ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΗΛΕΚΤΡΟΝΙΚΗΣ ΚΑΙ ΣΥΣΤΗΜΑΤΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

FPGA Acceleration of Multi-Temporal Change Detection on High Resolution Images

Μυλωνάκης Ε. Δημήτριος
Α.Μ. : 03115742

Επιβλέπων : Δημήτριος Ι. Σούντρης
Καθηγητής ΕΜΠ

Αθήνα
Μάρτιος 2022

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΗΛΕΚΤΡΟΝΙΚΗΣ ΚΑΙ ΣΥΣΤΗΜΑΤΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

FPGA Acceleration of Multi-Temporal Change Detection on High Resolution Images

Μυλωνάκης Ε. Δημήτριος
Α.Μ. : 03115742

Επιβλέπων : Δημήτριος Ι. Σούντρης
Καθηγητής ΕΜΠ

Τριμελής Επιτροπή Εξέτασης

(Υπογραφή)	(Υπογραφή)	(Υπογραφή)
.....	.....	.....
Δημήτριος Σούντρης	Διονύσιος Πνευματικάτος	Σωτήριος Ξύδης
Καθηγητής	Καθηγητής	Επίκουρος Καθηγητής
ΕΜΠ	ΕΜΠ	Χαροκόπειο Πανεπιστήμιο

Ημερομηνία Εξέτασης:
9 Μαρτίου 2022

Copyright © - All rights reserved Μυλωνάκης Δημήτριος, 2022.
Με επιφύλαξη παντός δικαιώματος.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ΄ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα.

Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

(Υπογραφή)

.....

Μυλωνάκης Δημήτριος
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.
©2022 - All rights reserved.

Περίληψη

Ο τομέας του Earth Observation αναπτύσσεται γρήγορα χάρη στην υιοθέτηση big data τεχνολογιών. Αυτό είναι δυνατό χάρη στις αποτελεσματικές υποδομές αποθήκευσης και επεξεργασίας δεδομένων, αλλά κυρίως χάρη στην ανάπτυξη εφαρμογών ανάλυσης δεδομένων με τεχνικές μηχανικής εκμάθησης.

Έχουν αναπτυχθεί αρκετές εφαρμογές που πραγματοποιούν ανίχνευση αλλαγών σε δορυφορικές εικόνες του Earth Observation με τη βοήθεια μοντέλων μηχανικής εκμάθησης. Η εξέταση των αλλαγών μιας καθορισμένης περιοχής σε μια χρονική περίοδο παράγει μεγάλο όγκο δεδομένων που οδηγεί σε υψηλές απαιτήσεις όσον αφορά τη γρήγορη πρόσβαση, την αποθήκευση και τον υπολογισμό. Σε αυτή τη διπλωματική εργασία, εστιάζουμε σε μια τέτοια εφαρμογή, η οποία δημιουργεί γενικούς χάρτες ανίχνευσης αλλαγών για ζεύγη διαδοχικών dot products του δορυφόρου Sentinel-2 που αντιπροσωπεύουν ακριβώς το ίδιο οπτικό πεδίο. Στόχος της διατριβής είναι η βελτίωση του χρόνου απόκρισης του εργαλείου χωρίς απώλεια ακρίβειας. Για το σκοπό αυτό, αξιοποιούμε την υπολογιστική ισχύ ετερογενών πόρων, στοχεύοντας το FPGA της Intel Strarix 10 και χρησιμοποιούμε την HLS OpenCL για να δημιουργήσουμε έναν αποτελεσματικό επιταχυντή. Η μελέτη μας υπολογίζει πρώτα το χρονικό προφίλ των εφαρμογών προκειμένου να εντοπίσει τα σημεία συμφόρησης απόδοσης. Στη συνέχεια, εστιάζουμε τις προσπάθειές μας για επιτάχυνση στη βελτιστοποίηση των σημείων συμφόρησης με τη βοήθεια ενσωματωμένων τεχνικών βελτιστοποίησης εργαλείων HLS καθώς και αρχιτεκτονικών και αλγοριθμικών βελτιστοποιήσεων. Χρησιμοποιούμε fine-grain and coarse-grain παραλληλισμό και εξερευνούμε τον χώρο σχεδιασμού για να εντοπίσουμε βελτιστοποιημένες αρχιτεκτονικές με διαφορετικούς στόχους, δηλαδή τόσο την επίδοση(performance) όσο και την παραγωγικότητα(throughput). Ο επιταχυντής είναι ενσωματωμένος στην αρχική εφαρμογή python και αξιολογείται σε πραγματικές εικόνες Sentilel-2. Η καλύτερη αρχιτεκτονική προσφέρει συνολική επιτάχυνση x7 σε σχέση με τη βασική γραμμή λογισμικού.

Λέξεις Κλειδιά — Πολυχρονική Ανίχνευση Αλλαγών, Κωδικοποίηση εικόνων, Intel Stratix 10, High Level Synthesis, OpenCL, Τεχνητά Νευρωνικά Δίκτυα

Abstract

The Earth Observation domain is rapidly flourishing thanks to the adoption of big data technologies. This is possible thanks to efficient data storage and processing infrastructures, but most importantly thanks to the development of data analytic applications with machine learning techniques. Several applications have been developed that perform change detection on Earth Observation satellite images with the help of machine learning models. Examining changes of a designated area over a period of time produces a big amount of data leading to demanding requirements in terms of fast access, storage and computation. In this diploma thesis, we focus on such an application, that creates generic change detection maps for pairs of time-consecutive Sentinel-2 data products that represent exactly the same field of view. The goal of the thesis is to improve the response time of the tool without loss in accuracy. For this purpose, we leverage the computing power of heterogeneous resources, targeting an Intel Strarix 10 FPGAs and utilize the OpenCL High Level Synthesis framework to create an efficient accelerator. Our study first performs application profiling in order to identify the performance bottlenecks. We then focus our acceleration efforts on optimizing the bottlenecks with the help of built-in HLS tool optimization techniques as well as architectural and algorithmic optimizations. We employ both fine-grain and coarse grain parallelism and explore the design space to identify architectures optimized towards different objectives, i.e both performance and throughput. The accelerator is integrated into the original python application and evaluated over real Sentilel-2 images. The best architecture delivers an overall speedup of x7 over the software baseline.

Keywords — Multi-Temporal Change Detection, image encoding, Intel Stratix 10, High Level Synthesis, OpenCL, artificial neural networks

Ευχαριστίες

Ευχαριστώ θερμά τον επιβλέποντα καθηγητή μου, κ. Δημήτριο Σούντρη, για την εμπιστοσύνη που μου έδειξε και την ευκαιρία να εκπονήσω αυτή τη διπλωματική εργασία μέσω του εργαστηρίου του MicroLab. Ένα μεγάλο ευχαριστώ στην υποψήφια διδάκτρορα κ. Κωνσταντίνα Κολιογεώργη και στον Επίκουρο Καθηγητή του Χαροκόπειου Πανεπιστημίου, κ. Σωτήριο Ξύδη, για την ακούραστη καθοδήγηση και την πολύτιμη βοήθεια που μου προσέφεραν. Θα ήθελα ακόμα να ευχαριστήσω τους φίλους μου που συνέβαλλαν στο να γίνουν τα φοιτητικά χρόνια ένα από τα πιο ευχάριστα ταξίδια της ζωής μου. Τέλος ένα τεράστιο ευχαριστώ στην οικογένεια μου για την ανιδιοτελή εμπιστοσύνη και υποστηρίξη που μου έδωσαν καθ΄ όλη την ακαδημαϊκή μου πορεία.

Contents

List of Figures

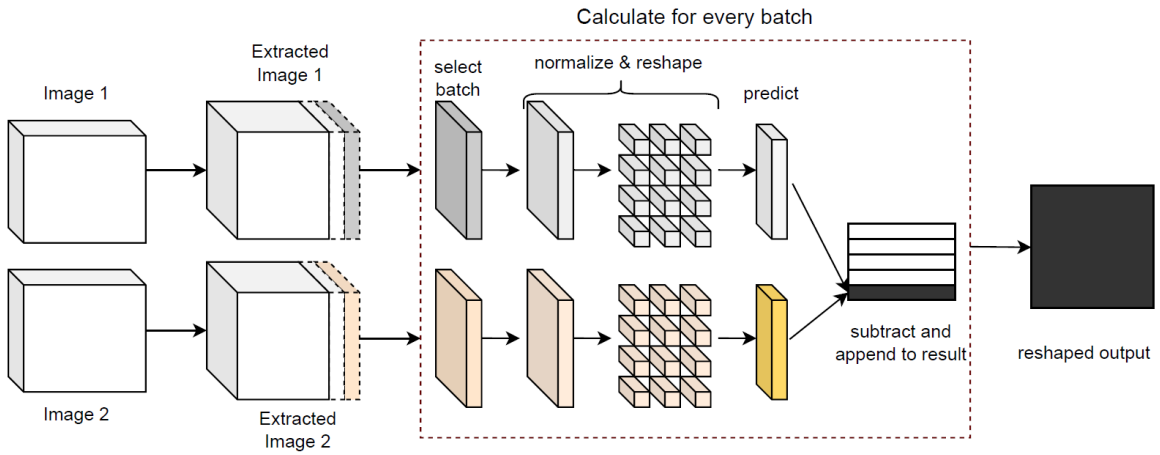
List of Tables

Εκτεταμένη Περίληψη

Επισκόπηση Προβλήματος

Το εργαλείο Ανίχνευσης Μεταβολών (Change Detection tool) είναι μια εφαρμογή που χρησιμοποιείται για την κωδικοποίηση και τον υπολογισμό της διαφοράς ανά στοιχείο ανάμεσα σε δύο εικόνες. Οι διαφορές αυτές υπολογίζονται με τη βοήθεια ενός προ-εκπαιδευμένου μοντέλου που βασίζεται στα τεχνητά νευρωνικά δίκτυα, το οποίο δρα σε μικρά κομμάτια της αρχικής εικόνας. Το μοντέλο απαρτίζεται απο δύο στρώματα επεξεργασίας που επιδρούν πάνω σε εικόνες διαστάσεων (5,5,4). Η πολυπλοκότητα του εν λόγω αλγορίθμου βρίσκεται στην παραγωγή ενός τεράστιου αριθμού αποτελεσμάτων και για εναν μεγάλο αριθμό εικόνων.

Κατά τη διάρκεια της εκτέλεσης της διαδικασίας παράγεται και μεταβάλλεται μεγάλος αριθμός δεδομένων. Μια σχηματική απεικόνιση της συνολικής διεργασίας αποτυπώνεται στην εικόνα 1. Αρχικά, οι δύο εικόνες φορτώνονται στον χρόνο εκτέλεσης απο την μνήμη. Στη συνέχεια, αφού ολοκληρωθεί η διαδικασία εξαγωγής κατά την οποία οι εικόνες αποκτούν υπολογιστικό βάθος, ξεκινά ο κύριος βρόχος επεξεργασίας. Μέσα σε εκείνον γίνεται επιλογή μια ομάδας εικόνων για κάθε μια από τις δύο αρχικές. Η ομάδα αυτά υπόκειται πρώτα σε μια διαδικασία κανονικοποίησης των τιμών της ενώ στη συνέχεια οι εικόνες ανασχηματίζονται σε μια μορφή, ικανή να προσπελαθεί και να επεξεργαστεί από το προ-εκπαιδευμένο νευρωνικό δίκτυο. Το μοντέλο αποτελείται από ένα στρώμα επιπεδοποίησης των δεδομένων, κατά το οποίο οι τρισδιάστατες εικόνες μετασχηματίζονται σε μονοδιάστατους πίνακες, καθώς και ένα Πυκνό Στρώμα (Dense Layer). Τα αποτελέσματα του Dense Layer κανονικοποιούνται μέσω μιας διεργασίας που ονομάζεται L2 Normalization προτού επιστρέψουν στο κυρίως πρόγραμμα. Τα επεξεργασμένα δεδομένα των ομάδων αφαιρούνται ανά στοιχείο και αποθηκεύονται σε έναν τελικό πίνακα. Η διαδικασία επαναλαμβάνεται μέχρις ότου όλες οι ομάδες των αρχικών εικόνων έχουν υπολογιστεί.



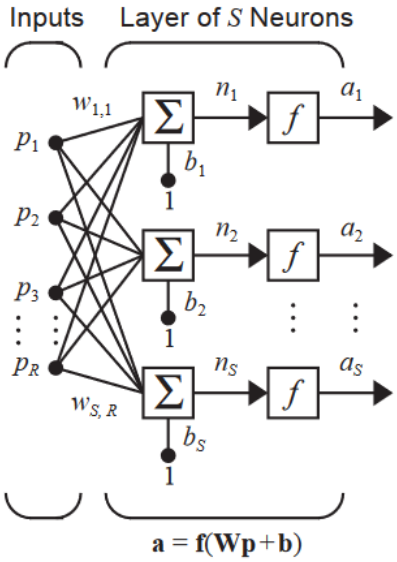
Εικόνα 1: Σχηματική Αναπαράσταση

Εξ΄ αιτίας του μεγάλου όγκου δεδομένων που πρέπει να επεξεργαστούν, η συνολική διεργασία γίνεται μια κοστοβόρα και χρονοβόρα διαδικασία. Ο κύριος βρόχος επεξεργασίας σπαταλά περίπου 74% του συνολικού χρόνου της εφαρμογής. Ο εναπομείνων χρόνος χρησιμοποείται στην εξαγωγή των εικόνων και την αποθήκευση τους στη μνήμη. Με βάση αυτές τις μετρήσεις, κρίνεται απαραίτητο να στρέψουμε την προσοχή μας στον κύριο βρόχο επεξεργασίας. Από το profiling της εφαρμογής αποκτήσαμε μια ακόμα ευρύτερη εικόνα για το διαμοιρασμό του χρόνου στις επιμέρους διαδικασίες. Η κανονικοποίηση που γίνεται κατά την αρχή του αλγορίθμου ευθύνεται για το 72% του χρονου του βρόχο και περίπου για το μισό συνολικό χρόνο της εφαρμογής. Το υπόλοιπο 28% δίδεται για το κομμάτι της πρόβλεψης κατα 23% και ένα 5% για ανασχηματισμό των ομάδων δεδομένων και λοιπές πράξεις.

Θεωρητικό Υπόβαθρο

Θεωρία στην Ανίχνευση Μεταβολών Εικόνας

Ένας από τους πλέον αποδοτικότερους τρόπους για ανίχνευση μεταβολών μεταξύ δύο ή περισσότερων εικόνων είναι μέσω των Τεχνητών Νευρωνικών Δικτύων (Artificial Neural Networks-ANN). Τα Νευρωνικά Δίκτυα προσπαθούν να μιμηθούν τον τρόπο με τον οποίο δουλεύει ο ανθρώπινος εγκέφαλος, δημιουργώντας ένα δίκτυο στο οποίο κάποιοι νευρώνες είναι συνδεδεμένοι με κάποιους άλλους έτσι ώστε να μπορούν να μεταδώσουν πληροφορίες και να πάρουν αποφάσεις. Με στόχο την ¨πρόβλεψη' μιας εικόνας, σε αυτή τη διπλωματική θα ασχοληθούμε μόνο με ένα περιορισμένο κομμάτι των ANN, στρέφοντας την προσοχή μας αποκλειστικά σε ένα πλέγμα από νευρώνες, ή Dense Layer, όπως χρησιμοποιείται από την βιβλιοθήκη της Python.



Εικόνα 2: Στρώμα Νευρώνων-Αναπαράσταση

Μία σχηματική αναπαράσταση του πλέγματος απο νευρώνες φαίνεται στην εικόνα 2 ενώ για τον υπολογισμό του καθ' ενός από τα αποτελέσματα a, γίνεται χρήση της συνάρτησης:

$$a = f\left(\left(\sum_{i=0}^{n-1} w_i p_i\right) + b\right)$$

Όταν τα αποτελέσματα του πολλαπλασιασμού έχουν υπολογιστεί, δρα μια *Συνάρτηση Ενεργοποίησης*(Activation Function) ώστε να μετατρέψει το γινόμενο στην τελική τιμή εξόδου. Στη δική μας περίπτωση γίνεται χρήση της συνάρτησης Leaky Rectified Linear Unit (Leaky ReLU) η οποία αντιμετωπίζει αποτελεσματικά το πρόβλημα των νεκρών κόμβων. Σαν τελευταίο βήμα της διαδικάσιας, γίνεται κανονικοποίηση των στοιχείων εξόδου με μια διαδικασία που ονομάζεται L2 Normalization και υλοποιείται μέσω της συνάρτησης:

$$f(x) = \begin{cases} 0.1x, & \text{if } x < 0 \\ x, & \text{otherwise} \end{cases}$$

Εργαλεία High Level Synthesis

Τα εργαλεία High Level Synthesis (HLS) δίνουν την ευκαιρία στον σχεδιαστή να δουλέψει σε ένα μεγαλύτερο επίπεδο αφαίρεσης ενώ καταφέρνουν να δημιουργήσουν αποδοτικό hardware. Παράλληλα προσφέρουν στους software developers έναν εύκολο τρόπο να επιταγχύνουν τα υπολογιστικά χρονοβόρο κομμάτια του αλγορίθμου τους πάνω σε Field Programmable Gate Array (FPGA) ή άλλες πλατφόρμες High Performance Computing. Τα FPGA προσφέρουν αρχιτεκτονικές με μεγάλα επίπεδα παραλληλοποίησης και οφέλη στην επίδοση, το κόστος και την ενέργεια σε σχέση με τους παραδοσιακούς επεξεργαστές

Η ροή χρήσης των HLS, στην περίπτωσή μας, το Intel FPGA SDK for OpenCL[2] αποτελείται από δύο μέρη: τον προγραμματισμό και παραγωγή του FPGA bitstream και το host πρόγραμμα που διαχειρίζεται την εφαρμογή καθώς και τον επιταχυντή. Ο κεντρικός πυρήνας, υπεύθυνος για την παραγωγή του bitstream είναι γραμμένος σε OpenCL και περνάει πρώτα από ένα στάδιο hardware compilation για να δημιουργηθεί ένα αρχείο το οποίο ΄διαβάζεται΄ από το host πρόγραμμα ώστε το τελευταίο να χρησιμοποιήσει το FPGA. Στην πλευρά του host, το πρόγραμμα είναι γραμμένο σε C/C++ και συνδέεται με τις αντίστοιχες βιβλιοθήκες της OpenCL. Ο σχεδιαστής συνθέτει τόσο τον κώδικα του πυρήνα, ο οποίος θα ¨τρέξει' στο FPGA, όσο και αυτόν του host, ο οποίος εκτελείται σε ενσωματωμένο επεξεργαστή γενικού σκοπού και ελέγχει τους υλοποιημένους πυρήνες του FPGA. Κοιτώντας από ένα υψηλότερο επίπεδο, ο host, χρησιμοποιεί το επίπεδο πλατφόρμας OpenCL API για να υποβάλει διεργασίες στις συσκευές καθώς και να διαχειριστεί το συνολικό φόρτο εργασίας κατά το πλάτος όλων των συσκευών και των διεργασιών. Στη διπλωματική μας εργασία, θα εκμεταλλευτούμε το υψηλό επίπεδο ολοκλήρωσης των εργαλειών αυτών καθώς και τις ενσωματωμένες στατηγικές βελτιστοποίησης που προσφέρουν για να δημιουργήσουμε αποδοτικό hardware μέσα από έναν ενδελεχή σχεδιαστικό χώρο εξερεύνησης (design space exploration).

Intel Stratix 10

Η Intel Stratix 10 GX 2800 είναι μια σύγχρονη συσκευή υψηλής επίδοσης που περιέχει περισσότερα απο 2,7 εκ. Στοιχεία Λογικής (Logic Elements-LEs). Τα LEs αποτελούν την μικρότερη μονάδα λογικής μέσα σε ένα FPGA και προσφέρουν προηγμένα χαρακτηριστικά. Επιπλέον, τα Intel FPGAs αποτελούνται απο Κομμάτια Προσαρμοζόμενης Λογικής (Adaptive Logic Modules-ALMs) τα οποία είναι δομικά στοιχεία σχεδιασμένα να βελτιστοποιούν την επίδοση και την κατανάλωση. Κάθε ALM, ενθυλακώνει προσαρμοζόμενα Lookup Tables(LUTs) έναν πλήρη αθροιστή των δύο bits (FA) καθώς και τέσσερις καταχωρητές. Ακόμα μία μονάδα της Stratix 10, είναι ομάδες λογικών δομών, τα, logic array blocks(LABs), τα οποία αποτελούνται απο πολλαπλά ALM. Τa LABs μπορούν να σχεδιαστούν έτσι ώστε να υλοποιούν οποιαδήποτε μορφής συνάρτηση που επιθυμεί ο χρήστης, όπως αριθμητικές ή συναρτήσεις καταχωρητών. Η συσκευή υποστηρίζει συνολικά 640 bits dual-port SRAM σε κάθε υλοποιημένο MLAB. Τέλος, ο Πίνακας 1 περιέχει συγκεντρωτικά τους διαθέσιμους πόρους της συσκευής, πάνω στους οποίους θα χτίσουμε την εφαρμογή μας.

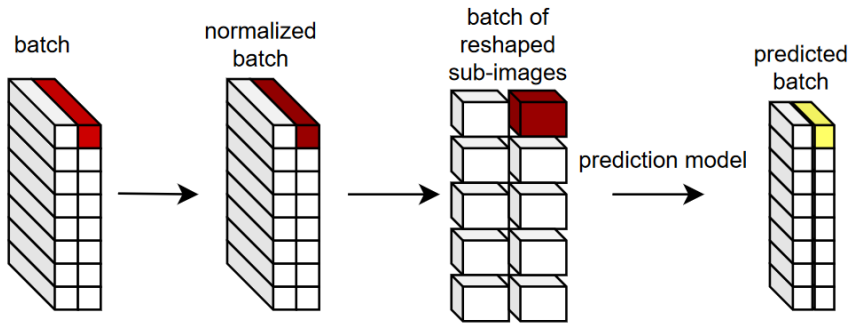
Πίνακας 1: Διαθέσιμοι Πόροι της Intel Stratix 10

LEs	ALMs	Ενσωματωμένη Μνήμη	DSPs
2753000	933120	244 Mb	5760

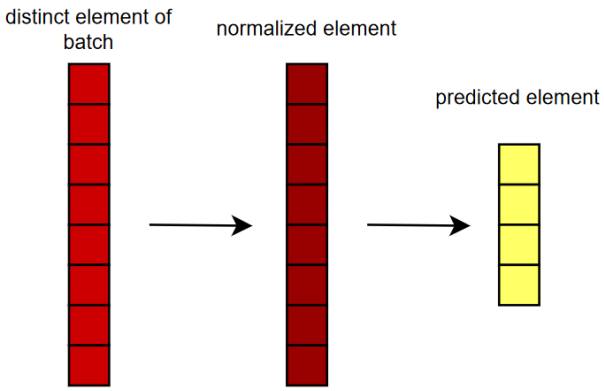
Αρχιτεκτονικές Βελτιστοποιήσεις μέσω HLS

Περιγραφή Μοντέλου Πρόβλεψης

Το πρώτο κομμάτι της εργασίας μας επικεντρώνεται στην προσέγγιση του προβλήματος από μια διαφορετική σκοπιά. Κατά την επίλυση του αλγορίθμου από λογισμικό, η κάθε εικόνα χωρίζεται σε υπο-ομάδες. Κάθε μια από αυτές είναι αντικείμενο μιας σειράς από μετασχηματισμούς έτσι ώστε να μπορεί να γίνει η επεξεργασία τους από τις βιβλιοθήκες που χρησιμοποιούνται. Η διαδικασία αυτή απεικονίζεται σχηματικά στην Εικόνα 3, εκεί όπου το κάθε στοιχείο της υπο-ομάδας μετατρέπεται από έναν πίνακα μεγέθους 100 σε έναν μεγέθους (5,5,4). Αυτό το στοιχείο στη συνέχεια είναι διαχειρίσιμο από το προ-εκπαιδευμένο μοντέλο πρόβλεψης. Μέσα στο μοντέλο, παρ΄ όλα αυτά, οι εικόνες ανασχηματίζονται ακόμα μία φορά σε έναν πίνακα 100 στοιχείων που χρησιμοποιούνται για την παραγωγή των 25 επιθυμητών αποτελεσμάτων. Αυτή η διαδικασία μπορεί να θεωρηθεί ως *Παραλληλισμός σε επίπεδο Ομάδας* αφού τα δεδομένα επεξεργάζονται σε μορφή ομάδας μέσω της χρήσης των βιβλιοθηκών της Tensorflow και της Python. Παρ΄ όλα αυτά, το κάθε αντικείμενο είναι ανεξάρτητο από τα υπόλοιπα, επομένως οι υπολογισμοί μπορούν να γίνουν σε κάθε ένα ξεχωριστά, όπως απεικονίζεται στην Εικόνα 4.



Εικόνα 3: Παρραλληλισμός σε επίπεδο Ομάδας(Software)



Εικόνα 4: Παραλληλισμός σε επίπεδο Στοιχείου(FPGA)

Η διαδικασία την οποία θα ακολουθήσουμε για τον σχεδιασμό του δικού μας μοντέλου πρόβλεψης συνοψίζεται επιγραμματικά παρακάτω:

1. **Αφαίρεση Μέσης Τιμής** : Κάθε μία από τις 100 τιμές του στοιχείου πρέπει να μειωθεί κατά την μέση τιμή του δείγματος. Οι υπολογισμοί για την εν λόγω διαδικασία είναι:

$$x_{mean} = x - \frac{1}{N}\sum x$$

   όπου το x αναφέρεται στις τιμές του δείγματος-εισόδου.

2. **Διαίρεση με Τυπική Απόκλιση** : Η διαδικασία συνεχίζεται με την διαίρεση των στοιχείων του δείγματος με την τυπική τους απόκλιση. Η μέση τιμή του δείγματος θα πρέπει να υπολογιστεί ξανά για το νέο δείγμα, όπως αυτο προέκυψε από το προηγούμενο βήμα.

$$x_{std} = \frac{x_{mean}}{\sqrt{\frac{\sum |x_{mean} - \mu|^2}{N}}}$$

3. **Dense Layer** : Εφόσον τα δεδομένα παραμένουν στον πυρήνα και δεν διενεργούνται εγγραφές στη μνήμη, τα δείγματα μπορούν να χρησιμοποιηθούν απευθείας σαν πίνακας 100 στοιχείων για το Dense Layer. Τα βήματα της εν λόγω διαδικασίας περιγράφηκαν παραπάνω στο τμήμα της θεωρίας.

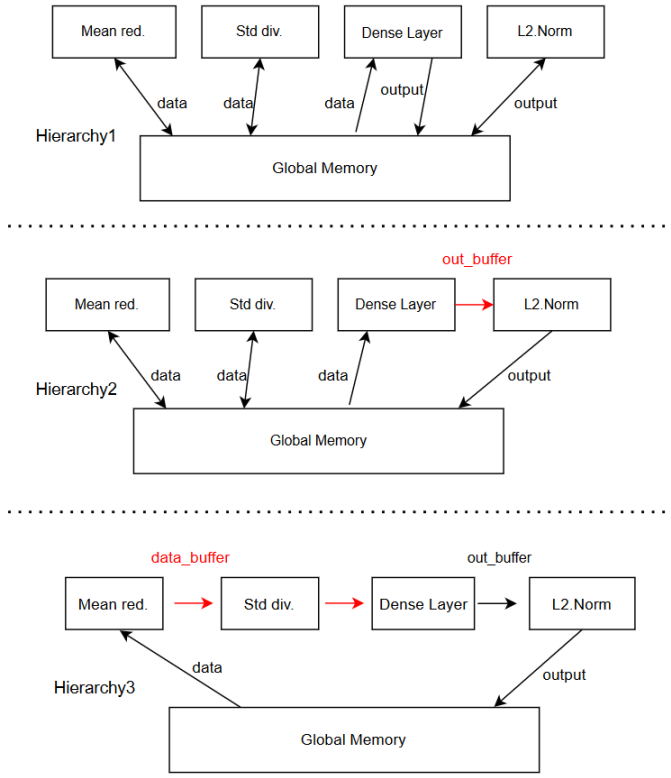
4. **L2 Normalization** : Η διαδικασία αυτή λαμβάνει χώρα όταν τα 25 αποτελέσματα έχουν παραχθεί απο το προηγούμενο στάδιο και αποτελεί άλλη μια μορφή κανονικοποίησης των δεδομένων.

Διαμόρφωση της Ιεραρχίας της Μνήμης

Μια πρωταρχική τεχνική βελτιστοποίησης που θα εφαρμόσουμε αφορά την επιλογή της βέλτιστης Ιεραρχίας Μνήμης. Αυτό το κομμάτι επικεντρώνεται στο κατά πόσο

η διασύνδεση με την κεντρική μνήμη μπορεί να αποτελέσει καθοριστικό παράγοντα στον περιορισμό της επίδοσης του πυρήνα.

