Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων και Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης και Υπολογιστων
Εργαστηριο Μικροϋπολογιστων και Ψηφιακων Συστηματων

DESIGN AND EVALUATION OF CNNs ON FPGA BY EXPLORING AND COMBINING APPROXIMATION TECHNIQUES

Διπλωματικη Εργασια

του

ΧΑΤΖΗΤΣΟΜΠΑΝΗ ΓΕΩΡΓΙΟΥ

Επιβλέπων: Δημήτριος Σούντρης
Καθηγητής Ε.Μ.Π.

Εργαστηριο Μικροϋπολογιστων και Ψηφιακων Συστηματων
Αθήνα, Ιούλιος 2020

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

DESIGN AND EVALUATION OF CNNs ON FPGA BY EXPLORING AND COMBINING APPROXIMATION TECHNIQUES

Διπλωματικη Εργασια

του

ΧΑΤΖΗΤΣΟΜΠΑΝΗ ΓΕΩΡΓΙΟΥ

Επιβλέπων: Δημήτριος Σούντρης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 17 Ιουλιου 2020.

(Υπογραφή)	*(Υπογραφή)*	*(Υπογραφή)*
.....	.....	.....
Δημήτριος Σούντρης	Παναγιώτης Τσανάκας	Διονύσιος Πνευματικάτος
Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π

Αθήνα, Ιούλιος 2020

(Υπογραφή)

.....

Χατζητσομπανης Γεωργιος

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

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

Ευχαριστίες

Στο σημείο αυτό θα ήθελα να ευχαριστήσω όλους οσους συντέλεσαν στην εκπόνηση της παρούσας διπλωματικής αλλά και της περαιτέρω πορείας μου σε ακαδημαϊκό επίπεδο. Πιο συγκεκριμένα θα ήθελα να ευχαριστήσω τον επιβλέβοντα της διπλωματικής κ. Δημήτριο Σούντρη, ο οποίος μου μετέφερε πληθώρα γνώσεων στον τομέα της ψηφιακής σχεδίασης μέσω των διαλέξεων του και των εργαστηρίων που ήταν υπεύθηνος.

Στην συνέχεια θα ήθελα να εκφράσω την ευγνομωσύνη μου προς τον μεταδιδακτορικό ερευνητή Γεώργιο Λεντάρη και τον υποψήφιο διδάκτορα Βασίλη Λέοντα, οι οποίοι συνέβαλαν σε σημαντικό βαθμό τοσο στην καλλιέργεια του τρόπου σκέψης μου ως προς την προσέγγιση σχετικών ζητημάτων, όσο και στην καλλιέργεια και ανάπτυξη νέων ιδεών και κατευθύνσεων που αφορούσαν την εκπόνηση αυτης της εργασίας. Θέλω να τους ευχαριστήσω ιδιαιτέρως για τον χρόνο που αφιέρωσαν ώστε να ανταλλάξουμε σκέψεις και ιδέες σχετικά με ζητήματα που προέκυψαν κατά την διάρκεια της διπλωματικής.

Τέλος, θα ήθελα να ευχαριστήσω την οικογένεια μου και όλους τους κοντινούς μου ανθρώπους για την στήριξη που μου παρέχουν.

Γιώργος Χατζητσομπάνης

Ιούλιος 2020

Περίληψη

Η παγκόσμια απαίτηση για γρηγορότερες εφαρμογές με χαμηλότερη κατανάλωση ισχύος έχει φέρει στην επιφάνεια αρκετούς επιταγχυντές Συνελικτικών Νευρωνικών Δικτύων (CNNs) σε εξειδικευμένες μονάδες επεργασίας όπως είναι όπως είναι οι κάρτες γραφικών (GPUs), τα FPGAs (Field Programmable Gate Arrays) και οι TPUs (Tensor Processing Units). Η κύρια πρόκληση είναι να αναπτύξουμε γενικούς κινητήρες επτιτάχυνσης οι οποίοι βελτιώνουν την απόδοση οποιουδήποτε δωσμένου συνελικτικού δικτύου. Στην παρούσα διπλωματική η διαδικασία της συνέλιξης η οποία είναι η πιο απαιτητική σε πόρους σε κάθε συνελικτικό δίκτυο σχεδιάστηκε με γλώσσα περιγραφής υλικού (VHDL) κάνοντας μια σε βάθος ανάλυση και εξερεύνηση της αρχιτεκτονικής για να βρούμε τις καλύτερες δυνατές λύσεις. Πιο συγκεκριμένα, σχεδιάσαμε έναν κλασσικό κινητήρα συνέλιξης που εκτελεί την κλασσική μαθηματική διαδικασία της συνέλιξης και έναν κινητήρα συνέλιξης ο οποίος βασίζεται στην προσέγγιση του αλγορίθμου Winograd και συγκρίναμε τις βελτιώσεις σε πόρους και απόδοση μεταξύ τους. Ως επόμενο βήμα, αναπτύξαμε προσεγγιστικούς πολλαπλασιαστές με υβριδική κωδικοποίηση στα μερικά γινόμενα για διάφορα μεγέθη δεδομένων, μια τεχνική Block Floating Point (BFP) και διάφορες τεχνικές απλής αποκοπής των χαμηλότερων σημασίας bit ενός αριθμού. Συνδυασμοί αυτών των προσεγγιστικών τεχνικών εξερευνήθηκαν και στις δύο προσεγγίσεις κινητήρων και εξετάστηκαν τα διάφορα πλεονεκτηματα τους σε σχέση με τον ρυθμό λειτουργίας, την κατανάλωση, την ακρίβεια κατηγοριοποίησης και τα μέγιστα δυνατά πλεονεκτήματα της κάθε αρχιτεκτονικής. Η επαλήθευση πραγματοποιήθηκε με την χρήση τριών διαφορεκτικών συνελικτικών δικτύων : CIFAR-10, MNIST και ένα δίκτυο για την αναγνώριση πλοίων από εικόνες δορυφόρων Ship Detection. Η επαλήθευση της ακρίβειας έγινε με την βοήθεια ενός μοντέλου λογισμικού σε C/C++ το οποίο εκτελούσε τις πράξεις σε επίπεδο bit. Τα τελικά αποτελέσματα μας δείχνουν οτι η ακρίβεια κατιγοριοποίησης μειώνεται κατά μόνο 0.1%-0.4% ανεξαιρέτου του μεγέθους της εικόνας εισόδου και του αριθμού των κλάσεων. Χρησιμοπιόντας την προσέγγιση BFP , καταφέραμε να τετραπλασιάσουμε την απόδοση των συνελικτικών δικτύων με floating point δεδομένα και να την διπλασιάσουμε σε αυτά με fixed-point δεδομένα στο Zynq-7020. Η προτεινόμενη μίξη κινητήρων μειώνει τις απαιτήσεις σε πόρους από 7% εώς 41% και βελτιώνει την απόδοση κάθε δωσμένου δικτύου χωρίς να χρειάζεται επανεκπαίδευση ή αλλαγή στη δομή του σε σύγκριση με άλλες μεθόδους συμπίεσης. Τέλος τα αποτελέσματα από τις τελικές αρχιτεκτονικές συγκρίθηκαν με την κάρτα γραφικών Jetson Nano (204 FPS[1] σε σύγκριση με 720 FPS) και επιτεύχθηκε δέκα φορές καλύτερη απόδοση/Watt.

[1]Πλαίσια εικόνας/δευτερόλεπτο

Λέξεις Κλειδιά

FPGA, Εξοικονόμηση Ενέργειας, Σχεδιασμός σε VDHL, Επιταχυντές συνελικτικών δικτύων, Εξερεύνηση αρχιτεκτονικών, Προσεγγιστικοί υπολογισμοί, Προσεγγιστικοί πολλαπλασιαστές

Abstract

The worldwide demand for faster applications with less power consumption has brought up to the table multiple Convolutional Neural Network (CNN) accelerators on specialized processing units such as FPGAs, GPUs and TPUs. The main challenge is to develop generic engines that improve the performance of any given CNN. In this thesis, the convolution engine, i.e., the most resource-hungry component of the entire CCN, is designed in hardware description language (VHDL) by performing an in-depth design space exploration to find optimal solutions. In particular, we designed a baseline convolution engine that performs the typical convolution operation and a convolution engine based on the Winograd algorithm, and compared the resource & performance gains among different design configurations of them. As a next step, we developed hybrid high-radix encoding multipliers for various bit-lengths, a block floating-point technique and various simple truncation methods. Combinations of these approximation techniques were adopted in both convolution engines, and trade-offs in terms of throughput-resources-accuracy as well as the maximum possible gains were examined. The evaluation was performed with three different CNNs : CIFAR-10, MNIST and a custom network for ship classification. The accuracy evaluation tests were performed with the assistant of bit-accurate software models in C/C++. The final results show that the classification accuracy degrades by only 0.1-0.4% regardless of image size and the number of categories. Using the block floating-point approach, we managed to quadruple the throughput of floating-point CNNs and double the throughput of fixed-point CNNs on Xilinx Zynq-7020. The proposed mixture of engines decreases the logic resources by 7% to 41% and improves the throughput of any given CNN, while maintaining its original training (the network does not require to be retrained as it happens in other compression methods). Finally, the results of our implementation on Xilinx Zynq-7020 were compared to Jetson Nano GPU (204 FPS via TensorFlow+CUDA acceleration compared to 720 FPS) and achieved 10× better performance/Watt.

Keywords

Prototype Design, FPGA Design, Energy Efficiency, RTL Design, VHDL Design, Approximations, CNN, Block Floating Point, Hybrid Approximate Multipliers, Data type exploration, FPGA accelerator

Contents

List of Figures

List of Tables

Εκτεταμένη Περίληψη

Εισαγωγή

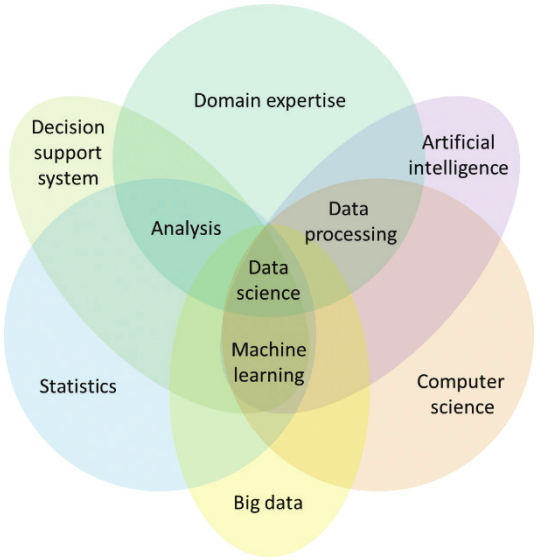
Τα Βαθιά Συνελικτικά Δίκτυα[1] έχουν δείξει πολλές σημαντικές βελτιώσεις σε πολλές εφαρμογές Τεχνητής Νοημοσύνης[2] , όπως είναι η όραση υπολογιστών [12], η επεξεργασία φυσικής γλώσσας [29], η αναγνώριση φωνής [1] και η μηχανική μετάφραση [2]. Η απόδοση τους βελτιώνεται με γρήγορους ρυθμούς.

Μηχανική Μάθηση

Η Μηχανική Μάθηση συναντιέται σε αρκετές πτυχές της καθημερινότητας των ανθρώπων. Η κατηγοριοποίηση αλληλογραφίας, το σύστημα προτάσεων διαφημίσεων ή προτάσεων σχετικών με τα ενδιαφέροντα του χρήστη, στην αναγνώριση κειμένου και φωνής ακόμα και σε αυτόνομα αυτοκίνητα ή αυτοματοποιήσεις διάφορων εργασιών που εκτελούνται από τον άνθρωπο. Η τεχνητή νοημοσύνη θέτει τα ερώτηματα ¨Τι είναι η ευφυΐα και πώς αυτή δουλέυει·' και ¨Μπορούμε να φτιάξουμε έξυπνες μηχανές·¨. Όπως υποδεικνύει και το όνομα, με την μηχανική μάθηση προσπαθούμε να εκπαιδεύσουμε τους υπολογιστές ώστε να μάθουν να λύνουν προβλήματα χωρίς να είναι εκτενώς προγραμματισμένοι για αυτόν τον σκοπό. Ένας πιο επίσημος ορισμός για την μάθηση είναι ο εξής : '

Ένα πρόγραμμα υπολογιστή, λέγεται οτι μαθαίνει από την εμπειρία E με δεδομένη κάποια εργασία T και κάποια μέτρηση της απόδοσης Π, αν η απόδοση του στην T όπως μετριέται απο την Π βελτιώνεται με την εμπειρία E¨. Στον πυρήνα της υπάρχει η υπόθεση οτι η γνώση μπορεί να αναχθεί από τα δεδομένα. Η Μηχανική Μάθηση προχωράει ένα βήμα μπροστά από τις ¨έξυπνες' μηχανές και υπόσχεται μια μεγαλύτερου εύρους και βάθους αυτοματοποίηση στις ανθρώπινες δραστηριότητες. Οι μονότονες και επαναληπτικές εργασίες σε έναν σύγχρονο κόσμο θα εκτελούνται από μηχανές.

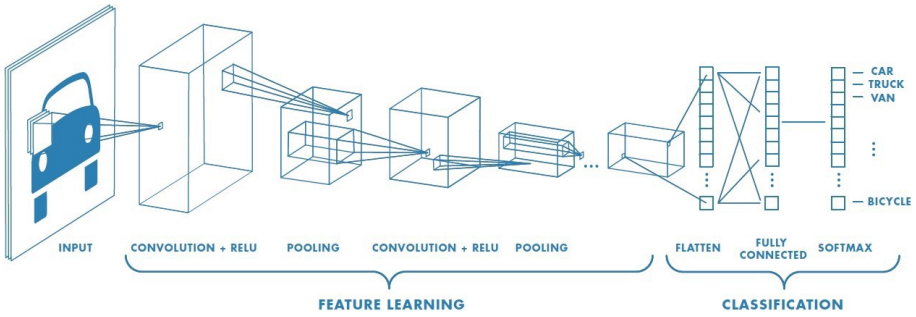
[1]Deep Neural Networks (DNNs)
[2]Artificial Intelligence (AI)



Σχήμα 1: Συσχέτιση της Μηχανικής Μάθησης σε σχέση με άλλα επιστημονικά πεδία

Συνελικτικά Νευρωνικά Δίκτυα

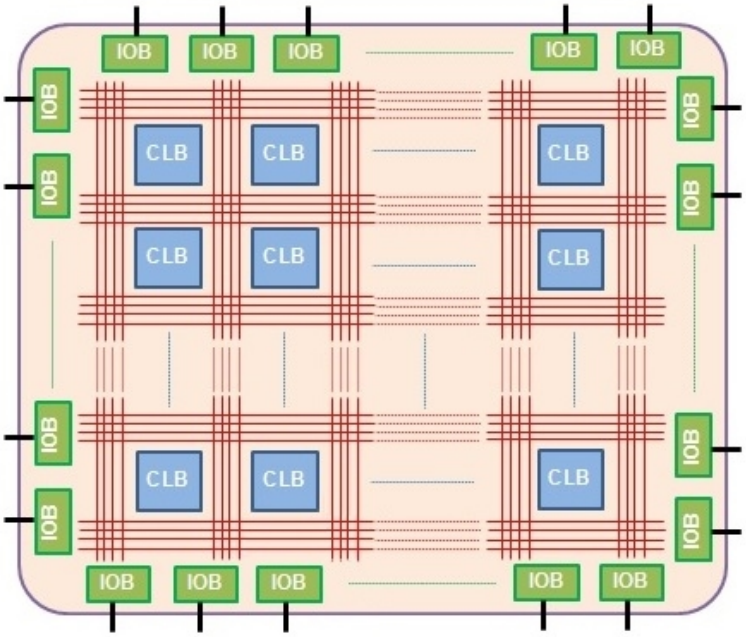
Τα Συνελικτικά Νευρωνικά Δίκτυα είναι μια ιδιαίτερη κατηγορία νευρωνικών δικτύων για επεξεργασία δεδομένων τα οποία έχουν μια γνωστή τοπολογία πλέγματος. Είναι φτιαγμένα από νευρώνες οι οποίοι έχουν εκπαιδεύσιμα βάρη και πολώσεις. Κάθε νευρώνας δέχεται κάποια είσοδο, εκτελεί ένα εσωτερικό γινόμενο το οποίο προαιρετικά ακολουθείται από μια μη γραμμικότητα. Αυτά τα δίκτυα έχουν εμπνευστεί από το μοντέλο το οποίο τα θηλαστικά λαμβάνουν την πληροφορία του κόσμου γύρω τους χρησιμοποιόντας μια κατάλληλη συστοιχία βιολογικών νευρώνων στον εγκέφαλο τους για να αναγνωρίσουν το αντικείμενο αυτό. Ως παράδειγμα μπορούμε να θεωρήσουμε ένα αυτοκίνητο και να εξετάσουμε τον τρόπο που ένας άνθρωπος το αγαγνωρίζει. Ο άνθρωπος ψάχνει για χαρακτηριστικά τα οποία ξεχωρίζουν ένα αυτοκίνητο από άλλα μέσα μεταφοράς όπως είναι οι τροχοί, τα μπροστινά φώτα, οι πόρτες, το ντεπόζιτο, το καπό, οι καθρέφτες και το παρμπρίζ και άλλα. Παρόμοια όταν αναγνωρίζει έναν τροχό ψάχνει για αντικείμενα κυκλικού σχήματος, σκούρου χρώματος τα οποία βρίσκονται κάτω από την κύρια δομή του αυτοκινήτου. Όλες αυτές οι μικρές πληροφορίες συνδυάζονται μαζί για να σχηματίσουν ένα συγκεκριμένο χαρακτηριστικό το οποίο είναι μοναδικό σε ένα αντικείμενο το οποίο αναγνωρίζουμε. Κάθε επίπεδο ενός συνελικτικού νευρωνικού δικτύου σχετίζεται με την παραγωγή πληροφορίας από τιμές οι οποίες έρχονται από προηγούμενα επίπεδα σε μια ακόμα πιο σύνθετη πληροφορία και την περαιτέρω διάδοση της σε επόμενα επίπεδα για να γινει περαιτέρω γενικοποίηση.



Σχήμα 2: Παράδειγμα δομής συνελικτικού δικτύου για την αναγνώριση μέσων μεταφοράς

Field Programmable Gate Arrays

Το FPGA είναι ένας πίνακας από διασυνδεδεμένα ψηφιακά υποκυκλώματα τα οποία υλοποιούν κοινή λειτουργία ενώ παράλληλα προσφέρουν πολύ υψηλά επίπεδα προσαρμοστικότητας. Το FPGA είναι ενας πίνακας από λογικές πύλες, και αυτός ο πίνακας μπορεί να προγραμματιστεί (πραγματικά να διαμορφωθεί) στο επίπεδο της συσκευής από τον χρήστη χωρίς εξάρτηση από την αρχική του κατασκευή. Αξίζει να σημειωθεί ότι δεν είναι μια απλή ¨συλλογή' από ανεξάρτητες πύλες Boolean. Αυτό θα ήταν μη αποδοτικό καθώς δεν θα εκμεταλλευόταν το πλεονέκτημα ότι οι κοινές λειτουργίες μπορούν να υλοποιηθούν πολύ πιο αποδοτικά.



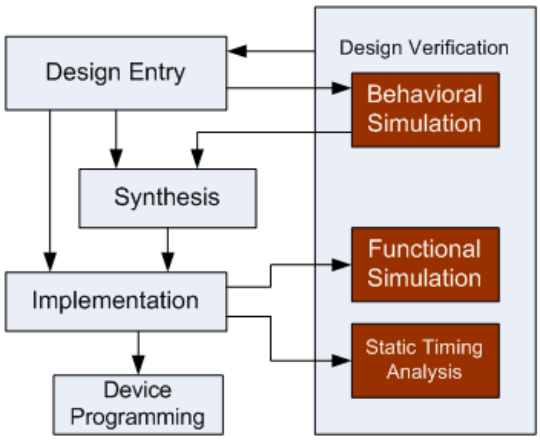
Σχήμα 3: Η πλακέτα ενός FPGA με τα βασικά στοιχεία της.

Τα CLBs πρέπει να αλληλεπιδρούν μεταξύ τους και με επιπλέον κυκλώματα. Για αυτόν τον σκοπό, το FPGA χρησιμοποιεί έναν πίνακα από προγραμματιζόμενες συνδέσεις και κομμάτια εισόδων/εξόδων. Το πρόγραμμα αποθηκεύεται σε κύτταρα SRAM τα οποία ελέγχουν την λειτουργικότητα των CLBs και τους διακόπτες που καθορίζουν τα μονοπάτια συνδέσεων. Η

γενική ιδέα είναι ότι τα CLBs περιέχουν πίνακες αναζήτησης (Look-up tables (LUTs)), στοιχεία αποθήκευσης (καταχωρητές ή flip-flops) και πολυπλέκτες οι οποίοι τους επιτρέπουν να πραγματοποιούν λογικές πράξεις Boolean, αποθήκευση δεδομένων και αριθμητικές πράξεις. Μια μονάδα εισόδου/εξόδου αποτελείται από διάφορα στοιχεία τα οποία αποτελούν τον δίαυλο επικοινωνίας μεταξύ των CLBs και άλλων στοιχείων της πλακέτας. Τέτοια στοιχεία είναι οι pull-up/pull-down αντιστάσεις, οι αντιστροφείς και οι ενδιάμεσες μνήμες.

Προγραμματισμός FPGA

Προκειμένου να προγραμματίσουμε ένα FPGA για μια συγκεκριμένη εργασία, πρέπει να μετατρέψουμε έναν πίνακα από CLBs σε ένα ψηφιακό κύκλωμα το οποίο είναι σχεδιασμένο ακριβώς για την εργασία που θέλουμε. Αυτό φαίνεται ως μια πολύπλοκη και χρονοβόρα διαδικασία. Παρόλα αυτά, η ανάπτυξη εφαρμογών σε FPGA δεν απαιτεί την πλήρη γνώση της εσωτερικής λειτουργίας των CLBs ή τον σχεδιασμό των εσωτερικών συνδέσεων επικοινωνίας, όπως ακριβώς σε έναν μικροελεγκτή δεν χρειάζεται η γνώση της χαμηλού επιπέδου γλώσσας (Assembly) του επεξεργαστή ή των εσωτερικών σημάτων ελέγχου. Στην πραγματικότητα, θα ήταν παραπλανητικό το να παρουσιάσουμε το FPGA ως ένα εξάρτημα το οποίο λειτουργεί αποκλειστικά μόνο του. Πάντα υποστηρίζονται από κώδικα ο οποίος αναλαμβάνει την περίπλοκη διαδικασία της μετατροπής ενός κυκλωματικού σχεδίου σε υλικό στα προγραμματιζόμενα bits που καθορίζουν την συμπεριφορά των διασυνδέσεων και των CLBs. Για αυτόν τον σκοπό έχουν δημιουργηθεί Γλώσσες Περιγραφής Υλικού (Hardware Description Languages-HDL) οι οποίες μας επιτρέπουν την ¨περιγραφή' του υλικού.



Σχήμα 4: Προγραμματισμός FPGA

Πράγματι, υπάρχει μια ομοιότητα μεταξύ του κώδικα περιγραφής υλικού και του κώδικα για λογισμικό σε υψηλό επίπεδο, αλλά είναι δύο θεμελιωδώς διαφορετικές έννοιες. Ο κώδικας σε λογισμικό αναπαριστά μια σειρά από λειτουργίες, ενώ ο κώδικας περιγραφής υλικού μπορεί να περιγραφεί καλύτερα ως ένα σχηματικό το οποίο χρησιμοποιεί κείμενο για να εισαγάγει στοιχεία και να φτιάξει διασυνδέσεις.

Προσεγγιστικοί Υπολογισμοί

Η φορητή και ενσωματωμένη φύση των υπολογιστικών συστημάτων της εποχής μας, έχει οδηγήσει σε μια αυξημένη ανάγκη για υπερβολικά χαμηλή κατανάλωση ισχύος, μικρή επιφάνεια και υψηλή απόδοση. Ο προσεγγιστικός υπολογισμός είναι ένα αναδυόμενο υπολογιστικό πρότυπο που μας επιτρέπει να επιτυγχάνουμε αυτά τα θετικά κάνοντας συμβιβασμούς στην αριθμητική ακρίβεια. Πολλά συστήματα σε τομείς όπως τα πολυμέσα, τα νευρωνικά δίκτυα και η ανάλυση big data παρουσιάζουν μια έμφυτη ανοχή σε ένα συγκεκριμένο επίπεδο ανακρίβειας στους υπολογισμούς και μπορούν να ωφεληθούν από τους προσεγγιστικούς υπολογισμούς. Οι υπολογιστικές και αποθηκευτικές απαιτήσεις των μοντέρνων συστημάτων έχουν ξεπεράσει κατά πολύ τους διαθέσιμους πόρους. Προβλέπεται ότι στην επερχόμενη δεκαετία, ο όγκος της πληροφορίας την οποία διαχειρίζονται τα παγκόσμια κέντρα δεδομένων θα αυξηθεί κατά πενήντα φορές, καθώς ο αριθμός των διαθέσιμων επεξεργαστών θα αυξηθεί μόνο κατά δέκα φορές. Στην πραγματικότητα, η κατανάλωση ηλεκτρικής ενέργειας μόνο των κέντρων δεδομένων της Αμερικής αναμένεται να αυξηθεί από 61 δισεκατομμύρια κιλοβατώρες το 2006 και 91 δισεκατομμύρια κιλοβατώρες το 2013 σε 140 δισεκατομμύρα κιλοβατώρες το 2020. Είναι προφανές ότι η απαίτηση για αυξανόμενη απόδοση σύντομα θα ξεπεράσει την ανάπτυξη στους διαθέσιμους πόρους. Οπότε η αποκλειστική παροχή πόρων δεν θα λύσει τον γρίφο της βιομηχανίας στο κοντινό μέλλον. Η προσεγγιστική λειτουργία στο υλικό, κυρίως ασχολείται με την σχεδίαση προσεγγιστικών αριθμητικών μονάδων, όπως είναι οι αθροιστές και οι πολλαπλασιαστές, σε διαφορετικά επίπεδα αφαίρεσης όπως είναι τα τρανζίστορ, το κυκλωματικό, το επίπεδο πυλών και της εφαρμογής. Μερικοί αξιοσημείωτοι προσεγγιστικοί αθροιστές περιέχουν υποθετικούς αθροιστές[42], κατακερματισμένους αθροιστές[43] και προσεγγιστικούς πλήρεις αθροιστές. Επίσης, στον τομέα των προσεγγιστικών πολλαπλασιαστών, οι οποίοι είναι το πιο απαιτητικό σε θέμα πόρων στοιχείο του υλικού, έχει γίνει σημαντική έρευνα [23, 7, 33, 39, 20].

Οι Προσεγγιστικοί Υπολογισμοί και η αποθήκευση πλεονεκτούν στην παρουσία ανεκτικών στα σφάλματα περιοχών κώδικα σε εφαρμογές και προφανείς περιορισμούς των χρηστών να διαχειριστούν έξυπνα την υλοποίηση, την αποθήκευση και την ακρίβεια του αποτελέσματος για πλεονεκτήματα στην απόδοση ή την ενέργεια. Στην πραγματικότητα ο προσεγγιστικός υπολογισμός εκμεταλλεύεται το κενό μεταξύ του επιπέδου ακρίβειας που απαιτείται από την εφαρμογή ή τον χρήστη και αυτού που παρέχεται από το υπολογιστικό σύστημα για να κάνει τις κατάλληλες βελτιστοποιήσεις.

Προσεγγιστικά Αριθμητικά Κυκλώματα

Προσεγγιστικοί Αθροιστές

Σε προσεγγιστικές υλοποίησεις, οι αθροιστές πολλών bit χωρίζονται σε δύο διαφορετικά μέρη : το ακριβές πάνω μέρος των πιο σημαντικών bit και το προσεγγιστικό κάτω μέρος των λιγότερο σημαντικών bit. Για κάθε χαμηλό bit , ένας προσεγγιστικός αθροιστής του ενός bit πραγματοποιεί μια τροποποιημένη, επομένως ανακριβή διαδικασία της πρόσθεσης. Αυτό συνήθως επιτυγχάνεται με την απλοποίηση ενός πλήρους αθροιστή σε κυκλωματικό επίπεδο,

αντίστοιχα με μια διαδικασία η οποία αλλάζει κάποιες εισόδους στον πίνακα αληθείας ενός πλήρους αθροιστή σε λειτουργικό επίπεδο.

Προσεγγιστικοί Πολλαπλασιαστές

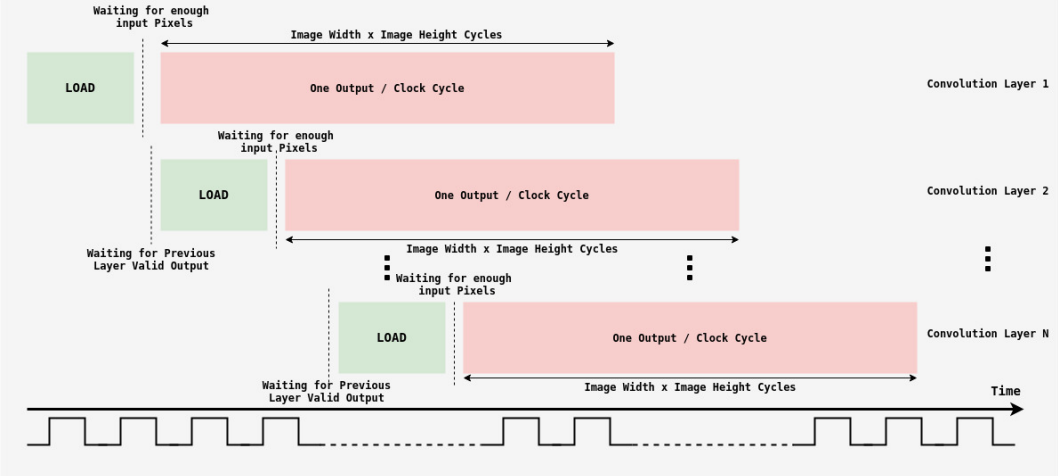
Σε αντίθεση με τον σχεδιασμό προσεγγιστικών αθροιστών, η σχεδίαση προσεγγιστικών πολλαπλασιαστών δεν έχει ερευνηθεί στο έπακρο. Προσεγγιστικοί πολλαπλασιαστές οι οποίοι χρησιμοποιούν τους υποθετικούς αθροιστές για να υπολογίσουν το άθροισμα των μερικών γινομένων έχουν σχεδιαστεί στα [36],[13]. Παρόλα αυτά, η άμεση εφαρμογή των προσεγγιστικών αθροιστών σε έναν πολλαπλασιαστή ίσως είναι μη αποδοτική όσον αφορά την ανταλλαγή ακρίβειας για εξοικονόμηση σε ενέργεια και επιφάνεια. Ένα σημαντικό στοιχείο στην σχεδίαση ενός προσεγγιστικού πολλαπλασιαστή, είναι η μείωση του κρίσιμου μονοπατιού στην άθροιση των μερικών γινομένων. Ο πολλαπλασιασμός συνήθως υλοποιείται από έναν συνδεδεμένο πίνακα αθροιστών. Στο [31] και [16] μερικά λιγότερα σημαντικά bits στα μερικά γινόμενα παραλείπονται (χρησιμοποιώντας μηχανισμούς αντιστάθμισης του σφάλματος) οπότε κάποιοι αθροιστές μπορούν να αφαιρεθούν από τον πίνακα οδηγώντας σε γρηγορότερη λειτουργία. Στο [18], ένας μεγάλος πολλαπλασιαστής κατασκευάζεται από 2×2 απλοποιημένους πολλαπλασιαστές για να μειώσει την αριθμητική και υπολογιστική πολυπλοκότητα. Ένας αποδοτικός συνδυασμός που χρησιμοποιεί προ-επεξεργασία και επιπρόσθετη αντιστάθμιση σφάλματος προτείνεται στο [27] για να μειώσει την καθυστέρηση του κρίσιμου μονοπατιού. Συνδυασμοί της παραγωγής μερικών γινομένων και προσεγγίσεων εφαρμόζονται συνδυαστικά για περαιτέρω μείωση της κατανάλωσης ισχύος [14], [41], [28]. Ο κύριος στόχος στην σημερινή έρευνα όσον αφορά τους προσεγγιστικούς πολλαπλασιαστές είναι να μειωθεί ο αριθμός των μερικών γινομένων χρησιμοποιώντας υβριδικές κωδικοποιήσεις [22] για να εφαρμόσουμε προσεγγίσεις στην παραγωγή μερικών γινομένων.

Σχεδιασμός σε VHDL

Σε αυτό το κεφάλαιο παρουσιάζεται η αρχιτεκτονική η οποία υλοποιεί το συνελικτικό επίπεδο ενός δοσμένου Συνελικτικού Νευρωνικού Δικτύου. Τα βήματα και οι διαφορετικές προσεγγίσεις οι οποίες οδήγησαν στο τελικό σχέδιο αναλύονται σε βάθος.

Σχεδιασμός Κλασσικής Συνέλιξης

Τα Συνελικτικά Νευρωνικά Δίκτυα είναι εκ φύσεως μαζικά παραλληλοποιήσιμοι αλγόριθμοι. Για παράδειγμα αν εξετάσουμε το πρώτο συνελικτικό επίπεδο ενός δοσμένου δικτύου, μπορούμε να δούμε ότι αποτελείται από τρία κανάλια εισόδου, ένα για κάθε χρώμα και 32 φίλτρα. Επομένως περιμένουμε 32 κανάλια ως έξοδο του πρώτου επιπέδου. Ο υπολογισμός του κάθε καναλιού εξόδου μπορεί να θεωρηθεί ως εντελώς ανεξάρτητη διαδικασία από τα υπόλοιπα. Ένα σχέδιο το οποίο υλοποιεί το συνελικτικό επίπεδο οφείλει να εκμεταλλευτεί την δυνατότητα υπολογισμού κάθε καναλιού παράλληλα. Επιπρόσθετα, υπάρχει η δυνατότητα διοχέτευσης μέρων του σχεδίου, δηλαδή να υπολογίζονται με τους διαθέσιμους πόρους τα επόμενα επίπεδα χωρίς την πλήρη ολοκλήρωση των προηγούμενων. Από την σκοπιά χαμηλού επιπέδου αυτής της διπλωματικής, το σχέδιο μπορεί να βελτιστοποιηθεί έτσι ώστε να επιτευχθεί μέγιστη παραλληλία, διοχέτευση υπολογισμών μεταξύ διαφορετικών επιπέδων του δικτύου και βελτιστοποιήσεις στη λογική ώστε να μειωθούν τα κρίσιμα μονοπάτια και να επιτευχθεί μεγαλύτερη συχνότητα ρολογιού.



Σχήμα 5: Διοχέτευση δεδομένων μεταξύ διαφορετικών επιπέδων του δικτύου. Εφόσον τα δεδομένα έχουν φορτωθεί και τα πρώτα έγκυρα δεδομένα έχουν παραχθεί, μπορούν να προωθηθούν σε επόμενα επίπεδα.

Ο κύριος σκοπός είναι η πλήρης αξιοποίηση μιας συσκευής (FPGA), προκειμένου να πραγματοποιηθούν οι υπολογισμοί. Το χαμηλού επιπέδου σχέδιο στην VHDL είναι πλήρως προσαρμόσιμο και μπορεί να υποστηρίξει οποιοδήποτε συνελικτικό νευρωνικό δίκτυο με τις κατάλληλες τροποποιήσεις. Ο τελικός κινητήρας που κατασκευάστηκε μπορεί να χρησιμοποιηθεί

ως επιταχυντής για οποιονδήποτε τύπο δικτύου με σεβασμό στους διαθέσιμους πόρους της εκάστοτε συσκευής.

Απαραίτητα Στοιχεία

Ο σχεδιασμός της συνελικτικής μονάδας απαιτεί στοιχεία τα οποία υλοποιούν τις παρακάτω λειτουργίες:

- Αποθήκευση των εισερχόμενων στοιχείων εισόδου (Pixel) τα οποία αποκτώνται σειριακά με κατάλληλο τρόπο σε ουρές FIFO[1] (RAM[2]), προκειμένου να υλοποιείται μετατροπή από σειριακά σε παράλληλα (Μετατροπέας από σειριακό σε παράλληλο-Serial To Parallel Converter).

- Έλεγχος της ροής δεδομένων και ενεργοποίηση των κατάλληλων σημάτων ελέγχου ώστε να εφαρμοστεί padding[3] όπου αυτό ειναι απαραίτητο (στα όρια της εικόνας) και να συνεχιστούν οι υπολογισμοί (Μονάδα Ελέγχου).

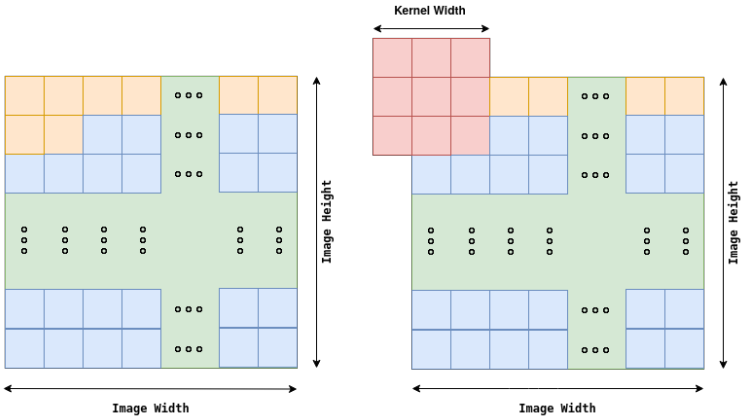
- Αποστολή των παράλληλων δεδομένων στη Συνελικτική Μονάδα και διεξαγωγή των κατάλληλων πολλαπλασιασμών και προσθέσεων μεταξύ της εικόνας εισόδου και του φίλτρου.

- Ενεργοποίηση κατάλληλων σημάτων για τα έγκυρα δεδομένα εξόδου προκειμένου να ενημερωθούν τα επόμενα επίπεδα του δικτύου.

Ροή Δεδομένων

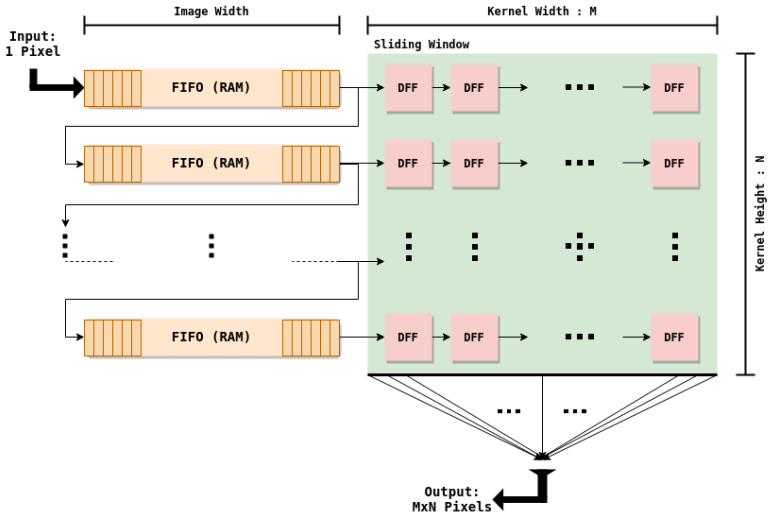
Θεωρούμε εικόνα εισόδου με μέγεθος $Image_{Width} \times Image_{Height}$ και ένα φίλτρο με μέγεθος $Kernel_{Width} \times Kernel_{Height}$. Τα δεδομένα ρέουν στη μονάδα στοιχείο προς στοιχείο για κάθε γραμμή. Σε κάθε κύκλο ρολογιού παίρνουμε ένα νέο στοιχείο ως είσοδο και αποθηκεύουμε το προηγούμενο στη μνήμη. Για να επιτευχθεί η μέγιστη απόδοση, σχεδιάστηκε το ακόλουθο σύστημα. Χρειάζεται να φορτωθούν : $\lfloor Kernel_{Height}/2 \rfloor \times (Image_{Width}) + \lceil Kernel_{Width}/2 \rceil$[4] στοιχεία στη μνήμη προκειμένου να ξεκινήσουν οι υπολογισμοί.

[1] First In First Out
[2] Random-Access Memory
[3] Ο τρόπος με τον οποίο διαχειρίζονται τα άκρα της εικόνας
[4] $\lfloor\rfloor : Floor, \lceil\rceil : Ceil$



Σχήμα 6: Τα πορτοκαλί στοιχεία πρέπει να φορτωθούν στη μνήμη προκειμένου να ξεκινήσουν οι υπολογισμοί για 3×3 φίλτρο.

Για την κατάλληλη αποθήκευση των δεδομένων εισόδου, το ακόλουθο σύστημα μνήμης σχεδιάστηκε. Παράξαμε $Kernel_{Height}$ στον αριθμό ουρές FIFO με μέγεθος $Image_{Width} \times Sizeof(Pixel)$ η κάθε μια. Αυτές οι FIFO είναι συνδεδεμένες μεταξύ τους σε αλυσίδα όπως φαίνεται στο επόμενο σχήμα.

