpenCL or XRT and it is compiled into an x86 executable that runs on a host processor (such as an Arm processor). The hardware accelerated kernels on the other side, are compiled into an executable device binary (.xclbin) that runs within the programmable logic (PL) region of a Xilinx device. Some Versal ACAP devices, like the one used in this thesis and described thoroughly in Chapter 2, integrate into the described traditional, embedded processor application acceleration flow, AI Engine design graphs (libadf.a).

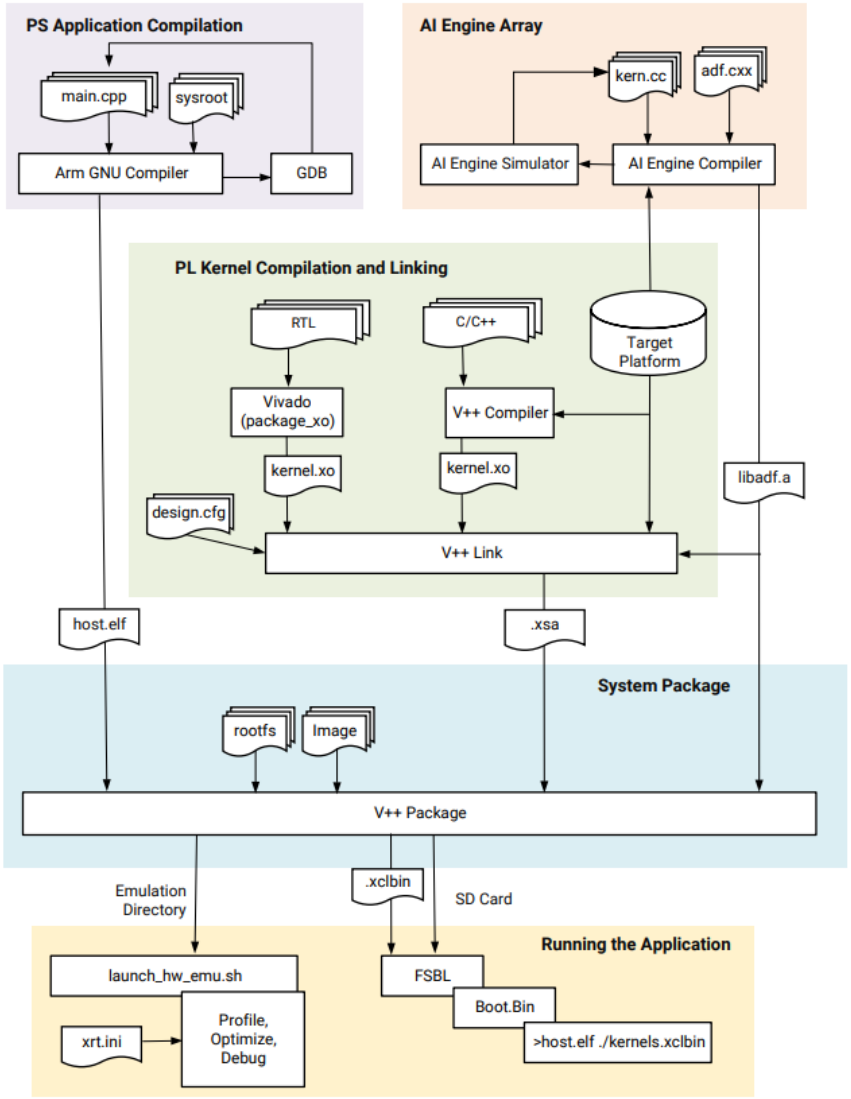


Figure 3.2: Application Development Flow for Versal ACAP and Zynq UltraScale+ MPSoC Devices [12].

The above diagram illustrates the steps to build and run an application that uses the PS (Arm

processor), the PL (FPGA Fabric) as well as the AI Engines when supported.

- The host application is compiled with the GNU Arm cross-compiler to create an ELF file and run on the Cortex®-A72 or Cortex-A53 core processor.

- The AI Engine graphs and kernels are built and simulated using the aiecompiler and aiesimulator tools of Vitis and produces the libadf.a file.

- The PL kernels are cocmpiled into the Xilix object (XO) file using the v++ Vitis compiler or Vitis HLS for C/C++ kernels, or the package_xo command for RTL kernels.

- The produced libadf.a and XO files are linked with the target platform via the v++ –link command to create the platform file (XSA) used to package the design.

- Finally the v++ –package command gathers all the aforementioned files and builds the package that will run on SW or HW emulation and debug, or that will create an SD card to run the application on the target HW platform.

3.4 Vitis HLS

Vitis High-Level Synthesis (HLS) is a high-level synthesis tool tightly integrated into Vitis application acceleration development flow. As shown in figure 3.2, Vitis HLS is responsible for compiling the hardware kernels for the Vitis tools by performing high-level synthesis. These kernels will be accelerated in the programmable logic (PL) region of Xilinx devices and can be written in C/C++ and OpenCL.

The Vitis HLS tool automates much of the design optimization methodology to achieve low latency and high throughput for the PL kernels. For example, the proper interfaces for function arguments and the pipeline pragmas for loop pipelining are inferred by default by Vitis HLS in the application acceleration flow. Vitis HLS also supports customization for interface standards or specific optimizations in order to achieve each design's objectives [23].

3.5 Vitis AI

Vitis AI is a Xilinx development platform for hardware-accelerated AI inference on Xilinx devices, including both edge devices and Alveo data center acceleration cards. It consists of optimized IP, that is, the deep-learning processor unit (DPU) cores, tools, libraries, frameworks, models, and example designs. Its core idea is to provide high efficiency and ease of use, leveraging the maximum potential of AI acceleration on Xilinx FPGAs and ACAPs.

3.5.1 Vitis AI Tools

Vitis AI tries to make hardware-accelerated AI inference easy for users without an FPGA expertise through multiple layers of abstraction. In this direction, Vitis AI stack supports mainstream frameworks like PyTorch and TensorFlow and an easy-to-use toolbox:

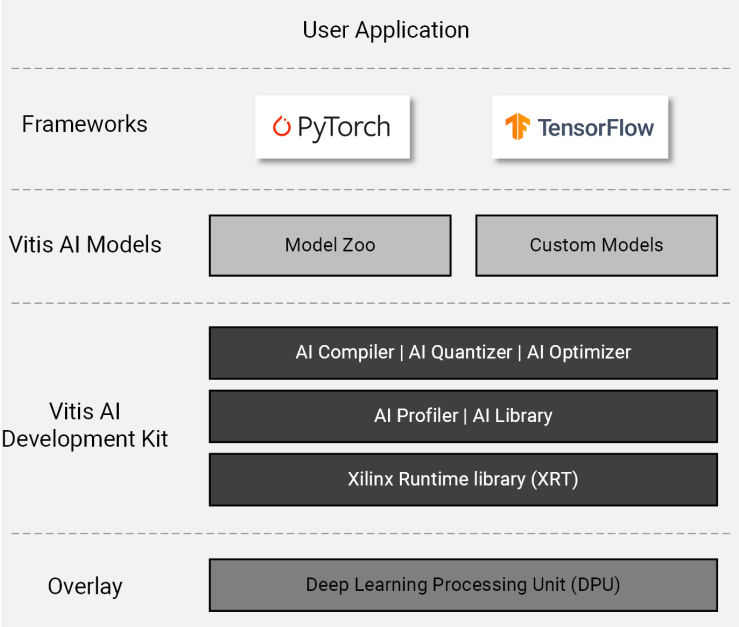


Figure 3.3: Vitis AI Stack [8]

Deep-Learning Processor Unit (DPU): A Domain Specific Architecture (DSA) optimized for accelerating deep neural networks widely adopted in the computer vision industry. It consists of parameterizable IP cores pre-implemented on the hardware and uses an efficient tensor-level instruction set. Vitis AI offers DPUs for varius Xilinx devices both for the edge (Zynq UltraScale+ MPSoC, Kria KV260, Versal cards) and for the cloud (Alveo cards). In this thesis, the DPUCVDX8G is used for the Versal VCK190 evaluation board that we will work on.

Vitis AI Model Zoo: Off-the-shelf optimized and retrainable deep-learning models from popular frameworks like PyTorch and TensorFlow.

Vitis AI Optimizer: Cutting-edge model prunning technology for low complexity with the minimum loss in accuracy.

Vitis AI Quantizer: Converts a 32-bit floating-point model (weights and activations) to a 8-bit fixed-point model, reducing the computation demands without much of accuracy degradation.

Vitis AI Compiler: Maps the AI model to the DPUs efficient instruction set and dataflow model performing various polished optimizations.

Vitis AI Profiler: Performs in-depth analysis of the efficiency and utilization of the AI application offering also a visual representation of the results.

Vitis AI Library: Easy-to-use, unified, high-level libraries and APIs for AI applications hardware abstraction and efficient application development.

Vitis AI Runtime: Unified high-level runtime API for both cloud and edge deployment suitable for the DPU devices.

Chapter 4

Design and Implementation on Versal AI Core ACAP

4.1 System Design

4.1.1 Network mapping on the device

As described in Section 1.3.3.1, the ESPCN model employs three types of layers:

- L Convolutional layers

- $L-1$ Tanh activation layers

- 1 PixelShuffle layer

For this thesis, the traditional approach of the 3-layer ESPCN was followed, thus, we set $L = 3$. Their implementation on the Versal SoC should be done according to their computational needs and the nature of the available hardware.

It is trivial to show that the convolution operation is the most computational intensive one and the one that requires a strong hardware acceleration. For this reason and in order to leverage the powerful compute capabilities of the recently introduced AI Engines (AIE) , we will implement the Conv layers of the model on the AIE part of the board. With appropriate data rearrangement, the convolutional computation will be transformed to matrix multiplication and optimized to be implemented in the AI Engine array.

Two options were considered for the Tanh implementation: a) a vectorized imlementation on the AIE using the polynomial approximation:

$$tanh(x) = \frac{x(27 + x^2)}{(27 + 9x^2)}$$

and b) a look-up table (LUT) based implementation on the programmable logic (PL) part of the device. The evaluation of both implementations showed that the computational needs of the polynomial were not concealed by the vectorization of the operations on the AIE. Therefore, the LUT implementation outperformed the approximation one on both accuracy and speed. Also, the PixelShuffle function and the data rearrangement required for the conversion of convolution to matrix multiplication involve scalar byte operations and interact with read/write memories. This set of operations are suitable for implementation in the PL rather than in the AIE array. Finally,

datamover modules are required to be implemented on the PL for the communication with the DDR through the NoC.

The proposed network mapping on the device and system design is presented in the following block diagram:

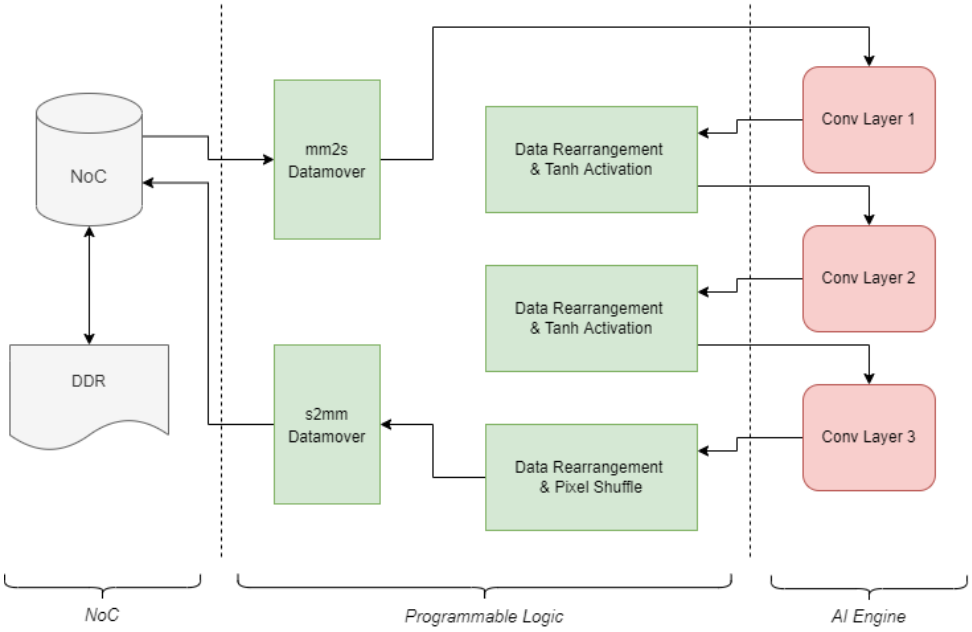


Figure 4.1: ESPCN System Block Diagram.