Γενικά, ένα design υψηλής επίδοσης επωφελείται από την μεγιστιποίηση του εύρους ζώνης της μνήμης (memory bandwidth). Ο Intel compiler προσπαθεί να βρει την βέλτιστη τεχνική διασύνδεσης με την κύρια μνήμη, σύμφωνα σύμφωνα με το μοτίβο προσπέλασης στη μνήμη, όπως αυτό περιγράφεται στον κώδικα του πυρήνα, ώστε να δημιουργήσει μια Μονάδα Φόρτωσης Αποθήκευσης(Load Store Unit-LSU). Burst-Coalecsed μονάδες δημιουργούνται στην περίπτωσή μας, γεγονός που βελτιστοποιεί το bandwidth φορτώνοντας τα περισσότερα δυνατά δεδομένα που ο σχεδιασμός καθορίζει υπό το κόστος της χρησιμοποίησης περισσότερων πόρων. Μία καλή πρακτική βελτίωσης του σχεδιασμού είναι να μειώνουμε

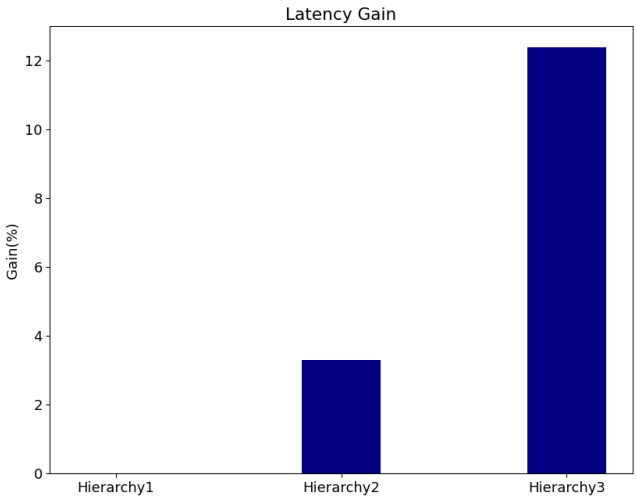


Εικόνα 5: Επισκόπηση Αρχιτεκτονικών-Ιεραρχία Μνήμης

το πλήθως των διασυνδέσεων των πυρήνων με την κύρια μνήμη με στόχο την μειωμένη κατανάλωση πόρων του συστήματος και την επιτάχυνση του design. Η εικόνα 5 δείχνει τον τρόπο που επιδιώξαμε να το επιτύχουμε στη συγκεκριμένη περίπτωση. Χρησιμοποιούνται εκτενώς καταχωρητές (buffers) ,που παραμένουν σαν μνήμες κοντά στον πυρήνα και αντικαθιστούν τις διασυνδέσεις με την κύρια μνήμη. Τα αποτελέσματα των αρχιτεκτονικών είναι διαθέσιμα παρακάτω:

Πίνακας 2: Σύγκριση στη χρήση Μνήμης

Design	**MLABs**	**RAM Blocks**
Hierarchy 1	201	425
Hierarchy 2	181	432
Hierarchy 3	1845	262

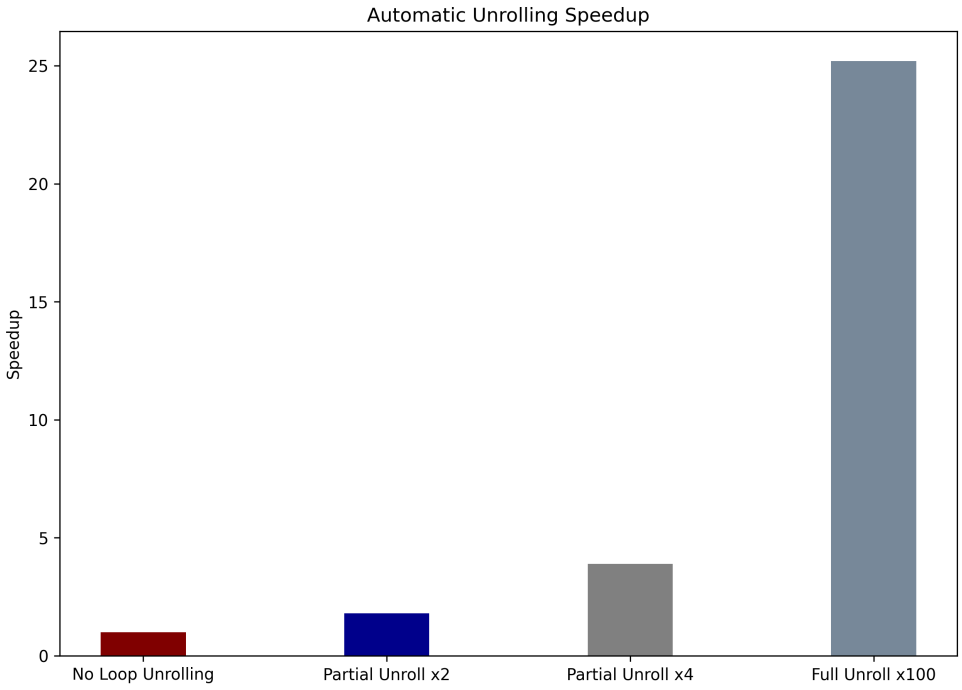


Εικόνα 6: Επίδοση των αρχιτεκτονικών

Τεχνική Βελτιστοποίησης Automatic Loop Unrolling

Το *Ξεδίπλωμα βρόχου*(loop unrolling) αποτελεί μια διάσημη τεχνική στην παράλληλη επεξεργασία και συνεπάγεται την αναπαραγωγή του σώματος του βρόχου πολλές φορές ώστε να χρησιμοποιηθεί για την επιτάχυνση του αλγορίθμου και τη μείωση του κόστους του βρόχου. Απαραίτητη προυπόθεση για να υλοποιηθεί αυτή η τεχνική είναι οι πράξεις μεταξύ των δεδομένων να είναι ανεξάρτητες και τότε το loop unrolling βοηθάει στην μείωση της καθυστέρησης του αλγορίθμου γενικότερα. Για να καθοδηγήσουμε τον compiler να ξεδιπλώσει βρόχους, χρησιμοποιούμε την εντολή *#pragma unroll* ενώ θα πρέπει να καθορίσουμε και τον παράγοντα που επιθυμούμε.

Στην περίπτωσή μας, και εφόσον το Change Detection είναι μια απαιτητική υπολογιστικά εφαρμογή, κρίνεται απαράιτητο να κάνουμε unroll όπου αυτό είναι εφικτό. Εκτός από την εξωτερική επανάλλειψη του βρόχου προβλεψης της εικόνας, η τεχνική εφαρμόζεται σε όλον τον υπόλοιπο κώδικα χωρίς πρόβλημα. Τα αποτελέσματα της εφαρμογής της τεχνικής αυτής είναι διακριτά στην παρακάτω εικόνα.

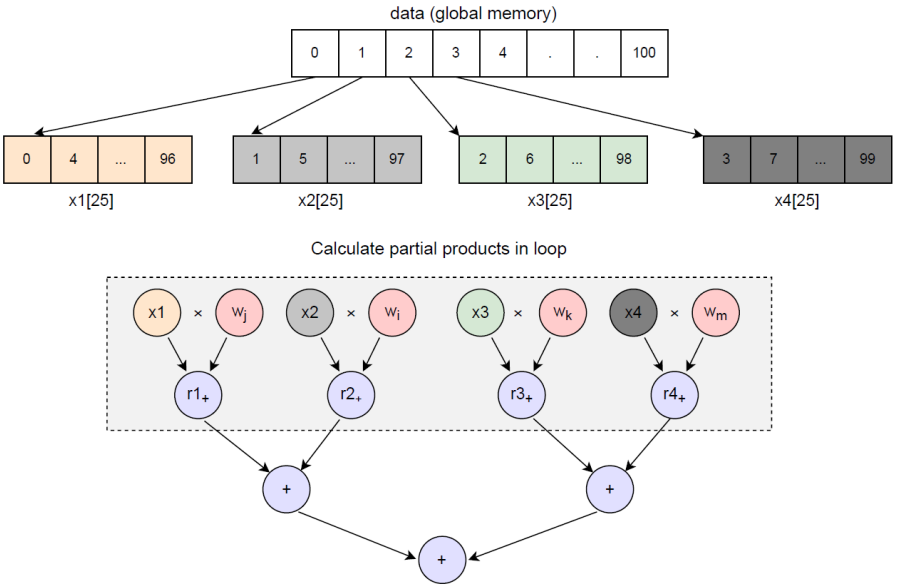


Εικόνα 7: Επιτάχυνση από τεχνική Auto Unrolling

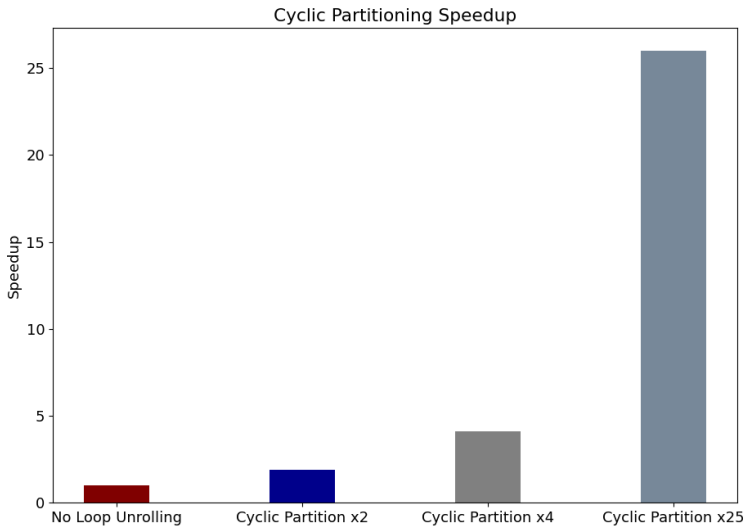
Περαιτέρω Τεχνικές Βελτιστοποίησης

Κυκλική Διαίρεση Πίνακα

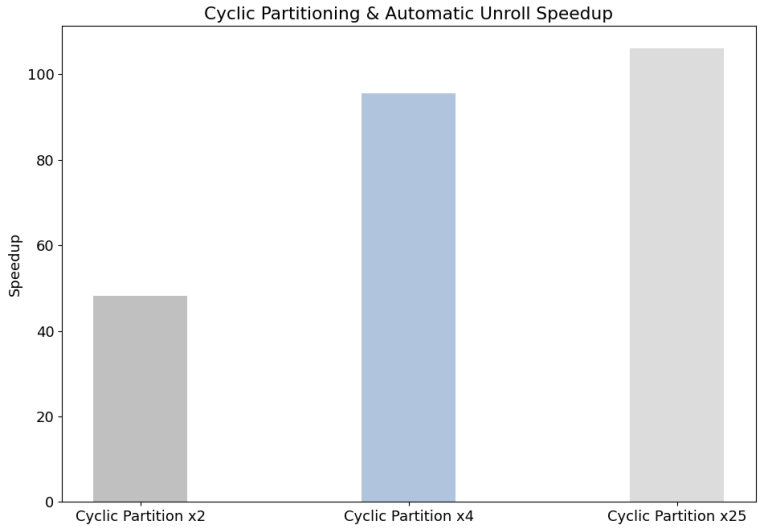
Τα προηγούμενα αποτελέσματα, παρ΄ ότι θετικά, δεν συμπεριφέρονται με τον βέλτιστο τρόπο, αφού για ξετύλιγμα με παράγοντα 100, ο πυρήνας καταφέρνει να τρέξει μόλις 25 φορές γρηγορότερα. Στο κεφάλαιο αυτό αναζητήσαμε λύσεις σε χειροκίνητες τεχνικές, όπως είναι η *Κυκλική Διαίρεση Πίνακα*, η σχηματική μορφή της όποιας φαίνεται στην εικόνα 8. Τα αποτελέσματα της στρατηγικής αυτής είναι εμφανή στην 9 ενώ παράλληλα αναζητήσαμε τρόπους να συνδυάσουμε την τεχνική αυτή με τις ενσωματωμένες τεχνικές του προηγούμενου τμήματος(pramga unroll) τα αποτελέσματα των οποίων βρίσκονται στην 10.



Εικόνα 8: Κυκλική Διαίρεση - Σχηματική Αναπαράσταση



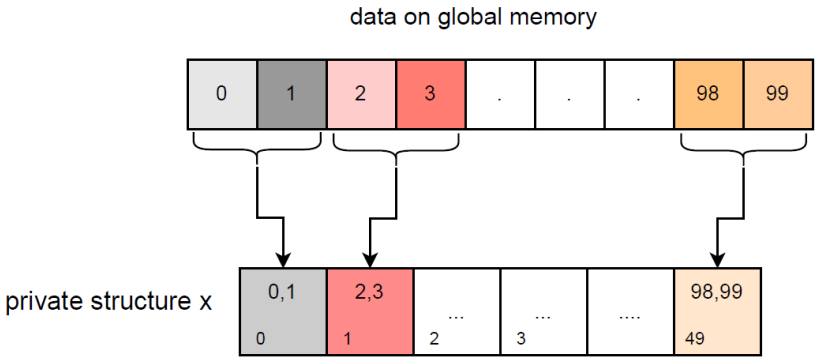
Εικόνα 9: Κυκλική Διαίρεση-Επίδοση



Εικόνα 10: Επιτάχυνση - Συνδυατικές Στρατηγικές

Αναδιαμόρφωση Μνήμης

Άλλη μια αρχιτεκτονική με την οποία πειραματιστήκαμε είναι αυτή της *Αναδιαμόρφωσης της Μνήμης*, μια υλοποίηση που βασίζεται στο vectorization των δεδομένων με σκοπό τη μείωση του κόστους του indexing μέσα στον πίνακα. Μια σχηματική απεικόνιση προσφέρεται στην εικόνα 11, ενώ ο πίνακας 3 περιέχει μια σύγκριση μεταξύ μη βελτιστοποιημένου hardware, και hardware στο οποίο έχουμε προσθέσει την εντολή #pramga unroll. Η αναδιαμόρφωση της μνήμης με αυτόν τον τρόπο φαίνεται να αποτελεί λύση στο πρόβλημα του της μείωσης της επίδοσης του αυτόματου ξετυλίγματος. Η συγκεκριμένη αρχιτεκτονική θα χρησιμοποιηθεί εκτενώς στα επόμενα κεφάλαια, μιας και προσφέρει την κάλυτερη επίδοση χωρίς να επιβαρύνει με περιττή κατανάλωση πόρων την συσκευή.



Εικόνα 11: Σχηματική Ανπαράσταση Αναδιαμόρφωσης Μνήμης

Πίνακας 3: Επίδοση για ένα Tile - Αναδιαμόρφωση Μνήμης

Design	**No Loop Unroll**	**Auto Full Unroll**
Original	7095ms	282ms
Reshape x2	6162ms	144ms
Reshape x4	7351ms	70ms
Reshape x8	7580ms	67ms
Reshape x16	7095ms	69ms
Reshape x32	7863ms	70ms

Αρχιτεκτονικές Ελαχιστοποίησης Καθυστέρησης

Οι τεχνικές που προηγήθηκαν αποτελούν ένα πρώτο επίπεδο παραλληλισμού. Η απόδοση μπορεί να βελτιωθεί περαιτέρω συνδυάζοντας τις καλύτερες τεχνικές μέχρι τώρα με κάποιες αυτόματες τεχνικές βελτιστοποίησης που παρέχονται από το HLS.

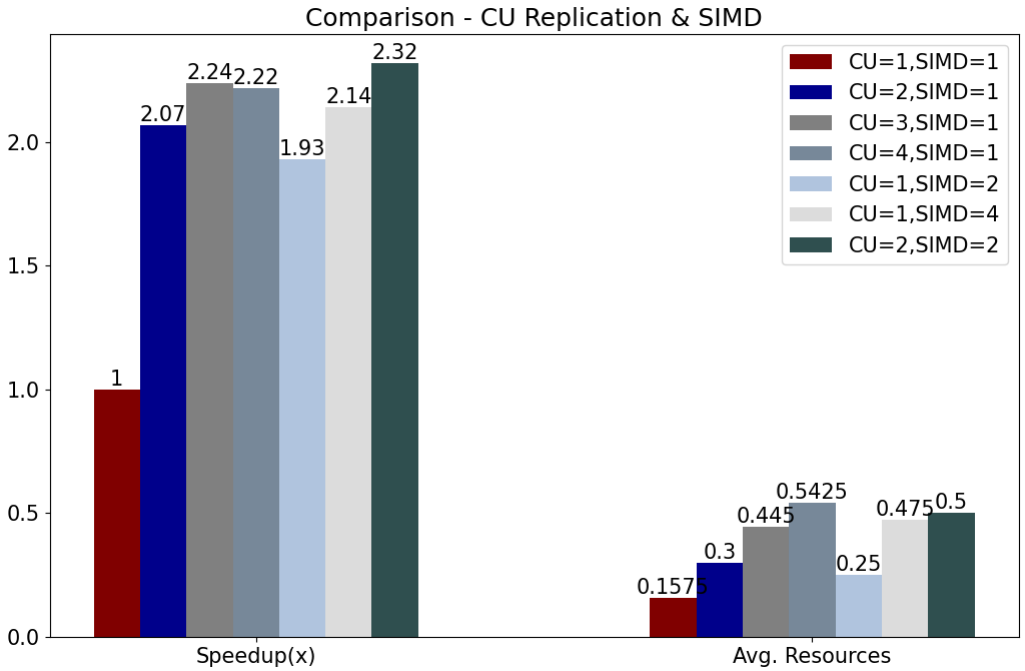
Οι τεχνικές του κεφαλαίου αυτού επικεντρώνονται στην εκθείαση του *Coarse-Grain* παραλληλισμού, το οποίο επιτυγχάνεται όταν ένα πρόγραμμα χωρίζεται σε μεγάλες διεργασίες που εκτελούνται παράλληλα. Ο τρόπος με τον οποίο θα επιδιώξουμε κάτι τέτοιο είναι μέσω των διαθέσιμων OpenCL directives, τα οποία παρουσιάζονται συνοπτικά ως εξής:

- **num_compute_units**: Πρόκειται για μία τεχνική που σκοπεύει στην μείωση της καθυστέρησης της συνολικής σχεδίασης μέσω εκτενέστερης κατανάλωσης των πόρων του συστήματος. Ο τρόπος με τον οποίο το επιτυγχάνει είναι μέσω της αντιγραφής της ήδη υλοποιημένης μονάδας επεξεργασίας κατά ένα παράγοντα που ορίζει ο χρήστης. Η επίλυση του προβλήματος επομένως διαμοιράζεται στους πυρήνες που έχει καθορίσει ο χρήστης με έναν αυτόματο τρόπο.

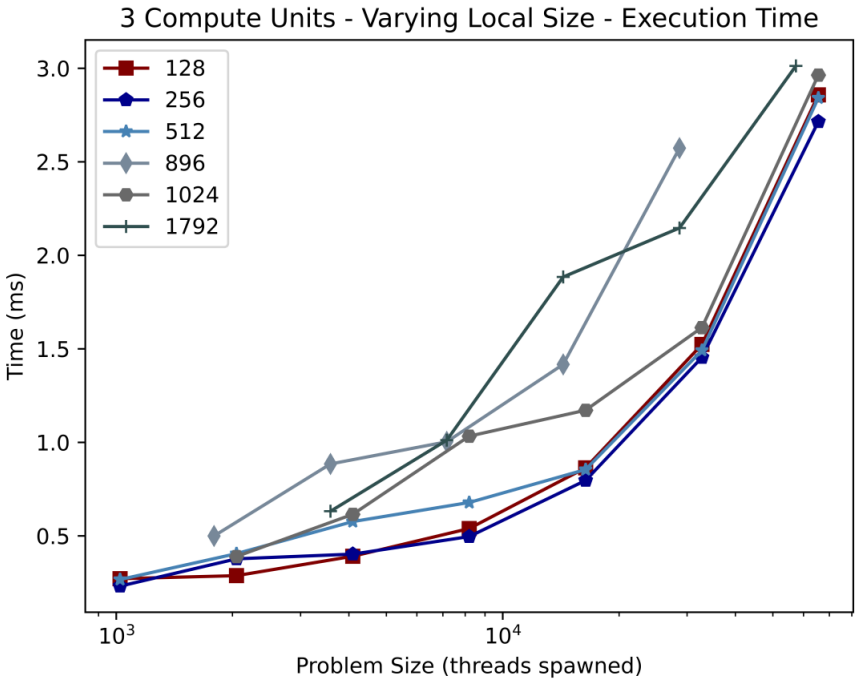
- **num_simd_work_items**: Το directive αυτό τροποποιεί τον φόρτο εργασίας μίας μονάδας επεξεργασίας ώστε να υπολογίζει περισσότερα δεδομένα στον ίδιο χρόνο. Παρόλο που οι προσβάσεις στην μνήμη μειώνονται ανά μονάδα επεξεργασίας, η τεχνική αυτή προσθέτει συχνά αρκετό βάρος στην κατασκευή του τελικού design.

- **reqd_work_size**: Μέσω της εντολής αυτής, ο compiler επιβάλλει βελτιστοποιήσεις στην τοπικότητα του σχεδιασμού χωρίς να σπαταλόνται πόροι του συστήματος. Καταφέρνει έτσι να δημιουργήσει αποδοτική δρομολόγηση ανάμεσα στα threads , πράγμα αναγκαίο για επίλυση μεγάλου μεγέθους προβλημάτων.

Παρακάτω παρουσιάζονται τα κύρια αποτελέσματα που προκύπτουν από την διερεύνηση αυτών των αρχιτεκτονικών.



Εικόνα 12: Σύγκριση μεταξύ Μονάδων Επεξεργασίας(CUs) & SIMD Αρχιτεκτονικών

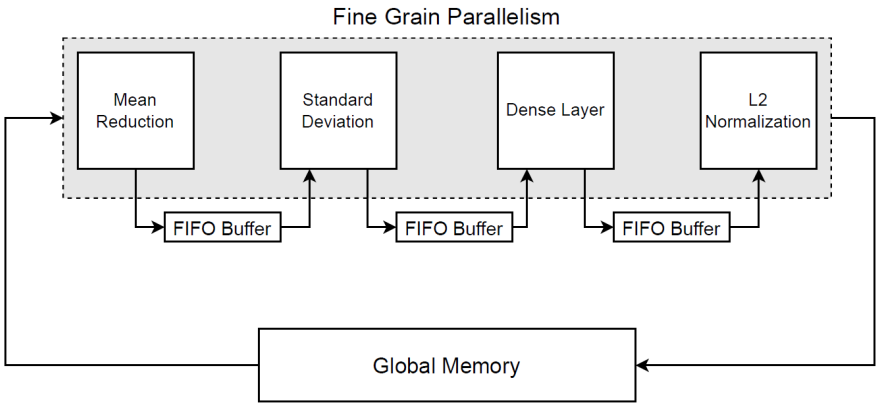


Εικόνα 13: Μεταβολή επίδοσης για 3 Υπολογιστικές Μονάδες

Αρχιτεκτονικές για Αύξηση της Διεκπεραίωσης Εργασιών

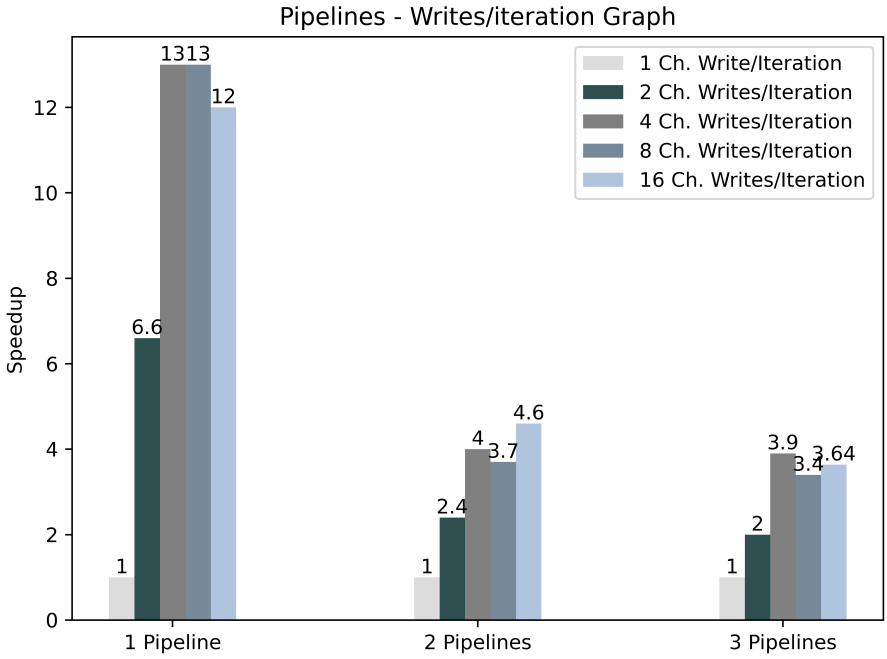
Το κεφάλαιο αυτό επικεντρώνεται στην δημιουργία αρχιτεκτονικών *Fine-Grained* το οποίο σημαίνει ότι το κυρίως πρόβλημα χωρίζεται σε μικρότερες διεργασίες που εκτελούνται παράλληλα ενώ η επικοινωνία του διασφαλίζεται μέσω *καναλίων*, που μπορούν να θεωρηθούν ως FIFO buffers. Τα κανάλια χρησιμοποιούνται λόγω της

υψηλής τους απόδοσης και χαμηλής καθυστέρησης. Μία σχηματική αναπαράσταση της προαναφερθείσας αρχιτεκτονικής δίνεται στο σχήμα 14. Σε αυτήν την περίπτωση, κάθε κομμάτι του αλγορίθμου, όπως αυτό έχει οριστεί στα προηγούμενα κεφάλαια, αποτελεί έναν ξεχωριστό πυρήνα που είναι υπεύθυνος για τον υπολογισμό ενός μέρος του συνολικού προβλήματος.



Εικόνα 14: Υλοποίηση Μονού Pipeline

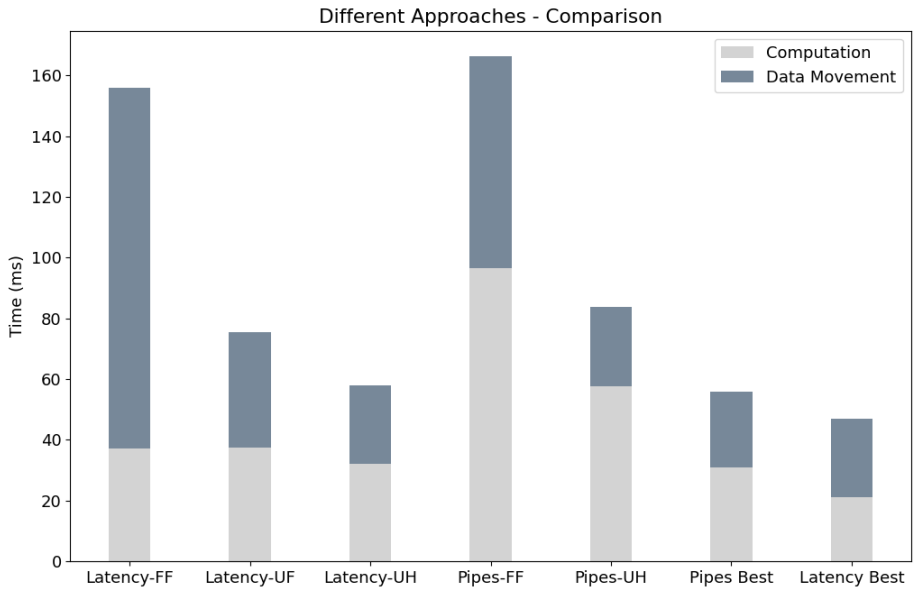
Γενικά, αρχιτεκτονικές αυτού του είδους οδηγούν σε λιγότερο κοστοβόρα design και επωφελούνται από μεγάλα μεγέθη προβλημάτων. Μια διερεύνηση σχετικά με το πλήθος των παράλληλων pipelines διενεργήθηκε εδώ, τα αποτελέσματα των οποίων είναι τα εξής:



Εικόνα 15: Εξέλιξη χρόνου εκτέλεσης σε συνάρτηση με το πλήθος των Pipelines

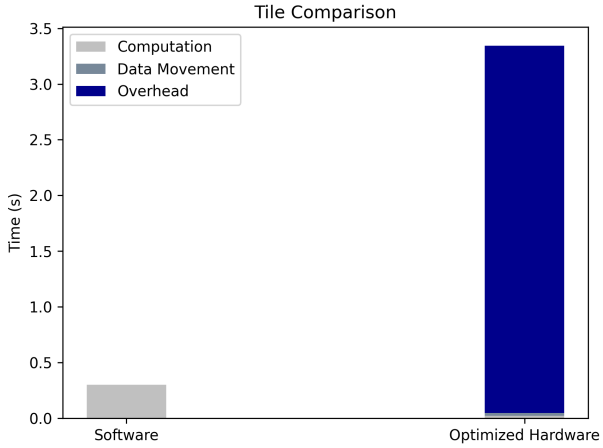
Ενσωμάτωση σε Λογισμικό

Στο τελευταίο κεφάλαιο της διπλωματικής, γίνεται μια συγκριτική παρουσίαση ανάμεσα σε όλες τις υλοποιημένες αρχιτεκτονικές σε συνδυασμό με την επίδραση του data movement από και προς το FPGA. Διευκρινίζεται ο τρόπος με τον οποίο επικοινωνούν τα host & device , ενώ τέλος προτείνεται μια υλοποίηση που αφορά την ενσωμάτωση της καλύτερης αρχιτεκτονικής με το λογισμικό.



Εικόνα 16: Σύγκριση Υλοποιημένων Αρχιτεκτονικών

Παρόλο που η βέλτιστη εκδοχή μας επιτυγχάνει 7x βελτίωση στον χρόνο υπολογισμού του μοντέλου, ο τρόπος διασύνδεσής του με το λογισμικό είναι αναποτελεσματικός και επιδέχεται περαιτέρω δουλειάς.



Εικόνα 17: Σύγκριση πλήρους Ενσωματωμένης Σχεδίασης

Chapter 1

Introduction

With the advent of the Copernicus program and its wealth of open data, the Earth Observation domain is increasingly adopting big data technologies. This adoption is firstly made possible by efficient data storage and processing infrastructures, but most importantly by the development of data analytic applications with machine learning techniques. In this context, Thales Alenia Space (TAS) proposes to develop an application of change detection on Earth Observation satellite images time series. Examining changes of a designated area over a period of time enables applications in many sectors, such as security, emergency, maritime and land surveillance[4].

TAS change detection tool provides generic change detection maps for pairs of time-consecutive Sentinel-2 data products that represent exactly the same field of view. The application can be considered as a pipeline with distinct parts, the first of which being the download of the two images in the form of data products from one of Copernicus access platforms. They are then transformed and preprocessed to produce a single Sentinel-2 Image with the bands of interest. After the images have been sorted in by time-series in chronological order, the computation of the change detection map begins. This computation is based on a pretrained neural network model, which is robust to lightning and atmospheric condition differences. Neural network algorithms help us detect features such as shapes, edges and motion in digital images. The resulting change detection maps are geoTiff rasters with the same size and geo-reference as the images they have been calculated by, and whose pixel values range between 0 and 1. These values represent the probability that a change has occurred between the acquisition dates of the two images[5][6].

The repetitiveness (every few days) and the free access of Sentinel-2 (S2) satellite images result in massive datasets. An important challenge in multitemporal change detection is the fast access and storage of a big amount of data and the computationally-intensive processing which is necessary. Furthermore, another challenge of change detection is sensitivity to atmospheric distortions, such as cloud or aerosol optical thickness. Given these circumstances,

we propose a solution concerning the acceleration of the design, allowing better accuracy and improved response times.

The algorithm throughout its execution creates and processes an immense amount of data, thus bottlenecks make their appearance at the total execution time. It seems necessary to turn our attention towards hardware acceleration. With this technique, a complex algorithmic system can be build to run efficiently while not being resource hungry. However, designing a HW accelerator with Hardware Description Languages such as VHDL or Verilog can be time consuming and a difficult to debug task. Using HLS (High Level Synthesis) Tools removes these barriers, as the functional description of an algorithm in the form of a C function is automatically translated into a hardware description. Although a HDL build offers the best possible outcome performance-wise, utilizing HLS tools enables quick development cycles, hence the exploration of a wide variety of architectural characteristics which is almost impossible to achieve in HDL given a small amount of time.

Our work consists of an exploration about the last part of the Multi-Temporal Change Detection computational part of the pipeline. We start of by analyzing the Change Detection tool as a process. Its profiling revealed bottlenecks in performance, since about 75% of the time spent in the module is used for the computation of the change detection map inside the main loop. Subsequently, the theoretical backround, necessary for the reader to understand the main terms that will be used, is presented. Information concerning the mathematical backround as well as the way that our tools work are included. In order to reduce the time consuming-part of the Change Detection tool using hardware acceleration, we developed a prediction model with the help of HLS Tools. The exploration continues with the application of several techniques and strategies with the intention of accelerating our prediction model to match the speed achieved by software or even surpassing it. At this point, the process is split into two parallel semantic ways, the one comprised of latency optimized architectures and the other of throughput based ones. Both these approaches manage to achieve desired outcomes, while they handle the performance-resource consumption tradeoff in different ways. Latency optimized designs offer better performance when predicting a predefined problem at the cost of utilizing more area, whereas throughput based architectures lack in performance for smaller problems, but are as competitive as latency ones for larger problems, while occupying significantly less area. At the end of this diploma thesis, an integration with the original software is proposed. The reader is offered a start-to-finish explanation on how the best-performing design is implemented to work alongside the software.

In this diploma thesis, we explore a big amount of both micro-architectural

traits and general architectures concerning hardware with the ultimate goal of accelerating the original design. In Chapter 2, a brief overview of Multi-Temporal Change Detection application is presented. The main parts of the algorithm are broken into distinct pieces. Bottlenecks that come up from application profiling when run on a general purpose processor are also included. Chapter 3 is about the theoretical background of neural networks, parallel computing and the design tools and platforms used for the acceleration process. In Chapter 4, a reconstruction and a first implementation code is presented as well as some base code level optimizations. Latency and Throughput optimized architectures are demonstrated in Chapters 5 and 6 respectively. Lastly, in Chapter 7, we propose an end-to-end implementation using Docker and the ctypes module. C++ and Python interconnection is also issued whereas 8 contains the Conclusions of the entire diploma thesis.