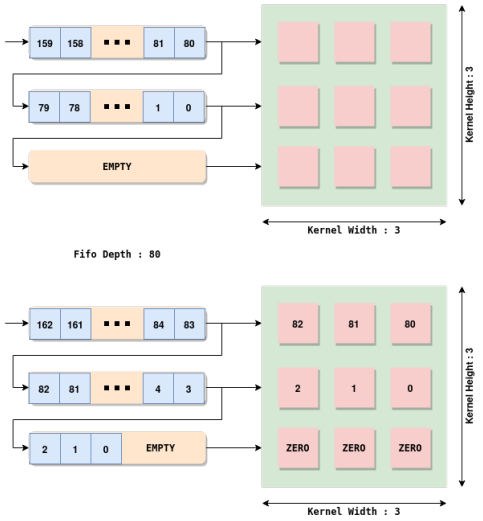


Σχήμα 7: Γενικός μετατροπέας από σειριακό σε παράλληλο. Τα δεδομένα εισόδου δίνονται ως είδοσος στη πρώτη FIFO της αλυσίδας. Το παράθυρο καταχωρητών έχει μέγεθος $N \times M$ που εξαρτάται από το μέγεθος του φίλτρου εισόδου.

Εφόσον οι μνήμες έχουν συνδεθεί με αυτόν τον τρόπο, πρέπει να λάβουμε υπόψιν την εσωτερική τους επικοινωνία μέσω των σημάτων ελέγχου τους. Κάθε μνήμη πρέπει να είναι σε διαρκή επικοινωνία με την προηγούμενη και την επόμενη για να γίνει σωστή αποθήκευση των δεδομένων.

Επίσης πρέπει να ληφθεί υπόψιν το κατάλληλο padding ανάλογα με την εφαρμογή. Τα μέρη της εικόνας που μας απασχολούν είναι : το άνω άκρο, το κάτω άκρο και τα πλαϊνά μέρη

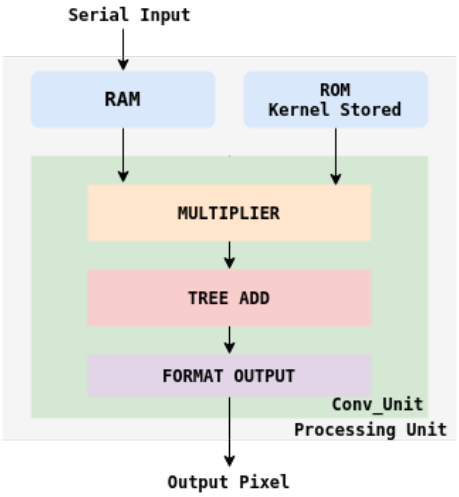
της εικόνας (αριστερά και δεξιά). Με χρήση κατάλληλων σημάτων ελέγχου όπως αναφέρθηκε προηγουμένως η μονάδα λαμβάνει υπόψιν αυτά τα όρια και παράγει τα κατάλληλα στοιχεία ώστε να εκτελε ίται ορθά η διαδικασία της συνέλιξης όπως φαίνεται στο επόμενο σχήμα.



Σχήμα 8: Παράδειγμα Zero Padding για την πρώτη γραμμή της εικόνας.

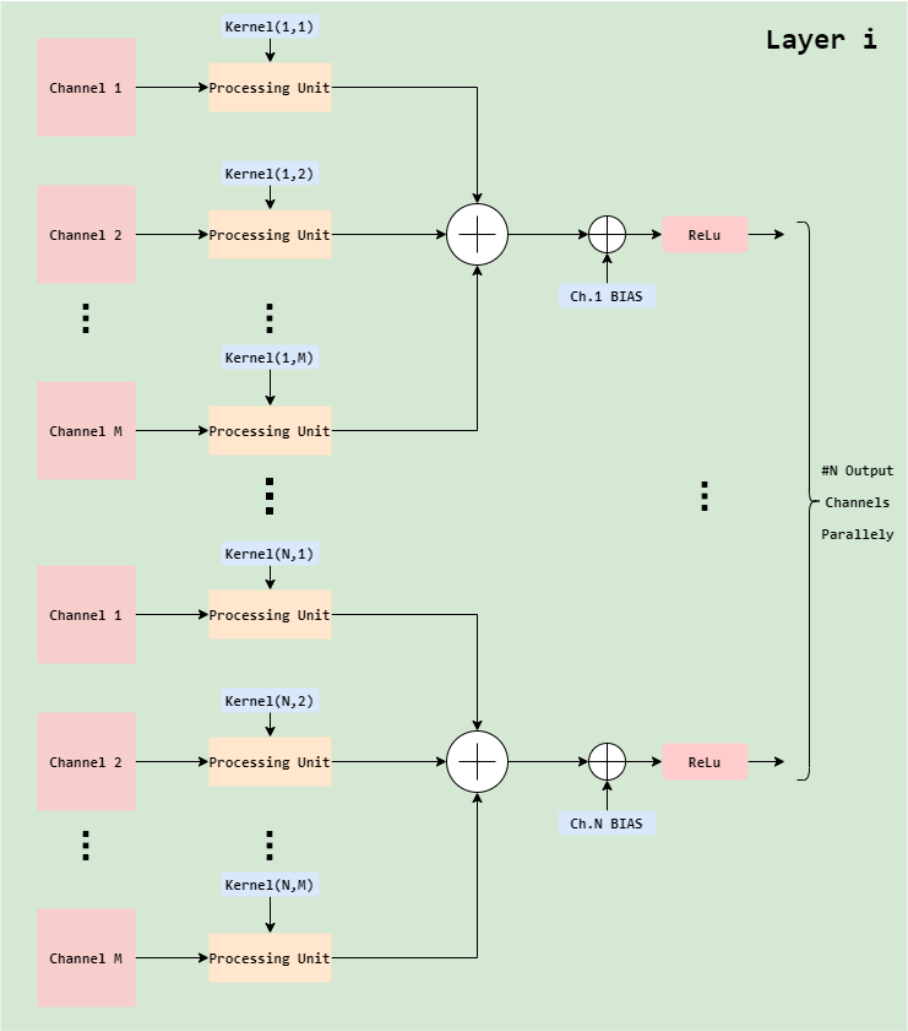
Συνελικτική Μονάδα

Σε αυτή την μονάδα λαμβάνουν χώρα οι υπολογισμοί. Το φίλτρο και η εικόνα δίνονται σαν είσοδοι σε αυτήν. Ορίζουμε $Kernel_{Width} := M, Kernel_{Height} := N$. Σε αυτήν την μονάδα $M \times N$ πολλαπλασιασμοί εκτελούνται παράλληλα για κάθε στοιχείο εξόδου. Στη συνέχεια αυτά τα γινόμενα δίνονται ως είσοδοι σε έναν $M \times N$ εισόδων Δέντρο-Αθροιστή για να υπολογιστεί το τελικό στοιχείο.



Σχήμα 9: Δομή Κλασσικής Μονάδας Επεξεργασίας

Η μονάδα Επεξεργασίας αποτελείται από μια μνήμη RAM κατασκευασμένη όπως εξηγήθηκε στη προηγούμενη ενότητα, το φίλτρο το οποίο είναι αποθηκευμένο σε μνήμη ROM[1] και την συνελικτική μονάδα στην οποία γίνονται οι υπολογισμοί. Με χρήση της μονάδας επεξεργασίας, μπορούμε να κατασκευάσουμε ολόκληρο το συνελικτικό επίπεδο ενός συνελκτικού νερωνικού δικτύου. Αρκεί να παραγάγουμε την συγκεκριμένη μονάδα κατάλληλες φορές και να την συνδυάσουμε με τα επιπλέον στοιχεία του επιπέδου. Για παράδειγμα για M κανάλια εισόδου και N κανάλια εξόδου, χρειάζεται αυτή η μονάδα να παραχθεί $M \times N$ φορές. Σε κάθε κανάλι εξόδου πρέπει να υπάρχει ένας αθροιστής-δέντρο των M εισόδων και στην συνέχεια να προστίθεται το bias του καναλιού εξόδου. Το τελικό αποτέλεσμα μπορεί να φιλτραριστεί από οποιαδήποτε συνάρτηση ενεργοποίησης (π.χ. ReLu). Τα αποτελέσματα του κάθε καναλιού εξόδου είναι συγχρονισμένα και σε κάθε κύκλο ρολογιού παράγεται και ένα αποτέλεσμα εξόδου για κάθε κανάλι.



Σχήμα 10: Κατασκευή ενός συνελικτικού επιπέδου με χρήση της Μονάδας Επεξεργασίας.

[1]Read Only Memory

Σχεδιασμός με χρήση Winograd

Εφόσον οι πόροι των FPGA είναι περιορισμένοι, προκύπτει η ανάγκη με αυτούς τους πόρους να πετύχουμε όσο το δυνατόν καλύτερη απόδοση. Η συνέλιξη με την χρήση του Γρήγορου Μετασχηματισμού Φουριέρ (FFT) είναι μια γρήγορη προσέγγιση αλλά απευθύνεται κυρίως σε μεγάλα φίλτρα. Τα τελευταία υπερσύγχρονα συνελικτικά νευρωνικά δίκτυα χρησιμοποιούν μικρά, 3×3 ή 5×5 φίλτρα για την εξαγωγή χαρακτηριστικών. Ο ελάχιστος αλγόριθμος φίλτρου του Winograd [19] για τον υπολογισμό m εξόδων με ένα FIR φίλτρο $r-$σημείων το οποίο αποκαλείται $F(m,r)$, απαιτεί

$$\mu(F(m,r)) = m + r - 1$$

πολλαπλασιασμούς. Επίσης, οι ελάχιστοι μιας διάστασης $F(m,r)$ και $F(n,s)$ μπορούν να εμφωλευτούν για να σχηματίσουν δισδιάστατους αλγορίθμους που υπολογίζουν $m \times n$ εξόδους με ένα $r \times s$ φίλτρο, το οποίο αποκαλείται $F(m \times n, r \times s)$.

$$\mu(F(m \times n, r \times s)) = \mu(F(m,r))\mu(F(n,s)) = (m + r - 1)(n + s - 1)$$

Με άλλα λόγια, για να υπολογίσουμε το $F(m,r)$ πρέπει να προσπελάσουμε ένα διάστημα με $m+r-1$ τιμές δεδομένων, και για να υπολογίσουμε το $F(m\times n, r\times s)$ πρέπει να προσπελάσουμε μια στοίβα με $(m + r - 1) \times (n + s - 1)$ τιμές δεδομένων. Έτσι, ο ελάχιστος αλγόριθμος για τα φίλτρα, χρειάζεται έναν πολλαπλασιασμό για κάθε είσοδο.

Υλοποίηση Winograd για φίλτρο μεγέθους 3×3

Ο κλασσικός αλγόριθμος για το $F(2,3)$, χρησιμοποιεί $2\times3 = 6$ πολλαπλασιασμούς. Ο Winograd έφτιαξε τον παρακάτω ελάχιστο αλγόριθμο :

$$F(2,3) = \begin{bmatrix} d_0 & d_1 & d_2 \\ d_1 & d_2 & d_3 \end{bmatrix} \begin{bmatrix} g_0 \\ g_1 \\ g_2 \end{bmatrix} = \begin{bmatrix} m_1 + m_2 + m_3 \\ m_2 - m_3 - m_4 \end{bmatrix}$$

Όπου

$$m_1 = (d_0 - d_2)g_0 \quad m_2 = (d_1 + d_2)\frac{(g_0 + g_2) + g_1}{2}$$
$$m_4 = (d_1 - d_3)g_2 \quad m_3 = (d_2 - d_1)\frac{(g_0 + g_2) - g_1}{2}$$

Ο αλγόριθμος χρησιμοποιεί απλά 4 πολλαπλασιασμούς και επομένως είναι ελάχιστος σύμφωνα με την εξίσωση
$\mu(F(2,3)) = 2 + 3 - 1 = 4$. Επίσης χρησιμοποιεί 4 προσθέσεις που έχουν σχέση με τα δεδομένα d, 3 προσθέσεις και 2 πολλαπλασιασμούς με σταθερά εμπλέκοντας το φίλτρο (το άθροισμα $g_0 + g_2$ στην παρένθεση αρκεί να υπολογιστεί μία φορά), και 4 προσθέσεις για να μειώσει τα γινόμενα του τελικού αποτελέσματος. Οι αλγόριθμοι για γρήγορο φιλτράρισμα μπορούν να γραφούν σε μορφή πινάκων όπως φαίνεται παρακάτω :

$$Y = A^T[(Gg) \odot (B^T d)]$$

Όπου $\odot$ υποδηλώνει στοιχείο προς στοιχείο πολλαπλασιασμό πινάκων. Για το $F(2,3)$, οι πίνακες είναι :

$$B^T = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \quad G = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix} \quad A^T = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix}$$

$$g = \begin{bmatrix} g_0 & g_1 & g_2 \end{bmatrix}^T \quad d = \begin{bmatrix} d_0 & d_1 & d_2 & d_3 \end{bmatrix}^T$$

Ο ελάχιστος μονοδιάστατος αλγόριθμος $F(m,r)$ εμφωλεύεται με τον εαυτό του για να αποκτήσουμε εναν δισδιάστατο ελάχιστο αλγόριθμο, $F(m \times m, r \times r)$

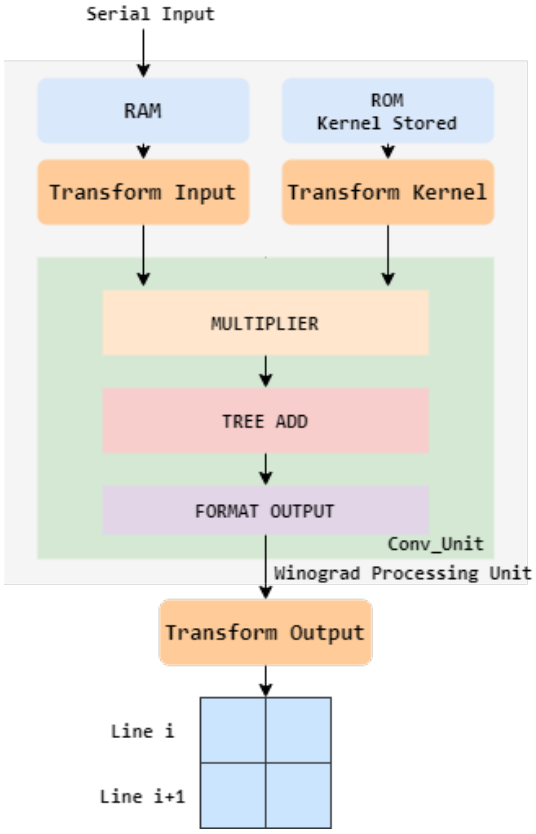
$$Y = A^T \Big[[GgG^T] \odot [B^T dB] \Big] A$$

Όπως αποδεικνύεται στο [19]
Όπου τώρα το g είναι ένα $r \times r$ φίλτρο και το d είναι μια $(m+r-1) \times (m+r-1)$ στοίβα εικόνας.
$F(2 \times 2, 3 \times 3)$ χρησιμοποιεί $4 \times 4 = 16$ πολλαπλασιασμούς για να υπολογίσει 4 αποτελέσματα εξόδου, ενώ ο κλασσικός αλγόριθμος συνέλιξης που παρουσιάστηκε προηγουμένως $4 \times 3 \times 3 = 36$. Οπότε υπάρχει μια μείωση της αριθμητικής πολυπλοκότητας της τάξης του : $\frac{36}{16} = 2.25$. Τώρα θα αναλύσουμε το επιπλέον κόστος που απαιτείται σε σύγκριση με τον κλασσικό αλγόριθμο της συνέλιξης. Προκειμένου να μετασχηματίσουμε τα δεδομένα εισόδου στην σωστή μορφή $B^T dB$ χρειάζονται 32 απλές πράξεις (προσθέσεις-αφαιρέσεις). Ο μετασχηματισμός του φίλτρου χρειάζεται 28 πράξεις floating point οι οποίες μπορούν να αναχθούν σε απλές πράξεις χρησιμοποιώντας fixed point δεδομένα. Ο μετασχηματισμός στην έξοδο απαιτεί επίσης 24 απλές πράξεις.
Προκειμένου να σχεδιάσουμε την υλοποίηση του Winograd για φίλτρο διαστάσεων 3×3 κάναμε τις ακόλουθες τροποποιήσεις στον αρχικό σχεδιασμό της κλασσικής συνέλιξης.

- Αρχικά, αντί για 3×3 στοίβες με βήμα 1 που χρησιμοποίηθηκαν στο κλασσικό σχέδιο, τώρα πρέπει να δουλέψουμε με 4×4 στοίβες με βήμα 2. Στο κλασσικό σχέδιο έιχαμε $P = (Image_{Width}) \times (Image_{Height})$ στοίβες για κάθε κανάλι εισόδου. Τώρα κάθε στοίβα έχει δύο στοιχεία τα οποία επικαλύπτουν τα γειτονικά τους και καταλήγουμε σε : $P' = (Image_{Width}/2) \times (Image_{height}/2)$ στοίβες για κάθε κανάλι εισόδου.
  Αντί για 3 ουρές FIFO που χρησιμοποιήθηκαν στο προηγούμενο σχέδιο τώρα πρέπει να χρησιμοποιήσουμε 4 έτσι ώστε να χωρέσουν τα απαραίτητα δεδομένα για να αρχίσουν οι υπολογισμοί. Στο παράθυρο ολίσθησης πρέπει να προστεθούν επιπλέον καταχωρητές για να έχουμε ανά πάσα στιγμή την 4×4 στοίβα διθέσιμη.

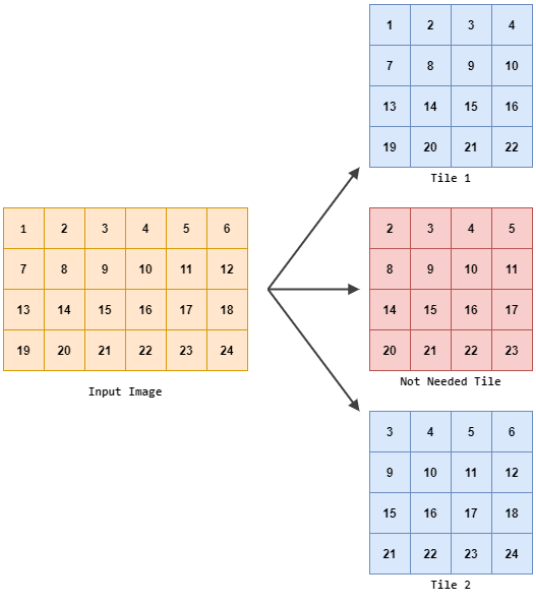
- Η ολίσθηση δεδομένων και η μετατροπή από σειριακό σε παράλληλο καθώς και οι περιορισμοί υλοποιούνται με ακριβώς τον ίδιο τρόπο όπως στην κλασσική υλοποίηση, σεβόμενα τις ιδιαιτερότητες του Winograd .

- Μονάδα Μετασχηματισμού Εισόδου : Δέχεται ως είσοδο μια 4×4 στοίβα, εκτελεί τις κατάλληλες αφαιρέσεις και προσθέσεις και δίνει ως έξοδο την κατάλληλης μορφής μεταχηματισμένη είσοδο.
- Μονάδα Μετασχηματισμού Φίλτρου : Δέχεται ως είσοδο το 3×3 φίλτρο και δίνει ως έξοδο το 4×4 μετασχηματισμένο φίλτρο.
- Μονάδα Μετασχηματισμού Εξόδου : Μετασχηματίζει την έξοδο από 4×4 στην τελική 2×2 έξοδο.



Σχήμα 11: Δομή Μονάδας Επεξεργασίας Winograd

Για να συγκρίνουμε τα πλεονεκτήματα του Winograd με την κλασσική υλοποίηση θα πρέπει να λάβουμε υπόψιν τις επιπλέον πράξεις που απαιτούνται έτσι ώστε να μετασχηματίσουμε τα δεδομένα. Όπως αναφέρθηκε προηγουμένως στην περίπτωση του Winograd οι στοίβες εισόδου πρέπει να έχουν βήμα 2 τόσο στο πλάτος όσο και στο ύψος. Εφόσον τα δεδομένα λαμβάνονται σειριακά, προκειμένου να υλοποιηθεί ο μετατροπέας από σειριακό σε παράλληλο με βήμα 2, πρέπει να περιμένουμε για έναν κύκλο ρολογιού έτσι ώστε να πάρουμε την επόμενη έγκυρη στοίβα. Είναι προφανές ότι σε αυτούς τους κύκλους ρολογιού, εκτός από το να γίνεται η επιθυμητή ολίσθηση δεδομένων στη μνήμη, δεν μπορεί να γίνει διοχέτευση κάποιας άλλης χρήσιμης λειτουργίας που να παράξει έξοδο καθώς δεν υπάρχουν έγκυρα δεδομένα διαθέσιμα στη μνήμη.

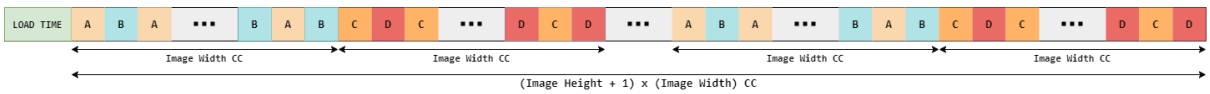


Σχήμα 12: Ανάμεσα σε δύο έγκυρες στοίβες υπάρχει μια μη έγκυρη την οποία δεν μπορούμε να αξιοποιήσουμε

Όπως φαίνεται στο Σχήμα 13, όσον αφορά το πλάτος ανάμεσα σε κάθε έγκυρο αποτέλεσμα υπάρχει ένα άκυρο και όσον αφορά το ύψος πρέπει να περιμένουμε για $Image_{Width}$ κύκλους ρολογιού για να φορτωθεί η επόμενη έγκυρη στοίβα.

Σχήμα 13: Υπο-χρησιμοποίηση της μονάδας για ένα κανάλι εισόδου. Τα κόκκινα τετράγωνα υποδηλώνουν άκυρες στοίβες ενώ τα μπλέ τις έγκυρες.

Όπως φαίνεται η μονάδα υπο-χρησιμοποιείται κατά παράγοντα 1/4. Για να κάνουμε πλήρη αξιοποίηση αυτής της μονάδας και να πετύχουμε την μέγιστη απόδοση, 4 κανάλια εισόδου πρέπει να επεξεργάζονται διαδοχικά. Στους κύκλους ρολογιού οι οποίοι δεν παράγουν έγκυρο αποτέλεσμα, με το κατάλληλο κύκλωμα ελέγχου μπορούμε να παράξουμε έγκυρα αποτελέσματα για άλλα κανάλια καθώς γίνονται οι απαραίτητες ολισθήσεις για το προηγούμενο κανάλι εισόδου.



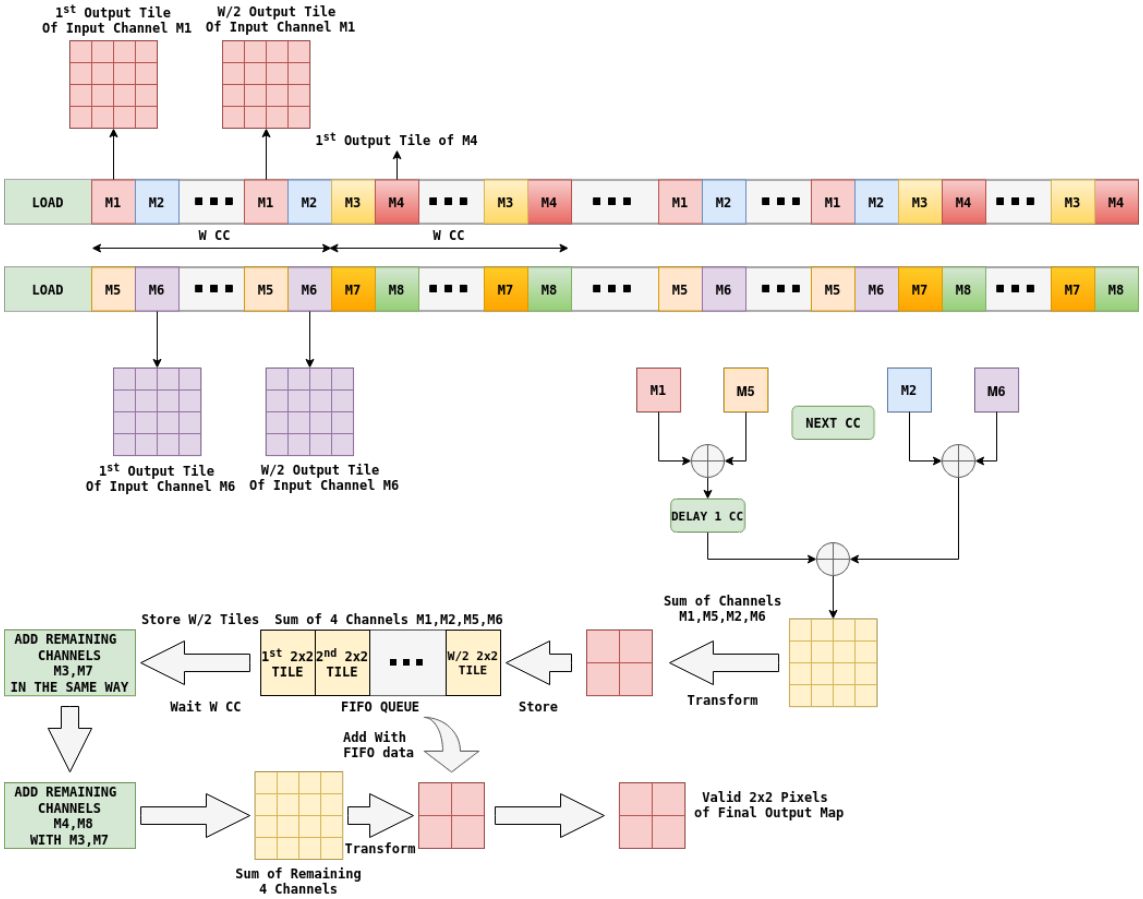
Σχήμα 14: Πλήρης αξιοποίηση της μονάδας για τέσσερα διαφορετικά κανάλια εισόδου.

Με την ίδια λογική, προσαρμόσαμε την υλοποίηση Winograd για να λειτουργεί για οποιοδήποτε αριθμό καναλιών εισόδου και να παράγει σωστά το αντίστοιχο κανάλι εξόδου. Όπως

Table 1: Σύγκριση Μονάδων Επεξεργασίας ($w \times w$ Image, 3×3 Kernel)

	Direct	**Winograd**	**Full Winograd**
Input Channels	1	1	4
RAM FIFOs (S2P)	3	4	16
DFFs (S2P)	9	16	64
ROMs	1	1	4
Multipliers	9	16	16
Adders	8	56	56
Latency	w+2 CCs	2w+3 CCs	2w+3 CCs
Throughput	1pixel/CC	1pixel/CC	4pixels/CC

φαίνεται στο Σχήμα 15, χρείαζεται να γίνει πρόσθεση των καναλιών εξόδου για κάθε κύκλο ρολογιού και στην συνέχεια αποθήκευση αυτών σε μια ουρά FIFO ώστε να προστεθούν και τα υπόλοιπα κανάλια τα οποία παράγονται μετά από $Image_{Width}$ κύκλους ρολογιού.

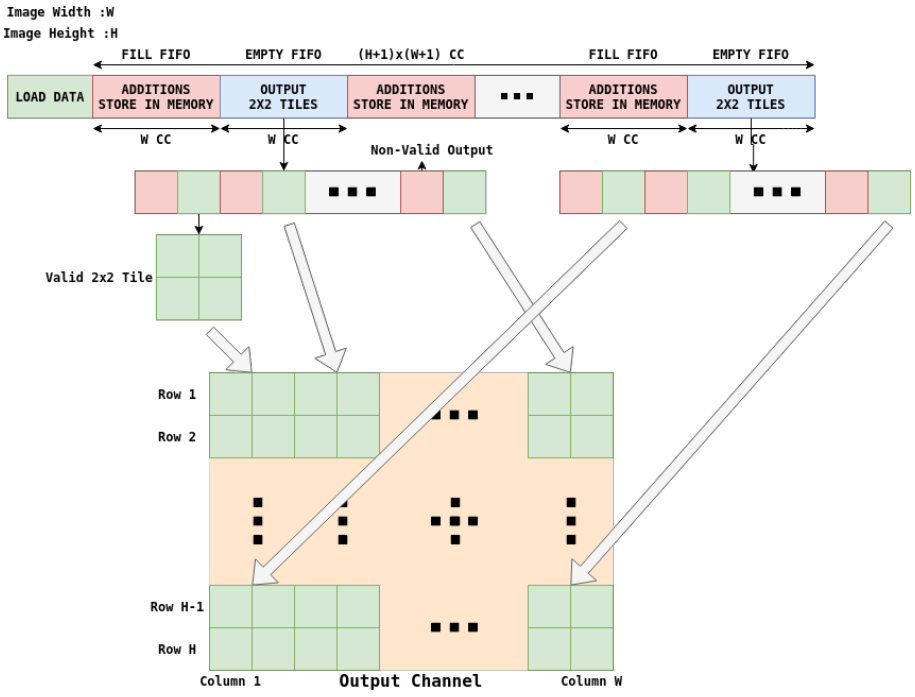


Σχήμα 15: Υλοποίηση για ένα κανάλι εξόδου με οχτώ κανάλια εισόδου

Table 2: Κλασσική (Direct) και Winograd υλοποίηση των συνελικτικών επιπέδων($W \times W$ Image, M channels, 3×3 Kernel, N filters)

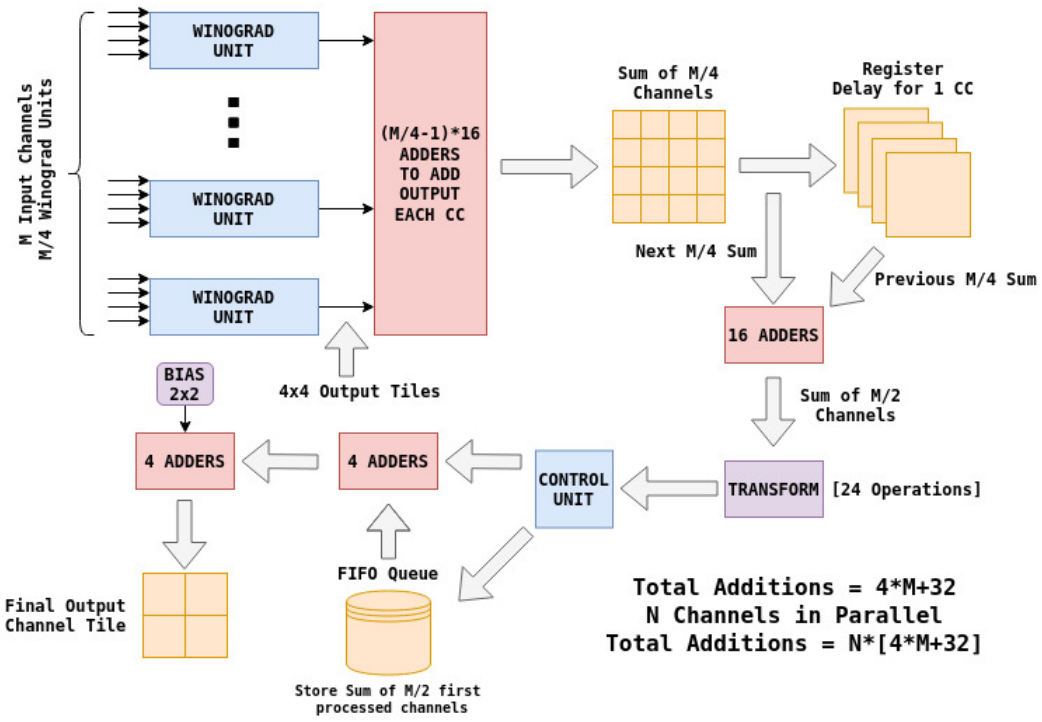
	Direct	**Full Winograd**
Units	$N \cdot M$	$N \cdot M/4$
- RAM FIFOs (S2P)	$N \cdot 3M$	$N \cdot 4M$
- DFFs (S2P)	$N \cdot 9M$	$N \cdot 4M$
- ROMs	$N \cdot M$	$N \cdot M$
- Multipliers	$N \cdot 9M$	$N \cdot 4M$
- Adders	$N \cdot 8M$	$N \cdot 8M - N \cdot 15M$[1]
RAMs	0	N FIFO RAM
Adders	$N \cdot (M-1)$	$N \cdot (4M+32)$
ReLu	N	$4N$
Latency	w+2 CCs	2w+3 CCs
Throughput	Npixels/CC	Npixels/CC

[1] $N \cdot 15M$ Αθροιστές απαιτούνται για να μετασχηματιστεί το φίλτρο.



Σχήμα 16: Υπολογισμός χάρτη εξόδου με χρήση του αλγορίθμου Winograd

Στο Σχήμα 17, παρουσιάζεται η υλοποίηση Winograd για M κανάλια εισόδου.



Σχήμα 17: Διαχείριση για M κανάλια εισόδου

Χρονοδρομολόγηση

Στην κλασσική υλοποίηση, όπως αναφέρθηκε προηγουμένως χρειάζονται $\lfloor Kernel_{Height}/2 \rfloor \times (Image_{Width}) + \lceil Kernel_{Width}/2 \rceil$ κύκλοι ρολογιού για να γίνει φόρτωση των απαραίτητων δεδομένων πριν παραχθούν έγκυρες έξοδοι. Στην συνέχεια $M \times N$ πολλαπλασιασμοί εκτελούνται παράλληλα σε έναν κύκλο ρολογιού. Αφού πραγματοποιηθούν οι πολλαπλασιασμοί τα $M \times N$ δεδομένα πρέπει να αθροιστούν για να υπολογιστεί το τελικό στοιχείο εξόδου. Αυτή η πρόσθεση γίνεται με έναν αθροιστή δέντρο της επιλογής του χρήστη. Συνεπώς για να λάβουμε την πρώτη έγκυρη έξοδο χρειάζονται :

$$\text{Cycles-First-Output} = \lfloor Kernel_{Height}/2 \rfloor \times (Image_{Width}) + \lceil Kernel_{Width}/2 \rceil + 3$$

Αφότου παραχθεί η πρώτη έξοδος η παραπάνω διαδικασία διοχετεύεται πλήρως. Σε κάθε κύκλο ρολογιού παράγεται ένα στοιχείο εξόδου. Για να γίνει πλήρης επεξεργασία μιας εικόνας εισόδου χρειάζονται :

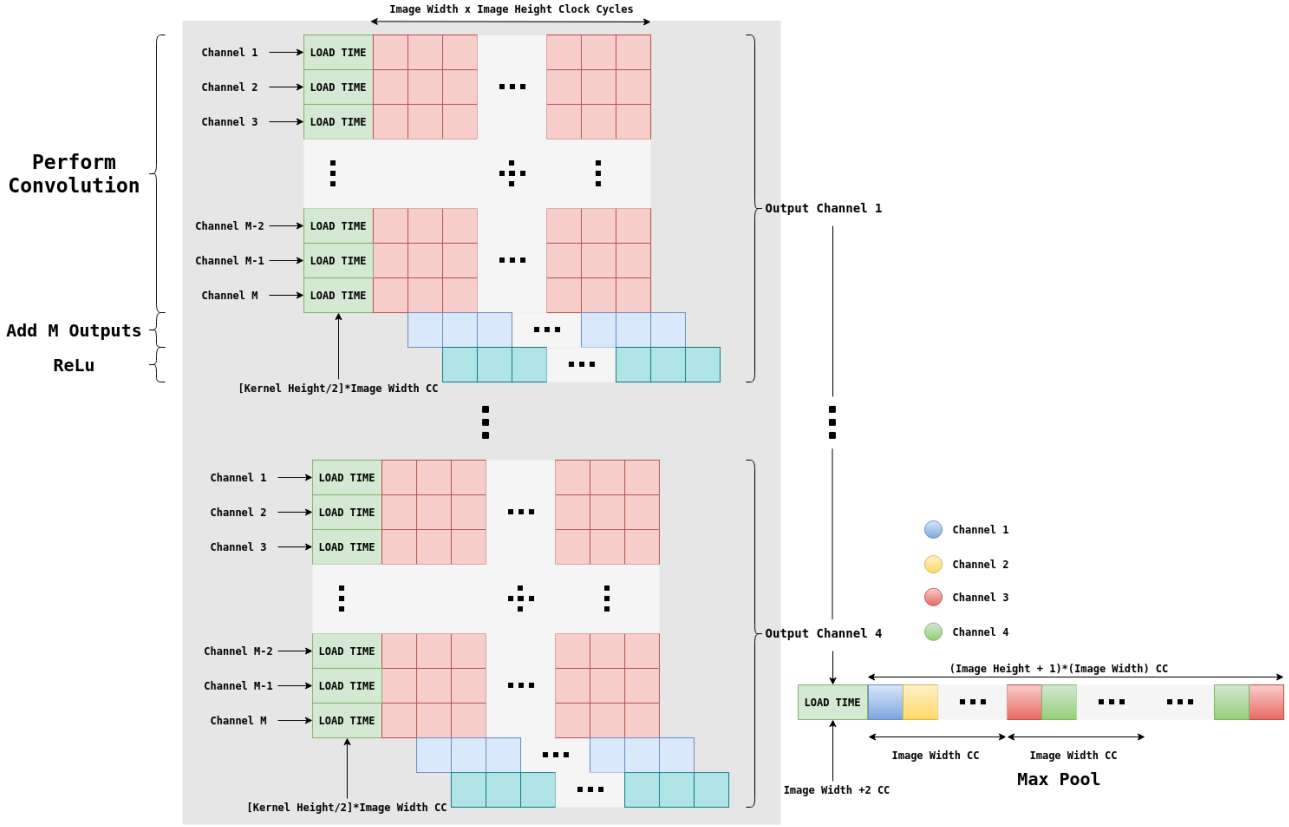
$$\text{Cycles-Full-Image} = \text{Cycles-First-Output} + (Image_{Width}) \times (Image_{Height})$$

Όσον αφορά το Max-Pooling μπορεί να γίνει παράλληλα με την διαδικασία της συνέλιξης. Χρειάζεται $Image_{Width} + 2$ κύκλους ρολογιού για να φορτώσει τα απαραίτητα στοιχεία και άλλους 2 κύκλους ρολογιού για να γίνουν οι συγκρίσεις. Μετά από αυτούς τους κύκλους ρολογιού παράγεται μια έγκυρη έξοδος σε κάθε επόμενο κύκλο.

$$\text{Cycles-Max-Pool} = Image_{Width} + 3 + (Image_{Width}) \times (Image_{Height})$$

Συνολικά για να πραγματοποιηθεί η συνέλιξη και η διαδικασία Max-Pool χρειάζονται :

$$\begin{aligned}\text{Cycles-Total} &= Image_{Width} + 2 + Image_{Width} + 2 + (Image_{Width}) \times (Image_{Height}) \\ &= 2 \times Image_{Width} + (Image_{Width}) \times (Image_{Height}) + 4 \quad \text{CCs}\end{aligned}$$



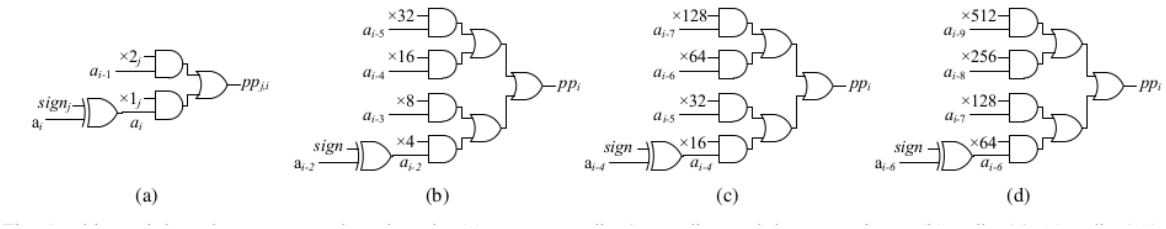
Σχήμα 18: Κύκλοι ρολογιού για M κανάλια εισόδου και 4 κανάλια εξόδου. Μια μονάδα Max-Pool απαιτείται για 4 κανάλια εξόδου.

Προσεγγίσεις

Προσεγγιστικός Υβριδικός Πολλαπλασιαστής Υψηλής Ακτίνας

Οι κωδικοποιήσεις υψηλής ακτίνας [41] προσφέρουν μείωση μερικών γινομένων και ως άμεσο αποτέλεσμα η πρόσθεση τους απαιτεί μικρότερα δέντρα αθροιστών οδηγώντας σε εξοικονόμηση ενέργειας, χώρου και μείωση της καθυστέρησης. Σε αντίθεση οι υψηλές κωδικοποιήσεις απαιτούν πολύπλοκα κυκλώματα κωδικοποίησης και κυκλώματα παραγωγής μερικών γινομένων ισοσταθμίζοντας τα πλεονεκτήματα της μείωσης του αριθμού των μερικών γινομένων. Στο [20] γίνεται παρουσίαση ενός τέτοιου πολλαπλασιαστή και οι προσεγγίσεις που οδηγούν στην μείωση της πολυπλοκότητας του κυκλώματος αναλύονται σε βάθος. Σε αυτή την τεχνική ο πολλαπλασιαστέος B κωδικοποιείται με χρήση της προσεγγιστικής κωδικοποίησης μεγάλης ακτίνας παράγοντας τον $\tilde{B}$ και πραγματοποιείται ο πολλαπλασιασμός $A \cdot \tilde{B}$ αντί του $A \cdot B$.