4.1.2 Network Training & Weight's Extraction

The software implementation of the ESPCN model, that is, the training and evaluation scripts were provided by the PyTorch official Github repositories [17]. The dataset used for training was the BSD300 dataset, which contains 300 images of human subjects. The images are divided into a training set of 200 images, and a test set of 100 images. The model, either on training or evaluation, works with the luminance channel of the YCbCr representation, that is, the Y channel.

Using the training script, we trained the network inside a Google Colaboratory with an NVIDIA Tesla T4 GPU. For this thesis, a 2 times upscale factor is considered with input images of size 16×16. Also, the training was done for 350 epochs with a learning rate equal to 0.001. After training, the weights and biases were extracted using the Netron online application [18].

4.2 Hardware-Specific Custom Implementation

4.2.1 AI Engine: The Convolutional Layer

After a thorough analysis of the Versal's architecture in Section 2.2 it is more than obvious that the compute intensive part of the network, that is, the convolution operation, is the one that will be mapped to the AIE region for acceleration. Following the LeNet tutorial provided by Xilinx [19], the convolution operation is decided to be done using matrix multiplication (mmul) to further leverage the capabilities of the AIE. The proposed design also uses a double tiling scheme to extract the maximum parallelism provided by the AIE array and the AIE API. AIE API is a portable programming interface, implemented as a C++ header-only library that provides types

and operations that get translated into efficient low-level intrinsics and higher-level abstractions such as iterators and multi-dimensional arrays [20].

4.2.1.1 Im2Col

The convolution operation, in fact, performs dot products between the weight kernels and local regions of the input. By taking advantage of this fact a convolutional layer can be transformed and viewed as one big matrix multiplication (MMul).

Im2Col stands for Image to Column and is a technique that transforms the Convolution operation into a mmul operation. Suppose we have a $1 \times 4 \times 4$ input image volume and a convolutional kernel of size 2×2. The input image matrix will be transformed into a matrix of columnar patches according to the convolution operation with the 2×2 filter as illustrated in the figure bellow:

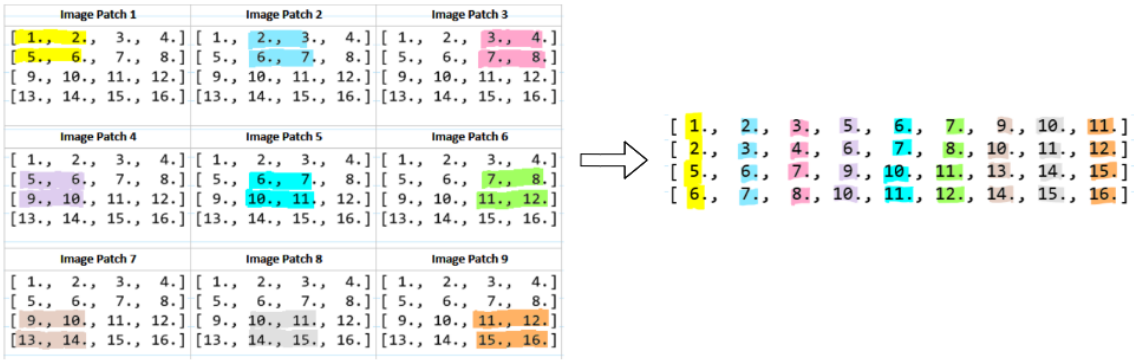


Figure 4.2: Input image volume transformed to a matrix of patches [9].

The result is a 4×9 matrix of patches. Also, the 2×2 filter will be flatten into a 1×4 matrix according to the im2col technique and finally the convolution operation can be implemented as a $1 \times 4 \times 9$ MMul.

4.2.1.2 Tiling Scheme

Let's take for example the first Conv layer of the network. We are working with images of size 16×16, keeping only the Y channel of their YCrCb representation as we described in section 4.1.2. So, an input image of size $1 \times 16 \times 16$ with a 5×5 convolution kernel, 64 output channels, padding 2×2 and stride 1×1 produces an input matrix of size 25×256 for the input according to the Im2Col method for transforming convolutions to MMuls analyzed in the previous section. Subsequently, the weight's matrix is of size 64×25. We zero-pad the two matrices and we have the following $64 \times 32 \times 256$ MMul:

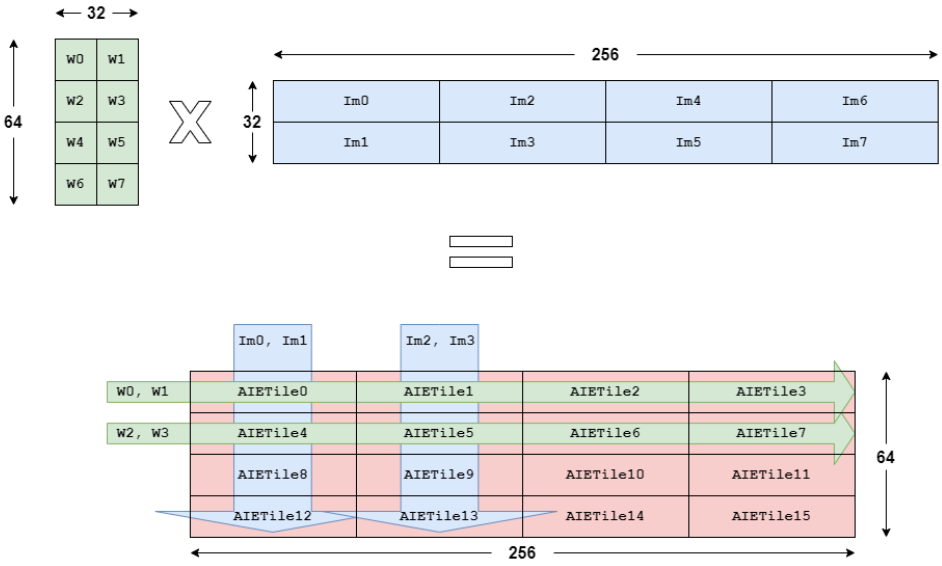


Figure 4.3: The L1 tiling scheme.

As we can see in Figure 4.3, the matrices are divided into tiles of size 16×16 and 16×64 for the weights and the input accordingly. Then, we assign each tile of the output matrix to one AIE tile. So, for the first layer we need 16 tiles that will perform an $16 \times 16 \times 64$ MMul.

For the MMul functionality we use the aie::mmul class template provided by the AIE API. This class template is parametrized with the matrix multiplication shape $M \times K \times N$, the data types and, optionally, the requested accumulation precision. For our design, float numbers are used so we choose the $4 \times 2 \times 4$ MMul shape from the allowed ones.

The following code snippet shows a sample blocked multiplication using the aie::mmul class. The matrices are assumed to be pre-tiled as defined by the mmul shape ($M \times K$ for A, $K \times N$ for B, and $M \times N$ for C). Four blocks of output are calculated in each iteration (C00,C01,C10,C11) using vectors, their aie:load_v/store_v and the mul/mac functions of the aie::mmul class that encapsulate the load/store/mul/mac intrinsics respectively. The restrict keyword in pointers' declaration allows for more aggressive optimisations by the compiler by stating that the data the pointers are pointing to are independent of each other:

```
template <unsigned M, unsigned K, unsigned N>
void mmul_blocked(unsigned rowA, unsigned colA, unsigned colB,
                  const float * __restrict pA,
                  const float * __restrict pB,
                  float * __restrict pC)
{
   using MMUL = aie::mmul<M, K, N, float, float>;

   for (unsigned z = 0; z < rowA; z += 2) chess_loop_range(2,) {
       float * __restrict pC1 = pC + (      z * colB +       0) * MMUL::size_C;
       float * __restrict pC2 = pC + ((z + 1) * colB +       0) * MMUL::size_C;

       for (unsigned j = 0; j < colB; j += 2) chess_loop_range(2,) {
           const float * __restrict pA1 = pA + (      z * colA +      0) * MMUL::
    size_A;
           const float * __restrict pA2 = pA + ((z + 1) * colA +       0) * MMUL::
    size_A;
           const float * __restrict pB1 = pB + (      0 * colB +       j) * MMUL::
```

```
    size_B;
17          const float * __restrict pB2 = pB + (      0 * colB + (j + 1)) * MMUL::
    size_B;
18
19          aie::vector<float, MMUL::size_A> A0 = aie::load_v<MMUL::size_A>(pA1);
20          pA1 += MMUL::size_A;
21          aie::vector<float, MMUL::size_A> A1 =  aie::load_v<MMUL::size_A>(pA2);
22          pA2 += MMUL::size_A;
23          aie::vector<float, MMUL::size_B> B0 = aie::load_v<MMUL::size_B>(pB1);
24          pB1 += MMUL::size_B * colB;
25          aie::vector<float, MMUL::size_B> B1 = aie::load_v<MMUL::size_B>(pB2);
26          pB2 += MMUL::size_B * colB;
27
28          MMUL C00; C00.mul(A0, B0);
29          MMUL C01; C01.mul(A0, B1);
30          MMUL C10; C10.mul(A1, B0);
31          MMUL C11; C11.mul(A1, B1);
32
33          for (unsigned i = 1; i < colA; ++i)
34          chess_prepare_for_pipelining chess_loop_range(3,) {
35              A0 = aie::load_v<MMUL::size_A>(pA1); pA1 += MMUL::size_A;
36              A1 = aie::load_v<MMUL::size_A>(pA2); pA2 += MMUL::size_A;
37              B0 = aie::load_v<MMUL::size_B>(pB1); pB1 += MMUL::size_B * colB;
38              B1 = aie::load_v<MMUL::size_B>(pB2); pB2 += MMUL::size_B * colB;
39
40              C00.mac(A0, B0);
41              C01.mac(A0, B1);
42              C10.mac(A1, B0);
43              C11.mac(A1, B1);
44          }
45
46          aie::store_v(pC1, C00.template to_vector<float>()); pC1 += MMUL::size_C;
47          aie::store_v(pC1, C01.template to_vector<float>()); pC1 += MMUL::size_C;
48          aie::store_v(pC2, C10.template to_vector<float>()); pC2 += MMUL::size_C;
49          aie::store_v(pC2, C11.template to_vector<float>()); pC2 += MMUL::size_C;
50      }
51  }
52 }
```

Listing 4.1: Blocked Matrix Multiplication

Some additional logic is added in the above code sample for the accumulation operation between the intermediate matrix products (e.g. $W0 \times Im0$ and $W1 \times Im1$) required for each result (e.g. AIETile0) in this example.

Therefore, we have 16 AIE tiles, each one performing a $16 \times 16 \times 64$ mmul tiled in $4 \times 2 \times 4$. That means that we employed a double tiling scheme: Inter-Tiling by splitting the computation in multiple tiles and Intra-Tiling by performing a tiled MMul inside each AIE tile. So, in total, the MMul size of each layer, the selected tiling scheme and the number of AIE tiles required in each layer is presented in the following table:

Table 4.1: Tiling Schemes in each Conv Layer

Layer	MMul Size	Tiling Size	Number of AIE Tiles
Conv1	64 x 32 x 256	16 x 16 x 64	16
Conv2	32 x 576 x 256	8 x 576 x 8	128
Conv3	4 x 288 x 256	4 x 288 x 16	16

4.2.1.3 AIE Kernels

The kernel functions of the AIE part of the application contain the C++ code for a tiled MMul as described in the previous subsection. We have three kernel source files, one for each Conv layer. This is done because, although the basic operation is a parametrized MMul, each layer should be associated with different weights and biases. Furthermore, the need for intermediate products dictated by the tiling scheme in the first layer introduces some additional logic. We will proceed with the first layer as an example to analyze the proposed implementation. The other two layers follow the same approach.