Chapter 2

Problem Overview

Change Detection Analysis

Change Detection is an application that is used to encode and calculate the per element difference between two large images. Inference tasks that operate on small fragments of an image in order to focus in one or multiple areas of interest within a single image are also examined. This technique is adopted by the change detection tool that was developed from Thales Alenia Space and computes the differences between time-consecutive Satellite images of the same field of view. These differences are identified by a neural network model that performs the detection on small tiles of the original images. The model comprises of two layers and operates on 4-channel 5-by-5 pixels tiles. In this case, the complexity lies in executing an immense number of inferences for an unbound number of images. Throughout the process, a great amount of data is produced and processed.

Originally, the workflow of the application starts with loading the two images from memory. They are then transformed into datatypes that can be processed by the libraries utilized. Before entering the main processing loop of the algorithm, they are extracted to gain extra per pixel depth. Once the main loop starts, a batch of size (batch size, 10976,100) is issued for each one of the two images. Each batch is normalized before getting predicted in a process that contains casting the correct datatype, reducing the mean value and dividing with standard deviation across the third axis. The normalized data are then reshaped into smaller images of size (5,5,4) resulting in a total amount of (batch size)x10976 images. These images are passed into the prediction model which consists of a Flatten Layer, where the images are reshaped into arrays of 100 elements, a Dense Layer of size 25, which inputs arrays of 100 and outputs 25 features and a L2 Normalization Layer where the 25 features get normalized through a process we will examine in the next Chapter. After the sub-images from the two batches have been predicted they are subtracted element-wise and appended to the output array. The loop iterates for every batch in both

images. Once the loop is over and every batch has been normalized, predicted, subtracted and appended to the final array, results are reshaped into an image of size (10976,10976) and after user-defined masks have been applied, the output image is stored back to memory and the process ends. A graphical representation of the complete process is presented in Figure 2.1.

Each Satellite image is about 350MB in storage size and corresponds to a large image of an area. Software loads them in *rgba* mode with *8 bit unsigned integer* precision, which leads to 480 MB in RAM utilization per image. The extraction part scales this data into 12 GB of 8 bit unsigned integers and shape (10976,10976,100) for each image. Once the main computing loop starts, each batch is transformed into its float representation of 16 bit precision in order to get normalized and inserted in the prediction model. Once the prediction model is finished, the data are in 32-bit form. Same precision remains until the application ends. Table 2.1 shows how data dimensions and precision changes for each image throughout the process whereas size reflects the total amount of load generated by the two images that the system has to sustain at every step.*Batch size* is specified by the user and can be equal to 64,128 or 256. Bigger batch size means more RAM utilization which can lead to systems crashes since every iteration of the prediction loop holds and processes more data at once.

Table 2.1: Data specifications throughout the process

Processing Step	**Data Dimensions**	**Data Precision**	**Size (MB)**
Image Load	10996,10996,4	8bit unsigned int	967.3
Image Extraction	10976,10976,100	8bit unsinged int	24945
Batch Selection	128,10976,100	8bit unsigned int	∼280
Normalized Data	128,10976,100	16bit float	∼560
Prediction Output	128x10976,25	32bit float	∼280
Output Image	10976x10976	32bit float	480

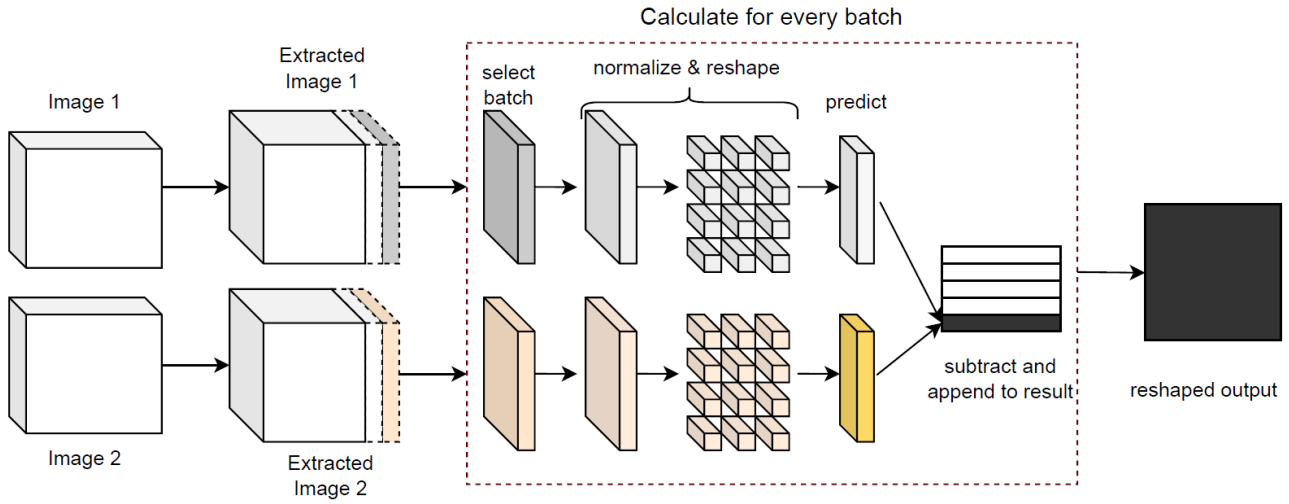


Figure 2.1: Change Detection Schematic Representation

The code is written in Python using some of the most popular libraries available. **Numpy** and **math** are utilized for main data processing parts such as setting specific data-types or calculating the mean value. **GDAL** library which is capable of programming and manipulating geospatial data is used for loading image metadata and storing the output as a *.tiff* extension image, a format useful when storing raster graphics images. Nonetheless, the heaviest part of the computation relies on the **Tensorflow** library, which is an open source platform, mainly used for training machine learning models while offering effective solutions in dataset preprocessing or normalization tasks. In Change Detection application tool, Tensorflow is responsible for loading the input images in *rgba* form and extracting them through a process that is similar to applying a Convolution Layer to an image. After the batches have been split, Tensorflow normalizes them and predicts the intermediate image outputs using a pretrained model. Lastly, it reshapes the output image and applies the mask on it before GDAL stores it back to memory. Although Tensorflow and all the utilized libraries are widely used and contributions for further development are made to them every day, they struggle to keep up with a real-time application which is so compute intensive such as Change Detection. In the following subsection we will examine why and how this incident occurs.

Application Profiling

Due to their large total size, as we saw from Table 2.1, producing the output of the extracted images becomes a resource-full and time consuming process. Results from Software profiling for 3 different values of *batch size* revealed the slowest parts of the algorithm as found in Figure 2.2. Loading the two images takes up to 17% of the total time. The main processing loop which was discussed in the previous subsection takes a total of 74% of the time. The remaining time is consumed in extracting the input images and writing back the output one, equal to about 9% of the total time. Taking into consideration these measurements, it feels necessary to turn our attention towards the main loop. Inside it, we split and calculate the difference between two predicted batches and append it to an output array in every iteration. From main loop profiling we find out that normalizing two full images in steps of size (batch size,10976,100) is rather costly and equals to 72% of the loop time consumption or about half of the total time that the complete application runs. The rest of time is spent on predicting the batches and reshaping and appending to the output array.

Lastly, it is interesting to focus on how time is spent across the normalization part of the algorithm. Figure 2.3 indicates that standard deviation division part is the most time consuming one with a 60% time share. Of course, that is to

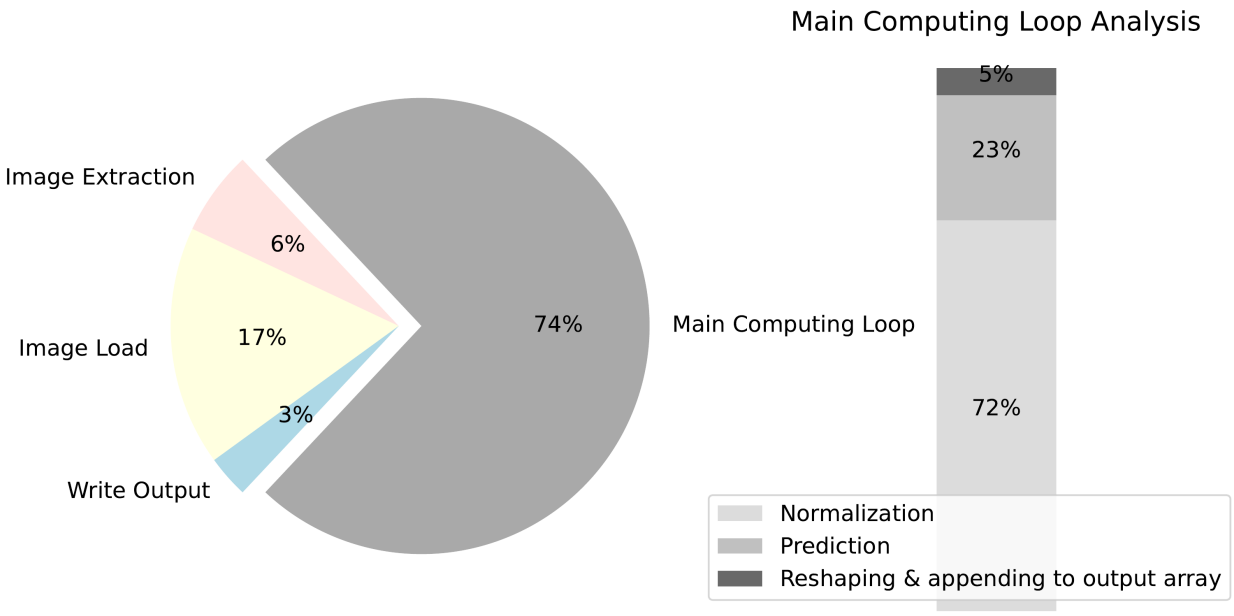


Figure 2.2: Profiling

be expected, since it requires the most compute intensive calculations relative to the other two. Mean reduction and correct Datatype casting are responsible for the rest of the remaining time.

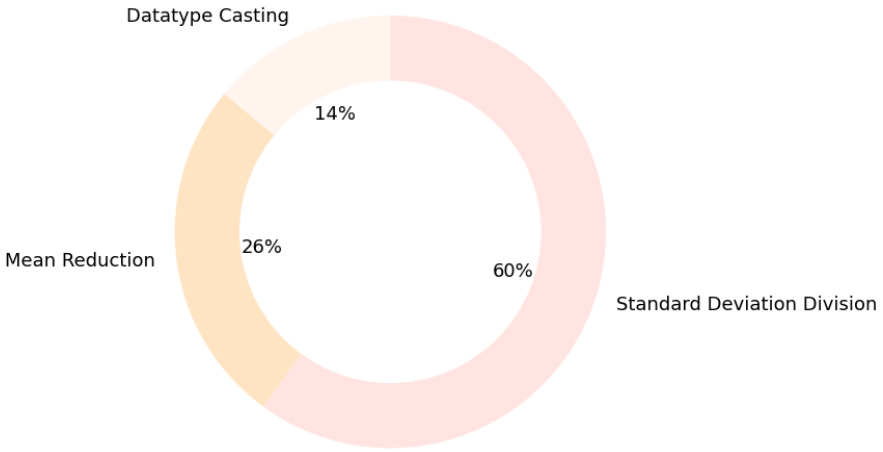


Figure 2.3: Normalization Profiling

Talking with absolute numbers, the complete algorithm spends about 110s when run on a high-performing general purpose processor. This amount of time is restricting for real-time applications where an unbound amount of images are passed in the model every second. This thesis will focus on trying to create a more effective environment for the algorithm, targeting its acceleration through the exploration of micro-architectures and complete build architectures on *Intel Stratix 10 FPGA*.

Chapter 3

Theoretical Backround

Backround information on Image change detection

One of the best ways for detecting changes between two or more images is through predictive Artificial Neural Networks. Neural Networks try to imitate the way that human brain works, creating a network where some neurons are connected to some other in order to pass information and make decisions. In this thesis, for the sake of predicting an image, we are going to deal with a limited amount of the vast ANN field of studies, turning our attention solely to a layer of neurons, also known as a Dense layer in Python's Tensorflow library.

A single input neuron can be seen in 3.1.

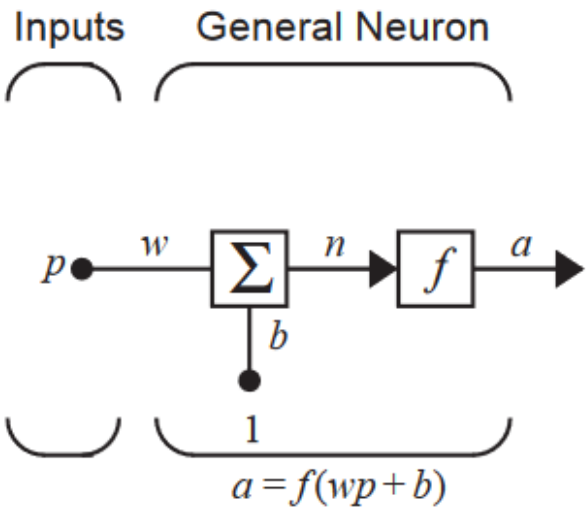


Figure 3.1: Single Input Neuron[1]

A single input neuron can be seen in Fig.3.1. A scalar input p is multiplied by weight w and summed with bias b to form the summed output n. This n product goes into the activation function f which calculates the neuron output y [1].

A single input neuron however is practically of no use on its own. Multiple inputs x should be applied on a neuron in order for it to be useful. This topology

is called a multiple input neuron or perceptron and its goal is to classify a set of inputs x in a successful and desired way [7], e.g used to implement an AND logic function. The equation responsible for producing the result is:

$$y = f\left(\left(\sum_{i=0}^{n-1} w_i x_i\right) + b\right)$$

and the topology can be found in 3.2.

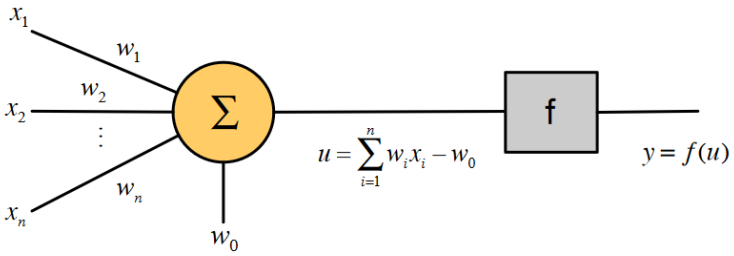


Figure 3.2: Perceptron

When more complex problems are on the line, perceptrons fall short. A single-layer network of S neurons are able to solve these kind of problems and will be the centre of our attention, as seen in *Figure* 3.3. This model is used in this thesis for predicting the two images. As input, there are 100 elements which produce 25 outputs. This model is used for the prediction of the two complete images of the application, whose per-predicted element difference we are eager to find.

The primary role of an *Activation Function* is to transform the produced sum from the computing node(s) into an output value. Activation functions introduce an additional step to the process but its computation is worth it, because without it, every neuron would be performing linear transformations on the inputs. There are several types of activation functions, such as Binary Step, Linear and Non-Linear activations. In our case, leaky Rectified Linear Unit (or ReLU) is implemented, which is a subcategory of linear transfer functions. Leaky ReLU deals effectively with the problem of dying nodes, which is noticed when the neuron becomes stuck in a perpetually inactive state and "dies". The equation that will be realized in this thesis is shown below :

$$f(x) = \begin{cases} 0.1x, & \text{if } x < 0 \\ x, & \text{otherwise} \end{cases}$$

Another important part of the process is normalizing the output. The goal of normalization is to change the values of numeric columns in the given dataset to a common scale, without distorting differences in the ranges of values. Normalization in this case is useful because the algorithm does not make assumptions

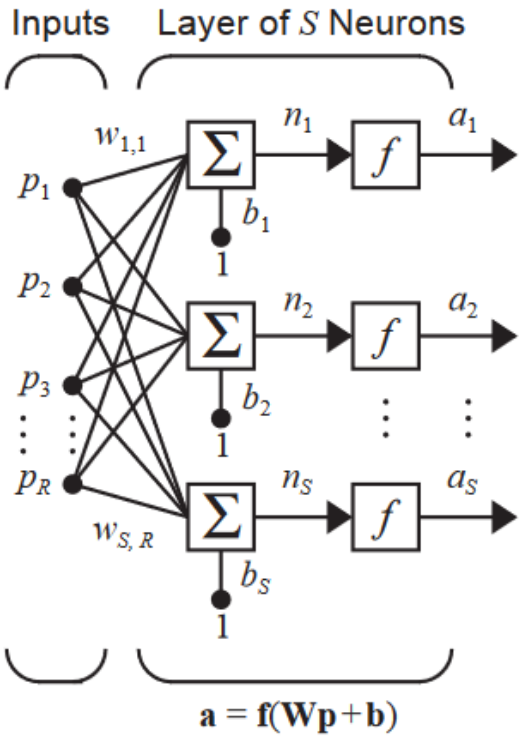


Figure 3.3: Layer of S neurons [1]

about the distribution of the data. It is performed on the predicted 25 features and contains division for each element by the sum of the square of input x. This method is called *L2 normalization* technique and is realized by the model to normalize the output batch, as shown below:

$$output = \frac{x}{\sqrt{\sum_{i=0}^{n-1} x_i^2}}$$

High Level Synthesis Tools

High Level Synthesis Tools give the designer the opportunity to work at a higher level of abstraction while creating high-performance hardware. It provides software developers with an easy way to accelerate the computationally intensive parts of their algorithms on a Field Programmable Gate Array (FPGA). The FPGA provides massively paralleled architecture with benefits in performance, cost and power over traditional processors. The main part of the application is thus executed on the systems' processor while a part of it is transformed into a Register Transfer Level (RTL) implementation that is synthesized into a bitstream that can run on the FPGA.

The flow of using HLS, in this case the *Intel FPGA SDK for OpenCL* is briefly descripted in Fig.3.4. Applications using the Intel FPGA SDK are composed of two parts: the FPGA programming bitstream(s), and the host program that manages the application and FPGA Accelerator. The kernel, written in

OpenCL, is firstly compiled to an image file that the host program uses to utilize the FPGA. In host side, the program, written in C or C++, is compiled and linked to the OpenCL runtime libraries. The designer synthesizes both kernel code, that is to be run on FPGA, and host code, which is run on a general-purpose embedded processor and controls the kernel(s) implemented on FPGA. At the top level, the OpenCL host uses the OpenCL API platform layer to query compute devices, submit work to them and manage the workload across compute contexts and queues([2],[8]).

HLS tools provide the ability to work in a organized and structured way in order to implement optimizations or other iterative modifications in the design. Every step of the process serves as a checkpoint for identifying functional errors and performance bottlenecks. At first, the ability to verify the functional correctness of the algorithm is granted by running the design in simulation mode, achieving that in a faster pace than traditional Hardware Description Languages do. After that, the synthesis of the design is issued as an intermediate compilation step. By doing so, the RTL design is produced while HLS offers the ability to control the OpenCL synthesis process through optimization strategies allowing the creation of specific high-performance implementations. Once synthesis is complete, the Intel FPGA SDK creates a *reports/report.html* file which contains valuable information about the design[3]. By reviewing this file, we can detect potential resource or data performance bottlenecks in our design and fix them by consulting the suggestions of Intel SDK. The RTL compilation output from the Design Tool is provided in the industry standard Hardware Description Language format of Verilog and is split in a *base.v* and a *<kernel name>.v* file. At the last step, the RTL implementation is integrated into a hardware system, in a process of full build compilation, which takes several hours to complete, producing a *.aocx* binary file that host can use.

One of the pros of using HLS tools is that they can compile any C code into an implementation of high performance while maintaining an efficient resource usage. This is accomplished by adding HLS-defined strategies, such as pragmas, that are taken into account during RTL compilation process and result in optimized IP builds. HLS creates the optimum implementation based on its own behavior, constraints and strategies that the user defines. The optimization strategies are selected so that the architecture satisfies the desired performance and resource consumption goals.

The report.html file that is produced from synthesis is a first step of estimating how the hardware build will perform and how many resources will be spent on it. After analyzing the report, optimization strategies can be applied to refine the implementation towards a more desired outcome[3]. It is hence useful to understand what kind of information we extract from the report file

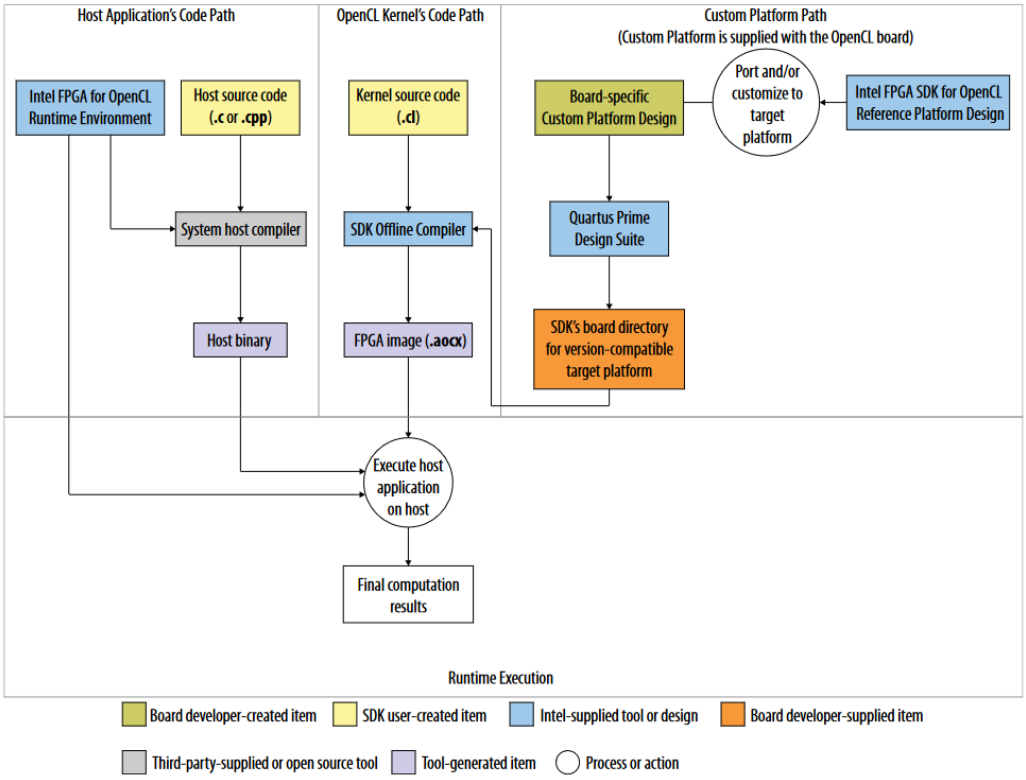


Figure 3.4: HLS flow using Intel FPGA SDK for OpenCL[2]

and what kind of metrics are used to assess performance in a design generated by HLS. Some of these metrics are :

- **Resource Usage**: Indicates how many hardware resources are spent for the implementation of the design. Lookup Tables (LUTs), Flip Flops (FFs), RAMs, and Digital Singal Processing blocks (DSPs) are the kind of resources that FPGA combines to synthesize and implement a design. An assessment of their usage is found in report file whereas after hardware implementation is complete, report contains the actual number of resources utilized (might be slightly different).
- **Latency** The latency of a function or computing block is the number of clock cycles that are required to complete their execution.
- **Initiation Interval** The Initiation Interval (II) is the amount of cycles before a loop can accept new data. When II cannot be defined, the report file presents other characteristics of the loop such as thread capacity.
- **Kernel Memory Usage** After the user declares memories for the design, the offline compiler checks the structure of the code, the loops and calculates potential bottlenecks. It is then decided whether some memories get replicated in order for more threads to be run in parallel, practically for the sake of increasing II.

OpenCLs' main component is the kernel. A **kernel**, which is represented by a set of blocks in the report.html file, is a unit responsible for implementing the algorithm the designer describes in his/her code. OpenCL kernel runs the code in parallel inside the FPGA over a predefined N-dimensional computational domain, called *NDRange*. These N elements are called "work-items" and can be grouped into batches called "work-groups". An NDRange kernel is beneficial for the design when multiple concurrent threads can run in parallel without any data dependencies occurring between them[8]. More often than not, providing a standard work-group size can help speed up the overall FPGA performance. Fig.3.5 depicts how work-groups are scheduled and queried inside a NDRange kernel.

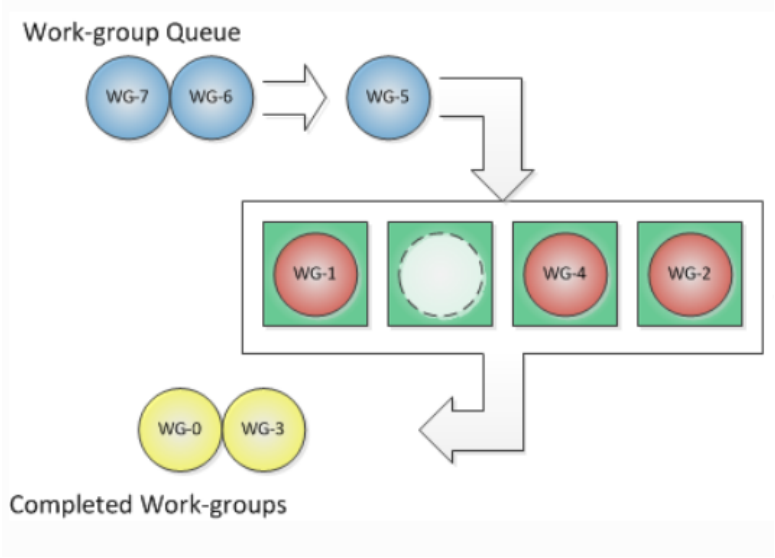


Figure 3.5: Scheduler Workflow after Local Size is specified

If N is equal to 1, the kernel is described as a single work item and in this case, the compiler applies different techniques aiming to accelerate it. Another useful component are loops. In single work item's case, the compiler tries to accelerate every loop in the design to allow multiple iterations to execute concurrently. On the contrast, NDRange loops do not get pipelined, but they can accept multiple work-items at the same time to be executed in parallel[3].

Every high performance computing design benefits from the potential ability of its HLS Tool to be able to maximize memory bandwidth. Intel FPGA SDK Compiler tries to recognize the global memory patterns of the design and realize the appropriate Load-Store Unit (LSU) for each separate occasion. Some of the most characteristic LSUs are Burst-Coalesced, Prefetching, Local Pipelined etc. Unfortunately, the designer cannot impact the LSU implementation in a direct way, however there are techniques that are used to "hide" global memory bottlenecks and increase total build throughtput. These techniques lie on the correct selection of memories inside the design. OpenCL supports three types

of memories other than global : *local, constant and private*. Local memory is visible to all the work-items across a work-group, constant memory resides in global but the kernel loads it into an on-chip cache and is shared by all work-groups at runtime and lastly private memories are visible to a single work-item, implemented using registers or block RAMs and provide the most bandwidth among all the memory types. Last but not least, two useful kernel components are channels and pipes, operating in a similar manner. They are mainly used for inter-kernel communication, in cases when one kernel produces some data and another reads them.

In addition to a general guideline that we are trying to follow by using effectively the components we discussed so far, Intel HLS Tool provides us with some extra strategies to optimize the design. Optimizations can be applied on loops or the complete design. In Table 3.1 some of them are briefly descripted.

Table 3.1: HLS Optimization Strategies

Strategy	**Description**
#pragma ivdep	The ivdep pragma instructs the compiler that there are no loop carried dependencies between the iterations of a loop resuting in potential reduction of logic utilization.
#pragma unroll	Used to replicate a loop body multiple times reducing loop control overhead and therefore latency.
#pramga coalesce	Directs the compiler to form a single loop out of two or more nested loops. Kernel area is thus reduced by eliminating some loop control overheads.
reqd_work_group size	Specifying the local size instructs an NDRange kernel to perform aggressive scheduling in work-items, resulting in more work-groups working in parallel. The size of local size should evenly divide the global problem size.
num_compute_units	Specifies the number of times that the kernel is replicated inside the FPGA. Compiler distributes work -groups across the compute units dynamically.

Stratix 10 device specifications

The Intel Stratix 10 GX 2800 FPGA is a modern high-performance device, containing more than 2.7 million Logic Elements (LEs) inside its kit. LEs form the smallest Unit of logic inside the FPGA and provide advanced features. Intel FPGAs are also composed of Adaptive Logic Modules (ALMs), which are building blocks designed to maximize performance and utilization. Each ALM encolses adaptive LUTs, a two-bits full adder (FA) and four registers. Another unit of Stratix 10 is a logic array block (LAB) which is made of multiple ALMs. LABs can be designed to realize any kind of function the user wants, such

as arithmetic or register functions. Combining up to a quarter of the total available LABs to serve as memory blocks forms the memory LABs (MLABs) in the FPGA. A maximum of 640 bits of simple dual-port SRAM is supported by every MLAB. Dual-port SRAMs are low-latency devices that only take a clock cycly to perform a read/write operation([9],[10]). A vast amount of Digital Signal Processing blocks (DSPs) that are used to accelerate multiplications and accumulations are also provided.

Table 3.2: Intel Stratix 10 available resources

LEs	**ALMs**	**Embedded Mem.**	**DSPs**
2753000	933120	244 Mb	5760

Intel Stratix 10 uses M20K blocks as RAM. They can be configured as true dual-port, simple dual-port and single-port RAM and ROM. Each M20K block supports three clock-enable controls which gives the opportunity for either clock-enable controls or no gating clock control. Clock muxing is balanced, which prevents skew between clock paths.

Chapter 4

HLS Architectural Optimizations

So far, the Satellite Image Change Detection tool has been analyzed, the libraries it utilizes to perform its tasks as well as the main bottlenecks that prevent it from being effective as a real-time application. We then examined the theoretical backround of encoding an image using neural networks, the HLS Tools that help us construct applications and the specifications of the device our application will be run on, the Intel Stratix 10 FPGA. In this Chapter, an embodiment of the Satellite Image Encoding Application in the FPGA is presented as a prediction model firstly. Subsequently, strategies and techniques that are available in OpenCL will be examined, how they are being applied to our prediction model and in which way they impact the model separately. These optimizations will either occur on a memory hierarchy level (Chapter 4), micro-architecturally inside some code's structures (4 and 4) or through specifications concerning the mathematical operations of the architecture (Chapter 4).

Prediction model description

As we examined in Chapter 2, Satellite Image Change Detection is a compute intensive task. Analyzing and profiling the application revealed bottlenecks that concern mainly the processing part rather than the input image loading and output image storing parts. In that sense, it is thought-provoking to experiment with FPGAs and check whether they can help us with the optimization of the main loop execution part. In this first subsection, a prediction model that will be realized inside the FPGA is presented, in order to replace the most time consuming part of the algorithm, as that is inferred by Section 2.

In software code, every batch that gets processed is subject to a number of reshapings that occur on it. Starting with batches of shape (batch size, 10976,100), every element of the sub-matrix (batch size, 10976) gets normalized across the third axis of size 100. An element of the sub-matrix is then turned into an image of size (5,5,4) which is a manageable data form from the pre-

trained tensorflow's prediction model. Inside the model however, images are again reshaped into a flattened array of 100 values that are used to produce the 25 outputs of the Dense Layer. A schematic representation is given in Figure 4.1. A specific element is highlighted to represent the changes that occur throughout the process. Every element is independently normalized and predicted from one another.