Ανάλογα με την προσεγγιστική παράμετρο k γίνεται η κωδικοποίηση του B και επομένως καθορίζεται η παραγωγή των μερικών γινομένων. Το k μπορεί να πάρει τιμές 4, 6, 8 και 10. Όσο μεγαλύτερη είναι η τιμή του, τόσο μειώνεται ο αριθμός των μερικών γινομένων και η προσέγγιση του πολλαπλασιασμού γίνεται μεγαλύτερη.

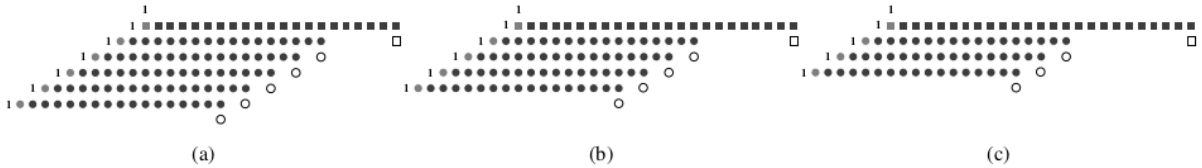


Σχήμα 19: $i-bit$ παραγωγή μερικών γινομένων βασισμένη σε : (α) ακριβές $radix-4$ κωδικοποίηση και τις προσεγγιστικές (β) $radix-64$, $(c) radix-256$, $(d) radix-1024$ κωδικοποιήσεις. $a_i : i-bit$ του A, $\alpha_i = \alpha_i \oplus sign$

Radix Encoding	**Partial Produxts**
Radix-4	$0, \pm A, \pm 2A$
Radix-64	$0, \pm 4A, \pm 8A, \pm 16A, \pm 32A$
Radix-256	$0, \pm 16A, \pm 32A, \pm 64A, \pm 128A$
Radix-1024	$0, \pm 64A, \pm 128A, \pm 256A, \pm 512A$

Table 3: Μερικά Γινόμενα ανάλογα με την κωδικοποίηση

Η επιλογή του συσσωρευτή των μερικών γινομένων μπορεί να γίνει ελεύθερα από τον χρήστη.



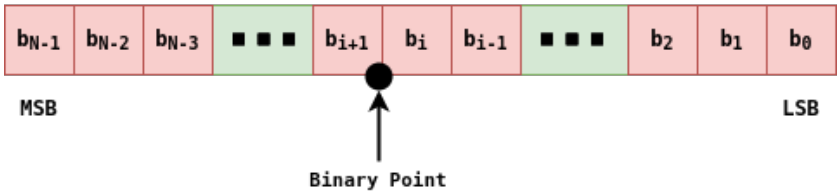
Σχήμα 20: Το δέντρο μερικών γινομένων βασισμένο στην υβριδική κωδικοποίηση του ακριβές $radix-4$ και του προσεγγιστικού (α) $radix-64$, $(b) radix-256$ και $(c) radix-1024$ κωδικοποίηση.■ : Μερικά γινόμενα από την προσεγγιστική κωδικοποίηση υψηλής ακτίνας, ●: Μερικά γινόμενα από την ακριβή $radix-4$ κωδικοποίηση, ■ και ● : αντεστραμμένα $MSBs$ των μερικών γινομένων, □ και ○ : παράγοντες προσήμου

Εξερεύνηση Τύπων Δεδομένων

Στο ψηφιακό υλικό, οι αριθμοί αποθηκεύονται σε δυαδικές λέξεις. Μια δυαδική λέξη είναι μια καθορισμένου μήκους ακολουθία από bits. Ο τρόπος με τον οποίο τα μέρη του υλικού ή οι συναρτήσεις του λογισμικού αντιλαμβάνονται αυτές τις ακολουθίες από 0 και 1 καθορίζεται από τον τύπο δεδομένων. Οι δυαδικοί αριθμοί, αναπαρίστανται είτε ως fixed point είτε ως floating point τύποι δεδομένων. Τα δημοφιλή εργαλεία νευρωνικών δικτύων, όπως είναι τα Tensorflow, Keras και Caffe λειτουργούν με 32-bit Floating Point δεδομένα. Εφόσον το εύρος μνήμης, η επιφάνεια και η κατανάλωση ενέργειας είναι οι κύριοι στόχοι της βελτιστοποίησης του σχεδίου μας, ερευνήσαμε σε βάθος διάφορους τύπους δεδομένων και τα πλεονεκτήματα- μειονεκτήματα τους σε διάφορες πτυχές όπως είναι η ακρίβεια και η απόδοση. Οι γενικής χρήσης επεξεργαστές δεν είναι βελτιστοποιημένοι για μισού ή μικρότερου μήκους floating point πράξεις. Εφόσον πολλές εφαρμογές σε υλικό χρησιμοποιούν τύπους δεδομένων με μήκος μικρότερο των 16-bit καθώς οι κάρτες γραφικών μπορούν να εκτελέσουν τέτοιες πράξεις πολύ αποδοτικά, πολλά εργαλεία για την εκπαίδευση νευρωνικών δικτύων υποστηρίζουν πλέον τέτοιους τύπους δεδομένων. Στην υλοποίηση μας, δεν υπάρχει ανάγκη να γίνει επανεκπαίδευση του δικτύου για διαφορετικούς τύπους δεδομένων, καθώς οι μετατροπές γίνονται στο πρώτο στάδιο του δικτύου.

Αναπαράσταση Fixed Point

Στον fixed point τύπο δεδομένων, κάθε N-bit Floating Point μήκους ακολουθία από bits χαρακτηρίζεται από το μήκος λέξης, την θέση του δυαδικού σημείου και το αν είναι προσημασμένος ή μη προσημασμένος. Η θέση του δυαδικού σημείου είναι ο παράγοντας που καθορίζει την κλιμάκωση και την ερμήνευση των τιμών.



Σχήμα 21: Αναπαράσταση N-bit Fixed Point αριθμού

Στο σχήμα 21 μια δυαδική αναπαράσταση ενός γενικού Fixed Point αριθμού (προσημασμένου ή μη) παρουσιάζεται. Σε αυτήν την αναπαράσταση :

- N είναι το μήκος λέξης σε bits.
- b_i είναι το i^{ost} δυαδικό ψηφίο.
- b_{N-1} και b_0 είναι οι θέσεις του πιο σημαντικού και λιγότερου σημαντικού ψηφίου αντίστοιχα.

Το δυαδικό σημείο φαίνεται $i+1$ θέσεις αριστερά από το λιγότερο σημαντικό ψηφίο οπότε ο αριθμός λέγεται ότι έχει $i+1$ κλασματικά bits ή κλάσμα με μήκος $i+1$. Οι αριθμοί Fixed Point μπορούν να κωδικοποιηθούν σύμφωνα με το ακόλουθο σχήμα :

$$X = 2^{-FractionalLength} \times (\text{Stored Integer})$$

$$X = (\frac{1}{2^b}) \times \Big[-2^{N-1}x_{N-1} + \sum_{n=0}^{N-2} 2^n x_n \Big]$$

Όπου X είναι η πραγματική αποθηκευμένη τιμή του ακέραιου και b το κλασματικό μήκος.

Η αναπαράσταση Fixed Point είναι πολύ κοντά στην αναπαράσταση ακέραιων αριθμών. Στην πραγματικότητα οι ακέραιοι μπορούν να θεωρηθούν ως μια ¨ειδική κατηγορία' Fixed Point με το κλασματικό μέρος στην θέση 0. Όλες οι αριθμητικές πράξεις στον υπολογιστή που εφαρμόζονται για ακέραιους αριθμούς μπορούν να εφαρμοστούν και για αριθμούς που αναπαρίστανται ως Fixed Point.

Αναπαράσταση Floating Point

Οι σύγχρονες εφαρμογές όπως είναι και τα Συνελικτικά Νευρωνικά Δίκτυα απαιτούν ακρίβεια. Καθώς η έξοδος του κάθε επιπέδου προωθείται ως είσοδος σε επόμενο επίπεδο, όταν προκύπτει ένα σφάλμα στην ακρίβεια σε κάποιο σημείο της διαδικασίας, είναι πιθανό οτι αυτό το σφάλμα θα γίνει αντιληπτό και ενδεχομένως να γίνει μεγαλύτερο σε επόμενα στάδια του δικτύου. Η Floating Point αναπαράσταση είναι αυτή που οι περισσότεροι σύγχρονοι υπολογιστές χρησιμοποιούν όταν αποθηκεύουν δεκαδικούς αριθμούς στη μνήμη. Μπορεί να δείξει τόσο πολύ μεγάλους όσο και πολύ μικρούς αριθμούς με αξιοσημείωτη ακρίβεια. Λόγω της

ευρείας χρήσης του για την αποθήκευση αριθμών, έχει τυποποιηθεί από το ινστιτούτο Ηλεκτρολόγων Μηχανικών (institute of Electrical and Electronic Engineers) στο πρότυπο IEEE 754. Αυτό το πρότυπο ορίζει τους εξής τύπους δεδομένων όταν αποθηκεύουμε Floating Point αριθμούς στη μνήμη :

- Half Precision : 16-bits of storage
- Single Precision : 32-bits of storage
- Double Precision : 64-bits of storage
- Quadruple Precision : 128-bits of storage

Όταν πρόκειται για την αποθήκευση στη μνήμη, αποθηκεύονται τρία κύρια μέρη του αριθμού : το πρόσημο, ο εκθέτης και η μαντίσσα.

Σχήμα 22: Floating Point Αναπαράσταση

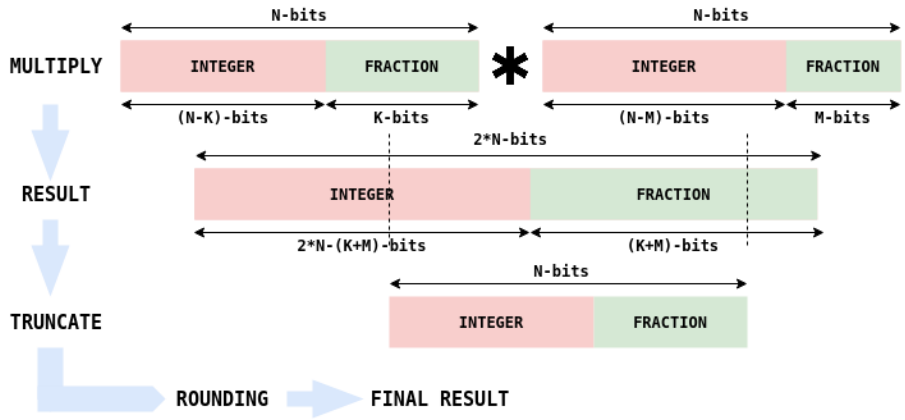
Στο πρότυπο IEEE 754 οι απλής ακρίβειας αριθμοί έχουν : 1 ψηφίο για το πρόσημο, 8 ψηφία για τον εκθέτη και 23 ψηφία για την μαντίσσα. Οι αριθμοί μπορούν να κωδικοποιηθούν σύμφωνα με την ακόλουθη εξίσωση :

$$X = (-1)^S \times 1.M \times 2^{(E-127)}$$

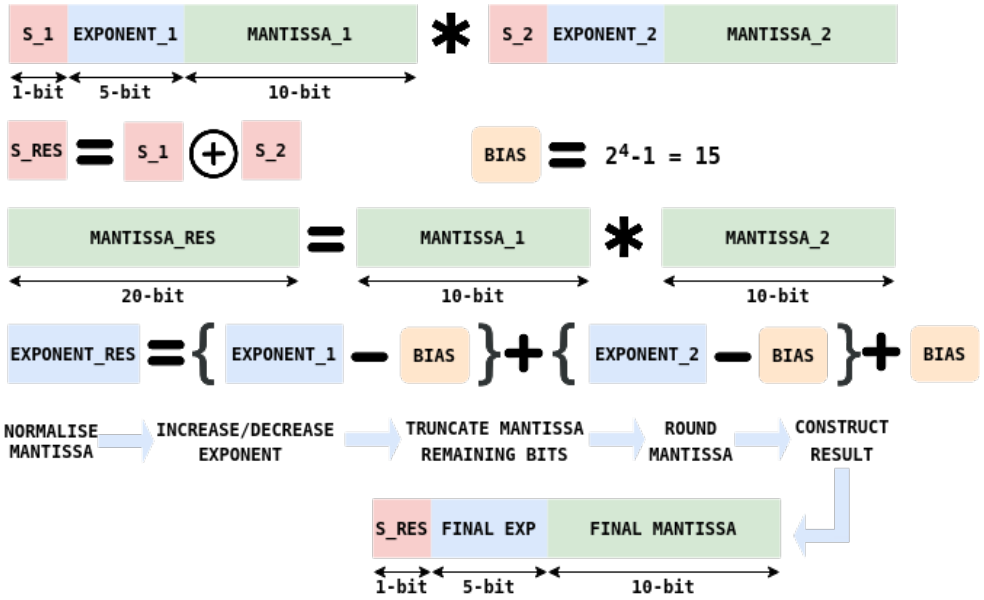
Όπου X είναι ο πραγματικός αριθμός ο οποίος θέλουμε να αναπαραστήσουμε.

Προσεγγίσεις στους διάφορους Τύπους Δεδομένων

Στην υλοποίηση μας, εστιάσαμε στον πολλαπλασιασμό που γίνεται στην διαδικασία της συνέλιξης μεταξύ της εικόνας εισόδου και του φίλτρου. Ανάλογα με τον τύπο δεδομένων, ο πολλαπλασιασμός διαφέρει. Στην περίπτωση των fixed point αριθμών ο πολλαπλασιασμός είναι πολλαπλασιασμός ακέραιων αριθμών με τις κατάλληλες τροποποιήσεις όπως φαίνεται στο Σχήμα 23.

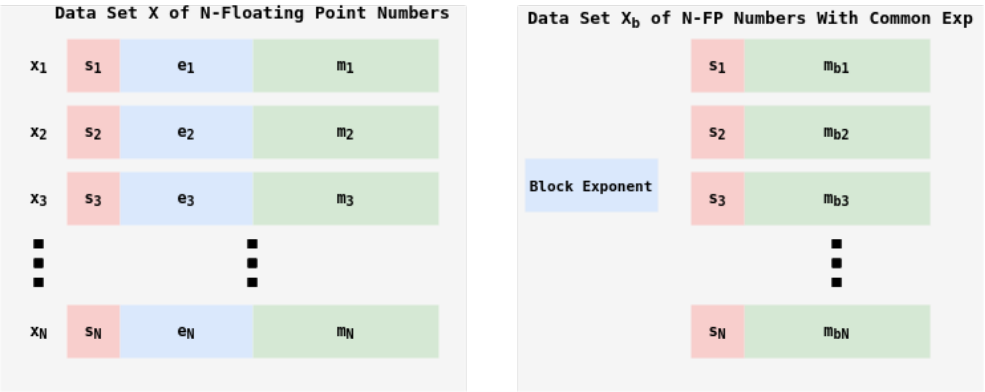


Σχήμα 23: Fixed Point Πολλαπλασιασμός



Σχήμα 24: Floating Point Πολλαπλασιασμός

Μπορούμε να εφαρμόσουμε απευθείας τον προσεγγιστικό πολλαπλασιαστή που παρουσιάστηκε προηγουμένως. Όσον αφορά τον πολλαπλασιασμό Floating Point όπως φαίνεται στο Σχήμα 24, είναι μια πιο περίπλοκη διαδικασία καθώς πρέπει να γίνει ανανέωση και των τριών στοιχείων του αποθηκευμένου αριθμού. Για αυτόν τον λόγο υπάρχουν ειδικές μονάδες για των πολλαπλασιασμό αυτών των αριθμών στους γενικής χρήσεως επεξεργαστές. Επίσης ο πολλαπλασιαστής που παρουσιάστηκε είναι αρχικά φτιαγμένος για αριθμούς με μήκος μεγαλύτερο του 16. Όταν αναφερόμαστε σε μισής ακρίβειας αριθμούς η μαντίσσα έχει μέγεθος 10 ψηφία.



Σχήμα 25: Block Floating Point Αναπαράσταση

Block Floating Point Αριθμητική

Στην προσπάθεια να ξεπεράσουμε την αυξημένη πολυπλοκότητα που προκύπτει από τον πολλαπλασιασμό Floating Point αριθμών, προτείνεται η αριθμητική Block Floating Point (BFP) η οποία είναι μια προσεγγιστική μέθοδος για να αναχθεί ο πολλαπλασιασμός Floating Point αριθμών σε Fixed Point.

Ένα σύνολο N αριθμών που αναπαρίσταται με την BFP αρχιτεκτονική αποτελείται από δύο μέρη : N μαντίσσα και έναν εκθέτη τον οποίο μοιράζονται οι N αριθμοί σε αυτό το σύνολο. Αν υποθέσουμε ότι το σύνολο X περιέχει N Floating Point αριθμούς, τότε μπορεί να εκφραστεί ως :

$$\mathbf{X} = (x_1, ..., x_i, ..., x_N) = (m_1 \times 2^{e_1}, ..., m_2 \times 2^{e_2}, ..., m_N \times 2^{e_N})$$

Ορίζουμε τον μεγαλύτερο εκθέτη στο $\mathbf{X}$ ώς τον εκθέτη του συνόλου αριθμών ϵ_x.

$$\epsilon_x = \max_i e_i \in \{1, 2, ..., N\}$$

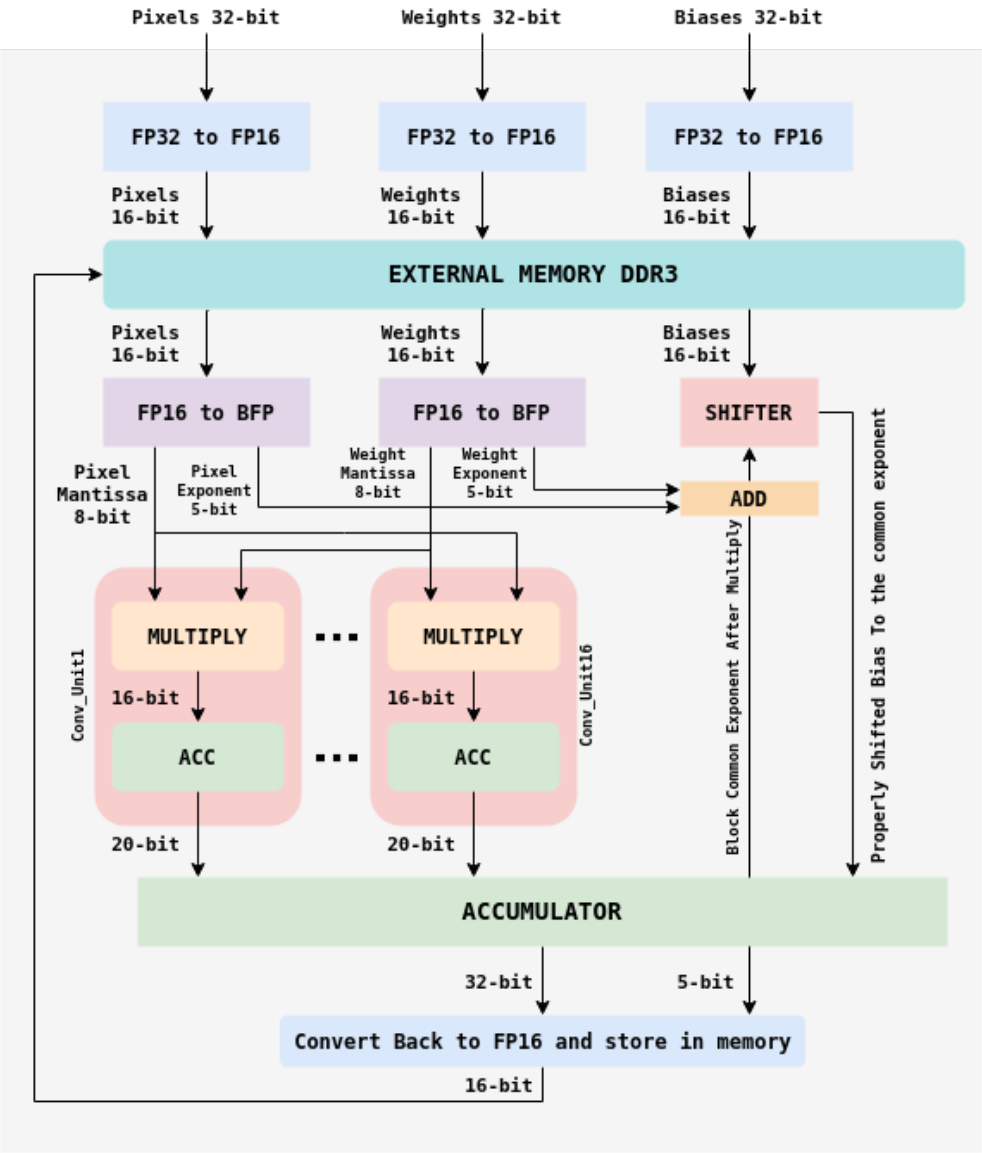
Αφού βρεθεί ο μέγιστος εκθέτης του συνόλου αριθμών, όλοι οι αριθμοί πρέπει να εκφραστούν σύμφωνα με αυτόν. Για να το πετύχουμε αυτό, η μαντίσσα του αριθμού i, m_i πρέπει να υποστεί δεξιά αλίσθηση κατά d_i ψηφία, όπου $d_i = \epsilon_x - e_i$.
Επομένως, η BFP μορφή του συνόλου $\mathbf{X}$ εκφράζεται ως εξής :

$$\mathbf{X}_b = (x_{b1}, ..., x_{bi}, ..., x_{bN}) = M_{bX} \times 2^{\epsilon_x} = (m_{b1}, ..., m_{bi}, ..., m_{bN}) \times 2^{\epsilon_x}$$

Όπου,

$$m_{bi} = m_i >> d_i$$

Είναι η τροποποιημένη κατά BFP μαντίσσα.



Σχήμα 26: Block Floating Point Συνέλιξη

Εφαρμογή της BFP αναπαράστασης στην συνέλιξη

Στο Σχήμα 25 παρουσιάζεται η ροή δεδομένων της BFP συνέλιξης για ένα κανάλι εξόδου. Υποθέτουμε ότι τα δεδομένα εισόδου είναι σε Single Precision Floating Point (32-bit) αναπαράσταση, όπως εκπαιδεύτηκε το δίκτυο σε κάποια πλατφόρμα νευρωνικών. Αρχικά γίνεται μετατροπή αυτών των δεδομένων σε Half Precision Floating Point (16-bit). Αρχικά, βρίσκουμε τον μέγιστο εκθέτη από όλα τα δεδομένα εισόδου. Στο παράδειγμα μας έχουμε 16 εικόνες ως κανάλια εισόδου και βρίσκουμε έναν κοινό εκθέτη από όλες αυτές. Στην συνέχεια για κάθε στοιχείο γίνεται κατάλληλη ολίσθηση της μαντίσσα για να αναπαριστά τον σωστό αριθμό. Στην μονάδα της συνέλιξης που λαμβάνουν χώρα οι πολλαπλασιασμοί, αρκεί να πολλαπλασιάσουμε τις μαντίσσα των φίλτρων και της εικόνας χωρίς να λάβουμε άμεσα υπόψιν τον εκθέτη. Το άθροισμα των εκθετών των καναλιών εισόδου και των φίλτρων χρησιμοποιείται

για να προσαρμοστεί κατάλληλα η πόλωση (Bias) και για να προσαρμοστεί σε τελικό στάδιο ο αριθμός ξανά στο πρότυπο IEEE 754-Half Precision..

Χρησιμοποιώντας αυτήν την αρχιτεκτονική, όλες οι πράξεις στην μονάδα που σχεδιάστηκε (πολλαπλασιασμοί-προσθέσεις) μπορούν να υλοποιηθούν αποκλειστικά με τις μαντίσσα ανανεώνοντας τον εκθέτη μόνο πριν το τελικό αποτέλεσμα. Η Block Floating Point μέθοδος μπορεί να χρησιμοποιηθεί ως μια αριθμητική προσέγγιση floating point χρησιμοποιώντας fixed point επεξεργαστή.

Παρουσίαση και Αξιολόγηση Αποτελεσμάτων

Σε αυτό το κεφάλαιο παρουσιάζονται τα τελικά αποτελέσματα όλης της εργασίας και οι συνδυασμοί που επιλέχθηκαν. Χρησιμοποιώντας τις μονάδες που περιγράφηκαν προηγουμένως κατασκευάστηκε το πλήρες συνελικτικό επίπεδο τριών διαφορετικών Συνελικτικών Νευρωνικών Δικτύων και μετρήθηκαν οι διάφορες διακυμάνσεις στην ακρίβεια και στην χρησιμοποίηση πόρων για διάφορους συνδυασμούς τεχνικών και τύπων δεδομένων.

Αποτελέσματα στο FPGA

Για τις μετρήσεις χρησιμοποιήθηκε το Zynq-7020 Evaluation Board της Xilinx.

Table 4: Πόροι των Συνελικτικών μονάδων μετά την αλλαγή στον μετατροπέα από σειριακό σε παράλληλο

Baseline 3×3	**LUTs**	**FFs**	**BRAM**	**LUTRAM**	**DSP**	**BUFG**	**MAX FREQ (MHz)**	**POWER (W)**
Fixed 16 Hard Multiplier with DSP	327	539	1	3	12	1	151.05	0.187
Fixed 16 Hard Multiplier w/o DSP	1659	892	1	11	0	1	125.78	0.267
Fixed 16 Radix-4	1941	3132	1	147	0	1	139.86	0.267
Fixed 16 Radix-64 (k=6)	1464	2222	1	127	0	1	148.58	0.248
Fixed 16 Radix-256 (k=8)	1198	1861	1	101	0	1	149.4	0.211
Fixed 16 Radix-1024 (k=10)	982	1482	1	98	0	1	133.34	0.196
Float 16 Vivado IP	1926	4402	1	47	25	1	140.84	0.24
BFP per Layer	699	1177	1	15	9	1	149.5	0.179
BFP per 3x3 Window	1895	2017	1	197	9	1	130.63	0.216
BFP per Layer no DSP	1688	1678	1	15	0	1	147.05	0.222

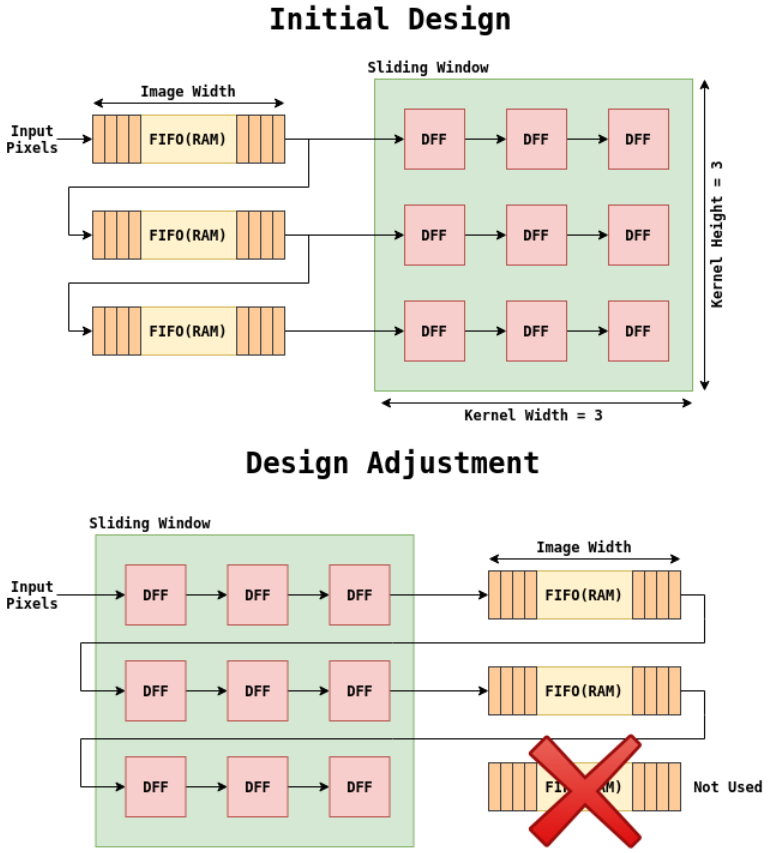
Στον Πίνακα 4, βλέπουμε ότι για τους προσεγγιστικούς πολλαπλασιαστές με την παράμετρο k=6 έχουμε μείωση 12% σε σχέση με τον ακριβή πολλαπλασιασμό, ενώ για $k = 8$ και $k = 10$ έχουμε μείωση 30% και 42% αντίστοιχα.

Στην συνέχεια παρουσιάζεται η δομή και οι απαιτήσεις σε μονάδες του συνελικτικού νευρωνικού δικτύου για τον εντοπισμό πλοίων από εικόνες δορυφόρων "Ship Detection".

Table 5: Απαιτήσεις σε πόρους για το Ship Detectionγια Fixed Point 16 αρχιτεκτονική

	Layer 1	**Layer 2**	**Layer 3**	**Layer 4**
Input Channels	3	32	16	64
Channel Size	128×128	64×64	32×32	16×16
Filters	32	16	64	32
Conv. Units (Table 4.1)	96	512	1024	2048
Kernel RAMB36s	0.5	2.5	4.5	8.5
Adder Trees	11 3-In	1 32-in	2 16-in	1 32-in
ReLu Units	11	1	2	1
MaxPool. Units	3	1	1	1
Output RAMB36s	59	8	8	4

Καθώς η χρησιμοποίηση της μνήμης της πλακέτας ήταν ένα φράγμα κατά την σχεδίαση του τελικού δικτύου, προχωρήσαμε σε μια μετατροπή στον μετατροπέα από σειριακό σε παράλληλο. Αλλάξαμε την σειρά της αλυσίδας των ουρών FIFO με το παράθυρο των καταχωρητών ώστε να γλιτώσουμε μια μνήμη FIFO. Η λειτουργία της μονάδας είναι ακριβώς η ίδια χωρίς να παρουσιάζεται κάποια αλλαγή στον τρόπο που πραγματοποιείται η ροή δεδομένων.



Σχήμα 27: Τροποποιήσεις στον μετατροπέα από σειριακό σε παράλληλο. Χωρίς να επηρεαστεί η λειτουργικότητα της μονάδας, χρειαζόμαστε μια λιγότερη μνήμη FIFO

Table 6: Sources of the Xilinx Zynq Z-7020 SoC

FPGA chip on Xilinx Zynq Z-7020 SoC	
Logic Cells	**85k**
Look-Up-Tables (LUTs)	**53200**
Flip-Flops	**106400**
DSP	**220**
36Kbit BRAM	**140**

Στον πίνακα 6 παρουσιάζονται οι διαθέσιμοι πόροι που καλούμαστε να αξιοποιήσουμε.

Στην συνέχεια, έχοντας τα αποτελέσματα της εκτέλεσης του Vivado, σχεδιάσαμε θεωρητικά τον πυρήνα του συνελικτικού δικτύου. Για την fixed point αρχιτεκτονική, καταλήξαμε στον παρακάτω πίνακα.

Table 7: Συνολικοί πόροι για τον Fixed Point σχεδιασμό στο ZC-702

	LUTs	**FFs**	**DSP**	**BRAMs**
32×Convolution Units	24400	38400	192	32
31 Adders of Two	496	496	0	0
3×Max-Pool	720	1029	0	6
11×ReLu	110	176	0	0
TOTAL	25726	40101	192	38
TOTAL(%)	48.35	37.68	87.27	27.14

Τοποθετήθηκαν παράλληλα 32 μονάδες συνέλιξης. Με αυτήν την τεχνική, ανά πάσα στιγμή μπορεί να εκτελείται η διαδικασία της συνέλιξης σε 32 διαφορετικές εικόνες εισόδου. Επίσης προσαρμόστηκαν κατάλληλα οι μονάδες Max-Pooling και ReLu λαμβάνοντας υπόψιν τις μέγιστες απαιτήσεις σε αυτές. Στην συνέχεια για να υπολογίσουμε θεωρητικά τον χρόνο εκτέλεσης και τις εικόνες ανα δευτερόλεπτο που μπορεί να επεξεργαστεί η συγκεκριμένη μονάδα, εργαστήκαμε ως εξής : Αρχικά υπολογίσαμε πόσες φορές πρέπει να εκτελεστεί η διαδικασία της συνέλιξης για το εκάστοτε συνελικτικό νευρωνικό δίκτυο. Για παράδειγμα, για το Ship Detection,έχουμε συνολικά $32 \times 3 + 32 \times 16 + 64 \times 16 + 64 \times 32 = 3680$ συνελίξεις εικόνων. Ο χρόνος εκτέλεσης κάθε συνέλιξης εξαρτάται από το μέγεθος της εικόνας εισόδου και συνεπώς από το συνελικτικό επίπεδο του δικτύου στο οποίο βρισκόμαστε καθώς όποτε γίνεται Max-Pooling το μέγεθος της εικόνας υποδιπλασιάζεται. Επομένως έχουμε για την ολοκλήρωση του πρώτου επιπέδου του δικτύου 3 επαναλήψεις, για την ολοκλήρωση του δεύτερου 16 επαναλήψεις, για το τρίτο 32 και για το τέταρτο 64. Τελικά, άμα θεωρήσουμε τον χρόνο που απαιτείται για να ολοκληρωθεί μια πλήρης συνέλιξη εικόνας διαστάσεων 128×128 (αρχική εικόνα) μαζί με το ακόλουθο Max-Pooling T_{128}, τότε έχουμε:

$$Time_{Total} = Time_{Layer_1} + Time_{Layer_2} + Time_{Layer_3} + Time_{Layer_4}$$

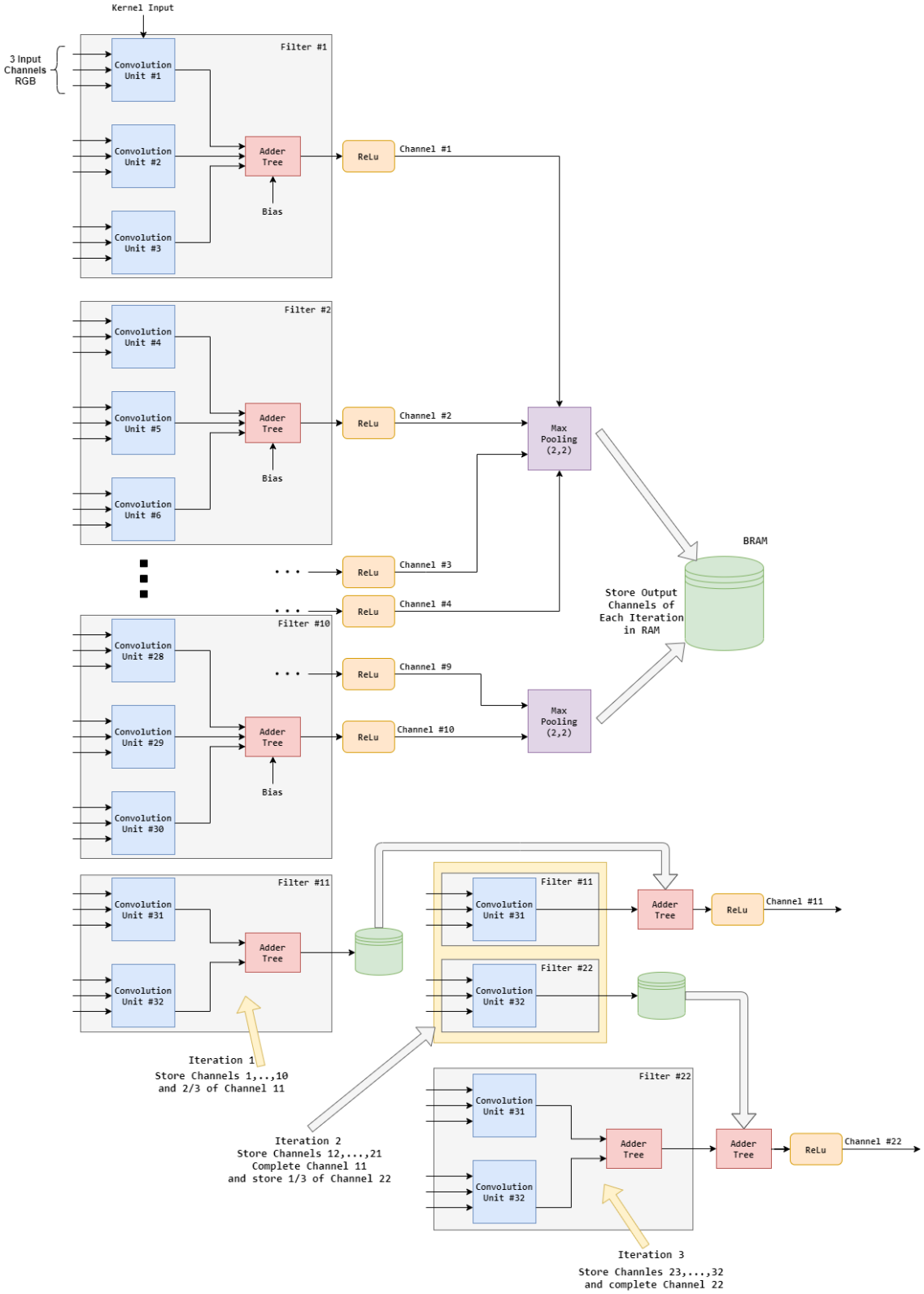
$$Time_{Total} = 3 \times T_{128} + 16 \times T_{64} + 32 \times T_{32} + 64 \times T_{16}$$

$$Time_{Total} = 3 \times T_{128} + 16 \times \frac{T_{128}}{2} + 32 \times \frac{T_{128}}{4} + 64 \times \frac{T_{128}}{8}$$

$$Time_{Total} = 27 \times T_{128} = 171312CCs$$

Για να βρούμε τον ρυθμό επεξεργασίας εικόνων ανά δευτερόλεπτο (FPS-Frame Per Second) τοποθετήσαμε στο Vivado την συγκεκριμένη τοπολογία μονάδων και βρήκαμε ότι το ρολόι λειτουργίας ήταν $\approx 8ns$. Επομένως $Time_{Total} = 171312 \times 8ns = 1.370496ns$.

$$FPS = \frac{1s}{1.370496ms} = 730$$



Σχήμα 28: Ο τρόπος με τον οποίο λειτουργούν οι μονάδες συνέλιξης στο πρώτο συνελικτικό επίπεδο του δικτύου για τον εντοπισμό πλοίων

Με τον ίδιο τρόπο κινηθήκαμε για να κατασκευάσουμε τις τελικές μονάδες επεξεργασίας με την προσέγγιση του αλγορίθμου Winograd καθώς και για άλλους τύπους δεδομένων (Floating Point, Block Floating Point). Στον επόμενο πίνακα φαίνονται οι προτεινόμενες αρχιτεκτονικές μετά την τελική εκτέλεση τους στην πλακέτα.

Table 8: Final CNN performance with proposed techniques (Zynq-7020)

configuration	paral.	**LUT**	**DSP**	**RAMB**	**MHz**	**FPS**
Typical Float.Point	8	37%	91%	78%	125	182
Prop. Block.Fl.Pt.	32	65%	100%	95%	125	730
Prop. B.Fl.P.WGD.	8x4	68%	59%	95%	124	724
Typical Fixed.Point	32	69%	58%	95%	118	689
Prop. Fixed.Point	32	60%	54%	95%	124	724
Prop. Fix.P.WGD.	8x4	41%	58%	95%	112	654

(loss is ∼0.2% between configurations)

Αξίζει να σημειωθεί ότι σε σχέση με την κλασσική υλοποίηση Fixed Point χωρίς προσεγγιστικούς πολλαπλασιασμούς είχαμε μείωση των απαιτούμενων πόρων κατά 13% με αντίστοιχη μείωση σε DSP 7% χωρίς μείωση στην ακρίβεια κατηγοριοποίησης. Όσον αφορά την υλοποίηση Winograd παρατηρήσαμε μείωση των απαιτούμενων πόρων κατά 41% με ακριβώς την ίδια χρήση DSP.

Όσον αφορά την Floating Point αναπαράσταση, χρησιμοποιώντας τις ειδικές Floating Point IP του Vivado καταφέραμε να έχουμε 8 παράλληλες μονάδες στο μέγιστο. Με την χρήση του BFP αυξήθηκαν οι μονάδες από 8 σε 32 δηλαδή τετραπλασιάστηκε η απόδοση της αρχιτεκτονικής. Με την υλοποίηση Winograd καταφέραμε να μειώσουμε κατά 41% τα απαιτούμενα DSP.

