ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Μικροϋπολογιστών και Ψηφιακών Συστημάτων

Acceleration of Image Recognition on Caffe framework using FPGAs

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Δημήτριος Ν. Δανόπουλος

Επιβλέπων: Δημήτριος Ι. Σούντρης
Αναπληρωτής Καθηγητής

Αθήνα, Μάιος 2018

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Μικροϋπολογιστών και Ψηφιακών Συστημάτων

Acceleration of Image Recognition on Caffe framework using FPGAs

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Δημήτριος Ν. Δανόπουλος

Επιβλέπων: Δημήτριος Ι. Σούντρης
Αναπληρωτής Καθηγητής

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 18η Μαΐου 2018 .

....	....	....
Δ. Σούντρης	Κ. Πεκμεστζή	Γ. Γκούμας
Αναπληρωτής Καθηγητής	Καθηγητής	Επίκουρος Καθηγητής

Αθήνα, Μάιος 2018

....
Δημήτριος Ν. Δανόπουλος

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

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

Περίληψη

Η Μηχανική Μάθηση έχει σημειώσει σημαντική εξέλιξη τα τελευταία χρόνια. Η αναγνώριση εικόνας καθίσταται σημαντικό στοιχείο σε όλο και περισσότερες εφαρμογές, από ιατρικές διαγνώσεις και αυτόνομα αυτοκίνητα μέχρι και σε μεγάλα κέντρα δεδομένων. Τα τελευταία χρόνια, state-of-the-art Συνελικτικά Νευρωνικά Δίκτυα Βαθιάς Μάθησης που μπορούν να καταγράψουν πολύπλοκα μη γραμμικά χαρακτηριστικά, έχουν δείξει τη δημοφιλία τους σε διάφορες εφαρμογές πραγματικού χρόνου, επιτυγχάνοντας ακρίβεια σε όλα τα τεστ κατανόησης εικόνας (Αναγνώριση Εικόνας, Ανίχνευση Εικόνας κ.λπ.). Ωστόσο, αυτή η δυνατότητα έρχεται με το κόστος των υψηλών απαιτήσεων υπολογισμών και μνήμης. Τα νευρωνικά δίκτυα απαιτούν δισεκατομμύρια αριθμητικές πράξεις και εκατομμύρια παραμέτρους, έχοντας έτσι πολύ υψηλή υπολογιστική πολυπλοκότητα.

Πολλές εφαρμογές νευρωνικών δικτύων αντιπροσωπεύουν μια υπολογιστική πρόκληση για τους επεξεργαστές γενικής χρήσης. Απαιτούν λύσεις υψηλής απόδοσης που ενσωματώνονται σε υπάρχοντα συστήματα με αυστηρούς περιορισμούς πραγματικού χρόνου και ισχύος. Ένα πρώτο σημαντικό βήμα της επιτάχυνσης αυτών των εφαρμογών είναι μια προσανατολισμένη προς το υλικό προσέγγιση που επιτρέπει γρήγορες και αποδοτικές λύσεις. Ως αποτέλεσμα, έχουν υιοθετηθεί επιταχυντές υλικού σε πλατφόρμες υψηλής απόδοσης, όπως η συστοιχία επιτόπια προγραμματιζόμενων πυλών (FPGAs), για τη βελτίωση της απόδοσης αυτών των εφαρμογών.

Αυτή η διπλωματική εργασία διερευνά τις δυνατότητες της επιτάχυνσης των νευρωνικών δικτύων με βάση το FPGA, χρησιμοποιώντας τη πλατφόρμα βαθιάς μάθησης Caffe και αποδεικνύει την βασιμότητα της ιδέας μια πλήρως λειτουργικής εφαρμογής σε ένα σύστημα Zynq System-on-Chip. Το ετερογενές σύστημα CPU-FPGA έχει σχεδιαστεί για την επιτάχυνση της αναγνώρισης εικόνας μέσω της πλατφόρμας Caffe, χρησιμοποιώντας τον επιταχυντή υλικού, επιτυγχάνοντας σημαντικά αποτελέσματα.

Ο επιταχυντής FPGA βασίζεται σε μια συνάρτηση που ονομάζεται GEMM (General Matrix Multiply), η οποία είναι το πιο υπολογιστικά συμφορητικό μέρος των αλγορίθμων αναγνώρισης εικόνας στη πλατφόρμα του Caffe. Αυτός ο αλγόριθμος πολλαπλασιασμού με πολλαπλούς ενσωματωμένους βρόχους επανάληψης υιοθετεί διάφορες τεχνικές βελτιστοποίησης τόσο στο λογισμικό όσο και στο υλικό για να ελαχιστοποιήσει τις προσβάσεις στη μνήμη και να παραλληλοποιήσει πλήρως τις αριθμητικές πράξεις. Ο επιταχυντής FPGA έχει υλοποιηθεί με Σύνθεση Υψηλού Επιπέδου στο περιβάλλον ανάπτυξης SDSoC για τη πλακέτα Xilinx Zynq ZC702 και φτάνει τη μέγιστη συχνότητα ρολογιού που υποστηρίζεται, η οποία είναι 200MHz με χρήση πόρων κοντά στο 80%. Η αξιολόγηση του επιταχυντή δείχνει ότι η εκτέλεση της συνάρτησης GEMM μπορεί να επιταχυνθεί έως και 380 φορές σε σχέση με την απλή εκδοχή σε ARM επεξεργαστή, ενώ το τελικό σύστημα με τον ενσωματωμένο επιταχυντή μπορεί να αυξήσει την απόδοση και την ενεργειακή κατανάλωση της αναγνώρισης εικόνας ως και 10% με λιγότερο από 0.4% μείωση στην ακρίβεια πρόβλεψης.

Λέξεις Κλειδιά: μηχανική μάθηση, συνελικτικά νευρωνικά δίκτυα, DNN, αναγνώριση εικόνας, Caffe, επιτάχυνση υλικού, FPGA, σύνθεση υψηλού επιπέδου, Zynq SoC

Abstract

Machine Learning has achieved major breakthroughs in recent years. Image Recognition is becoming a vital feature in ever more applications ranging from medical diagnostics and autonomous vehicles to big data centers. In recent years, state-of-the-art Deep Convolutional Neural Networks (DNNs) which can capture complex non-linear features have shown their popularity in various real world applications achieving record-breaking accuracies in all image understanding benchmarks (i.e. Image Recognition, Image Detection etc.). However, this ability comes at the cost of high computational and memory requirements. DNNs require billions of arithmetic operations and millions of parameters, thus they have a very high computational complexity.

Many DNN applications represent a computational challenge for general purpose processors. They demand high performance solutions that integrate into existing systems with tight real-time and power constraints. A first important step of the acceleration of these applications is a hardware-oriented approximation that enables fast and efficient solutions. As a result, hardware accelerators on high performance platforms, such as Field Programmable Gate Arrays (FPGAs), have been adopted to improve the performance of these applications.

This master thesis explores the potential of FPGA-based acceleration of DNNs, using Caffe Deep Learning Framework and demonstrates a fully functional proof-of-concept implementation on a Zynq System-on-Chip. The heterogeneous CPU-FPGA system is designed for the acceleration of image classification with Caffe framework by utilizing the hardware accelerator achieving significant results.

The FPGA accelerator is based on a function called GEMM (General Matrix Multiply) which is the most computational intensive part of the image classification algorithms in Caffe framework. This nested-loop matrix multiplication algorithm adopts several optimization techniques both in software and hardware to minimize the memory accesses and fully parallelize the arithmetic operations. The FPGA accelerator has been synthesized using High-Level Synthesis on SDSoC Development Environment for the Xilinx Zynq ZC702 board and reaches its maximum supported clock frequency of 200MHz with a device utilization of ~80%. The evaluation shows that the accelerator function can achieve up to 380 × speed-up over the simple ARM SW version while the final system with the integrated accelerator can boost the performance by 10% of the image classification with less than 0.4% accuracy drop while maintaining substantial energy-efficiency.

Keywords: machine learning, convolutional neural networks, DNN, image recognition, Caffe, hardware acceleration, FPGA, high-level synthesis, Zynq SoC

Ευχαριστίες

Αρχικά, θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή μου κ. Δημήτριο Σούντρη, για την εμπιστοσύνη που μου έδειξε και την ξεχωριστή εκπαιδευτική ευκαιρία που μου προσέφερε αυτή η διατριβή. Επιπλέον, εκφράζω την ειλικρινή μου ευγνωμοσύνη στον Μεταδιδακτορικό Ερευνητή κ. Χριστόφορο Κάχρη, για τη διαρκή υποστήριξη, ενθάρρυνση και τις συμβουλές του κατά τη διάρκεια της έρευνάς μου στο ΕΜΠ. Ήταν ένα πραγματικό προνόμιο και τιμή για μένα να μοιράζομαι την εξαιρετική του γνώση και τον ευχαριστώ επίσης για την οργάνωση αυτού του έργου και τη συμμετοχή μου σε διάφορα ενδιαφέροντα έργα και εκδηλώσεις. Ιδιαίτερες ευχαριστίες απευθύνω και σε όλο το προσωπικό του Εργαστηρίου Μικροϋπολογιστών (MicroLab) του ΕΜΠ για την πολύτιμη βοήθεια τους.

Τέλος, θέλω να εκφράσω την ευγνωμοσύνη μου σε όλη την οικογένειά μου που μου έδωσε την ενθάρρυνση και το κουράγιο κατά τη διάρκεια των σπουδών μου στο πανεπιστήμιο και ιδιαίτερα στη μητέρα μου για τη συνεχή άνευ όρων υποστήριξη και έμπνευση. Επίσης, μια ιδιαίτερη ευχαριστία προς τη Μαρία για όλη την αγάπη και την υποστήριξη.

Acknowledgments

First and foremost, I would like to thank Professor Dimitrios Soudris, the supervisor of this project, for the trust he showed me and the distinct educational opportunity that I was offered for this thesis. Moreover, I express my sincere gratitude to PostDoctoral Reasearcher Christoforos Kachris, my advisor, for his enduring support, encouragement and advice during my research at Technical University of Athens, Greece. It was a real privilege and an honor for me to share of his exceptional knowledge and I also thank him for arranging this project and involving me in various interesting projects and events. Special thanks also goes to the whole staff of the Microprocessors and Digital Systems Laboratory (MicroLab) of NTUA.

Finally, I want to express my gratitude to my whole family who gave me the encouragement and motivation throughout my studies in the university and especially my mother for the constant unconditional support and inspiration. Last but not least, a special thanks to Maria for all the love and support.

Contents

Εκτεταμένη Περίληψη

Εισαγωγή

Η συνεχής εκθετική αύξηση των μέσων ενημέρωσης, των IoT και των μεγάλων δεδομένων απαιτεί γενικά γρήγορες ταχύτητες επεξεργασίας, ενώ οι εφαρμογές πρέπει να διατηρούν χαμηλό κόστος ενέργειας και να διατηρούν μικρό χρόνο ανάπτυξης. Πολλά συστήματα υψηλής απόδοσης βασίζονται σε αλγόριθμους μηχανικής μάθησης (Machine Learning), όπως η ταξινόμηση εικόνων, οι αναλύσεις δεδομένων κ.λπ. που απαιτούνται για ενσωματωμένες και εφαρμογές μεγάλων δεδομένων.

Σε αυτό το πεδίο, τα Συνελικτικά Νευρωνικά Δίκτυα Βαθιάς Μάθησης ή στα αγγλικά Deep Convolutional Neural Networks (DNNs) έχουν αποκτήσει σημαντική έλξη λόγω του ότι προσφέρουν αξιόλογη ακρίβεια στην πρόβλεψη και μεγάλη ευελιξία. Αυτοί οι αλγόριθμοι εμπνευσμένοι από τον εγκέφαλο αποτελούνται από πολλαπλά στρώματα ανιχνευτών και ταξινομητών προτύπων και χρησιμοποιούν τεχνικές από τη μηχανική μάθηση, οι οποίες τους επιτρέπουν να ανταγωνίζονται την ακρίβεια των ανθρώπων όταν πρόκειται για παράδειγμα για αναγνώριση εικόνων.

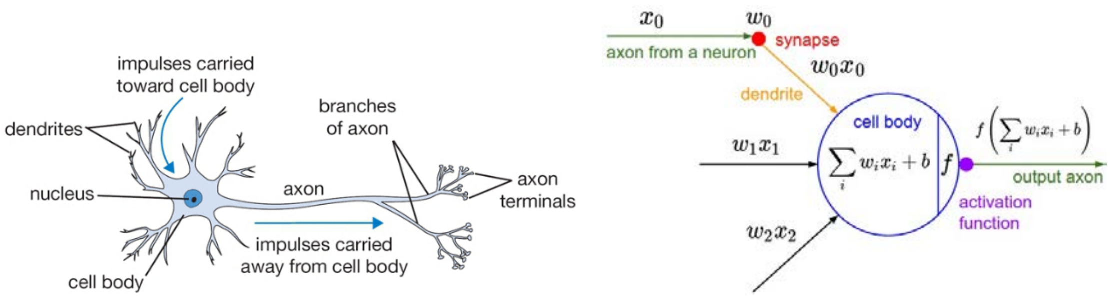
Παρόλο που απαιτείται σοβαρός υπολογισμός για την ανάλυση των μεγάλων ποσοτήτων δεδομένων, η χρήση συστημάτων υψηλών επιδόσεων φαίνεται πολλά υποσχόμενη, αλλά η πρόκληση της μείωσης του υψηλού ενεργειακού κόστους και των χρόνων επεξεργασίας παραμένει. Από την άλλη πλευρά, οι υλοποιήσεις από τέτοια συστήματα όπως η συστοιχία επιτόπια προγραμματιζόμενων πυλών ή αλλιώς στα αγγλικά Field-programmable gate array (FPGA) έχουν δει μεγάλη πρόοδο, καθώς οι νέες αναδυόμενες τεχνικές αξιοποιούν την αρχιτεκτονική των FPGAs εκμεταλλευόμενη τους επιταχυντές υλικού υψηλής απόδοσης (hardware accelerators) με μικρά κόστη ενέργειας, διατηρώντας παράλληλα την προσαρμοστικότητα των γρήγορων πρωτοτύπων. Με τη χρήση επιταχυντών υλικού αυξάνεται ο συνολικός ρυθμός εκτέλεσης των προγραμμάτων λόγω του εξαιρετικά παραλληλοποιήσιμου μαζικού αριθμού πράξεων που χρειάζονται οι αλγόριθμοι των DNN και επίσης μειώνεται η κατανάλωση ενέργειας. Το Caffe, ένα μια πλατφόρμα βαθιάς μάθησης του UC Berkley, έχει ήδη εφαρμοστεί και βελτιστοποιηθεί αρχικά μόνο σε δύο διαφορετικές αρχιτεκτονικές για CPU και GPU και μπορεί εύκολα να διαμορφωθεί χωρίς μεγάλες αλλαγές κώδικα.
Στην παρούσα εργασία παρουσιάζουμε:

- Μια τροποποιημένη έκδοση του Caffe για εύκολη μεταφορά του στον επεξεργαστή ARM (Zynq 7000) του FPGA SoC.
- Έναν επιταχυντή υλικού σχεδιασμένο στο περιβάλλον Xilinx SDSoC που εκμεταλλεύεται τα πλεονεκτήματα του FPGA και είναι κρίσιμος για τον αλγόριθμο αναγνώρισης εικόνας.
- Ένα σύστημα βασισμένο σε CPU-FPGA, εξαιρετικά ετερογενές προγραμματιζόμενο SoC που υποστηρίζει το περιβάλλον του Caffe και χρησιμοποιεί τον επιταχυντή υλικού, επιτυγχάνοντας σημαντική ταχύτητα και αποδοτικότητα ισχύος σε σύγκριση με τον επεξεργαστή ARM Zynq.