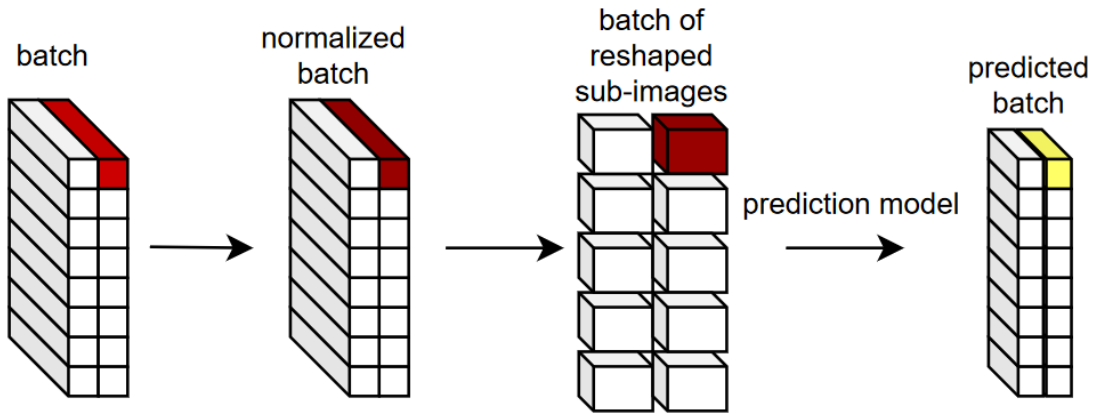


Figure 4.1: Batch Level Approach (Software)

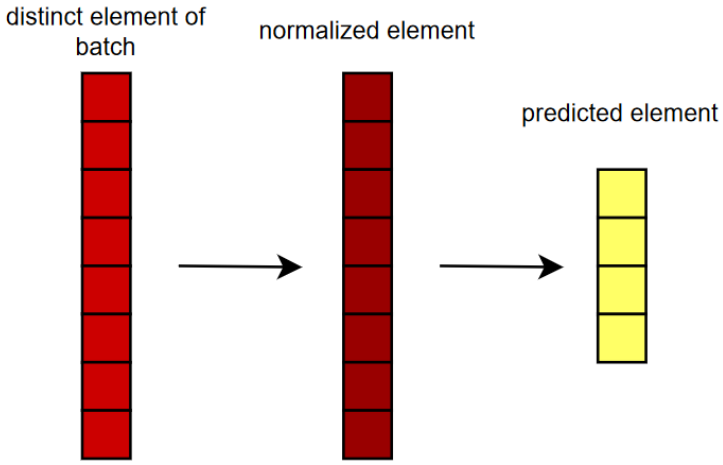


Figure 4.2: Element Level Approach (FPGA)

The above described approach can be presumed as *batch level* parallelism, in the sense that the data get processed in a group-like way using Python's and Tensorflow's libraries. However, every element inside the batch produces its results self-reliantly and hence calculations can occur in every distinct element of the batches freely and in an out of order style[11]. Figure 4.2 shows how every elemenent will be treated inside our implementation. Our desire is to build a kernel that inputs threads accompanied by 100 data values which output 25 values with the ultimate goal of scaling the design for a complete image. We chose this *element-wise* parallelism in order to take advantage of OpenCL's ability to schedule a large amount of threads using the *NDRange* kernel cast.

The total problem size will be equal to the size of the first 2 dimension of the input batch, which is (batch size,10976) in a two-dimensional space or (batch size)x10976 in a one-dimensional space, and these are the amount of threads that will be deployed in total. Each thread corresponds to an element of a batch, "carrying" 100 values and it follows the procedure below in a strict manner in order to produce the predicted element:

1. **Mean Reduction** : Each one of the 100 values has to be subtracted by the dataset's mean value. The output of the first step is calculated as:

$$x_{mean} = x - \frac{1}{N}\sum x$$

   where x is equal to the input values.

2. **Standard Deviation Division** : The process continues with the division of the elements by their standard deviation value. Mean value has to be calculated again from the mean-reduced dataset. The formula for this step is:

$$x_{std} = \frac{x_{mean}}{\sqrt{\frac{\sum |x_{mean} - \mu|^2}{N}}}$$

3. **Dense Layer** : Since data remains in the same kernel (values are in buffers) and no write-backs to memory occur, they can be immediately used as an array of 100 in the input of a Dense Layer. Dense Layer and the steps that are followed was analytically discussed above in subsection 3.1.

4. **L2 Normalization** : This method occurs once the 25 outputs from Dense Layer are calculated and constitutes another normalization method applied in the design overall; its steps are found in subsection 3.1 as well.

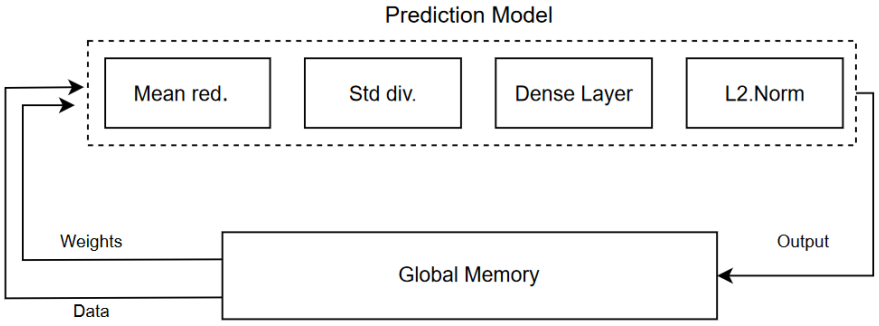


Figure 4.3: FPGA Image Encoding Prediction Model

A systemic overview of the the algorithm is found in Figure 4.3. Weights are used as *read only* values, whereas data are inserted and change throughout

mean reduction and standard deviation division until used by Dense Layer. Lastly, bias is also needed for Dense Layer calculations, but it is provided as a constant array of 25 values. The output of the Dense layer is based on the dot product between 2 matrices, the data and the weights. Weights come in the form of a (25,100) matrix and every time, a column is issued in order for the product to be calculated. Figure 4.4 shows a schematic representation of this process. Every value of the weights column is per-element multiplied by the data ones. After the dot product has been computed, the corresponding bias value is added to it, the activation function, leaky ReLU, is applied and hence the output is produced. This process is repeated until all outputs have been calculated.

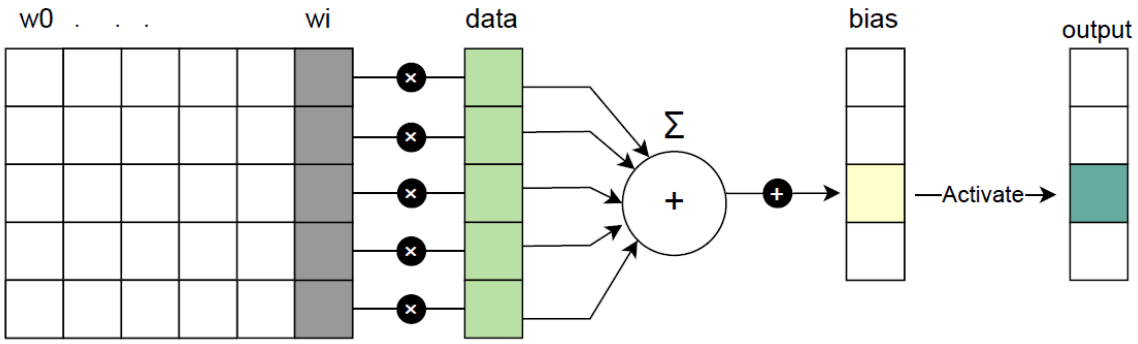


Figure 4.4: Dense Layer-Dot Product Flow

Lastly, as an intro for the next Sections, we present the OpenCL implementation of the complete prediction model we have discussed so far. In the next parts of Chapter 4, our main focus will be to find architectures and HLS techniques to optimize certain parts of the code and integrate them with software.

```
__constant float bias[25] = {-0.03938436,1.77568137..};
__kernel void prediction_model(__global float data,
        __constant float weights,
        __global float output)
{
  int element = get_global_id(0);
    float mean,sum,mean_sum;
  int oof, off,of1;
  of1 = element*100;
  //.............................................
  //............Mean reduction part..............
  //.............................................
    for (int j = 0; j<100; j++){
    mean_sum += data[of1+j];
        }
    mean_sum /= 100.0;
    for (int j=0;j<100;j++) {
    data[of1+j] -= mean_sum;
        }
    //.............................................
  //.........Std division part.............
  //.....................................
    for (int i=0; i<100;i++){
```

```
    mean += data[of1+i];
  }
  mean /=100.0;
  for (int i=0;i<100;i++){
        sum += (data[of1+i] - mean)*(data[of1+i] - mean);
        }
    sum /=100;
    float dev1 = sqrt(sum);
    for (int i=0; i<100; i++) {
        data[off+i]/=dev1;
  }
    //.......................................
  /............Dense Layer.................
  //.....................................
   off = element*25;
   for (int index=0; index<25;index++){
        output[off+index] = 0.0;
        for (int i=0; i<100; i++){
            output[off+index] += data[of1+i]*weights[index*100+i];
                }
        output[off+index]+= bias[index];
        if (output[off+index]<0) {output[off+index] *=0.1;}
        }
    //.........................................
  //............L2 Normalization............
  //.....................................
    sum = 0.0;
    for (int i=0; i<25; i++){
        sum += output[off+i]*output[off+i];
        }
    sum = sqrt(sum);
    if(sum<0.0144)sum = 0.0144;
    for(int i=0; i<25; i++){
        output[off+i] = output[off+i]/sum;
        }
}
```

Listing 4.1: Prediction Model Implementation

Memory hierarchy configuration

Technique

As a first optimization technique, optimal memory hierarchy configuration will be issued. Data are inserted as a parameter in our prediction kernel whereas an output buffer carries the correct output values. This first subsection focuses on whether global interconnection can play a decisive role in limiting the performance of the kernel. Note that in this section, *weights* parameter will be ignored from the schemas since its relative size is far smaller than that of the data and output one and it is used as *read only* variable. It is hence passed as a ___*constant float* parameter of size 10kB, which means that at runtime, it is read and written in a memory fragment near kernel, as discussed in Chapter 3.2.

Generally, a high-performance design benefits from maximizing its memory bandwidth. Intel SDK compiler tries to find the optimal global interconnect method based on the kernel code and the memory access patterns in order to construct a Load Store Unit (LSU). In our case, the Burst-Coalesced Units which are formed, optimize the bandwidth by buffering the largest possible amount of data the design specifies in the cost of utilizing more resources. The compiler modifies some Burst-coalesced units when data dependencies need to be resolved, e.g a Read After Write (RAW) hazard, by adding a write-acknowledgment signal, or when memory accesses are not aligned to the external word size by matching the correct addresses. Naturally, modifications lead to an extensive usage of FPGA's resources and may cause throughput reduction[12].

Figure 4.5 depicts a high level overview of the architectures we will examine. The code of the first implementation (Hierarchy1) is the one found in Listing 4.1. A total of 12 global interconnections are formed, two for reading and one storing the mean-reducted data, four for the deviation-division part, two for reading the data and weights in dense layer and one storing back, and lastly two for reading and normalizing the output values.

It is generally a good practice to try and reduce the per-kernel global interconnection, as proposed by *Intel SDK Best Practice Guide* and by reviewing the *report.html* file, resulting in reduced resource usage and an accelerated design overall. In that sense, the private address space is utilized. By declaring a variable or an array as *__private*, the compiler tries to infer a register as it implements them. Registers are used whenever dynamic indexing is not required and they can lead in very efficient hardware, performance and resource-wise. The two remaining builds of Figure 4.5 make use of private arrays, the one (Hierarchy2) using a buffer of size 25 in Dense Layer's calculations as this consists the most compute intensive part of the algorithm, and the other one (Hierarchy 3) utilizes a buffer for input data as well, resulting in a total of 1 global read and 1 global store.

Results

Figures 4.6-4.9 depict how area consumption is distributed and how it affects the performance while changing memory hierarchy. Hierarchy 1 that holds the most global interconnections is averaging more resources than the other 2 designs without performing well enough. The last design that holds a single global load and a single global store is performing about 12% better than the first while occupying less FPGA area.

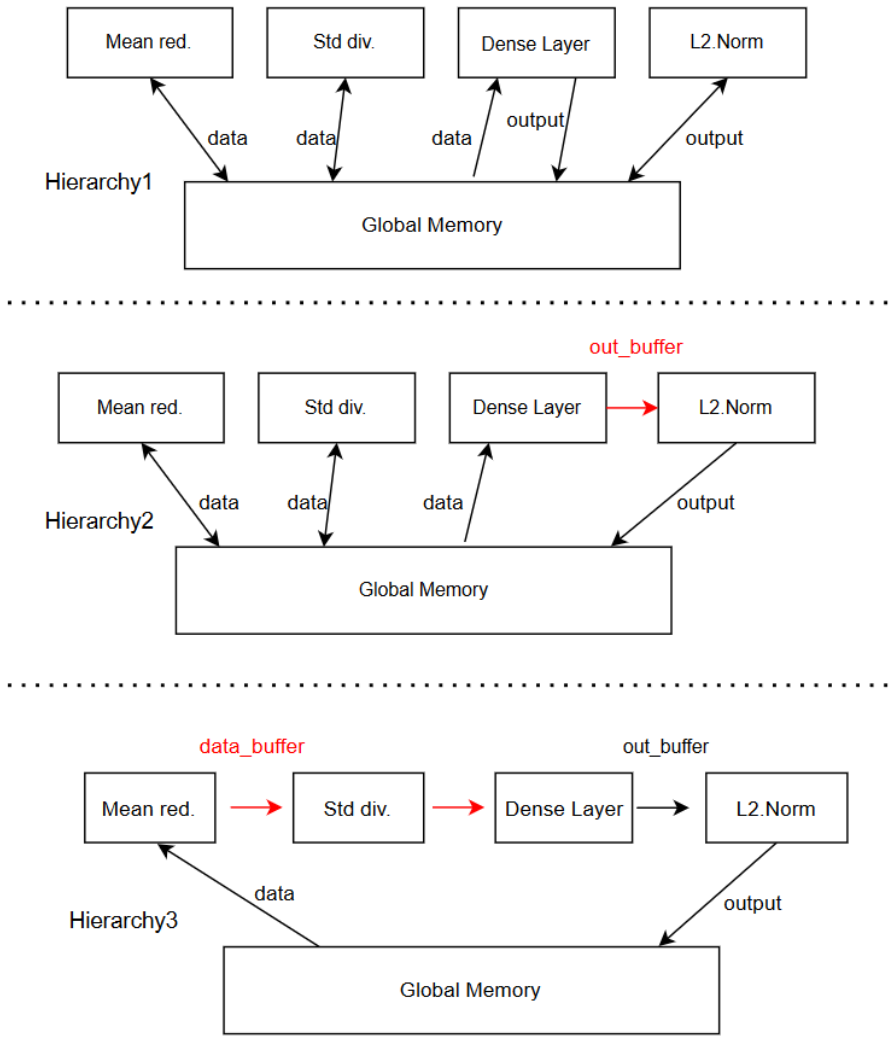


Figure 4.5: Architecture Overview

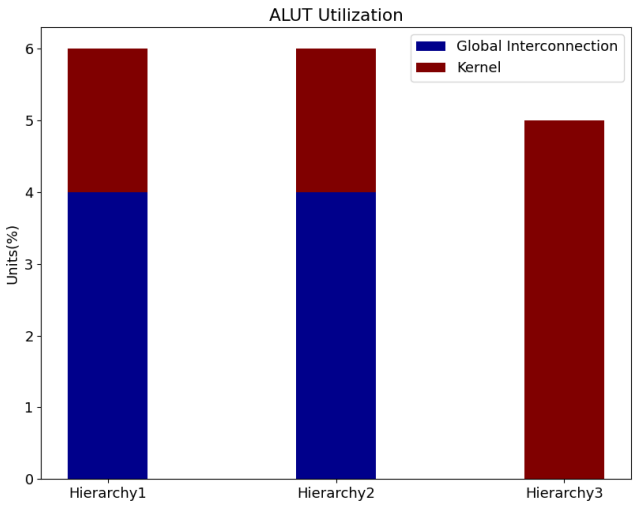


Figure 4.6: ALUT Utilization

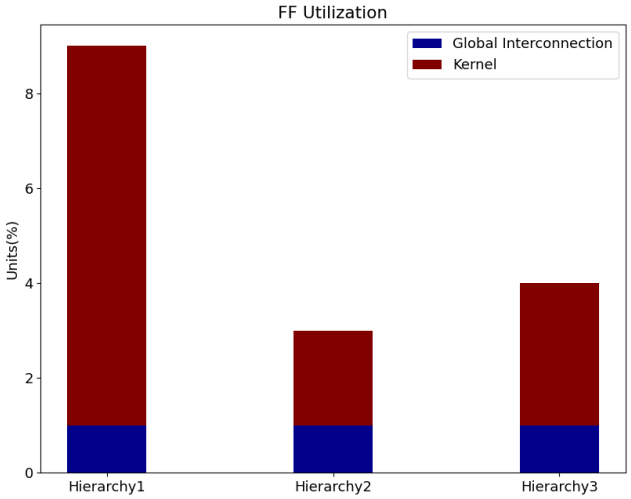


Figure 4.7: FF Utilization

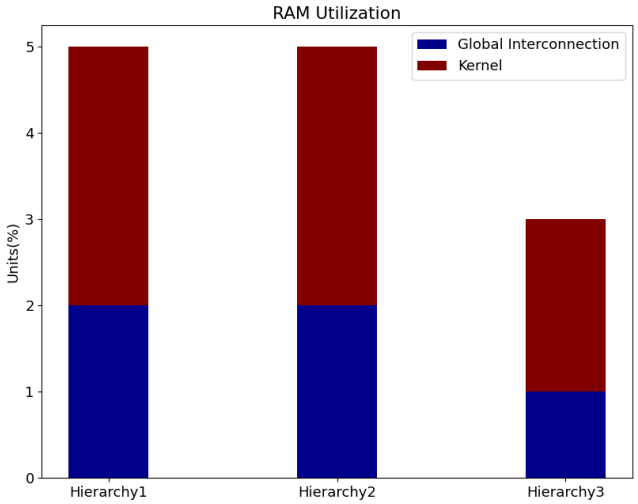


Figure 4.8: RAM Utilization

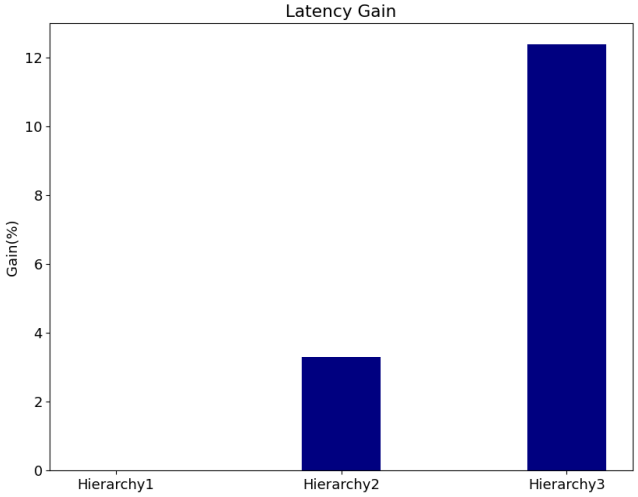


Figure 4.9: Performance of designs

Table 4.1: Memory Usage Comparison in Units

Design	MLABs	RAM Blocks
Hierarchy 1	201	425
Hierarchy 2	181	432
Hierarchy 3	1845	262

Latency gain is relatively low although we are using buffers. Private memories are implemented through MLABs and their performance is close to an SRAM structure since they utilize LUTs. However, Intel Stratix 10 contains M20K RAM blocks which are occupied in the cases where data are read directly from global memory. Table 4.1 shows how MLABs' and RAM blocks' utilization changes between the designs[9]. This means that in applications where a bigger amount of data are processed less times, RAM utilization will perform as good as using private memories. In our case though, a small amount of data are processed repetively until the output is produced hence private memories are used.

In general, we will use extensively the last model presented here (Hierarchy 3), since it is the only one that offers further optimization techniques to be applied on the design. Extensive usage of global interconnections leads to wiring over-utilization and designs fail to build. One global load and one global store help us experiment with some strategies such as loop unrolling and data reshaping, that we will experiment with further in this Chapter, as well as techniques that are applied directly on the kernel and scale the total design, as will be presented in Chapter 5.

Automatic Loop Unrolling

Sections 4 and 4 focus on finding ways to increase architectural granularity and play a decisive role in speeding up the design in the cost of resource consumption. The first technique that will be applied and tested is ***Automatic Loop Unrolling*** and it is provided as an OpenCL *pragma* addition[3].

Unrolling a loop is a famous design concept among parallel and high performance computing and it means replicating a loop body multiple times in order to speed up the algorithm and reduce loop control overhead on the FPGA[13][12]. In cases where there are no data dependencies and the compiler can perform loop iterations in parallel, unrolling loops can help reduce latency. The Intel FPGA SDK unrolls simple loops on its own during compilation time but further defining unroll factor through *#pragma unroll N* can help speed up the total build. Inserting pragma unroll N before a loop instructs the offline compiler to unfold the loop by a factor of N. If N is not defined, the compiler tries to fully unroll it, whereas specifying N equal to 1. prevents the

loop from unrolling at all. Of course, unrolling comes with the price of utilizing more resources and the danger of the compiler failing to build because of resource over-utilization.

Consider the code fragment below which calculates the result of a single multiple input neuron. By assigning a value of 2 as the unroll factor, the offline compiler is directed to unroll the loop twice, resulting in a total of 50 iterations instead of 100.

```
#pragma unroll 2
for (int i=0; i<100; i++){
    result += x[i]*w[k+i];
    }
```

Listing 4.2: Unroll Example

Image Change Detection is a compute intensive application. We firstly need to normalize the 100 elements through mean reduction and standard deviation division for each element, then use the normalized dataset to predict 25 output values and lastly perform a L2.Normalization on the output values. However, calculating the mean value or the standard deviation one can occur in parallel since the data responsible for producing them are read-only and independent from one another, although they have to be calculated in a sequential manner and not out of order. We can implement #pragma unroll in every distinct section of our code, reducing the total amount of iterations that need to occur. This way the compiler is instructed to unroll the loops 100 times. Listing 4.3 shows how standard deviation division is computed and applied when we instruct the compiler to fully unroll the loops. We experimented with unroll factors 2,4 and 100 to check how they affect the overall performance and area usage.

```
#pragma unroll
for (int i=0;i<100;i++){
    sum += (d[i] - mean)*(d[i] - mean);
    }
sum /=100;
float dev = sqrt(sum);
dev = 1.0/dev;
#pragma unroll
for (int i=0; i<100; i++) {
    d[i]*=dev;
    }
```

Listing 4.3: Standard Deviation division per element using loop unrolling technique

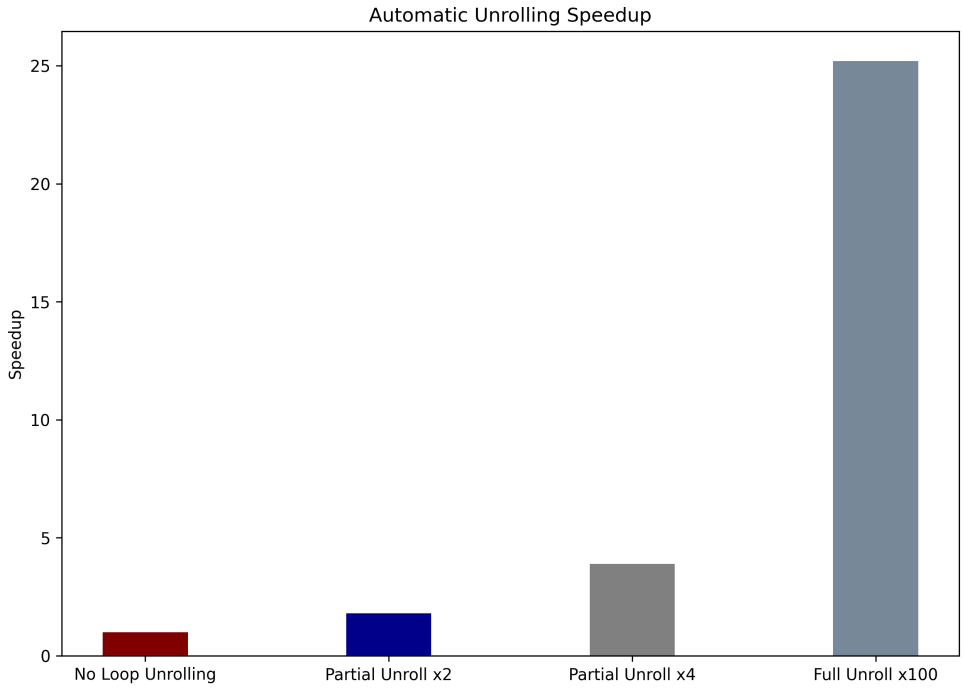


Figure 4.10: Auto Unrolling Speedup

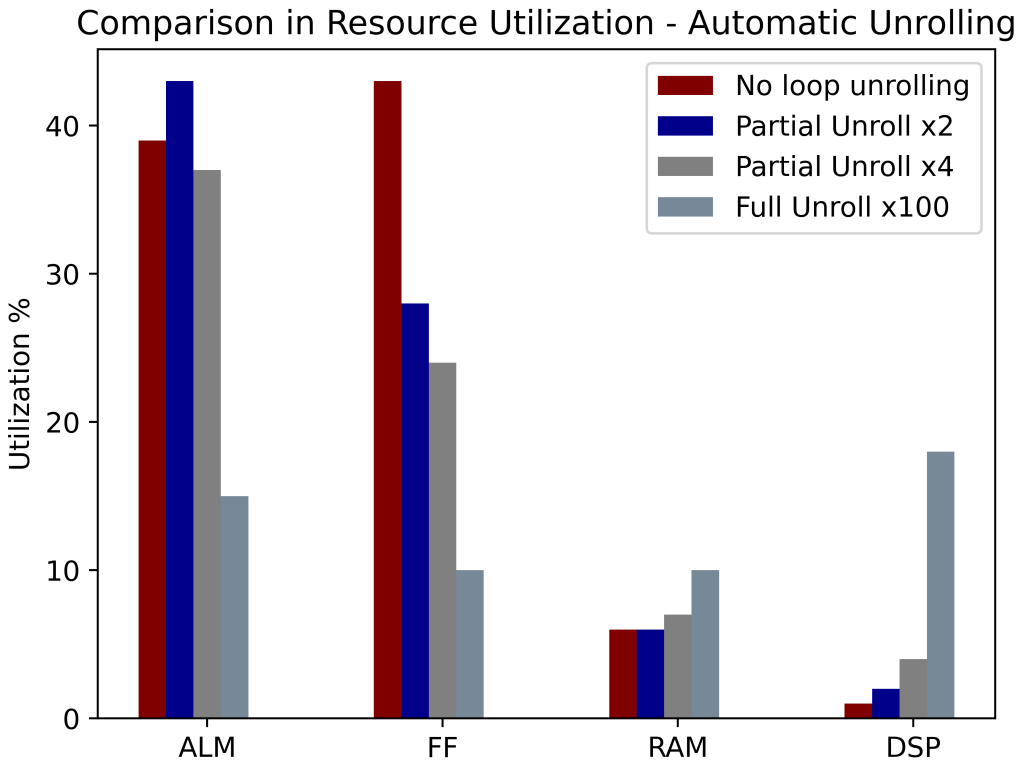


Figure 4.11: Auto Unrolling Resources Consumed

Figures 4.10 and 4.11 depict how unrolling helps the architecture. Resource

consumption for full loop unroll falls of as far as control logic is concerned while DSPs and RAMs get utilized more. Partial Unroll x2 seems to be the most resource hungry architecture without performing that well. Partially unrolling by a factor of two speeds up the design by 1.8 times, auto unrolling 4 times results in a 3.9x speedup whereas fully unrolling the loops speeds up the design by 25 times. The last speedup seems to be problematic since compiler was instructed to unroll 100 times but performance hit a roadblock, possibly because of data dependencies that the compiler cannot resolve on its own. In the next Section, we will try to understand why this incident occurs and present a way to manually overcome this difficulty.

Advancing HLS Granularity

Cyclic Partitioning

A way of increasing the amount of operations that occur during a loop iteration is through ***Manual Unrolling***. Although directives and strategies provided by Intel SDK are optimized and help accelerate a design, manually instructing the compiler to build a desired architecture is always an interesting approach when hardware-specific applications, and therefore FPGAs, are concerned.

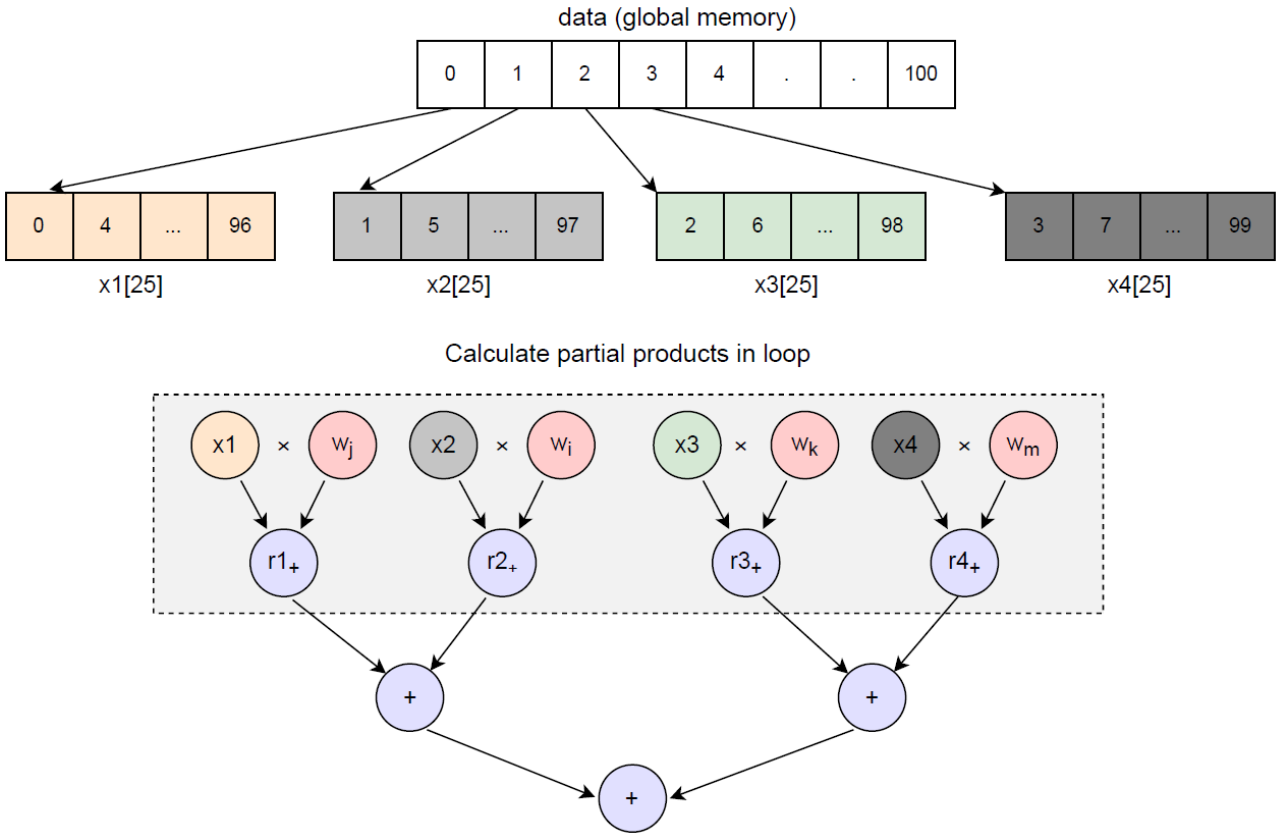


Figure 4.12: Cyclic Partitioning - Calculations' Flow Schematic

Cyclic Partitioning is about splitting the data in separate files and distribut-

ing them in the files in a cyclic way[14]. Figure 4.12 shows, firstly, how this is done when a 4 cyclic partition is issued. This way, we are certain that when, for example, a multiplication is issued, the four arrays are parsed in parallel without extra logic added for the loop to complete. In order to avoid data dependency bottlenecks, four variables are used to calculate the partial products. Lastly, the weights file is written as a constant near core and *report.html* file informs us that it automatically gets replicated to ensure the maximum cache hits ratio possible. When the partial products are ready, they are being added in a tree-based way to ensure maximum data independence until the final result is ready. In the next section we will see that this method can be implemented during compilation time, making the design easier to read without any performance loss.