Σύγκριση με άλλες συσκευές

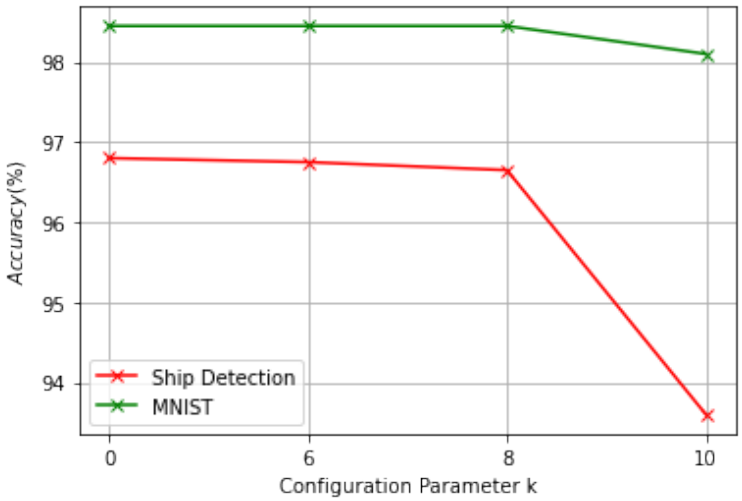
Συγκρινόμενη με άλλες ανταγωνιστικές συσκευές, η προτεινόμενη υλοποίηση πέτυχε 10 φορές καλύτερη απόδοση/Watt από την Jetson Nano Mobile GPU (204 FPS via tensorflow/cuda acceleration) και 2.5 φορές καλύτερη απόδοση/Watt από την Myriad2 DSP (105 FPS via C/C++ coding). Τα πλήρως συνδεδεμένα επίπεδα του δικτύου εκτελέστηκαν από τον επεξεργαστή ARM Cortex-a9 of Zynq και χρειάστηκαν $1.2ms$ για να εκτελεστούν παράλληλα με τις συνελίξεις του δικτύου.

Δοκιμές για τον έλεγχο της ακρίβειας

Για να γίνει επαλήθευση των διάφορων προτεινόμενων αρχιτεκτονικών ως προς την διατήρηση ακρίβειας, έγιναν οι κατάλληλες τροποποιήσεις σε κώδικα C/C++ αντίστοιχης διπλωματικής του Ευάγγελου Πετρόγγονα. Κατασκευάστηκαν οι κατάλληλες συναρτήσεις οι οποίες κάνανε πράξεις ψηφίο προς ψηφίο σε λογισμικό έτσι ώστε να γίνεται προσομοίωση του κώδικα χαμηλού επιπέδου στο υλικό.

Table 9: Δοκιμές ακρίβειας για διάφορες αρχιτεκτονικές Fixed Point

	TensorFlow	**FP 16**	**FP 16 k=6**	**FP 16 k=8**	**FP 16 k=10**
Ship Detection	96.8%	96.8%	96.75%	96.65%	93.6%
MNIST	98.45%	98.45%	98.45%	98.45%	98.1%
CIFAR-10	75.5%	75.5%	75.35%	75.1%	71%



Σχήμα 29: Διακυμάνσεις της ακρίβειας κατηγοριοποίησης σε συνάρτηση με την παράμετρο k του προσεγγιστικού πολλαπλασιαστή

Table 10: Δοκιμές ακρίβειας για διάφορες αρχιτεκτονικές Floating Point

	Float 16	**Float 14**	**Float 12**	**Float 16 Appr.**	**Float 14 Appr.**	**Float 12 Appr.**
Ship Detection	96.8%	96.3%	95.4%	96.8%	96.5%	96.7%
MNIST	98.45%	97.5%	97.15%	98.45%	98.1%	97.2%
CIFAR-10	75.5%	75.45%	70.3%	75.5%	75.4%	70%

Table 11: Δοκιμές ακρίβειας για διάφορες αρχιτεκτονικές Block Floating Point

	10-bit Mantissa	**8-bit Mantissa**	**BFP 10-bit with Approx**
Ship Detection	96.8%	96.7%	96.7%
MNIST	98.45%	98.45%	98.3%
CIFAR-10	75.5%	75.3 %	75.25%

Η μέγιστη πτώση ακρίβειας που παρατηρήθηκε ήταν για το CIFAR-10 όταν η παράμετρος k πήρε την τιμή 10 (4.5%). Όλες οι άλλες προτεινόμενες αρχιτεκτονικές διατήρησαν υψηλή ακρίβεια. Από τις προτεινόμενες αρχιτεκτονικές η μεγαλύτερη πτώση ήταν 0.2% όσον αφορά τις Fixed Point προσεγγίσεις και 1% για 14-bit μήκους Floating Point (5-bit exponent, 6-bit

mantissa).

Chapter 1

Introduction

Deep neural networks (DNNs) have shown significant improvements in many Artificial Intelligence (AI) applications, including computer vision [12], natural language processing [29], speech recognition [1] and machine translation [2]. The performance of DNNs is improving rapidly.

However, this accuracy improvement comes at the cost of high computational complexity. For example AlexNet [17] that is one of the most influential networks published in computer vision takes 1.4GOPS to process a single 224×224 image. Res-Net-152 has managed to increase the accuracy from 84.7%(2012) to 96.5% in 2015 in a cost of 22.6GOPS, more than an order of magnitude more computation. Running Res-Net-152 in a self-driving car with 8 cameras at 1080p 30 FPS (frames per second) requires the hardware to deliver (30FPS×8×1920×1280/(224×224) = 265 Teraop/sec computational throughput. Usign more than one neural networks in each camera will make the computation even larger.

1.1 Machine Learning

Before the functionality of Convolutional Neural Networks (CNNs) is further analyzed, we have to put them in context: CNNs belong to a wide range of algorithms in the field of Machine Learning (ML). ML is already integrated in several aspects of everyday life. E-mail spam filters, voice, text and speech recognition, web search engines, recommendation in lists of music, movies and user preferences to autonomous driving.

Machine Learning is a computational sub-field of Artificial Intelligence. Artificial Intelligence itself poses two main questions: "What is intelligence and how does it work?" and "Can we build intelligent machines?". Correlated with the latter one and as its name suggests, in Machine Learning we try to train computers, in a way that they can learn to solve problems without being explicitly programmed. Using a more formal definition for "learning" in this context : "A computer program is said to learn from experience E, with respect to some task T and some performance measure P, if it's performance on T as measured by P is improved with experience E". At the core of Machine Learning lies

the assumption that knowledge can be derived from data. Based on this assumption, the majority of ML algorithms so far are data-driven in contrast to other AI approaches which may be symbolic,knowledge-based etc. Machine Learning takes steps towards "intelligent" machines, promising a wider range and greater depth of automation in human activities. Both the theoretical work and its technological applications contribute to the material preconditions for a world where monotonous and repetitive tasks will be carried out by machines.

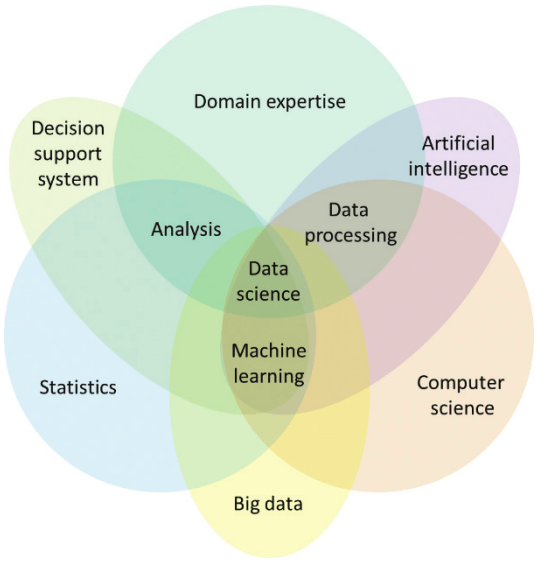


Figure 1.1: Machine Learning relation to other fields

At a high-level, machine learning is simply the study of teaching a computer program or algorithm how to progressively improve upon a set task that it is given. On the research-side of things, machine learning can be viewed through the lens of theoretical and mathematical modeling of how this process works. However, more practically it is the study of how to build applications that exhibit this iterative improvement. There are many ways to frame this idea, but largely there are three major recognized categories: supervised learning, unsupervised learning, and reinforcement learning.

Types of Machine Learning

- **Supervised Learning** describes a class of problem that involves using a model to learn a mapping between input examples and the target variable. Models are fit on training data comprised of inputs and outputs and used to make predictions on test sets where only the inputs are provided and the outputs from the model are compared to the withheld target variables and used to estimate the skill of the model. There are two main types of supervised learning problems: they are classification that involves predicting a class label and regression that involves predicting a numerical value. Both classification and regression problems may have one or more input variables and input variables may be any data type, such as numerical or categorical.

Some machine learning algorithms are described as "supervised" machine learning algorithms as they are designed for supervised machine learning problems. Popular examples include: decision trees, support vector machines, and many more.

- **Unsupervised Learning** describes a class of problems that involves using a model to describe or extract relationships in data. Compared to supervised learning, unsupervised learning operates upon only the input data without outputs or target variables. As such, unsupervised learning does not have a teacher correcting the model, as in the case of supervised learning. There are many types of unsupervised learning, although there are two main problems that are often encountered by a practitioner: they are clustering that involves finding groups in the data and density estimation that involves summarizing the distribution of data. An example of a clustering algorithm is k-Means where k refers to the number of clusters to discover in the data. An example of a density estimation algorithm is Kernel Density Estimation that involves using small groups of closely related data samples to estimate the distribution for new points in the problem space.

The problems we dealt with in this thesis are classification problems, so supervised learning will be further discussed.

1.1.1 Introduction to Artificial Neural Networks

A single neuron also called a perceptron is the basic building block in Artificial Neural Networks. Neurons are the basic computational units and are consisted by three main parts. The input data expressed in numerical form, the activation function and the output data. A neuron receives a number of input signals x_i multiplied by weights w_i. These weighted input data are summed biased with a fixed value b_i and are fed into the activation function Φ that produce the final output.

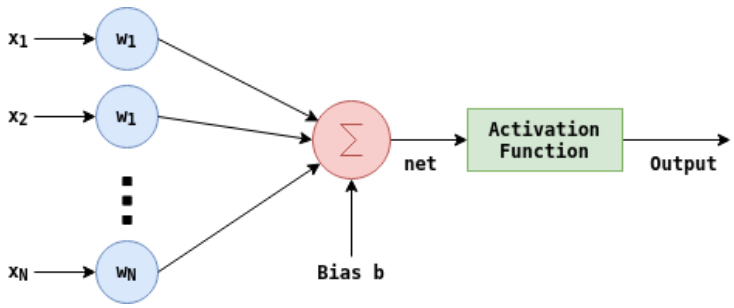


Figure 1.2: Single Neuron Model

$$Net = \sum_{i=1}^{N} x_i * w_i + b$$

$$Output = \Phi(net) = \Phi(\sum_{i=1}^{N} x_i * w_i + b)$$

A single neuron is the basic building block in Artificial Neural Networks. In order to construct a complex Neural Network for a specific task we have to combine multiple neurons with proper weights and activation functions.

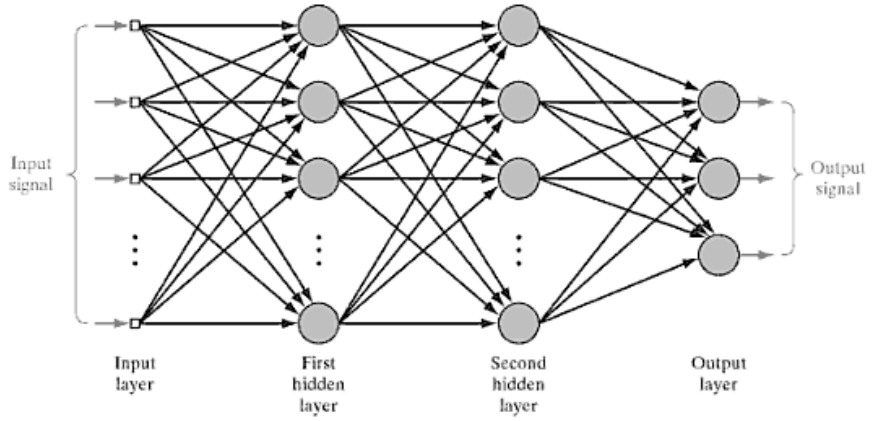


Figure 1.3: Architectural Graph of multilayer perceptron with two hidden layers.

The first step in order to construct a neural network is to train the model using the training data. In order to do so, weights are initialized randomly. Considering training, we should take care of the loss function. Loss function shows how good a neural network is on a specific task. The intuitive way to implement it is to take each training example, pass it though the network, get the output value and subtract it from the actual value that was expected in the output.

$$J(y, \hat{y}) = \frac{1}{m} \sum_{i=1}^{m} (y_i - \hat{y}_i)^2$$

Where i is the index of training example, y stands for the output number we expect from the network and $\hat{y}$ for the number we actually got by passing our example through the network. We want as small number as possible regarding Loss Function. If the Loss Function has a big value, that means our network does not perform very well.

Since initial weights are randomly initialized, we expect a bad performance of the network. In the process of training, we want to start with a bad performing neural network and wind up with a network with high accuracy. In terms of loss function, we should get the minimum value in the end of training. In each iteration, weights are adjusted in order to achieve higher accuracy. The problem of training is equivalent to the problem

of minimizing the loss function. There are a lot of algorithms that optimize functions. These algorithms can be gradient-based or not, in a sense that they are not only using information provided by the function, but they are also use its gradient. One of the most common training algorithms is Stochastic Gradient Descent. Learning rate is an important parameter when training Neural Networks. If learning rate is too small gradient descent method may be really slow. In the other way if learning rate is too large, gradient descent may overshoot the minimum Loss Function value which means it may fail to converge or it could even diverge.

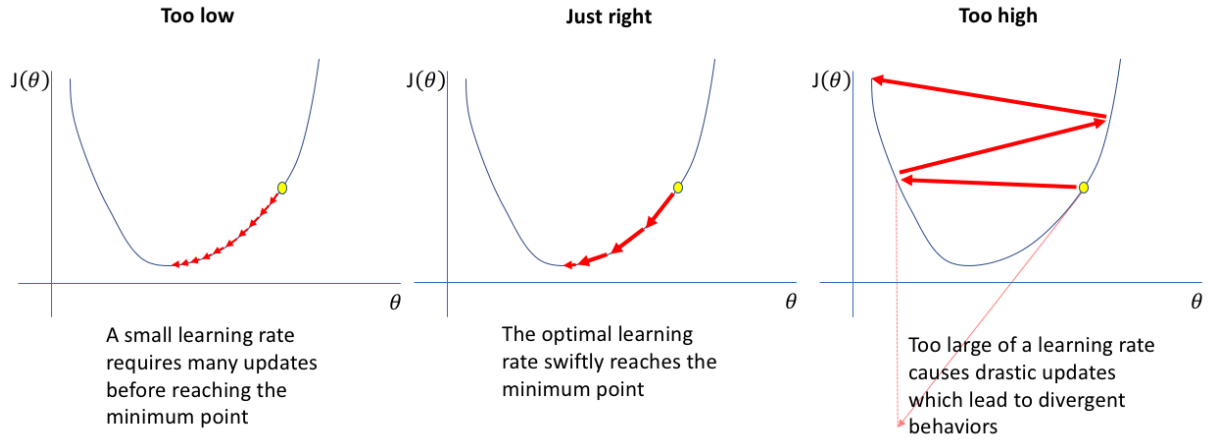


Figure 1.4: Converge to Minimum Cost on Gradient Descent with different Learning Rates

1.1.2 Introduction to Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a special kind of neural networks for processing data that has a known grid-like topology. Data can be thought as 1D time-series data and image data that can be thought as a 2D grid of pixels. As the name indicates these networks employ the mathematical operation called convolution. CNNs are very similar to ordinary Neural Networks. They are made up of neurons that have learnable weights and biases. Each neuron receives some inputs, performs a dot product and optionally follows it with a non-linearity. CNNs are biologically inspired models by research of D.H. Hubel and T.N. Wiesel. They proposed and explanation for the way in which mammals visually perceive the world around then using a layered architecture of neurons in the brain and this in turn inspired engineers to attempt to develop similar pattern recognition mechanisms in computer vision. As an example, let's consider a car. How does a human recognize that is a car? Humans search for the characteristics that are unique to a car like wheels, head-lights, doors, rear trunk, glass windows, hood and other features that differ it from other models of transport. Similarly when recognizing a wheel, we look for circular-shaped objects, comparatively dark colored with a rough texture positioned below the main structure of the car. All these little details are taken into account

to form some basic information. These little information together bunch up to form a particular characteristic that is unique to an object that we are recognizing. "A simple CNN is a sequence of layers, and every layer of a CNN transforms one volume of activation to another through a differentiable function." What it actually means is that, each layer is associated with converting the information from the values, available in the previous layers, into some more complex information and pass on to the next layers for further generalization.

The CNN is a combination of two basic building blocks

- **The Convolution Block**. This block consists of the Convolution Layer and the Pooling Layer. This layer forms the essential component of feature extraction.

- **The Fully Connected Block** . Consists of a fully connected neural network architecture. This layer performs the task of classification based on the input from the convolutional block.

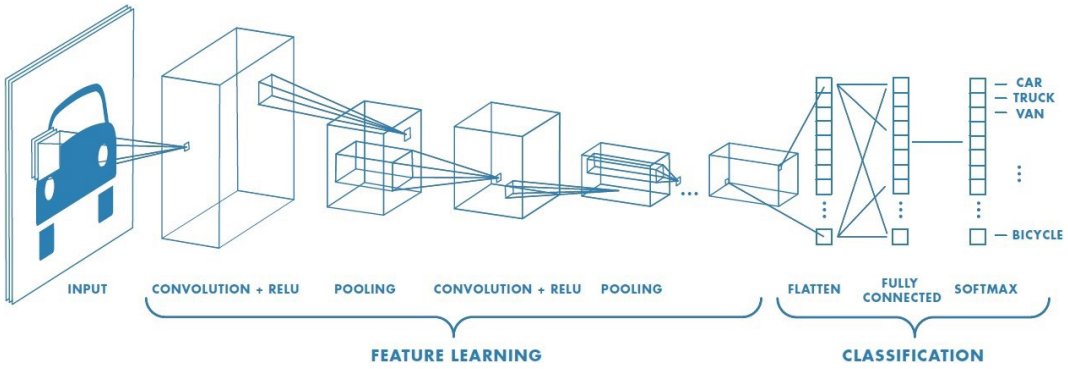


Figure 1.5: Example of a Convolutional Neural Network that categorizes means of transport from an input image

Filters or Kernels are also an image that depict a particular feature for example a curve or a dot. Convolution is a special operation applied on a particular matrix (usually the Image matrix), using another matrix (usually the Filter matrix). Kernel is simply a small matrix of weights. This kernel slides over the 2D input data, performing an elementwise multiplication with the part of the input it is currently on and then summing up the results into a single output pixel.

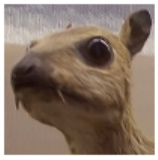
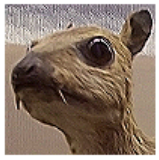
Original	Gaussian Blur	Sharpen	Edge Detection
$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$
			

Figure 1.6: Convolution of an Input Image with classic Computer Vision Kernels.

1.1.3 2D Convolution Operation

When we apply 2D convolution, one tricky issue is that we tend to lose pixels on the perimeter of our image. Since we typically use small kernels compared to image size, we might only lose a few pixels but this may sum up when we apply many convolutional layers connected in a row. One straightforward solution to this problem is to add extra pixels of filler around the boundary of our input image, thus increasing the effective size of the image. Typically we set the extra pixels to 0 called Zero-Padding.

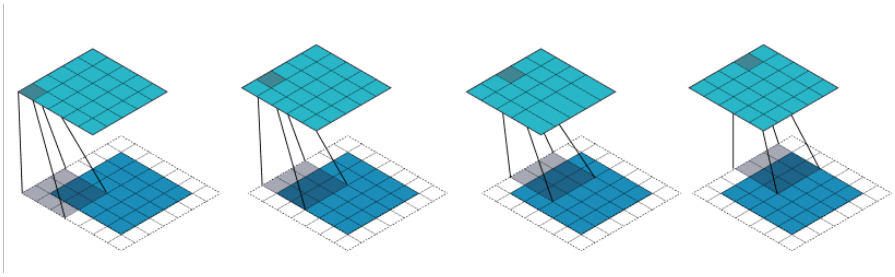


Figure 1.7: 2D convolution using "Same" Padding for 3x3 Kernel and 5x5 Image. Borders of Image are extended by zeros and we have the same size 5x5 output Image.

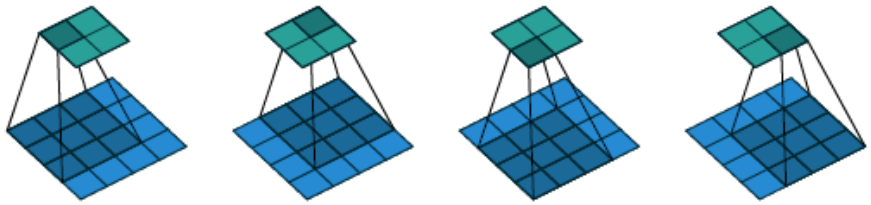


Figure 1.8: 2D convolution without Padding. The output Image has reduced size 2x2 compared to the 4x4 input Image.

As mentioned before when running a convolution layer, we want an output with a lower size than the input. One way to accomplish this is by using a pooling layer. Yet

another way to do it is to use a stride. The idea of stride is to skip some of the slide locations of the kernel. A stride of 1 means to pick slides a pixel apart, so basically every single slide, acting as a standard convolution. A strife of 2 means picking slides 2 pixels apart, skipping every other stride in the process, downsizing by roughly a factor of 2. A stride of 3 means skipping every 2 slides, downsizing by roughly a factor 3 and so on.

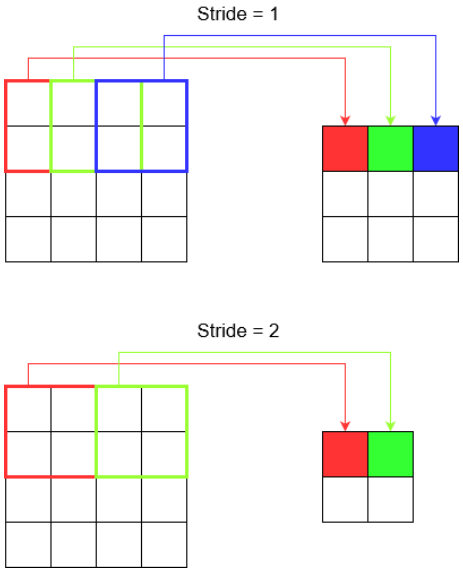


Figure 1.9: Comparing two different types of stride. When stride=1, we have 9 3x3 tiles to be processed, though a 3x3 output. When stride=2, tiles are reduced to 4 and so does the output.

When a 2D input Image x is convolved with a kernel h of size NxM and the output image is y, then to calculate output pixel (i, j) we use the following formula.

$$y(i,j) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} x[i+m, j+n] * h[n,m], \quad i \in ImageWidth, \quad j \in ImageHeight$$

1.1.4 Convolution Layer

When a convolution layer is constructed, each layer takes as input a stack M of 2D Images. The size of this stack is called "Input Feature Maps". Each one of the input feature maps has size of $Image_{Width} \times Image_{Height} = Im_w \times Im_h$. If the number of input feature maps is M, then the input of the layer is a 3D matrix with size $M \times Im_w \times Im_h$. Since the input of each layer is essentially a 3D matrix, we tend to think the neurons of a CNN as being arranged in 3D (width, height, depth).

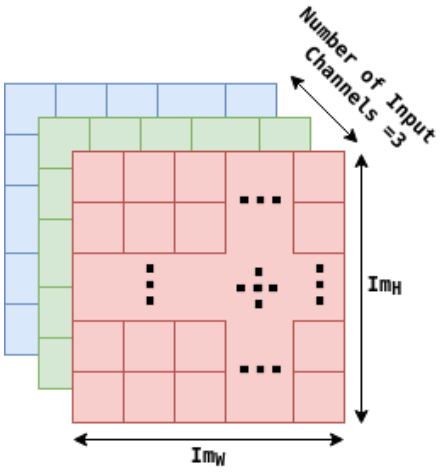


Figure 1.10: Example of an RGB Image. A 3D matrix of size $Im_W \times Im_H \times 3$.

A convolution layer extracts N output feature maps from M input feature maps by convolving each one of the M input maps with N filters. Each one of these N filters has size $M \times Kernel_{Width} \times Kernel_{Height}$. In the majority of applications $Kernel_{Width} = Kernel_{Height}$. So we suppose that the Kernel has dimensions $K \times K$. For each convolution layer with N channels, we have $N \times M \times K \times K$ kernel values (Weights) . The forward pass of each one of the convolutional layers is computed as:

$$\begin{aligned} \forall \quad n = \{1, ..., N\} \quad & \text{Number of Output Feature Maps} \\ \forall \quad i = \{1, ..., Im_h\} \quad & \text{Feature Map Rows} \\ \forall \quad j = \{1, ..., Im_w\} \quad & \text{Feature Map Columns} \\ F[n, i, j] = b[n] + & \sum_{m=1}^{M} \sum_{x=0}^{K=1} \sum_{y=0}^{K-1} \Phi[m, i + x, j + y] \cdot w[n, m, x, y] \end{aligned}$$

Where

- F is a tensor of output feature maps
- b[n] is the bias term applied to each pixel of the output feature map n
- Φ is a tensor of input feature maps
- w is a tensor of filters

The 3D operation can be performed by adding the results of multiple 2D operations. In the equation above, the inner two sums perform the 2D cross-correlation over an input feature map, while the outer-sum realized the 3D operation by adding the results of all the M input feature maps at each kernel location. The size of the output feature maps is dependent to Padding and Stride as it was mentioned before. Thus the size of the output map can be calculated with the following equation:

$$\text{Output Map Size} = H = \frac{Im_w - K + 2 \cdot Padding}{Stride} + 1$$

The size of the output feature map. if we suggest that the input image has size $Im_w \times Im_w$ is $H \times H$. This is also the number of times that the 2D kernel fits inside the 2D input feature map. The 2D convolution operation, if we suggest that we have every pixel of the Image available in memory, can be computed in parallel. The computation of each output pixel is independent from any other. All the M Input Maps and all the N output maps can be calculated simultaneously. Although the number of multiplication that need to be performed in a convolutional layer is significant big. For example for 3 input feature maps and 16 output channels (16 filters) and a kernel of size 3×3, $H \times H \times 9 \times 3 \times 16 = 432 \times H^2$ multiplications need to be calculated. For an input image of size 128×128, that leads to 7077888 multiplications for only 3 Input Feature Maps. These multiplications can be calculated in parallel but the hardware resources we need in order to calculate them in parallel do not allow it.

1.1.5 Pooling Layer

Convolutional layers in a convolutional network systematically apply learned filters to input images in order to create feature maps that summarize the presence of those features in the input. Stacking convolutional layers in deep models allows layers close to the input to learn low-level features (e.g. lines) and layers deeper in the model to learn high-order or more abstract features, like shapes or specific objects. A limitation of the feature map output is that they record the precise position of features in the input. This means that small movements in the position of the feature in the input image will result in a different feature map. This can happen with re-cropping, rotation, shifting and other minor changes to the input image.

A common approach to addressing this problem from signal processing is called down sampling. This is where a lower resolution version of an input signal is created that still contains the large or important structural elements, without the fine detail that may not be as useful to the task. Down sampling can be achieved with convolutional layers by changing the stride of the convolution across the image. A more robust and common approach is to use a pooling layer. A pooling layer is added after the convolutional layer. The addition of a pooling layer after the convolutional layer is a common pattern used for ordering layers within a convolutional neural network that may be repeated one or more times in a given model. Pooling involves selecting a pooling operation, much like a filter to be applied to feature maps. The size of the pooling operation or filter is smaller than the size of the feature map; specifically, it is almost always 2×2 pixels applied with a stride of two pixels. This means that the pooling layer will always reduce the size of each feature map by a factor of 2, e.g. each dimension is halved, reducing the number of pixels or values in each feature map to one quarter of the size. For example, a pooling layer applied to a feature map of 6×6 (36 pixels) will result in an output pooled feature map of 3×3 (9 pixels). Two common functions used in the pooling operation are :

- **Average Pooling** Calculate the average value of each patch on the feature map

- **Maximum Pooling** Calculate the maximum value of each patch of the feature map

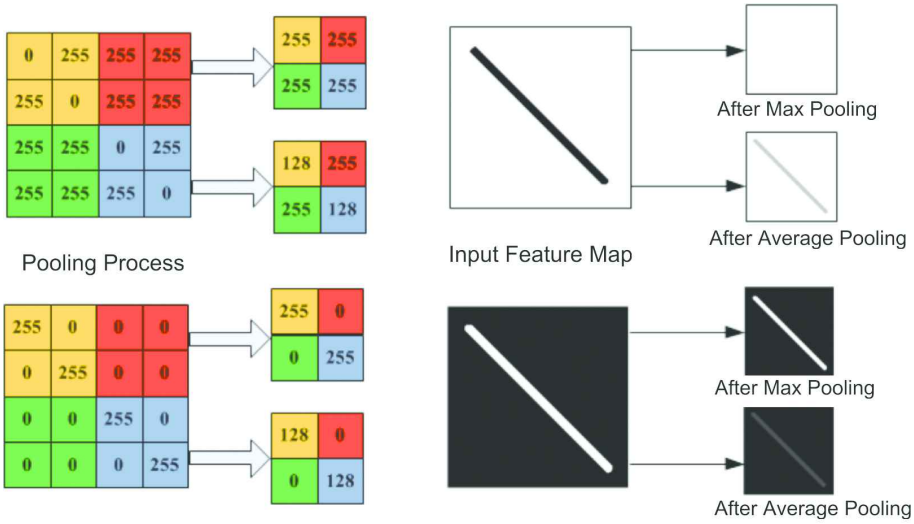


Figure 1.11: The two different common methods of pooling and their impact on the input feature map

The result of using a pooling layer and creating down sampled or pooled feature maps is a summarized version of the features detected in the input. They are useful as small changes in the location of the feature in the input detected by the convolutional layer will result in a pooled feature map with the feature in the same location. This capability added by pooling is called the model's invariance to local translation.

The results of max pooling are down sampled or pooled feature maps that highlight the most present feature in the patch, not the average presence of the feature in the case of average pooling. This has been found to work better in practice than average pooling for computer vision tasks like image classification.

1.1.6 Activation Function

The activation function is a node that is put at the end of or in between Neural Networks. They help to decide if the neuron would fire or not. If the input value of a neuron exceeds a threshold then the activation function fires this neuron otherwise it disables it.

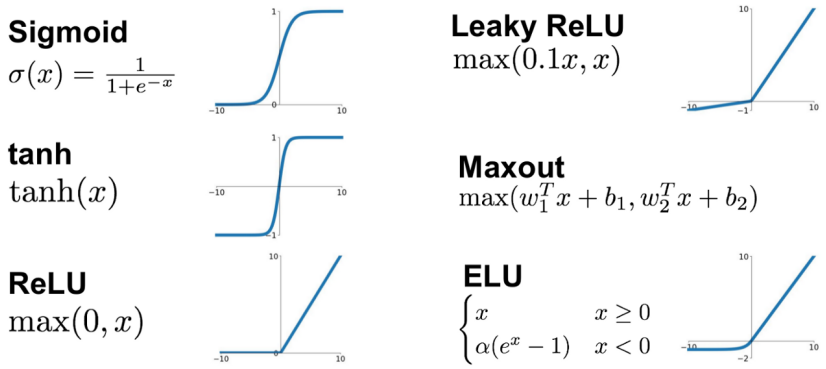


Figure 1.12: Most Popular Activation Functions in Neural Networks

ReLu function is the most widely used activation function in neural networks today. One of the greatest advantage *ReLu* has over other activation functions is that it does not activate all neurons at the same time. From the image for *ReLu* function above, we will notice that it converts all negative inputs to zero and the neuron does not get activated. This makes it very computational efficient as few neurons are activated per time. It does not saturate at the positive region. In practice, *ReLu* converges six times faster than *tanh* and *sigmoid* activation functions. Some disadvantage *ReLu* presents is that it is saturated at the negative region, meaning that the gradient at that region is zero. With the gradient equal to zero, during backpropagation all the weights will not be updated, to fix this, we use Leaky *ReLu*. Also, *ReLu* functions are not zero-centered. This means that for it to get to its optimal point, it will have to use a zig-zag path which may be longer.

1.1.7 Fully Connected Layer in CNN

The objective of a fully connected layer is to take the results of the convolution/pooling process and use them to classify the image into a label (in a simple classification example). The output of convolution/pooling is flattened into a single vector of values, each representing a probability that a certain feature belongs to a label. For example if the image is of a cat, features representing things like whiskers or fur should have high probabilities of the label "cat". The figure below illustrates how the input values flow into the first layer of neurons. They are multiplied by weights and pass through an activation function just like in a classic artificial neural network. Then they pass forward to the output layer, in which every neuron represent a classification label.

The fully connected part of the CNN network goes through its own backpropagation process to determine the most accurate weights. Each neuron receives weights that prioritize the most appropriate label. Finally, the neurons "vote" on each of the labels and the winner of that vote is the classification decision.

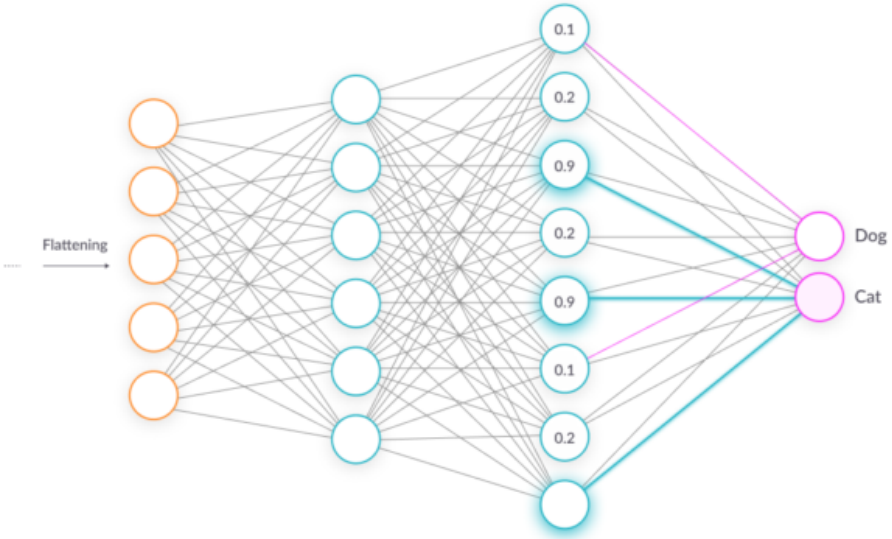


Figure 1.13: The Fully Connected Part of a CNN to classify Dog and Cat Images

1.2 Field Programmable Gate Arrays

FPGA stands for field-programmable gate array. At its core, an FPGA is an array of inerconnected digital subcircuits that implement common function while also offering very high levels of flexibility. An FPGA is an array of logic gates, and this array can be programmed (actually configured) in the field, i.e., by the user of the device as opposed to the people who designed it. However, an FPGA is not a vast collection of individual Boolean gates. This would be a very suboptimal way to provide configurable-logic functionality because it would not take advantage of the fact that common operations can be implemented much more efficiently as fixed modules. The same principle is evident in the world of discrete digital ICs (Integrated Circuits). We can buy ICs that consist of AND, OR gates and so forth- but we would not want to build a shift register out of individual gates. Instead, we would buy a shift register IC. An FPGA, then, is much more than an array of gates. It is an array of carefully designed and interconnected digital subcircuits that efficiently implement common function while also offering very high levels of flexibility. The digital subcircuits are called Configurable Logic Blocks (CLBs), and they form the core of the FPGA's programmable-logic capabilities.

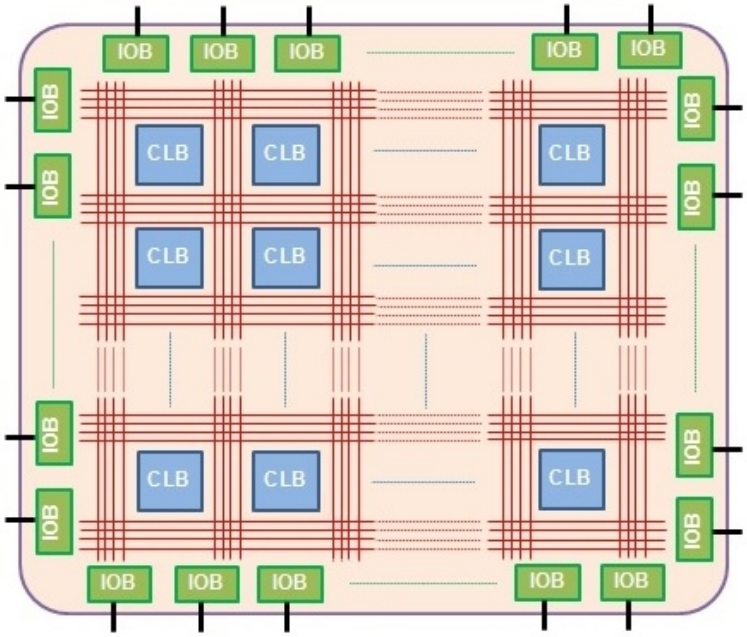


Figure 1.14: FPGA Board

The CLBs need to interact with one another and with external circuitry. For those purposes, the FPGA uses a matrix of programmable interconnects and input/output (I/O) blocks. The FPGA's "program" is stored in SRAM cells that influence the functionality of the CLBs and control the switches that establish the connection pathways. CLB's internal structure and operation is complicated. The general idea is that CLBs include look-up tables (LUTs), storage elements (flip-flops or registers) and multiplexers that allow the CLB to perform Boolean, data-storage and arithmetic operations. An I/O block consists of various components that facilitate communication between the CLBs and other components on the board. These include pull-up/pull-down resistors, buffers and inverters.

1.2.1 FPGA Programming

In order to program an FPGA for a specific task, we have to turn an array of CLBs into a digital circuit that does precisely what we want. It seems like a rather complicated task. However, FPGA development does not require thorough knowledge of CLB functionality or painstaking arrangement of internal interconnects, just as microcontroller development does not require thorough knowledge of a processor's assembly-language instructions or internal control signals. Actually, it is somewhat misleading to present an FPGA as a standalone component. FPGAs are always supported by development software that carries out the complicated process of converting a hardware design into the programming bits that determine the behavior of interconnects and CLBs. The question now, is how do we "explain" to the software what the FPGA hardware needs to do. For this purpose Hardware Description Languages (HDL) are created, that allow us to "describe" hardware. The most common are VHDL and Verilog. Despite the apparent similarity between HDL

code and code written in a high-level software programming language, the two are fundamentally different. Software code specifies a sequence of operations, whereas HDL code is more like a schematic that uses text to introduce components and create interconnections. In earlier FPGAs, there was no processor to run any software; hence implementing an application implied designing the circuit from scratch. So, we could have configured and FPGA to be as simple as an OR gate or as complex as a multi-core processor. We can have a long way since and the basic FPGA architecture has developed through the addition of more specialized programmable function blocks like ALUs, block RAM, multiplexers, DSP-48 and microprocessors.

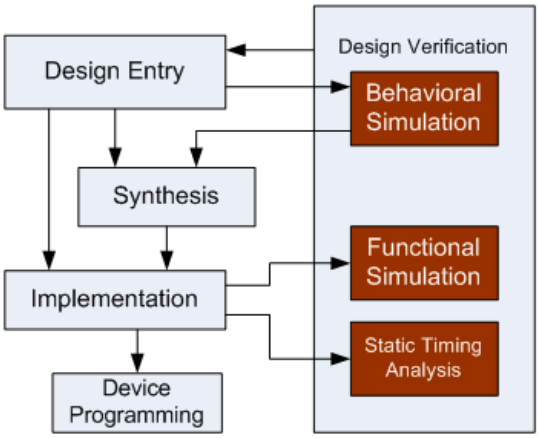


Figure 1.15: FPGA Programming-Mapping

FPGA architectural design flow comprises design entry, logic synthesis, design implementation, device programming and design verification. However, the exact steps vary with manufactures.

Design Entry

The description of the logic can be made using either a schematic editor, a finite state machine (FSM) editor, or a hardware descriptio language (HDL). This is done by selecting components from a given library and providing a direct mapping of the design functions to selected computing blocks. When designs with a very large amount of function become difficult to manage graphically, HDL may be used to capture the design either in a structural or in a behavioral way. Besides VHDL and Verilog, which are the most established HDLs, several C-like languages are also available such as Handel-C, Impulse C, and SystemC.

Logic Synthesis

This process translates the above VHDL code into a device netlist format for depicting a complete circuit with logical elements. Synthesis involves checking the code syntax and analyzing the hierarchy of the design architecture. Next, the code is compiled along with optimization and the generated netlist is saved as a **.ngc** file.

Design Implementation
The design implementation consists of the following steps:

- **Translate** : This process combines all the input netlists into the logic design file which is saves as a **.ngd** file. Here user constraint file assigns the ports to the physical elements.
- **Map** : This involves mapping the logic defined by the **.ngd** file into the components of FPGA and then generating a **.ncd** file.
- **Place and Route** : Here routing places the sub-blocks from the above process into the logic blocks according to the constraints and then connect these blocks.

The above mentioned routed design must be loaded and converted into a format supported by the FPGA. Hence, the routed .ncd file is given to the BitGen program, which generates a bitstream file that contains all the programming information for an FPGA.

Design Verification
This is done all along with the design flow for ensuring that the logic behaves as intended. The following simulations are involved in this process:

- Behavioral Simulation (RTL Simulation)
- Functional Simulation
- Static Time Simulation

These simulations are done in order to emulate the behavior of the components by providing test patterns to the inputs of the design and observing the outputs. In our specific case of 2D convolution, the outputs of Behavioral Simulation, where directly compared to the outputs of MATLAB or Python codes performing the exact same operations in software.