Θεωρητικό Υπόβαθρο

Μηχανική Μάθηση Η εκμάθηση μηχανών στη βάση της είναι η πρακτική της χρήσης αλγορίθμων για την ανάλυση δεδομένων, την εκμάθηση από αυτήν και, στη συνέχεια, για να γίνει μια απόφαση ή πρόβλεψη για κάτι στον κόσμο. Έτσι, αντί για την προ-υπάρχουσα ρουτίνα της χειροκίνητης τροποποίησης του λογισμικού με ένα συγκεκριμένο σύνολο οδηγιών για την εκτέλεση μιας συγκεκριμένης εργασίας, το μηχάνημα «εκπαιδεύεται» χρησιμοποιώντας μεγάλα ποσά δεδομένων και αλγορίθμων που του δίνουν τη δυνατότητα να μάθει πώς να εκτελέσει την εργασία. Έτσι, ο βασικός στόχος είναι να γενικευτεί και να καλυτερεύσει από την εμπειρία του. Γενίκευση σε αυτό το πλαίσιο είναι η ικανότητα μιας μηχανής εκμάθησης να εκτελεί με ακρίβεια νέα παραδείγματα / εργασίες αφού έχει εκπαιδευτεί σε ένα σετ δεδομένων. Τα παραδείγματα εκπαίδευσης προέρχονται από κάποια γενικά άγνωστη κατανομή πιθανότητας και ο εκπαιδευόμενος πρέπει να οικοδομήσει ένα γενικό μοντέλο σχετικά με αυτό το χώρο που του επιτρέπει να παράγει επαρκώς ακριβείς προβλέψεις σε νέες περιπτώσεις.

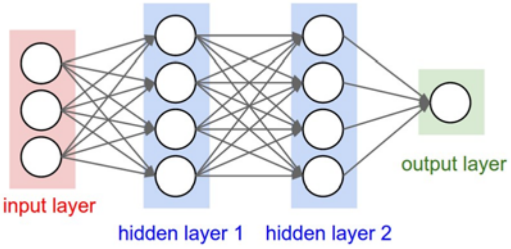
Νευρωνικά Δίκτυα Τα Συνελικτικά Νευρωνικά Δίκτυα (CNNs) είναι ένα είδος μοντέλου μηχανικής μάθησης που έχει μεγάλη πρακτική αξία στον τομέα της αναγνώρισης προτύπων. Διακρίνονται για την υπερσύγχρονη συμπεριφορά τους επειδή μπορούν να δημιουργήσουν αυτόματα τόσο χαρακτηριστικά υψηλού επιπέδου όσο και χαμηλού επιπέδου. Ένα συνελικτικό νευρωνικό δίκτυο είναι εμπνευσμένο από βιολογικές διεργασίες στις οποίες το πρότυπο σύνδεσης μεταξύ των νευρώνων εμπνέεται από την οργάνωση του ζωτικού οπτικού φλοιού. Το βασικό δομικό στοιχείο στα τεχνητά νευρωνικά δίκτυα είναι ο νευρώνας ο οποίος ανταποκρίνεται σε ερεθίσματα μόνο σε μια περιορισμένη περιοχή του οπτικού πεδίου, με αποτέλεσμα πολλοί μαζί να αντιλαμβάνονται συγκεκριμένα χαρακτηριστικά μιας εικόνας. Το παρακάτω διάγραμμα δείχνει ένα σχέδιο ενός βιολογικού νευρώνα (αριστερά) και ενός κοινού μαθηματικού μοντέλου (δεξιά). Μέσω συνάψεων οι οποίες ελέγχουν την ένταση κάθε ερεθίσματος παράγεται ένα τελικό ερέθισμα, το οποίο είναι το άθροισμα όλων των δενδριτών στον νευρώνα.



Εικόνα 1: Βιολογικός νευρώνας (αριστερά) και το μαθηματικό του μοντέλό (δεξιά) [1].

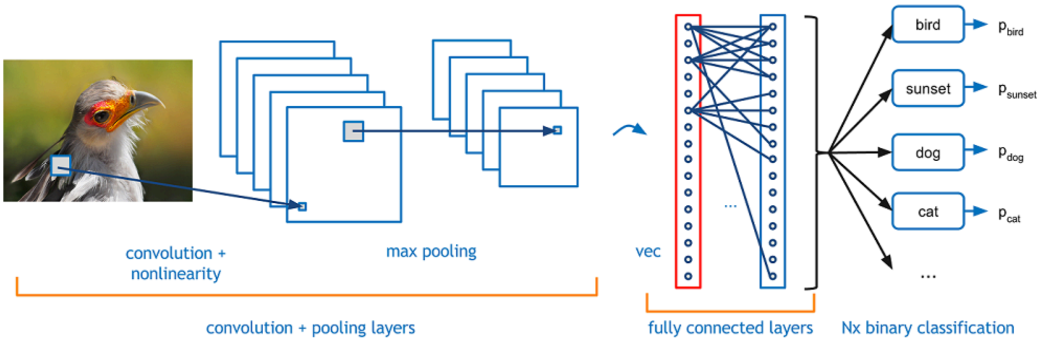
Λειτουργία Νευρωνικών Δικτύων Βαθιάς Μάθησης Ένα νευρωνικό δίκτυο δημιουργείται διασυνδέοντας πολλούς τεχνητούς νευρώνες. Οι νευρώνες είναι διατεταγμένοι σε ένα κατευθυνόμενο ακυκλικό γράφημα σε ένα δίκτυο προώθησης ώστε οι έξοδοι ορισμένων νευρώνων μπορούν να γίνουν εισροές σε άλλους νευρώνες. Τα βάρη στην είσοδο τους καθορίζουν πόσο σχετικοί ή όχι είναι στην εργασία που εκτελείται. Υπάρχουν στρώματα εισόδου και εξόδου και επιπλέον κρυφά επίπεδα που πολλές φορές αυξάνουν το μέγεθος και την πολυπλοκότητα του δικτύου. Αυτά μπορεί να περιλαμβάνουν επίπεδα όπως:

- Επίπεδο Συνέλιξης (Convolution): προσομοιώνει την απόκριση ενός μεμονωμένου ερεθίσματος κάνοντας την πράξη της συνέλιξης μεταξύ της εικόνας εισόδου και των βαρών διαφόρων φίλτρων.
- Επίπεδο Συγκέντρωσης (Pooling): πρόκειται για μια διαδικασία δειγματοληψίας που συνοψίζει τις εξόδους γειτονικών γκρουπ νευρώνων εντός ενός παραθύρου (patch) με μια αντιπροσωπευτική τιμή.
- Επίπεδο Μη Γραμμικότητας (Non-Linearity): χαρακτηριστικό επίπεδο των CNN που συνήθως εφαρμόζει μια διορθωτική μη γραμμική συνάρτηση (πχ. ReLU,$tanh(x)$, sigmoid).
- Πλήρως Συνδεδεμένο Επίπεδο (Fully Connected): συνδέει κάθε νευρώνα σε ένα στρώμα με κάθε νευρώνα σε ένα άλλο στρώμα αθροίζοντας τα αποτελέσματα για να πάρει την έξοδό του.



Εικόνα 2: Οργάνωση ενός νευρωνικού δικτύου με πολλαπλά επίπεδα [1].

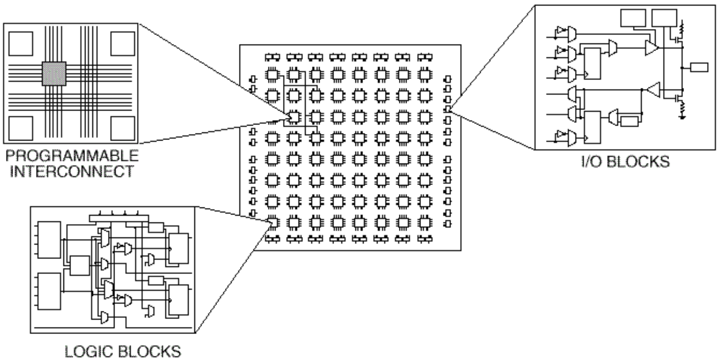
Επίσης τα νευρωνικά δίκτυα μπορούν να εκπαιδευτούν και οι παράμετροί τους μαθαίνονται κατά τη διάρκεια αυτής της φάσης. Το δίκτυο μπορεί να μάθει και να βρει τα βέλτιστα βάρη, καθορίζοντας μια λειτουργία απώλειας και χρησιμοποιώντας τον αλγόριθμο οπίσθιας τροφοδότησης (backpropagation) για την προσαρμογή των βαρών. Μετά την εκμάθηση, το δίκτυο είναι έτοιμο να αναγνωρίσει εικόνες. Ο στόχος είναι η εξαγωγή του συμπεράσματος (inference), το οποίο είναι η τελική έξοδος για την πρόβλεψη της εικόνας και καθορίζεται κατόπιν μετά από τον υπολογισμό όλων των ενδιάμεσων επιπέδων (επίπεδα συνέλιξης, συγκέντρωσης κτλ.).



Εικόνα 3: Πρόβλεψη εικόνας μετά από τον υπολογισμό της εξόδου του νευρωνικού δικτύου [2].

Αρχιτεκτονική FPGA Field-Programmable Gate Array (FPGA) ή στα ελληνικά συστοιχία επιτόπια προγραμματιζόμενων πυλών είναι ένα ολοκληρωμένο κύκλωμα το οποίο "προγραμματίζεται" και διαμορφώνεται αναλόγως την κατάσταση του προβλήματος. Η διαμόρφωση του FPGA καθορίζεται γενικά με τη χρήση μιας γλώσσας περιγραφής υλικού (HDL), παρόμοιας με εκείνη που χρησιμοποιείται για ένα ολοκληρωμένο κύκλωμα εξειδικευμένης εφαρμογής (ASIC) και αποσκοπεί στην εκτέλεση μιας υπολογιστικά έντονης εργασίας. Το πλεονέκτημά τους είναι ότι πολλές φορές είναι σημαντικά ταχύτερα για ορισμένες εφαρμογές λόγω της παραλληλοποιήσιμης φύσης τους.

Σε αντίθεση με τους κοινούς επεξεργαστές, τα συστήματα FPGAs έχουν ανεξάρτητους κόμβους επεξεργασίας (PEs) που ανατίθενται σε ένα ειδικό τμήμα του τσιπ και μπορούν να εργαστούν αυτόνομα χωρίς καμιά άλλη επίδραση από άλλα λογικά μπλοκ. Τα FPGAs αποτελούνται από 2D (ή και 3D) συστοιχίες λογικών μπλοκ οι οποίες συνδέονται μέσω προγραμματιζόμενων διασυνδέσεων για την υλοποίηση ενός αναδιαμορφώσιμου ψηφιακού κυκλώματος με σκοπό την εκτέλεση και επιτάχυνση κάποιας συγκεκριμένης λειτουργίας ή πράξης.



Εικόνα 4: Εσωτερική αρχιτεκτονική ενός FPGA [9]

Συχνά, τα FPGAs ενσωματώνονται σε «ετερογενή» συστήματα δηλαδή συστήματα που χρησιμοποιούν περισσότερα από ένα είδος επεξεργαστών για την εξιδεικευμένη αντιμετώπιση διαφόρων προβλημάτων. Για παράδειγμα η οικογένεια προγραμματιζόμενων συστημάτων Zynq-7000 SoC της Xilinx ενσωματώνει την προγραμματισιμότητα του λογισμικού ενός επεξεργαστή ARM με την προγραμματιζόμενη τεχνολογία του FPGA. Παραδοσιακά, οι προγραμματισμοί τέτοιων συστημάτων γίνονταν χρησιμοποιώντας μια γλώσσα περιγραφής υλικού (HDL) όπως VHDL ή Verilog αλλά τώρα έχουν βρεθεί βελτιστοποιημένες μεθοδολογίες, όπως η σύνθεση υψηλού επιπέδου (HLS), η οποία καθιστά την σύνθεση υλικού ευκολότερη, με πλατφόρμες όπως το SDSoC της Xilinx. Το εργαλείο αυτό καθιστά τον προγραμματισμό λογισμικού και υλικού πολύ πιο εύκολο, προσφέροντας για την σύνθεση υλικού γλώσσα προγραμματισμού υψηλού επιπέδου (C, C++), επιτρέποντας στους προγραμματιστές να εκμεταλλευτούν τα πλεονεκτήματα υλικού χωρίς να έχουν μεγάλη τεχνογνωσία υλικού.

Τα FPGAs και οι επιταχυντές υλικού, διαθέτουν σημαντικά πλεονεκτήματα έναντι άλλων συστημάτων (CPU, GPU) καθώς είναι εξαιρετικά παραλληλοποιήσιμοι έχοντας ταυτόχρονα μικρή ενεργειακή κατανάλωση. Το μερίδιο αγοράς των FPGA στην διεθνή αγορά των υπολογιστικών συστημάτων είναι αξιόλογο, με εφαρμογές σε ενσωματωμένα συστήματα όσο και στο υπολογιστικό νέφος (cloud computing) που για παράδειγμα παρέχει η υπηρεσία Amazon EC2.

Υλοποίηση DNN σε Zynq SoC

Τα ενσωματωμένα συστήματα έχουν συνήθως πολύ συγκεκριμένες απαιτήσεις και περιορισμούς όσον αφορά την υπολογιστική ισχύ και ενέργει και την διαχείριση των περιορισμένων πόρων σε πραγματικό χρόνο (πχ. περιορισμένη μνήμη). Τα Zynq SoCs είναι ιδανικά για την αποτελεσματική υλοποίηση του DNN καθώς επιτρέπουν τη δημιουργία προσαρμοσμένων κυκλωμάτων στο υλικό, συντονισμένα ακριβώς με τις ανάγκες του αλγορίθμου. Το αποτέλεσμα είναι η κορυφαία απόδοση ανά watt που συχνά ξεπερνά τα ενσωματωμένα συστήματα CPU και GPU. Σε αυτή την παράγραφο παρουσιάζουμε μια νέα μέθοδο για την εύκολη προσαρμογή ενός DNN που εκτελείται στο περιβάλλοντου Caffe σε ένα σύστημα με βάση το Zynq χρησιμοποιώντας την πλακέτα ZC702 της Xilinx, η οποία αποτελείται από επεξεργαστή ARM 32 bit με δύο πυρήνες.

1. Το πρώτο βήμα προκειμένου να μεταφερθεί ολόκληρο το περιβάλλον του Caffe στο Zynq 7000 SoC για να τρέξει στον πυρήνα ARM ήταν να μεταγλωττιστούν οι απαιτούμενες εξαρτήσεις βιβλιοθήκης του Caffe (3d party libraries) και στη συνέχεια ολόκληρη το περιβάλλον του και μετέπειτα να συνδεθούν όλες οι βιβλιοθήκες για την ορθή λειτουργία του (πχ. OpenCV, Boost, OpenBlas, HDF5, LMDB, Glog κτλ.). Για αυτά τα βήματα χρησιμοποιήσαμε τον cross-compiler ARM που συμπεριελήφθη στο περιβάλλον SDSoC. Συγκεκριμένα, χρειάστηκε χειροκίνητα να κατεβάσουμε τα αρχεία προέλευσης κάθε βιβλιοθήκης που χρειάζεται και εξαρτάται το Caffe και στη συνέχεια να τα εγκαταστήσουμε ένα προς ένα χρησιμοποιώντας το ARM toolchain «*arm-linux-gnueabihf*», καθορίζοντας την κατάλληλη διαμόρφωση για κάθε βιβλιοθήκη και ξεπερνώντας αρκετές επιπλοκές σε ορισμένες εγκαταστάσεις. Είναι σημαντικό να τονιστεί ότι στην εγκατάσταση και σύνδεση των βιβλιοθηκών επιλέχτηκε η υποστήριξη για τα NEON instrinsics που επιτρέπουν την επέκταση της SIMD αρχιτεκτονικής για λειτουργίες πολλαπλών πράξεων αλλά και η χρήση νημάτων (pthreads).

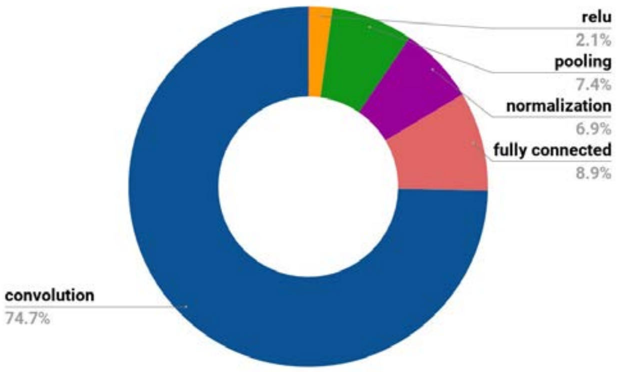
2. Το δεύτερο βήμα ήταν να βρεθεί μια κατάλληλη αρχιτεκτονική μοντέλου DNN για να χωρέσει στην περιορισμένη μνήμη του ενσωματωμένου SoC. Για την μεταγλώτισση του Caffe, η διεπαφή διατηρήθηκε αμετάβλητη και το μόνο μέρος που έπρεπε να τροποποιηθεί ήταν το Makefile. Μετά από την επιτυχή κατασκευή των δυναμικών βιβλιοθηκών (dynamic libraries) για το Caffe, έγινε η σύνδεση (linking) ρυθμίζοντας κατάλληλα το Makefile και επιλέγοντας αντί τον προ-υπάρχων μεταγλωττιστή g++, τον ARM cross-compiler για το ενσωματωμένο σύστημα.