We experimented with this technique as seen in the example in Listing 4.4. The same part of the code as in Listing 4.2 is presented so that the reader can easily spot the transformations and thought process we followed to achieve it. The size of the arrays is equal to 25 and contain the 100 data of an element, split in a cyclic way.

```
for (int i=0; i<25; i++){
    offset = 4*i;
    result1 += x1[i]*w[k+offset];
    result2 += x2[i]*w[k+offset+1];
    result3 += x3[i]*w[k+offset+2];
    result4 += x4[i]*w[k+offset+3];
    }
    result1+=result2;
    result3+=result4;
    result = result3+result1;
```

Listing 4.4: Unroll Example

Memory Reshaping

OpenCL supports custom data structures to be implemented in a design. In this sense, we experimented with *Private Memory Reshaping*. The way in which this idea is realized is presented in Listing 4.5 for an unroll factor of 4; the same part of the code is once more issued, whereas Figure 4.13 shows how data are distributed from global memory into a private float structure. Practically, it is a similar way of manually unrolling a loop, splitting data in a sequential rather than a cyclic manner. We refer to them as *Memory Reshaping* in our measurements to differentiate them semantically from the previous technique and since we alternate the way the memory is used, approaching a more simd-like way of parsing the data[15].

```
#pragma unroll
for (int j = 0; j<100; j++){
    f1=j&(0x0003);
    mean_sum+=d[j/4].dat[f1];
```

```
    }
mean_sum /= 100.0;
#pragma unroll
for (int j=0;j<100;j++) {
    f2=j&(0x0003);
    d[j/4].dat[f2] -=mean_sum;
    }
```

Listing 4.5: Implementation Code

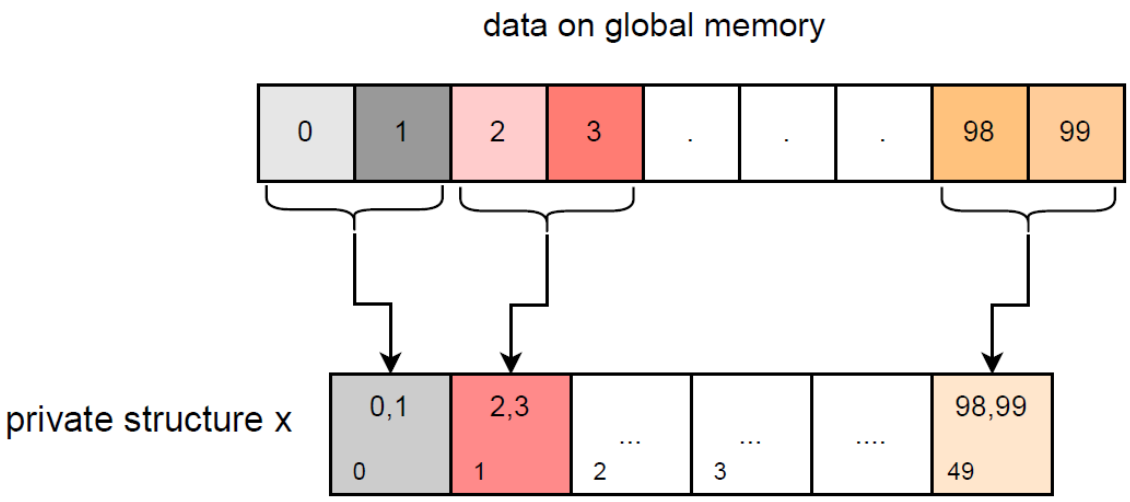


Figure 4.13: Reshape Partitioning Schematic

Results

Figures 4.14 and 4.15 show the speedup achieved after applying Cyclic Unrolling and the resources consumed. The performance of the first two builds are slightly better than that of automatic unrolling by about 4% whereas the utilized area remains almost the same. However, cyclically Partitioning the input array by a factor of 25, that is 25 arrays of size 4 each are created, speeds up the design by almost 25x and reaches the performance of *maximum automatic unroll*.

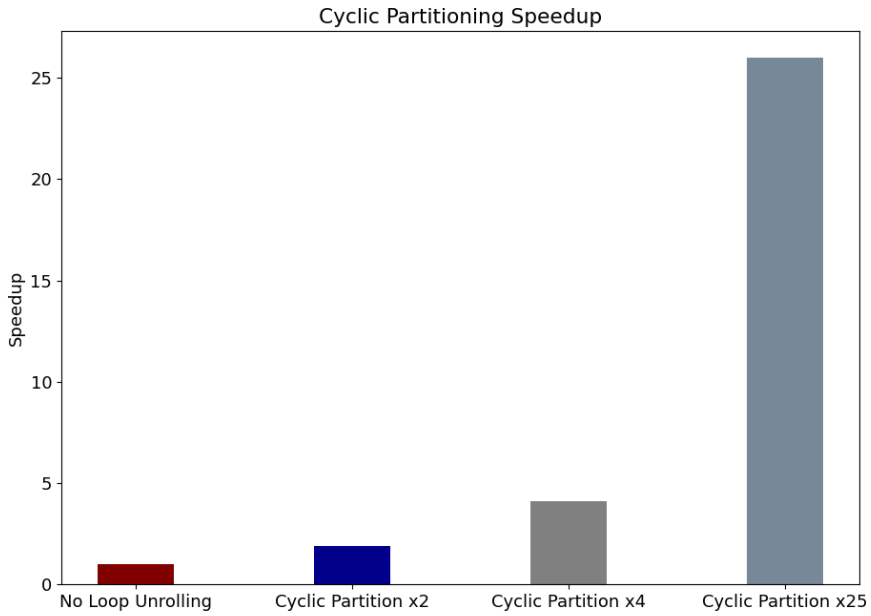


Figure 4.14: Cyclic Partitioning Performance

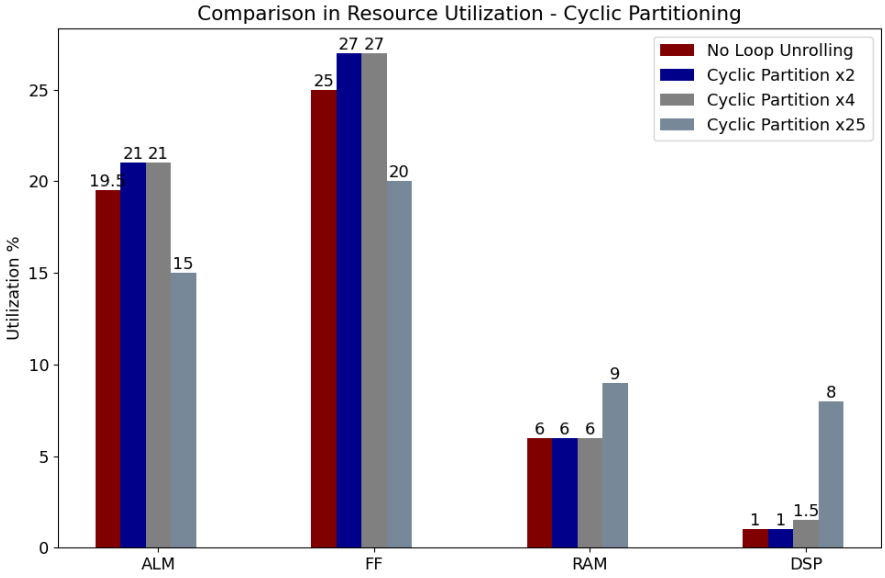


Figure 4.15: Cyclic Partitioning Resources

We experimented with combining the techniques and built in directives presented in 4, to achieve maximum speedup in the cost of utilizing more resources. For that reason every loop in the design will be assigned with the #pragma unroll specification alongside the two discussed techniques. OpenCL has to deal with a smaller amount of iterations since it is instructed to support more operations at once. Figure 4.16 holds the results of performance when automatic unrolling is also applied to the loops. The same amount of resources is utilized across every design of the combinations of manual with auto unrolling, independently from the unroll factor. However, a huge speedup can be seen, reaching the scale of 100x when compared to the original - no unroll - architecture. In

general, wiring's , ALMs' and FFs' utilization is reduced whereas DSPs' and RAMs' are increased.

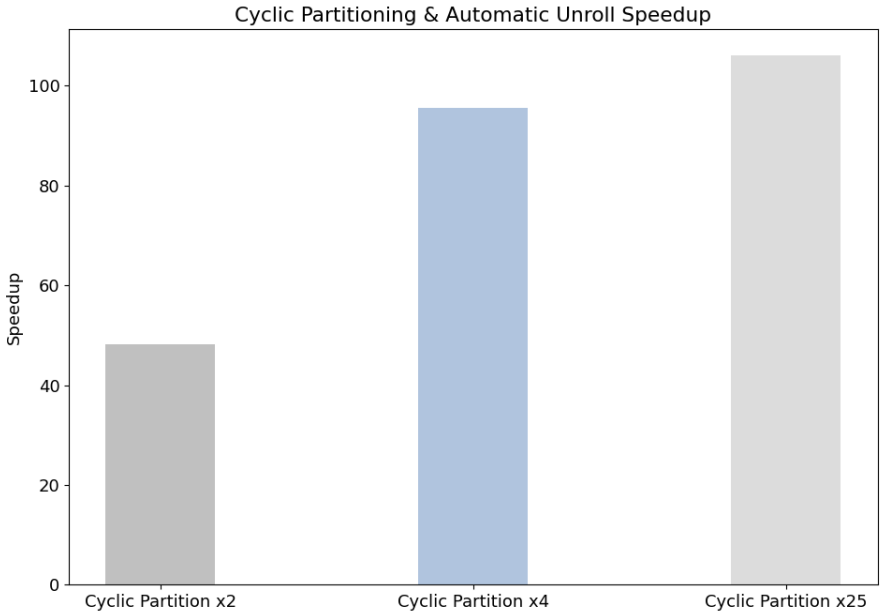


Figure 4.16: Combinational Unroll Speedup

Table 4.2 refers to the *Memory Reshaping* technique and contains the results when different reshape factors are applied to the data. The first column results refer to the loops remaining as they are, whereas the second ones are the results from pramga unroll effect. There are a couple of things to notice: the -no pramga- architectures are slower than the original version of the design since some logic is added to resolve the indexing of the matrix, however applying the pramga addition seems to overcome the bottleneck presented in Figure 4.10.

Table 4.2: Tile Prediction Performance - Memory Reshaping

Design	**No Loop Unroll**	**Auto Full Unroll**
Original	7095ms	282ms
Reshape x2	6162ms	144ms
Reshape x4	7351ms	70ms
Reshape x8	7580ms	67ms
Reshape x16	7095ms	69ms
Reshape x32	7863ms	70ms

Applying *#pragma unroll* to the Memory Reshaping technique leads to resource consumption identical to the Cyclic Partitioning with #pragma unroll. In general, these two combinatorial designs are very similar as far as performance and resources are concerned. However, Memory Reshaping will be extensively used from now on, as this design strategy helps us utilize and experiment with FIFO buffers effectively, as will be examined in Chapter 6. What is

more, Reshape factor x4 is selected instead of 8, since it occupies 5% less wires, which is crucial when scaling will be issued in the next Chapters.

Optimization of Arithmetic Operations

Automatic Tree Balancing

As a last optimization technique, we will deal with balancing floating point operations in our design. Intel SDK for FPGAs provides the ability to instruct the compiler to follow a balanced-tree implementation when the order of arithmetic operations is somewhat complex. By assigning *-fp-relaxed* option while constructing the architecture, the compiler tries to expose the accumulations of an expression in order to prioritize heavier computations, such as multiplications and divisions, before additions or comparisons.

Consider an example like this

$$result = (((A * B) + C) + (D * E)) + (F * G);$$

The compiler implements this equation based on parenthesis priority if no balancing option is specified, resulting in a schematic design as seen in Figure 4.17. However, specifying the *-fp-relaxed* option, the resulting design is depicted in Figure 4.18. This balancing generates less consuming hardware implementations since less control logic is required for intermediate stalling. Another suggested technique from *Best Practice Guide* is that of floating operations' rounding reduction, achieved by adding the *-fpc* option during compilation time. However, this option is not realized in our architecture, since we are already disturbing data precision by storing back half precision floats of 16-bits as we saw in the previous Section.

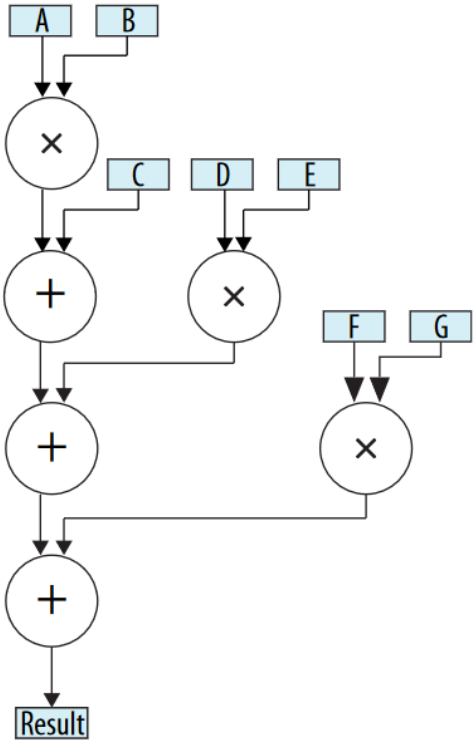


Figure 4.17: Imbalanced Tree Structure

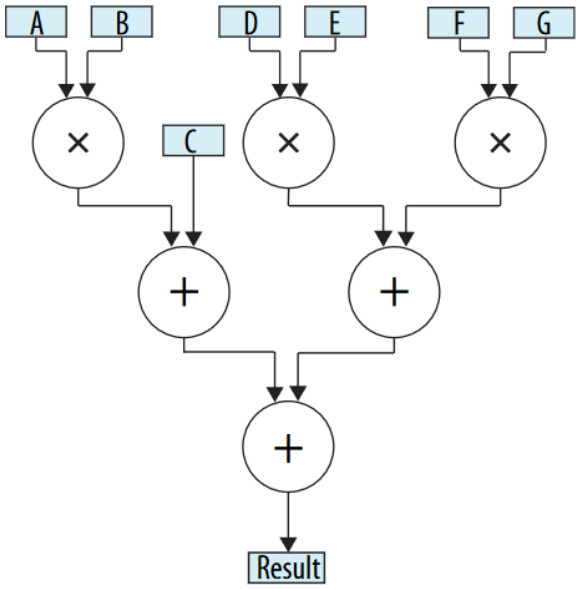


Figure 4.18: Balanced Tree Structure

This compilation option is an automatic implementation of tree-structured calculations. Typically, if we design the architecture in a manual tree-based way, e.g. as in Figure 4.12, informing the compiler to use *fp-relaxed* fails to build the design and the compilation stops immediately. However, applying this option to the rest of the designs helps reduce drastically the area consumption. **RAMs** get **31%** less utilized, **ALMs 2%** and **FFs 11%** decreased whereas

the achieved **Fmax** is increased by **2.5%**. Lastly, since the order of operations changes, we checked whether the results' quality drops off. As a worst case scenario, the results differ by **0.001%** when compared to the original.

Avoiding Expensive Arithmetic Operations

Last but not least, dividing with any *value* has been exchanged with multiplications with *1/value*. Cases where this change applies is during division of the dataset with the standard deviation value and L2.Normalization on output array. By making this adjustment, resource consumption is heavily decreased.

Table 4.3 shows briefly how multiplying instead of dividing in every occasion affects the area consumed. It is crucial to produce low-area cost and low-latency performing designs because in the next Chapters scaling and scheduling will be tested. The quality of the output results remains unchanged throughout the tested samples, whereas the execution time gets a **2.4%** boost alongside the area gains.

Table 4.3: Applied Directives and their parameters

Resource	**% Decrease**
ALUT	75
FF	56
RAM	33
DSP	27
Logic Utilization (Wiring)	16

Chapter 5

Latency Optimized Architectures

Coarse Level Parallelism

In Chapter 4 we examined ways to effectively create RTL based on structural modifications of the C code as well as HLS directives and strategies that refer to the micro architecture of the accelerator. Although we achieved a first level of optimization, the performance can be further improved by combining these modifications with HLS directives that refer to the general architecture of the build. In this section, we are going to showcase how OpenCL and Intel FPGA SDK helps us achieve coarse-grain parallelism and how it impacts our accelerator overall.

Coarse-grained parallelism means a program is split into large tasks that execute in parallel[16]. OpenCL and its HLS directives offer multiple ways to optimize an application, the selection of which depends on the nature of the algorithm under study. For our case, the selection of directives for the Satellite Image Encoding application is driven by our desire to accomplish instruction level parallelism. We will inherit our best performing and most efficient micro architectures from Chapter 4 and use them to synthesize a solution that is latency optimized through some HLS directives. For this reason, we will use as a base implementation the one that contains:

1. Buffers, visible only to the work-item that utilizes them. Global data are firstly imported into the kernel and saved temporarily into private memories (4).
2. Buffers that hold the data inside the kernel get reshaped into a structure of size 25, each containing 4 elements (4).
3. Each loop of the design gets further automatically unrolled (4).
4. During compilation, *-fp-relaxed* option is enabled (4).
5. Every division operation is replaced by multiplication (4).

As discussed in Section 3, the total size of a problem is defined by NDRange. The NDRange is decomposed into work-groups forming blocks that cover the index space. Each work-group is composed of work-items. The index space is also known as **global size** and the work-groups as **local size**. Referring to *Intel FPGA SDK Best Practice Guide*[3], a first level strategy of improving our kernel's performance is by specifying the **max_work_group_size** or **reqd_work_group_size**. These attributes allow the compiler to perform aggressive optimizations to match the kernel to hardware resources without excess logic, thus scheduling work-groups more effectively. Specifying a smaller work-group size than the default, that is 256, might lead to excessive hardware consumption. The *reqd_work_group_size* attribute instructs the compiler to allocate exactly the correct amount of hardware to manage the number of work-items per work-group. This allocation results in hardware resource savings and improves efficiency in the implementation. Last but not least, host side code has to match this exact local size, otherwise the kernel fails to launch during run-time.

Another HLS directive we experimented with was **num_compute_units** and **num_simd_work_items**[12]. Both attributes boost throughput by increasing the amount of hardware on the platform. The **num_simd_work_items** attribute modifies the amount of work a compute unit can perform in parallel in a single work-group whereas **num_compute_units** modifies the number of compute units to which work groups can be scheduled, which also modifies the number of times a kernel accesses global memory. A graphical comparison between these two is found in Fig.5.1. More often than not, the best practice is for both techniques to be applied on the architecture at the same time, as seen in Fig.5.2. Next subsection will present an exploration on the impact that the aforementioned directives have on our best performing architecture.

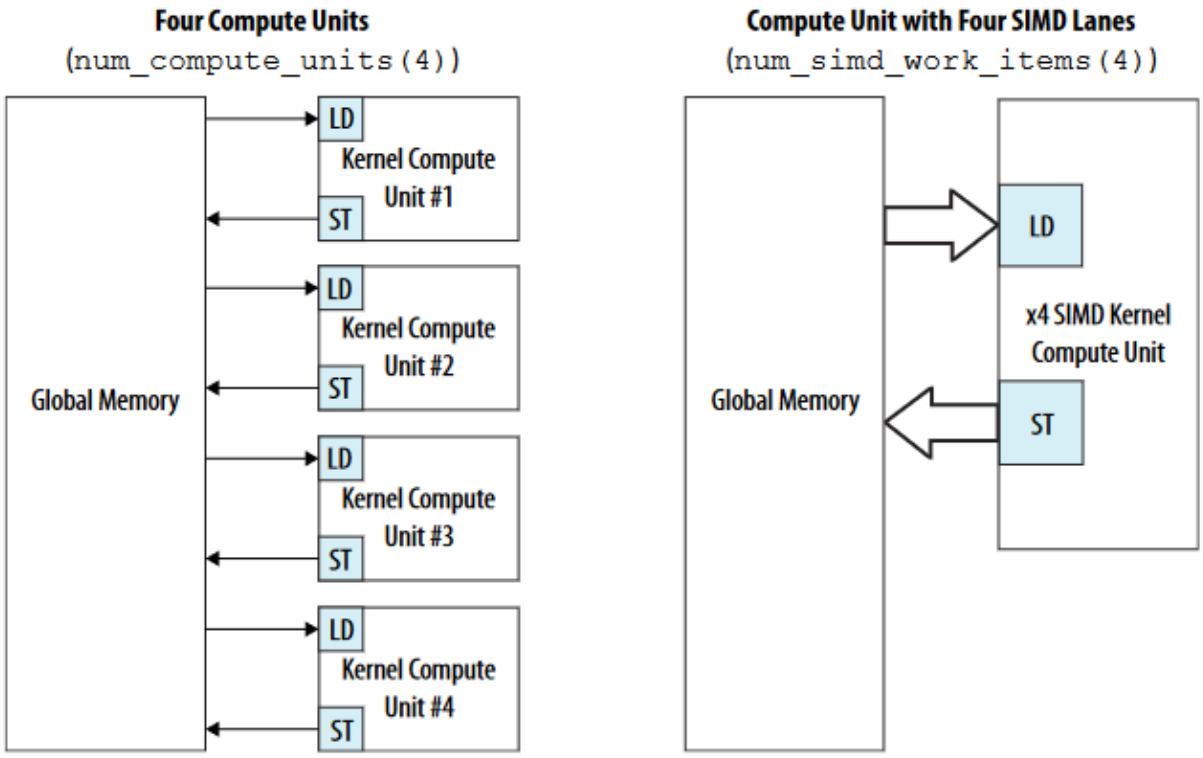


Figure 5.1: Compute Units vs SIMD Memory Patterns[3]

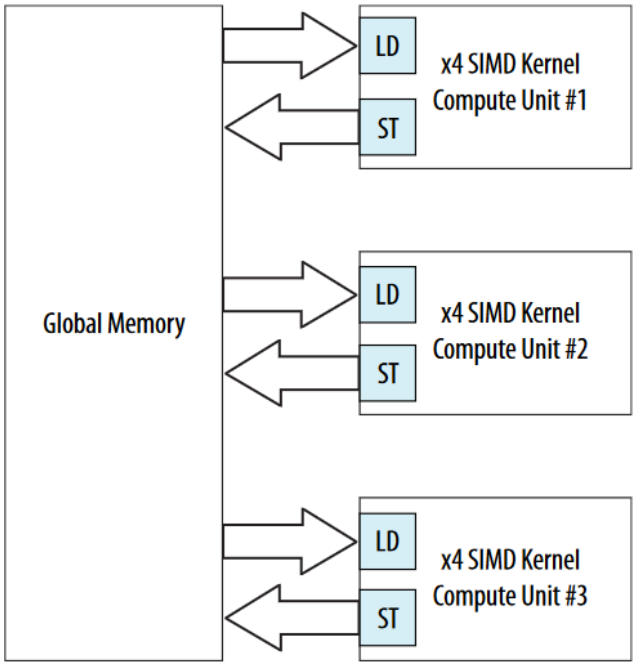


Figure 5.2: SIMD & Compute Unit Replication[3]

Application on Original Code

Increasing the number of compute units in a design is an easy to execute but relatively risky technique for scaling the architecture. The original problem is firstly split into N identical kernels, each one taking over a split amount of the

total problem. There are hence formed N times more global interconnections, which are independently used by the kernels they refer to[9]. A potential bottleneck by using this technique is that wiring gets immensely utilized, resulting in bigger critical paths and therefore the achieved Fmax drops off for each kernel, or in cases, designs might fail to be built. Figures 5.3 and 5.4 present the scaling graph and resources consumed when replicating the compute unit described above. For the case of four compute units, each kernel gets an Fmax equal to 146 Mhz which is a 39% decrease compared to the original one, thus resulting in an identical OpenCL execution time to the three replications, while consuming more area. Although resources are proportionally distributed while replication grows, scaling indicates that when small Image Prediction problems are issued, replicating the kernel does not result in an observable acceleration. However, as the problem size grows, so does the value of the technique. In Full Image prediction, three compute units with averagely three times more area utilized, score 2.33x faster than the base one, whereas two compute units complete the task in 1.88x less time.

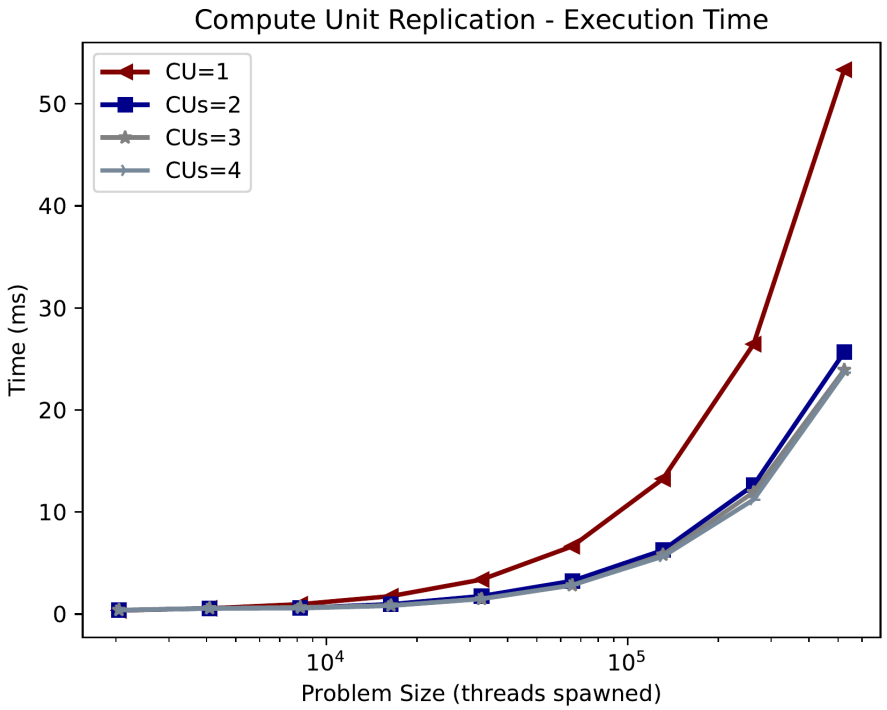


Figure 5.3: Scaling Graphs for Multiple Compute Units

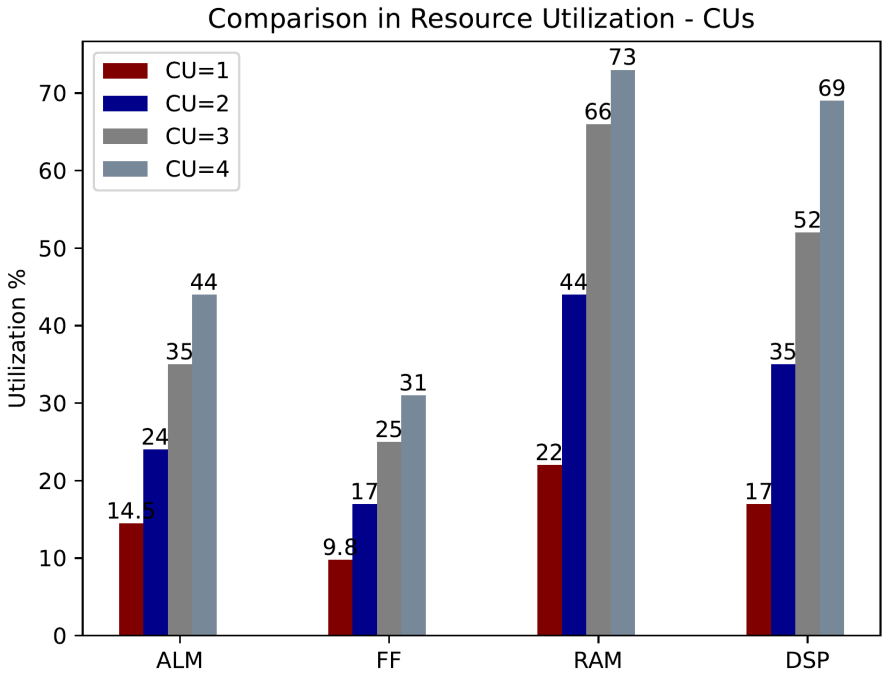


Figure 5.4: Multiple Compute Units Resource Consumption

As far as the Single Instruction Multiple Data method is concerned, the two following graphs show how the execution time scales and how many resources are spent on them. The problem of Fmax is visible here for the SIMD=4 case. Table 5.1 contains the achieved maximum frequency for every case, with maximum among them being SIMD = 2. Instructing the kernel to issue 2 data values at once at every operation speeds up the design by **49%** whereas quadrupling this instruction results to a **10%** additional speedup with resources not following the performance gain and instead getting rather bigger.