For binding each kernel with a specific set of weights and biases, we use the C++ kernel class support. The C++ kernel class allows internal states for each kernel instance to be encapsulated within the corresponding class object. The following code snippet showcases the declaration of the MMUL_T_1 class for the first Conv layer that should be placed in a header file:

```
class MMUL_T_1
{
private:
  float (&wgts)[WSIZE1];
  float (&b)[SIZE_OUT1];
  float (&intrmdtRes)[SIZE_OUT1];

public:
  MMUL_T_1(float(&weights)[WSIZE1], float(&zeros)[SIZE_OUT1],
  float(&bias)[SIZE_OUT1]);

  void mmul1(const int RowA_tile, const int ColA_tile, const int ColB_tile,
  float* A_in, float* C_out, int tile, int shift);

  void mmul1_top(input_window_float* in, output_window_float* out);

  static void registerKernelClass()
  {
    REGISTER_FUNCTION(MMUL_T_1::mmul1_top);
    REGISTER_PARAMETER(wgts);
    REGISTER_PARAMETER(b);
    REGISTER_PARAMETER(intrmdtRes);
  }
};
```

Listing 4.2: MMUL_T_1 Class Declaration

and its corresponding constructor:

```
MMUL_T_1::MMUL_T_1(float(&weights)[WSIZE1], float(&zeros)[SIZE_OUT1],
float(&bias)[SIZE_OUT1])
: wgts(weights), intrmdtRes(zeros), b(bias)
{ }
```

Listing 4.3: MMUL_T_1 Class Constructor

A static void registerKernelClass() method must be written to host the REGISTER_FUNCTION and REGISTER_PARAMETER macros. The REGISTER_FUNCTION macro is used to register the class run method that will be executed on the AIE core to perform the kernel functionality, in the above example, the mmul1_top() void function. Also, REGISTER_PARAMETER macro is used to let the AIE compiler that the member variables used as its argument are intended to be allocated by the compiler.

The kernel class constructor is located in the same source file as the one that contains the kernel function, that is, the run method of the kernel class. The constructor initializers like the wgts(weights) one, initializes wgts, for example, to the reference to an array allocated externally. This external allocation is done using the kernel::create_object method inside the ADF graph representation that will be analyzed in the following subsection.

Let's take a closer look on the run method of the MMUL_T_1 class, or else, the kernel function mmul1_top():

```
void MMUL_T_1::mmul1_top(input_window_float* in, output_window_float* out)
{
  int shift = 9;
  set_sat();
  set_rnd(rnd_sym_inf);
  unsigned int i, times=SIZE_OUT1/32;

/* Compute the output tile */
  for (i = 0; i < ITERS1; i++)
  {
    mmul1(ROW_A1_TILE >> 2, COL_A1_TILE >> 1, COL_B1_TILE >> 2,
    (float *) in -> ptr, (float *) intrmdtRes, i, shift);
  }

/* Add the bias */
  float * ptr = intrmdtRes; aie::vector<float, 32> res = aie::load_v<32>(ptr);
  float * bias_p = b; aie::vector<float, 32> bias_v = aie::load_v<32>(bias_p);
  aie::vector<float, 32> out_v = aie::add(res,bias_v);
  float * out_ptr = (float *) out -> ptr; aie::store_v(out_ptr, out_v);

  for (i = 1; i < times; i++)
  {
    ptr += 32;  bias_p += 32;  out_ptr += 32;
    res = aie::load_v<32>(ptr);
    bias_v = aie::load_v<32>(bias_p);
    out_v = aie::add(res,bias_v);
    aie::store_v(out_ptr, out_v);
  }
}
```

Listing 4.4: The run method of the MMUL_T_1 Class

As we can notice in the code, we chose to use a window-based communication for the kernels' input or output data. This gives the ability to kernels to perform random access within the window of data. Then, we set the appropriate round and saturation mode for the accumulation result and call the mmul1() function to compute the output tile. This is done iteratively by calculating and accumulating the intermediate results. In the end, the bias is added to the result.

The mmul1() function is the one that performs the actual MMul operation based on the Blocked Matrix Multiplication code provided by the AIE API and presented in Listing 4.1. The difference here is that if we call the function to calculate a result not for the first time, it initially loads the

previous intermediate result and then adds to it the current calculated product:

```
void MMUL_T_1::mmul1(
        const int RowA_tile,
        const int ColA_tile,
        const int ColB_tile,
        float* B_in,
        float* C_out,
    int tile,
  int shift
) {
  /*      Matrix dimensions      */
  constexpr size_t sizeTileA = 4 * 2;
  constexpr size_t sizeTileB = 2 * 4;
  constexpr size_t sizeTileC = 4 * 4;

  /*          Mul Intrinsic       */
  using MMUL = aie::mmul<4, 2, 4, float, float>;
    unsigned int i,j,z;

  for (z=0; z<RowA_tile; z+=2)
  {
    /*         Output vector          */
    float * __restrict pC1 = C_out + (      z * ColB_tile +  0) * sizeTileC;
    float * __restrict pC2 = C_out + ((z + 1) * ColB_tile +  0) * sizeTileC;

    for (j=0; j<ColB_tile; j+=2)
    {
          const float * __restrict pA1 = wgts + z * ColA_tile * sizeTileA +
          + tile*sizeTileA*RowA_tile*ColA_tile;
          const float * __restrict pA2 = wgts + (z+1) * ColA_tile * sizeTileA +
          + tile*sizeTileA*RowA_tile*ColA_tile;
          const float * __restrict pB1 = B_in + j * sizeTileB +
          + tile*sizeTileB*ColA_tile*ColB_tile;
          const float * __restrict pB2 = B_in + (j+1) * sizeTileB +
          + tile*sizeTileB*ColA_tile*ColB_tile;

          aie::vector<float, sizeTileA> A0 = aie::load_v<sizeTileA>(pA1);
          pA1 += sizeTileA;
          aie::vector<float, sizeTileA> A1 = aie::load_v<sizeTileA>(pA2);
          pA2 += sizeTileA;
          aie::vector<float, sizeTileB> B0 = aie::load_v<sizeTileB>(pB1);
          pB1 += sizeTileB * ColB_tile;
          aie::vector<float, sizeTileB> B1 = aie::load_v<sizeTileB>(pB2);
          pB2 += sizeTileB * ColB_tile;

          MMUL C00; C00.mul(A0, B0); MMUL C01; C01.mul(A0, B1);
          MMUL C10; C10.mul(A1, B0); MMUL C11; C11.mul(A1, B1);

          for (i = 1; i < ColA_tile; ++i)
          {
              A0 = aie::load_v<sizeTileA>(pA1); pA1 += sizeTileA;
              A1 = aie::load_v<sizeTileA>(pA2); pA2 += sizeTileA;
              B0 = aie::load_v<sizeTileB>(pB1); pB1 += sizeTileB * ColB_tile;
              B1 = aie::load_v<sizeTileB>(pB2); pB2 += sizeTileB * ColB_tile;

              C00.mac(A0, B0); C01.mac(A0, B1);
              C10.mac(A1, B0); C11.mac(A1, B1);
          }
```

```
            if (tile == 0){
               aie::store_v(pC1, C00.template to_vector<float>(shift));
               pC1 += sizeTileC;
               aie::store_v(pC1, C01.template to_vector<float>(shift));
               pC1 += sizeTileC;
               aie::store_v(pC2, C10.template to_vector<float>(shift));
               pC2 += sizeTileC;
               aie::store_v(pC2, C11.template to_vector<float>(shift));
               pC2 += sizeTileC;
            }
            else{
               aie::vector<float, sizeTileC> C_00 = aie::load_v<sizeTileC>(pC1);
               C_00 = aie::add(C_00, C00.template to_vector<float>(shift));
               aie::store_v(pC1, C_00); pC1 += sizeTileC;
               aie::vector<float, sizeTileC> C_01 = aie::load_v<sizeTileC>(pC1);
               C_01 = aie::add(C_01, C01.template to_vector<float>(shift));
               aie::store_v(pC1, C_01); pC1 += sizeTileC;

               aie::vector<float, sizeTileC> C_10 = aie::load_v<sizeTileC>(pC2);
               C_10 = aie::add(C_10, C10.template to_vector<float>(shift));
               aie::store_v(pC2, C_10); pC2 += sizeTileC;
               aie::vector<float, sizeTileC> C_11 = aie::load_v<sizeTileC>(pC2);
               C_11 = aie::add(C_11, C11.template to_vector<float>(shift));
               aie::store_v(pC2, C_11); pC2 += sizeTileC;
            }
         }
      }
}
```

Listing 4.5: The mmul1() function.

4.2.1.4 ADF Graph Specification

In order to achieve code reusability and legibility we separate the ADF specification into two different header files. The first one, naming subgraph.h, instantiates a layersSubgraph class including the declaration and configuration of the nodes of the ADF graph, that is the AIE kernels, and the edges, that is, the window connections with the I/O data. The second header file, naming graph.h, hosts the layersGraph class that instantiates a layersSubgraph object and connects its I/O ports with the PLIO ports. We, again, present the part of the aforementioned files regarding only the first layer for brevity, as the other two layers follow the same logic:

```
#ifndef __SUBSYS_H__
#define __SUBSYS_H__

#include <adf.h>
#include "mmul_core.h"
#include "weights_lut.h"
#include "bias_lut.h"

using namespace adf;
template <int COL_OFFSET, int ROW_OFFSET>
class layersSubgraph : public adf::graph {
private:
  kernel layer1[NUM_TILES1];
  ...
  ...
public:
```

```
  input_port in1[NUM_TILES1/4];
  ...
  ...
  output_port out1[NUM_TILES1];
  ...
  ...

  layersSubgraph() {
    /* LAYER 1 */
    for (unsigned int j=0; j<NUM_TILES1; j++)
    {
      if (j < 4 ){
        layer1[j] = kernel::create_object<MMUL_T_1>(W1_1, zeros1, bias1_1);
        single_buffer(in1[j]);
        location<kernel>(layer1[j]) = tile(COL_OFFSET,ROW_OFFSET+(2*j));
      }
      else if (j < 8){
        layer1[j] = kernel::create_object<MMUL_T_1>(W1_2, zeros1, bias1_2);
        location<kernel>(layer1[j]) = tile(COL_OFFSET+1,ROW_OFFSET+(2*(j&3))+1);
      }
      else if (j < 12){
        layer1[j] = kernel::create_object<MMUL_T_1>(W1_3, zeros1, bias1_3);
        location<kernel>(layer1[j]) = tile(COL_OFFSET+2,ROW_OFFSET+(2*(j&3)));

      }
      else{
        layer1[j] = kernel::create_object<MMUL_T_1>(W1_4, zeros1, bias1_4);
        location<kernel>(layer1[j]) = tile(COL_OFFSET+3,ROW_OFFSET+(2*(j&3))+1);
      }
      source(layer1[j]) = "mmul_core1.cc";
      runtime<ratio>(layer1[j]) = 0.8;

      location<buffer>(layer1[j].out[0]) = location<kernel>(layer1[j]);
      location<buffer>(layer1[j].in[0]) = location<kernel>(layer1[j]);
      location<parameter>(layer1[j].param[0]) = location<kernel>(layer1[j]);
      location<parameter>(layer1[j].param[1]) = location<kernel>(layer1[j]);
      location<parameter>(layer1[j].param[2]) = location<kernel>(layer1[j]);

      connect< window <WIN_SIZE_IN1> >(in1[j&3], layer1[j].in[0]);
      connect< window <WIN_SIZE_OUT1> >(layer1[j].out[0], out1[j]);

      single_buffer(layer1[j].in[0]); single_buffer(layer1[j].out[0]);
      single_buffer(out1[j]);
    }

    /* LAYER 2 */
    ...

    /* LAYER 3 */
     ...

    location<graph>(*this) = bounding_box(COL_OFFSET,ROW_OFFSET,COL_OFFSET+39,
    ROW_OFFSET+7);
  }
};
#endif //__SUBSYS_H__
```

Listing 4.6: subgraph.h

First of all, we declare the input and output ports of the graph. For the first Conv layer $NUM\_TILES1 = 16$, so we have 4 input ports and 16 output ports for the 16 AIE tiles comprising this layer. We need only 4 ports as input because as illustrated in figure 4.2 the AIE tiles of each column consume the same part of the input image. The output of the first layer, obviously has 16 ports as each AIE tile produces each own part of the result.