3. Τέλος, οι αυστηροί περιορισμοί στην χρήση DNN σε μικρά ενσωματωμένα συστήματα περιλαμβάνουν την περιορισμένη μνήμη της συσκευής (on-chip memory). Για αυτόν τον λόγο, προτιμήθηκε η χρήση σχετικά μικρών μοντέλων νευρωνικών δικτύων (πχ. GoogleNet, SqueezeNet) τα οποία έχουν μικρές απαιτήσεις και παράλληλα προσφέρουν αποδεκτή ακρίβεια.

Σχεδιασμός του επιταχυντή υλικού για FPGA

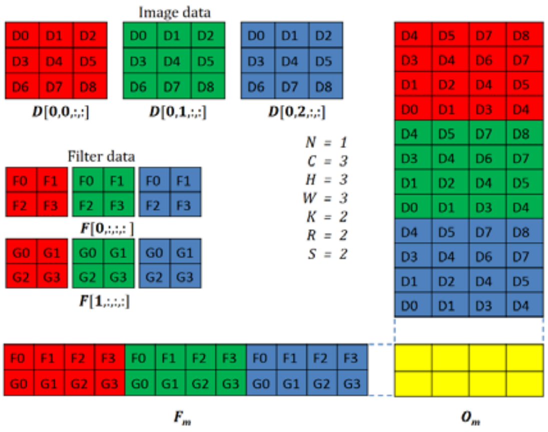
Οι επιταχυντές υλικού χρησιμοποιούνται για να εκφορτώσουν συγκεκριμένες εργασίες από τη CPU, βελτιώνοντας την παγκόσμια απόδοση του συστήματος και μειώνοντας τη δυναμική κατανάλωση ενέργειας. Τα FPGAs μέσα από την αναδιαμορφώσιμη αρχιτεκτονικής τους, μπορούν να αναδείξουν τον παραλληλισμό ενός συγκεκριμένου υπολογιστικά εντατικού κώδικα στο υλικό. Σε αυτή την παράγραφο αναλύουμε τον επιταχυντή υλικού που επιλέξαμε να αναπτύξαμε σε σύνθεση υψηλού επιπέδου (C++) χρησιμοποιώντας το Xilinx SDSoC για τον προγραμματισμό του λογισμικού και υλικού, επιδιώκοντας να επιταχύνουμε την εκτέλεσή του σε σχέση με τον επεξεργαστή ARM του Zynq SoC.

Καθορισμός συνάρτησης για επιτάχυνση Προκειμένου να οριστεί το μέρος του κώδικα προς επιτάχυνση, εκτελέσαμε αρχικά μια αξιολόγηση χρόνου (profiling) στο περιβάλλον του Caffe για τον εντοπισμό των σημείων συμφόρησης μνήμης ή υπολογισμού. Από την Εικόνα 5 φαίνονται τα επίπεδα του νευρωνικού δικτύου που παίρνουν περισσότερο χρόνο (convolution layers). Η Εικόνα 2 δείχνει τον τρόπο με τον οποίο κάνει συνέλιξη το Caffe με τα διάφορα φίλτρα.



Εικόνα 5: Profiling στο Caffe.

Όπως φαίνεται από την εικόνα, τα επίπεδα συνέλιξης στο Caffe (convolution layers) παίρνουν περίπου το 75% του συνολικού χρόνου εκτέλεσης του δικτύου. Οπότε η εξεύρεση των υπολογισμών για επιτάχυνση θα εντοπιστεί σε αυτό το επίπεδο.

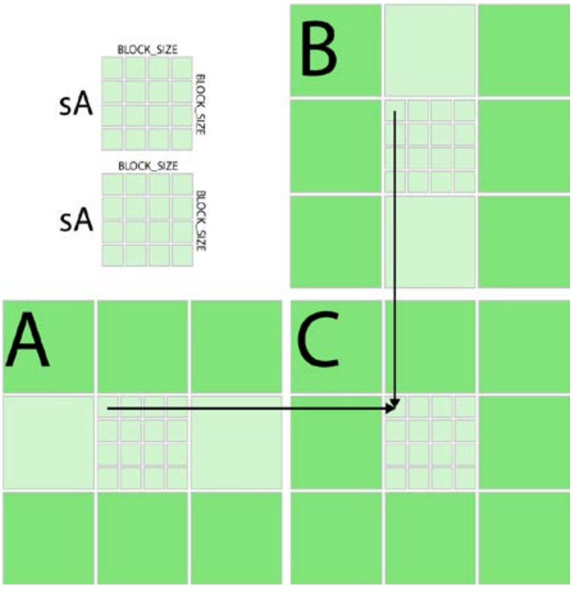


Εικόνα 6: Η συνέλιξη στο Caffe [3].

Όπως παριστάνει η εικόνα, το Caffe πραγματοποιεί συνέλιξη μεταξύ των εισόδων και των φίλτρων του δικτύου μέσα από πράξεις πολλαπλασιασμού πινάκων. Αυτές γίνονται στο Caffe, μέσα από την συνάρτηση GEMM (General Matrix Multiply) που πρόκειται για την συνάρτηση πινάκων $C = \alpha AB + \beta C$ και εκτελείται από την βιβλιοθήκη BLAS στον ARM επεξεργαστή.

Βελτιστοποίηση συνάρτησης για επιτάχυνση Αφού ορίστηκε η συνάρτηση που θα επιταχυνθεί μέσα από το υλικό (GEMM), η διαδικασία προχώρησε στα επόμενα βήματα που αφορούν την ορθή λειτουργία της συνάρτησης GEMM αλλά και τη βελτιστοποίηση την συνάρτησης σε λογισμικό και υλικό.

- *Βελτιστοποίηση Λογισμικού*: O πολλαπλασιασμός των πινάκων πραγματοποιήθηκε με το «σπάσιμο» του πίνακα σε μικρές παρτίδες και τον υπολογισμός των επιμέρους μπλοκ του πίνακα με την τελική την συνένωση αυτών για την οριστική έξοδο του πίνακα C. Αυτό βοήθησε στη ταχύτερη επικοινωνία στη μνήμη καθώς τα μπλοκ φορτώνονταν στην γρήγορη μνήμη cache του επεξεργαστή ARM (μελετήθηκε και το κατάλληλο μέγεθος μπλοκ). Συγκεκριμένα, για τον ορισμό των μπλοκ σε συνεχή μνήμη, χρησιμοποιήσαμε την εντολή *το sds_alloc()* αντί για *malloc()* (αυτό είναι επίσης μια απαίτηση για την εξασφάλιση της γρήγορης επικοινωνίας με τους πυρήνες υλικού που συζητάμε στην επόμενη παράγραφο). Επιπλέον, μια μικρή βελτιστοποίηση που κάναμε ήταν να αντιγράψουμε τη μήτρα B πριν από την αντίστοιχη μήτρα A. Αυτό τελικά αποκρύπτει ένα μέρος της συνολικής καθυστέρησης φορτίου, αφού η πρόσβαση της λειτουργικής μονάδας του A αντιστοιχεί φυσικά στη σειρά με την οποία φορτώνεται, επιτρέποντας στον υπολογισμό να αρχίσει μόλις το πρώτο στοιχείο του A επιστρέψει από τη μνήμη. Τέλος, για τη γρήγορη αντιγραφή και εκκαθάριση των μπλοκ κάθε φορά, χρησιμοποιήσαμε λειτουργίες *memcpy()* και *memset()* αντίστοιχα, οι οποίες είναι ίσως οι ταχύτερες λειτουργίες για το χειρισμό αυτών των υπολογισμών μνήμης στο λογισμικό.



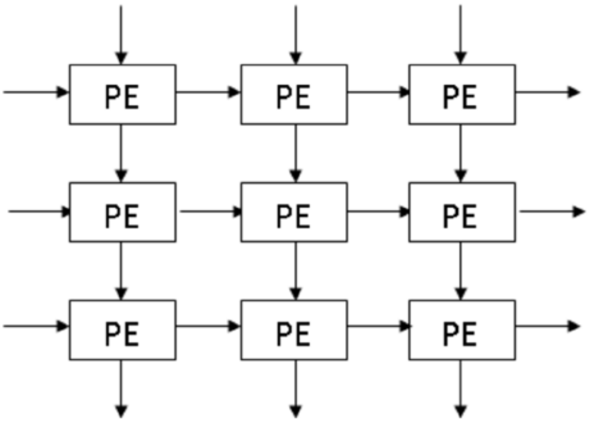
Εικόνα 7: GEMM στον επεξεργαστή.

Όπως παρουσιάζεται στην εικόνα η συνάρτηση GEMM, γίνεται διαμερισμός της πράξης του πολλαπλασιασμού πινάκων σε μικρότερους και τελική συνένωση με το άθροισμά τους στο τελικό πίνακα C. Αξίζει να τονισθεί ότι η συνάρτηση υποστηρίζει και διαφορετικά/ακανόνιστα μεγέθη πινάκων καθώς τα νευρωνικά δίκτυα στο Caffe πολλές φορές απαιτούν «περίεργες» διαστάσεις πινάκων (πχ. A=128x288 B=288x784).

Ακολουθεί η παράγραφος που περιγράφει τις βελτιστοποιήσεις στο υλικό [4].

Όπως αναφέρθηκε, η ο υπολογισμός του τελικού πίνακα γίνεται με μικρότερες πράξεις υποπινάκων οι οποίες και τελικά έχουν όλο το υπολογιστικό φορτίο πράξεων (MACC operations). Σε αυτή την παράγραφο περιγράφουμε τις κυριότερες τεχνικές βελτιστοποίησης του GEMM σε επίπεδο υλικού.

- *Βελτιστοποίηση Υλικού*: Για την καθοδήγηση της σύνθεσης και της χρήσης του υλικού χρησιμοποιήθηκαν, μέσα από το SDSoC, συγκεκριμένες ντιρεκτίβες (*pragmas)* που καθοδηγούν τους πόρους του υλικού και την λειτουργία τους. Όσον αφορά τη διασύνδεση του επιταχυντή με το λογισμικό επιλέχτηκε πρότυπο AXI SIMPLE_DMA με θύρες συστήματος ACP για σύνδεση με την κύρια μνήμη μέσω του διαύλου AXI4 στο Zynq ZC702. Προστέθηκε επίσης η οδηγία `#pragma SDS data access_pattern (array: SEQUENTIAL)` στις επιμέρους συναρτήσεις υλικού που προσδιορίζει συνεχές (sequential) μοτίβο πρόσβασης με την διεπαφή (ap_fifo) για γρήγορη μεταφορά δεδομένων. Επίσης, επιλέξαμε να αποθηκεύσουμε τα δεδομένα του υλικού σε προσωρινούς πίνακες ορισμένους κοντά στον υπολογισμό των πράξεων, προκειμένου να μειώσουμε όσο το δυνατόν περισσότερο την επικοινωνία μνήμης. Για την καθοδήγηση της χρήσης on-chip μνήμης (BRAM) χρησιμοποιήσαμε την οδηγία `#pragma HLS RESOURCE variable = <array> core = RAM_2P_BRAM` για τα μπλοκ πινάκων A και B. Τέλος, για την παραλληλοποίηση της εκτέλεσης τον πράξεων επιλέξαμε την τεχνική της διασωλήνωση (pipeline) τοποθετώντας σε κάθε βρόχο του κώδικα υλικού την ντιρεκτίβα `#pragma HLS pipeline II=1` αφού προσέξαμε να μην έχουμε αλληλεξαρτήσεις αποτελεσμάτων μεταξύ τους. Σε συνδυασμό με αυτή την τεχνική, για να αποδεσμεύσουμε τον αριθμό των υπολογιστικών μονάδων (PEs) που ήταν αρχικά περιορισμένες σε ένα μπλοκ μνήμης που δεσμεύσαμε, διαμελίσαμε ισόποσα τους καταχωρητές μνήμης σε περισσότερα BRAMs επιτρέποντας περισσότερες θύρες πρόσβασης και μονάδες υπολογισμού (πχ. DSPs). Αυτό έγινε με τη χρήση της οδηγίας `#pragma HLS array_partition variable=<array> block factor=<N> dim=<dim>` μετά την δήλωση των υποπινάκων δίνοντας προσοχή στις ρυθμίσεις της ντιρεκτίβας (πχ. block factor, dim κτλ).

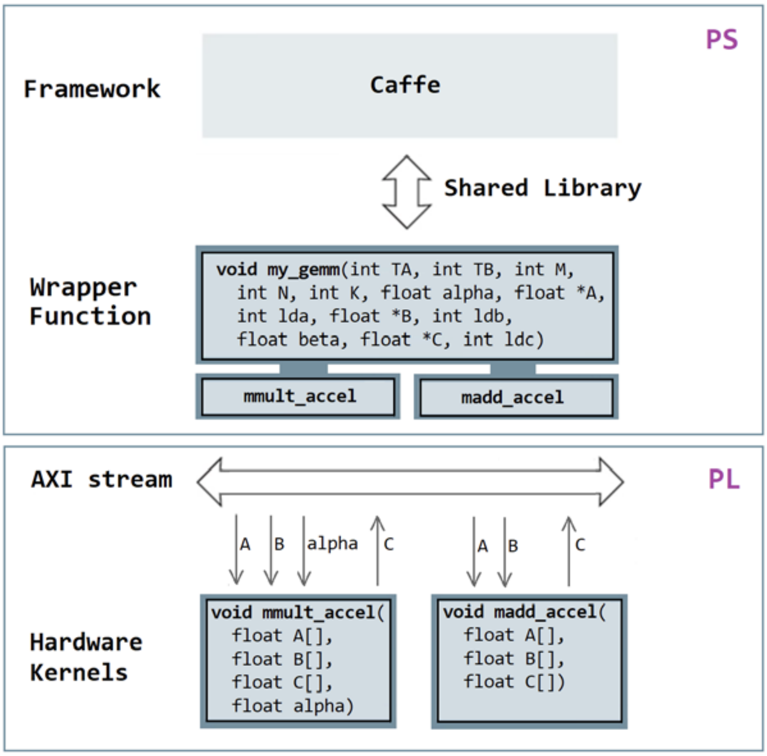


Εικόνα 8: Αρχιτεκτονική υλικού [5].

Όπως φαίνεται στην εικόνα η αρχιτεκτονική του υλικού του FPGA σχεδιάστηκε ώστε να είναι συστολική. Δηλαδή, τα δεδομένα φτάνουν ταυτόχρονα (ανά κύκλο) στις μονάδες επεξεργασίας από διαφορετικά σημεία κάνοντας τους εκάστοτε υπολογισμούς με διασωλήνωση, κατέχοντας μικρή καθυστέρηση μεταφοράς (latency) και γρήγορη ταχύτητα επεξεργασίας.

Κατασκευή και λειτουργία του τελικού συστήματος Μετά τη μεταφορά του Caffe στον ARM του Zynq SoC και το σχεδιασμό του επιταχυντή FPGA (GEMM) που θα χρησιμοποιηθεί για την επιτάχυνση του αλγόριθμου ταξινόμησης εικόνας ακολουθεί η ενσωμάτωσή του με το περιβάλλον του Caffe για την πλήρη εκτέλεση του στο FPGA SoC. Αυτή η διαδικασία τα εξής βήματα:

Προκειμένου να ενσωματωθεί ο επιταχυντής υλικού με το περιβάλλον του Caffe και να το μεταφέρουμε στο FPGA SoC, έπρεπε να το εξάγουμε ως κοινή/μοιραζόμενη βιβλιοθήκη αντί για binary εφαρμογή και στη συνέχεια να την συνδέσουμε με το περιβάλλον του Caffe. Μέσα από το SDSoC Development Environment καταφέραμε όχι μόνο να δημιουργήσουμε τη δυναμική/κοινή βιβλιοθήκη (dynamic library) του επιταχυντή, αλλά και το bitstream μαζί με την κάρτα SD από την οποία θα ξεκινούσε το σύστημά μας. Η κοινή βιβλιοθήκη ήταν κατάλληλη για σύνδεση και η λειτουργία του επιταχυντή ενσωματώθηκε με επιτυχία στο πλαίσιο του Caffe, επικοινωνώντας με το υλικό του FPGA όταν χρειαζόταν. Η διαδικασία παρουσιάζεται στην παρακάτω εικόνα.



```
void my_gemm(int TA, int TB, int M,
  int N, int K, float alpha, float *A,
  int lda, float *B, int ldb,
  float beta, float *C, int ldc)
```

```
void mmult_accel(
  float A[],
  float B[],
  float C[],
  float alpha)
```

```
void madd_accel(
  float A[],
  float B[],
  float C[])
```

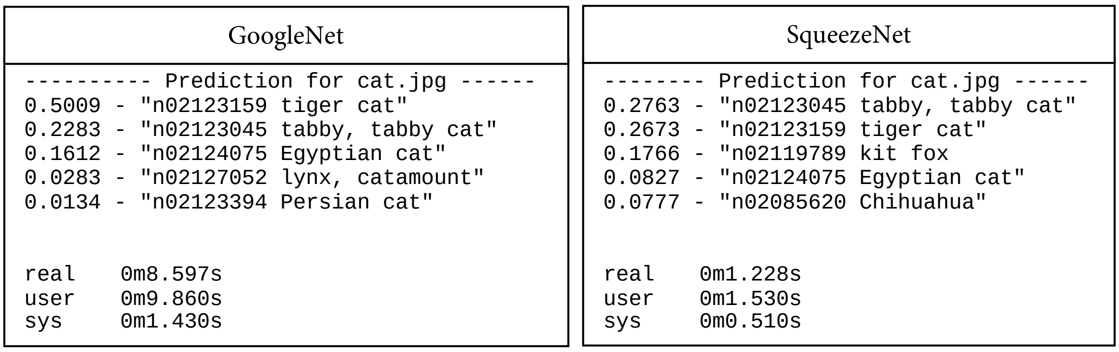
Εικόνα 9:
Λειτουργία συστήματος.