1.2.2 Advantages of FPGAs

FPGA enables us to program product features, adapt to new standards and reconfigure hardware for specific applications even after the product has been installed in the field-hence the term "field-programmable". All modern personal computers including desktops, notebooks, smartphones and tablets, are examples of general-purpose computers. General-Purpose computing incorporates "Von Neumann" approach, which states that an incstrution fetch and a data operation cannot occur simultaneously. Therefore, being sequential machines, their performance is also limited.

On the other hand, we have the Application Specific Integrated Circuits (ASICs) which are customized for a particural task like a digital voice recorded or a high-efficiency Bitcoin miner. An ASIC uses a spatial approach to implement only one application and provides maximum performance. However, it can not be used for tasks other than those for which it has been originally designed.

Figure 1.16: General Purpose Processors compared to ASICs

FPGAs are less energy efficient when compared to ASICs and also not suitable for large volume predictions. However they are reprogrammable and have low NRE(Non-Recurring Engineering) costs when compared to an ASIC.

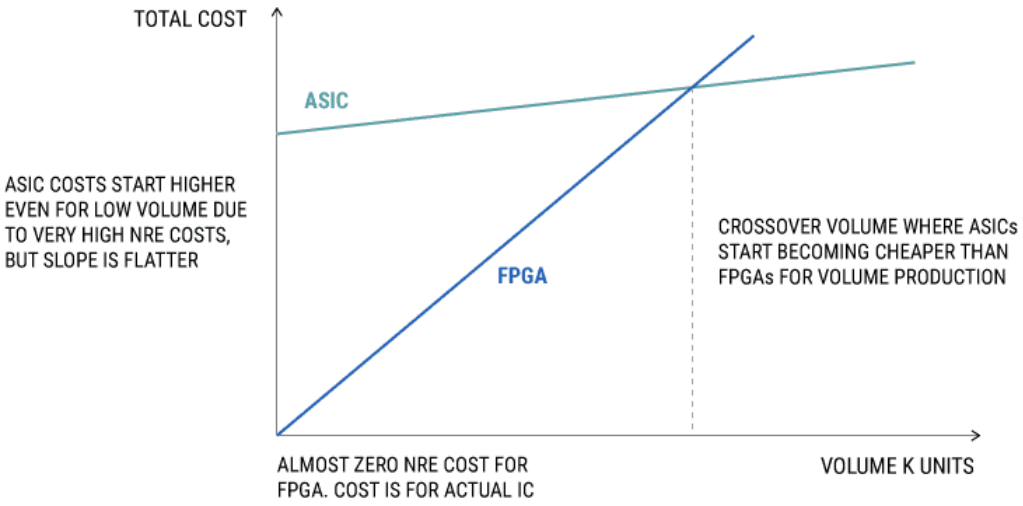


Figure 1.17: FPGA vs ASIC Cost Analysis

FPGAs used to be chosen for lower speed and complex designs in the past, but nowadays FPGAs can easily surpass the 500 MHz performance benchmark.

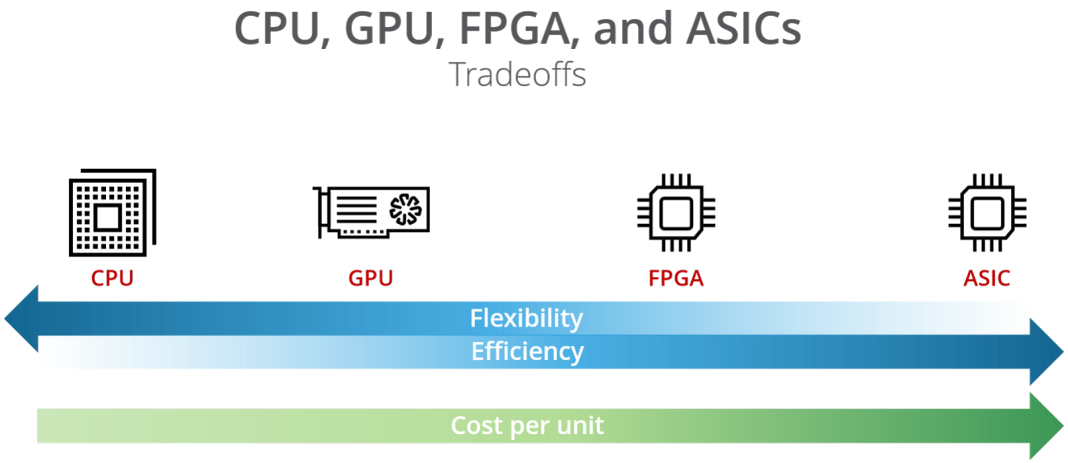


Figure 1.18: Tradeoffs between different data accelerators

FPGA vs GPU

It is clear that the application and also the project goal are very important to chose the right Hardware platform. FPGAs have shown stronger potential over GPUs for the new generation of machine learning algorithms where DNN comes to play massively. The main winning points of FPGAs over GPUs would be the flexibility by FPGAs to play with different data types - such as binary, ternary and even custom ones - as well as the power efficiency and adaptability to irregural parallelism of sparse DNN algorithms. However, the challenge for FPGA vendors is to provide ans easy-to-use platform. Building any type of advanced FPGA designs such as for machine learning require advanced FPGA design and verification tools. Simulation is the de-facto verification methodology for verifying FPGA designs using mixed-language HDL with SystemC/C/C+ testbenches.

	CPU	GPU	FPGA	ASIC
Compute Adaptability (to a variety of situations)	High	Medium	Low	None
Compute power	Medium	High	High	Medium
Latency	Medium	High	Low	Ultra low
Throughput	Low	High	High	High
Parallelism	Low	High	High	High
Power efficiency	Medium	Low	Medium	High

Figure 1.19: Summary of different processors and their tradeoffs

1.3 Approximate Computing

1.3.1 Introduction

The pervasive, portable, embedded and mobile nature of present age computing systems has led to an increasing demand for ultra low power consumption, small footprint and high performance. Approximate computing is nascent computing paradigm that allow us to achieve these objectives by compromising the arithmetic accuracy. Many systems used in domains, like multimedia, neural networks and big data analysis, exhibit an inherent tolerance to a certain level of inaccuracies in computation and thus can benefit from approximate computing. The computational and storage demands of modern systems have far exceeded the available resources. It is expected that, in the coming decade, the amount of information managed by worldwide data centers will grow 50-fold, while the number of processors will increase only tenfold. In fact,the electricity consumption of just the US data centers is expected to increase from 61 billion kWh (kilowatt hour) in 2006 [Mittal 2014a] and 91 billion kWh in 2013 to 140 billion kWh in 2020 [NRDC 2013]. It is clear that rising performance demands will soon outpace the growth in resource budgets; hence, over provisioning of resources alone will not solve the conundrum that awaits the computing industry in the near future. Functional approximation, in hardware, mostly deals with the design of approximate arithmetic units, such as adders and multipliers, at different abstraction levels, i.e. transistor, gate, RTL and application. Some notable approximate adders include speculative adders [42], segmented adders[43] and approximate full adders. Also, in the field of approximate multipliers, i.e., the most power-hungry component of hardware, accelerators, significant research has been conducted [23, 7, 33, 39, 20].

A promising solution for this dilemma is approximate computing (AC), which is based on the intuitive observation that, while performing exact computation or maintaining peak-level service demand require a high amount of resources, allowing selective approximation or occasional violation of the specification can provide disproportionate gains in efficiency. For example, for a k-means clustering algorithm, up to 50×energy saving can be achieved by allowing a classification accuracy loss of 5 percent [5]. Similarly, a neural approximation approach can accelerate an inverse kinematics application by up to 26×compared to the GPU execution, while incurring an error of less than 5 percent [8].

Approximate computing and storage approach leverages the presence of error-tolerant code regions in applications and perceptual limitations of users to intelligently trade off implementation, storage, and/or result accuracy for performance or energy gains. In brief, AC exploits the gap between the level of accuracy required by the applications/users and that provided by the computing system, for achieving di-verse optimizations. Thus, AC has the potential to benefit a wide range of applications/frameworks, for example, data analytics, scientific computing, multimedia and signal processing, machine learning and MapReduce, and so forth. The term AC spans a wide set of research activities ranging from programming languages [11] to transistor level [10].

1.3.2 Approximate Arithmetic Circuits

Approximate Adders

In approximate implementations, multiple-bit adders are divided into two modules: the (accurate) upper part of more significant bits and the (approximate) lower part of less significant bits. For each lower bit, a single-bit approximate adder implements a modified, thus inexact function of the addition. This is often accomplished by simplifying a full adder design at the circuit level, equivalent to a process that alters some entries in the truth table of a full adder at the functional level.

Approximate Multipliers

While the design of approximate adders has received a lot of attention, the design of approximate multipliers has not. Approximate multipliers that use the speculative adders to compute sum of partial products have been designed in [36],[13]. However, the straightforward application of approximate adders in a multiplier may be inefficient in terms of of trading off accuracy for savings in energy and area. A key design aspect in an approximate multiplier, is to reduce the critical path of adding the partial products. Multiplication is usually implemented by a cascaded array of adders. In [31] and [16] some less significant bits in the partial products are omitted (using some error compensation mechanisms) and thus some adders can be removed from the cascaded array for a faster operation. In [18], a large multiplier is constructed by 2×2 simplified multipliers to reduce arithmetic and computation complexity. An efficient design that uses input pre-processing and additional error compensation is proposed in [27] to reduce the critical path delay. Combinations of both partial product generation and approximations are applied in collaboration to further reduce power consumption [14], [41], [28]. The main goal in nowadays research considering approximate multipliers is to reduce the number of partial products using hybrid radix encoding [22] to apply approximations on the partial product generation.

1.4 Related Work

In this section we will briefly review previous research on CNNs and FPGA based accelerators.

1.4.1 CNN on FPGA

A proposed engine that accelerated CNNs with a combination of Winograd and GEMM (general element-wise matrix multiplication) has been proposed in [15]. A software high-level programming model, that overcomes the difficulties of low level hardware programming and accelerate a CNN from TensorFlow has been proposed in [32] and [40]. Moreover, library-based frameworks based on TensorFlow have been proposed, which automatically

generate high-throughput CNN inference engines for FPGAs [34] and ASICs [21]. There have also been proposals to reduce the precision and bit width. Angel-Eye [9], a programmable and flexible CNN accelerator architecture reduce the bit-width down to 8-bit with negligible accuracy loss. Modifications on the algorithm that the convolution is computed (Depthwise Separable Convolution) achieved significant speedup in [3]. A tool for performance modeling for CNN Inference Accelerators on FPGA [30] was designed in order to determine the designs' bounds in the early design phase, before the FPGA implementation. In [38] fast convolution units (FCUs) are designed using fast finite impulse response algorithm (FFA) increasing both resources efficiency and throughput.

1.4.2 Approximations on CNNs on FPGAs

In [24] a Block Floating Point accelerator that significantly reduced memory bandwidth was proposed. A compression technique for YOLO CNN [35] that retrains and quantizes the network using binary weight and flexible low-bit activation achieved 2.63% lower mAP (mean average precision) that the same network with full precision. Architectural innovations to reduce the number of neural network parameters, such as replacing fully-connected layers with convolutional layers or global average pooling, have been used in network in Network [26] and GoogleNet [37] and achieved remarkable results. HashedNets [4] used a hash function to randomly group connection weights into has tables to reduce bit width.

Chapter 2

VHDL Core Design

This chapter presents the architecture of the Engine we designed, that implements the convolution unit of a given CNN. The steps and different approaches that led to the final design are further analyzed.

2.1 Direct Design Approach

Convolutional Neural Networks are inherently massively parallel algorithms. For example, if we examine the first convolutional layer of a given network, we can see that it consists of 3 input channels one for each color (RGB) and 32 Filters. So we expect 32 output channels as an output of our first layer. The computation of each channel can be considered as a completely independent procedure from the others. A design that implements a convolution layer should utilize the potential of computing each channel in parallel. In addition there is a potential to pipeline parts of the design. From our low-level design perspective our design can be optimized in order to achieve the maximum parallel channels computation, maximum pipeline between different layers and optimizations in logic in order to decrease critical paths and enable a higher clock frequency.

The main purpose is to fully utilize a given device (FPGA or ASIC), in order to perform computations. The low-level design in VHDL is fully generic and can be reconfigured according to the network requirements. The final engine constructed, can be used as an accelerator for every type and structure of CNNs, respecting the target device's resources available.

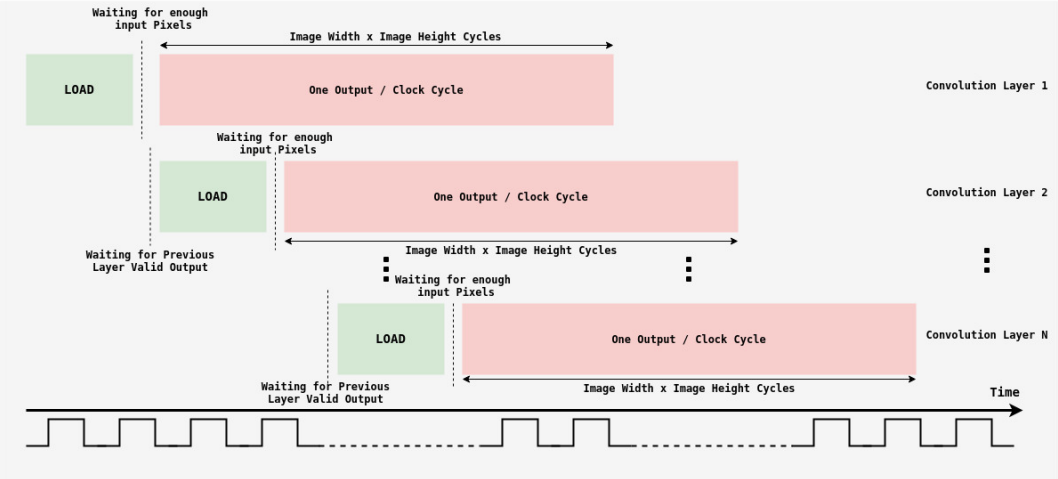


Figure 2.1: Data Pipeline Between different Layers of a CNN. Once Data are loaded and the first valid outputs are generated, they can be directly feeded to another layer

2.2 Components Required

Our Convolution Engine Design requires components that implement the following operations:

- Store the incoming input pixels that are obtained serially in a proper way in FIFO queues (RAM), in order to perform a serial to parallel conversion (S2P).
- Keep track of the data flow and raise the appropriate signals, in order to apply padding when necessary (in image's bounds) and continue operations (Control Unit).
- Pass the data in parallel to the Convolution Unit and perform the appropriate multiplications and accumulations between the input image and the kernel.
- Raise valid_out signals in order to inform any next Layers or other components that the output pixel is ready.

2.2.1 Data Flow

We consider an image with size Image-Width×Image-Height and a Kernel with size Kernel-Width×Kernel-Height. The Data flows into the component pixel by pixel on each line. So, in one clock cycle we get a new pixel as input and we store the previous one in the memory. To achieve maximum throughput, the following system was implemented in order to perform the first computation between the image and the kernel : $\lfloor Kernel_{Height}/2 \rfloor \times (Image_{Width}) + \lceil Kernel_{Width}/2 \rceil$[1] pixels should be loaded in the memory.

[1] $\lfloor\rfloor : Floor, \lceil\rceil : Ceil$

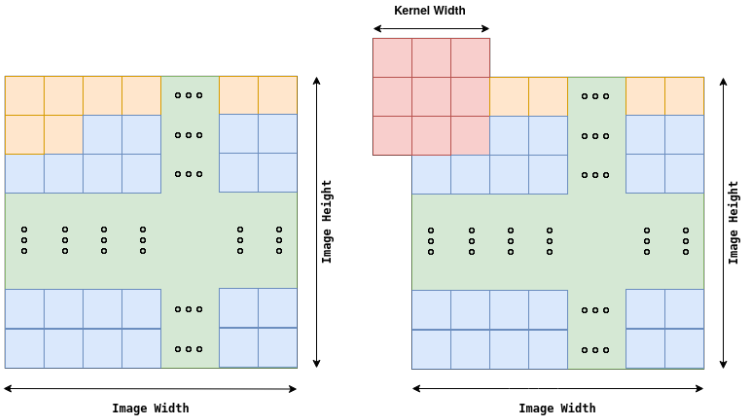


Figure 2.2: Orange pixels should be loaded in memory in order to begin computations with 3x3 Kernel

For the input data to be properly stored, the following memory system was designed. We generated Kernel-Height in number FIFO queues of Image-Width×Sizeof(Pixel) size each one. These FIFO queues are connected with each other in a chain as shown in the next figure.

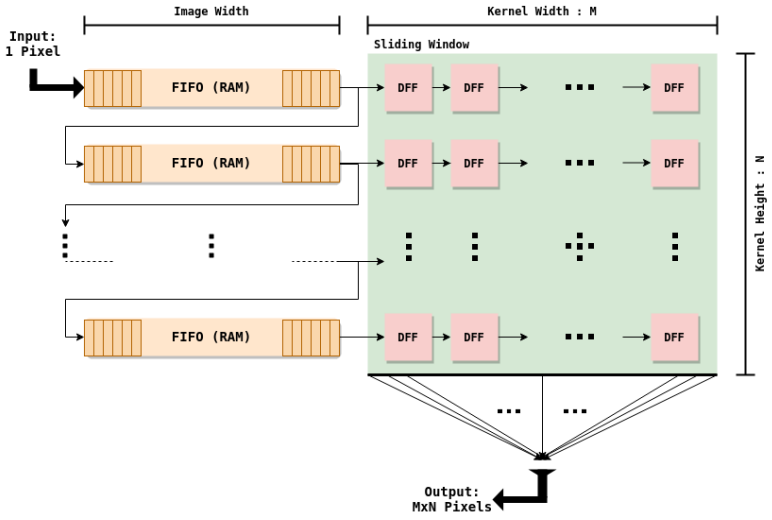


Figure 2.3: Generic Serial To Parallel Converter. Input Pixels are given as input to the first FIFO queue. Register Window has size $M \times N$ depended on the input kernel

Once the memories are set in the above way, we should take care of their inner communication using their control signals. Each memory in the chain must communicate with the next and the previous one to store the data.

Write Enable signal of the first FIFO is high when the first input valid data appears. When FIFO#1 stores all the first row of the input image, it enables Read Enable and Write Enable for the next FIFO in chain. Same happens for FIFO$^{(i-1)}$ when communicating with FIFO^{i}. In order to correctly design our RTL we took care of the bounds of the image. In a two-dimension convolution we should consider the following restrictions:

- **Zero-Pad First Lines of the Image**
  Assuming that we have stored enough pixels in the memory, we can start computations and give valid outputs. The first limitation we should take care of is Zero-Padding on the first lines of the Image. If $Lines_{Loaded}$ are less than $Kernel_{Height}$ then the remaining $Kernel_{Height} - Lines_{Loaded}$ lines must be filled with Zeros. (or any other type of Padding e.g. Mirror-Padding) . These zeros are produced in the Registers with appropriate control signals generated by the Control Unit in order to reduce Data Transfer from Memory. When $Lines_{Loaded} = Kernel_{Height}$ there are enough input pixels to perform convolution and we don't need to Zero-Pad the first lines any longer.

- **Zero-Pad Last Lines of the Image**
  In the same way we Zero-Pad the first lines, we should handle the last lines of the Image. If $Lines_{Loaded}$ are more than $Image_{Height} - \lfloor Kernel_{Height}/2 \rfloor$, in exactly the same way, control signals are generated to Zero-Pad the next
  $Image_{Height} - Lines_{Loaded}$ lines.

- **Zero-Pad Right and Left Limits of the Image**
  While computing an output pixel that is placed either on the left or on the right bound of the image a sequence of Zeros should be produced starting from one Zero to final $\lfloor Kernel_{Width}/2 \rfloor$ Zeros. These signals are produced by the Control Unit and Zeros are generated into the Registers.

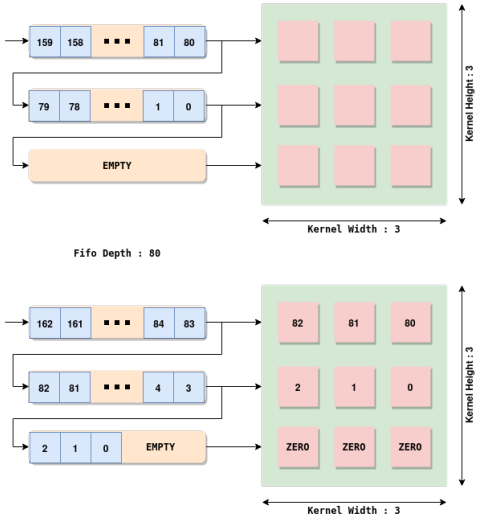


Figure 2.4: Zero Padding Example for the first row of an image

The above Memory System implements a Serial to Parallel Converter. In each Clock Cycle there are $(Kernel_{Height}) \times (Kernel_{Width})$ pixels ready for processing. As mentioned above in order to compute the first correct output we should wait for $\lfloor Kernel_{Height}/2 \rfloor \times (Image_{Width}) + \lceil Kernel_{Width}/2 \rceil$ Clock Cycles. Each Clock Cycle after this one, data are well-formatted and ready to be processed in the convolution unit.

2.2.2 Convolution Unit

This is the unit in which computations take place. Kernel in appropriate size is given as input to the component. We set $Kernel_{Width} := M, Kernel_{Height} := N$. In this component M×N multiplications are performed for each output element. Then these products are given as input to a M×N input Tree-Adder to accumulate the final sum.

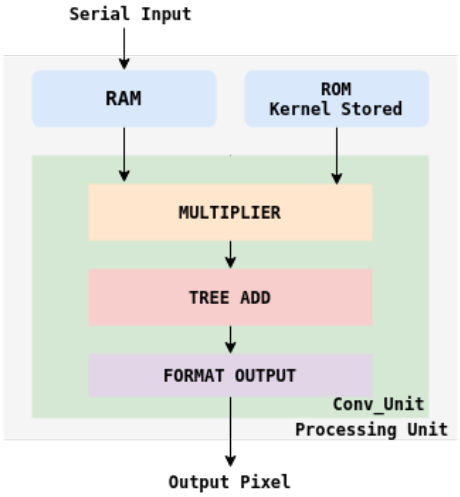


Figure 2.5: Baseline Processing Unit Component structure

The Processing Unit contains a RAM memory (S2P) constructed as mentioned before, the Kernel stored in ROM and the Convolution Unit that performs the computations. The previous components can be combined in order to construct a layer of the CNN on the FPGA. Every channel can be fully parallelized, since there are no dependencies between the layers. In order to achieve this, we must combine the Processing Units constructed in such a way so as to get the maximum utilization from each layer.

We consider M Input Channels and N Filters and we expect N different Output Feature Maps. In order to construct this we have to generate our Processing Unit M×N times. Every channel and filter component is fully synchronized with each other. In each output channel pixels from each input feature map can be directly added by using a M input Tree-Adder. After the computation of this sum, Bias of the current channel is added and the final result is filtered by ReLu or any other activation function. The same happens in each of the N output channels. In this way the outputs is synchronous generated pixel by pixel and is forwarded to the next Layer of the network.

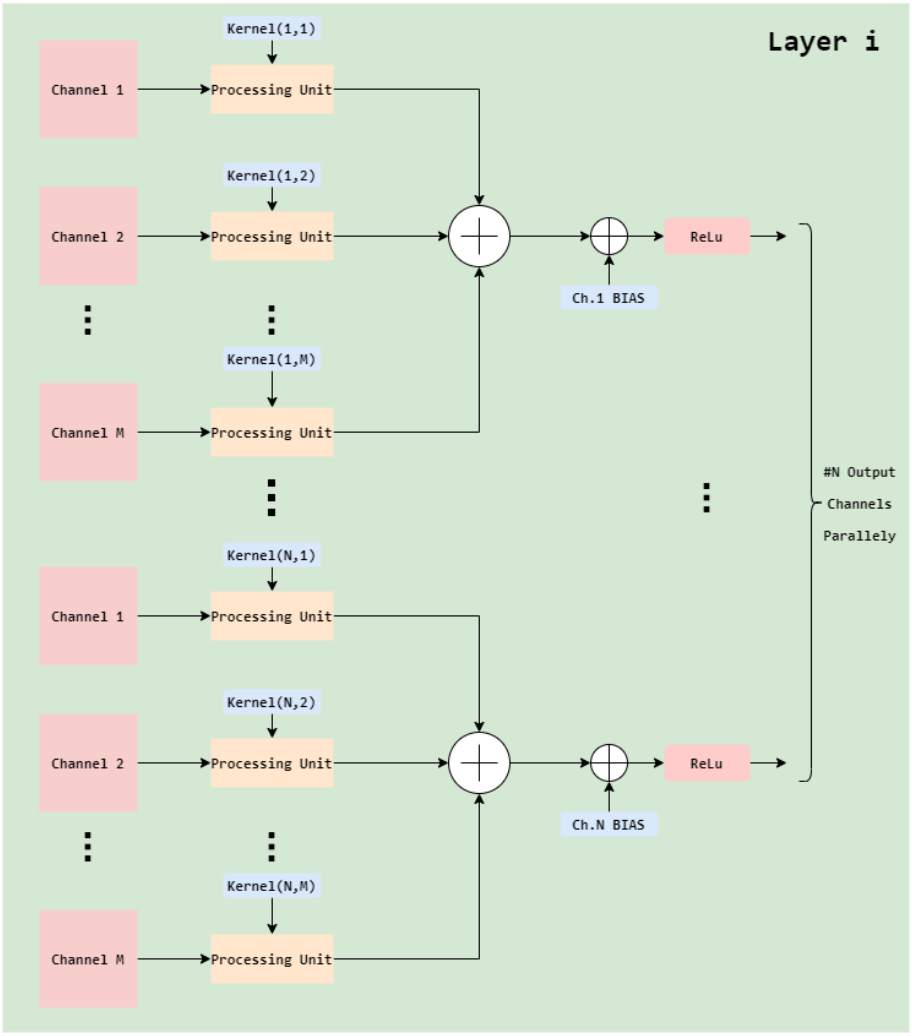


Figure 2.6: CNN Layer Construction Using Processing Units

2.3 Winograd Design Approach

Since FPGA resources are limited there is a need to maximize utilization and maximum Throughput. Conventional FFT based convolution is fast for large filters, but state of the art convolutional neural networks use small, 3×3 or 5×5 filters to extract features. Winograd's [19] minimal filtering algorithms for computing m outputs with an $r-$tap FIR filter, which we call $F(m,r)$, requires

$$\mu(F(m,r)) = m + r - 1$$

multiplications. Also minimal 1D algorithms $F(m,r)$ and $F(n,s)$ can be nested to form minimal 2D algorithms for computing $m \times n$ outputs with an $r \times s$ filter, which is called $F(m \times n, r \times s)$.

$$\mu(F(m \times n, r \times s)) = \mu(F(m,r))\mu(F(n,s)) = (m + r - 1)(n + s - 1)$$

In other words, to compute $F(m,r)$ we must access an interval of $m + r - 1$ data values, and to compute $F(m \times n, r \times s)$ we must access a tile of $(m + r - 1) \times (n + s - 1)$ data values. Therefore the minimal filtering algorithm requires one multiplication per input.

2.3.1 Winograd Implementation for 3×3 Kernel

The standard algorithm for $F(2,3)$, uses $2{\times}3 = 6$ multiplications. Winograd documented the following minimal algorithm:

$$F(2,3) = \begin{bmatrix} d_0 & d_1 & d_2 \\ d_1 & d_2 & d_3 \end{bmatrix} \begin{bmatrix} g_0 \\ g_1 \\ g_2 \end{bmatrix} = \begin{bmatrix} m_1 + m_2 + m_3 \\ m_2 - m_3 - m_4 \end{bmatrix}$$

where

$$m_1 = (d_0 - d_2)g_0 \quad m_2 = (d_1 + d_2)\frac{(g_0 + g_2) + g_1}{2}$$
$$m_4 = (d_1 - d_3)g_2 \quad m_3 = (d_2 - d_1)\frac{(g_0 + g_2) - g_1}{2}$$

The algorithm just uses 4 multiplication and is therefore minimal by the formula $\mu(F(2,3)) = 2 + 3 - 1 = 4$. It also uses 4 additions involving the data d, 3 additions and 2 multiplications by a constant involving the filter (the sum $g_0 + g_2$ in parenthesis can be computed once), and 4 additions to reduce the products to the final result. Fast filtering algorithms can be written in matrix form as :

$$Y = A^T[(Gg) \odot (B^T d)]$$

Where $\odot$ indicates element-wise multiplication. For $F(2,3)$, the matrices are:

$$B^T = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \quad G = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix} \quad A^T = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix}$$

$$g = \begin{bmatrix} g_0 & g_1 & g_2 \end{bmatrix}^T \quad d = \begin{bmatrix} d_0 & d_1 & d_2 & d_3 \end{bmatrix}^T$$

A minimal 1D algorithm $F(m, r)$ is nested with itself to obtain a minimal 2D algorithm, $F(m \times m, r \times r)$

$$Y = A^T \Big[[GgG^T] \odot [B^T dB] \Big] A$$

As proven in Lavin-Gray Fast Convolution [19]
Where now g is an $r \times r$ filter and d is an $(m + r - 1) \times (m + r - 1)$ image tile.
$F(2 \times 2, 3 \times 3)$ uses $4 \times 4 = 16$ multiplications to calculate 4 output products, whereas the standard algorithm we presented before uses $4 \times 3 \times 3 = 36$. So there is an arithmetic complexity reduction of $\frac{36}{16} = 2.25$. Now we will analyze the extra cost required in comparison with the standard convolution algorithm. In order to transform the input data in the correct Winograd form $B^T dB$ 32 simple operations are needed. (additions - subtractions). The Kernel Transform needs 28 floating point instructions that can be adjusted to a fixed point notation in our FPGA implementation and the inverse transformation uses 24 simple operations.

In order to design the Winograd Algorithm in FPGA for 3×3 Kernel we made the following changes in our initial standard convolution design.

- First of all, instead of 3×3 tiles with stride 1 that are used in the standard design now we have to work with 4×4 tiles with stride 2. In the standard design there were $P = (Image_{Width}) \times (Image_{Height})$ tiles per channel. Now every tile has 2 elements that overlap their neighboring ones resulting in :
  $P' = (Image_{Width}/2) \times (Image_{height}/2)$ tiles per channel.
  Instead of 3 FIFO queues that were used in the previous design now we have to use 4 in order to fit the data required to initiate computations. Another, register in the sliding Register Window that follows the FIFO chain, must be added in order to have the 4×4 tile available.

- Data sliding from serial to parallel including all the appropriate limitations are implemented in exact the same way as the standard design taking care of the Winograd peculiarities.

- Input Transform Component : Gets as input a 4×4 stride, performs appropriate additions-subtractions and gives an output of the proper form to continue the computations.

- Filter Transform Component : Gets as input the 3×3 Kernel and gives as output the 4×4 transformed Kernel.
- Output Transform Component : Transforms the output from 4×4 to 2×2 with the appropriate additions.

In order to compare the tradeoff between standard and Winograd Design we should take into consideration extra operations needed to properly transform data. As mentioned above in this case the input tiles must have a stride of two both in width and in height. Since data are obtained serially, in order to design a serial to parallel converter with stride two in width, we have to wait for one clock cycle to obtain the next valid tile. It is obvious that in these clock cycles, except from sliding data in the memory, another operation that gives an output can not be pipelined since there are not valid data available. In the following section the problem of Winograd component under utilization is further discussed and a solution is proposed.

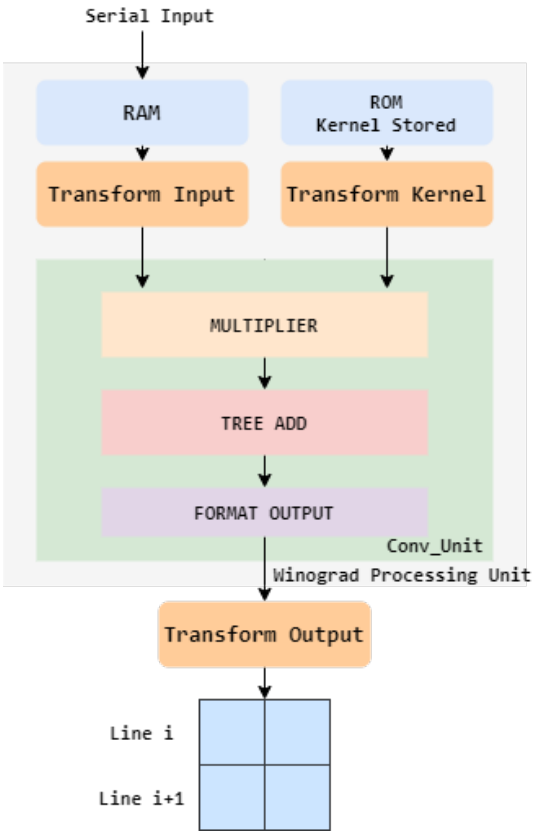


Figure 2.7: Winograd Convolution Unit Structure

2.3.2 Winograd Engine Utilization Techniques

Our dominant concern regarding the optimal design of the Winograd component was to tackle the under-utilization of the Design. Since Stride = 2, while converting the input data from serial to parallel, there exists dead time between two different strides as shown in the next figure.

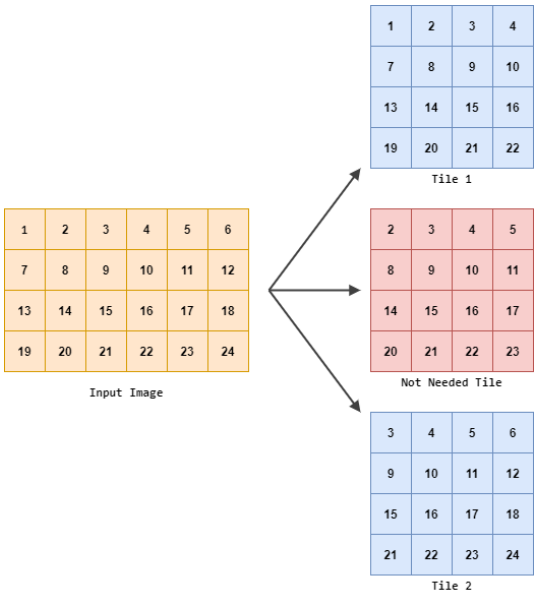


Figure 2.8: Between two valid Winograd tiles there exists an unnecessary one that we can not make use of

As for the horizontal tile striding, there exists one clock cycle dead time between two useful 4×4 tiles due to the serially obtained data. The same phenomenon is apparent in the vertical dimension, since we must skip a whole Image Line ($Image_{Width}$ pixels).

Figure 2.9: Winograd Component Under Utilization for 1 Input Channel. Red squares indicate invalid tiles while the blue the valid tiles.

It is obvious that the component is underutilized, by a factor of 1/4. In order to fully utilize the RTL design and get the maximum throughput, 4 different input channels should be processed simultaneously. For this purpose four different RAMs with the same design logic as before must be used. If we consider four different input channels A, B, C and D, then all channels must be loaded at the beginning of the loading time period. As it has been analyzed before, $3 \times (Image_{Width}) + 3$ Clock Cycles are needed to properly start the convolution operation. After the loading operation, the first two channels A and B are

processed consecutively for each Clock Cycle.

To achieve this operation, once enough data is stored in the memory (data are parallely given to memory for each input channel), by using control signals we feed one tile from channel A at the first clock cycle and one tile from channel B at the exact following clock cycle, while the data in the memory of both channels keeps loading. The generation of $Image_{Width}$ sums from each channel shows that the first row of both channels is correctly processed. As it is depicted in Figure 2.9, Image Width dead clock cycles are required in order to load the next line of both channels A and B. In order to utilize this period of time, we repeat the exact same operation as before for channels C and D. Channels C and D are properly loaded during the loading time, so as to be readily provided as input to the convolution unit. While channels C and D are processed, input data keep flowing in for each of them, as well as for channels A and B. Once Image Width clock cycles have passed while operating with channels C and D, $Image_{Width}$ outputs for both of them are generated and the necessary input pixels for each of channels A and B are loaded and we can perform the next two lines computations with them. We keep repeating the same component management until all four input channels are processed. Working in this way, there are no dead clock cycles during the whole Winograd convolution procedure.

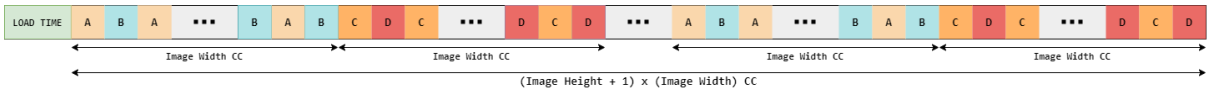


Figure 2.10: Winograd Component Full Utilization for 4 Input Channels

In order to fully describe a convolutional layer using the Winograd architecture, the following system must be designed. As described above, to fully utilize a Winograd Component we have to feed it with 4 different input channels, otherwise it will be underutilized. As it is seen in Figure 2.6, in the baseline architecture output pixels of each input channel are popping out synchronized in each Clock Cycle. Output Pixels $O(i, j)$ of each input channel have to be all added to the bias to form the right output pixel $Out(i, j)$. Considering the Winograd design, the main problem that occurs is that 4×4 output tiles A,B,C and D of each Input Channel do not pop out of the component in the same clock cycle.

In order to format the correct output map, if we suggest that we are working with 4 input channels A,B,C and D, these 4 channels should be added together and then transformed to form the correct 2×2 output tile. After the Load Time, the first 4×4 output tile of channel A is given as an output. In the exact following Clock Cycle the first output tile of channel B is given as an output. In the following Clock Cycles, the $2^{nd}, 3^{rd}, ..., (Width/2)$ output tile of each of channels A and B are given as outputs consecutively. To properly form the output map we have to add the i^{th} tile of each channel together. We store the i^{th} tile of channel A in a register and in the next clock cycle we add the i^{th} tile of channel B to it. Then we transform this 4×4 sum, we store it in a FIFO memory and we repeat the same procedure for the same row between A and B. After Image Width CC have passed we have formed W/2 4×4 tiles and transformed them

to their 2×2 equivalent. These 2×2 tiles are sequentially stored in the FIFO memory until we have stored W/2 tiles that correspond to the first row of the Output Map. To fully form the output channel, channels C and D must be added to the previous sum. So once channels C and D give valid outputs, we repeat the same additions between them, we transform them from 4×4 to 2×2 and then we start getting outputs from the FIFO queue and add them together. Once they are all added, we receive a valid 2×2 tile which we can pass to the main memory. The same pattern is repeated for the following CCs, until the FIFO queue its empty. Then we repeat the FIFO load with the next row of channels A and B for Image Width CC before we give valid outputs again.

To process M input channels with the Winograd component, (m) mod (4) should equal to 0. M/4 Winograd engines should be generated in parallel. To format correct output map as described above, M channels must be added together. We generalize the above pattern with M channels. In each clock cycle, for the first W Image Width CC the output tiles of M/4 Engines should be added together, transformed and stored in memory. Once (Image Width/2) tiles are stored in memory, then we add in the exact same way the (Image Width/2) tiles remaining from the M/2 remaining channels. In Figure 2.5 an example of 8 input channels is shown.