Proceeding, we can see here the kernel::object method that creates a representation of the MMUL_T_1 kernel instance and initializes the wgts, intrmdtRes and b constructor parameters presented in Listing 4.3. Furthermore, we constrain the input and output ports to be implemented as single buffers because the local data memory is at a premium, we specify the source file containing the definition of the kernel function and we set the run-time ratio, a core usage fraction for the kernel. Using the connect method, we establish the window connections for the input and the output data of each kernel by specifying the number of bytes between the two ports.

An important constraint used in this design is the location one. By specifying the exact location of each kernel and its corresponding buffers or parameters, we can control and guide the AIE mapper to find a legal solution for the design. This is extremely useful for building large graphs, like ours, because without these constraints, the mapper may fail to find a legal solution. And indeed that is the case for our design. Below, we can see the mapping of our design in the AIE array visualized using the vitis_analyzer tool:

Figure 4.4: The proposed designed mapped into the whole AIE array.

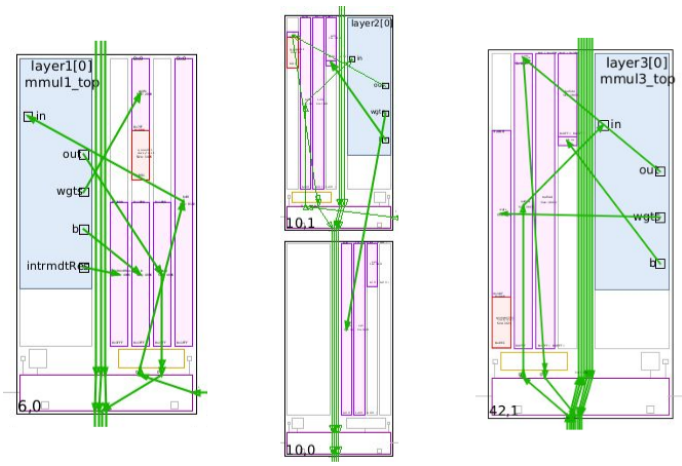


Figure 4.5: From left to right: an AIE tile for layer 1, an AIE tile pair for layer 2 and an AIE tile for layer 3.

In figure 4.4 we can see the whole AIE array with its 400 tiles. In accordance with table 4.1 we use $16 + 128 \times 2 + 16$ AIE tiles. This $\times 2$ for the second layer arises from the fact that the memory requirements of the kernels of the second layer exceed the available local memory. Thus, memory from neighbouring tiles must be utilized. In figure 4.5 we zoom into one AIE tile from every layer. The purple rectangles are the I/O, weights, bias and intermediate result (only for the first layer) buffers. The ciel rectangle is the kernel function on the AIE core and the green arrows represent the memory accesses. The red rectangles are the system memory (heap & stack).

Proceeding to the graph.h header file, we can see how the layersSubgraph is connected with the PL:

```
#include <adf.h>
#include "subgraph.h"
#include "mmul_core.h"

using namespace adf;

template<int COL0>
class layersGraph: public graph{

public:
    input_plio  in1[NUM_TILES1/4];
    ...
    ...

    output_plio out1[NUM_TILES1];
    ...
    ...

    layersSubgraph<COL_OFF1, ROW_OFF1> mygraph;

    layersGraph(){
        // Layer1
        for(unsigned k=0; k<NUM_TILES1; k++) {
          if (k<4){
            in1[k]=input_plio::create("DataInL1_"+std::to_string(k), plio_128_bits,
            "data/l1_tiles/coltile"+std::to_string(k)+"_128plio.txt");
            connect<>(in1[k].out[0], mygraph.in1[k]);
          }
            out1[k]=output_plio::create("DataOutL1_"+std::to_string(k),
            plio_128_bits, "data/l1_out/tile"+std::to_string(k)+".txt");
            connect<>(mygraph.out1[k], out1[k].in[0]);
        }

        //  Layer2
        ...
        //  Layer3
        ...
        }
    };
};
```

Listing 4.7: graph.h

The input_plio/ouput_plio::create constructors specify the name of the PLIO connections that will be used later, on the linking stage, and other simulation parameters like the width of the PLIO data connection and the test bench data files associated with each PLIO connection. Below we can see how the graph looks like for the 1st row of tiles of the first layer again using the vitis_analyzer

tool:

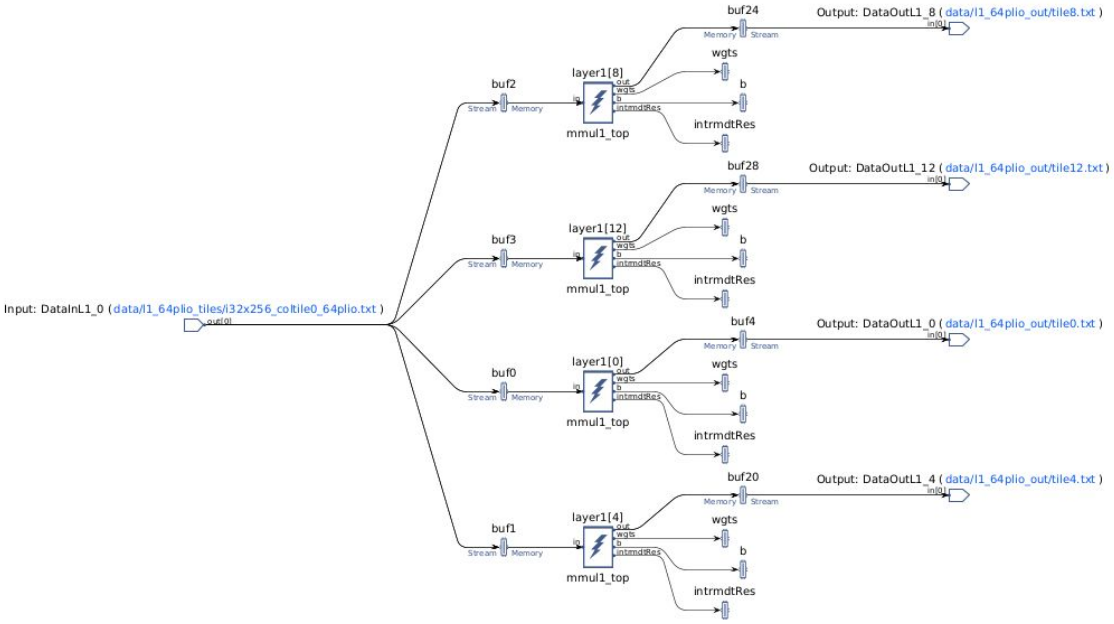


Figure 4.6: Graph Illustration

Finally, a top-level application file (graph.cpp) is defined:

```
#include "graph.h"
#include "mmul_core.h"

layersGraph<COL_OFF1> mmul_graph;

#if defined  (__AIESIM__) || defined(__X86SIM__)
int main(void) {
  mmul_graph.init();
  mmul_graph.run(1);
  mmul_graph.end();
    return 0;
}
#endif
```

Listing 4.8: graph.cpp

The main() initializes the mmul_graph instance, executes it for 1 iteration and then terminates it. It acts as a driver for the SW and AIE emulation flows. Therefore, this main() must be excluded from hardware and hardware emulation flows and the guard macros __AIESIM__ and __X86SIM__ provide that flexibility.

4.2.2 PL Kernels

As the block diagram 4.1 illustrates, the PL part of the VCK190 board will host the implementation of the Tanh activation layer, the PixelShuffle layer, the proper data handling functionality and the datamovers for the communication with the DDR memory. The details of their implementation will be described in the following sections. The PL is programmed according to the traditional FPGA programming paradigm using HLS C code.

4.2.2.1 Datamovers

The PL-based datamovers are comprised of two PL kernels: the mm2s and the s2mm kernel. The mm2s kernel gets the input images from the DDR memory through the NoC and sends them appropriately to the AIE tiles of the first layer through their input_plio ports. The s2mm kernel receives the output images from the PixelShuffle layer and streams out the data to the DDR memory through the NoC. The interface for the connections with the DDR memory is a memory-mapped AXI4 interface and the interface between the PL kernels and the AI Engines is an AXI4-Stream interface, compatible with the AIE array's interface.

```
void mm2s(float* mem0, hls::stream<float>& s0, hls::stream<float>& s1,
            hls::stream<float>& s2, hls::stream<float>& s3, int size, int iter){

#pragma HLS INTERFACE m_axi port=mem0 offset=slave bundle=gmem
#pragma HLS INTERFACE s_axilite port=mem0 bundle=control
#pragma HLS INTERFACE s_axilite port=size bundle=control
#pragma HLS INTERFACE s_axilite port=iter bundle=control
#pragma HLS interface s_axilite port=return bundle=control

  float x0,x1,x2,x3;

  for(int j = 0; j < iter; j++){
    for(int i = 0; i < size; i++) {
#pragma HLS PIPELINE II=1
      x0 = mem0[i+j*4*size];        x1 = mem0[i+size+j*4*size];
      x2 = mem0[i+2*size+j*4*size]; x3 = mem0[i+3*size+j*4*size];
      s0.write(x0);  s1.write(x1);  s2.write(x2);  s3.write(x3);
    }
  }
}

void s2mm(float* mem, hls::stream<float>& s0, int size){

#pragma HLS INTERFACE m_axi port=mem offset=slave bundle=gmem
#pragma HLS INTERFACE s_axilite port=mem bundle=control
#pragma HLS INTERFACE s_axilite port=size bundle=control
#pragma HLS interface s_axilite port=return bundle=control

  float x0;

  for(int i = 0; i < size; i++) {
#pragma HLS PIPELINE II=1
      x0 = s0.read();
      mem[i] = x0;
  }
}
```

Listing 4.9: mm2s and s2mm datamovers

4.2.2.2 Activation Layer

This Layer contains the Tanh activation functionality and all the required data rearrangement logic to walk from one Conv layer to another. More specifically, as Section 4.2.1.2 describes we have a double tiling scheme for the MMul operation. Also, there is an Im2Col operation (analyzed in section 4.2.1.1) that, according to the parameters of the Conv layer, transforms the input image or the feature maps to enable the use of MMul instead of a convolution. However, for a proper

use of this operation the matrices should not be tiled. Therefore, an activation layer between two Conv layers, employs the data manipulation pipeline illustrated by the below block diagram:

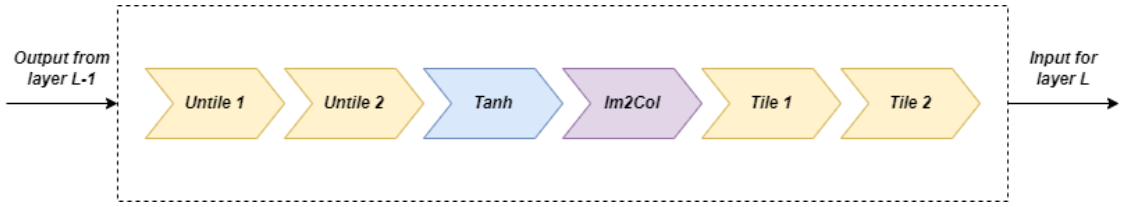


Figure 4.7: Activation Layer Block Diagram.