Όπως φαίνεται στο σχήμα, ολόκληρη η διαδικασία ξεκινάει από το Caffe όπου ο χρήστης τρέχει την εντολή αναγνώρισης εικόνας στον επεξεργαστή και στη συνέχεια, κάθε φορά που χρειάζεται να γίνει μια κλήση GEMM, φορτώνει την συνάρτηση μας *my_gemm,* εκτελώντας τις πράξεις στο υλικό και επικοινωνώντας μέσω του διαύλου AXI DMA (stream).

Τέλος, η δημιουργία της κάρτας SD για την εκκίνησης του συστήματος έγινε με εργαλεία SDK από το περιβάλλον SDSoC. Λήφθηκε υπόψη το bitstream από τον επιταχυντή υλικού καθώς προστέθηκε στην εικόνα εκκίνησης. Επιπλέον, για το ενσωματωμένο σύστημα, προτιμήθηκε το ελαφρύ λειτουργικό σύστημα της Xilinx Petalinux αφού παρείχε όλες τις απαραίτητες λειτουργίες που έχει ένα αντίστοιχο λειτουργικό Linux. Για την ρύθμιση του FPGA για λειτουργία, επιλέχτηκε σειριακό πρωτόκολλο επικοινωνίας με το Desktop PC μέσω της θύρας UART, επικοινωνώντας με το λειτουργικό μέσα από το πρόγραμμα TerraTerm.

Αξιολόγηση και Αποτελέσματα

Απόδοση Caffe στον ενσωματωμένο επεξεργαστή Στην παράγραφο αυτή περιορίζουμε τα αποτελέσματα σε μια περίληψη των πιο σημαντικών χαρακτηριστικών της λειτουργίας του περιβάλλοντος του Caffe στον ARM CPU. Αρχικά, παρέχουμε αρκετές μετρήσεις στο Caffe σχετικά με τη λειτουργία του (ακρίβεια πρόβλεψης, χρόνος εκτέλεσης κ.λπ.) και στη συνέχεια αναλύουμε ορισμένα χαρακτηριστικά ορισμένων μοντέλων νευρωνικών δικτύων παρουσιάζοντας την εσωτερική δομή πίσω από την λειτουργία τους.



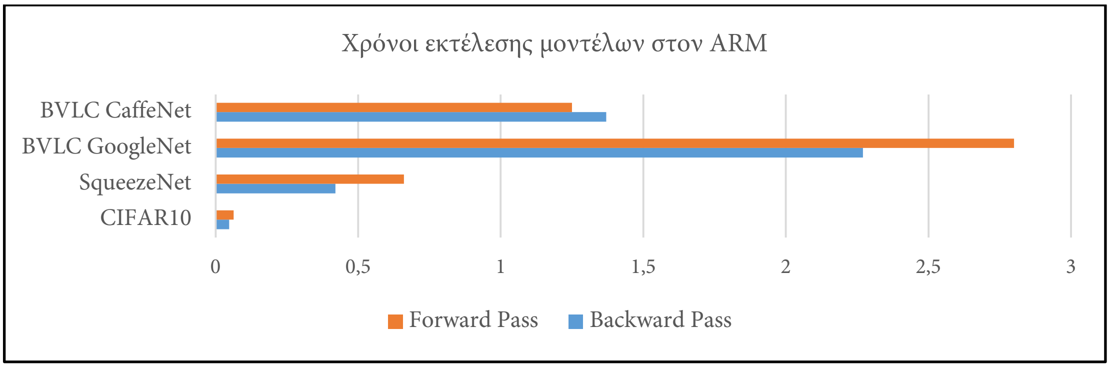
GoogleNet

```
---------- Prediction for cat.jpg ------
0.5009 - "n02123159 tiger cat"
0.2283 - "n02123045 tabby, tabby cat"
0.1612 - "n02124075 Egyptian cat"
0.0283 - "n02127052 lynx, catamount"
0.0134 - "n02123394 Persian cat"


real    0m8.597s
user    0m9.860s
sys     0m1.430s
```

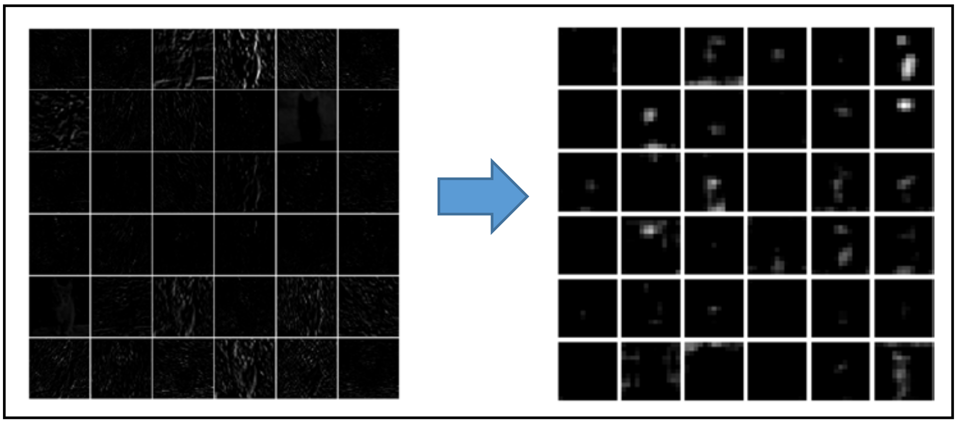
SqueezeNet

```
-------- Prediction for cat.jpg ------
0.2763 - "n02123045 tabby, tabby cat"
0.2673 - "n02123159 tiger cat"
0.1766 - "n02119789 kit fox
0.0827 - "n02124075 Egyptian cat"
0.0777 - "n02085620 Chihuahua"


real    0m1.228s
user    0m1.530s
sys     0m0.510s
```

Εικόνα 10: Αναγνώριση εικόνας μέσα από το Caffe στον ARM σε διαφορετικά μοντέλα.



Εικόνα 11: Σύγκριση χρόνων για συμπέρασμα (forward pass) και εκμάθηση (backward pass).



Εικόνα 12: Σύγκριση απόκρισης εξόδου από 1^o σε 5^o επίπεδο συνέλιξης στο μοντέλο του CaffeNet

Απόδοση επιταχυντή στο FPGA Στην παράγραφο ξεκινάμε με την ανάλυση λειτουργίας του επιταχυντή σχετικά με την καθυστέρηση, την ανάλυση κίνησης δεδομένων, την χρήση των διαθέσιμων πόρων κ.λπ. Στη συνέχεια, δίνουμε αρκετές μετρήσεις για τον επιταχυντή υλικού σχετικά με την επιτάχυνση από το ARM χρησιμοποιώντας ένα πρόγραμμα C ++ σαν test-bench για τον προσδιορισμό της απόδοσης.

Πίνακας 1: Κατανομή πόρων επιταχυντή σε ZC702 FPGA
Floating Point (αριστερά), Fixed Point (δεξιά)

Resource	Used	Total	%Utilization
DSP	164	220	75,55
BRAM	128	140	91,43
LUT	45557	53200	85,63
FF	24445	106400	22,97

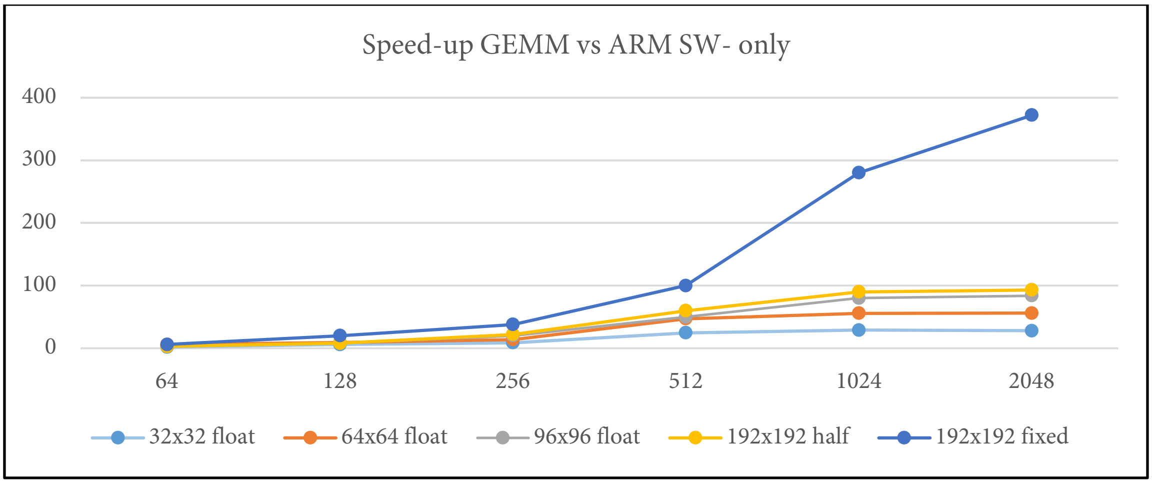
Resource	Used	Total	%Utilization
DSP	193	220	87,73
BRAM	96	140	68,57
LUT	36681	53200	68,95
FF	26683	106400	25,08

Πίνακας 2: Σύνοψη καθυστέρησης στους βρόχους (κύκλοι ρολογιού).
Floating Point (αριστερά), Fixed Point (δεξιά)

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
-Loop1	36875	36875	13	1	1	36864	yes
-Loop2	222143	222143	966	6	1	36864	yes

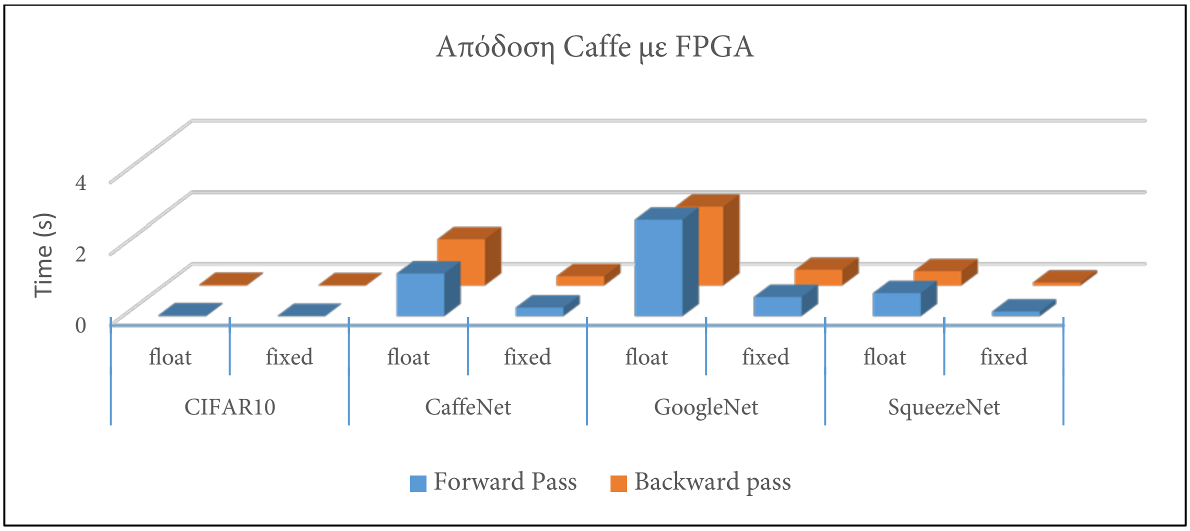
Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
-Loop1	36870	36870	8	1	1	36864	yes
-Loop2	37254	37254	392	1	1	36864	yes

Τέλος για την αξιολόγηση του επιταχυντή έγιναν διάφορες δοκιμές όσον αφορά το μέγεθος του μπλοκ ή την ακρίβεια στην πράξη (πχ. Float/Fixed point). Πρωτεύουσα θέση κατείχε η υλοποίηση σε fixed point καθώς ελαχιστοποιούσε τον απαιτούμενο αριθμό πόρων και την καθυστέρηση επιτυγχάνοντας μέχρι και 38 GFLOPs σε επίδοση kernels υλικού με ρολόι 200MHz και τελική επιτάχυνση συνάρτησης GEMM στα 380×.

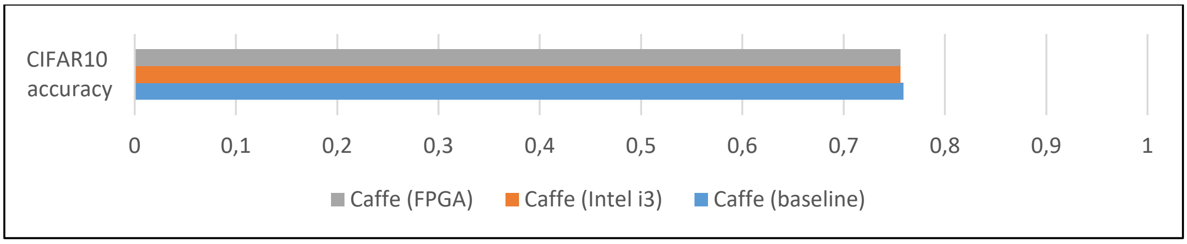


Εικόνα 13: Σύγκριση χρόνου σε διαφορετικές υλοποιήσεις της συνάρτησης GEMM

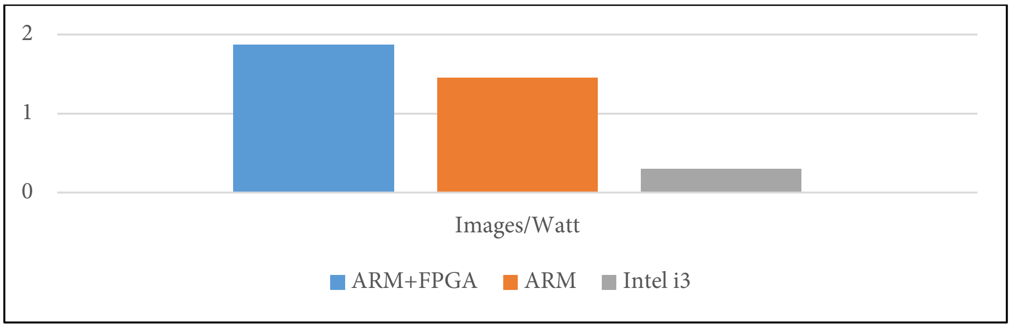
Απόδοση Caffe στο FPGA SoC Μετά την ενσωμάτωση του επιταχυντή υλικού που σχεδιάσαμε με το πλαίσιο Caffe, θέλαμε να μετρήσουμε τη συνολική απόδοση του Caffe στο ετερογενές CPU-FPGA SoC (ZC702) και να το συγκρίνουμε με την υλοποίηση αριθμητικής ακρίβειας float και fixed point στον επιταχυντή (η υλοποίηση fixed είναι προσομοιωμένη). Έτσι, σε αυτή τη παράγραφο μετράμε την απόδοση αναγνώρισης εικόνας και την ακρίβεια στο νέο τροποποιημένο πλαίσιο του Caffe στο FPGA που χρησιμοποιεί τον επιταχυντή GEMM και αξιολογούμε την κατανάλωση ισχύος και ενέργειας του συστήματός μας.