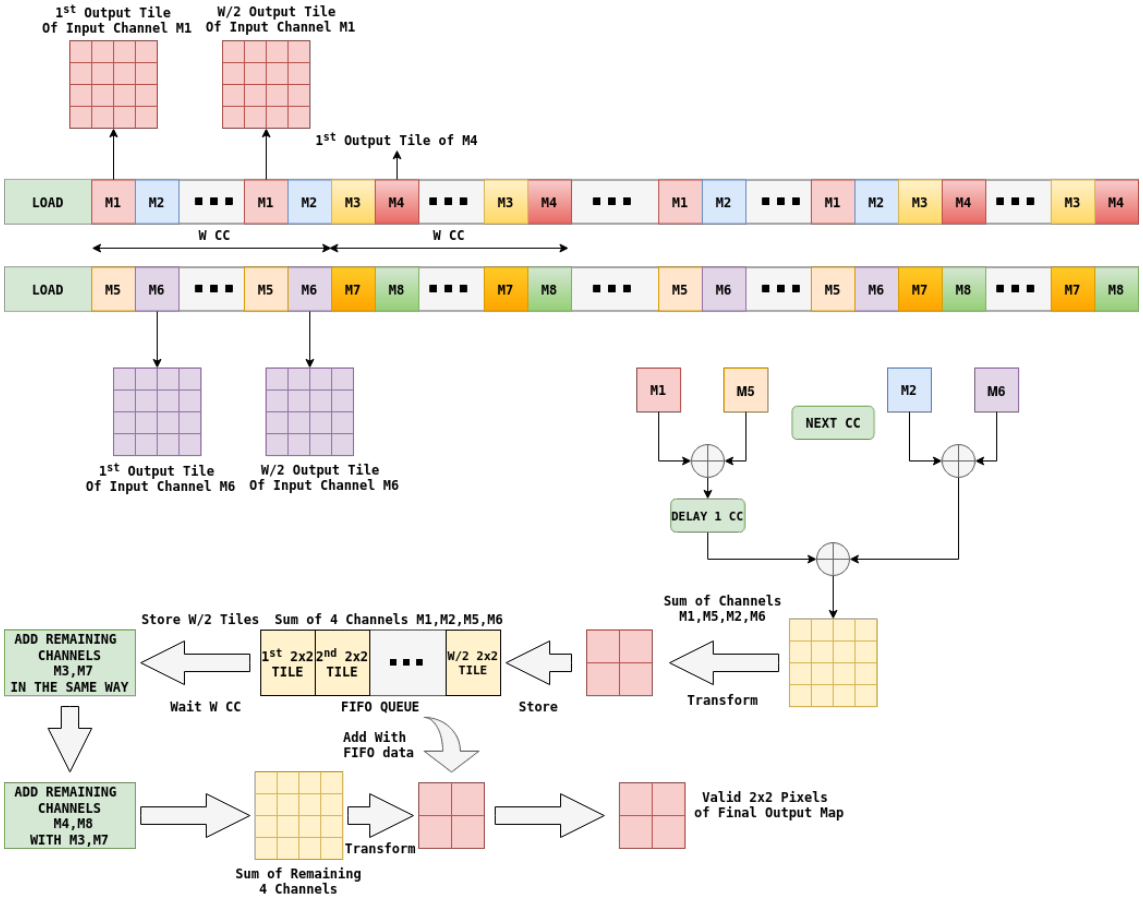


Figure 2.11: One Winograd Output Channel For eight Input Channels

FIFO Queue should have capacity of $(W/2) \times (2 \times 2) \times (SizeOfData)$. Considering one output channel, in order to properly have the correct output, we must wait for $2 \times ImageWidth + 3$ CC to load data. After these CCs we should wait for Image Width CCs to store all the added tiles from M/2 channels in FIFO memory. During this period we don't have a valid output since data from remaining channels aren't yet produced. After Image Width CCs have passed since the generation of the first tile of Channel M1 or M5, the first tiles of M3 and M8 will be produced. So the first valid output will be generated after LoadTime+ImageWidth+[AdditionsCycles] CCs. For the next ImageWidth CCs ImageWidth/2 in number 2×2 tiles will be generated. These #ImageWidth/2 tiles will consist the first two rows of the output map Image of the output channel. After ImageWidth CC the first two rows will be completed and then we have to wait again for ImageWidth CC to perform the same operations between the new tiles of the M input channels. Concluding, in order to correctly design Winograd for the CNN, we have Image Width CC that we generate 4 pixels/2CC = 2 pixels/CC and Image Width CC that FIFO loading is happening and we don't generate valid outputs. We have a total average throughput of 1 pixel/CC if we consider the intermediate loading Clock Cycles, as in the Baseline architecture.

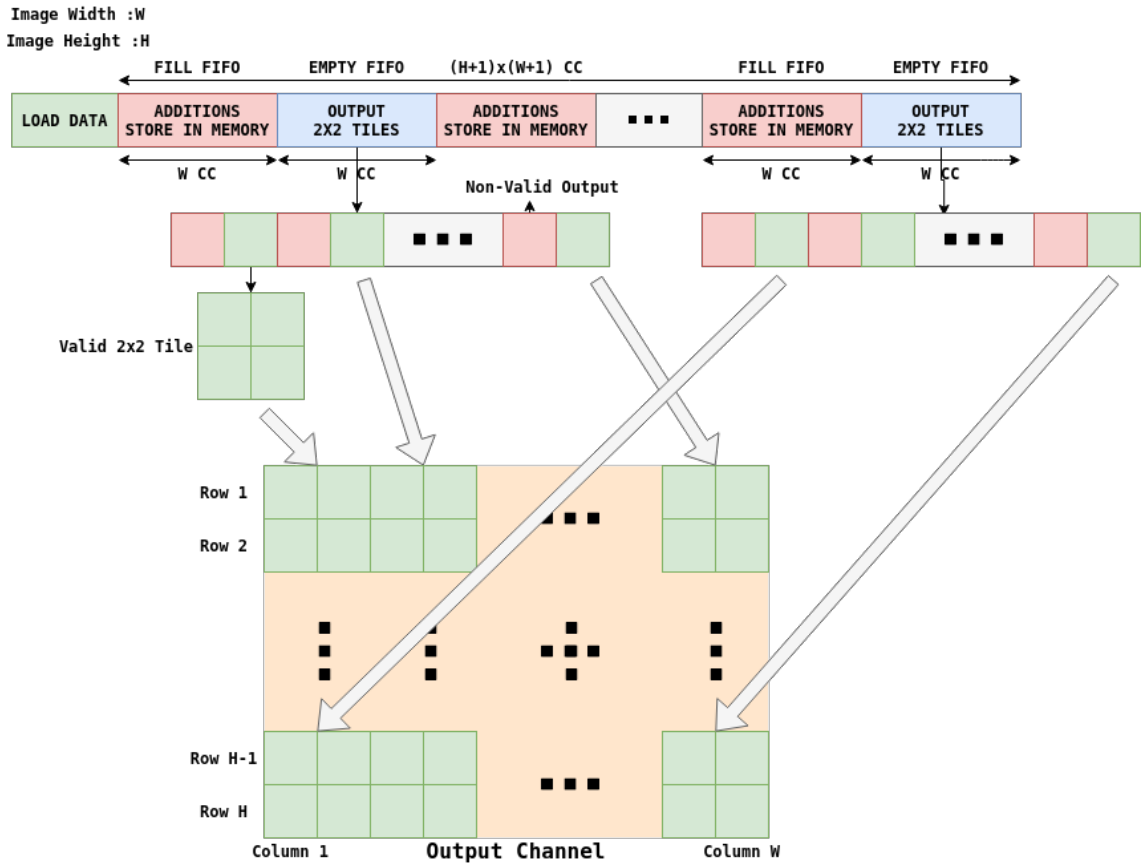


Figure 2.12: Output Map Creation Using Winograd

In order to properly design a convolutional Layer, components that perform the activation function operation and max-pooling were created. ReLu is one of the most common activation functions used in state of the art neural networks. Considering max pooling (2,2), since we have stride 2 both in horizontal and vertical dimension, we had to implement exact the same method as we did with Winograd fully utilization. So, in order the max pooling component to be fully utilized it has to work with 4 input maps at once to fully utilize the component. We noticed, that in the standard convolution design implementation in order to perform max-pool since data are obtained one by one in the output channel, a memory structure to store enough pixels to perform the operation had to be implemented.

On the other side, in the Winograd design since for each channel we get 2×2 output tiles, these tiles can be directly max pooled and give the maximum pixel out of 4 as output without using a memory structure. This can be really important when we have to max pool multiple channels, since for each channel in the standard design to perform a max pool(2,2) operation, we need a RAM memory consisted of 2 FIFO queues, as implemented before. For N output channels $2 \times N$ memory units must be generated. Since we try to use the least possible memory resources out of the device the Winograd approach can noticeably reduce memory units that are used for max pooling. *ReLu* activation function is just checking the sign bit of each pixel and if the pixel is negative, it's output is set to zero, otherwise the output is the exact same as the input. In the following figures one output Layer computation is described.

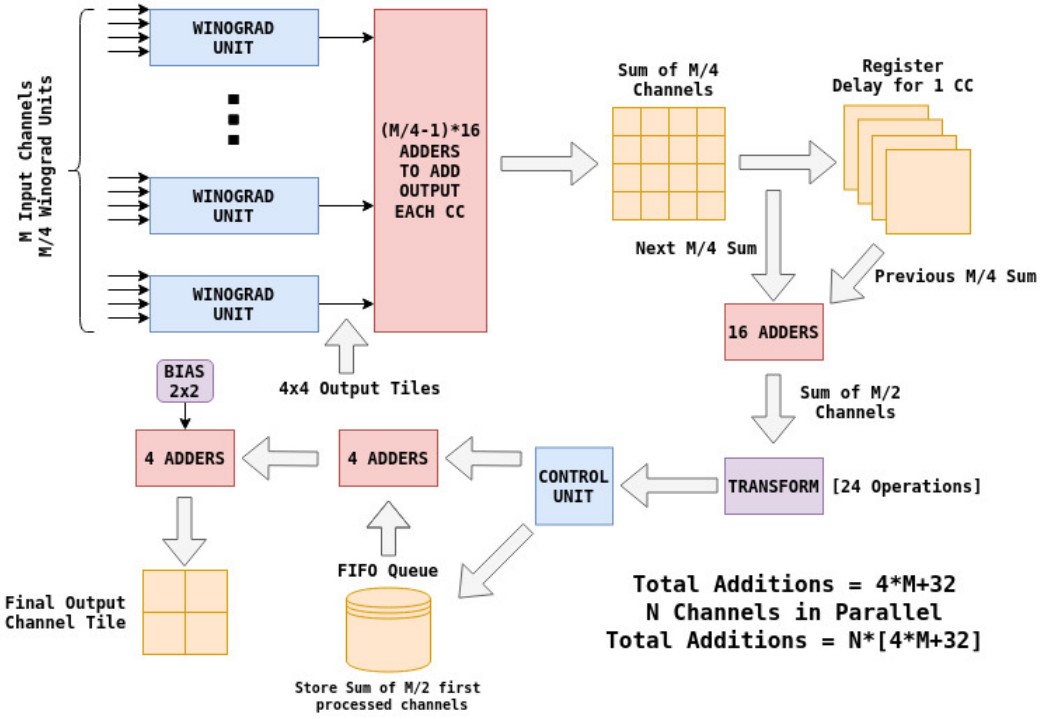


Figure 2.13: Winograd Management For M Input Channels

Table 2.1: Direct and Winograd Processing Units ($w \times w$ Image, 3×3 Kernel)

	Direct	**Winograd**	**Full Winograd**
Input Channels	1	1	4
RAM FIFOs (S2P)	3	4	16
DFFs (S2P)	9	16	64
ROMs	1	1	4
Multipliers	9	16	16
Adders	8	56	56
Latency	w+2 CCs	2w+3 CCs	2w+3 CCs
Throughput	1pixel/CC	1pixel/CC	4pixels/CC

Table 2.2: Direct and Winograd Convolutional Layers ($W \times W$ Image, M channels, 3×3 Kernel, N filters)

	Direct	**Full Winograd**
Units	$N \cdot M$	$N \cdot M/4$
- RAM FIFOs (S2P)	$N \cdot 3M$	$N \cdot 4M$
- DFFs (S2P)	$N \cdot 9M$	$N \cdot 4M$
- ROMs	$N \cdot M$	$N \cdot M$
- Multipliers	$N \cdot 9M$	$N \cdot 4M$
- Adders	$N \cdot 8M$	$N \cdot 8M$ – $N \cdot 15M$[1]
RAMs	0	N FIFO RAM
Adders	$N \cdot (M-1)$	$N \cdot (4M+32)$
ReLu	N	$4N$
Latency	w+2 CCs	2w+3 CCs
Throughput	Npixels/CC	Npixels/CC

[1] $N \cdot 15M$ Adders are required in order to online transform the kernel.

In Table 2.1 and in Table 2.2 an analysis of the exact resources of the Baseline and Winograd implementations is presentated. Winograd need another 32 additions in order to transform the input kernel. We suppose that the kernel is given as input 4×4 already transformed offline. Since the kernel is transformed once for the whole procedure on each engine, it does not need to be transformed online. In case we need the kernel transformation, we totally have $N \cdot (32 + 28) \cdot M/4 =$ additions in the Winograd component.

2.4 Scheduling

The main purpose of this section is to fully analyze scheduling of both main components we designed and further explore parallelization and pipelining between different layers.

2.4.1 Baseline Architecture

Considering our baseline architecture, for one input channel we will further analyze the amount of time needed for the system to process a single Image Input. As mentioned before, in order our serial to parallel converter to store enough pixels to perform convolution and give valid output, $\lfloor Kernel_{Height}/2 \rfloor \times (Image_{Width}) + \lceil Kernel_{Width}/2 \rceil$ clock cycles are needed. After valid $M \times N$ pixels are streamed into the convolution unit, $M \times N$ multiplications are perfmormed parallely in one clock cycle. After each multiplication is completed $M \times N$ outputs have to be added together to calculate the final output pixel. In order to do so we use a $M \times N$ Tree Adder that performs the whole adding operation in one clock cycle. Any kind of Tree Adder can be freely chosen for better results. After the sum is ready, final output of the component is formatted checking overflow or underflow and normalizing the result to fit in the same bandwidth as the input. Format operation needs another clock cycle to be correctly performed. In order to generate the first valid output we need :

$$\text{Cycles-First-Output} = \lfloor Kernel_{Height}/2 \rfloor \times (Image_{Width}) + \lceil Kernel_{Width}/2 \rceil + 3$$

After the first output is generated the above process is fully pipelined. In each clock cycle another output pixel is generated. To fully process an Input Image we need :

$$\text{Cycles-Full-Image} = \text{Cycles-First-Output} + (Image_{Width}) \times (Image_{Height})$$

We expect that the FPGA will have a continuous stream of images at its input to classify and we want to perform this classification as fast as possible, as it's happening in real time Image classification applications. In software CNN applications, the next layer (i) waits for the fully output of the previous layer $(i-1)$, to start processing. In our design, since resources are available, once layer $(i-1)$ starts to generate valid outputs, these outputs can be directly loaded to the next layer and the next layer can start performing operations since enough input pixels are stored in it. All the operations in one Layer can be fully parallelized by generating our Processing Unit $(InputMaps) \times (OutputMaps)$ times.

As for the Max-Pooling operation, it can be pipelined with the whole convolution operation. As stated before, in Baseline convolution design, Max-Pool uses two FIFO queues. These FIFO queues need $Image_{Width} + 2$ CCs to properly load the image and two clock cycles to compare. After these CCs a valid output is generated per CC. In total we need :

$$\text{Cycles-Max-Pool} = Image_{Width} + 3 + (Image_{Width}) \times (Image_{Height})$$

Totally in order to perform a convolution and a max-pool operation we need:

$$\begin{aligned}\text{Cycles-Total} &= Image_{Width} + 2 + Image_{Width} + 2 + (Image_{Width}) \times (Image_{Height}) \\ &= 2 \times Image_{Width} + (Image_{Width}) \times (Image_{Height}) + 4 \quad \text{CCs}\end{aligned}$$

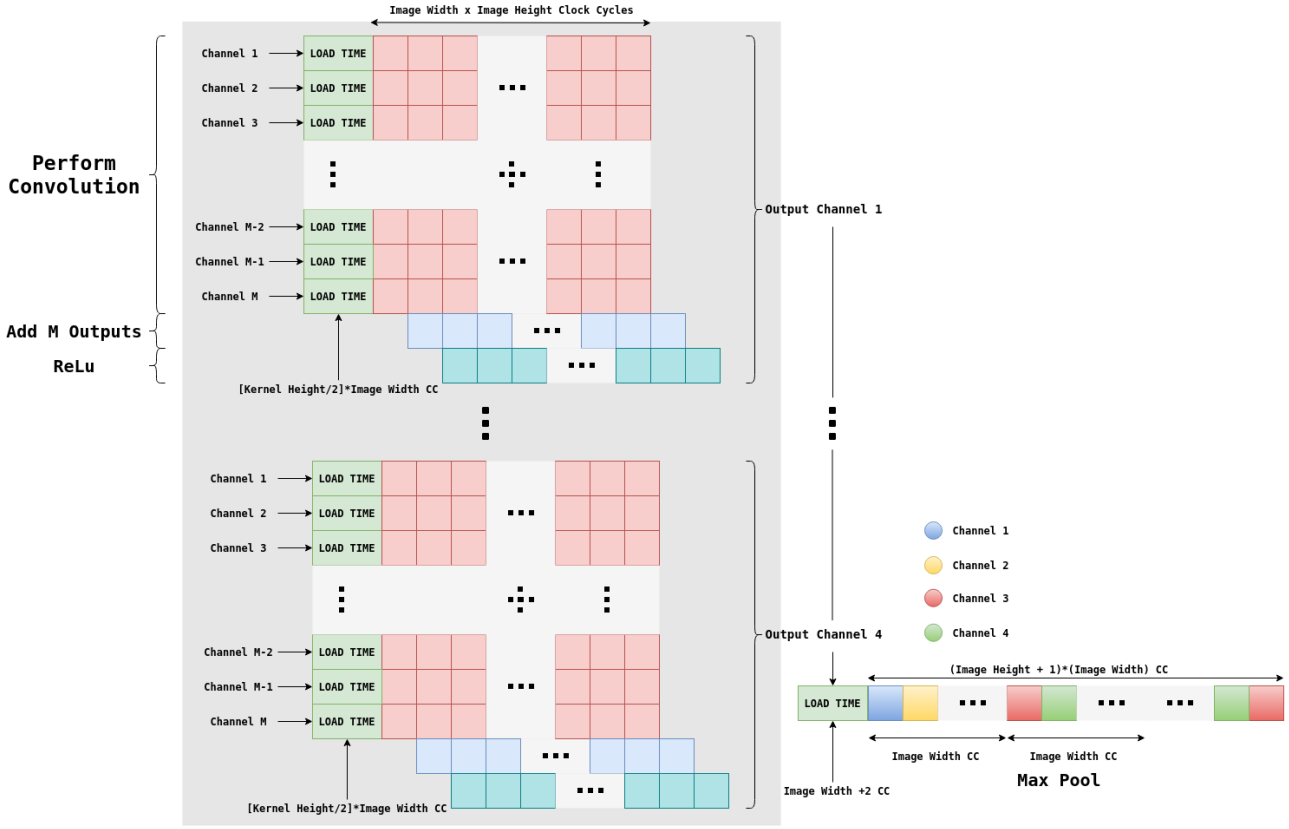


Figure 2.14: Clock Cycles for M input Maps and 4 Output Channels. One Max-Pooling component is required to deal with four output channels. This convolution and max-pooling procedure are pipelined.

2.4.2 Winograd Architecture

To start giving valid output tiles on 3×3 Winograd, $2\times\text{Image}_{Width} + 3$ CCs are necessary to load enough pixels. As it was explained in previous section, one Winograd component requires $(Image_{Height}+1)\times\text{Image}_{Width}$ CCs to produce all the correct output tiles for 4 input channels. These tiles must be all added together to generate the correct tiles of the Output Channel.

$$\text{Cycles-First-Output} = 2 \times Image_{Width} + 4$$

In order to fully complete the convolution operation in four different input images, Winograd component need:

$$\text{Cycles-Four-Images} = \text{Cycles-First-Output} + (Image_{Height} + 1) \times Image_{Width}$$

As for Max-Pooling operation, as the tiles are of size 2×2, we can just use a comparator without serial to parallel conversion that includes memories and loading time. Max-Pooling in Winograd can be done in one CC exactly the next CC the 2×2 tile is generated. Comparing time required to fully complete baseline and Winograd design, Baseline needs $Image_{Width}$ less clock cycles.

Chapter 3

Approximations

3.1 Approximate Hybrid High Radix Multiplier

High radix encodings [41] offer partial products reduction, and as a result, their accumulation requires smallerr trees, leading to energy, area, and/or delay savings. However, high radix encodings require complex encoding and partial product generation circuits, negating thus the benefits of the partial products reduction. [20] introduces a hybrid high radix encoding and its performed approximations for simplifying its circuit complexity are presented. In this technique the multiplicand B is encoded using the approximate high radix encoding, generating $\tilde{B}$, and the approximate multiplication $A \cdot \tilde{B}$ is performed. This multiplier, can be easily adjusted in different size of multipliers, if we appropriate adjust the encoding table to generate the encoded part of the multiplicand.

3.1.1 Hybrid High Radix Encoding

Configuration parameter k, is indicating the number of bits to be encoded with the high radix-2^k encoding. We consider that B has length of n bits. B has to be divided in two groups : the MSB part of $n-k$ bits and the LSB part of k bits. The configuration parameter, $k \geq 4$, is an even number, $k = 2m, m \in \mathbb{Z}$, with $m \geq 2$. The MSB part is encoded using the radix-4 (modified Booth) encoding, while the LSB part is encoded with the high radix-2^k encoding.

$$B = -b_{n-1}2^{n-1} + \sum_{i=0}^{n-2} b_i 2_i = \sum_{j=k/2}^{n/2-1} y_j^{R4} 4^j + y_0^{R2^k}$$

where

$$y_j^{R4} = -2b_{2j+1} + b_{2j} + b_{2j-1}$$

and

$$y_0^{R2^k} = -2^{k-1}b_{k-1} + 2^{k-2}b_{k-2} + \ldots + b_0$$

The radix-4 encoding included $(n-k)/2$ digits $y_j \in [0, \pm 1, \pm 2]$, while $y_0^{R2^k} \in [\pm 1, \pm 2, \pm 3, \ldots, \pm(2^{k-1}-1), \pm 2^{k-1}]$ correspond to the radix-2^k encoding.

Overall, B is encoded with $(n-k)/2+1$ digits. The above hybrid high radix encoding is characterized by increased logic complexity, due to the high radix values of $y_0^{R2^k}$ that are not powers of two, and thus, and approximate version is designed. In order to retain high accuracy, the radix-4 encoding of the MSB is performed accurately. In particular, in the approximate encoding, all the values that are not power of two and the k 4 smallest powers of two as well ,are rounded to the nearest of the 4 largest powers of two or 0, so that the sum of all the values of the approximate digit $\hat{y}_0^{R2^k}$ is 0. Only the four largest powers of two are chosen, so that the radix-2^k circuit requires only about double the area in comparison with the radix-4 encoder.Therefore, B is encoded as follows:

$$\tilde{B} = \sum_{j=k/2}^{n/2-1} y_j^{R4} 4^j + \hat{y}_0^{R2^k}$$

where

$$y_j \in \{0, \pm 1, \pm 2\}$$

and

$$\hat{y}_0^{R2^k} \in \{0, \pm 2^{k-4}, \pm 2^{k-3}, \pm 2^{k-2}, \pm 2^{k-1}\}$$

Table 3.1: APPROXIMATE RADIX-2^k ENCODING TABLE

R2^k Digit		**Output**				
$y_0^{R2^k}$	$\hat{y_0}^{R2^k}$	$sign$	$\times 2^{k-1}$	$\times 2^{k-2}$	$\times 2^{k-3}$	$\times 2^{k-4}$
$[0, 2^{k-5})$	0	0	0	0	0	0
$[2^{k-5}, 2^{k-4}+2^{k-5})$	2^{k-4}	0	0	0	0	**1**
$[2^{k-4}+2^{k-5}, 2^{k-3}+2^{k-4})$	2^{k-3}	0	0	0	**1**	0
$[2^{k-3}+2^{k-4}, 2^{k-2}+2^{k-3})$	2^{k-2}	0	0	**1**	0	0
$[2^{k-2}+2^{k-3}, 2^{k-1})$	2^{k-1}	0	**1**	0	0	0
$[-2^{k-1}, -2^{k-2}-2^{k-3})$	-2^{k-1}	**1**	**1**	0	0	0
$[-2^{k-2}-2^{k-3}, -2^{k-3}-2^{k-4})$	-2^{k-2}	**1**	0	**1**	0	0
$[-2^{k-3}-2^{k-4}, -2^{k-4}-2^{k-5})$	-2^{k-3}	**1**	0	0	**1**	0
$[-2^{k-4}-2^{k-5}, -2^{k-5}$	-2^{k-4}	**1**	0	0	0	**1**
$[-2^{k-5}, 0)$	0	**1**	0	0	0	0

Table 3.2: ACCURATE RADIX-4 ENCODING TABLE

Input			**R4 Digit**	**Output**		
b_{2j+1}	b_{2j}	b_{2j-1}	y_j^{R4}	$sign_j$	$\times 2_j$	$\times 1_j$
0	0	0	0	0	0	0
0	0	1	1	0	0	**1**
0	1	0	1	0	0	**1**
0	1	1	2	0	**1**	0
1	0	0	-2	**1**	**1**	0
1	0	1	-1	**1**	0	**1**
1	1	0	-1	**1**	0	**1**
1	1	1	0	**1**	0	0

Using Table 3.1, the output signals $sign, \times 2^{k-1}, \times 2^{k-2}, \times 2^{k-3}, \times 2^{k-4}$ define the radix - 2^k digit $\hat{y}_0^{R2^k}$. It's logic equations are the following:

$$sign = b_{k-1}$$
$$\times 2^{k-4} = (\bar{b}_{k-2} \cdot \bar{b}_{k-3} \cdot \bar{b}_{k-4} + b_{k-2} \cdot b_{k-3} \cdot b_{k-4}) \cdot (b_{k-4} \oplus b_{k-5})$$
$$\times 2^{k-3} = \bar{b}_{k-1} \cdot \bar{b}_{k-2} \cdot (\bar{b}_{k-3} \cdot b_{k-4} \cdot b_{k-5} + b_{k-3} \cdot \bar{b}_{k-4}) + b_{k-1} \cdot b_{k-2} \cdot (b_{k-3} \cdot \bar{b}_{k-4} \cdot \bar{b}_{k-5} + \bar{b}_{k-3} \cdot b_{k-4})$$
$$\times 2^{k-2} = \bar{b}_{k-2} \cdot b_{k-3} \cdot (b_{k-1} + b_{k-4}) + b_{k-2} \cdot \bar{b}_{k-3} \cdot (\bar{b}_{k-1} + \bar{b}_{k-4})$$
$$\times 2^{k-1} = \bar{b}_{k-1} \cdot b_{k-2} \cdot b_{k-3} + b_{k-1} \cdot \bar{b}_{k-2} \cdot \bar{b}_{k-3}$$

In the same way logic equations that define the digit y_j^{R4}, are generated from Table 3.2.

$$sign_j = b_{2j+1}$$
$$\times 1_j = b_{2j-1} \oplus b2j$$
$$\times 2_j = (b_{2j+1} \oplus b2j) \cdot \overline{(b_{2j-1} \oplus b_{2j})}$$

This approximate hybrid radix encoding technique is explored with its application to 16-bit signed numbers, for k = 6, 8, 10, namely, the LSBs are encoded using the radix-64, radix-256, radix-1024 encoding respectively. As for radix-64 encoding, the bits of B are grouped as:

$$\overbrace{b_{15}b_{14}\underbrace{b_{13}b_{12}}_{y_6^{R4}}}^{y_7^{R4}}\overbrace{\underbrace{b_{11}b_{10}}_{y_5^{R4}}\underbrace{b_9b_8}_{y_4^{R4}}}^{y_5^{R4}}\overbrace{b_7\underbrace{b_6b_5b_4b_3b_2b_1b_0}_{y_0^{R64}}}^{y_3^{R4}}$$

3.1.2 Partial Product Generation

In this hybrid multiplier, these is a reduction of $k/2-1$ partial products generated in the multiplication $A \cdot \tilde{B}$. In Figure 3.1 four partial products generators are presented. The partial products created from each encoding are shown in Table 3.3.

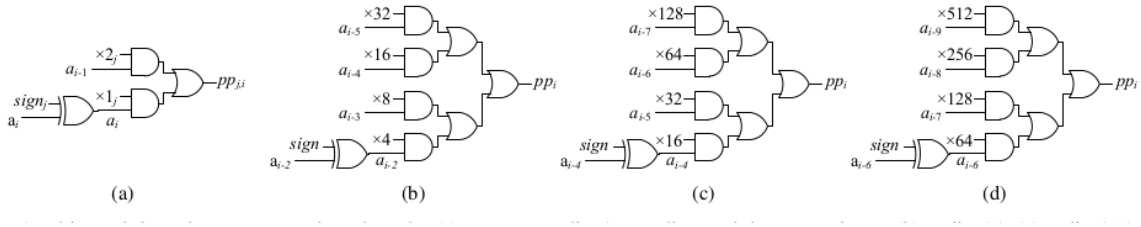


Figure 3.1: $i-bit$ partial product generator based on the (a) accurate radix-4 encoding and the approximate (b) radix-64, (c)radix-256, (d) radix-1024 encoding. $a_i : i-bit$ of operand A, $\alpha_i = \mathrm{a}_i \oplus sign$

Radix Encoding	**Partial Produxts**
Radix-4	$0, \pm A, \pm 2A$
Radix-64	$0, \pm 4A, \pm 8A, \pm 16A, \pm 32A$
Radix-256	$0, \pm 16A, \pm 32A, \pm 64A, \pm 128A$
Radix-1024	$0, \pm 64A, \pm 128A, \pm 256A, \pm 512A$

Table 3.3: Partial Products per Radix Encoding

In addition, the three hybrid high radix encodings create the partial product trees shown in Figure 3.2. These trees also include the encoding's correction terms (constant terms and sign factors). The implementation of the partial product accumulation can be chosen by the designer.

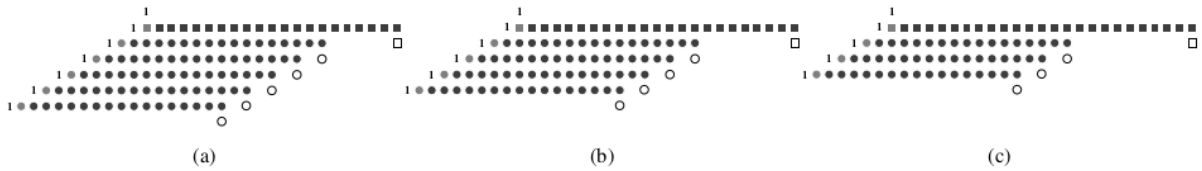


Figure 3.2: Partial product tree based on the hybrid encoding of accurate radix-4 and approximate (a) radix-64, (b) radix-256 and (c) radix-1024 encoding.■ : partial product bits from the approximate high radix encoding, ●: partial product bits from the accurate radix-4 encoding, ■ and ● : inverted MSBs of the partial products, □ and ○ : sign factors

3.2 Data Type Exploration

In a digital hardware, numbers are stored in binary words. A binary word is a fixed-length sequence of bits. How hardware components or software functions interpret this sequence of 1's and 0's is defined by the data type. Binary numbers, are represented as either fixed point or floating point data types. Popular Neural Network tools, such as TensorFlow, Keras or Caffe work by default with 32 - bit Floating point data. Since memory bandwidth, area on chip and power consumption are our main goals in order to optimize our design, we further explored data types and the tradeoff between main design metrics, like accuracy and throughput. General purpose CPUs are not optimized for half or smaller floating point operations. Neural Network tools, since new applications are mainly using data types smaller than 16-bit and GPU implementations are performing them fast enough, are offering user the choice to train the model with half float data. In our approach, there is no need for the network to be retrained with different data types since every conversion is taking place in the first stage of the network.

3.2.1 Fixed Point Architecture

In Fixed Point data type architecture, each N-bit length sequence of bits is characterized by the word length in bits, the position of the binary point, and whether it is signed or unsigned. The position of the binary point is the factor by which fixed point values are scaled and interpreted.

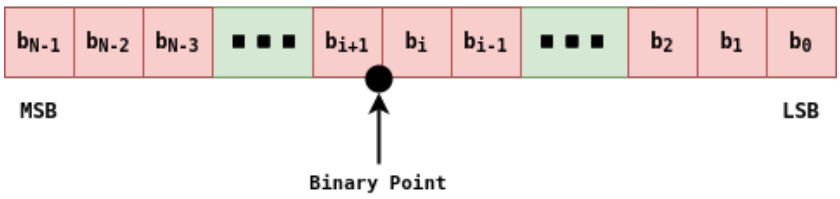


Figure 3.3: N-bit Fixed Point Representation

In Figure 3.1 a binary representation of a generalized fixed point number (signed or unsigned) is shown. In this format :

- N is the word length in bits
- b_i is the i^{th} binary digit
- b_{N-1} and b_0 are the locations of the most and least significant bits respectively.

The binary point is shown $i+1$ places to the left of the LSB, therefore, the number is said to have $i+1$ fractional bits, or a fraction with the length of $i+1$. Fixed point numbers can be encoded according to the following scheme:

$$X = 2^{-FractionalLength} \times (\text{Stored Integer})$$

$$X = (\frac{1}{2^b}) \times \Big[-2^{N-1} x_{N-1} + \sum_{n=0}^{N-2} 2^n x_n \Big]$$

Where, X is the Real Value and b is the fractional length.

Where Stored Integer is the raw binary number, without taking care of the binary point. Fixed point numbers are indeed a close relative to integer representation. These two only differ in the position of the binary point. In fact, we can even consider integer representation as a "special case" of fixed point numbers, where the binary point is at position 0. All the arithmetic operations on a computer that can operate on integers, can therefore be applied to fixed point numbers as well.

Therefore, the benefit of fixed point arithmetic is that they are straight-forward and efficient as integer arithmetic in computers. We can reuse all the hardware that is built for integer arithmetic to perform real numbers arithmetic, using fixed point representation. This is really significant when we approach prototype boards like ASIC or FPGA implementations, since these boards don't have built-in processing units for floating point data types. In other words, fixed point arithmetic comes for free on computers. One of the main issues we have to take care of when we use fixed point arithmetic, is to properly balance integer and fractional part of any given number. When binary point position is moved to the left, the precision of the fractional part of the number is increased, since smaller values can be represented. Given a constant word length, this has a direct effect in the integer part of the number. Integer part range is decreased, since less bits are dedicated to represent it. In exactly the same way, we can increase the precision (range) of the integer part of the number, having as a consequence to decrease the precision of the fractional part. In order to apply fixed point arithmetic to Convolutional Neural Networks, the following steps have to be followed. First of all, the network have to be trained in order to achieve our desired accuracy. Since the final network weights are obtained, a statistic analysis of data in a layer should take place. Each layer of the network should be treated separately. Once the statistical analysis is done, minimum and maximum values of each layer's weights and input values are known. Binary point position, should be placed in the optimal position in order to lose less precision as possible and achieve highest accuracy in the evaluation test.

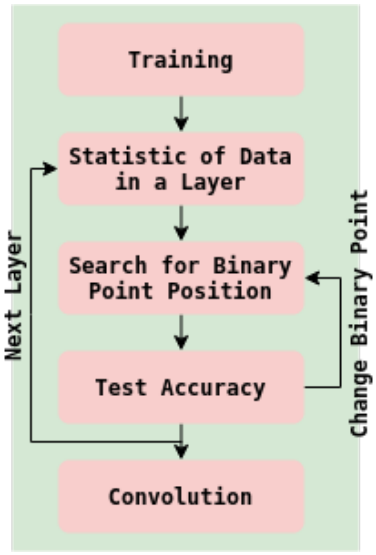


Figure 3.4: CNN Training with fixed point arithmetic

Fixed point operations in FPGA are way faster than floating point and require less energy and area to perform. Although fixed point representations have a relatively limited range of values that they can represent. During our statistical analysis maximum and minimum possible value that can be stored must be taken into account.

$$\textbf{Minimum} = -2^{N-1}/2^b$$
$$\textbf{Maximum} = (2^{N-1} - 1)/2^b$$

If the numbers we want to represent fit into this range, then we can use fixed point representation, otherwise floating point notation or word length extension is required.

3.2.2 Floating Point Architecture

Modern applications, especially CNN require precision. A CNN is combined by multiple convolutional layers. Since the output of each layer is propagated as input to the next layer, if a precision error occurs at a specific stage of the procedure, it's most likely that this error will be propagated and become noticeable in the following layers. Floating Point Notation is an alternative to the fixed point notation and is the representation that most modern computers use when storing fractional numbers in memory. Floating Point is a way to represent very large or very small numbers precisely using scientific notation in binary. Because of its wide use, the format used to store Floating Point numbers in memory has been standardized by the institute of Electrical and Electronic Engineers in IEEE 754. This standard defines a number of different representations that can be used when storing Floating Point Numbers in memory. This standard defines the following types:

- Half Precision : 16-bits of storage
- Single Precision : 32-bits of storage
- Double Precision : 64-bits of storage
- Quadruple Precision : 128-bits of storage

When it comes to storing Floating Point numbers in memory, only three critical parts of that basic structure are stored: Sign, Exponent and Mantissa.

Figure 3.5: Floating Point Notation

In the IEEE 754 standard single precision float numbers, are divided in three sections. 1 bit S for the sign, 8 bits E for the exponent and 23 bits M for the mantissa. Floating point numbers can be encoded with the formula:

$$X = (-1)^S \times 1.M \times 2^{(E-127)}$$

Where X is the real value number we want to interpret.
In our applications, we try to achieve the smallest memory bandwidth as possible. To achieve this half float representation or even custom float representations with even smaller word length in bits are introduced. General purpose CPUs have Floating Point Units (FPUs) and ALUs in order to perform floating point operations. Considering FPGA, by adopting a lower bit length data format, the performance of CNN accelerator in terms of chip area, power efficiency and memory requirement can be improved significantly. Furthermore, as FPGA lacks FP arithmetic units, low-bit integer (Fixed Point) formats have been used to reduce memory bandwidth and computational resource requirements.

3.3 Approximations in the Convolution Processing Unit

Considering the Fixed Point model, since FPGA platforms offer multiplication units that use DSP slices in order to perform multiplication, we can straight forward design the architecture that was presented in Chapter 2. As described before, in order to perform the convolution operation, $M \times N$ filter has to be multiplied element by element with a $M \times N$ tile of the input Image. In order the multiplies to be performed, in certain platforms that contain multiply units we can use hard multiplier (*). In prototype boards, we don't have this advantage and multipliers have to be designed. The Approximate Multiplier described in section 3.1, can perform multiply between two N-bit numbers, adjusting the configuration parameter k depending on the aggressiveness of the approximation.

Our first design approach was with hard multiplier, taking advantage of the multiplication unit. Considering that this design can be adjusted in prototype boards, multiplication operation inside the convolution unit was replaced with the approximate multiplier. Approx multiplier was compared with modified Booth multiplier in terms of resource utilization and speed.

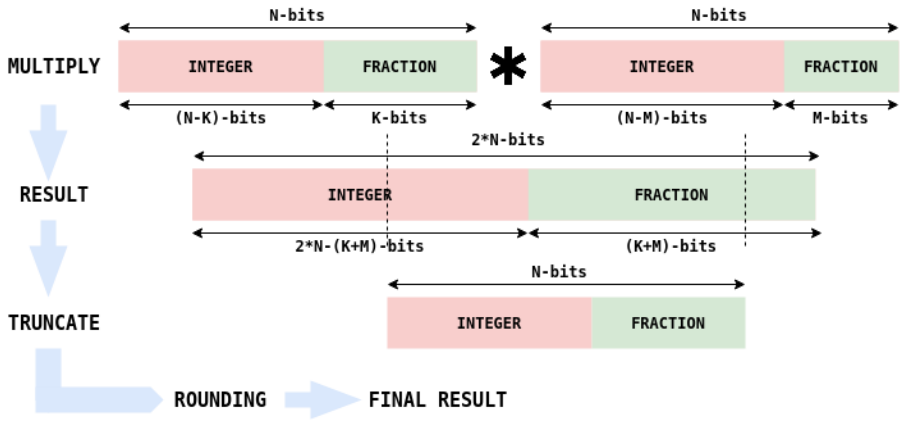


Figure 3.6: Fixed Point Multiplication

Considering Fixed Point Notation, when we multiply 2 N-bit integer numbers with fraction K and M-bits, result with double the word length 2*N-bits occurs. New fraction will consist of (K+M)-bits and new integer part of [2*N-(K+M)]-bits. Since a common binary point is selected for all input data and another one for all weight data, the same procedure will be repeated for all multiply operations in a single layer. After the numbers are multiplied, the result has double the length in bits, significantly increasing the memory bandwidth of our design. In order to keep memory requirements low, we adjust the result to the same bit-length as the initial numbers pre-multiplication (N-bits). In order to decide which bits are going to be truncated the statistical analysis done before is investigated. We find the maximum Integer value and we keep those integer part bits that can represent it to prevent overflow. Then with the remaining bits to reach N-bits, we keep the most significant fractional bits in order to maintain as much precision as possible. Before finally truncate the remaining fractional bits, a rounding method takes place to maximize the precision of the truncated result.

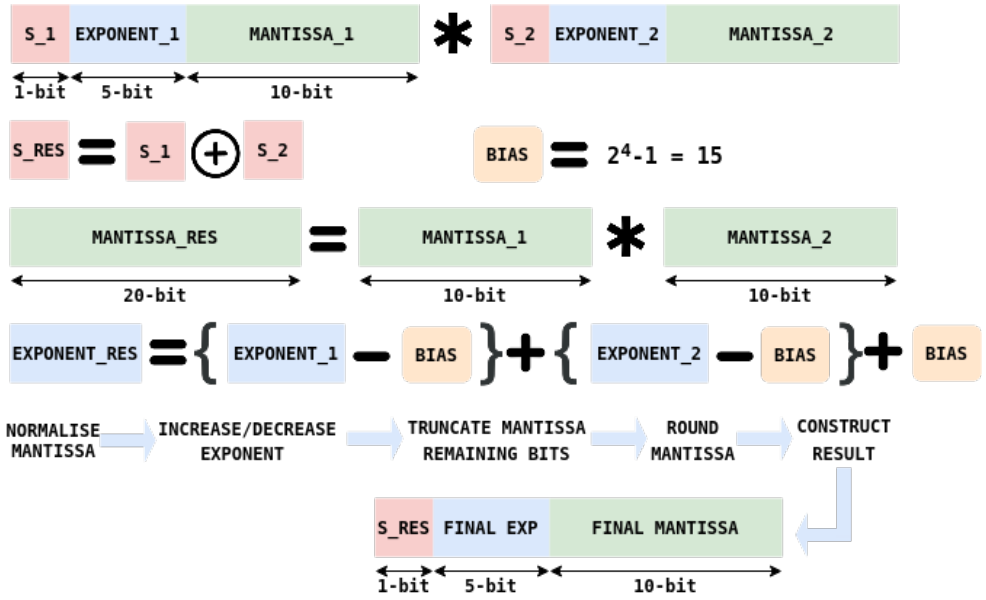


Figure 3.7: Floating Point Multiplication

For Fixed Point Notation, multiplication is way more complicated. The three different parts of the number, have to be treated separately at first in order to compute Mantissa_Res ,Exponent_Res and Sign_Res. Final result sign can be decided instantly since the MSB of each number dedicates it's sign. Bias should be subtracted from both exponents in order to get the true exponent from each one and then they should be added together plus the bias to get Exponent_Res. Mantissas should be multiplied to construct the 20-bit Mantissa of the multiplication result. Since the result of the multiplication of two 10-bit mantissas has the length of 20-bit and we want a standard length of 16-bits Floating Point format, the Mantissa_Res should be adjusted in a way that it fits to 10-bit and lose minimum precision. When mantissas are multiplied, hidden bit should be used. In this way we have to multiply HiddenBit.MANTISSA_1 and HiddenBit.MANTISSA_2. Result consists of 22-bits. In order to normalise this result, we start from the MSB and we find the first '1'. If the first '1' is left from the binary point, in order to normalise we have to shift the mantissa left and the exponent is increased by one each time we shift left. If the first '1' is right from the binary point, then the mantissa have to be shifted right and the exponent is decreased by one each time the mantissa is shifted right. When mantissa is properly normalised, the exponent has it's proper value and mantissa has to be truncated in order to fit in 10-bits. So we count 10 bits from the first '1' and we truncate the remaining bits leading to the LSB. Mantissa now has to be rounded using a rounding technique in order to avoid as less precision loss as possible while we truncate. Finally if all the above steps are followed, final result of the multiplication is stored in 16-bits without losing precision. Same technique is used for single and double precision float formats with the proper adjustments considering lengths.