Tiling & Untiling

The output from each Conv layer is read by the PL activation layers as a 1-D matrix. Using the below functions, we achieve to untile and tile these matrices appropriately, given the dimensions of the input and the desired tile dimensions:

```
void tile(float* input, int in_rows, int in_cols, int t_rows, int t_cols, float*
    output)
{
  unsigned int i, j, k, l;
  tiling_loop:
    for (i=0; i<(in_rows/t_rows); i++){
      for (j=0; j<(in_cols/t_cols); j++){
          for (k=0; k<t_rows; k++){
            for (l=0; l<t_cols; l++){
                  output[i*in_cols*t_rows+j*t_rows*t_cols+k*t_cols+l] =
                  = input[i*in_cols*t_rows+j*t_cols+k*in_cols+l];
            }
          }
      }
    }
}

void untile(float* input, int in_rows, int in_cols, int t_rows, int t_cols, float*
    output)
{
  unsigned int i, j, k, l;
  untiling_loop:
    for (i = 0; i < (in_rows/t_rows); i++){
      for (j = 0; j < t_rows; j++){
          for (k = 0; k < (in_cols/t_cols); k++){
            for (l = 0; l < t_cols; l++){
                  output[i*in_cols*t_rows+j*in_cols+k*t_cols+l] =
                  = input[i*in_cols*t_rows+j*t_cols+k*t_rows*t_cols+l];
            }
          }
      }
    }
}
```

Listing 4.10: Tile & Untile operations

Tanh Implementation

As discussed in section 4.1.1 the Tanh activation functionality is decided to be implemented by a look-up table. Following the same approach as [24], we use a look-up table with 1024 Tanh values in the range of -4 to +4. For inputs greater than 4 or less than -4 the activation will be 1

or -1 accordingly. To index the look-up table, the given float values are converted to integer with scaling:

```
int float2fix(float n, int sft)
{
  return round(n * pow(2.0, sft));
}

void tanh(float* in, float* tanh_l1)
{
  for(int i=0; i<L1_OUT_SIZE; i++)
  {
    #pragma HLS PIPELINE II=1
    if (in[i] >= 4)
      tanh_l1[i] = 1.0;
    else if (in[i] < -4)
      tanh_l1[i] = -1.0;
    else
    {
      tanh_l1[i] = tanh_lut4[(LUT_SIZE/2) + float2fix(in[i], 7)];
    }
  }
}
```

Listing 4.11: Tanh activation

Im2Col Implementation

The Im2Col method is implemented using the two following functions from the Caffe's repository on Github [25]:

```
float im2col_get_pixel(float *im, int height, int width,
                       int row, int col, int channel, int pad)
{
    row -= pad;
    col -= pad;

    if (row < 0 || col < 0 ||
        row >= height || col >= width) return 0;
    return im[col + width*(row + height*channel)];
}

void im2col(float* data_im,
     int channels,  int height,  int width,
     int ksize,  int stride, int pad, float* data_col)
{
    int c,h,w;
    int height_col = (height + 2*pad - ksize) / stride + 1;
    int width_col = (width + 2*pad - ksize) / stride + 1;

    int channels_col = channels * ksize * ksize;
    im2col_loop:
    for (c = 0; c < channels_col; ++c) {
        int w_offset = c % ksize;
        int h_offset = (c / ksize) % ksize;
        int c_im = c / ksize / ksize;
        for (h = 0; h < height_col; ++h) {
            for (w = 0; w < width_col; ++w) {
                int im_row = h_offset + h * stride;
                int im_col = w_offset + w * stride;
```

```
                int col_index = (c * height_col + h) * width_col + w;
                data_col[col_index] = im2col_get_pixel(data_im, height, width,
                        im_row, im_col, c_im, pad);
            }
        }
    }
}
```

Listing 4.12: Im2Col Implementation

Overall the functions listed above are synthesizing the final activation layer, let's say the one after the first Conv Layer, as follows:

```
extern "C" {
void activation(hls::stream<float>& x0, ..., hls::stream<float>& x15,
        hls::stream<float>& y0,  ..., hls::stream<float>& y31)
{

  #pragma HLS INTERFACE axis port = x0
    ...
  #pragma HLS INTERFACE axis port = x15

  #pragma HLS INTERFACE axis port = y0
  ...
  #pragma HLS INTERFACE axis port = y31
    #pragma HLS INTERFACE ap_ctrl_none port = return

  float conv1_out[L1_OUT_SIZE], unt1_l1[L1_OUT_SIZE], unt1_l2[L1_OUT_SIZE],
  tanh_l1[L1_OUT_SIZE], im2col_l2[L2_COL_SIZE], im2col_l2_t[L2_COL_SIZE],
  im2col_l2_in[L2_COL_SIZE];

  /*array partition to create parallel read/write ports*/
  #pragma HLS array_partition variable=conv1_out block factor=16 dim=1
  #pragma HLS array_partition variable=im2col_l2_in block factor=16 dim=1

  l1_read:
  read_input(x0, ..., x15, conv1_out);

  l1_untiling4x4:
  untile(conv1_out, L1_OUT_H*(L1_OUT_W/L1_OUT_TILE_W), L1_OUT_TILE_W, 4, 4,
  unt1_l1);

  l1_untiling64x16:
  untile(unt1_l1, L1_OUT_H, L1_OUT_W, L1_OUT_TILE_H, L1_OUT_TILE_W, unt2_l1);

  l1_tanh:
  tanh(unt2_l1, tanh_l1);

  l1_im2col:
  im2col(tanh_l1, L2_IN_C, L2_IM_H, L2_IM_W, L2_K, L2_S, L2_P, im2col_l2);

  l1_tiling576x8:
  tile(im2col_l2, L2_COL_H, L2_COL_W, L2_COL_TILE_H, L2_COL_TILE_W, im2col_l2_t);

  l1_tiling2x4:
  tile(im2col_l2_t, L2_COL_H*(L2_COL_W/L2_COL_TILE_W), L2_COL_TILE_W, 2, 4,
  im2col_l2_in);

  layer_1_write:
```

```
  write_output(y0, ..., y31, im2col_l2_in);
}
}
```

Listing 4.13: Activation Layer Main Function

To guide the synthesis and optimize the hardware design we use some #pragma directives. More specifically, we specify the interfaces of the input and output ports to be AXI4-Stream interfaces and the

```
    #pragma HLS INTERFACE ap_ctrl_none port = return
```

directive dictates data driven execution which enables the kernel to run when data is available, and stall when data is not. Also the array partition directive effectively increases the amount of read and write ports to potentially improve the throughput of the design.

4.2.2.3 PixelShuffle Layer

The third Conv layer is followed by a PixelShuffle layer instead of an Activation layer as described in the previous section. This way, the Sub-Pixel Convolutional layer analyzed in section 1.3.3.1 is implemented. The dataflow logic follows the same logic as the on illustrated in the block diagram 4.7 of the activation layer with the difference that here we only need the untiling and the PixelShuffle operations.

The untile operation is the same as described in the previous section while the PixelShuffle operation implements the data rearrangement illustrated in figure 1.7:

```
void pxlshfl(float* input, float* output)
{
  for(int i = 0; i < L3_IM_H; i++){
    for(int j = 0; j < L3_IM_W; j++){
      output[2*OUT_W*i+2*j] = input[L3_IM_W*i+j];
      output[2*OUT_W*i+2*j+1] = input[L3_IM_W*i+j+L3_OUT_W];
      output[2*OUT_W*i+2*j+OUT_W] = input[L3_IM_W*i+j+2*L3_OUT_W];
      output[2*OUT_W*i+2*j+OUT_W+1] = input[L3_IM_W*i+j+3*L3_OUT_W];
    }
  }
}
```

Listing 4.14: The PixelShuffle operation

```
extern "C" {
void pixelshuffle(hls::stream<float>& x0, ..., hls::stream<float>& x15,
        hls::stream<float>& y0)
{

  #pragma HLS INTERFACE axis port = x0
    ...
  #pragma HLS INTERFACE axis port = x15

  #pragma HLS INTERFACE axis port = y0

    #pragma HLS INTERFACE ap_ctrl_none port=return


  float conv3_out[L3_OUT_SIZE], unt1_l3[L3_OUT_SIZE], unt2_l3[L3_OUT_SIZE],
  out[L3_OUT_SIZE];

```

```
  /* array partition to create parallel read/write ports */
  #pragma HLS array_partition variable=conv3_out block factor=16 dim=1
  #pragma HLS array_partition variable=out block factor=16 dim=1

  l3_read:
  read_input3(x0, ..., x15, conv3_out);

  l3_untiling4x4:
  untile3(conv3_out, L3_OUT_H*(L3_OUT_W/L3_OUT_TILE_W), L3_OUT_TILE_W, 4, 4,
    unt1_l3);

  l3_untiling4x16:
  untile3(unt1_l3, L3_OUT_H, L3_OUT_W, L3_OUT_TILE_H, L3_OUT_TILE_W, unt2_l3);

  l3_pxlshfl:
  pxlshfl(unt2_l3, out);

  l3_write:
  write_output3(y0, out);
}
}
```

Listing 4.15: The PixelShuffle layer

4.2.3 Hardware Link

The AI Engine graph compilation results in a libadf.a binary whereas the C/C++ PL kernels are compiled into Xilinx object (XO) files. After that, the Vitis linker takes its turn. As described in section 3.3.2 and illustrated in figure 3.2, using the v++ link command the aforementioned compiled binaries are linked with the platform to create the platform file (XSA), used to package the design.

When an AI Engine kernel application is present in the design, the Vitis linker requires some instruction on how to connect the PL kernels to the AIE graph. For this reason, a configuration file is provided with an additional [connectivity] section:

```
[clock]
/* Define the clock frequencies for each PL kernel */
freqHz=312500000:mm2s_1.ap_clk
freqHz=260000000:activation_1.ap_clk
freqHz=260000000:activation2_1.ap_clk
freqHz=260000000:pixelshuffle_1.ap_clk
freqHz=312500000:s2mm_1.ap_clk

[connectivity]
/* Connect the mm2s datamover with the 1st Conv layer */
stream_connect=mm2s_1.s0:ai_engine_0.DataInL1_0
...
stream_connect=mm2s_1.s3:ai_engine_0.DataInL1_3
/* Connect the output of the 1st Conv layer with the 1st Activation layer */
stream_connect=ai_engine_0.DataOutL1_0:activation_1.x0
...
stream_connect=ai_engine_0.DataOutL1_15:activation_1.x15
/* Connect the output of the 1st Activation layer with the 2nd Conv layer */
stream_connect=activation_1.y0:ai_engine_0.DataInL2_0
...
stream_connect=activation_1.y31:ai_engine_0.DataInL2_31
```

```
/* Connect the output of the 2nd Conv layer with the 2nd Activation layer */
stream_connect=ai_engine_0.DataOutL2_0:activation2_1.x0
...
stream_connect=ai_engine_0.DataOutL2_127:activation2_1.x127
/* Connect the output of the 2nd Activation layer with the 3rd Conv layer */
stream_connect=activation2_1.y0:ai_engine_0.DataInL3_0
...
stream_connect=activation2_1.y15:ai_engine_0.DataInL3_15
/* Connect the output of the 3rd Conv layer with the PixelShuffle layer */
stream_connect=ai_engine_0.DataOutL3_0:pixelshuffle_1.x0
...
stream_connect=ai_engine_0.DataOutL3_15:pixelshuffle_1.x15
/* Connect the output of the PixelShuffle layer with the s2mm datamover */
stream_connect=pixelshuffle_1.y0:s2mm_1.s0
```

Listing 4.16: System configuration file

The frequencies defined are the maximum frequencies that the PL kernels can afford. The stream_connect option defines AXI4-Stream connections between the ports of the AI Engine graph and the streaming ports of the PL kernels.

4.2.4 PS Code: The Host Application

Our design uses the embedded processing system (PS) as an external controller to orchestrate the data movements between the AI Engine graph and PL kernels. The PS host application is written in C/C++, using API calls to control the initialization, running, and closing of the AI Engine graph and the PL kernels. In Linux operating systems, the ADF API controls the AI Engine graph. The Xilinx Runtime (XRT) API is used to control PL kernels. However, Xilinx provides an OpenSource XRT C/C++ API that can also be used for controlling the execution of the AI Engine graph, when programming the host code for Linux.

Thus, the PS host application (host_xrt.cpp) does the following:

- Includes all the necessary header files for the XRT C API control flow.

- Includes the graph.h file and instantiates a layersGraph ADF graph object.