Table 5.1: Fmax for SIMD

No SIMD	**SIMD = 2**	**SIMD = 4**
256 MHz	295MHz	219 MHz

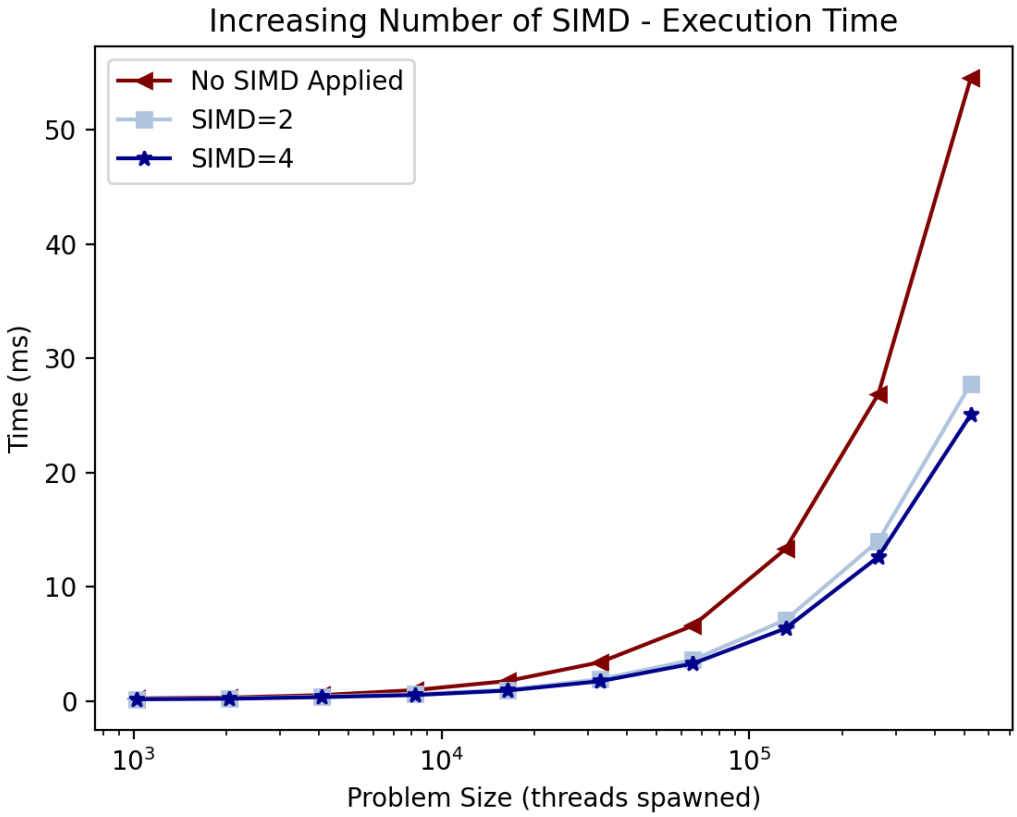


Figure 5.5: Scaling Graphs for SIMD

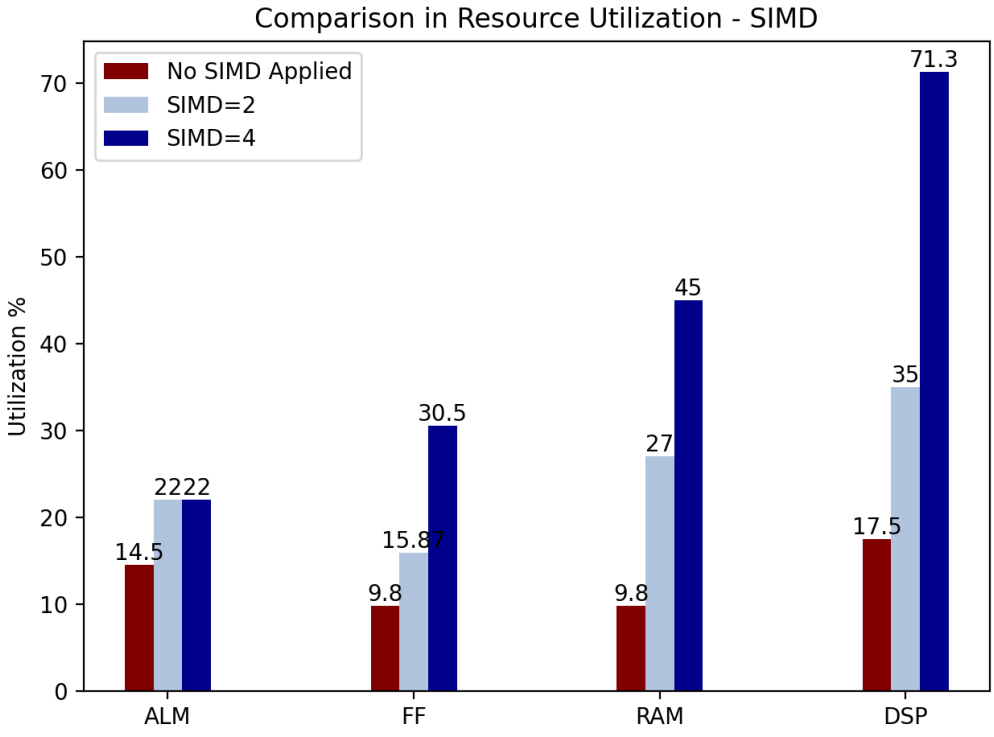


Figure 5.6: Resources Graphs for SIMD

To complete our exploration space concerning the replication of the kernel and the usage of SIMD architecture, we checked how a combination of both these techniques affects the design overall. Fig.5.7 shows a cumulative comparison of the HLS strategies. It is obvious that replicating the kernel 4 times and using SIMD equal to 4 do not produce the desired performance, since they achieve same or less speedup compared to other designs while steadily consuming more resources. Although combining the techniques offers a slight performance advantage, wiring utilization reaches its limits. Instead, we will use the design which contains the optimal kernel, replicated 3 times and further explore the other strategy discussed in the beginning of 4 about local size.

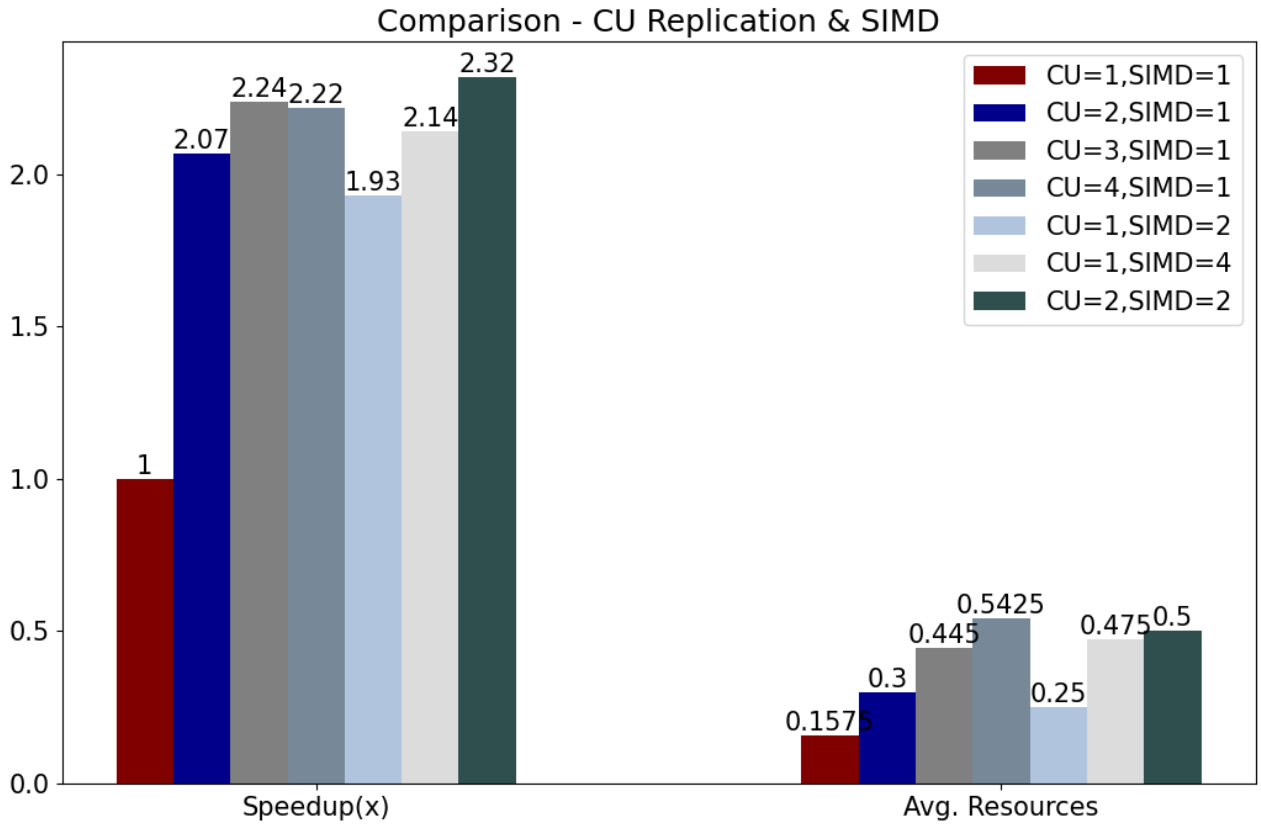


Figure 5.7: Compute Units and SIMD Comparison

As a last metric for this Chapter, local size is issued. Concerning a single compute unit and local sizes up to 512 that are a power of 2, the performance seems to be identical as the problem grows, Fig.5.8 shows. Specifically, for a low thread number, the performance varies for a maximum of 10%, but while the problem size widens, this variance drops to about 1%. Fig.5.9 shows how the scheduler copes with the threads when 3 compute units are realized. One might notice that *no size specified* and *64 local size* are absent. This occurs because compiler fails to build the design, as it lays an immense amount of workload onto the scheduler that cannot be programmed. For a thread count of 128, 256 and 512 the architecture seems to work reliably, whereas increasing

the locality size leads to an unpredictable behaviour. The performance spikes for both high and low data thread loads.

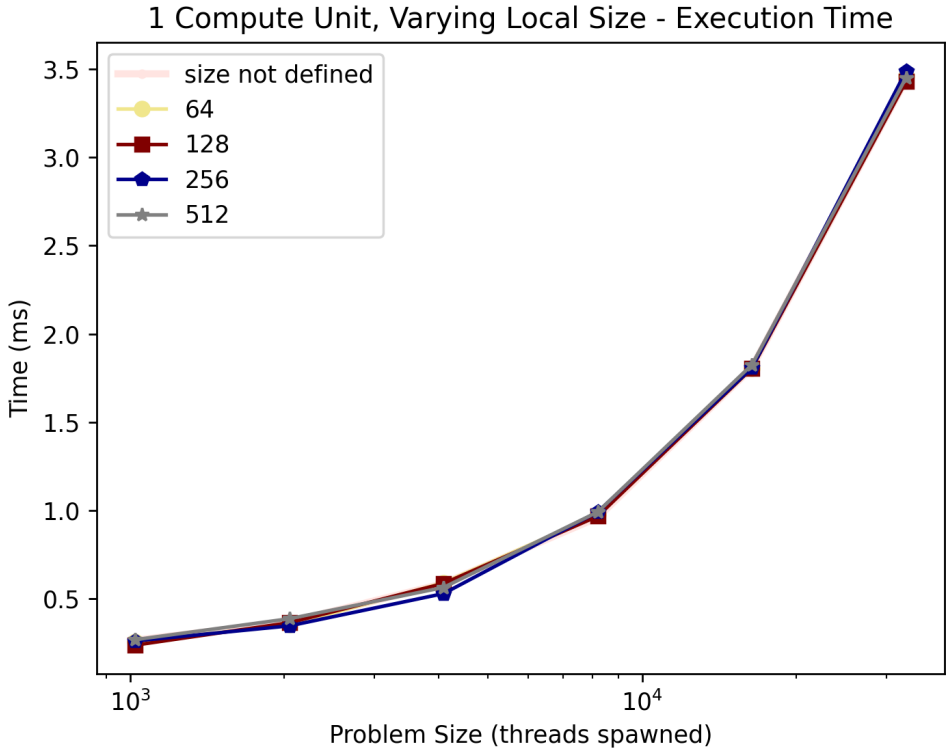


Figure 5.8: 1 CU - Local Size

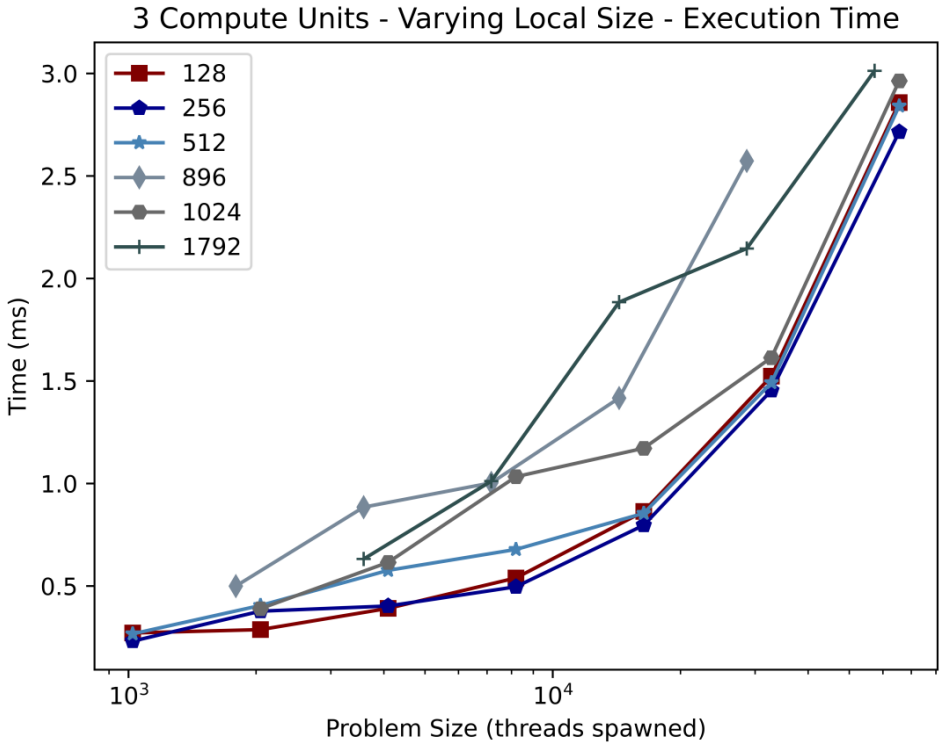


Figure 5.9: 3 CUs - Varying Local Size

Last Performance Tips

Measurements so far have made it clear what our best design for a latency optimized architecture should look like. 3 compute units are utilized, each one requiring 256 work-items per work-group. The kernel body is the one described in Listing 4.1 with *#pragma unroll* added before every kernel. Input data are read from global memory and written to a private structure of size [25,4] responding to memory reshaping equal to 4. Weights are passed as constant floats and data in 8-bit unsigned character form. Output is in half precision mode to reduce output data movement overhead. Dividing a dataset with a value, as in standard deviation or L2.Normalization part, is replaced with multiplication with 1/value for less area and faster operations. The overall design is speeded up by a **213x** factor when compared to the original prediction model, presented in Section 4.

As some last optimization tips, further reshapings were issued for the data. Reshaping the data into a [4,25] matrix instead of [25,4] speeds up the design by 18% in OpenCL execution time. Furthermore, informing the compiler about the values of the weights as a *hw_parameter* matrix, that means writing weights in a separate file and using them as constant instead of a kernel parameter, is beneficial for the performance. Figures 5.10 and 5.11 present how scaling and resource consumption changes when they are written as static values. RAMs are used more whereas FFs and ALMs less, but performance gets an overall increase by almost 22%.

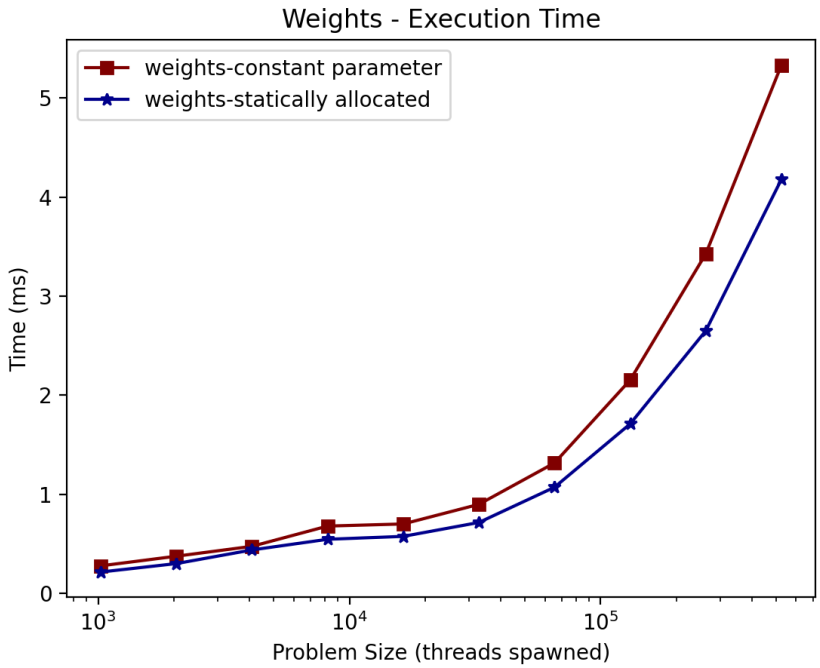


Figure 5.10: Weights scale comparison

To sum things up, the latency optimized design space exploration achieves its goal. The overall latency gain when compared to the unoptimized one is equal to **330x**. Integration Chapter7 will further compare this performance with software counterpart and the limitations put by data movement. Next Chapter, however, focuses on creating a throughput-optimized designs, using as less as possible resources while performing as effectively.

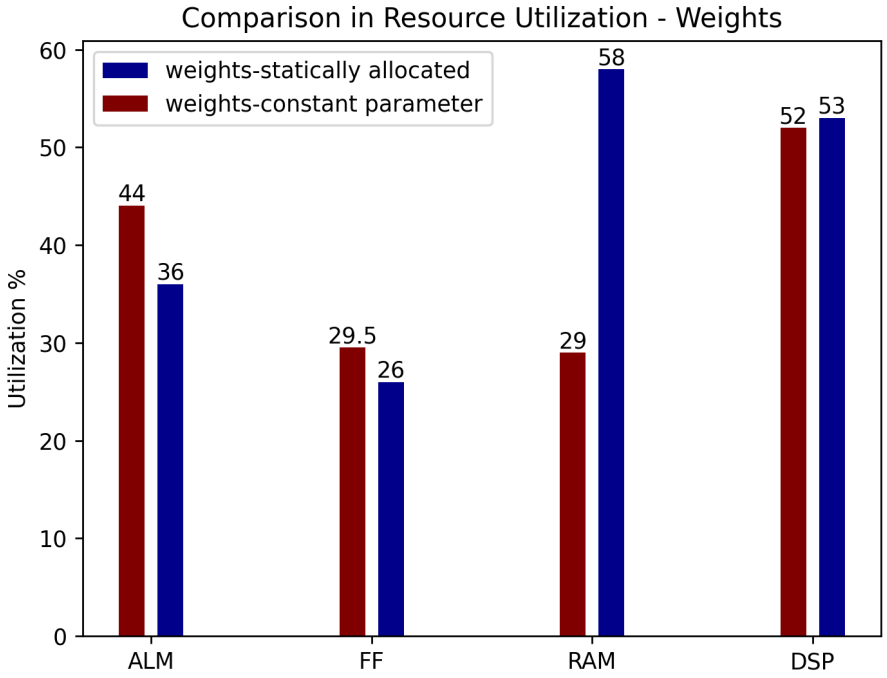


Figure 5.11: Weights resource

Chapter 6

Throughput Optimized Architectures

Fine Grained Parallelism

In contrast with the case we examined in Chapter 5, this part of the thesis focuses on creating fine-grained architectures, capable of predicting a large amount of sub-images while consuming less resources. The idea is that images are being sequentially streamed into the design and thus we are trying to focus on a more throughput-based architecture. Fine-Grain Parallelism means a program is split into tasks which execute in parallel and communicate with each other when sharing results is necessary[16]. In Satellite Image Encoding Algorithm, the parts that the design can be broken into are rather distinct and *Channels* will be utilized to achieve high speed communication between the kernels. We, hence, enforce the application to acquire a pipeline-execution model, which is not available by default when creating NDRange kernels in OpenCL. The main parts of the code are :

1. Mean Reduction.
2. Standard Deviation Division.
3. Dense Layer
4. L2 Normalization

Each one of these semantically defined elements will be constructed as separate kernels.

The mechanism which provides the ability for data to be shared among various kernels is ***OpenCL Channels***. Channels are FIFO buffers, allowing kernels to communicate directly with high efficiency and low latency and are enabled to be used by adding :

```
1 #pragma OPENCL EXTENSION cl_intel_channels : enable
```

in the code. In the majority of cases, due to memory access patterns inside the algorithm, FPGA cannot utilize the maximum memory bandwidth, available in

the device. A way to overcome this bottleneck is through *private memory* integration as we saw in Section 4 which results in a rather big MLAB utilization. Another way to highlight the maximum available memory bandwidth is with channels, which realize a smaller amount of memory blocks overall. Fig.6.1 presents how memory interconnections are replicated when compute units in Coarse-Grain Parallelism are created, each compute unit being responsible for predicting a portion of the total problem from start to finish, whereas Fig.6.2 indicates how the algorithm is approached when FIFO buffers are used. Each distinct kernel calculates a part of the prediction model algorithm and writes the results to a buffer-channel. The channel is then read by the next kernel, and so on, up to the point where data are written back to memory. In reality, Fig. 6.2 depicts our first implemented architecture when fine grained parallelism is approached.

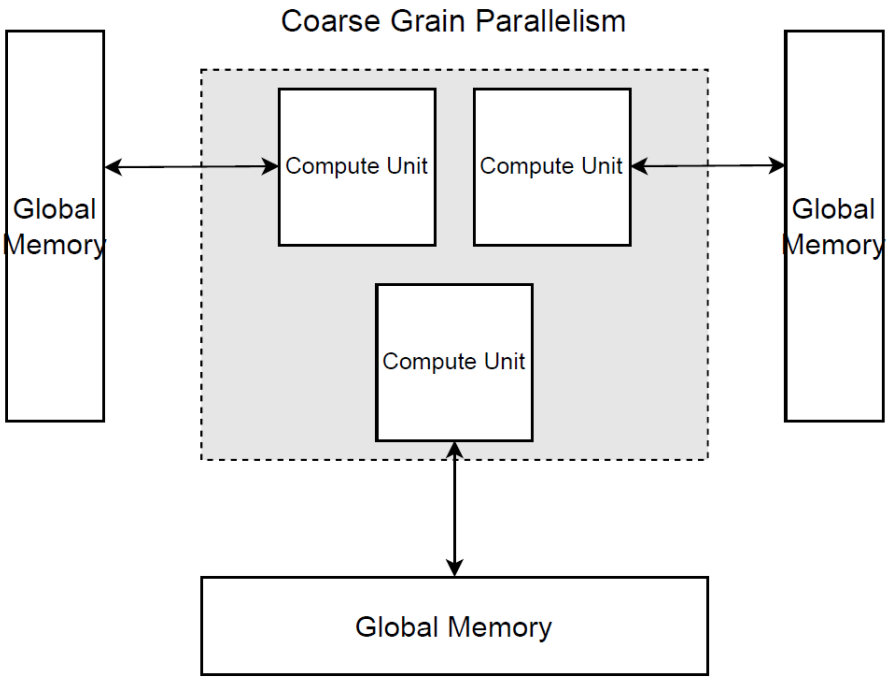


Figure 6.1: Coarse Grained

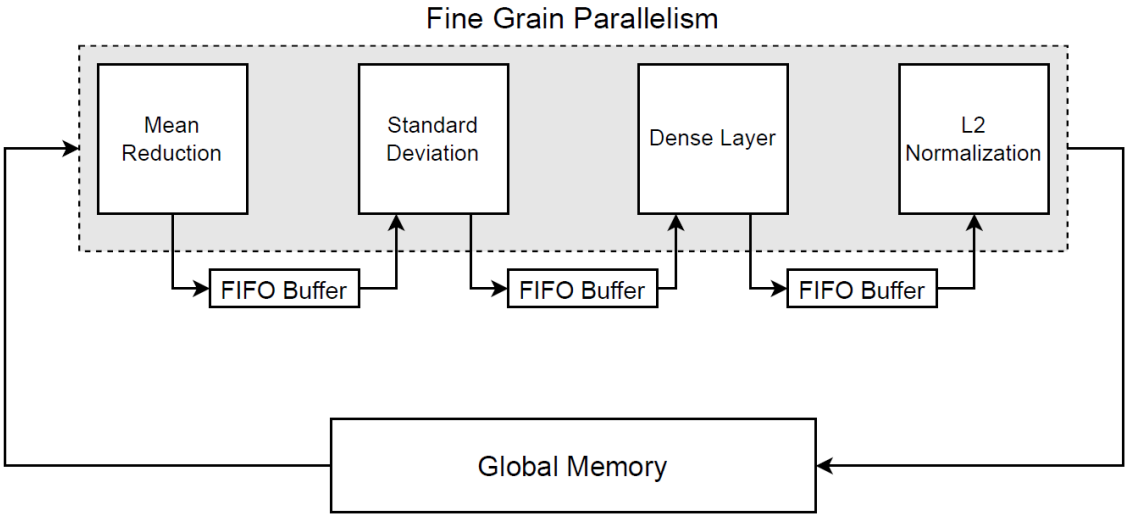


Figure 6.2: Single Pipeline Implementation

A major issue concerning channel buffers is multiple work-item ordering. In general, Intel FPGA SDK consults the designers to avoid using channels alongside NDRange casts, as channel-ordering problems might occur and output values being wrong. However, multiple work-item accesses can be useful in scenarios where data in the channel are independent or when they are implemented for control logic. In our case, each element is independent from one another and we are certain that every kernel executes the same amount of channel accesses, since no branch conditions in channel reads or writes are presented. A deterministic ordering of the multiple work-items' accesses is hence guaranteed. Even though by defining local work size in the kernel, we instruct the FPGA to execute multiple work-items in parallel, channel execution follows a serial execution pattern. This means that work-items with smaller IDs execute first. Channels support a maximum of one write and one read per clock cycle and can be implemented both in a blocking and a non-blocking way. Here, blocking reads and writes are implemented as we are trying to avoid non-deterministic patterns in the design which may lead to inaccuracies or executions failing to complete.

In order to further explore architecture optimizations, a general analysis of the code and implemented kernels is needed. Latency is not equally distributed across the kernels, thus a potential bottleneck appears in the design described in Figure 6.2. Mean Reduction kernel's performance is lowered due to loading the images from global memory. Ideally, we would like each kernel to read data in the fastest possible pace from channels rather than communicating with global memory. To alleviate this latency bottleneck from Mean Reduction kernel, an input kernel and as well as a buffer intermediate one are implemented. The complete architecture is depicted in Figure 6.3. Input kernel reads data from global memory and writes them to a channel. Of course, mean reduction kernel

could instantly read from that channel, however no problem would be solved, as further stalls would occur between mean kernel reading from channel and input kernel writing on it. By realizing an intermediate kernel, working as a buffer, mean reduction can read the input while input writes to buffer.

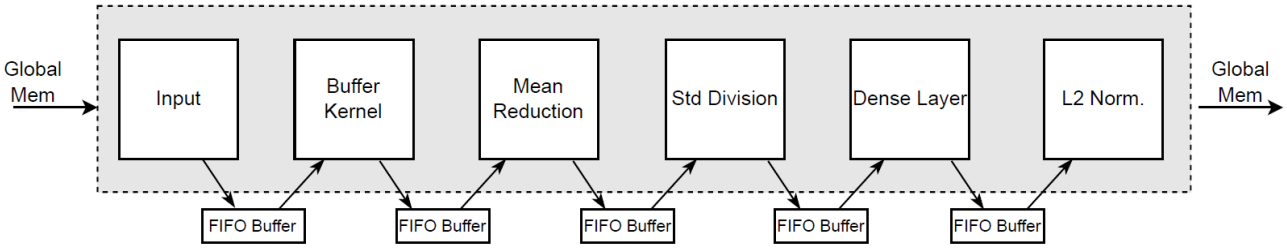


Figure 6.3: Single Pipeline & Buffered Input Implementation

Channels cannot be used alongside *num_simd_work_items*, that means kernels using channels cannot be vectorized. Vectorizing such kernels creates multiple channel accesses inside the same kernel and requires arbitration, which negates the advantages of vectorization. However, a last technique involving hard-copying the kernels and channels twice is presented here, as seen in Fig.6.4, which can lead to two images being casted at once and executed in parallel using their own Fine Grained Architecture. Next Section contains measurements about scaling and resource consumption about all the designs.

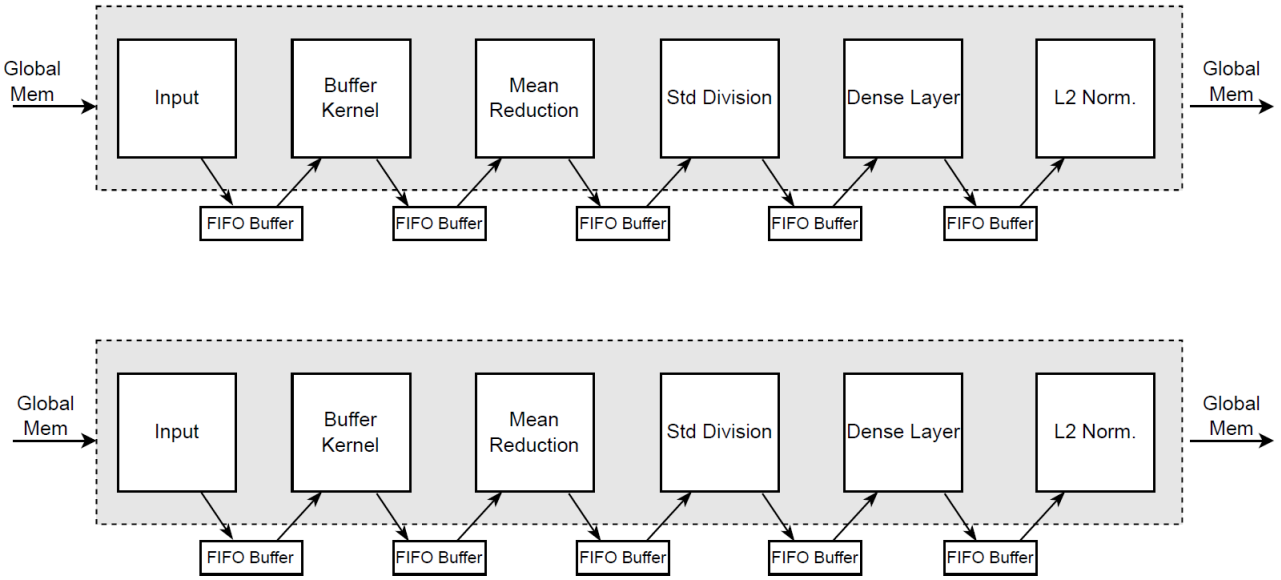


Figure 6.4: Double Pipelines & Buffered Input Implementation

Application on Original Code

The Application part of the code contains measurements from the three, so far, discussed architectures as well as a wider exploration concerning the amount of writes that occur in a single cycle relatively to the amount of pipelines implemented. Concerning micro-architecture specifications, we will once more use the best optimization techniques that resulted from Chapter 3 analysis. Data are inserted in *unsigned character* form and stored in a *reshaped structure* of size [25][4]. Every loop in the designs, other than the loading and outer Dense loop, get fully unrolled by *#pragma unroll* and inside every loop, manual unroll follows the data structure. Data are stored back in *16-bit float precision* and every distinct kernel requires a fixed number of *256 work-items* as its local size. The two Figures below present how the prediction model scales by utilizing these techniques and how different the resource consumption is, when compared to each other.