On the Fixed Point notation hard multiplier is straight forward replaced by the approximate hybrid multiplier described in 3.1. Considering the Floating Point notation, in order to apply the approximate multiplier we had to build our own component according to IEEE 754 standard floating point operations and replace the mantissa hard multiplication with the approximate one. Since our target was small Floating Point formats (smaller than the half one) and the approximate hybrid multiplier was designed for bigger word length, the resources and energy benefits for mantissas with length less than 10-bit was not significant, although it offers gains compared to the modified booth algorithm.

When dealing with Floating Point values, the main issues appeared were that due to the different exponent values of each real number, all decimal points must be aligned to the same exponent before we add their mantissas (including hidden bit). After the addition the final result must be again normalised in exact the same way as we did in multiplication. All this procedure increases the logic complexity of the hardware. In order to decrease this complexity and gain advantage of approximations in the floating point notation, Block Floating Point notation was designed.

3.4 Block Floating Point

In order to overcome the main disadvantages that occur from the increased complexity of the Floating Point operations, a Block Floating Point [6] engine was constructed. The design can reduce memory and off-chip memory requirements with small accuracy loss.

3.4.1 Block Floating Point Arithmetic

A N-data block represented with the BFP format consists of two parts: N mantissas and one exponent shared by the N numbers in a block. The process of the BFP conversion is defined as follows. Assuming that $\mathbf{X}$ is a data set containing N FP numbers, we can express the set as

$$\mathbf{X} = (x_1, ..., x_i, ..., x_N) = (m_1 \times 2^{e_1}, ..., m_2 \times 2^{e_2}, ..., m_N \times 2^{e_N})$$

We define the largest exponent in $\mathbf{X}$ as the block exponent ϵ_x.

$$\epsilon_x = \max_i e_i \in \{1, 2, ..., N\}$$

Where m_i is the mantissa and e_i the exponent of number X_i in data set $\mathbf{X}$. After we derive the common block exponent ϵ_x, all the numbers in the data set have to be expressed with the same exponent. In order to do so the mantissa number m_i is right shifted by d_i bits, where $d_i = \epsilon_x - e_i$. Thus, the BFP format of $\mathbf{X}$, is expressed as follows.

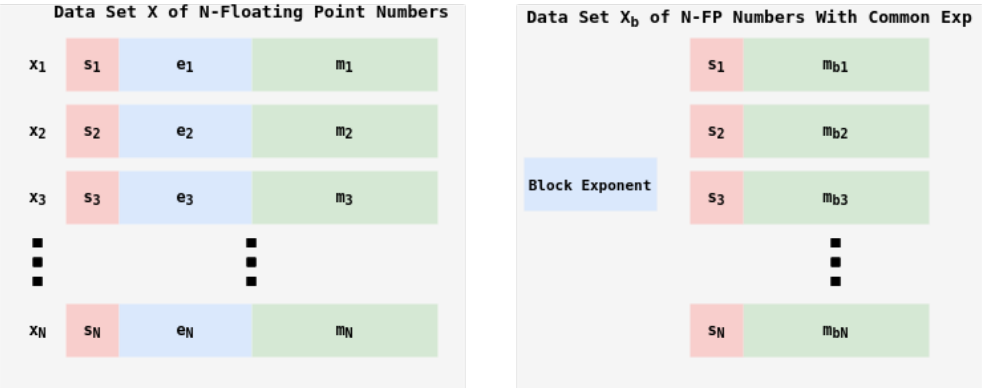


Figure 3.8: Block Floating Point Notation

$$\mathbf{X}_b = (x_{b1}, ..., x_{bi}, ..., x_{bN}) = M_{bX} \times 2^{\epsilon_x} = (m_{b1}, ..., m_{bi}, ..., m_{bN}) \times 2^{\epsilon_x}$$

Where,

$$m_{bi} = m_i >> d_i$$

is the BFP formatted mantissa.

Using this format, all the operations inside the CNN engine (multiplications, additions) can be done exclusively with the mantissas by updating the exponent before the final result. Block Floating Point is a method used to provide an arithmetic approaching floating point while using a fixed-point processor. Emulating floating-point behavior on a fixed-point processor tends to be very cycle intensive, since the emulation routine must manipulate all arithmetic computations to artificially mimic floating-point math on a fixed-point device.

3.4.2 Convolution Operation using BFP Notation per Layer

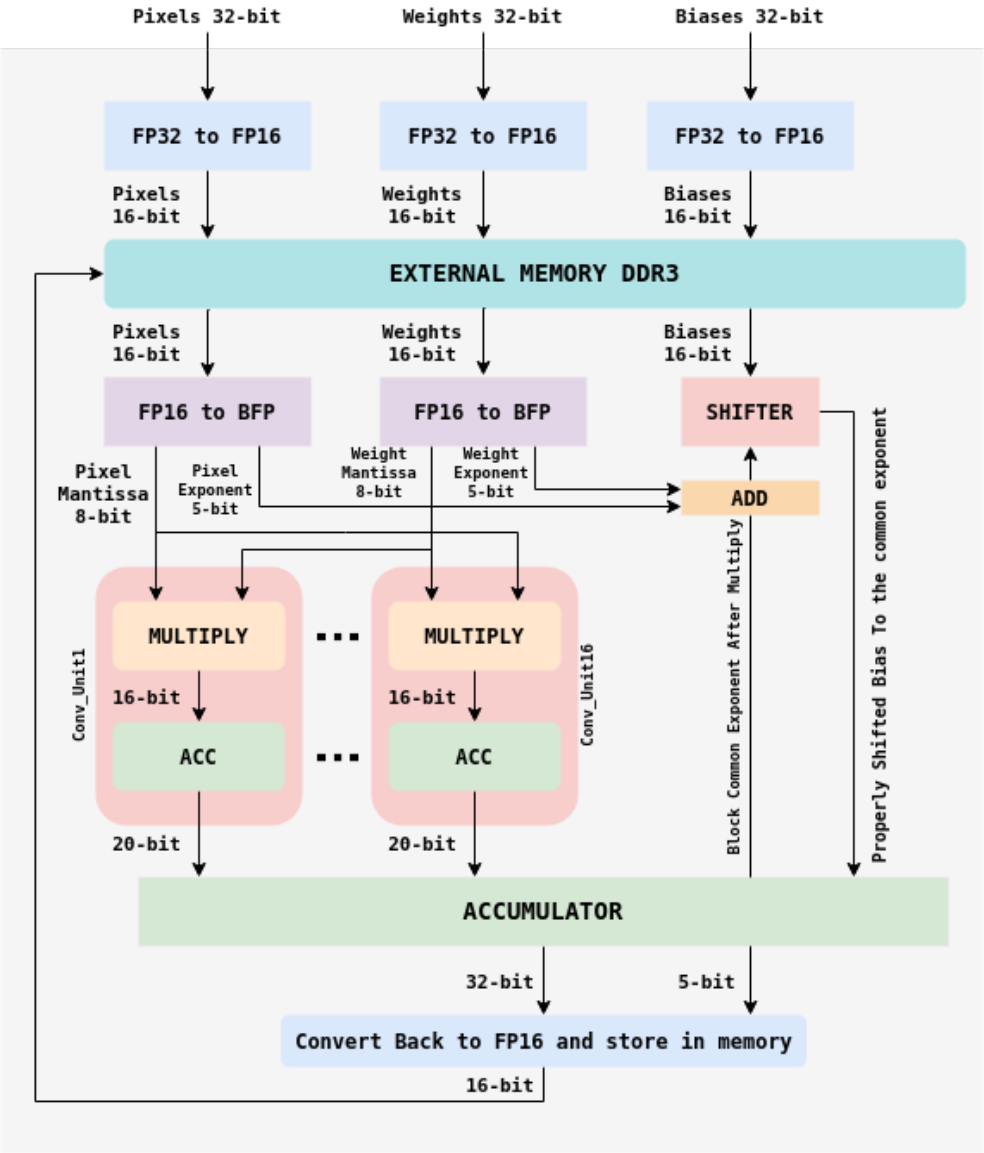


Figure 3.9: Block Floating Point Convolution

In Figure 3.7 the data flow of the BFP convolution in one output channel is shown [25]. We suggest that input pixels are in a Floating Point 32-bit format as the network is trained in such formats in popular CNN platforms. The input Pixels, Weights and Biases are first converted to their 16-bit floating point equivalent and stored in the DDR3 memory. We consider one common exponent (maximum) per output channel. In Figure 3.7 we have 16 Input Maps and one common exponent is found for these input maps. If we have $W \times L$ Input Map size and 16 Input Maps then we have a block of $16 \times W \times L$ numbers as input. From these numbers, we find the maximum exponent and we properly right shift the Mantissas according to d_i in order to properly represent the stored value.

The shift operation to convert to BFP format is taking place in the FP16 to BFP component. In order to perform the BFP multiplication operation between one input pixel a and one weight b, the following calculation takes place.

$$a = m_i^a \times 2^{\epsilon_a} \quad (InputPixel)$$
$$b = m_i^b \times 2^{\epsilon_b} \quad (Weight)$$
$$a \cdot b = (m_i^a \times 2^{\epsilon_a}) \times (m_i^b \times 2^{\epsilon_b}) = 2^{\epsilon_a + \epsilon_b} \times (m_i^a \cdot m_i^b)$$

If a and b have N elements then,

$$a \cdot b = \sum_{i=1}^{N} (m_i^a \times 2^{\epsilon_a}) \times (m_i^b \times 2^{\epsilon_b}) = 2^{\epsilon_a + \epsilon_b} \times (m_i^a \cdot m_i^b) = 2^{\epsilon_a + \epsilon_b} \times (m^a \cdot m^b)$$

The product $m^a \cdot m^b$ is computed entirely in fixed point arithmetic. Every element of the input Data Set has the common exponent ϵ_a and every element of the Weight Data Set has the common exponent ϵ_b. As it was explained in Figure 3.5 the exponents are added and the mantissas are multiplied. So in each Conv_Unit for each 3×3 tile of the input image 9 fixed point 8-bit multiplications are performed between the Input mantissa and Weight mantissa. To prevent overflow, the accumulated result is extended in 20-bit format. The exact same operation takes place in every of the sixteen Conv_Units. This operation is performed parallely for each input map. Once the 9 products are added together, then we have to add all the 16 channels and the bias to produce the output map of the channel. In order to properly adjust the bias to the exponent, we use a shifter that gets as input $\epsilon_a + \epsilon_b$ and shifts the bias left or right to have the same exponent. The final sum of every channel and the bias adopt a 32-bit format. The 32-bit mantissa is normalised and truncated in order to fit in 8-bit format and the exponent is increased or decreased. The output value is converted back to Floating Point 16-bit format and stored in memory. Same operation is taking place for every output channel of the layer. Once every output channel of the layer is completed, the maximum exponent of these is registered as the Block Exponent of the next layer.

The accuracy loss in this architecture mainly derives from the BFP2FP and FP2BFP conversions. Truncation and rounding are two common methods to handle shifted bits. Truncation mode introduces a biased error. This error will accumulate between layers and eventually produce an obvious deviation. Because the bit length is limited, the value that FP number can accurately represent is finite. Between the two adjacent FP numbers, there must be an infinite number of real numbers that cannot be accurately represented by FP numbers. In the IEEE 754 standard, these numbers are approximated by nearest FP numbers. We used the round-to-nearest (RN) mode to minimize the error.

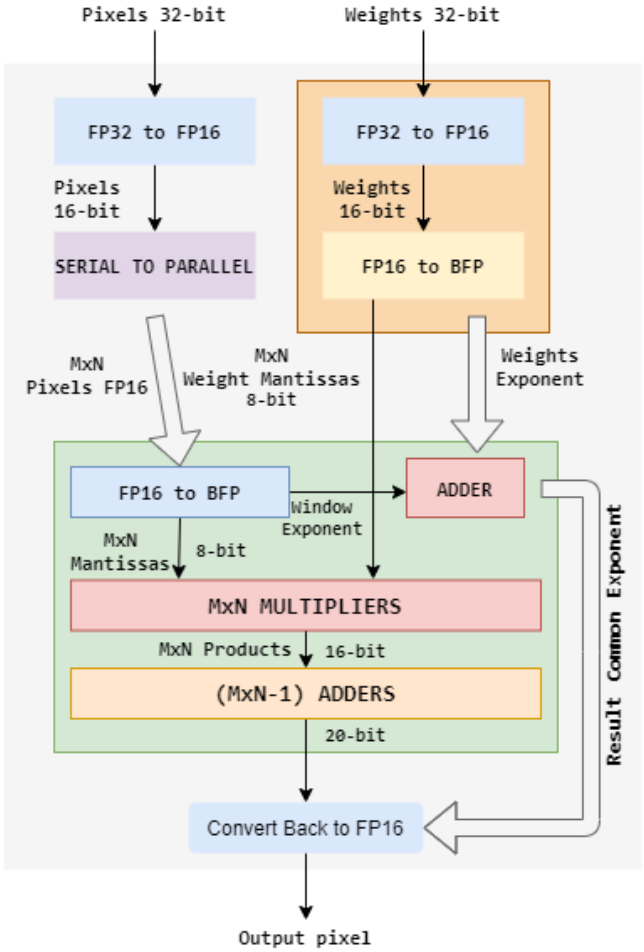


Figure 3.10: M×N Input Kernel Transformed to BFP Notation. M×N Input Pixels share the same exponent in each convolution operation. Exponents are added and final result in FP16 format.

3.4.3 BFP Convolution Per Window

In the first approach, a common exponent from all the input maps was obtained. Since different input maps in most cases describe different features extracted, there might be diversities between the values that each map contain. If these values diverge by a significant margin, a common exponent to describe all of them could lead to precision loss since a big exponent might be inappropriate to preserve accuracy in really small values. To face this problem, a common exponent per convolution window was introduced. A block exponent is chosen from the input $M \times N$ Kernel. Each Clock Cycle, after the Loading Time of the Serial To Parallel Converter, $M \times N$ Input Pixels window is generated to perform convolution operation. From this window, we find the maximum exponent, we set it as the $M \times N$ Data Set Block Exponent, mantissas are properly right shifted and 8-bit Input Pixels Mantissas are Multiplied with the 8-bit Weight Mantissas. The products generated by the input pixels with weights multiplication, share the same exponent as described in

previous section and can be directly added together to calculate the final sum. Then the 20-bit final sum with the exponent obtained by the addition of the two exponents, format the FP16 result.

Comparing this method to the method described in previous section, we lose minimal accuracy and we do not have to implement a DDR3 memory to store data between layers. In this method output data can be directly pipelined to next layer without using memory, since maximum exponent is found online. In the first method if we had M input channels and Image Size $W \times L$, we obtained a block exponent from $M \times W \times L$ values compared to $M \times N$. Since these $M \times N$ windows are neighboring, these values will not decline and a common exponent will properly describe all of them in most cases.

Chapter 4

Evaluation

In this chapter, the results of our work will be presented. These results were obtained from the multiple different architectures we proposed. Using the components described in the previous chapters. we created the convolution core of three Convolutional Neural Networks and we measured how the classification accuracy was affected while multiple approximation techniques were combined together.

4.1 Test Setup

The convolution units described in previous Chapters, were generic considering data size, data type and kernel size. Hence, these units can be properly combined according to the target device restriction, in order to construct a Convolutional Neural Network of our choice. In order to test resources utilization, throughput and accuracy deviations, we examined three different CNNs.

- **CIFAR 10**

The CIFAR-10 dataset consists of 60000 32x32 colour images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images. The dataset is divided into five training batches and one test batch, each with 10000 images. The test batch contains exactly 1000 randomly-selected images from each class. The training batches contain the remaining images in random order, but some training batches may contain more images from one class than another. Between them, the training batches contain exactly 5000 images from each class. In the following figure classes in the dataset, as well as 10 random images from each class are presented.

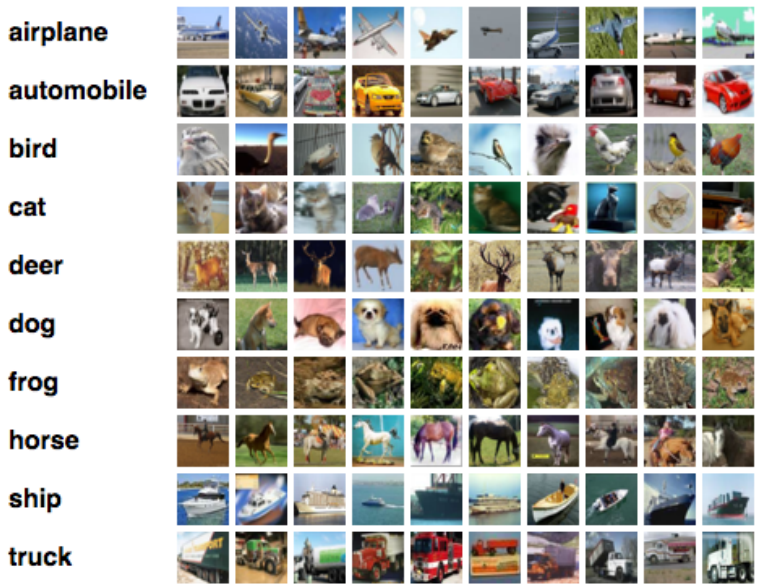


Figure 4.1: CIFAR 10 CNN dataset presentation

We used the following CNN model for the CIFAR-10 dataset classification.

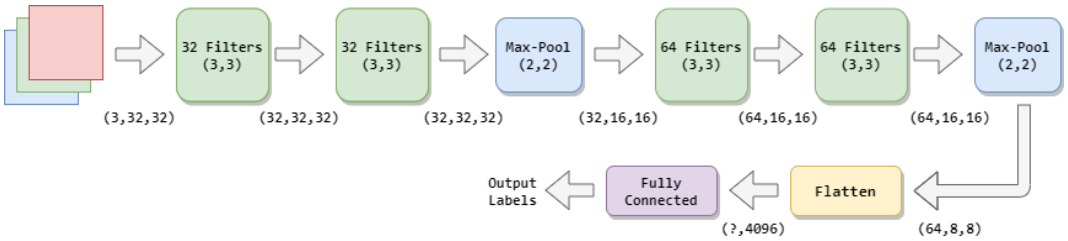


Figure 4.2: CIFAR-10 Model

- **MNIST**

The MNIST database (Modified National Institute of Standards and Technology database) is a large database of handwritten digits that is commonly used for training various image processing systems. The database is also widely used for training and testing in the field of machine learning. It was created by "re-mixing" the samples from NIST's original datasets. The creators felt that since NIST's training dataset was taken from American Census Bureau employees, while the testing dataset was taken from American high school students, it was not well-suited for machine learning experiments. Furthermore, the black and white images from NIST were normalized to fit into a 28x28 pixel bounding box and anti-aliased, which introduced grayscale levels. The MNIST database contains 60,000 training images and 10,000 testing images. Half of the training set and half of the test set were taken from NIST's training dataset, while the other half of the training set and the other half of the test set were taken from NIST's testing dataset.

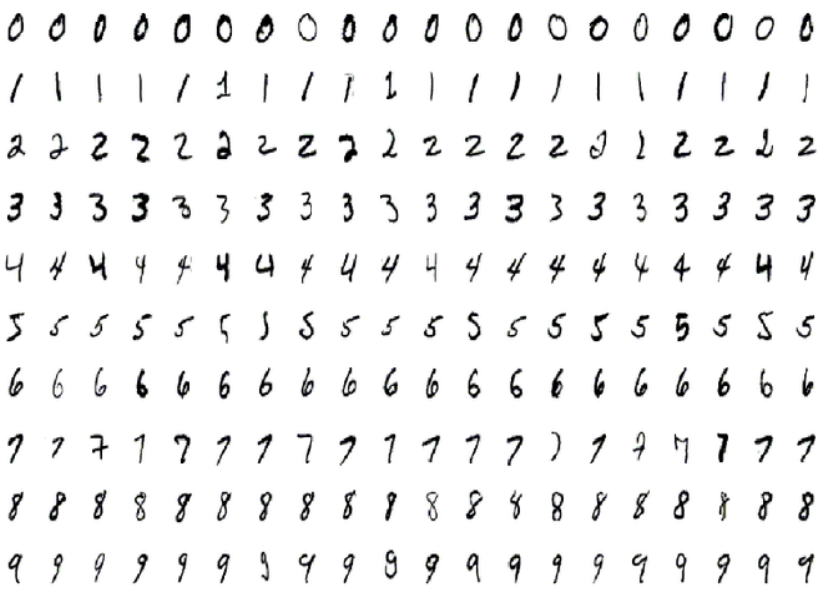


Figure 4.3: MNIST Dataset Presentation

The follwoing CNN model was used for the MNIST dataset classification.

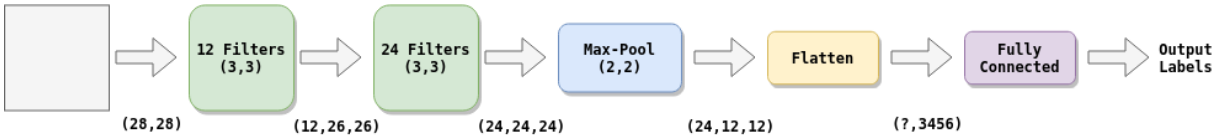


Figure 4.4: MNIST Model

- **Ships in Satellite Imagery**

Satellite imagery provides unique insights into various markets, including agriculture, defense and intelligence, energy, and finance. New commercial imagery providers, such as Planet, are using constellations of small satellites to capture images of the entire Earth every day. This flood of new imagery is outgrowing the ability for organizations to manually look at each image that gets captured, and there is a need for machine learning and computer vision algorithms to help automate the analysis process. The aim of this dataset is to help address the difficult task of detecting the location of large ships in satellite images. Automating this process can be applied to many issues including monitoring port activity levels and supply chain analysis.

The dataset consists of image chips extracted from Planet satellite imagery collected over the San Francisco Bay and San Pedro Bay areas of California. It includes 4000 80x80 RGB images labeled with either a "ship" or "no-ship" classification. Image chips were derived from PlanetScope full-frame visual scene products, which are orthorectified to a

3 meter pixel size. The "ship" class includes 1000 images. Images in this class are near-centered on the body of a single ship. Ships of different sizes, orientations, and atmospheric collection conditions are included. Example images from this class are shown below.

Figure 4.5: Sample of Images that were classed as "ships"

The "no-ship" class includes 3000 images. A third of these are a random sampling of different landcover features - water, vegetion, bare earth, buildings, etc. - that do not include any portion of an ship. The next third are "partial ships" that contain only a portion of an ship, but not enough to meet the full definition of the "ship" class. The last third are images that have previously been mislabeled by machine learning models, typically caused by bright pixels or strong linear features. Example images from this class are shown below.

Figure 4.6: Sample of Images that were classed as "non-ships"

The following network model was designed to classify the Ships in Satellite Imagery dataset. Since another project in Myriad 2 required images to have dimension that is power of two, the Ships in Satellite Imagery dataset images dimensions were changed from 80×80 to 128×128 without affecting the classification accuracy.

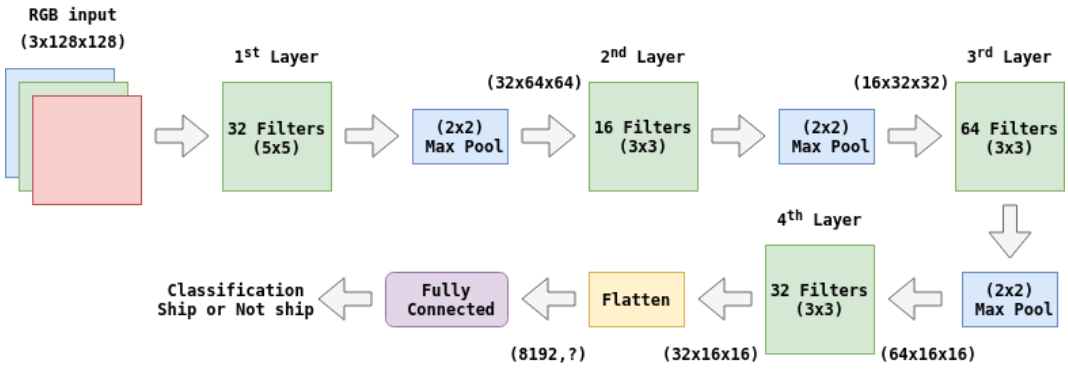


Figure 4.7: Ship Detection CNN model

The board used to setup the Engines wass Zynq-7020. The Zynq-7000 family offers the flexibility and scalability of an FPGA, while providing performance, power, and ease of use typically associated with ASIC and ASSPs. The range of devices in the Zynq-7000 family allows designers to target cost-sensitive as well as high-performance applications from a single platform using industry-standard tools. While each device in the Zynq-7000 family contains the same PS, the PL and I/O resources vary between the devices. The Zynq-7000 architecture enables implementation of custom logic in the PL and custom software in the PS. It allows for the realization of unique and differentiated system functions. The integration of the PS with the PL allows levels of performance that two-chip solutions (e.g., an ASSP with an FPGA) cannot match due to their limited I/O bandwidth, latency, and power budgets.

Xilinx offers a large number of soft IP for the Zynq-7000 family. Stand-alone and Linux device drivers are available for the peripherals in the PS and the PL. The Vivado® Design Suite development environment enables a rapid product development for software, hardware, and systems engineers. Adoption of the ARM-based PS also brings a broad range of third-party tools and IP providers in combination with Xilinx's existing PL ecosystem. The inclusion of an application processor enables high-level operating system support, e.g., Linux. Other standard operating systems used with the Cortex-A9 processor are also available for the Zynq-7000 family.

4.2 FPGA Results

	LUTs	FFs	BRAM	LUTRAM	DSP	BUFG	Max Freq	Power
Baseline Architecture	327	539	1	3	12	1	151.05MHz	0.187W
Winograd Architecture	1096	1087	1.5	4	16	1	147.058MHz	0.264W

Table 4.1: Winograd Without Kernel Transform (We suppose the Kernel is transformed once offline) and without output transform (4x4 tiles output)

From Table 4.2, we can extract that the $k = 6$ approximate multipliers result in 12% logic resource gain over the exact multiplier without DSP, the $k = 8$ in 30% and the $k = 10$ in 42%.

Table 4.2: Resources of Convolution Components after the Serial to Parallel Adjustment

Baseline 3×3	**LUTs**	**FFs**	**BRAM**	**LUTRAM**	**DSP**	**BUFG**	**MAX FREQ (MHz)**	**POWER (W)**
Fixed 16 Hard Multiplier with DSP	327	539	1	3	12	1	151.05	0.187
Fixed 16 Hard Multiplier w/o DSP	1659	892	1	11	0	1	125.78	0.267
Fixed 16 Radix-4	1941	3132	1	147	0	1	139.86	0.267
Fixed 16 Radix-64 (k=6)	1464	2222	1	127	0	1	148.58	0.248
Fixed 16 Radix-256 (k=8)	1198	1861	1	101	0	1	149.4	0.211
Fixed 16 Radix-1024 (k=10)	982	1482	1	98	0	1	133.34	0.196
Float 16 Vivado IP	1926	4402	1	47	25	1	140.84	0.24
BFP per Layer	699	1177	1	15	9	1	149.5	0.179
BFP per 3x3 Window	1895	2017	1	197	9	1	130.63	0.216
BFP per Layer no DSP	1688	1678	1	15	0	1	147.05	0.222

Table 4.3: Resource Requirements of Ship Detection CNN for Fixed16 Baseline Approach

	Layer 1	**Layer 2**	**Layer 3**	**Layer 4**
Input Channels	3	32	16	64
Channel Size	128×128	64×64	32×32	16×16
Filters	32	16	64	32
Conv. Units (Table 4.1)	96	512	1024	2048
Kernel RAMB36s	0.5	2.5	4.5	8.5
Adder Trees	11 3-In	1 32-in	2 16-in	1 32-in
ReLu Units	11	1	2	1
MaxPool. Units	3	1	1	1
Output RAMB36s	59	8	8	4

4.2.1 Ship Detection CNN

If we consider Ship Detection CNN (tha to perigrapsw sto prohgoumeno section) that is consisted of 4 Convolution Layers of 32, 16, 64 and 32 filters. Target device is ZC702 Evaluation Board. According to Table 4.2 we set up our device in a way to get maximum utilization out of it. Since we have 220 DSP slices available, we calculate our convolution components to get the maximum utilization out of them. Each convolution component use 12 DSP slices. So $220/12 = 18$ components with DSP. Considering LUTs and FFs utilization we have : $327 \times 18 = 5886$ LUTs and $539 \times 18 = 9702$ FFs. There are $53200 - 5886 = 47314$ remaining LUTs and $106400 - 9702 = 96698$ FFs. Since DSP slices are no more available, we use the non-DSP approximate $k = 6$ components in order to build our engine. From Table 4.2 each convolution component use 1198 LUTs and 1861 FFs. There are 47314 remaining LUTs, $47314/1198 = 39.49 = 39$ components. In order to fully utilize the board, we generate 16 components with DSP and 16 components without DSP. Considering resources utilization, when these units are generated they use : $16 \times 327 + 16 \times 1198 = 24400$ LUTs and $16 \times 539 + 16 \times 1861 = 38400$ FFs. As for memory requirements, in order to decrease the memory bandwidth of this design, we make the following adjustment in the convolution units designed. Initially, each convolution unit used 3 18Kbit Block RAMs = 1.5 36Kbit BRAMs.

Adjustments in Serial to Parallel Converter

In the initial design, $Kernel_{Height}$ FIFO queues where chained connected as it was described in Chapter 2. The outputs of these FIFOs were fed into the $Kernel_{Height} \times Kernel_{Width}$ register window. We swapped the row of the Register Window and the FIFOs Queues. By making this swap, Input Data first pass from the Register Window and then they are stored in memory, instead of sliding from the memory to the register window as it happened in the initial design.

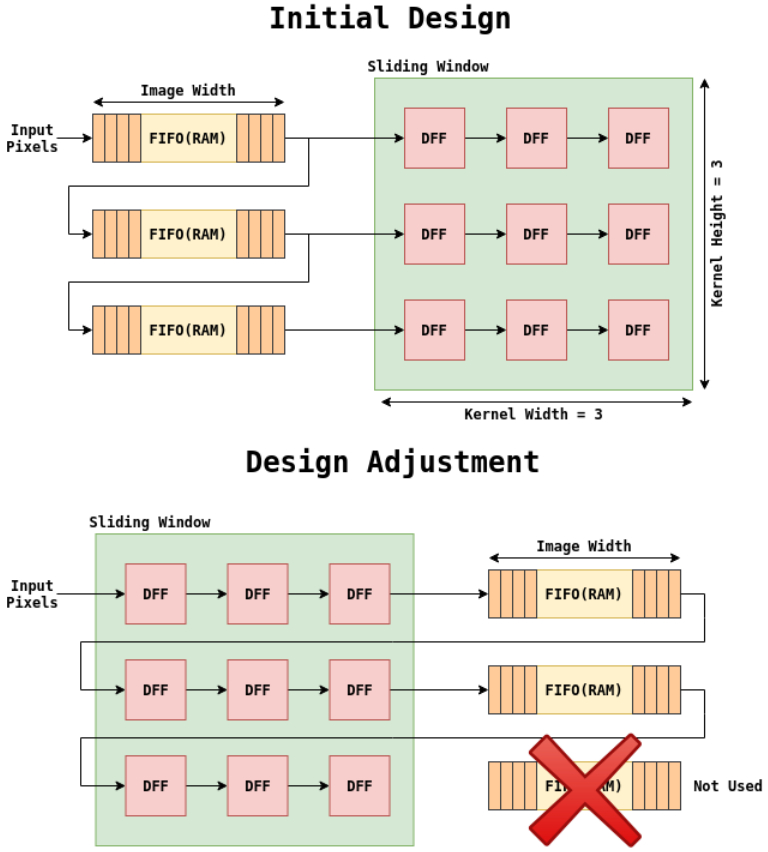


Figure 4.8: Adjustments in Serial To Parallel Converter. One less FIFO memory is required in this design modification.

With this Serial to Parallel adjustment the number we have to generate one less FIFO queue in each unit while we do not have any drawback in performance or resources utilization. Data in the third raw of the sliding window are not going to be reused so they do not need to be stored. With this design, two 18KBit Block RAMs have to be generated in each convolution unit leading to one 36KBit BRAM for each one. The same swap in serial to parallel converted is implemented in the Max-Pool component, reducing the required memory for each channel from one 36Kbit BRAM to one 18KBit BRAM (0.5 36K). In order to generate 32 parallel convolution units, we need $32 \times 1 = 32$ 36KBit BRAMs.

Table 4.4: Sources of the Xilinx Zynq Z-7020 SoC

FPGA chip on Xilinx Zynq Z-7020 SoC	
Logic Cells	**85k**
Look-Up-Tables (LUTs)	**53200**
Flip-Flops	**106400**
DSP	**220**
36Kbit BRAM	**140**

Now we have to take care of extra components that we need in order to properly set up the CNN on the board. We need ReLu and Max-Pooling components on each output channel and adders to properly add input channels. In order to properly calculate needed resources we implement 32 Convolution Components. First Layer of the network has 3 input channels and 32 Filters. Each 3 of the input channels have to be convolved with the proper Kernel and then be summed together to properly form an output channel.

First Layer

To finish with the first layer, $3 \times 32 = 96$ Input Maps must be processed in total. In the way the 32 Engines are set, as its shown in Figure 4.1 10 output channels are generated in parallel. In order to finish with the first layer 32 Output Channels must be generated. Following the above scheme, in the first iteration we finish with the first 10 channels. To deal with 10 output channels, we need 30 convolution units. With the two remaining convolution units on the Engine, we process two input channels of the 11^{th} filter and we produce their partial sum. We store these outputs in the memory, and we repeat the exact same operation with the following 10 filters, so we have 21 fully completed output channels and one of the three channels processed for another filter ready in total. In the third iteration, 10 more output channels are produced in the first 30 convolution units and two more input channels are processed in the 31^{st} and 32^{nd} convolution units. In total, in three iterations 32 output maps are generated. As described in Chapter 2 Max-Pooling component can operate with 4 Channels to be fully utilized. So to produce 11 output channels at maximum (2^{nd} and 3^{rd} iteration), we need 11 ReLus and 3 Max-Pooling Components. The 3^{rd} Max-Pooling component will deal with 3 channels instead of 4. In order the channels to be added together by 3 we need 10 Adder-Trees of 3 and one adder of two. That results to $10 \times 2 + 1 = 21$ Adders of 2 inputs. Max-Pool and ReLu are pipelined to the convolution operation. Once data are Max-Pooled, they can be stored in Memory. To store the first layer in memory, we need : $32 \times 64 \times 64 \times 16 - bit = 2097152$ bit. ZC702 has 36Kbit BRAMs. We need $2097152/36000 = 59$ BRAMs to store the whole output of the first layer in memory. Totally after the first channel we have 32 BRAMs from the convolution units and 59 BRAMs to properly store the output data. That concludes to totally 91/140 BRAMs that the device offers.

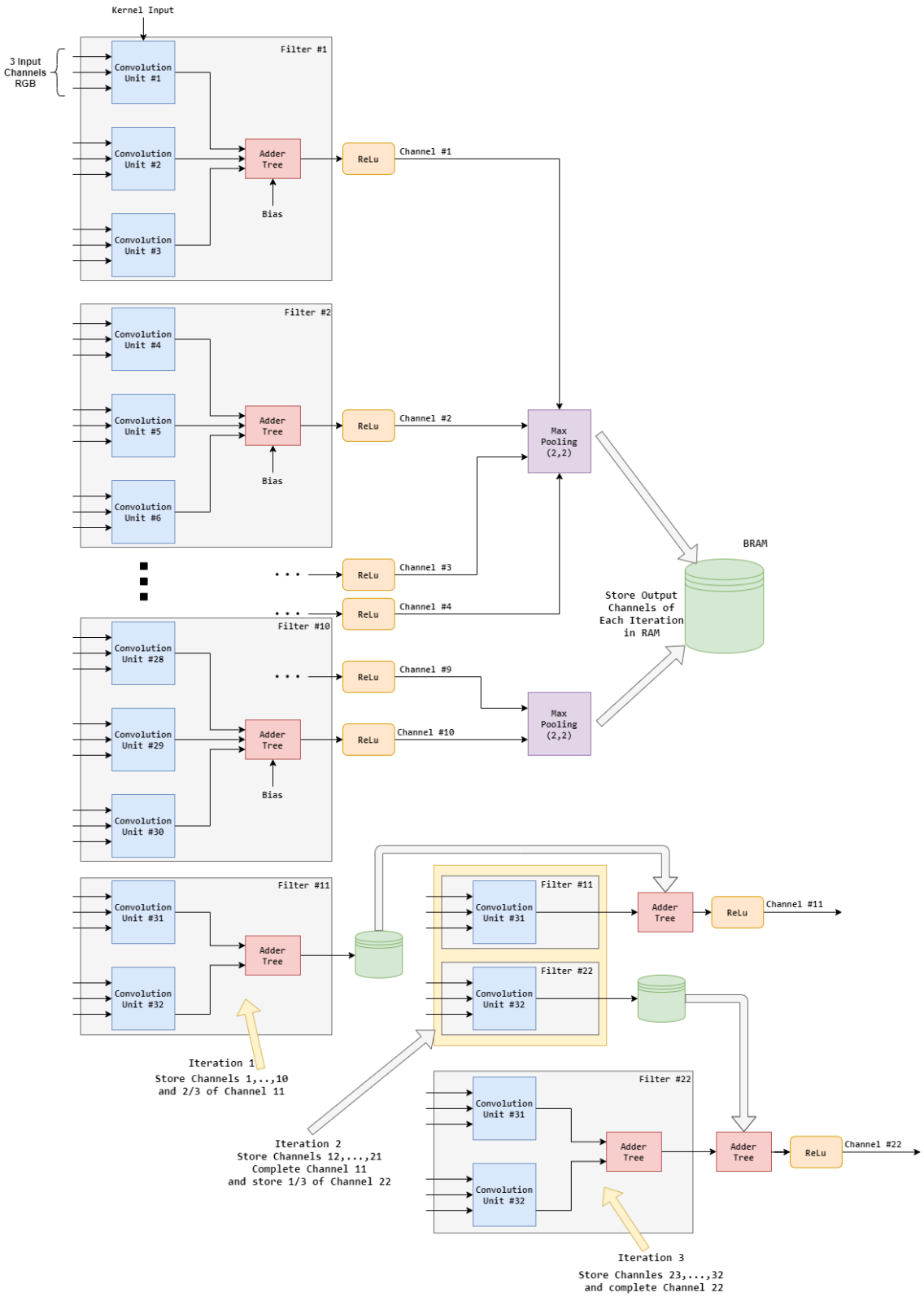


Figure 4.9: First Layer of Ship Detection CNN. 32 Convolution Engines are set in parallel. Three iterations are required to fully complete the first Layer. In First Iteration Channels 1 to 10 are generated and stored in memory while Channel 11 needs one more Input Channel to be processed in order to be completed. In each of the following two iterations, 10 new output Channels are generated in parallel and the other two are semi processed to totally form 32 output channels.

Second Layer

In the second Layer, 32 Input Maps of size 64×64 are fed to its input. Second Layer has 16 Filters, so $32 \times 16 = 512$ convolution operations have to be calculated. In this layer, in contrast to the first layer instead of adding outputs by 3, now we have to add them by 32. In each pass for the second layer, we suppose that 32/33 convolutions units are used. In each pass of the second layer one output channel is generated. In order to add the results of each one of the 32 Input Maps, we need one Adder-Tree of 32 numbers. This adder tree is consisted of 31 adders of two. Since one channel is produced in each pass, we use one uder-utilized by a factor of 1/4 max-pooling component for this layer. In order to complete second layer we need to pass the Engine 16 times. To store the output data of the second layer in memory we need $16 \times 32 \times 32 \times 16 - bit = 262144bit$. In terms of BRAMs we need $262144/36000 = 8$ BRAMs.