- Opens the device and loads the XCLBIN binary file:

```
const char* xclbinFilename = argv[1];
auto dhdl = xrtDeviceOpen(0);
auto xclbin = load_xclbin(dhdl, xclbinFilename);
auto top = reinterpret_cast<const axlf*>(xclbin.data());
```

- Allocate buffers for input data and results in global memory and initializes the input data in global memory:

```
xrtBufferHandle in_bohdl0 = xrtBOAlloc(dhdl, input_size_in_bytes, 0, 0);
auto in_bomapped0 = reinterpret_cast<float*>(xrtBOMap(in_bohdl0));
memcpy(in_bomapped0, input_img, input_size_in_bytes);

xrtBufferHandle out_bohdl = xrtBOAlloc(dhdl, output_size_in_bytes, 0, 0);
auto out_bomapped = reinterpret_cast<float*>(xrtBOMap(out_bohdl));
```

- Opens the PL datamovers and obtains kernel handles. Then, obtains the corresponding run handles, set the kernels' arguments and starts the kernels:

```
xrtKernelHandle mm2s_khdl = xrtPLKernelOpen(dhdl, top->m_header.uuid,
                                            ,"mm2s:{mm2s_1}");
xrtKernelHandle s2mm_khdl = xrtPLKernelOpen(dhdl, top->m_header.uuid,
                                            ,"s2mm:{s2mm_1}");

xrtRunHandle mm2s_rhdl = xrtRunOpen(mm2s_khdl);
int rval = xrtRunSetArg(mm2s_rhdl, 0, in_bohdl0);
rval     = xrtRunSetArg(mm2s_rhdl, 5, INPUT_TILE_SIZE);
rval     = xrtRunSetArg(mm2s_rhdl, 6, NO_IMAGES);

xrtRunHandle s2mm_rhdl = xrtRunOpen(s2mm_khdl);
rval = xrtRunSetArg(s2mm_rhdl, 0, out_bohdl);
rval = xrtRunSetArg(s2mm_rhdl, 2, OUTPUT_IMG_LENGTH);

xrtRunStart(mm2s_rhdl);
xrtRunStart(s2mm_rhdl);
```

- Opens the graph, obtains handle, resets and executes the graph:

```
adf::registerXRT(dhdl, top->m_header.uuid);

auto graphHandle = xrtGraphOpen(dhdl, top->m_header.uuid, "mmul_graph");

int ret = xrtGraphReset(graphHandle);

ret = xrtGraphRun(graphHandle, GRAPH_ITER_CNT);
```

- Waits for the datamovers' execution to finish and closes the run and kernel handles:

```
auto state_mm2s = xrtRunWait(mm2s_rhdl);
auto state_s2mm = xrtRunWait(s2mm_rhdl);

xrtRunClose(mm2s_rhdl);
xrtKernelClose(mm2s_khdl);
xrtRunClose(s2mm_rhdl);
xrtKernelClose(s2mm_khdl);
```

- Verifies the output results and releases the allocated resources:

```
xrtBOFree(in_bohdl0);
xrtBOFree(out_bohdl);

xrtGraphClose(graphHandle);

xrtDeviceClose(dhdl);
```

In addition to the PS host application, the AI Engine control code must also be compiled. This control code (aie_control_xrt.cpp) is generated by the AI Engine compiler when compiling the AI Engine design graph and kernel code. The AI Engine control code is used by the PS host application to do the following:

- Control the initial loading of the AI Engine kernels.
- Run the graph for several iterations, update the run time parameters associated with the graph, exit, and reset the AI Engine tiles.

4.3 Hardware-Agnostic Vitis AI Implementation

The ESPCN model was also implemented using the Vitis AI software, which can automatically compile several different neural network models to run on Xilinx DPUs. As our baseline CPU implementation of the network was developed in PyTorch, we will follow the Vitis AI PyTorch development flow:

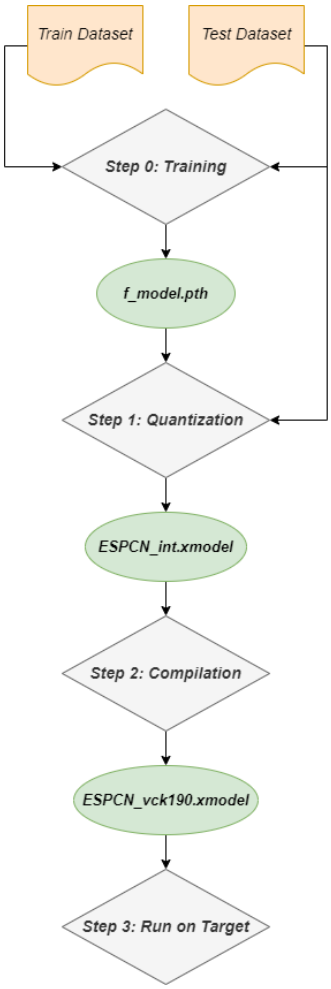


Figure 4.8: Vitis AI PyTorch development flow

The Vitis AI software resides into a docker image. The following sections analyze the steps that the aforementioned flow includes.

4.3.1 Training

The first step is the training and evaluation of the model. We train the ESPCN model again, with the same configuration as the Google Colab training done for the HW implementation described above. The output model is saved as a 'f_model.pth' file. This will be our floating-point model.

4.3.2 Quantization

The Xilinx DPU accelerators execute models that have their parameters in integer format, so, the next step will be the quantization of the trained, floating-point model. The Vitis AI quantizer takes this model as input and performs pre-processing (folds batchnorms and removes nodes not required for inference), and then quantizes the weights, biases and activations to the given bit width. This tool is included in the vai_q_pytorch python package and should be imported:

```
from pytorch_nndct.apis import torch_quantizer
```

To capture activation statistics and improve the accuracy of quantized models, the Vitis AI quantizer must run several iterations of inference to calibrate the activations. A calibration image dataset is, therefore, required. After calibration, the quantized model is transformed into a DPU deployable model, or else an XMODEL file.

```
# load trained model
model = Net(upscale_factor).to(device)
model.load_state_dict(torch.load(os.path.join(float_model,'f_model.pth')))

quantizer = torch_quantizer(quant_mode, model, (rand_in), output_dir=quant_model)
quantized_model = quantizer.quant_model

# data loader
test_set = get_test_set(upscale_factor)
test_loader = torch.utils.data.DataLoader(dataset=test_set,
                                          batch_size=batchsize,
                                          shuffle=False)
# evaluate
test(quantized_model, device, test_loader)

# export config
if quant_mode == 'calib':
    quantizer.export_quant_config()
if quant_mode == 'test':
    quantizer.export_xmodel(deploy_check=False, output_dir=quant_model)
```

Inspector

Before quantizing the floating-point model, we can optionally use the "inspector". The inspector is a tool provided by Vitis AI used to inspect the model before quantizing it. Inspector will output the device mapping information, indicating which operators will run on which part of the hardware platform, on the DPU or on the CPU. Principally, DPU is faster than CPU, so the optimal will be to run as many operators as possible on DPU devices. The output for the ESPCN model in a PNG format, reports:

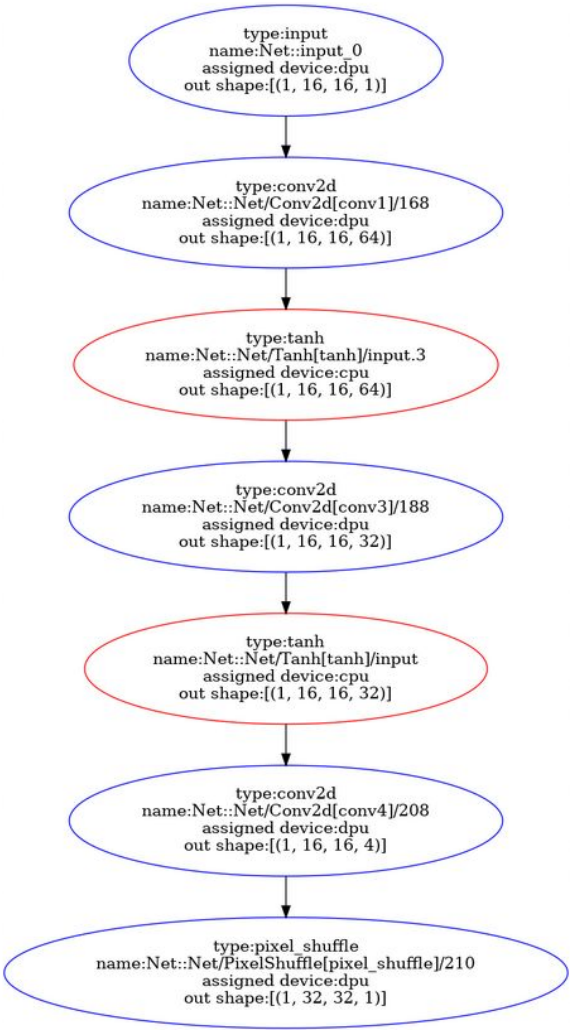


Figure 4.9: Inspector's output

As we can see the Tanh operation is not supported by the DPU and it is assigned on the CPU. Therefore, we will have to implement it separately in the application code.

4.3.3 Compilation

Xilinx Intermediate Representation (XIR) is a graph-based intermediate representation of the AI algorithms which is designed for compilation and efficient deployment on the DPU. The XIR-based Vitis AI compiler (vai_c_xir) takes the quantized model as input and after transforming it into the XIR format, it applies various optimizations to the graph. Then, breaks up the graph into several subgraphs on the basis of whether the operation can be executed on the DPU or not. For the DPU subgraphs, the compiler generates the instruction stream and attaches to it. Finally, the optimized graph with the necessary information and instructions for the Vitis AI Runtime (VART) is serialized into a compiled XMODEL file.

4.3.4 Run on Target

To run the application on the target VCK190 board, we need to copy the input images, XMODEL and application code on it. The application code is written in Python and implements

the following functionality:

- Opens the input XMODEL, deserializes it into a XIR graph object and gets a list of subgraphs:

```
g = xir.Graph.deserialize(model)
subgraphs = g.get_root_subgraph().toposort_child_subgraph()

dpu_subgraph0 = subgraphs[0]
...
dpu_subgraph6 = subgraphs[6]
```

- Creates the DPURunner objects that control the execution of the subgraphs that will run on the DPU:

```
dpu_1 = vart.Runner.create_runner(dpu_subgraph1, "run")
dpu_3 = vart.Runner.create_runner(dpu_subgraph3, "run")
dpu_5 = vart.Runner.create_runner(dpu_subgraph5, "run")
```

- Pre-processes the input images, i.e., convert the images to the YCbCr representation and keeps only the Y channel.

- Gets the input and output tensors from the DPURunner objects and initializes appropriately the batch size supported by the DPU target on-board and the output tensors:

```
inputTensor_1 = dpu_1.get_input_tensors()
outputTensor_1 = dpu_1.get_output_tensors()
inputTensor_3 = dpu_3.get_input_tensors()
outputTensor_3 = dpu_3.get_output_tensors()
inputTensor_5 = dpu_5.get_input_tensors()
outputTensor_5 = dpu_5.get_output_tensors()

input_ndim1 = tuple(inputTensor_1[0].dims)
output_ndim1 = tuple(outputTensor_1[0].dims)
input_ndim3 = tuple(inputTensor_3[0].dims)
output_ndim3 = tuple(outputTensor_3[0].dims)
input_ndim5 = tuple(inputTensor_5[0].dims)
output_ndim5 = tuple(outputTensor_5[0].dims)

batchSize = input_ndim1[0]

out1 = np.zeros([batchSize,16,16,64], dtype='float32')
out3 = np.zeros([batchSize,16,16,32], dtype='float32')
out5 = np.zeros([batchSize,32,32,1], dtype='float32')
```

- Initializes with zeros the input buffers and executes the subgraphs:

```
execute_async(dpu_1, {"Net__input_0_fix":inputData[0], "
    Net__Net_Conv2d_conv1__168_fix": out1})

inp2 = out1.copy()
out2 = Tanh(inp2)

execute_async(dpu_3, {"Net__Net_Tanh_tanh__input_3_fix":out2, "
    Net__Net_Conv2d_conv2__188_fix": out3})

inp4 = out3.copy()
out4 = Tanh(inp4)
```

```
10
11 execute_async(dpu_5, {"Net__Net_Tanh_tanh__input_fix":out4, "
       Net__Net_PixelShuffle_pixel_shuffle__210_fix": out5})
12
13 nn_out = out5.copy()
```

Note here that the Tanh functionality is implemented separetely to run on the CPU:

```
1 def Tanh(xx):
2     x = np.asarray( xx, dtype="float32" )
3     t=(np.exp(x)-np.exp(-x))/(np.exp(x)+np.exp(-x))
4     return t
```

- Post-processes the output and calculates the achieved PSNR.