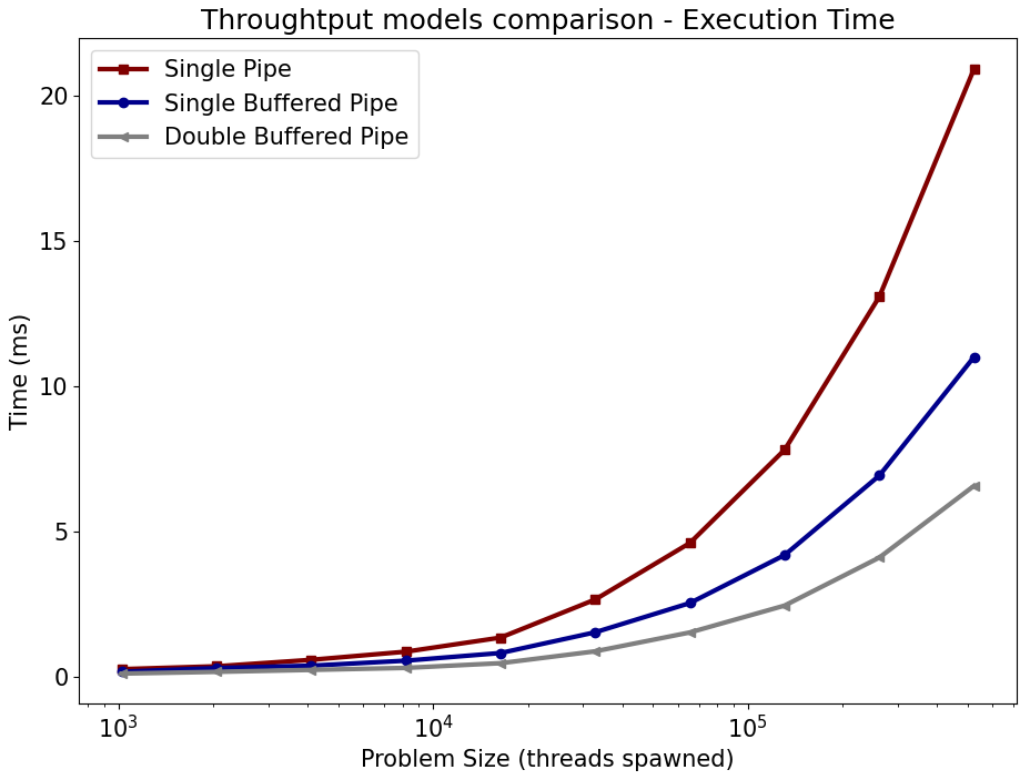


Figure 6.5: Performance Scaling - Throughput

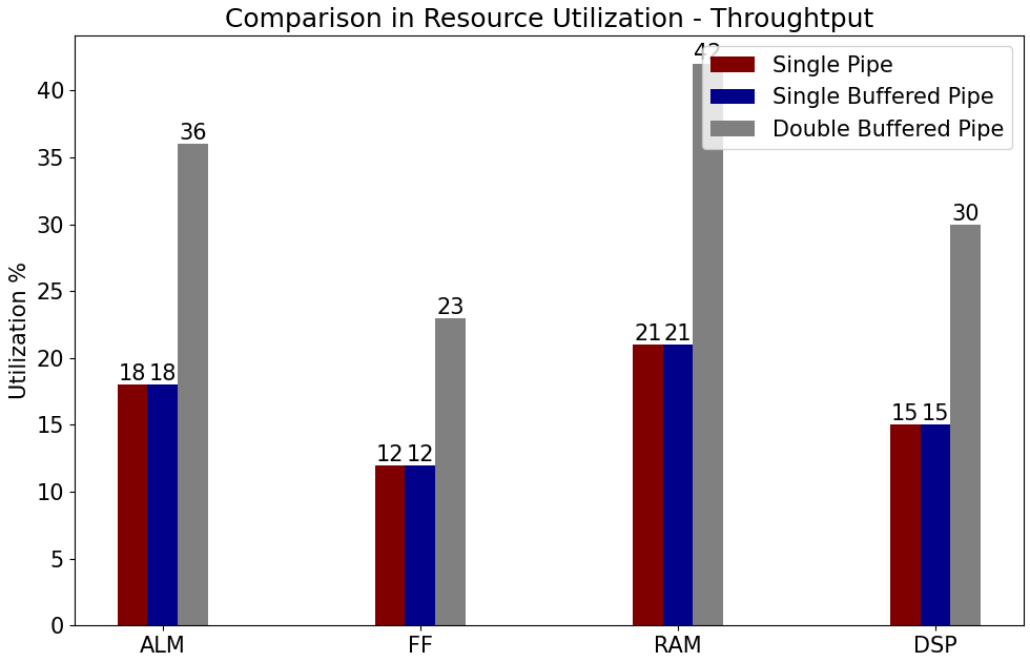


Figure 6.6: Resources - Throughput

Realizing an input kernel and an intermediate buffer speeds up the design by about **35%** without consuming more resources, whereas duplicating it results in a **65%** increase in performance. Of course, duplicating the design means RAMs and DSPs are consuming proportionally more area. When comparing to the latency optimized builds of Chapter 5, we notice that for a lower number of threads, latency optimized builds score higher performance, however as thread size increases, throughput based architectures are gradually closing the gap and even outperform the latency optimized ones. This comparison refers to identical builds, for example double throughput based architecture and 2 compute units from latency optimized design. In a real-time application, if we could implement the application to be running constantly, throughput based architectures should be implemented, whereas if 2 images need to be predicted statically, latency designs should be considered.

In order to conclude the exploration of the best architecture, a case study concerning multiple pipelines and writes per iteration is presented. Channel-buffers in OpenCL are elements of variable width and can be altered to fit wider words. This feature will be exploited to design an architecture that writes more than one element on the buffer at once. Memory Reshape technique we examined in Chapter 4 will be extensively used to achieve this kind of communication between separate kernels. Of course, in order for the rest of micro-architectural options to remain the same, input and output logic has been added to match the communication standard whereas the core of each kernel remains untouched. The *Pipeline* variable is presented to showcase that more than 2 parallel pipelines will be realized. However, when pipe amount

reaches 4, all designs but the original fail to build due to extensive resource usage. Figures 6.8 to 6.11 depict in which way the execution time changes for a specific write/iteration number compared to the amount of pipelines implemented. Fig.6.7 presents a cumulative comparison in the speedup achieved when compared to 1 write/iteration for different pipeline architectures. By observing these results, we can deduct that for a single and three implemented Pipelines, writing 4 elements at once on the channels performs the best. On the contrary, when 2 Pipelines are realized, writing 16 elements at once on the buffer offers a slightly better performance.

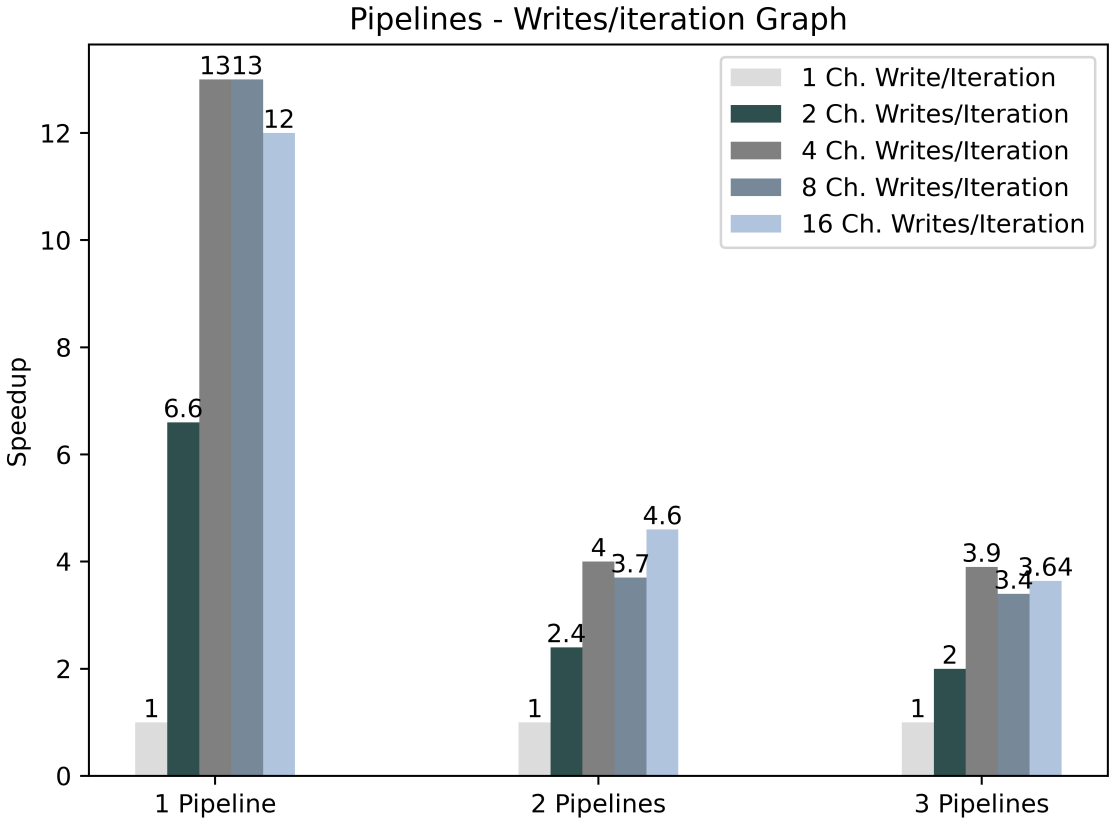


Figure 6.7: Pipeline and Write/Iteration execution time development

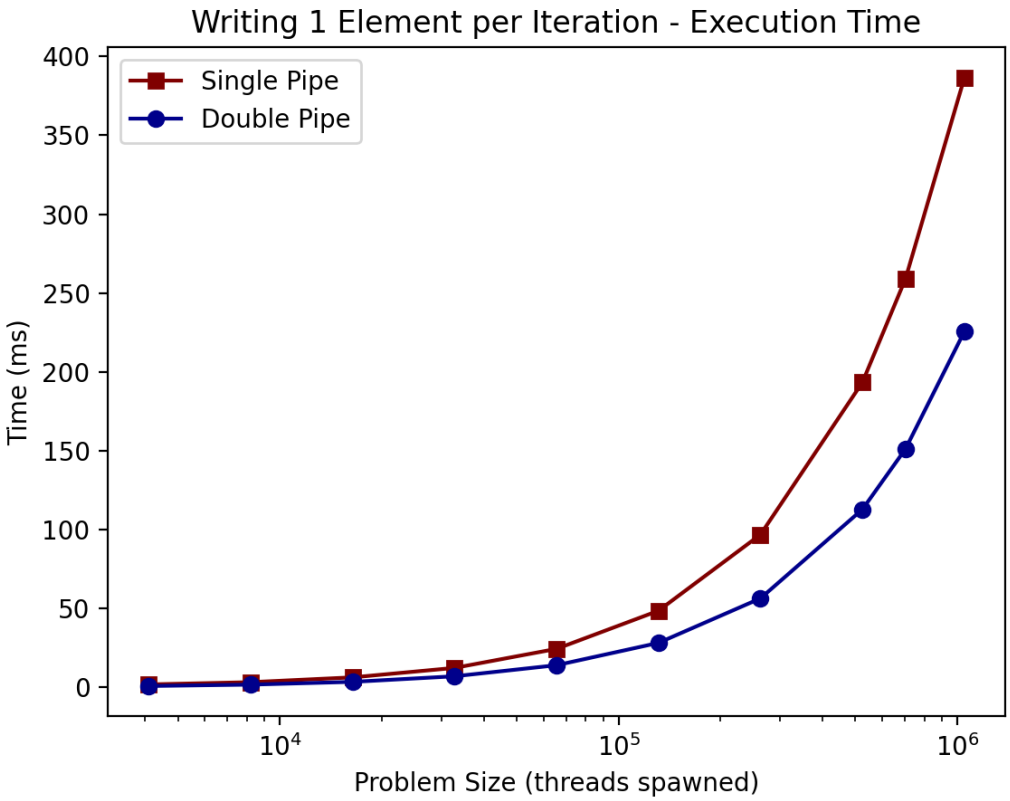


Figure 6.8: One Element Channel Enqueuing

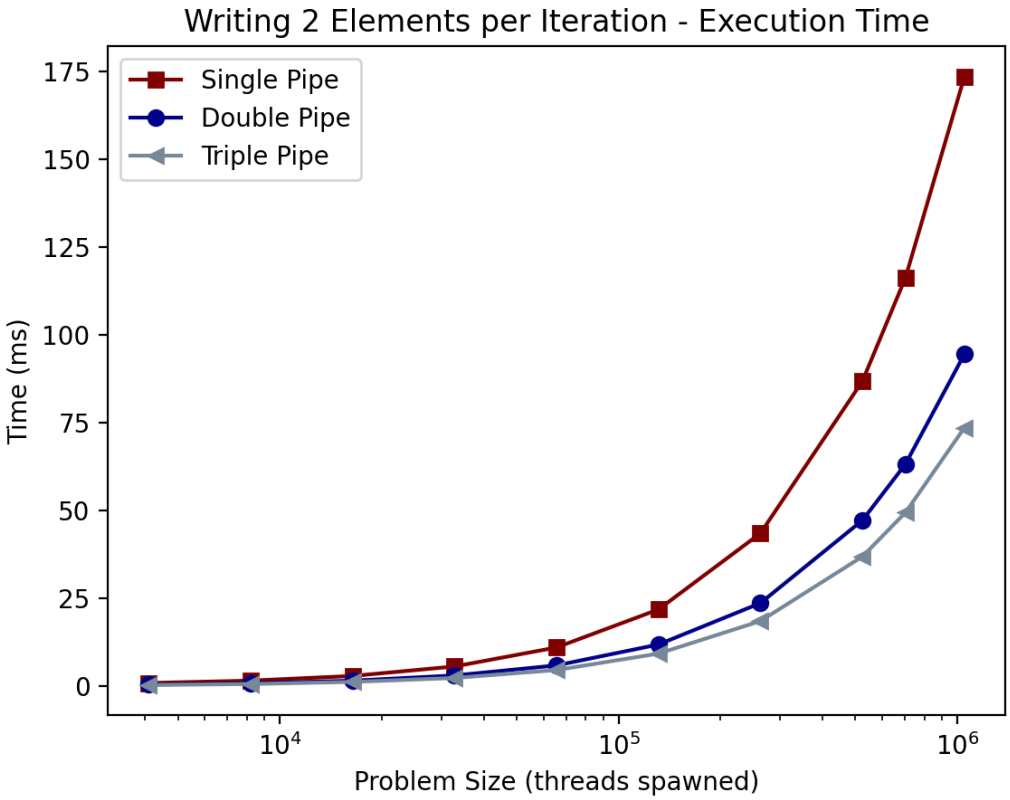


Figure 6.9: Two Elements Channel Enqueueing

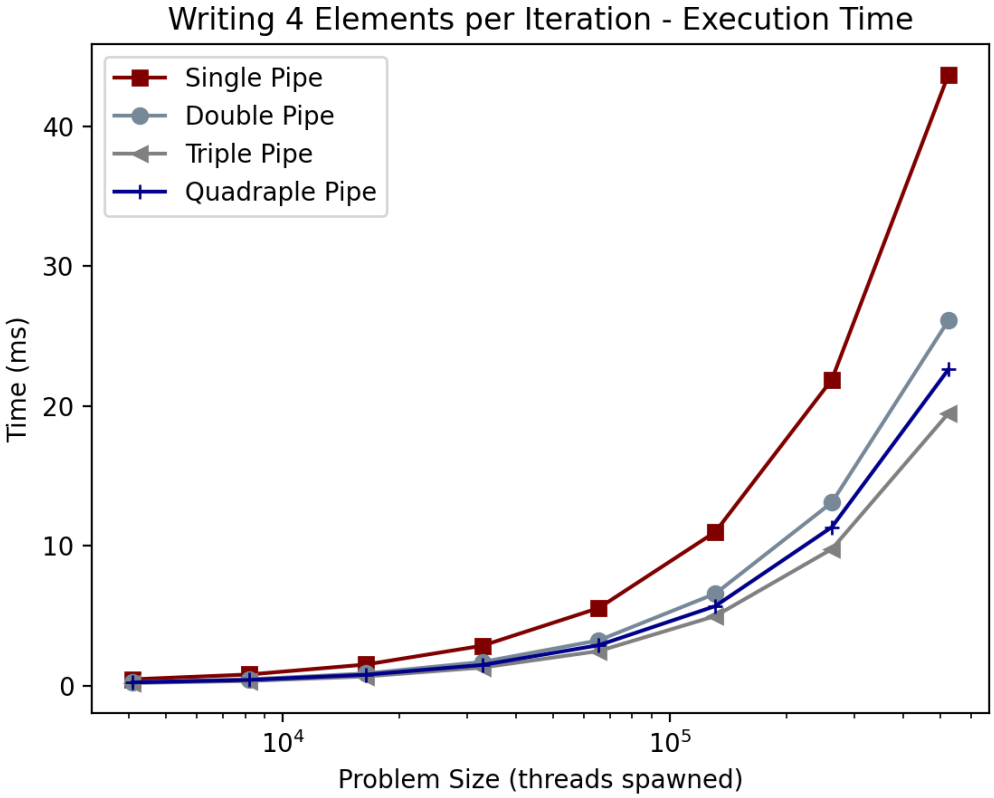


Figure 6.10: 4 items enqueuing

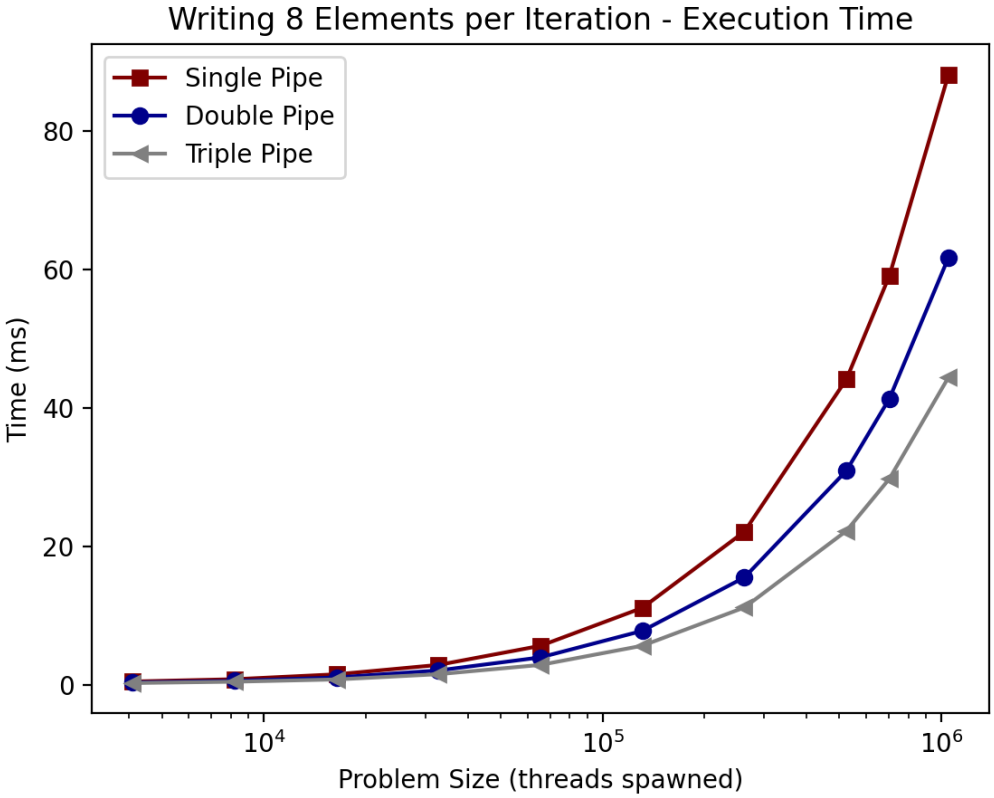


Figure 6.11: 8 elements

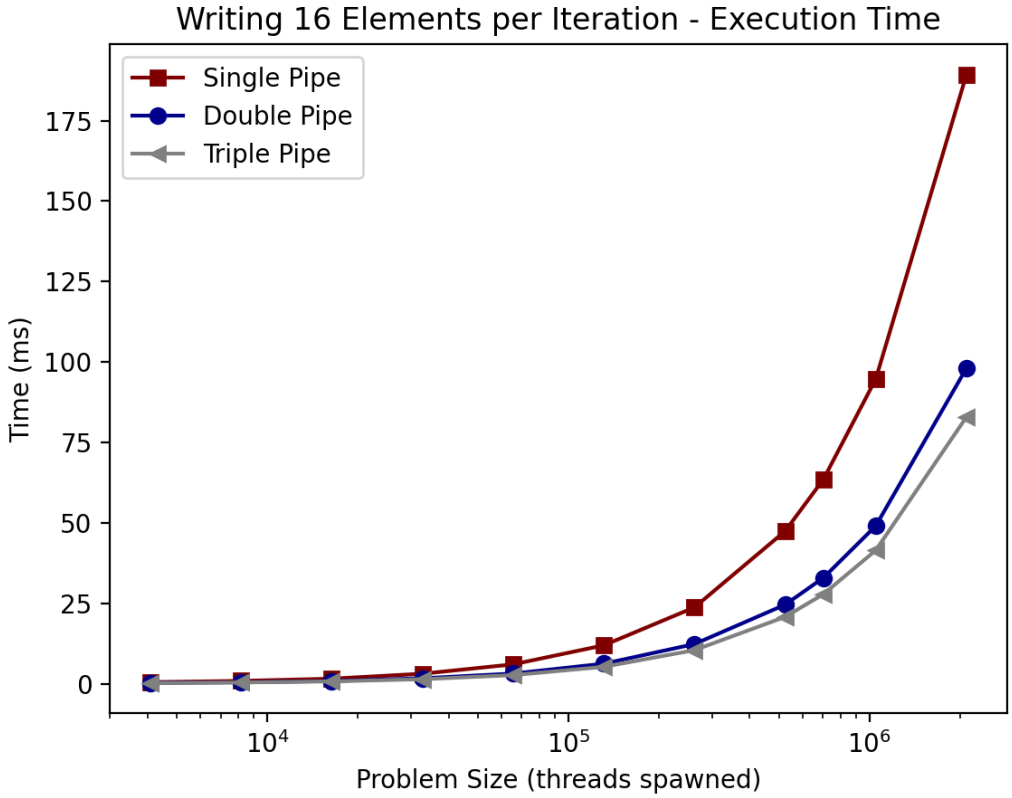


Figure 6.12: 16 elements

Chapter 7

Integration

The last Chapter concerns the procedure we followed in order to integrate the design, both locally, using "dummy" datasets, and in a look-alike environment in addition to the Python software, replacing the compute intensive parts with our code. The first one was done to check data movement overhead whereas the second in order to check the validity in output data when compared to the original software algorithm.

Experimental Setup

Implementation Using Host Code

A host code is used to implement the built design. After the compilation process is successfully completed, a *.aocx* file is created. The host code, usually written in C/C++, loads the kernel into the OpenCL runtime by utilizing the *clCreateProgramWithBinary* function from *CL/opencl.h* library. After the *cl_program* has been created, *clBuildProgram* function is used to create the executable program for the specified device. *clCreateCommandQueue* generates a host command queue for a device, through which kernels, created by *clCreateKernel*, can be run on. *clCreateBuffer* is used to produce the buffers that contain the data necessary for the kernel to run, *clEnqueuWriteBuffer* connects the buffer from host side into device memory and starts passing the data through a PCIE card. *clEnqueuNDRangeKernel* instructs the host runtime to execute the created command queue(s) and thus the realized kernel(s) on the FPGA. This command is cast before *EnqueueReadBuffer* which reads the data from a buffer device object and maps them back to host memory. Once the process is complete, *clReleaseEvent* funtion is called to remove an event from runtime memory[8].

In OpenCL, most runtime API functions return an error code so that error handling is easy to perform, whether successful or not. Events and synchronization are also offered by the libraries. *clFinish(queue)* synchronizes the

asynchronous operations running on the device, such as NDRange enqueuing, by waiting for all events in the device queue to finish. Lastly, events are used to access profiling information for the operation they represent. Queue profiling is enabled by adding *CL_QUEUE_PROFILING_ENABLE* flag in the command queue function and is useful when the profiling of the algorithm is of great importance. In our case, we want to measure both OpenCL device Execution time and data movement, input and output, overhead. The OpenCL execution time was our driving force for optimizing the prediction model algorithm in Chapters 4 through 6.

A significant note to be made at this point would be the difference between executing a single latency optimized kernel (Chapter 5) and multiple throughput optimized kernels (Chapter 6). As far as a single kernel is concerned, only one command queue is created for the process to complete, even if the kernel is replicated by a number of times through *num_compute_units* OpenCL command. The FPGA handles the splitting and scheduling of the total given problem. On the contrast, issuing multiple kernels that communicate through OpenCL channels or pipes, requires the user to manually instruct each kernel to run autonomously as a separate problem solving unit. Regarding the architectures presented in Chapter 6, a command queue is created for every single kernel and therefore, an *NDRange* cast is issued for everyone of them in an ordered style. The user himself has no authority over the channel buffers, rather than simply instructing the kernel that utilizes the channel to execute, as briefly explained in Fig.7.1 below. Host side is also responsible for generating and mapping the input and output data for the FPGA to use as well as their general usage policy, such as *READ_ONLY*, *WRITE_ONLY* or *READ_WRITE*.

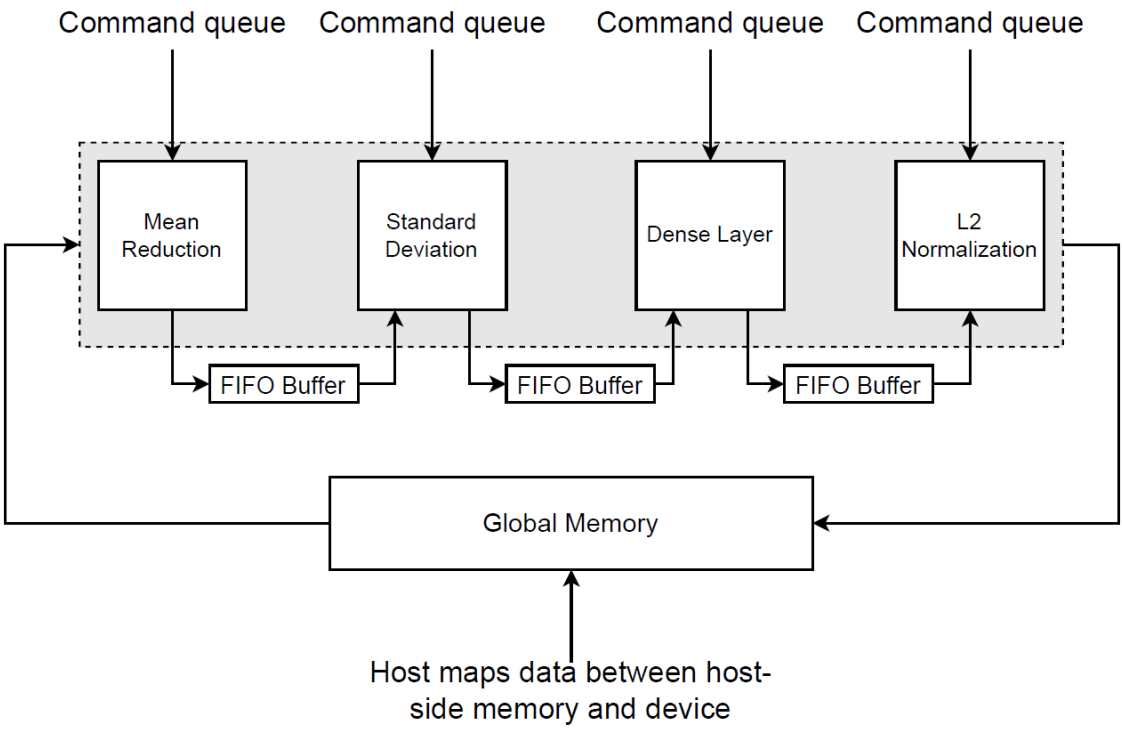


Figure 7.1: Host-Device Execution Preparation

Docker - application deployment

Docker is a software framework for building, running and managing containers on servers and the cloud. A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings. Containerized software will always run the same, regardless of the infrastructure. A schematic representation of Docker is seen in Figure 7.2. The Docker approach to containerization focuses on the ability to take down a part of an application to update or repair, without having to take down the whole app. We will use Docker to build and integrate our application quickly. This way we can control everything, necessary for the application to run. Moreover, when deploying a docker image, we can freely connect folders from our local system to the docker and get root privileges on them.

As we discussed in Chapter 2, the software application utilizes a great amount of Python Libraries to be able to run. A docker image is deployed containing the FPGA device as well as the OpenCL root folders. Inside the docker, we install and troubleshoot potential problems about libraries. Specifically, the original Satellite Image Encoding Software runs on *Python 3.6*, *tensorflow 2.4.1*, *psutil 5.8.0*, *confluent-kafka 1.6.0*, *Pillow 8.1.0* etc. A problem occurs when gdal library, responsible for decoding all geographic location in-

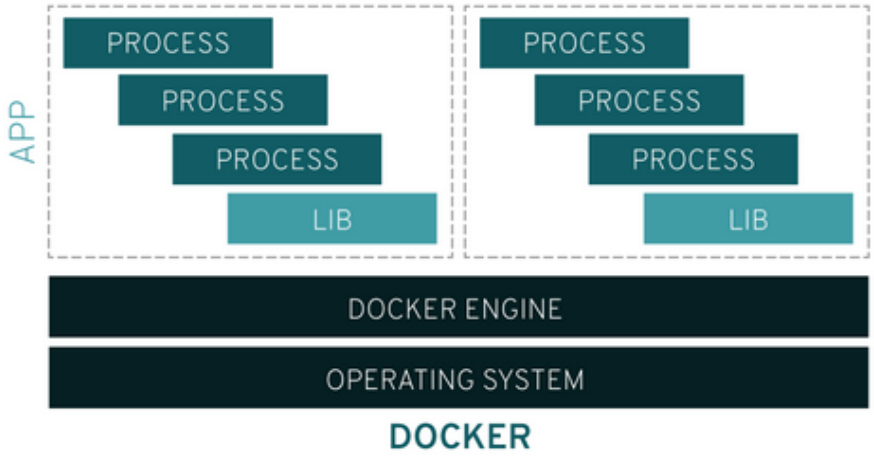


Figure 7.2: Docker Overview

formation in the algorithm, is issued and is resolved by installing with *–global-option=build_ext –global-option='-I/usr/include/gdal' GDAL==%(gdal-config –version)*. Docker was firstly used for Software Profiling, the results of which were presented in section 2.

Python-C++ interconnection

The goal of last thesis' section is to integrate our best performing FPGA-based application into the existing software algorithm. We want to replace the slowest parts of the algorithm with our implementation and check whether a speedup is accomplished and if it is good enough for the application to be used for real-time applications. This will be done by using an extra Python library, called ctypes. Ctypes provides C compatible data types and allows calling functions from shared libraries. The *.aocx* that will be used is the one created and presented in Chapter 5 which is a 330x faster implementation than the original prediction model, presented in 4.1 . As far as the host code is concerned, the base host code will remain the same, adding the extension *extern "C"* in the beginning of the code. The compilation command is changed into *g++ -shared -fPIC -Wl,-export-dynamic* which produces a .so file. Python utilizes the *ctypes.CDLL* method to load the shared object. After having set the input and output arguments' types, acceptable by ctypes, the shared object's function is called by its name. Note that between the two applications, data are being handled in a different way. An extra alternation has to occur in software code before calling the function, and that is all data have to get serialized, a process similar to the models' Flatten Layer.

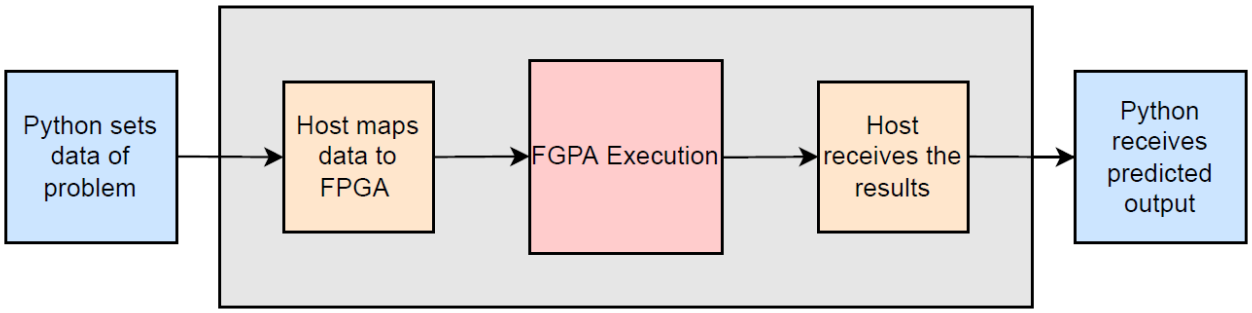


Figure 7.3: Integrated Process Diagram

A complete overview of the integration we have discussed so far is seen in Fig.7.3 and is provided as a schematic explanation of the process we followed in order to integrate the application. The gray-delimited box contains the execution and connection part between host and fpga side, the details of which are found in 7 Section. The host prepares and maps the data before executing *NDRange* command so that FPGA starts processing the data. The part that is found in this subsection concerns the rest of the diagram outside the gray box. Python runs as is up to the point where we want to accelerate the design. At that point, Python calls an *acceleration* function which belongs to the host side through *ctypes*. The interconnection between the data occurs through pointers to memory, both for input and output values. After the results have been computed, Python reads from the output pointer and continues with its execution as before. The part of the code that is replaced by our FPGA acceleration architecture is the one of *Main Computing Loop*, the impact of which is crucial to the total performance of the application, as seen in 2.

Impact of Input Data Compression

Technique

Before deploying the integrated version with Python, global memory interconnection is optimized. As seen in Table 2.1 in Chapter 2, data are extracted in uint8 numpy form and exported in 32bit float precision as an output. In this Section, *Input Data Compression* will be issued for our prediction model and tested in order to check whether we can take advantage of simpler to manage datatypes without losing in output precision.