Εικόνα 14: Σύγκριση χρόνου για συμπέρασμα (forward pass) και εκμάθηση (backward pass) για διαφορετικές υλοποιήσεις της συνάρτησης GEMM σε διαφορετικά μοντέλα.



Εικόνα 15: Σύγκριση ακρίβειας πρόβλεψης σε διαφορετικές υλοποιήσεις



Εικόνα 16: Απόδοση ενέργειας τελικού συστήματος σε διάφορες αρχιτεκτονικές. Η υλοποίηση με FPGA καταναλώνει 3,905W και όπως φαίνεται στη εικόνα έχει 7 φορές καλύτερη ενεργειακή απόδοση/Watt σε σχέση με τον Intel i3 επεξεργαστή (προσομοιωμένο με υλοποίηση fixed point αριθμητική ακρίβεια).

Συμπεράσματα

Τα Νευρωνικά Δίκτυα Βαθιάς Μάθησης (DNNs) αντιπροσωπεύουν ένα καθολικό μοντέλο, το οποίο μπορεί να αξιοποιηθεί για να επιλύσει μια μεγάλη ποικιλία προβλημάτων. Στον τομέα αυτό, τα FPGAs μπορούν να βελτιώσουν σημαντικά την απόδοση και την ενέργεια αυτών των εφαρμογών χρησιμοποιώντας επιταχυντές υλικού (hardware accelerators). Ωστόσο, πλατφόρμες για υλοποίηση DNN, όπως το Caffe, δεν υποστηρίζουν επίσημα τη ευκολοδιάκριτη χρήση τέτοιων συστημάτων επιτάχυνσης.

Αυτό το έργο παρουσιάζει ξεκάθαρα την ανάπτυξη DNN στο Xilinx FPGA SoC χρησιμοποιώντας το περιβάλλον του Caffe ταυτόχρονα με τη χρήση επιταχυντών υλικού για να επιτύχει καλύτερη απόδοση. Μεταγλωττίσαμε πρώτα ολόκληρο το περιβάλλον του Caffe και όλες τις εξαρτήσεις βιβλιοθηκών για να δουλέψουν και να τρέχουν στον ενσωματωμένο επεξεργαστή ARM με τη μέθοδο cross-compilation. Στη συνέχεια, μέσω της πλήρους αξιολόγησης της λειτουργίας του περιβάλλοντος, προσδιορίσαμε τα σημεία υπολογιστικής συμφόρησης για να καθορίσουμε τη σωστή συνάρτηση επιτάχυνσης υλικού η οποία ήταν τελικά η συνάρτηση GEMM (General Matrix Multiplication). Στη συνέχεια περιγράψαμε τις στρατηγικές βελτιστοποίησης στο λογισμικό αλλά και στον επιταχυντή υλικού ώστε να αξιοποιήσουμε τον μαζικό παραλληλισμό και τους διαύλους επικοινωνίας της μνήμης του FPGA. O επιταχυντής FPGA έχει συντεθεί με Σύνθεση Υψηλού Επιπέδου (C++) χρησιμοποιώντας το περιβάλλον ανάπτυξης Xilinx SDSoC για το Xilinx Zynq ZC702 SoC και φτάνει επιτάχυνση μέχρι και 380x με τη μέγιστη συχνότητα ρολογιού των 200MHz και με χρήση πόρων στο ~80%. Η αξιολόγηση και η επαλήθευση της συνάρτησης υλικού ήταν επιτυχής, παράγοντας σωστά αποτελέσματα για διαφορετικά μεγέθη και διαστάσεις πινάκων. Τέλος, ενσωματώσαμε τον επιταχυντή υλικού με το πλαίσιο Caffe και το έτρεξε με επιτυχία στο ετερογενές σύστημα CPU-FPGA το οποίο αξιοποιεί την αρχιτεκτονική FPGA με το επιταχυντή υλικού που σχεδιάσαμε. Για την τελική αξιολόγηση του συστήματος μετρήσαμε την απόδοση της αναγνώρισης εικόνων και τα αποτελέσματα έδειξαν ότι το προτεινόμενο σύστημα μπορεί να μειώσει το χρόνο αναγνώρισης μέχρι και 10% σε σύγκριση με τον επεξεργαστή ARM και επίσης να μειώσει την κατανάλωση ενέργειας έχοντας λιγότερο από 0.4% μείωση ακρίβειας.

Το σύστημα βασισμένο σε CPU-FPGA με υποστήριξη του περιβάλλοντος Caffe έχει συναρμολογηθεί σε ένα πλήρως λειτουργικό σύστημα στην πλατφόρμα Xilinx Zynq-7000 All Programmable SoC. Αυτό το σχέδιο καταδεικνύει σαφώς τη σκοπιμότητα υλοποίησης ενσωματωμένων DNN βασισμένα σε FPGA. Η τρέχουσα λύση παρουσιάζει ήδη μια αποδοτική εκτέλεση και λειτουργία του συστήματος αυτού καθώς έχουν επισημανθεί αρκετές τεχνικές για σημαντικά κέρδη στην απόδοση και την κατανάλωση ενέργειας.

1 Introduction

1.1 Motivation

The continuing exponential increase of media, IoTs and big data in general requires faster and faster processing speeds while the applications must maintain a low power cost and keep the development time small. Many high performance systems rely on machine learning (ML) algorithms such as image classification, data analytics etc. which are required for embedded and big data applications as well. Hardware and device makers are in a mad dash to create or acquire the perfect chip for performing deep learning training and inference.

In this field, Deep Convolutional Neural Networks (DNNs) have gained significant traction due to the fact that they offer state-of-the-art accuracies and important flexibility. These brain-inspired algorithms use techniques from machine learning which enables them to rival the accuracy of humans when it comes to classification of images. Although a serious amount of computation is needed to analyze the large amounts of data, the use of multicore systems seems promising but the challenge of reducing the high energy cost and the processing times remains. FPGA implementations on the other hand have seen great advancement as new emerging techniques leverage the FPGA architecture taking advantage of the high performance hardware accelerators with few power costs while keeping the adaptability of fast prototyping. With the utilization of hardware accelerators the total throughput is increased because of the highly parallelizable massive number of multiply-accumulate operations that DNN algorithms need and also the energy consumption is decreased. Caffe, a deep learning framework of UC Berkley, has already been implemented and optimized in two different architectures for CPU and GPU only and can be easily configured without hard-coding.
In this paper we present:

- A modified version of Caffe to effortlessly port it into the ARM (Zynq 7000 based) processor of FPGA.
- A hardware accelerator designed on Xilinx SDSoC Environment that exploits the benefits of FPGA and is crucial for the image classification algorithm.
- A CPU-FPGA-based system, a highly heterogeneous all-programmable SoC that supports the Caffe framework and utilizes the hardware accelerator achieving significant speed and power efficiency compared to the ARM Zynq processor.

1.2 Chapter Organization

The thesis is organized as follows:

Chapter 2 provides background information on Machine Learning and Neural Networks, architecture and design of FPGAs, and at the end gives an analysis of Caffe Deep Learning framework. In the first section, a brief overview is given of Machine Learning concepts and related applications in this field. Next, a detailed presentation of Neural Networks is provided and then Deep Neural Networks are analyzed along with Image Recognition implementations. Then, in the second section, we discuss the design and architecture of FPGAs along with presenting design methods, such as High Level Synthesis, and their purpose in creating hardware accelerators for boosting applications performance. Last, we provide information of the operation of Caffe Deep Learning Framework.

Chapter 3 is dedicated on how we ported the whole Caffe framework in ARM CPU of the Zynq SoC in order to run it on the embedded system. Detailed information is given on the steps of this process in order to run Caffe with full support (image recognition, model benchmarks etc.) on the ARM processor and optimize in order to overcome the embedded system's physical constraints.

Chapter 3 consists of the design and optimization schemes used for the FPGA hardware accelerator that was used to accelerate the Caffe framework and specifically image recognition in the embedded SoC. The optimization and design strategies included both software and hardware techniques for maximum performance in throughput, parallelism and communication.

Chapter 5 gives a detailed analysis and evaluation of Caffe running on the embedded CPU, the designed hardware accelerator and last the results of the final heterogeneous CPU-FPGA system that runs and accelerates image recognition on Caffe utilizing the hardware accelerator. This chapter provides the appropriate testing and benchmark results of the proposed system and the performance and hardware analysis of the FPGA accelerator.

Chapter 6 summarizes the important results obtained by this research and gives suggestions on future work valuable to the reader.

2 Background

2.1 Introduction to Machine Learning

Machine Learning at its most basic is the practice of using algorithms to parse data, learn from it, and then make a determination or prediction about something in the world. So rather than hand-coding software routines with a specific set of instructions to accomplish a particular task, the machine is “trained” using large amounts of data and algorithms that give it the ability to learn how to perform the task. Thus, the core objective is to generalize from its experience and perform accurately on new, unseen examples/tasks after having experienced a learning data set. The training examples come from some generally unknown probability distribution and the learner has to build a general model about this space that enables it to produce sufficiently accurate predictions in new cases.

Machine learning tasks are typically classified into two broad categories, depending on whether there is a learning "signal" or "feedback" available to a learning system:

- Supervised learning: The computer is presented with example inputs and their desired outputs, given by a "teacher", and the goal is to learn a general rule that maps inputs to outputs.
- Unsupervised learning: No labels are given to the learning algorithm, leaving it on its own to find structure in its input. Unsupervised learning can be a goal in itself (discovering hidden patterns in data) or a means towards an end (feature learning).

There are many approaches to machine learning tasks. Some of them include:

- Artificial neural network (ANN): a learning algorithm, usually called "neural network" which is inspired by biological neural networks. They are usually used to model complex relationships between inputs and outputs, to find patterns in data, or capture the statistical structure in unknown distributions.
- Clustering: an analysis that assigns a set of observations into subsets (called clusters) so that observations within the same cluster are similar according to some predesignated criterion or criteria, while observations drawn from different clusters are dissimilar.
- Genetic algorithm: a heuristic search that mimics the process of natural selection, and uses methods such as mutation and crossover to generate new genotype in the hope of finding good solutions to a given problem.

2.2 Introduction to Convolutional Neural Networks

A convolutional neural network is one type of machine learning model which has great practical value in the field of pattern recognition. They are distinguished for their state-of-the-art behavior because they can automatically create both high level and low level features.

2.2.1 Brain analogy

Convolutional neural networks (CNNs) were inspired by biological processes in which the connectivity pattern between neurons is inspired by the organization of the animal visual cortex. Individual cortical neurons respond to stimuli only in a restricted region of the visual field known as the receptive field. The receptive fields of different neurons partially overlap such that they cover the entire visual field. So, in a way it behaves similarly to the human brain, that is to simulate its densely interconnected brain cells using digital neurons that trigger or respond when they 'sense' certain features regardless of the feature position in the visual field.

Modeling of a neuron The basic building block in artificial neural networks is the neuron. The diagram below shows a drawing of a biological neuron (left) and a common mathematical model (right). Each neuron receives input signals from its dendrites and produces output signals along its (single) axon. The axon eventually branches out and connects via synapses to dendrites of other neurons. In the computational model of a neuron, the signals that travel along the axons (e.g. x0) interact multiplicatively (e.g. w0x0) with the dendrites of the other neuron based on the synaptic strength at that synapse (e.g. w0). The idea is that the synaptic strengths (the weights w) are learnable and control the strength of influence of one neuron on another. In the basic model, the dendrites carry the signal to the cell body where they all get summed. If the final sum is above a certain threshold, the neuron can fire, sending a spike along its axon. Based on this rate code interpretation, we model the firing rate of the neuron with an *activation function* f, which represents the frequency of the spikes along the axon. Historically, a common choice of activation function is the *sigmoid function* σ, since it takes a real-valued input (the signal strength after the sum) and squashes it to range between 0-1.

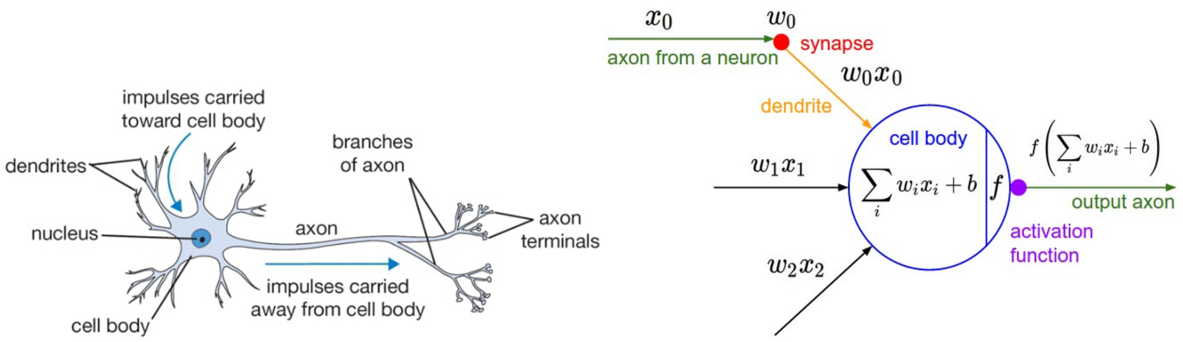


Figure 2.1: A biological neuron (left) and its mathematic model (right) [1].

Neural Network A neural network is formed by interconnecting many artificial neurons. The neurons are arranged in a directed acyclic graph in a *feed-forward network* although some architectures are using multilayer perceptrons in which the neurons are organized in layers. In other words, the outputs of some neurons can become inputs to other neurons. Cycles are not allowed since that would imply an infinite loop in the forward pass of a network. There are input and output layers and additional hidden layers that many times increase network size and complexity. As stated before, each neuron assigns a weighting to its input that is how correct or incorrect it is relative to the task being performed. The final output is then determined by the total of those weightings.

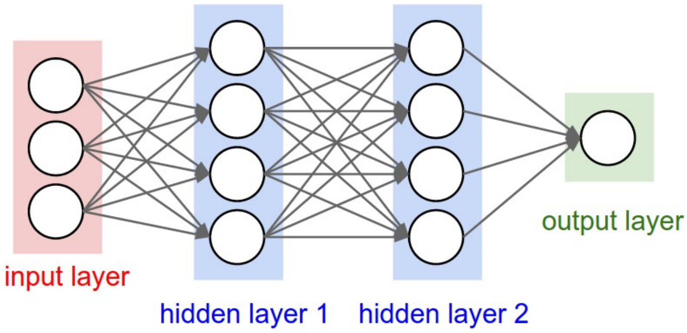


Figure 2.2: The organization of multiple layers of neurons [1].

2.2.2 Operation of a multi-layer network

Layers Types Convolutional Neural Networks use many different interconnected layers, as shown in the previous figure, specifically designed among other tasks to recognize or detect 2-dimensional image data. A Neural Network can have different layers performing unique tasks aiming to give a specific output that is passed through the other layers.
Some of the fundamental layers found in many models are:

- *Convolutional Layer* applies a convolution operation to the input, passing the result to the next layer. The convolution emulates the response of an individual neuron to visual stimuli. Each convolutional neuron processes data only for its receptive field. Convolutional layers take several feature maps as input and using convolution with the filter weights $k\ x\ k$ acquired from the training process they produce feature maps as output. For filters larger than $1\ x\ 1$, border effects reduce the output dimensions. To avoid this effect, the input image is typically padded with zeros on each side thus reducing the output dimensions.
- *Nonlinearity Layer* apply a non-linear activation function to each input pixel. The most popular activation function is the *Rectified Linear Unit (ReLU)* which computes $f(x) = \max(0, \mathrm{x})$ and clips all negative elements to zero. Other networks use sigmoidal functions such as $f(x) = 1/(1 + e^{-x})$ or $f(x) = \tanh(x)$.

- *Pooling Layer* combines the outputs of neuron clusters at one layer into a single neuron in the next layer by summarizing multiple input pixels into one output pixel. For example, max pooling uses the maximum value from each of a cluster of neurons at the prior layer. Another example is average pooling, which uses the average value from each of a cluster of neurons at the prior layer. They are usually applied to a patch of $2x2$ or $3x3$ input pixels, but can also be applied as global pooling to the whole input image.
- *Fully Connected Layer* connects every neuron in one layer to every neuron in another layer. Each output value of a Fully Connected layer looks at every value in the input layer, multiplies them all by the corresponding weight it has for that input index, and sums the results to get its output. They can be visualized as one dimensional and perform the high level reasoning in the neural network.
- *Dropout Layer* is a popular method to combat overfitting in large CNNs. These layers randomly drop a selectable percentage of their connections during training which prevents the network from learning very precise mappings and forces some abstraction and redundancy to be built into the learned weights.
- *Softmax Layer* are often used in the final layer of a neural network-based classifier. It converts the raw class scores z_i into class probabilities P_i according to $P_i = e^{z_i} / \sum_k e^{z_k}$, which result in a vector P that sums up to 1.