Chapter 5

Experimental Evaluation & Results

This Chapter is dedicated to the evaluation of the proposed implementations. This evaluation takes into consideration three metrics:

- Image Quality
- Latency & Throughput
- Power Consumption

The following sections will provide detailed information about how the evaluation took place, present the results and discuss the overall performance, based on each one of the aforementioned metrics.

5.1 Image Quality Results

The ESPCN model was evaluated using the PSNR metric, as defined in 1.3.3. The evaluation dataset was the Set5 dataset, consisting of 5 images ("baby", "bird", "butterfly", "head", "woman") commonly used for testing performance of Image Super-Resolution models.

(a) Baby **(b)** Bird **(c)** Butterfly **(d)** Head **(e)** Woman

Figure 5.1: Set5 images.

In order to prepare the 16×16 image required for the model's input, we take a 32×32 crop of an "interesting" part of the dataset's images and then use the resize() function from OpenCV to perform a bicubic downscaling with a factor of 2. The following table summarizes the PSNR results in dBs for the three available implementations:

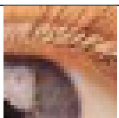
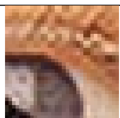
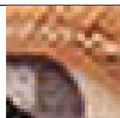
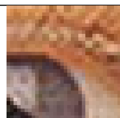
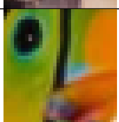
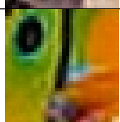
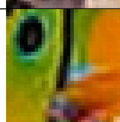
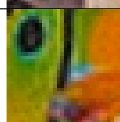
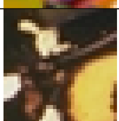
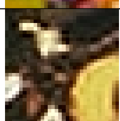
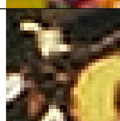
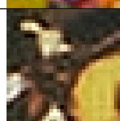
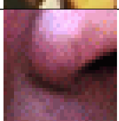
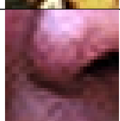
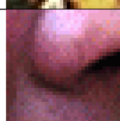
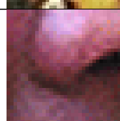
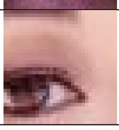
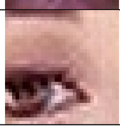
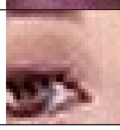
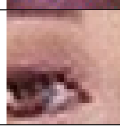
Table 5.1: PSNR Results

Image	CPU Implementation	HW-Specific Implementation	Vitis AI Implementation
baby	24.58 dB	24.56 dB	17.11 dB
bird	27.62 dB	27.55 dB	15.49 dB
butterfly	22.78 dB	22.77 dB	12.42 dB
head	30.74 dB	30.61 dB	14.37 dB
woman	26.14 dB	26.10 dB	15.94 dB

As we can see, our custom, hardware-specific implementation achieves great results, really close to the CPU baseline, with a maximum deviation of about **0.4%**. This deviation is due to the LUT-based implementation of the Tanh activation function. The Vitis AI implementation on the other hand, has an overly bad performance. Although we expected some degradation in accuracy due to the quantization step, the deviation here is huge. Xilinx provides a disclaimer stating that "For some networks such as Mobilenet, the accuracy loss might be large". However, fast fine-tuning algorithms or quantization aware training (QAT) are recommended to possibly improve the accuracy of quantized models.

In the table below we present the image quality results. The HR Images indicate the target images, that are a 32×32 crop of the original images presented in figure 5.1. The CPU SR images are the baseline software model's output, the HW SR images are the output of the proposed hardware-specific implementation, while the VAI SR images are the Vitis-AI output images. At this point, the author wants to put a disclaimer that, due to the small image dimensions even the HR target images are not fulfilling the human visual expectations. Limited to so small images we present this results only as a proof of concept.

Table 5.2: Image Quality Results

HR Image	CPU SR Image	HW SR Image	VAI SR Image
			
			
			
			
			

5.2 Latency & Throughput Results

In order to compare the performance of the implementations present in this thesis in terms of execution time, we consider only the inference time. For the CPU baseline we execute the evaluation script on the dual-core Arm Cortex-A72 processor present in the PS part of the VKC190 board (section 2.2.3). To measure the inference time we use the *time* Python package:

```
import time

start = time.time()
out = model(input)
end = time.time()
total = end - start
```

Using the same package we are going to measure inference time on the Vitis AI implementation too:

```
start = time.time()
execute_async(dpu_1, {"Net__input_0_fix":inputData[0],
                      "Net__Net_Conv2d_conv1__168_fix": out1})

inp2 = out1.copy()
out2 = Tanh(inp2)

execute_async(dpu_3, {"Net__Net_Tanh_tanh__input_3_fix":out2,
                      "Net__Net_Conv2d_conv2__188_fix": out3})

inp4 = out3.copy()
out4 = Tanh(inp4)

execute_async(dpu_5, {"Net__Net_Tanh_tanh__input_fix":out4,
                      "Net__Net_PixelShuffle_pixel_shuffle__210_fix": out5})

nn_out = out5.copy()
end = time.time()
total = end - start
```

For the hardware design, we use the std::chrono library:

```
auto time_start = std::chrono::steady_clock::now();
xrtRunStart(mm2s_rhdl);
xrtRunStart(s2mm_rhdl);

ret = xrtGraphRun(graphHandle, GRAPH_ITER_CNT);

auto state_mm2s = xrtRunWait(mm2s_rhdl);
auto state_s2mm = xrtRunWait(s2mm_rhdl);

auto time_stop = std::chrono::steady_clock::now();
std::chrono::duration<double> elapsed_seconds= time_stop - time_start;
```

To begin with, we measure the inference time for 1 image to determine latency:

Table 5.3: Latency Results

Implementation	Latency (ms)	Frames per Second (FPS)
CPU	5,5	181,82
Vitis AI	16,5	60,61
Custom HW-Specific	3,7	270,27

Also, the diagram below shows the throughput measurements:

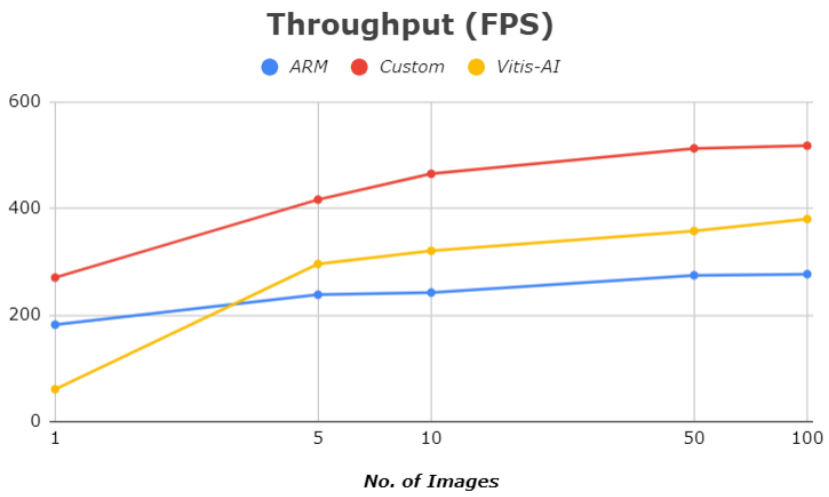


Figure 5.2: Throughput measurements

The first thing to notice here, is that the Vitis AI implementation has a significantly lower performance compared to both the CPU baseline and the hardware implementation regarding the system's latency. This is an expected behaviour due to the Tanh incompatibility with the DPU. As described in Section 4.3, to implement the network the Tanh operation was assigned to the CPU. Thus, the back and forth between the DPU and the CPU brought a great overhead in the performance of the design. However, the Vitis AI implementation's throughput scales better than the CPU baseline, not exceeding of course the hardware design's performance.

The custom, hardware-specific implementation on the contrary, is performing really well, both in terms of latency and throughput. We gain approximately a **1,5x** speedup compared to the CPU baseline and **4,5x** compared to Vitis AI when speaking about latency. Furthermore, as figure 5.2 shows, our implementation outperforms both CPU's and Vitis' AI throughput, reaching **518** FPS.

5.3 Power Results

Power efficiency is an important aspect, especially for edge applications. Xilinx has released a power tool designed to showcase the power features of the Versal ACAP devices. It is called Power Advantage Tool (PAT) and consists of:

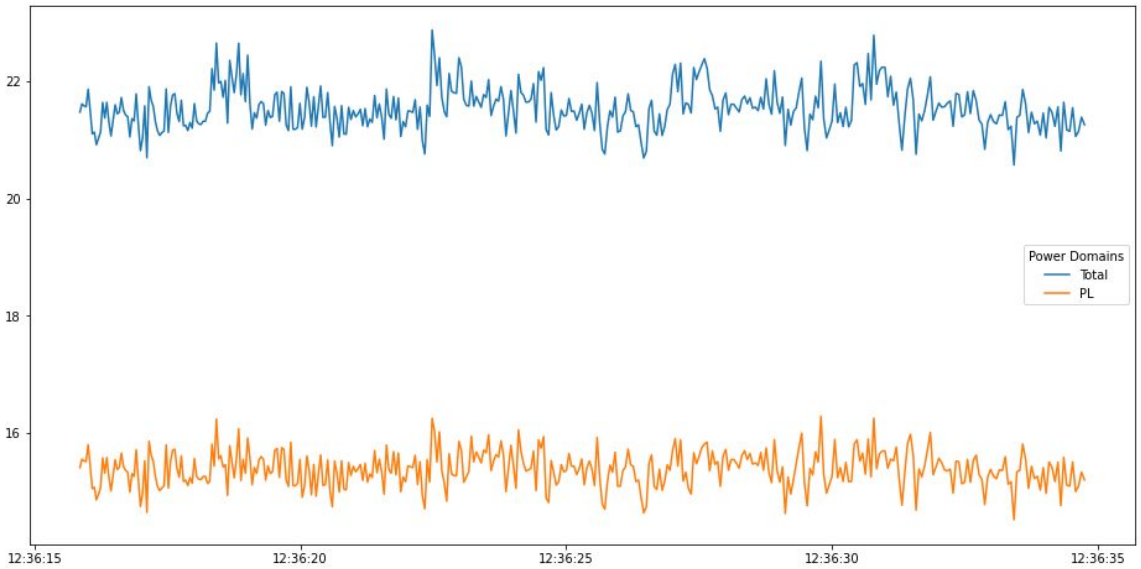
1. The poweradvantage.py Python library that provides a portable power measurement system
2. A Jupyter Notebook Python code named power_advantage_tool.ipynb that provided extendable usage examples of the poweradvantage.py library.

The PAT comes pre-installed as part of the Board Evaluation and Management (BEAM) tool. BEAM is a web-based GUI application connected to the System Controller's webserver that allows users to monitor and control the board by taking measurements or modifying parameters such as clocks, voltages or power.