Utilizing private memories in the kernel lets us choose the preferred datatype independently from global memory interconnection. OpenCL contains a suite of functions called *convert_T* which provides a full set of type conversions between supported scalar and vector datatypes. Furthermore, *vstore* command

is used to write an element back in memory, also supporting popular scalar and vector data. The uint8 datatype form utilized in Python is similar to *unsigned char* for the FPGA. Given the fact that FPGA global memory is about 32GB, by inserting unsigned character of 8-bits as our data precision, the kernel can calculate a complete image in one NDRange run. Tests in this section were conducted for 32-bit float input and output, 8-bit unsigned char input and 32-bit float output, 8-bit unsigned char input and half precision 16-bit float output which is the simplest possible global memory interconnection available for our application. Of course, inside the design, all input data are transformed into 32-bit floats and mathematical operations occur with float precision from start to finish, hence cannot be considered as a data precision scaling exploration.

Results

Figures 7.4 and 7.5 hold the performance, area usage and achieved Fmax for the pre-discussed kernels and concern OpenCL execution time. Area usage is about the same across all kernels, the biggest value of frequency is achieved for *half float output* as well as some gain in OpenCL execution time is noticed there, equal to 15% when compared to float precision output. A reason why this occurs might be that half precision data are getting stored back to memory faster than floats, so kernel "makes room" for the next threads in a faster pace overall. Of course, since half precision is implemented, it is crucial to check whether output data precision falls off. Mean output precision divergence between 32bit 16bit float in output array is 0.01%, which is acceptable, and is due to the normalization part of the code that holds the dataset from getting immensely diverged.

On the sense of implementing the design with our host code, data compression has a huge impact on the created data movement overhead, that is writing the total of input data to global memory and reading back from it. The two following graphs depict how changing the length in data affects the data movement bottleneck. Specifically, changing from float input datatype to unsigned characters and utilizing *convet_T* function inside the FPGA with the cost of utilizing more DSPs, increases the performance by a **42%** factor for the same amount of data written into the device's global memory. Changing the output datatype precision speeds up the design by a **50%** in complete image prediction time whereas a 19% decrease is found for lower sizes. We have to take into account some notes regarding the architecture and what grounds these comparisons are made on. When input data are considered, unsigned characters are the 1/4th of floats in bit length. When an X amount of float data are passed into the kernel, a 4x amount are at the same time passed inside the unsigned char precision kernel; thus the first one has to further stall and wait for data to

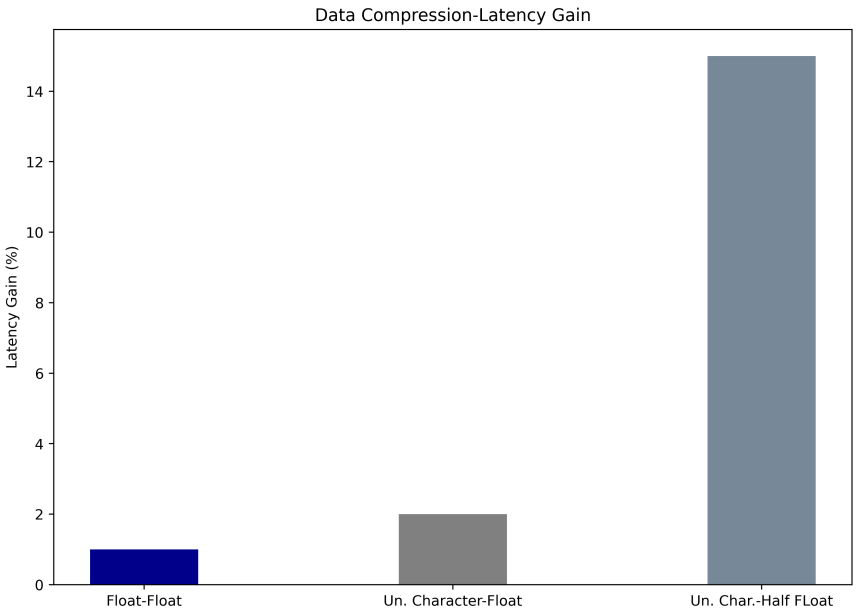


Figure 7.4: Latency Gain

be written so it can continue its execution. The device we experimented with has a total of 34GB available as global memory. A complete image's size is equal to about 12GB when passed as unsigned character values. This means that float precision cannot support a full image prediction in one NDRange cast.

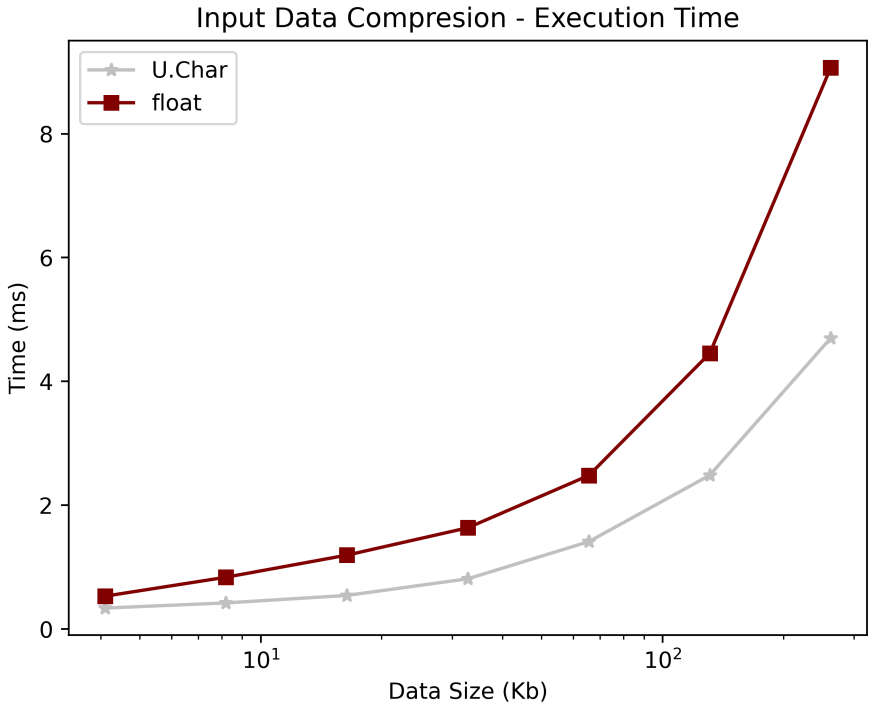


Figure 7.6: Input Data Compression Overhead

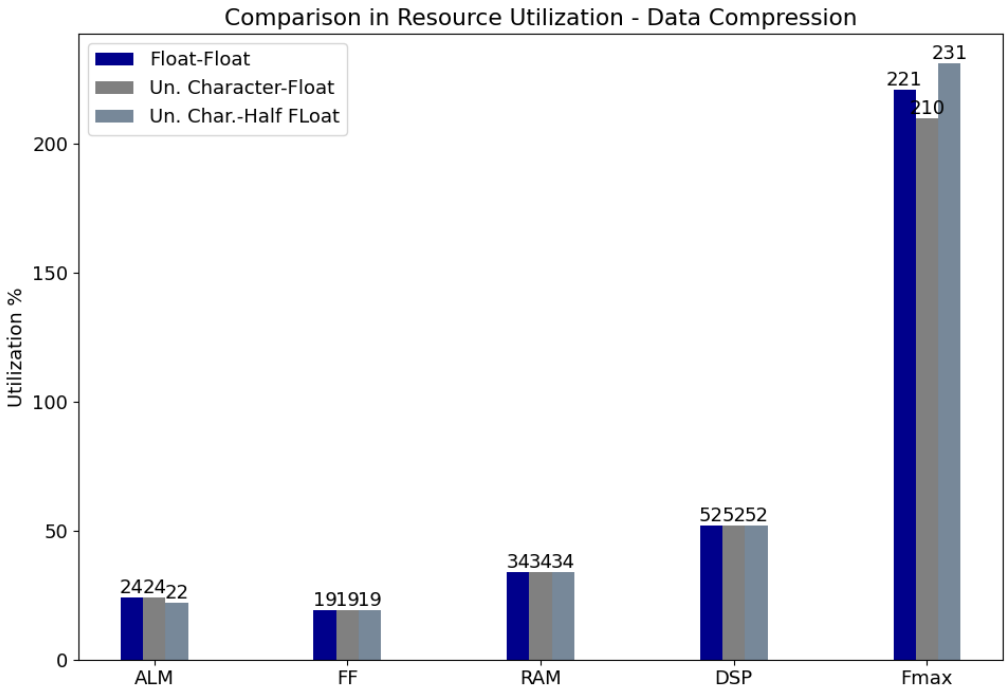


Figure 7.5: Resource Utilization

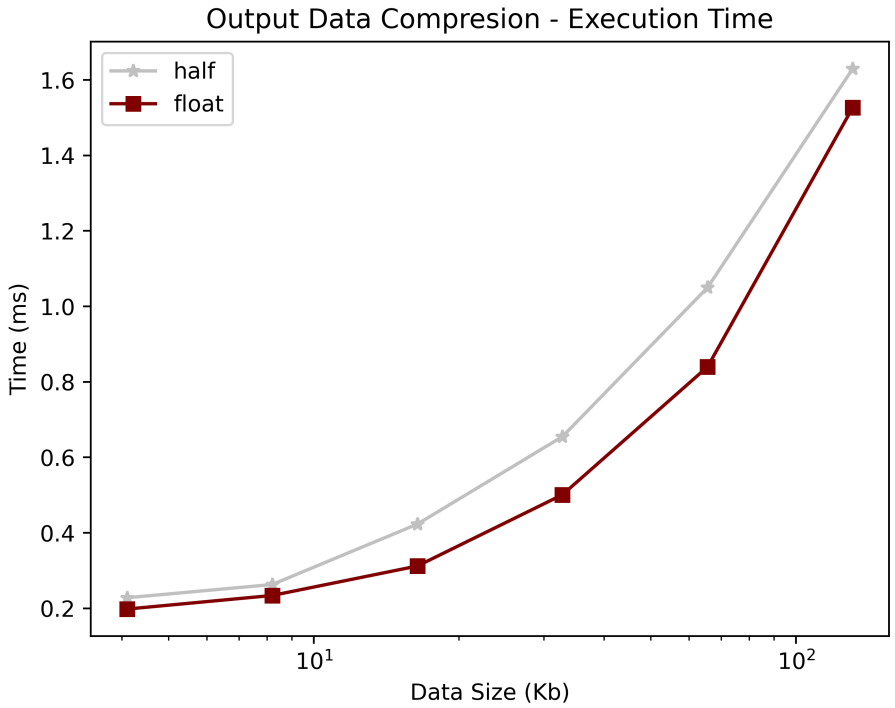


Figure 7.7: Output Data Compression Overhead

Fully Integrated Application - Measurements

To sum up with our whole design space exploration of this diploma thesis, the current subsection presents crucial information and comparisons in performance between all different kinds of realizations that were encountered throughout the entirety of our work. A comparison in the fully integrated design is also presented, as that is seen in 7.3, and concerns a fraction of the total image prediction time.

Firstly, Fig.7.8 shows the performance achieved by some of the main designs that resulted from Chapters 4,5 and 6. For the sake of not overloading the graph with additional information, only designs that showed substantial progress in performance are included. Starting with the first 3 depicted designs, the impact of compressing is visible, as data movement overhead is reduced heavily. A quite similar behaviour can be seen for pipeline implementations as well, however that specific overhead is relatively smaller, since the problem is split into smaller tasks, one of which is to just inform a kernel about the existence of input data, during which time the rest of computing kernels are already running. However, pipeline implementations reach an upper performance limit and latency optimized build progress once more. The best Pipeline-performing architecture is selected to be the one of 4 parallel architectures (see Chapter 6 and is performing slightly worse than that of latency optimized-best case (5).

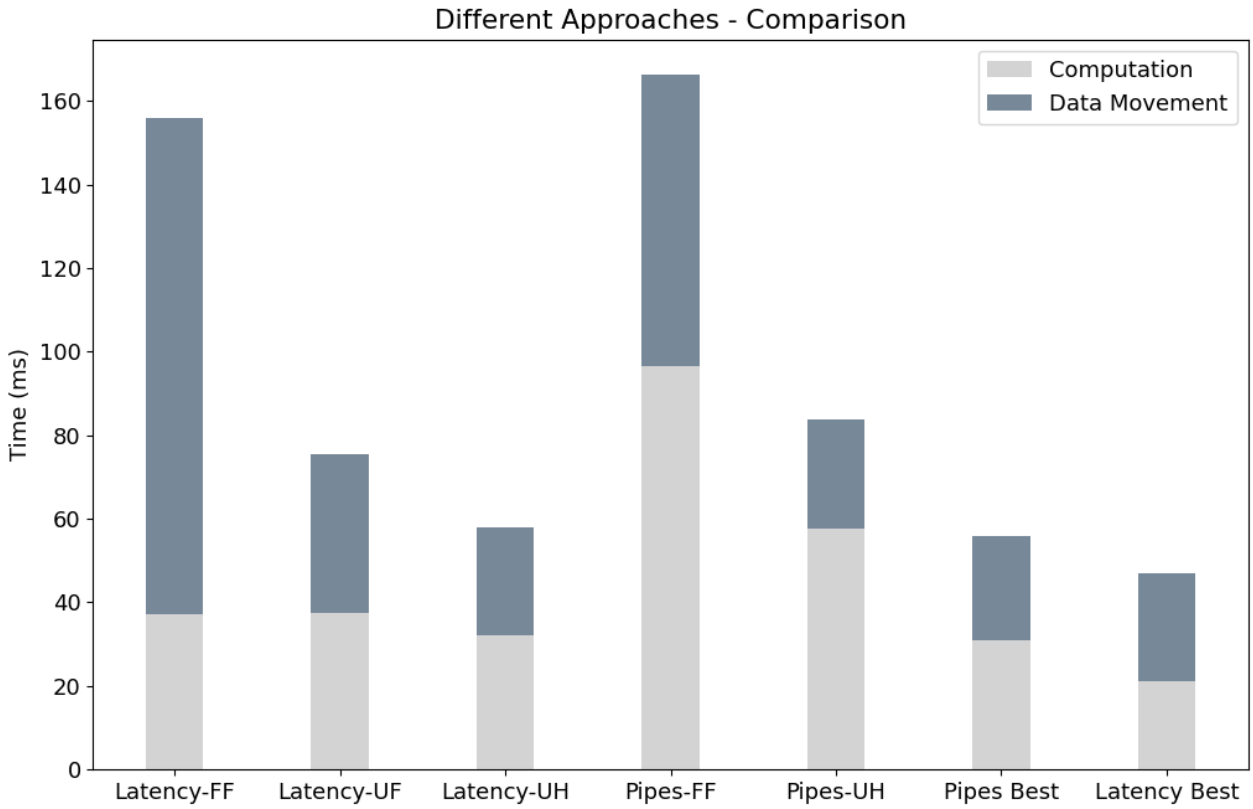


Figure 7.8: Comparison between Main Implementation Strategies

Out of the total of 8 depicted above implementations, we selected the best performing one to compare with Python software counterpart. First of all, Fig.7.9 shows how much slower is our first design, the one found in 4.1 when compared to Software with Python being about **26x** faster than hardware. That is to be expected since *Tensorflow* libraries that play a decisive role in normalizing and predicting the image are constantly improving to perform in the most effective way.

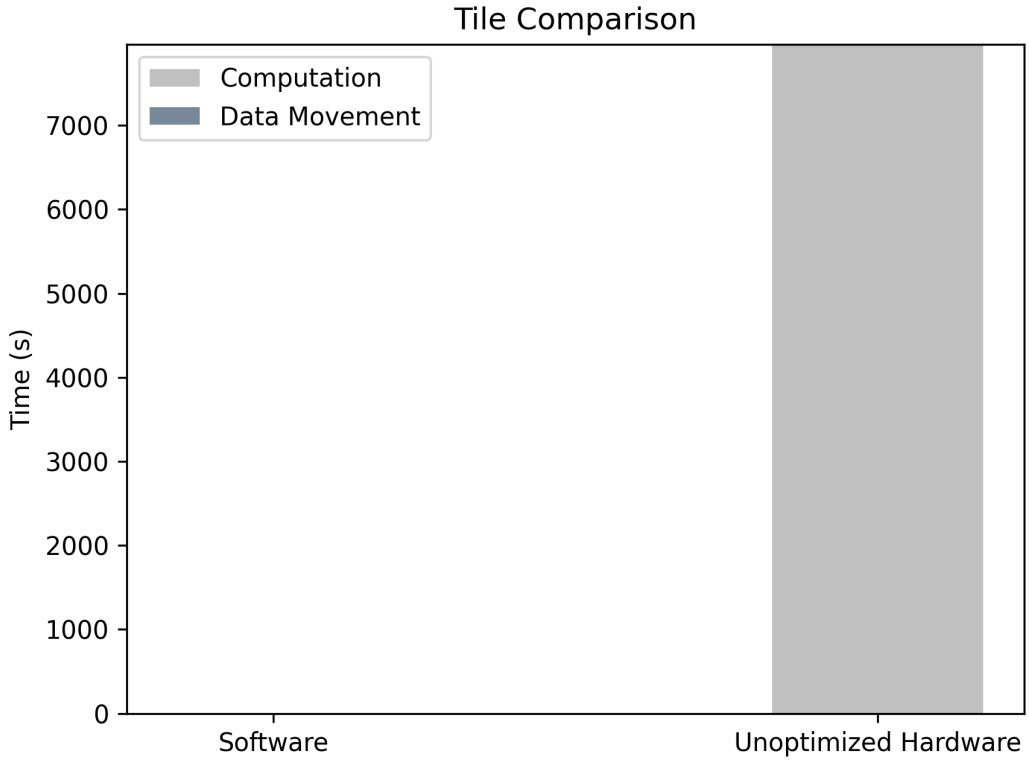


Figure 7.9: Hardware Unoptimized Comparison

However, our exploration and the usage of a rather power-full device, as that is Intel Stratix 10, lead us to a rather desired outcome. Fig.7.10 shows how much faster our best-performing implementation works when compared to Python for the full image prediction process. The FPGA scores an overall of ***7.7 sec*** for complete image normalization and prediction resulting in an overall ***7.14x*** speedup compared to software. On the throughput optimized spectrum, the best build performs about *5.79x* times faster than software, with its computation times dropping even lower than that of latency optimized, however data scheduling places a heavy overhead on them. In general, it would be a great practice to increase the problem size for the FPGA, since that is where *NDRange* thrives. The more the problem increases, that more the benefits result from implementing FPGA kernels.

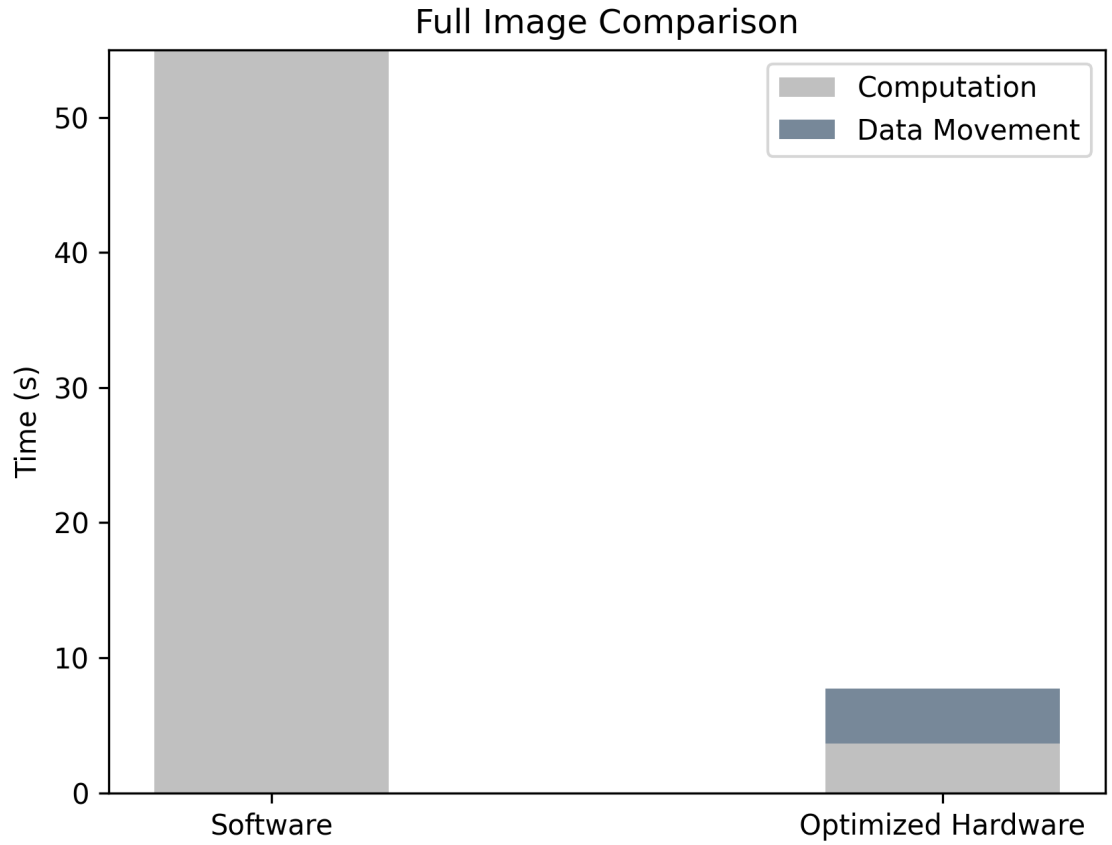


Figure 7.10: Hardware Optimized Comparison

To conclude with the integration part, Fig.7.11 shows the results from realizing the idea given in Fig.7.3. Although the gray-delimited box performs quite better than software, as we saw above, the overhead from *Python-C++ Interconnection* is far bigger and a general delimiter to the application performance. This overhead scales accordingly to the problem's growth. That means that for a 128 tile size (responding to 1404928 threads) the overhead is about 15% bigger that than of 64 tile size, 256's is about 10% bigger and so on. This overhead is due to an amount of issues, some of them being *Data Alignment*. FPGA demands 8-byte data alignment in order to use its *Direct Memory Access* (DMA) feature. However, our limited time did not give us the opportunity to manually construct these kind of datatypes on the software side, hence the overhead results mainly from replicating the data inside the host and writing the output through a loop.

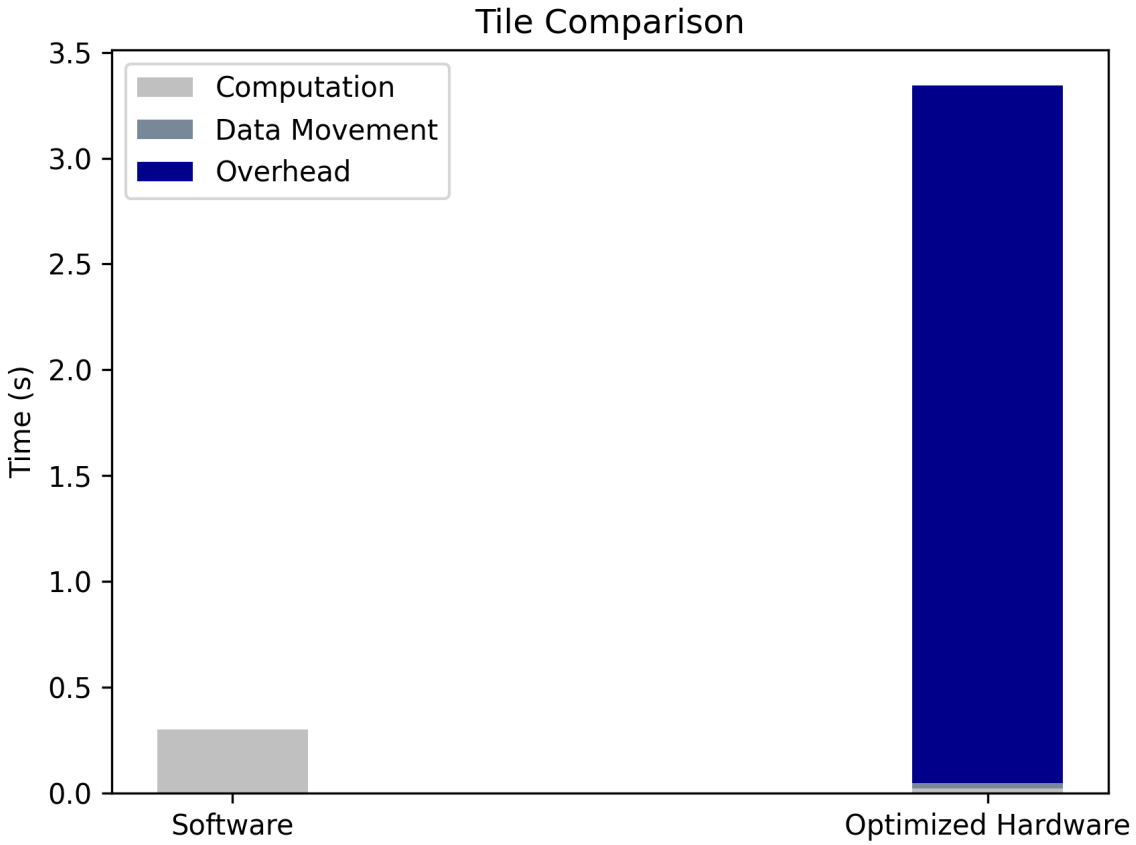


Figure 7.11: Fully Integrated Comparison

Chapter 8

Conclusions

Thesis Summary

Detecting multi-temporal changes over the same field of view in high resolution images poses a big challenge in the area of image processing. The ever growing amount of calculations than need to occur as well as the movement of a rather big amount of data make the application a difficult to implement task. Python's high levels of integration gives an advantage to the designer through a wide range of libraries, such as Tensorflow, to effectively complete the task without dealing with complex arithmetic functions. Nonetheless, the amount of operations remains immense, and even the constantly evolving and sophisticated libraries fail to leave up to the task when general purpose processing is involved. In this thesis, an exploration on whether HPC can help alleviate the extended response times in the processing part of the algorithm is presented.

In our thesis, HLS tools are utilized in order to create efficient hardware that will deal with the Image Encoding problem. Firstly, a new approach to the prediction model is demonstrated in the name of exposing a bigger amount of threads, since OpenCL and FPGAs thrive when dealing with larger problems than that of general purpose computers. The exploration begun by creating an unoptimized hardware implementation and by using HLS Tools as well as architectural techniques, we tried to improve the overall performance with respect to the total resources consumed at each step of the process. The micro-architectural exploration revealed high amounts of performance gains relative to the original hardware, whereas scaling the most -performance and resource-effective design further improved the overall response time. In the end of our design, we created an architecture that can predict a full high resolution image in one run, as opposed to a time-consuming loop, realized in software. Lastly, an integration with software was conducted in the direction of suggesting an overall solution to the problem.

Results of applying micro-architectural optimizations achieved an overall of 100x speedup whereas combining these techniques with scaling the whole de-

sign produced a 330x acceleration when compared to the original, unoptimized hardware implementation. The software design was accelerated by about 7x, however this speedup hit a roadblock when integration with software was issued. Data aligning and generically ctypes place a sizeable overhead to the full implementation.

Future Work

Although the design space that was explored was quite large, there is always room for a more thorough study to be conducted. What is more, integration chapter's results revealed bottlenecks that leave room for further improvement. There are various ideas and techniques that one can take advantage of and fulfill this thesis' suggested propositions.

Reducing Integration Overhead: The main problems with integration are that data are not aligned and ctypes puts a time barrier between run-times. Intel Stratix 10 requires an 8-byte data alignment in order to use its Direct Memory Access protocol that offers high transfer speed from host to device and vice-versa. An exploration on how to effectively pass data from Python software to host application should hence be conducted. Furthermore, ctypes could be replaced by other Python-C++ Interconnection modules, or even the subprocess library could be used, since host application's responsibility is to just set the data for the FPGA and no heavy data processing is conducted during its execution.

Output Accuracy Improvement: Since hardware implementation has proved to effectively and reliably calculate the prediction part of the algorithm, the precision in calculations and hence the output accuracy of the module could be increased. If we manage to increase the accuracy of the total prediction without performance taking a heavy toll, the application overall would act in a more robust way and therefore be used in other areas as well, such as detecting climate-related hazards and natural disasters in all countries.

Comparison with HPC Devices: As a last direction, a comparison between other HPC devices could be issued. For example, comparing Statix 10 FPGA with Xilinx FPGAs could be interesting, since they possess other tools of architecturally improving hardware, although both are utilizing OpenCL. Another comparison could be conducted between FPGAs and GPGPUs, since GPGPUs give the opportunity of approaching the problem in a more fine-grained way.

Bibliography

[1] M. T. Hagan, H. B. Demuth, and M. Beale, *Neural network design*. PWS Publishing Co., 1997.

[2] "Intel® fpga sdk for opencl™ pro edition programming guide," https://www.intel.com/content/www/us/en/docs/programmable/683846/21-4/intel-fpga-sdk-for-opencl-pro-edition.html, pp. 9–14,25–49,69–86,104–133, 2018.

[3] "Intel® fpga sdk for opencl™ pro edition best practices guide," https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/opencl-sdk/archives/aocl-best-practices-guide-18-1.pdf, pp. 7–89,122–163.

[4] M. Aubrun, "Status of the sentinel-2 satellite images use case," https://www.evolve-h2020.eu/news-events/news/status-of-the-sentinel-2-satellite-images-use-case/, 2020.

[5] "Sentinel-2," https://sentinel.esa.int/web/sentinel/missions/sentinel-2.

[6] "The sentinel-2 satellite images use case," https://www.evolve-h2020.eu/image_temp/evolve_newsletter_3_ac_20201116.pdf.

[7] S. S. Haykin *et al.*, "Neural networks and learning machines/simon haykin." p. 48–56, 2009.

[8] J. Tompson and K. Schlachter, "An introduction to the opencl programming model," *Person Education*, vol. 49, p. 31, 2012.

[9] P. Gorlani, T. Kenter, and C. Plessl, "Opencl implementation of cannon's matrix multiplication algorithm on intel stratix 10 fpgas," in *2019 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 2019, pp. 99–107.

[10] P. Machado, E. Kanjo, and A. O. A. Lotfi, "Neurohsmd: Neuromorphic hybrid spiking motion detector," *arXiv preprint arXiv:2112.06102*, 2021.

[11] K. L. de Jong and A. S. Bosman, "Unsupervised change detection in satellite images using convolutional neural networks," in *2019 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2019, pp. 1–8.

[12] R. Domingo, R. Salvador, H. Fabelo, D. Madronal, S. Ortega, R. Lazcano, E. Juárez, G. Callicó, and C. Sanz, "High-level design using intel fpga opencl: A hyperspectral imaging spatial-spectral classifier," in *2017 12th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*. IEEE, 2017, pp. 1–8.

[13] J. Cardoso, J. Coutinho, and P. Diniz, "Source code transformations and optimizations," *Embedded Computing for High Performance: Efficient Mapping of Computations Using Customization, Code Transformations and Compilation, Cardoso J, Coutinho J, Diniz P (eds.), Morgan Kaufmann*, pp. 137–183, 2017.

[14] K. Koliogeorgi, "Optimizing ecg signal analysis by building fpga-based accelerators using high level synthesis," 2016.

[15] D. Wang, K. Xu, and D. Jiang, "Pipecnn: An opencl-based open-source fpga accelerator for convolution neural networks," in *2017 International Conference on Field Programmable Technology (ICFPT)*. IEEE, 2017, pp. 279–282.

[16] B. Gaster, L. Howes, D. Kaeli, P. Mistry, and D. Schaa, "Introduction to parallel programming," *Heterogeneous Computing with OpenCL; Elsevier Inc.: Amsterdam, The Netherlands*, pp. 1–13, 2013.