Training of a Neural Network Neural networks can be trained and their parameters are learned during this phase. The network can learn and find the optimal weights, by defining a loss function and using the *backpropagation* algorithm to adapt weights. The most popular approach is called supervised training and requires a set of labeled examples. The training starts with small, maybe random, initial weights and each example is fed through the network many times to produce better results (*feed-forward pass*). The network is considers to be trained after reaching the target performance results on the training data. The *backpropagation* algorithm works by calculating the loss from the current network output and a ground truth and computing a weight update from it by passing the error backwards through the network (*backward-pass*). The goal of the learning process is to minimize this loss by updating training weights. The magnitude of these updates is determined by the so-called *learning rate.*

Below is a basic optimization algorithm called *Stochastic Gradient Descent* that minimizes the loss function. It consists in using a few examples to compute the gradient of the parameters with respect to loss function:

$$\theta_{t+1} = \theta_t - \lambda \cdot \nabla_{\theta_t} L\big(f_{\theta_t}(x_i), y_i\big)$$

There is no proof of good convergence. However, this algorithm reaches good local minima in practice, even when the parameters are randomly initialized. One of the reason could be the stochastic property of this algorithm, allowing the latter to optimize different loss functions and thus to get out of bad minima. The other reason could be that a lot of local minima are almost as accurate as the global minima. Answers to this question are still under active research.

Image Classification After training, the neural network is built and is ready to recognize images on new data through a process called *inference.* In this setting, the aim is to compute the output of the network. Colors of the image are represented as RGB values (a combination of red, green and blue ranging from 0 to 255). Computers could then extract the RGB value of each pixel and put the result in an array for interpretation which will be fed throughout all the layers of the network. Then the input image is scanned for features using small filters. This feature extraction starts with the input image where each pixel represents the input for the neurons grouped in features. The neurons in the feature maps are organized in 2-dimensional grids.

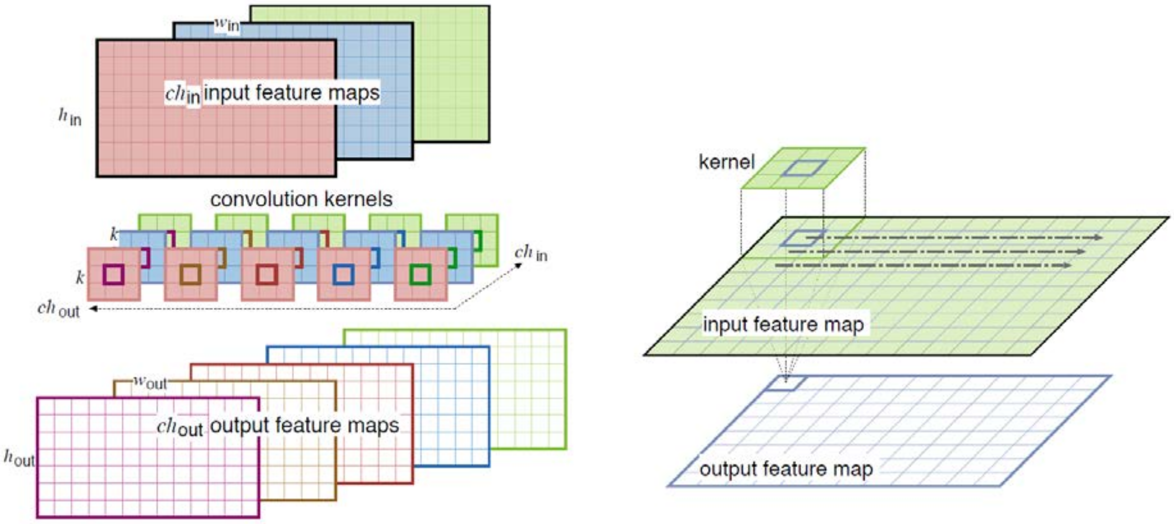


Figure 2.3: Left: Illustration of data inside a CNN Layer (left). The ch_{in} input feature maps are transformed into ch_{out} output feature maps by applying $ch_{in}\ x\ ch_{out}$ filter kernels of size $k\ x\ k$.
Right: Illustration of a 2D convolution between 3x3 kernel and an input feature map by sliding the kernel over the input pixels and performing MACC operations at each pixel position [6].

The input is followed by alternating layers of convolution, pooling and others. The process of a CNN can involve numerous hidden layers besides convolutional and pooling which the data is fed through. After passing all layers, the network produces a final result vector with a possibility P_i for each category of our model.

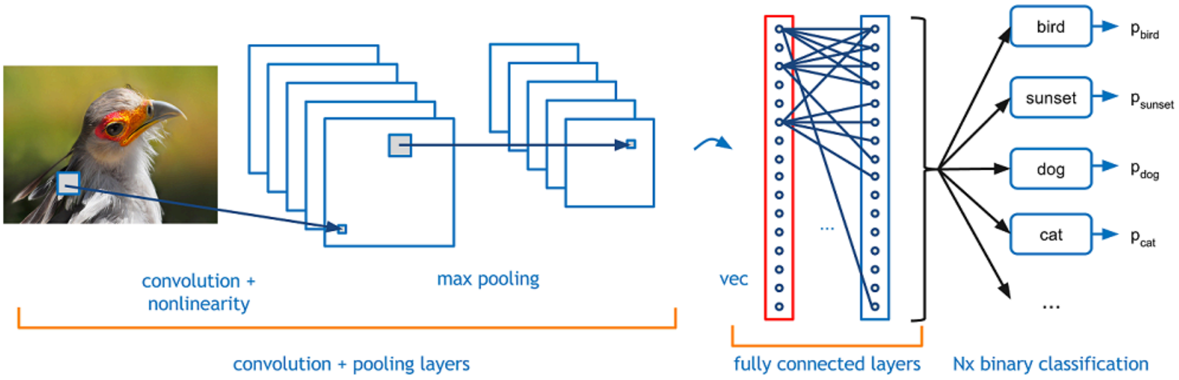


Figure 2.4: The organization of multiple layers of neurons [2].

2.2.3 Network Topologies

A lot of convolutional architectures have been developed from the 1990's. In this section, we make an inventory of the most known architectures. Each one represent a step further for more advanced visual recognition.

LeNet This kind of architecture is one of the first successful applications of CNNs. It was developed by Yann LeCun in the 1990's and was used to read zip codes and digits. This architecture, with regard to the modern ones, differs on many points. Thus, we will limit ourselves on the most known, LeNet-5, and we will not delve into the details. In overall this network was the origin of much of the recent architectures, and a true inspiration for many people in the field.
LeNet-5 features can be summarized as:

- sequence of 3 layers: convolution, pooling, non-linearity
- inputs are normalized using mean and standard deviation to accelerate training
- sparse connection matrix between layers to avoid large computational cost
- hyperbolic tangent or sigmoid as non-linearity function,
- trainable average pooling as pooling function,
- fully connected layers as final classifier,
- mean squared error as loss function.

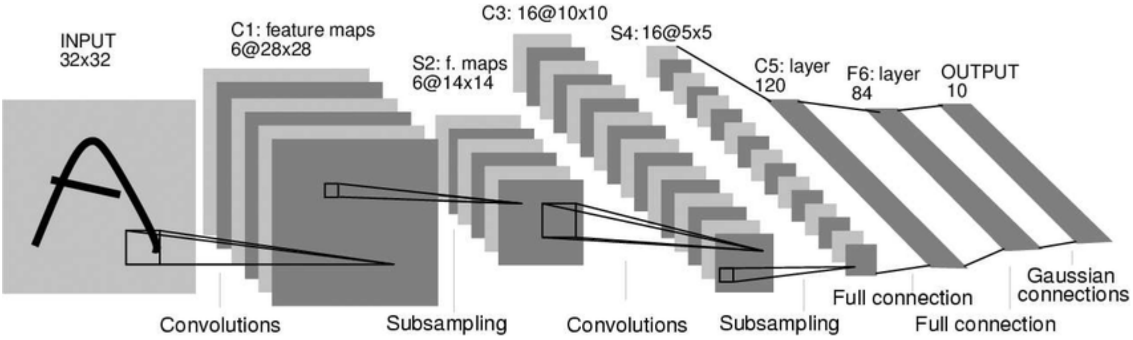


Figure 2.5: Architecture of LeNet-5, a convolutional neural network for digits recognition [7].

AlexNet It is one of the first work that popularized convolutional networks in computer vision. AlexNet was submitted to the ImageNet ILSVRC challenge of 2012 and significantly outperformed the other hand crafted models (accuracy top5 of 84% compared to the second runner-up with 74%). This network, compared to LeNet, was deeper (60 millions of parameters) and bigger (5 convolutional layers, 3 max pooling and 3 fully-connected layers). It popularized:

- the ReLU as non-linearity function of choice
- the method of stacking convolutional layers plus non-linearity on top of each other without being immediately followed by a pooling layer

- the method of overlapping Max Pooling, avoiding the averaging effects of Average Pooling.

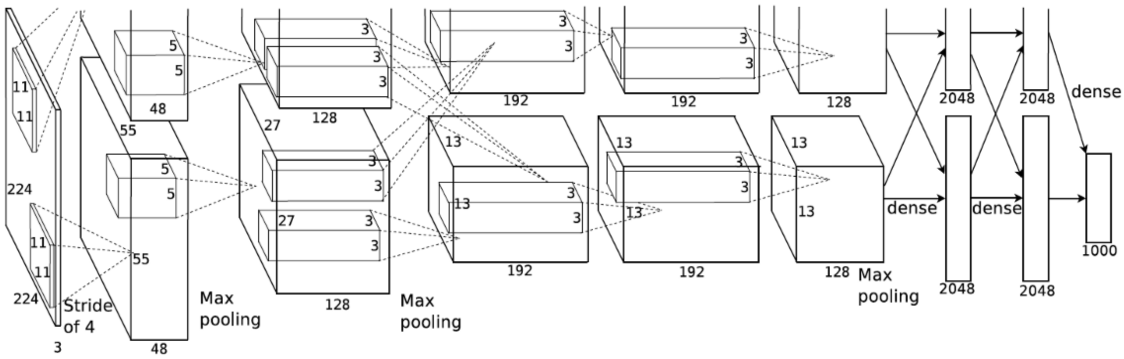


Figure 2.6: Architecture of AlexNet [8].

VggNet It was the runner-up architecture of ILSVRC2014 with almost 140 millions of parameters. Its main contributions were to show that depth is a critical component for good performance, to use much smaller 3×3 filters in each convolutional layers and also to combine them as a sequence of convolutions. The great advantage of VggNet was the insight that multiple 3×3 convolution in sequence can emulate the effect of larger receptive fields, for examples 5×5 and 7×7. However, having so many weights, one forward pass requires nearly 16 billion MACC operations (multiply-accumulate operations).

GoogleNet GoogLeNet or Inception was the winner architecture of ILSVRC2014. Its main contribution was the development of an Inception Module that dramatically reduced the number of parameters (40 millions). Also, it eliminated a large amount of parameters by using average pooling instead of fully connected layers at the top of the convolutional layers. Further versions of the GoogLeNet has been released. The most recent architecture available is InceptionV3. Notably, it uses batch normalization.

ResNet It was the winner architecture of ILSVRC2015 with 152 layers. Its main contribution was to use batch normalization and special skip connections for training deeper architectures. ResNet with 1000 layers can be trained with those techniques. However, it has been empirically found that ResNet usually operates on blocks of relatively low depth which act in parallel, rather than serially flow the entire length of the network.

SqueezeNet It differs from the other CNN architectures in this list due to the fact that the design was not record-breaking in accuracy. Instead, the authors developed a network with an accuracy similar to AlexNet but with 50x less parameters. This parameter reduction has been achieved by using fire modules, a reduce-expand micro-architecture comparable to Inception modules, and careful balancing of the architecture. The 18-layer SqueezeNet uses 7x7, 3x3 and 1x1 convolutions, 3x3 max pooling, dropout and global average pooling, but neither fully connected, nor Batch Normalization layers. One forward pass requires only 860 million MACC operations and 1.24 million parameters are enough to achieve less than 19.7% single-crop top-5 error.

Comparison of Topologies A common performance measure for deep CNNs is their classification accuracy. Most researchers give their performance numbers in top-1 and top-5 measure. The top-1 error measure gives the percentages of images that were classified incorrectly on the test set. Since the ImageNet data set has very fine-grained image classes which are sometime even hard for humans to distinguish, most researchers prefer to use the top-5 error measure. Hence, an image is considered as classified correctly if one of the top 5 predictions are correct. On the other hand, some research tend to focus on creating smaller networks that can achieve good classification performance. In the table below, there can be seen the tradeoff between accuracy and network size. For example VGG has a great accuracy but it is also the largest network in terms of parameter size and MACC operations. On the contrary, SqueezeNet which is a small network designed for mobile devices is not as accurate but has a drastically reduced size.

Table 2.1: Comparison of different CNN Topologies for Image Classification on ImageNet

	#conv. layers	*#MACCs [millions]*	*#params [millions]*	*#activations [millions]*	*ImageNet top-5 error*
AlexNet	5	1140	62.4	2.4	19.7%
VGG-16	16	15470	138.3	29.0	8.1%
GoogleNet	22	1600	7.0	10.4	9.2%
ResNet-50	50	3870	25.6	46.9	7.0%
Inception v3	48	5710	23.8	32.6	5.6%
SqueezeNet	18	860	1.2	12.7	19.7%

2.3 FPGAs

A field-programmable gate array (FPGA) is an integrated circuit designed to be configured by a customer or a designer after manufacturing – hence "field-programmable" and its development started in the late 1980s. The FPGA configuration is generally specified using a hardware description language (HDL), similar to that used for an application-specific integrated circuit (ASIC) and they aim to perform a computational intensive task. Their advantage lies in that they are sometimes significantly faster for some applications because of their parallel nature and optimality in terms of the number of gates used for a certain process.

2.3.1 Introduction to FPGA

An FPGA is a semiconductor device and usually contains a large integrated circuit that can be used to create custom logic functions and perform specific tasks as a digital circuit. Reprogrammable silicon also has the same flexibility of software running on a processor-based system, but it is not limited by the number of processing cores available. Unlike processors, FPGAs are truly parallel in nature, so different processing operations do not have to compete for the same resources. Each independent processing task is assigned to a dedicated section of the chip, and can function autonomously without any influence from other logic blocks. As a result, the performance of one part of the application is not affected when you add more processing because the application logic is implemented in hardware circuits rather than executing on top of an OS, drivers, and application software.

FPGA Architecture FPGAs consist of 2D array of configurable logic blocks (also 3D stacked architectures have been introduced) which are connected via programmable interconnects to implement a reconfigurable digital circuit and I/O blocks to allow the circuit to access the outside world. FPGAs resource specifications often include the number of configurable logic blocks, number of fixed function logic blocks such as multipliers, and size of memory resources like embedded block RAM.

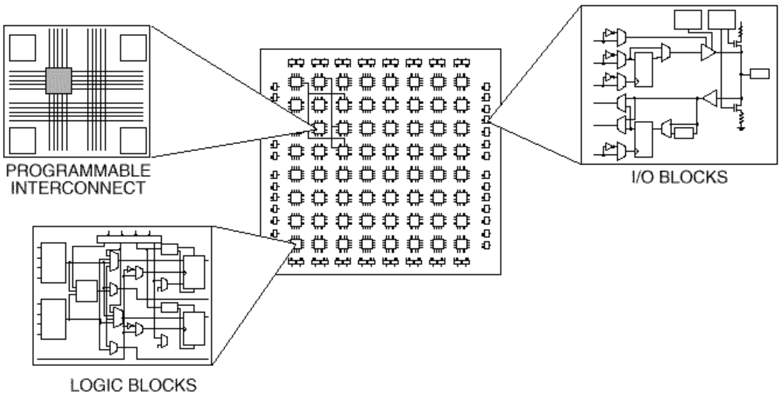


Figure 2.7: The inside architecture of the FPGA [9].