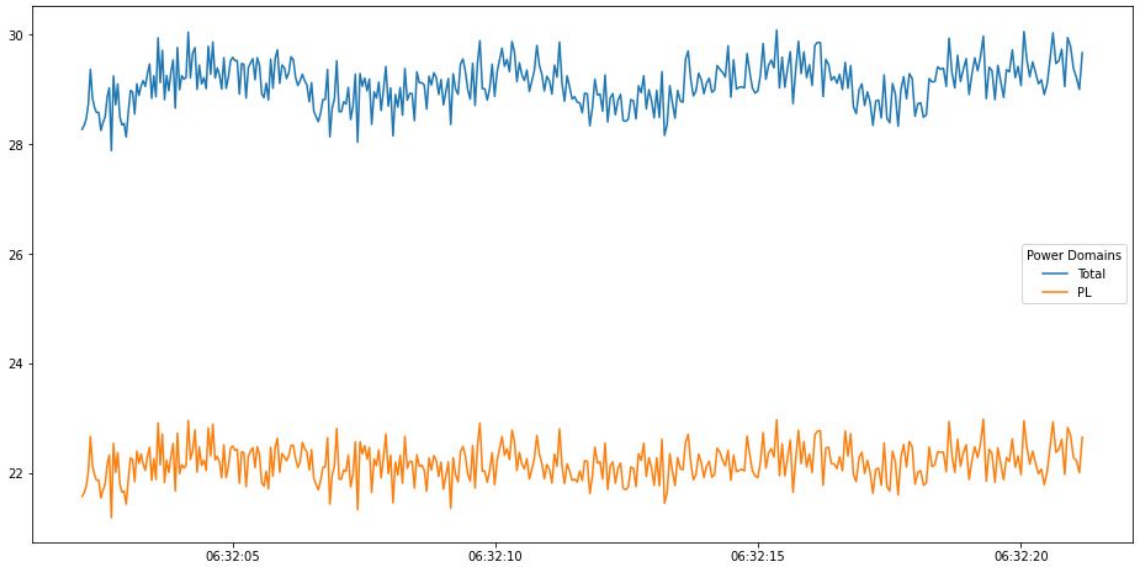
To test the power consumption of our application we utilize the PAT, using its Jupyter Notebook code. We can take measurements from different power domains and power islands on the board. We choose to measure the total power consumption to make an overall evaluation and the power cosumption of the PL power domain that includes the programmable logic, PL GTs, the coherent PCIe module (CPM4) and the AI Engines. We modify the design to run for 500 images in order to have a sufficient AIE running time to measure power. We also modify the given code to increase its sample frequency and plot the total power:

```
1  from poweradvantage import poweradvantage
2  import time
3  ...
4
5  pa = poweradvantage("VCK190", "SC")
6  df = pd.DataFrame()
7  name = pa.getname()
8
9  for i in range(400):
10     dt_object = datetime.fromtimestamp(datetime.timestamp(datetime.now()))
11     power = pa.getpower()
12     df0 = pd.DataFrame.from_records(zip(*power),columns=name,index=[0,0,dt_object])
13     df = df.append(df0.iloc[[2]].copy())
14
15 df_total=df.sum(axis='columns')
16 figure(figsize=(16, 6))
17 plt.plot(df_total)
18 plt.show()
```

The power measurement takes place in line 11 and the time required for 1 iteration is approximately 0,04 seconds. We perform 400 iterations so we have approximately 16 seconds of measurements. We execute the above code and run our application 4 times on the board. Each inference for 500 images takes 0,98 seconds for the HW-specific implementation and 1,3 seconds for the Vitis AI implementation, so the 16 seconds for measurements are more than enough. The power diagrams we get are the following:



(a) HW-specific implementation



(b) Vitis AI implementation

Figure 5.3: Power measurements.

In the 5.3a diagram we can see 4 spikes that correspond to the 4 executions of our HW-specific application. Respectively, the 4 executions of the Vitis AI application correspond to the 4 curves observed in the 5.3b diagram.

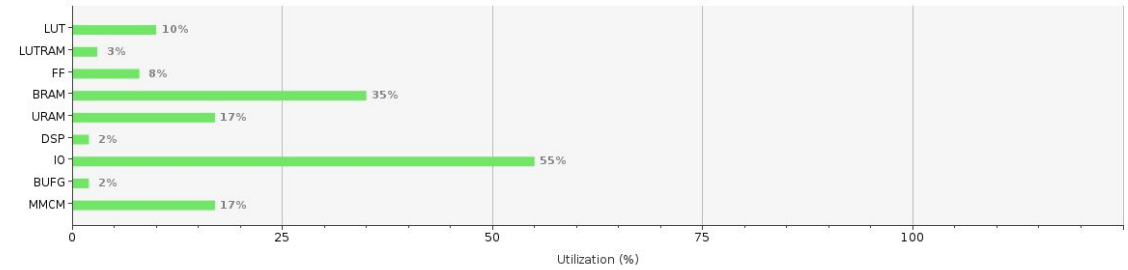
First of all, we can observe that no power management techniques (like clock-gating) are present by default on the board. In both cases we don't have extreme deviations from the idle state's power, especially in the AI Engines (PL domain). It is also noticeable that when the DPU is present on the board the total power is a bit higher.

5.4 Resource Utilization

Using the vitis_analyzer tool we can inspect the resource utilization on the PL, from the PL kernels, and using the XPE (Xilinx Power Tool) we can do the same for the AI Engine:

Utilization		
Core	160	40%
PL Stream	232	42%
NoC Stream	1	3%

Figure 5.4: AI Engine Resource Utilization



Resource	Utilization	Available	Utilization %
LUT	92654	899840	10.30
LUTRAM	13474	449920	2.99
FF	136430	1799680	7.58
BRAM	341.50	967	35.32
URAM	81	463	17.49
DSP	30	1968	1.52
IO	382	692	55.20
BUFG	15	980	1.53
MMCM	2	12	16.67

Figure 5.5: PL Utilization

Regarding the AIE utilization we must note that regarding the AIE Array, as described in 4.2.1.4, $16 + 128 \times 2 + 16 = 288$ AIE tiles are utilized for memory purposes. Although, there is still room for more aggressive parallelization. In addition, we achieve low resource utilization on the PL to also be able to increase the PL clock frequencies.

Chapter 6

Epilogue

6.1 Conclusion

This thesis was an initial attempt to explore and leverage the capabilities of one of the Xilinx Versal ACAP devices in order to tackle the challenges of AI inference on the edge. Along with the Versal ACAP devices, Xilinx also launched Vitis AI, a development environment with a specialized toolkit to automate the design process of ML applications on such devices. On this end, our application scenario was the acceleration of an image super-resolution (SR) deep learning model on the VCK190 board with both a hardware-specific and a Vitis AI implementation. The image SR is a computer vision task of great importance if anyone considers its numerous real-world applications. The selected model was the ESPCN model, an efficient sub-pixel convolutional neural network.

The proposed, hardware-specific implementation covered all three major parts of the Versal ACAPs, that is, the AI Engines (AIE), the Programmable Logic (PL) and the Processing System (PS). In this manner, the best of these three worlds were combined to give an efficient, accelerated model. More specifically, this design achieved a throughput of about **518 FPS** for 500 16×16 input image with a latency speedup of **1,49x** and **4,46x** compared to the CPU baseline and the Vitis AI implementation respectively. The loss in PSNR compared to the baseline model was only 0,4% due to the Tanh LUT-based implementation. On the Vitis AI implementation, the network was INT-8 quantized and divided between the DPU (an optimized IP designed by Xilinx for the Versal ACAPs) and the CPU. The need for data transfers between these two cores, finally imposed a significant performance loss, added to the image quality losses due to quantization.

On a final note, the trade off between hardware-abstraction and performance remains. In this thesis, we concluded that the performance gains came with a design that required labor and hardware expertise. However, as David Patterson quotes "the last 50 years of computer architecture show that raising the HW/SW interface, architecture opportunities are created".

6.2 Future Work

As previously stated, this thesis also had an exploratory facet. Thus, there is still plenty of room for improvements. The author's suggestions are:

1. For the **hardware-specific implementation**:
    - Modify the design to work for larger images.
    - Quantize the network to INT-8 to save resources and increase the MACs/cycle.
    - More exhaustive design space exploration for the tiling scheme in order to achieve high performance and save resources.
    - Explore different approaches for the matrix multiplication (Mmul) partition on the AIE, to pursue further throughput scaling and less resource utilization
    - Experiment with larger AXI4 stream widths to improve the communication between the AIE and the PL.
    - Try to increase the PL frequency or use more resources.
2. For the **Vitis AI implementation**:
    - Perform fine-tuning or/and quantization aware training to possibly increase the performance of the quantized model.
    - Implement the Tanh activation layer in the programmable logic instead of the CPU to examine potential performance gains.

Bibliography

[1] [Online]. Available: https://www.researchgate.net/figure/A-biological-neuron-in-comparison-to-an-artificial-neural-network-a-human-neuron-b_fig2_339446790

[2] [Online]. Available: https://mriquestions.com/neural-network-types.html

[3] W. Shi, J. Caballero, F. Huszár, J. Totz, A. P. Aitken, R. Bishop, D. Rueckert, and Z. Wang, "Real-time single image and video super-resolution using an efficient sub-pixel convolutional neural network," 2016. [Online]. Available: https://arxiv.org/abs/1609.05158

[4] "An overview of espcn: An efficient sub-pixel convolutional neural network." [Online]. Available: https://medium.com/@zhuocen93/an-overview-of-espcn-an-efficient-sub-pixel-convolutional-neural-network-b76d0a6c875e

[5] "Versal ACAP AI Engine," Xilinx, Architecture Manual AM009, October 2021.

[6] "Versal ACAP AI Engine Programming Environment," Xilinx, User Guide UG1076, May 2022.

[7] [Online]. Available: https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html

[8] "Vitis AI User Guide," Xilinx, User Guide UG1414, June 2022.

[9] "An illustrated explanation of performing 2d convolutions using matrix multiplications." [Online]. Available: https://medium.com/@_init_/an-illustrated-explanation-of-performing-2d-convolutions-using-matrix-multiplications-1e8de8cd2544

[10] D. Amodei and D. Hernandez, "Ai an compute." [Online]. Available: https://openai.com/blog/ai-and-compute/

[11] D. P. J. Hennesy, *Computer Architecture: A Quantitative Approach*, 2019.

[12] "Vitis Unified Software Platform Documentation," Xilinx, User Guide UG1393, May 2022.

[13] D. Patterson, "The past is prologue: A new golden age for computer architecture." [Online]. Available: https://cra.org/wp-content/uploads/2018/07/2018_CRA_Snowbird_Keynote_Patterson.pdf

[14] S. Haykin, *Neural Networks and Learning Systems*. Pearson, 2009.

[15] "Convolutional neural networks for visual recognition (module 2)." [Online]. Available: https://cs231n.github.io/convolutional-networks/#conv

[16] “Versal ai core series vck190 evaluation kit.” [Online]. Available: https://www.xilinx.com/products/boards-and-kits/vck190.html

[17] “Superresolution using an efficient sub-pixel convolutional neural network - pytorch.” [Online]. Available: https://github.com/pytorch/examples/tree/main/super_resolution

[18] “Netron app.” [Online]. Available: https://netron.app

[19] “Versal AI Engine LeNet Tutorial.” [Online]. Available: https://github.com/Xilinx/Vitis-Tutorials/tree/2022.1/AI_Engine_Development/Design_Tutorials/01-aie_lenet_tutorial

[20] “AI Engine API User Guide.” [Online]. Available: https://www.xilinx.com/htmldocs/xilinx2022_1/aiengine_api/aie_api/doc/index.html

[21] “Versal: The First Adaptive Compute Acceleration Platform (ACAP),” Xilinx, White Paper WP505, September 2020.

[22] Z. Wang, J. Chen, and S. C. H. Hoi, “Deep learning for image super-resolution: A survey,” 2019. [Online]. Available: https://arxiv.org/abs/1902.06068

[23] “Vitis High-Level Synthesis User Guide,” Xilinx, User Guide UG1399, April 2022.

[24] D. Danopoulos, K. Anagnostopoulos, C. Kachris, and D. Soudris, “Fpga acceleration of generative adversarial networks for image reconstruction,” Jul 2021. [Online]. Available: http://dx.doi.org/10.1109/MOCAST52088.2021.9493361

[25] [Online]. Available: https://github.com/BVLC/caffe