Third Layer

In the third layer, 16 Input Maps of size 32×32 are given as input. Third layer has 64 filters, so $64 \times 16 = 1024$ convolution operations have to be calculated. In order to properly add the 16 Input channels, using 32 convolution units in parallel we need two Adder-Trees of 16 inputs each one, leading to $2 \times 15 = 30$ adders of two. Since two channels are generated in each pass of the engine, we use one Max-Pool component under-utilized by a factor of 1/2. In order to fully complete the third layer that has 64 output channels we should use the Engine $64/2 = 32$ times. Considering memory requirements, to fully store the output of the third channel, we need $64 \times 16 \times 16 \times 16 - bit = 262144$ bit, leading to eight 36kBit Block RAMs.

Fourth Layer

Fourth Layer, has 32 Filters and 64 Input Map channels fed to its input. $32 \times 64 = 2048$ convolution operations have to take place in this layer. Since 64 Input channels have to be summed together and we use 32 convolution components, we have to partially compute the sum of the 32 first input channels, store it in the memory and then add it to the sum of the following 32 channels that are generated in the following pass in order to properly form an output channel. One output channel is generated every two passes of the engine, so in order to generate 32 output channels, we need $2 \times 32 = 64$ passes. One Adder-Tree for 32 Numbers is used in each pass (31 Adders of two). Starting from the first pass, we store $16 \times 16 \times 16 - bit = 4096bits$ output in memory. This output has to be summed with the output of the exact following pass to form a valid output channel. During the procedure of calculations that take place in the second pass, the serial outputs produced using an adder of two inputs, are added together and stored in memory. For example, the first pixel from the sum of the first 16 channels that is stored in memory, is directly added to the first output generated in the second pass and replace in position in memory. To properly store the output of the fourth layer, we need $32 \times 16 \times 16 \times 16 - bit = 131072/36000 = 4$ BRAMs.

Engine Requirements

In order to implement the design described in the previous paragraphs, we have to sum up the required resources. We need to generate 32 Convolution Units, 31 Adders of two, 3 Max-Pooling components and 11 ReLus. In terms of Memory, each Max-Pool component need 2 BRAMs to deal with 4 channels in parallel. As it's shown in Table 4.5 Max-Pool component has 240 LUTs and 343 FFs. So 3×240 =720 LUTs and 3×343 =1029 FFs.

Table 4.5: Resources of Extra Components in Convolution Layer

	LUTs	**FFs**	**BRAM**
Max-Pool	240	343	0.5/Channel
ReLu	10	16	0
Adder-Tree	496	496	0

Table 4.6: Total Resources to design the Engine on ZC-702

	LUTs	**FFs**	**DSP**	**BRAMs**
32×Convolution Units	24400	38400	192	32
31 Adders of Two	496	496	0	0
3×Max-Pool	720	1029	0	6
11×ReLu	110	176	0	0
TOTAL	25726	40101	192	38
TOTAL(%)	48.35	37.68	87.27	27.14

Another issue we have to take care of is managing the input weights. In this particular CNN there are $32 \times 3 + 16 \times 32 + 16 \times 64 + 64 \times 32 = 3680$ different 3×3 Kernels. Each Kernel is consisted of 9 pixels with 16-bit length each of them. So in memory requirements we totally need $3680 \times 9 \times 16 - bit = 529920/36000 = 15$ BRAMs to store every Kernel in ROM. ROM in an FPGA can be implemented either as Block RAM or LUTRAM. In order the Engine to properly operate, the following memory system was designed. There is a need to store the input RGB $3 \times 128 \times 128$ Image to broadcast it to the 32 parallel engines. That leads to $3 \times 128 \times 128 \times 8 - bit = 393216 bits/36000 Bit = 11$ RAM Blocks. To reduce latency, we use 32 18Kbit RAM Blocks in order to store the Input Image. Considering the big memory, we use 128 RAM Blocks of 16Kbit. After the data of each layer are produced, are stored in the big memory and after their usage of the next layer free the space they retained in memory. Considering the input weights, instead of using 15 Block Rams we can reduce the number of BRAMs and use LUTRAM instead to a proportion of 5 BRAMs and the equivalent amount of LUTRAM to save these data. According to this analysis we totally need 16(Input Data)+64(Layer Output)+32(Convolution Units)+6(Max-Pool)+15(Kernel Storage)=133 36Kbit RAM blocks. We use 133/140 = 0.95% of the device memory blocks.

Time Required to Properly Operate

As it was analyzed in Chapter 2, each convolution unit needs $(W+2)$ CCs to start giving valid outputs. After this loading time, each clock cycle we have a valid output. Our Input Image has size 128×128. In order to fully process a single input image, we need : $T_{Image_{128x128}} = 2*128+3+128*128 = 16643$ CCs. In order to properly add these outputs by 3 we need 2 extra CCs and 1 extra CC for the ReLu activation. Once these values have passed from the ReLu activation, they have to be Max-Pooled. As was described before, Max-Pool component can deal with four different input channels at once. After the first $[W+2+1(Multiplies)+4(Tree-Add)+2(Add-3-Channels)+1(ReLu)] = W+10$ CCs we have valid outputs. These valid outputs are stored in Max-Pool FIFO memory. In order to operate, Max-Pool need to Load $W+4$ pixels. Once these pixels are stored, in each CC one valid-output from one of the 4 channels is generated. In order to properly Max-Pool 4 channels, $W \times (W+1)$ CCs need to pass. This procedure can be pipelined after the ReLu activation. After $W+10+W+4$ Pixels we have the first Max-Pool operation. Max-Pool need 2 CCs to operate. Totally after $2W+14+2$ we have the first valid Max-Pool output that is stored in memory. The first valid output for the 128×128 Image is generated at $128+10 = 138$ CCs. The first valid output for the max-pool component is generated at $138+128+4+2 = 272$ CCs. After this point $128 \times (128+1) = 16512$ CCs need to pass in order to perform convolution operation on the whole image. As a result, to finish with one iteration of the first layer, 16512 CCs have to pass. As it was described before, we need 3 iterations to finish with the first layer. In total, we need $Time_{Layer_1} = 3 \times 16512 = 49536$ CCs to complete the operations on the first layer and store it in memory.

We repeat the same calculations for the other layers. The second layer gets as input an Image with size $(W/2) \times (W/2) = 64 \times 64$. The difference from first layer, is that now we have to add 32 values in a 32-input Adder-tree. This operation needs 5 CCs. First valid-output of ReLu in :$(W/2)+2+1+4+5+1 = (W/2)+13$ CCs. Max-Pooling for one channel, need $(W/2) \times (W/2)$ CCs to be fully completed after the loading time. So we have $Time_{Image_{64\times64}} = (W/2)+13+(W/2)+4+2+(W/2) \times (W/2) = 4243$ CCs for one iteration of layer 2 to be completed. In order to complete layer 2 we need to repeat the same procedure 16 times, so we totally have: $Time_{Layer_2} = 16 \times Time_{Image_{64\times64}} = 16 \times 4243 = 67408$ CCs. As for third layer, we we have an input image with size : $(W/4) \times (W/4) = 32 \times 32$. Considering data adding, we have to add data by 16. We need 4 CCs and 2 16 input Adder-Trees. $First-Output_{Image_{32\times32}} = (W/4)+2+1+4+4+1 = (W/4)+12$ CCs. ReLu is operating with 2 channles at once now. $Time_{Image_{32\times32}} = (W/4)+12+(W/4)+4+2+(W/4) \times (W/4+1) = 1106$ CCs. We need 32 iterations to complete third layer. $Time_{Layer_3} = 32 \times Time_{Image_{32\times32}} = 32 \times 1106 = 35392$ CCs. Now on fourth layer, we add 32 data at once, but we can't perform Max-Pool before the next iteration's data are generated. In this case, 32 iterations that only perform Convolution and ReLu and other 32 that perform Convolution, ReLu and Max-Pool will take place. If we suppose the first iteration, we need $(W/8)+2+1+4+5+1 = (W/8)+13$ CCs to have

valid outputs from ReLu. These outputs are stored in RAM. To finish the first iteration, we totally need $Time_{Full-NoMax} = (W/8)+13+(W/8)\times(W/8) = 285$ CCs. Once the first iteration is completed, we start generating valid ReLu output after $(W/8)+13$ CCs again. These data generated from the second iteration, are directly added to the data stored in memory from the first iteration. So we need 1 extra CC to add 2 Values together and pass them to the Max-Pool component. Now in exaclty the same way as the previous layers, we need $Time_{Full-Max} = (W/8)+13+(W/8)+4+2+1+(W/8)\times(W/8) = 308$ CCs. As a result we have the time to fully complete the final convolutional layer : $Time_{Layer_4} = 32 \times Time_{Full-NoMax} + 32\times_{Full-Max} = 32 \times 285 + 32 \times 308 = 18976$ CCs. Finally from the previous analysis, we can extract the final time our Engine needs to perform all the computations of the convolutional layers of this specific CNN.

$$Time_{Total} = Time_{Layer_1} + Time_{Layer_2} + Time_{Layer_3} + Time_{Layer_4}$$
$$Time_{Total} = 49536 + 67408 + 35392 + 18976 = 171312CCs$$

If we want to find FPS, we suppose that clock frequency $\approx$ 125MHz, so clock period $\approx$ 8ns. So $Time_{Total} = 171312 \times 8ns = 1.370496ms$.

$$FPS = \frac{1s}{1.370496ms} = 730$$

4.2.2 Winograd Implementation

In this section, the equivalent implementation of fixed point 16 with Winograd convolution units will be further analyzed. Each Winograd component can deal with 4 input channels.

Table 4.7: Resources of Winograd Convolution Components with Output Transform and Kernel Transform

Winograd 3x3	**LUTs**	**FFs**	**BRAM**	**LUTRAM**	**DSP**	**BUFG**	**MAX FREQ (MHz)**	**POWER (W)**
Fixed 16 Hard Multiplier+DSP	1096	1027	2	4	16	1	147.06	0.264
Fixed 16 Hard Multiplier no DSP	1985	1488	2	4	0	1	123.45	0.302
Fixed 16 Radix-64 (k=6)	2090	4168	2	91	0	1	131.57	0.277
Fixed 16 Radix-256 (k=8)	1791	3616	2	132	0	1	129.8	0.266
Fixed 16 Radix-1024(k=10)	1460	2817	2	132	0	1	128.2	0.256
BFP per Layer	699	1177	1.5	15	16	1	149.5	0.280
BFP per Layer no DSP	1688	1717	6	15	0	1	91.32	0.310

Following the same extrapolation technique as we did with Baseline (Normal) convolution, we calculate the total resources to set up the CNN on our target board. Considering DSP resources, each Winograd convolution unit requires 16 DSPs and we have 220 DSP slices available. We can generate $220/16 = 13$ Winograd components with DSP at maximum. As it was described in chapter two, each Winograd component can compute four different input channels. By that means, if 13 components are generated, 52 channels can be computed in parallel. In order to compute 32 Input Channels at each time using Winograd, we must generate $M/4 = 32/4 = 8$ components. Considering resources utilization, $8 \times 1096 = 8766$ LUTs, $8 \times 1027 = 8216$ FFs and 128 DSPs are used. As it was analyzed in Chapter 2, in order to produce valid outputs for the first layer, considering we have 3 input channels and 32 filters, we have to properly manage the channels and the filters to fully utilize the engine. In order to properly deal with one channel output, we need 16 adders of two inputs to add two consecutive 4×4 tiles, 24 to transform from 4×4 to 2×2, 4 adders of two in order to add two 2×2 tiles and other 4 to add the bias to the 4 elements produced to the output. Summed up, we need 48 adders of two in total for each output channel. In Layer 1 that 10 output channels will be produced in parallel, there is a need of $10 \times 48 = 480$ adders. In order to store the first sums of tiles a buffer with size $(W/2) \times 2 \times 2 \times 16 - bit = 64 \times 2 \times 2 \times 16 - bit = 4096bits$ needs to be implemented for each of the 10 output channels. Considering Channel 11 as it happened with baseline architecture, the 11^{th} channel will be 2/3 generated so it needs 16 adders of two and 24

to transform. That leads to 40 adders of two. Channel 11 does not need to be stored in buffer but in main memory. As for Max-Pooling, since output tiles are 2×2, they can be straight up compared without serial to parallel converter and find the maximum of each 4 elements. 4 ReLus have to be generated for each output channel since outputs are 2×2, leading to 44 ReLus for Layer 1. Totally we need 480 adders of two, 11 Max-Pooling comparators of 4 elements and 44 ReLus. In Layer 2, we need $N \times [4 \times M + 32] = 160$ adders of two, 4 ReLus for each output tile and one Max-Pooling component. Following the same

Table 4.8: Total Resources to Design Winograd Engine on ZC-702

	LUTs	**FFs**	**DSP**	**BRAMs**
8×Winograd Units	8766	8216	128	48
8×Control Units	∼2000	∼2000	0	0
480 Adders of Two	7680	7680	0	0
11×Max-Pool	620	620	0	0
11×ReLu	440	440	0	0
TOTAL	19506	18956	128	48
TOTAL(%)	36.67	17.815	58.18	34.28

technique, we analyzed Layers 3 and 4 and we found that the maximum components we need in order the Engine to operate properly are at maximum: 480 adders of two, 11 Max-Pooling components and 11 ReLus. In addition, for each Winograd Component to deal with 4 input channels simultaneously there is a need of a control unit for each component that consumes about 300 LUTs and FFs. Finally as it was firstly mentioned in our theoretical analysis, LUTs utilization is reduced by $\frac{48.35-36.76}{48.35}\% = 23.97\%$. DSP utilization is significantly reduced, since instead of 36 multiplications, only 16 multiplications are performed for each 2×2 output.

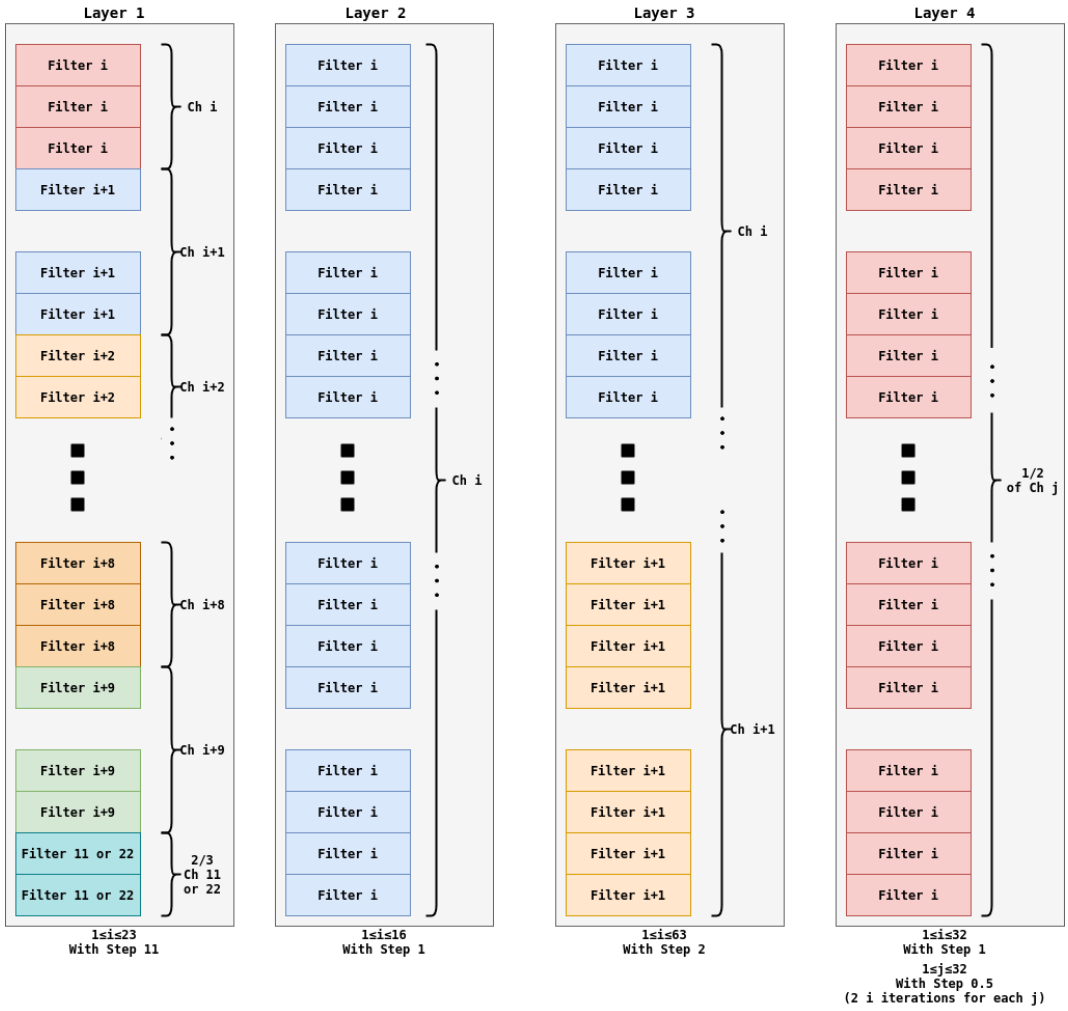


Figure 4.10: Winograd Engine with 32 parallel units for each Layer. 3, 16, 32 and 64 iterations to finish each layer respectively. Filter Management for each Layer is presented.

Since Winograd convolution components deal with 4×4 input tiles, 3 FIFOs are required (after the serial to parallel adjustment) in order to feed input data to the computation component. For 32 Input Channels, we need $32 \times 1.5 = 48$ 36Kbit RAM Blocks instead of 32 that were used in Baseline Architecture. We implement the same memory system, as we did in Baseline. In total we need $48 + 16 + 64 + 15 = 143$ RAM blocks. Since RAM blocks are over-utilized, as it was mentioned before, since Kernels are stored in ROM memory, we can use available LUTRAM in order to store 1/3 of the input Kernel there. After this modification, 138/140 RAM Blocks (98.57%) are used.

4.3 Floating Point Architecture

As it was analyzed in Chapter 3, floating point architecture offers higher precision that is really significant in multi layer Convolutional Neural Networks. This higher accuracy comes at the cost of higher computational and logical cost.

4.3.1 Vivado Floating Point Architecture with IPs

The first approach, was to implement a convolution unit with Vivado Floating Point operations IPs. In order to do so, for each unit, 9 floating point multipliers and 9 floating point adders are necessary. In order to add M Input Channels to produce an output channel $M-1$ floating point adders need to be generated. Considering resources utilization, for Baseline architecture, as it is presented in Table 4.2, each convolution unit for one Input Channel consumes 1926 LUTs, 4402 FFs and 25 DSPs. With the same extrapolation technique we followed in previous Engines, we generate $220/25 = 8$ convolution units in parallel. In the worst case that all of the eight input channels are added together, we need an Adder-Tree of 8 inputs. That leads to 7 Adders of two. Each Floating Point adder consumes 131 LUTs, 304 FFs and 2 DSPs.

Table 4.9: Total Resources to Design Floating-Point Engine on ZC-702

	LUTs	**FFs**	**DSP**	**BRAMs**
8×Floating-Point Units	15408	35216	200	32
8×FP Adders of two	917	7680	16	0
1×Max-Pool	240	343	0	1
3×ReLu	30	30	0	0
TOTAL	16595	40569	216	33
TOTAL(%)	36.67	38.13	91	23.57

Following the same technique with Fixed Point architectures, we calculate iterations required for each layer to be fully completed. Memory requirements are exactly the same with fixed point 16-bit architecture. We use exactly the same memory system. Totally, 16+64+33+15 = 128/140 (91.42%) 36Kbit RAM blocks are used. As for time analysis, if T_{128} is the time needed to operate 128×128 convolution and max-pooling of the first layer, then we can extract the total time :

$$T_{64} = \frac{T_{128}}{2} \quad , \quad T_{32} = \frac{T_{64}}{2} = \frac{T_{128}}{4} \quad T_{16} = \frac{T_{32}}{2} = \frac{T_{64}}{4} = \frac{T_{128}}{8}$$

$$T_{TOTAL} = 12 \times T_{128} + 64 \times T_{64} + 128 \times T_{32} + 256 \times T_{16}$$

$$T_{TOTAL} = (12 + 32 + 32 + 32) \times T_{128} \approx 108 \times T_{128}$$

According to the time analysis done in the previous chapter for Fixed Point architecture, it is obvious that since we reduced the parallel convolution units from 32 to 8 (/8), that frames per second (FPS) are decreased by a factor of four.

4.3.2 Block Floating Point Architecture

Since the architecture that used Floating Point Xilinx IPs had the restriction of high resource utilization, we use Block Floating Point architecture as described in Chapter 3 in order to increase maximum throughput for the target device. Each BFP convolution unit consumes 699 LUTs, 1177 FFs and 9 DSPs. We follow a similar architecture as the Fixed Point Architecture. If we generate 32 convolution units in parallel, 16 with DSP and 16 without DSP then $16 \times 699 + 16 \times 1688 = 38192$ LUTs, $16 \times 1177 + 16 \times 1678 = 45680$ FFs and $16 \times 9 = 144$ DSPs are used. Since each output channel share a common exponent, floating point addition can be thought as an integer addition of 8-bit mantissas for each input channel. Thus, according to the analysis for Fixed-Point we need 31 Adders of two 8-bit integers.

Table 4.10: Total Resources to Design BFP Engine on ZC-702

	LUTs	**FFs**	**DSP**	**BRAMs**
32×BFP Units	38192	45680	144	32
31×8-bit Adders of two	248	248	0	0
3×Max-Pool	720	1029	0	6
11×ReLu	110	176	0	0
TOTAL	39720	47133	144	38
TOTAL(%)	74.66	44.29	65.45	27.14

Iterations needed to fully complete all the layers of the network, are exactly the same as the fixed point architecture. In this way, we can extract:

$$T_{TOTAL} = 3 \times T_{128} + 16 \times T_{64} + 32 \times T_{32} + 64 \times T_{16}$$

$$T_{TOTAL} = (3 + 8 + 8 + 8) \times T_{128} = 27 \times T_{128} T_{TOTAL_{BFP}} = \frac{T_{TOTAL_{FP}}}{4}$$

The main advantage of the Block Floating Point Architecture is that with small sacrifice in precision we can maintain high Frames Per Second (FPS) as we were working with fixed point arithmetics.

4.3.3 Place & Route of complete networks in XCZ7020

The results in previous section were extracted via extrapolation of the resources we obtained from the implemented results of each component. Vivado was used to properly set up the whole convolution engine and validate if the specific designs can be routed and placed in Zynq-7020 Evaluation Board. The proposed techniques were successfully routed and placed.

Table 4.11 shows the results of 4 'Proposed' implementations with either floating or fixed point arithmetic. They are compared to 'Typical' implementations using only default multipliers. The typical FP16 CNN achieves maximum parallelization only 8x due to increased cost and constraints of the Xilinx IP FPU (it relies on DSPs). The typical FP16 implementation achieves similar throughput as our mobile GPU test Jetson Nano. However, with the proposed BFP architecture we achieve 4x higher throughput with negligible accuracy loss. For fixed point techniques the proposed Winograd implementation led to a 40% reduction in terms of LUTs retaining the same percentage in DSP usage.

Table 4.11: Final CNN performance with proposed techniques (Zynq-7020)

configuration	paral.	**LUT**	**DSP**	**RAMB**	**MHz**	**FPS**
Typical Float.Point	8	37%	91%	78%	125	182
Prop. Block.Fl.Pt.	32	65%	100%	95%	125	730
Prop. B.Fl.P.WGD.	8x4	68%	59%	95%	124	724
Typical Fixed.Point	32	69%	58%	95%	118	689
Prop. Fixed.Point	32	60%	54%	95%	124	724
Prop. Fix.P.WGD.	8x4	41%	58%	95%	112	654

(loss is ∼0.2% between configurations)

4.3.4 Comparisons to other devices

Compared to other competitive embedded devices, the proposed implementation achieves 10x better performance/Watt than Jetson Nano GPU (204 FPS via tensorflow/cuda acceleration) and 2.5x better performance/Watt than Myriad2 DSP (105 FPS via C/C++ coding). Zynq's power consumption is 2.5W for f clk =125MHz (plus 1W for the board). We also note that the Fully Connected layers of our CNN are executed by the processor ARM Cortex-a9 of Zynq (1.2 ms to fully execute all the FC layer operations), in parallel to the CNN's convolutions, via AXI stream PS-PL communication, in only 1msec (such time is masked during batch processing). The acceleration achieved is 438x vs Zynq's ARM and 6.5x vs a single-threaded desktop Intel core i7-8700.

4.4 Accuracy Evaluation

In order to test all the architectures we described in the previous section, we made modifications in Evangelos Petrogonas CNN Engine in C. This Engine was initially designed to perform the convolution operation in C for Float data and weights. We made the appropriate modifications to this engine in order to operate with different types of data (Fixed Point, Custom Floating Point Notations) and to operate approximate multiplications instead the "classic" multiplication operator ($*$). The main purpose is to calculate the classification accuracy of each CNN and how the different approaches made to the Engine affected it. The following models that are going to be presented were initially trained with TensorFlow Keras with single precision float typical notations.

Table 4.12: Accuracy Tests for Different Fixed Point (FP) Architecture

	TensorFlow	**FP 16**	**FP 16 k=6**	**FP 16 k=8**	**FP 16 k=10**
Ship Detection	96.8%	96.8%	96.75%	96.65%	93.6%
MNIST	98.45%	98.45%	98.45%	98.45%	98.1%
CIFAR-10	75.5%	75.5%	75.35%	75.1%	71%

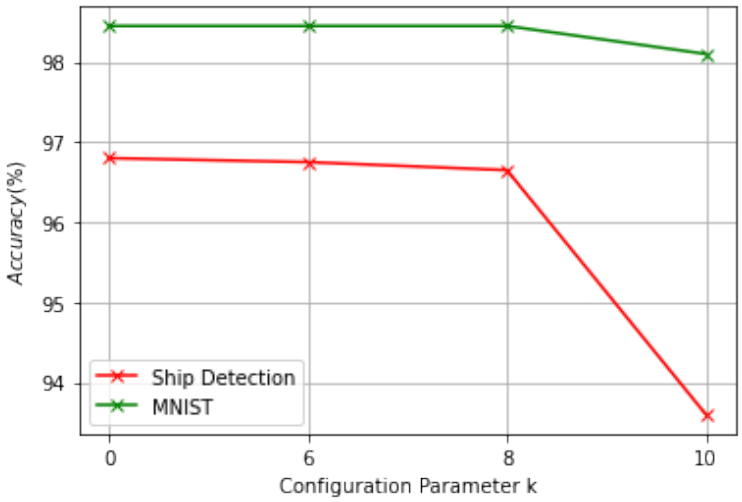


Figure 4.11: Classification Accuracy is affected by configuration parameter k

Table 4.13: Accuracy Tests for Different Floating Point Architectures

	Float 16	**Float 14**	**Float 12**	**Float 16 Appr.**	**Float 14 Appr.**	**Float 12 Appr.**
Ship Detection	96.8%	96.3%	95.4%	96.8%	96.5%	96.7%
MNIST	98.45%	97.5%	97.15%	98.45%	98.1%	97.2%
CIFAR-10	75.5%	75.45%	70.3%	75.5%	75.4%	70%

As mentioned in Chapter 3, multiplication in floating point notations, is a multiplication between Mantissas with the appropriate adjustments to the exponent. The Approximate Multiplier was initially designed for 16-bit length numbers. This multiplier was adjusted to 10-bit and 8-bit multiplication architectures with configuration parameter k=4. This modification does not offer a significant reduction in hardware utilization, since only one Radix-4 (Modified Booth) multiplier is replaced. Accuracy using k=4 in 8-bit and 10-bit formats was not significantly reduced. Another type of approximate multiplier that is especially designed for smaller length numbers could be used in order to further reduce resources utilization in small floating point format multiplications.

Table 4.14: Accuracy Tests for Block Floating Point Architectures

	10-bit Mantissa	**8-bit Mantissa**	**BFP 10-bit with Approx**
Ship Detection	96.8%	96.7%	96.7%
MNIST	98.45%	98.45%	98.3%
CIFAR-10	75.5%	75.3 %	75.25%

The $k = 10$ approximate was not appropriate for our range of numbers. As it is analyzed in [41] when the number B encoded is in $[-1500, 1500]$ RAD1024 exhibits RED (Relative Error Distance) more than 10%. Given that the majority of the input/output values between different layers of the specific CNNs were in the range $[-2, 2]$ their integer value was in the intervals that RAD1024 exhibits a significant error. To properly integrate the $k = 10$ multiplier in a CNN design, we should carefully place it in parts of the network that the integer representations of the fixed point values are out of this interval. This can be done at the first layers of a given network and it could offer high logic resources gains compared to the accurate multiplier. Weights can not be encoded by this multiplier, since the majority of their values are close to zero.

The maximum accuracy loss is observed for the CIFAR-10 fixed point with configuration parameter k=10 (4.5%). All the other proposed architectures maintain high accuracy. In any other proposed architecture the highest accuracy loss is 0.2% considering fixed point (with maximum k=8) and 1% for custom floating point architectures (5-bit exponent, 6-bit mantissa).

Chapter 5

Conclusion

This thesis main goal was to design generic components that can fully construct a desired CNN in a FPGA board. The usual approach in CNN inference today, is to compress the model using pruning or quantization techniques. These techniques can significantly reduce the model's size and address the over-fitting problem, but it is necessary to re train the network after the compression modifications. The main advantage of our design is for a given network with standard 8/16/32-bit formats, to reduce resources utilization without modifying the original model and hence further accelerate the network's performance and reduce consumption.

The engine in its current form, to efficiency use approximation multiplier without serious drawbacks in accuracy supports data with length greater than 16-bit. Given that the engine is fully generic considering data length and type, the constructed engine without multiply approximations can properly operate for any given data type and length but without using approximations. The combination of the Winograd convolution algorithm with the approximate multipliers further reduced the resources utilization making the engine even more efficient, offering a margin to parallelize even more convolution components to increase throughput or save resources to use them in other operations. As for floating point notations, the block floating point arithmetic is already considered as an approximation but we can use multiplication approximation too to reduce resources utilization. More specifically with the proposed Winograd fixed point engine, we achieved a logic resources reduction of 40% with a slight DSP increase of 7% compared to the baseline approach. As for Block Floating Point, the baseline approach could only be implemented on ZC702 with 100% DSP utilization to parallely operate 32 engines. With the proposed Winograd BFP approach, we managed to reduce DSP utilization by 42% and only increase the logic resources by 4%. The Winograd BFP can offer quadruple throughput compared to the Typical Floating Point engine that is constructed by Xilinx floating point IPs. These proposed solutions degraded classification accuracy by 0.2% at maximum.

The engine constructed achieved 10× greater performance/Watt compared to Jetson Nano GPU and 2.5× better performance than Myriad 2 DSP. Compared to a high end desktop CPU (Intel i7 8700) it achieved 6.5× better performance and 438× compared to

Zynq's ARM.

5.1 Future Work

- **Extend the approximations in more operations inside the CNN**
  As it was mentioned in introduction, there is a huge amount of research in the field of approximate adders. In the baseline convolution component a significant percentage of LUTs utilization is used for add/sub operations. The basic Vivado adder operation can be replaced with an appropriate approximate adder that fulfils the design's goals. The main goal is to maintain a high classification accuracy but depending on different engines, adders that satisfy even more criteria must be examined. For example in a certain problem we may need to target low power consumption and in another project we may target low LUT utilization. The selection of the appropriate approximate adder is up to the user and the optimization problem and every adder can be used since it does not highly downgrade the classification accuracy. The main purpose of approximate computing, is to insert approximate operations in almost every aspect of the design. In this direction, approximate components that perform the ReLu and Max-Pooling operations can be designed. Since memory was a main obstacle in our approach to further increase component parallelizations, modifications leading to a more efficient memory system can be implemented.

- **Evaluate Approximations on Different Type of Networks**
  The approximate multiplier used in CNNs, can be tested in different types of state of the art neural networks, such as Recurrent Neural Networks (RNN), Long-Short Term Memory Networks (LSTM) in order to evaluate how their behavior is affected when one or multiple types of approximations are inserted in their structure.

- **Explore more approximations algorithm to also exploit small numbers**
  Since network compression and quantization is a common technique to reduce a model's size, the engine designed should support any CNN model that is given as input. In order to support small data sizes and offer significant advantages, an approximate multiplier that is proper for small bit length numbers must be designed or inserted in the current design. For example, with the selection of an appropriate multiplier, the user of the engine can get advantages for every type of model even with 7-bits or less in weights representation. This multiplier can be used in floating point notation when the mantissas of two numbers are multiplied. Considering single precision float (23-bit mantissa), the current multiplier used in this thesis (High Radix Hybrid Multiplier) is adequate. Although, for half precision floating point format (10-bit mantissa) or for custom representations (e.g. 14-bit with 8-bit mantissa) another multiplier or a combination of approximate adders multiplication should be implemented. Once the proposed approximate engine design fulfils all these requirements, the user can implement any type of compressed network with

custom data lengths and types, taking advantage of the throughput increase and resources reduction.

Bibliography

[1] D. Amodei, S. Ananthanarayanan, R. Anubhai, J. Bai, E. Battenberg, C. Case, J. Casper, B. Catanzaro, Q. Cheng, G. Chen, J. Chen, J. Chen, Z. Chen, M. Chrzanowski, A. Coates, G. Diamos, K. Ding, N. Du, E. Elsen, and Z. Zhu. Deep speech 2: End-to-end speech recognition in english and mandarin. 12 2015.

[2] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. *ArXiv*, 1409, 09 2014.

[3] L. Bai, Y. Zhao, and X. Huang. A cnn accelerator on fpga using depthwise separable convolution. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 65(10):1415–1419, 2018.

[4] W. Chen, J. Wilson, S. Tyree, and K. Weinberger. Compressing neural networks with the hashing trick. *Compressing Neural Networks with the Hashing Trick*, 04 2015.

[5] P. Düben, J. Schlachter, Parishkrati, S. Yenugula, J. Augustine, C. Enz, K. Palem, and T. Palmer. Opportunities for energy efficient computing: a study of inexact general purpose processors for high-performance and big-data applications. 03 2015.

[6] D. Elam and C. Iovescu. A block floating point implementation for an n-point fft on the tms 320 c 55 x dsp. 2003.

[7] D. Esposito, A. G. M. Strollo, E. Napoli, D. De Caro, and N. Petra. Approximate multipliers based on new approximate compressors. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 65(12):4169–4182, 2018.

[8] B. Grigorian, N. Farahpour, and G. Reinman. Brainiac: Bringing reliable accuracy into neurally-implemented approximate computing. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 615–626, 2015.

[9] K. Guo, L. Sui, J. Qiu, J. Yu, J. Wang, S. Yao, S. Han, Y. Wang, and H. Yang. Angel-eye: A complete design flow for mapping cnn onto embedded fpga. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(1):35–47, 2018.

[10] V. Gupta, D. Mohapatra, A. Raghunathan, and K. Roy. Low-power digital signal processing using approximate adders. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 32:124–137, 01 2013.

[11] L. C. D. Hadi Esmaeilzadeh, Adrian Sampson. Architecture support for disciplined approximate programming. In *ASPLOS XVII: Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, page 301–312, March 2012.

[12] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[13] J. Huang, J. Lach, and G. Robins. A methodology for energy-quality tradeoff using imprecise hardware. In *DAC Design Automation Conference 2012*, pages 504–509, 2012.

[14] H. Jiang, J. Han, F. Qiao, and F. Lombardi. Approximate radix-8 booth multipliers for low-power and high-performance operation. *IEEE Transactions on Computers*, 65(8):2638–2644, 2016.

[15] S. Kala, B. R. Jose, J. Mathew, and S. Nalesh. High-performance cnn accelerator on fpga using unified winograd-gemm architecture. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(12):2816–2828, 2019.

[16] Khaing Yin Kyaw, Wang Ling Goh, and Kiat Seng Yeo. Low-power high-speed multiplier for error-tolerant application. In *2010 IEEE International Conference of Electron Devices and Solid-State Circuits (EDSSC)*, pages 1–4, 2010.

[17] A. Krizhevsky, I. Sutskever, and G. Hinton. Imagenet classification with deep convolutional neural networks. *Neural Information Processing Systems*, 25, 01 2012.

[18] P. Kulkarni, P. Gupta, and M. Ercegovac. Trading accuracy for power with an underdesigned multiplier architecture. In *2011 24th Internatioal Conference on VLSI Design*, pages 346–351, 2011.

[19] A. Lavin and S. Gray. Fast algorithms for convolutional neural networks. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4013–4021, 2016.

[20] V. Leon, K. Asimakopoulos, S. Xydis, D. Soudris, and K. Pekmestzi. Cooperative arithmetic-aware approximation techniques for energy-efficient multipliers. In *Proceedings of the 56th Annual Design Automation Conference 2019*, DAC '19, New York, NY, USA, 2019. Association for Computing Machinery.

[21] V. Leon, S. Mouselinos, K. Koliogeorgi, S. Xydis, D. Soudris, and K. Pekmestzi. A tensorflow extension framework for optimized generation of hardware cnn inference engines. In *MDPI Technologies*, volume 8, pages 1–15, 2020.

[22] V. Leon, G. Zervakis, D. Soudris, and K. Pekmestzi. Approximate hybrid high radix encoding for energy-efficient inexact multipliers. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 26(3):421–430, 2018.

[23] V. Leon, G. Zervakis, S. Xydis, D. Soudris, and K. Pekmestzi. Walking through the energy-error pareto frontier of approximate multipliers. *IEEE Micro*, 38(4):40–49, 2018.

[24] X. Lian, Z. Liu, Z. Song, J. Dai, W. Zhou, and X. Ji. High-performance fpga-based cnn accelerator with block-floating-point arithmetic. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(8):1874–1885, 2019.

[25] X. Lian, Z. Liu, Z. Song, J. Dai, W. Zhou, and X. Ji. High-performance fpga-based cnn accelerator with block-floating-point arithmetic. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(8):1874–1885, 2019.

[26] M. Lin, Q. Chen, and S. Yan. Network in network. 12 2013.

[27] C. Liu, J. Han, and F. Lombardi. A low-power, high-performance approximate multiplier with configurable partial error recovery. In *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1–4, 2014.

[28] W. Liu, L. Qian, C. Wang, H. Jiang, J. Han, and F. Lombardi. Design of approximate radix-4 booth multipliers for error-tolerant computing. *IEEE Transactions on Computers*, 66(8):1435–1441, 2017.

[29] M.-T. Luong, H. Pham, and C. Manning. Effective approaches to attention-based neural machine translation. 08 2015.

[30] Y. Ma, Y. Cao, S. Vrudhula, and J. Seo. Performance modeling for cnn inference accelerators on fpga. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(4):843–856, 2020.

[31] H. R. Mahdiani, A. Ahmadi, S. M. Fakhraie, and C. Lucas. Bio-inspired imprecise computational blocks for efficient vlsi implementation of soft-computing applications. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 57(4):850–862, 2010.

[32] K. Majumder and U. Bondhugula. A flexible fpga accelerator for convolutional neural networks, 12 2019.

[33] M. Masadeh, O. Hasan, and S. Tahar. Comparative study of approximate multipliers, 2018.

[34] S. Mouselinos, V. Leon, S. Xydis, D. Soudris, and K. Pekmestzi. Tf2fpga: A framework for projecting and accelerating tensorflow cnns on fpga platforms. In *2019 8th International Conference on Modern Circuits and Systems Technologies (MOCAST)*, pages 1–4, 2019.

[35] D. T. Nguyen, T. N. Nguyen, H. Kim, and H. Lee. A high-throughput and power-efficient fpga implementation of yolo cnn for object detection. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(8):1861–1873, 2019.

[36] Shih-Lien Lu. Speeding up processing with approximation circuits. *Computer*, 37(3):67–73, 2004.

[37] C. Szegedy, Wei Liu, Yangqing Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9, 2015.

[38] J. Wang, J. Lin, and Z. Wang. Efficient hardware architectures for deep convolutional neural network. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 65(6):1941–1953, 2018.

[39] T. Yang, T. Ukezono, and T. Sato. Design of a low-power and small-area approximate multiplier using first the approximate and then the accurate compression method. In *Proceedings of the 2019 on Great Lakes Symposium on VLSI*, GLSVLSI '19, page 39–44, New York, NY, USA, 2019. Association for Computing Machinery.

[40] Y. Yu, C. Wu, T. Zhao, K. Wang, and L. He. Opu: An fpga-based overlay processor for convolutional neural networks. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 28(1):35–47, 2020.

[41] G. Zervakis, S. Xydis, K. Tsoumanis, D. Soudris, and K. Pekmestzi. Hybrid approximate multiplier architectures for improved power-accuracy trade-offs. In *2015 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pages 79–84, 2015.

[42] Z. Zhang, Y. He, J. He, X. Yi, Q. Li, and B. Zhang. Optimal slope ranking: An approximate computing approach for circuit pruning. In *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–4, May 2018.

[43] N. Zhu, W. Goh, and K. S. Yeo. An enhanced low-power high-speed adder for error-tolerant application. pages 69 – 72, 01 2010.