The basic structure components of the FPGA are described below:

- Configurable Logic Blocks (CLB): In general, a logic block consists of a few logical cells (called ALM, LE, slice etc.). A typical cell consists of a 4-input LUT, a full adder (FA) and a D-type flip-flop, as shown below. The Lookup Tables (LUT) are in this figure split into two 3-input LUTs. In normal mode those are combined into a 4-input LUT through the left mux. In arithmetic mode, their outputs are fed to the FA. The selection of mode is programmed into the middle multiplexer. The output can be either synchronous or asynchronous, depending on the programming of the mux to the right, in the figure example. In practice, entire or parts of the FA are put as functions into the LUTs in order to save space [10].

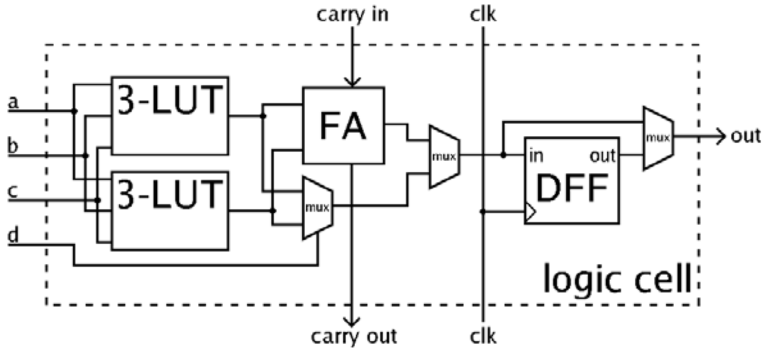


Figure 2.8: Simplified example illustration of a logic cell [10].

- Programmable Interconnects: The interconnects can be thought of as a network of wire bundles running vertically and horizontally between the logic slices. They provide connections among logic blocks and I/O blocks to implement any user-defined circuit. The routing interconnect of an FPGA consists of wires and programmable switches that form the required connection. These programmable switches are configured using the programmable technology.
- External I/O Blocks: Since clock signals are normally routed via special-purpose dedicated routing networks in commercial FPGAs, they and other signals are separately managed. These blocks enable the FPGA to communicate with other parts of the system and send/receive external signals.
- Hard Blocks: Modern FPGA families expand upon the above capabilities to include higher level functionality fixed into the silicon. Having these common functions embedded into the silicon reduces the area required and gives those functions increased speed compared to building them from primitives. Examples of these include multipliers, generic Digital Signal Processor (DSP) blocks, embedded processors, high speed I/O logic, embedded memories and even IP cores such as Ethernet MACs etc. These cores exist alongside the programmable fabric, but they are built out of transistors instead of LUTs so they have greater level performance and power consumption while not consuming a significant amount of fabric resources, leaving more of the fabric free for the application-specific logic [11].

Comparison with other architectures FPGA-based products are basically very effective for low to medium volume production as they are easy to program and debug, and have less cost and faster time-to-market. All these major advantages of an FPGA come through their reconfigurability which makes them general purpose and field programmable. But, the very same reconfigurability is the major cause of its disadvantages; thus making it larger, slower and more power consuming than ASICs [12]. In contrast to FPGAs, ASICs do not suffer any area or timing overhead from configuration logic and generic interconnects, and therefore typically result in the smallest, fastest and most energy-efficient systems. However, the sophisticated fabrication processes for ASICs results in lengthy development cycles and very high upfront costs, which demands a first-time-right design methodology and very extensive design verification. Also the advantage of FPGA-based systems over traditional processor-based systems such as desktop computers, smartphones, most embedded systems, and also over GPUs, is the availability of freely programmable general-purpose logic blocks. These can be arranged into heavily specialized accelerators for very specific tasks, resulting in improved processing speed, higher throughput and energy savings. This advantage comes at the price of reduced agility and increased complexity during the development, where the designer needs to carefully consider the available hardware resources and the efficient mapping of his algorithm into the FPGA architecture. Furthermore, some algorithmic problems do not map well onto the rigid block structures found on FPGAs [12] [6].

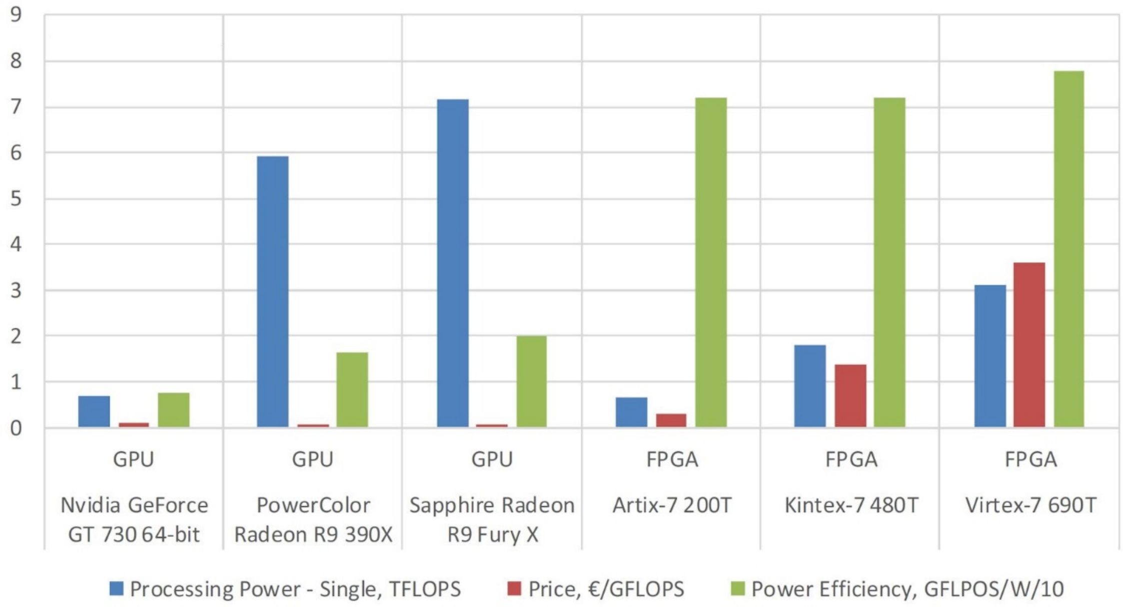


Figure 2.9: A comparison between high-end GPUs and FPGAs [6].

2.3.2 Introduction to High Level Synthesis

High-level synthesis (HLS) is a methodology that provides optimized hardware synthesis from high-level programming language specifications such as C/C++ and System C. HLS tools allow designers to use a software program to specify the target system functionality, enabling them to exploit hardware advantages without building up hardware expertise.

Traditionally, FPGAs are programmed using a Hardware Description Language (HDL) such as VHDL or Verilog. Most designs are described at Register Transfer Level (RTL), where the programmer specifies the algorithm using a multitude of parallel processes which operate on vectors of binary signals and simple integer data types derived from them. These processes describe combinational logic, basic arithmetic operations as well as registers, and are driven by the rising and falling edges of a clock signal. RTL descriptions are very close to the logic gates and wires which are actually available in the underlying FPGA or ASIC technology, and therefore the hardware that results from RTL synthesis can be closely controlled. However, the process of breaking down a given algorithm into logic blocks, processes and finite state machines on the register transfer level is very tedious and error-prone. Many design decisions have to be made before writing any code, and later changes are quite difficult and costly. This prevents iterative optimizations and demands a lot of intuition, experience and expert knowledge from designers. Increasing the level of abstraction with HLS High-Level Synthesis tries to lower this barrier by enabling designers to specify their algorithms in a high-level programming language such as C, C++ or SystemC. Many implementation details are abstracted away and handled by the HLS compiler, which converts the sequential software description into a concurrent hardware description, usually at RTL level [6] [13].

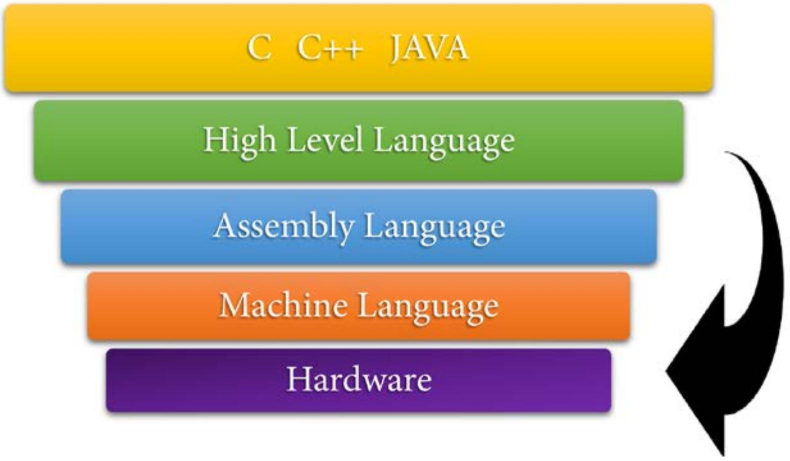


Figure 2.10: An illustration that shows the transition from High Level Language to Hardware.

SDSoC Development Environment The SDSoC development environment is an HLS tool that provides a familiar embedded C/C++/OpenCL application development experience including an easy to use Eclipse IDE and a comprehensive design environment for heterogeneous systems deployment [14]. SDSoC delivers system level profiling, automated software acceleration in programmable logic, automated system connectivity generation, and libraries to speed programming. It also enables end user and third party platform developers to rapidly define, integrate, and verify system level solutions and enable their end customers with a customized programming environment [15]. So in this way, designers can use loops, arrays, structs, floats, arithmetic operations, function calls, and even object-oriented classes. These are automatically converted into counters, memories, computation cores and handshake protocols as well as accompanying state machines and schedules with the most efficient way possible. The compilation can be influenced using scripted compiler directives or embedded compiler pragmas, which are instruction directives interpreted directly by the SDSoC compiler. Operations are by default scheduled to be executed concurrently and as early as possible. Using the compiler pragmas, the designer can further influence the inference of memories and interfaces, the parallelization of loops and tasks, the synthesis of computation pipelines, etc.

Although such tools exists that make the development of specific applications much easier and faster, there are still some difficulties on HLS synthesis tools. First, the results achieved especially for large applications with complex module and control-flow hierarchy are not as high as expected. Very often, the programming style of the source code has a severe impact on the quality of the synthesized implementation. Also many existing HLS compilers impose proprietary extensions or restrictions (e.g. exclusion of while loops) on the programming model of the specifications that they accept as input, and various heuristics on the HLS transformations that they utilize. At last, HLS tools have long learning curves meaning that only experienced users can develop efficient code on these platforms.

Amazon AWS Cloud Amazon provides a web service for Cloud Computing which is the on-demand delivery of compute power, database storage, applications, and other IT resources through a cloud services platform via the internet with pay-as-you-go pricing [16]. Specifically, Amazon Elastic Compute Cloud (Amazon EC2) is a web service that provides this web-scale cloud computing capability [17]. It provides with complete control of the computing resources and lets users run on Amazon's proven computing environment and build or test their applications on a secure environment.

Amazon provides along with these cloud service specific platforms, such as the EC2 F1 which is a compute instance with field programmable gate arrays (FPGAs) that users can program and test their custom hardware accelerations for their application [18]. F1 instances are easy to program and come with everything that has to do with developing, simulating, debugging and compiling the hardware acceleration code in High Level Synthesis.

2.3.3 Heterogeneous Systems

Heterogeneous computing refers to systems that use more than one kind of processor or cores. These systems gain performance or energy efficiency not just by adding the same type of processors, but by adding dissimilar coprocessors, usually incorporating specialized processing capabilities to handle particular tasks. Usually heterogeneity in the context of computing refers to different instruction-set architectures (ISA), where the main processor has one and other processors have another - usually a very different - architecture (maybe more than one), not just a different microarchitecture [19].

Zynq-7000 SoC The Zynq-7000 All Programmable SoC family integrates the software programmability of an ARM-based processor with the hardware programmability of an FPGA, enabling key analytics and hardware acceleration while integrating CPU, DSP, ASSP, and mixed signal functionality on a single device. Consisting of dual-core Zynq-7000 devices, the Zynq-7000 devices provide excellent performance-per-watt and maximum design flexibility.

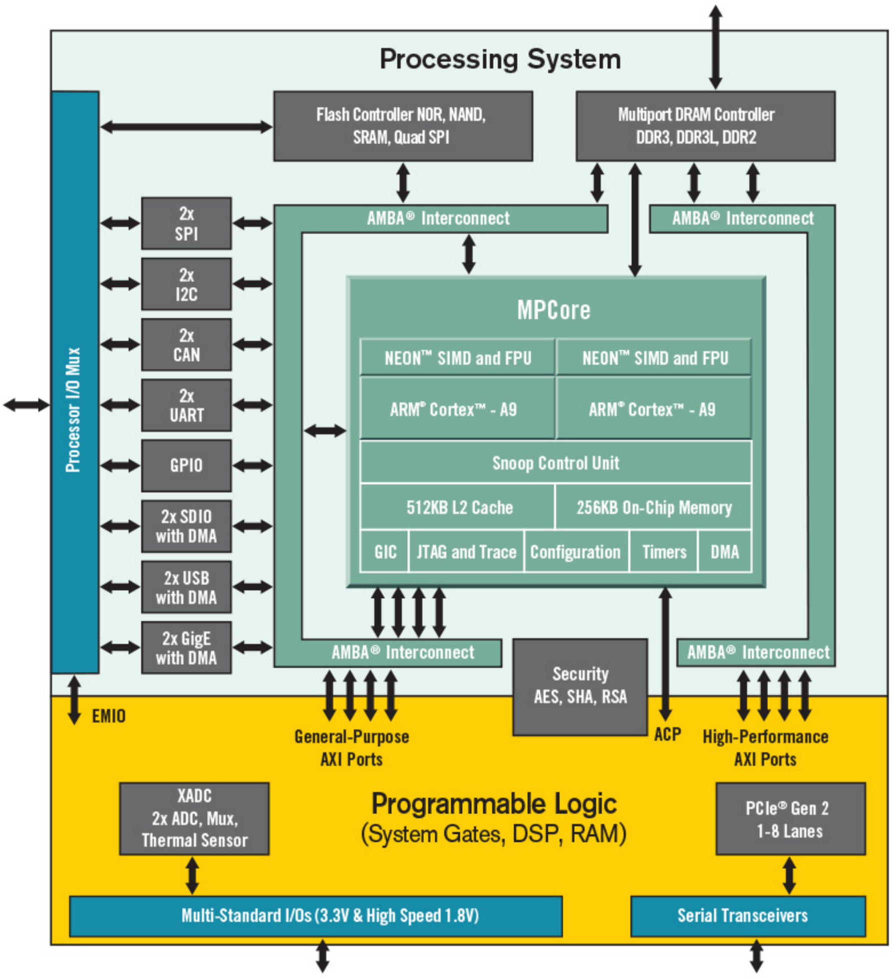


Figure 2.11: Zynq 7000 SoC [20].

Zynq 7000 Processing System (PS)

- Application Processing Unit (APU): At the basis, a dual-core ARM Cortex-A9 processor comprises the APU. As additional functionality to the main ARM processor, the APU contains NEON engines that provide Single Instruction Multiple Data (SIMD)[1] facilities to enable strategic acceleration of media and DSP type algorithms. NEON instructions are an extension to the standard ARM instruction set, and can either be used explicitly, or by ensuring that the C code follows an expected form and thus allows NEON operations to be inferred by the compiler. In addition to NEON, there are also extensions for the Floating Point Unit (FPU). These are referred to as Floating Point Extensions, or sometimes VFP Extensions (Vector Floating Point) for historical reasons. The unit provides hardware acceleration of floating point operations in compliance with the IEEE 754 standard and supports single and double precision formats, with some additional support for half-precision and integer conversion [20].

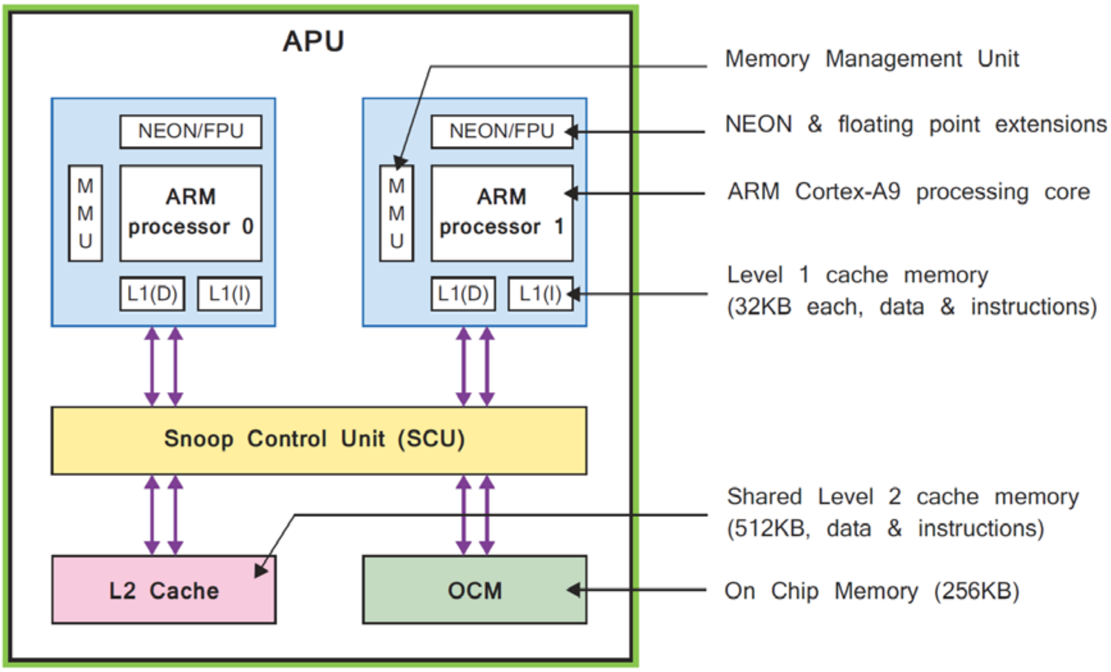

APU
NEON/FPU
MMU
ARM processor 0
L1(D)
L1(I)
NEON/FPU
MMU
ARM processor 1
L1(D)
L1(I)
Snoop Control Unit (SCU)
L2 Cache
OCM
Memory Management Unit
NEON & floating point extensions
ARM Cortex-A9 processing core
Level 1 cache memory (32KB each, data & instructions)
Shared Level 2 cache memory (512KB, data & instructions)
On Chip Memory (256KB)

Figure 2.12: Simplified block diagram of Zynq 7000 APU [20].

- PS External Interfaces: The Zynq PS features a variety of interfaces, both between the PS and PL, and between the PS and external components such as peripheral interfaces, cache memory, memory interfaces, interconnect, and clock generation circuitry. Information about each of these interfaces is presented in Table 2.2 [20].

[1] SIMD is where one instruction acts on multiple data items carrying out the same operation for all data.

Table 2.2: Comparison of different CNN Topologies for Image Classification on ImageNet

I/O Interface	Description
SPI	Serial Peripheral Interface
I2C	I2C bus
CAN	Controller Area Network
UART	Universal Asynchronous Receiver Transmitter
GPIO	General Purpose Input/Output
SD	SD card memory interface
USB	Universal Serial Bus
GigE	Ethernet MAC peripheral

Zynq 7000 Programmable Logic (PL)

- Soft Blocks: The PL is predominantly composed of general purpose FPGA logic fabric, which is composed of slices and Configurable Logic Blocks (CLBs), Lookup Tables (LUT), Flip-flops (FF) and there are also Input/Output Blocks (IOBs) for interfacing.
- Hard Blocks: In addition to the general fabric, there are two special purpose components: Block RAMs for dense memory requirements and DSP slices for high-speed arithmetic. Both of these resources are integrated into the logic array in a column arrangement, embedded into the fabric logic

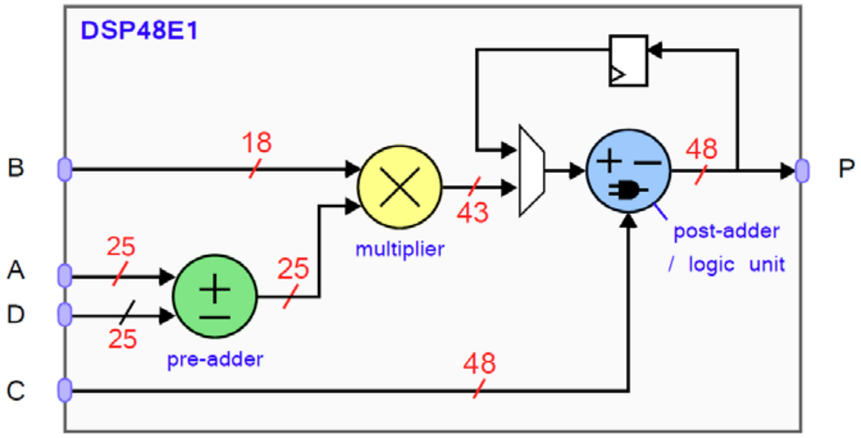


Figure 2.13: The DSP48E1 slice [20].

2.4 Caffe Framework

Caffe is a deep learning framework distinguished for its expression, speed, and modularity. It is developed by Berkeley AI Research (BAIR) and by community contributors. Caffe supports many different types of deep learning architectures geared towards image classification and image segmentation. It supports CNN, RCNN, LSTM and fully connected neural network designs designed in a different APIs (C++, Python, Matlab) and it is widely used in large-scale industrial applications such as vision, speech, and multimedia.

2.4.1 Anatomy of Caffe

Deep networks are compositional models that are naturally represented as a collection of inter-connected layers that work on chunks of data. Caffe defines a net layer-by-layer in its own model schema. The network defines the entire model bottom-to-top from input data to loss. As data and derivatives flow through the network in the forward and backward passes Caffe stores, communicates, and manipulates the information as *blobs*: the blob is the standard array and unified memory interface for the framework. The layer comes next as the foundation of both model and computation. The net follows as the collection and connection of layers. The details of blob describe how information is stored and communicated in and across layers and nets [21].

The Blob A Blob is a wrapper over the actual data being processed and passed along by Caffe, and also under the hood provides synchronization capability between the CPU and the GPU. Mathematically, a blob is an N-dimensional array stored in a C-contiguous fashion. Caffe stores and communicates data using blobs. Blobs provide a unified memory interface holding data; e.g., batches of images, model parameters, and derivatives for optimization. Blobs conceal the computational and mental overhead of mixed CPU/GPU operation by synchronizing from the CPU host to the GPU device as needed. Memory on the host and device is allocated on demand (lazily) for efficient memory usage. The conventional blob dimensions for batches of image data are number *N x channel K x height H x width W*. Blob memory is row-major in layout, so the rightmost dimension changes fastest as seen in the following equation. For example, in a 4D blob, the value at index *(n, k, h, w)* is physically located at index *((n * K + k) * H + h) * W + w*.[2] Also there are two different ways to access them: the constant way, which does not change the values, and the mutable way, which changes the values [21].

[2] Number N is the batch size of the data. Batch processing achieves better throughput for communication and device processing. For an ImageNet training batch of 256 images N = 256.
Channel K is the feature dimension e.g. for RGB images K = 3.

Net connections and operation The layer is the essence of a model and the fundamental unit of computation. The net is a set of layers connected in a computation graph – a directed acyclic graph (DAG) to be exact. Layers convolve filters, pool, take inner products, apply nonlinearities like rectified-linear and sigmoid and other elementwise transformations, normalize, load data, and compute losses like softmax. Each layer type defines three critical computations: setup, forward, and backward [21].

- Setup: initialize the layer and its connections once at model initialization.
- Forward: given input from bottom compute the output and send to the top.
- Backward: given the gradient computes the gradients with respect to the parameters and to the inputs, which are in turn back-propagated to earlier layers.

Caffe does all the bookkeeping for any DAG of layers to ensure correctness of the forward and backward passes. Also developing custom layers requires minimal effort by the compositionality of the network and modularity of the code. A typical net begins with a data layer that loads from disk and ends with a loss layer that computes the objective for a task such as classification or reconstruction.

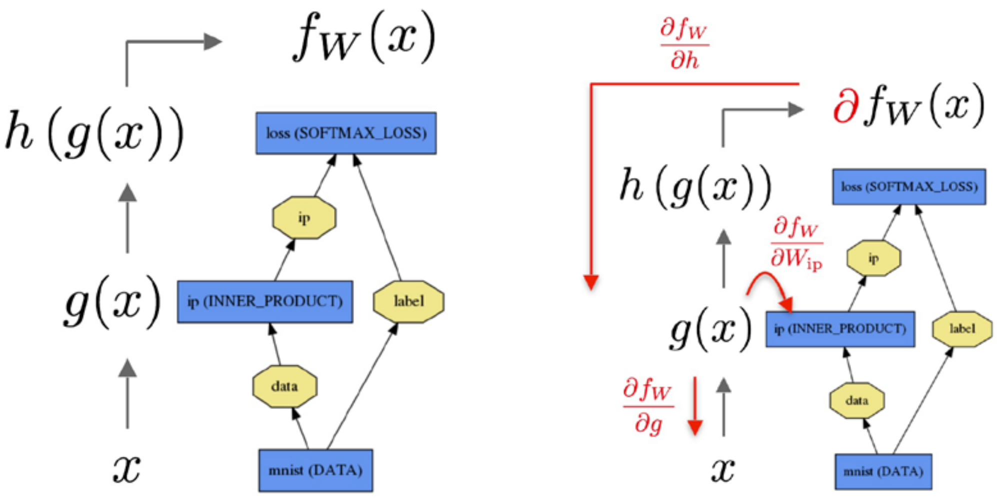


Figure 2.14: Caffe forward pass (left) and backward pass (right) on a simple logistic classifier [21].

Model format The models are defined in plaintext protocol buffer schema (*prototxt*) while the learned models are serialized as binary protocol buffer (*binaryproto*) *caffemodel* files. The model format is defined by the protobuf schema in caffe.proto. Caffe uses Google Protocol Buffer for the following strengths: minimal-size binary strings when serialized, efficient serialization, a human-readable text format compatible with the binary version, and efficient interface implementations in multiple languages, most notably C++ and Python. This all contributes to the flexibility and extensibility of modeling in Caffe.

2.4.2 Operation of the framework

As stated before, the forward and backward passes are the essential computations of a network. The forward pass computes the output of a layer given the input for inference. In forward, Caffe composes the computation of each layer to compute the “function” represented by the model going from bottom to top layers. In contrary, the backward pass computes the gradient given the loss for learning. In backward, Caffe reverse-composes the gradient of every layer to compute the gradient of the whole model by automatic differentiation. This is back-propagation and goes from top to bottom. The backward pass begins with the loss and computes the gradient with respect to the output $\frac{\partial f_w}{\partial h}$. The gradient with respect to the rest of the model is computed layer-by-layer through the chain rule. Layers with parameters, like the inner product layer, compute the gradient with respect to their parameters $\frac{\partial f_w}{\partial W_{ip}}$ during the backward step. Every layer type has *forward_{cpu,gpu}()* and *backward_{cpu,gpu}()* methods to compute its steps according to the mode of computation. A layer may only implement CPU or GPU mode due to constraints or convenience [21].

Caffe framework include many different layers but the basic categories are the following:

- Data Layers: data enters Caffe through data layers. They lie at the bottom of nets. Data can come from efficient databases (LevelDB or LMDB), directly from memory, or, when efficiency is not critical, from files on disk in HDF5 or common image formats. Common input preprocessing (mean subtraction, scaling, random cropping, and mirroring) is available by specifying transformation parameters by some of the layers [9].
- Vision Layers: vision layers (ex. convolution, pooling, crop etc.) usually take images as input and produce other images as output, although they can take data of many other types and dimensions. A typical “image” in the real-world may have one color channel ($c = 1$), as in a grayscale image, or three color channels ($c = 3$) as in an RGB (red, green, blue) image. But in this context, the distinguishing characteristic of an image is its spatial structure: usually an image has some non-trivial height $h > 1$ and width $w > 1$. This 2D geometry naturally lends itself to certain decisions about how to process the input. In particular, most of the vision layers work by applying a particular operation to some region of the input to produce a corresponding region of the output. In contrast, other layers (with few exceptions) ignore the spatial structure of the input, effectively treating it as “one big vector” with dimension $c \cdot h \cdot w$ [21].
- Activation Layers: in general, activation layers (ex. ReLU, Sigmoid, Bias etc.) are element-wise operators, taking one bottom blob and producing one top blob of the same size.
- Loss Layers: they drive learning by comparing an output to a target and assigning cost to minimize. The loss itself is computed by the forward pass and the gradient is computed by the backward pass.

2.4.3 Caffe tools

Caffe has command line, Python, and MATLAB interfaces for day-to-day usage, interfacing with research code, and rapid prototyping but in this work we are interested only with Command Line and Python interfaces which we describe next [21].

Command Line Interface

Training The command line interface provides the *caffe* tool for model training, scoring, and diagnostics. This tool is found under *caffe/build/tools* after compilation of this interface. Caffe train and learns models from scratch, resumes learning from saved snapshots, and fine-tunes models to new data and tasks. There are 4 steps in training a CNN using Caffe:

- Step 1 - Data preparation: In this step, we clean the images and store them in a format that can be used by Caffe.
- Step 2 - Model definition: In this step, we choose a CNN architecture to deploy and we define its parameters in a configuration file with extension *.prototxt*.
- Step 3 - Solver definition: The solver is responsible for model optimization. We define the solver parameters in a configuration file with extension *.prototxt*.
- Step 4 - Model training: We train the model by executing one Caffe command from the terminal. After training the model, we will get the trained model in a file with extension *.caffemodel* or we can pause training and save snapshot in a *.solverstate* file.

After the training phase, the *.caffemodel* trained model is used to make predictions of new unseen data. For example in order to train a network using LeNet network on 2nd GPU we can run:

```
#  caffe train -solver examples/mnist/lenet_solver.prototxt -gpu 2
```

Testing Caffe *test* scores models by running them in the test phase and reports the net output as its score. The net architecture must be properly defined to output an accuracy measure or loss as its output. The per-batch score is reported and then the grand average is reported last.
For example to score the learned LeNet model on the validation set as defined in the model architeture lenet_train_test.prototxt we can run:

```
#    caffe    test    -model    examples/mnist/lenet_train_test.prototxt    -weights
examples/mnist/lenet_iter_10000.caffemodel -gpu 0 -iterations 100
```

Benchmarking Caffe *time* benchmarks model execution layer-by-layer through timing and synchronization. This is useful to check system performance and measure relative execution times for models
For example in order to time a model architecture with the given weights on the first GPU for 10 iterations we can run the following command:

```
#    caffe    time    -model    examples/mnist/lenet_train_test.prototxt    -weights
examples/mnist/lenet_iter_10000.caffemodel -gpu 0 -iterations 10
```

Diagnostics Caffe *device_query* reports GPU details for reference and checking device ordinals for running on a given device in multi-GPU machines.
For example to query the first device:

```
# caffe device_query -gpu 0
```

Python Interface

The Python interface *pycaffe* is the Caffe module and its scripts are in *caffe/python* folder after the compilation of this interface. With this API we can import Caffe to load models, do forward and backward computations, handle IO, visualize networks, and even instrument model solving. All model data, derivatives, and parameters are exposed for reading and writing [9]. Some basic facts of this interface are:

- *caffe.Net* which is the central interface for loading, configuring, and running models. Also *caffe.Classifier* and *caffe.Detector* provide convenience interfaces for common tasks.
- *caffe.SGDSolver* exposes the solving interface.
- *caffe.io* handles input / output with preprocessing and protocol buffers.
- *caffe.draw* visualizes network architectures.
- Caffe blobs are exposed as *numpy* ndarrays for ease-of-use and efficiency.

Also IPython notebooks are found in *caffe/examples* with many features.

Classification example

Caffe, at its core, is written in C++. It is possible to use the C++ API of Caffe to implement an image classification application as well as Python code presented in one of the Notebook examples. For example we can run a classification command using a simple C++ code that is proposed in *examples/cpp_classification* folder. The C++ example is built automatically when compiling Caffe as a *classification.bin* file.

We can first download a pre-trained CaffeNet model to use with the classification example, from the *Model Zoo*[2] using the following script:

```
# ./scripts/download_model_binary.py models/bvlc_reference_caffenet
```

The ImageNet labels file, also known as the *synset* file, is also required in order to map a prediction to the name of the class:

```
# ./data/ilsvrc12/get_ilsvrc_aux.sh
```

Using the files that were downloaded, we can classify the provided cat image (*examples/images/cat.jpg*) using this command:

```
# ./build/examples/cpp_classification/classification.bin \
  models/bvlc_reference_caffenet/deploy.prototxt \
  models/bvlc_reference_caffenet/bvlc_reference_caffenet.caffemodel \
  data/ilsvrc12/imagenet_mean.binaryproto \
  data/ilsvrc12/synset_words.txt \
  examples/images/cat.jpg
```

[2] Model Zoo is a database with models that are learned and applied for problems ranging from simple regression, to large-scale visual classification, to Siamese networks for image similarity, to speech and robotics applications. They share a standard format for packaging in Caffe and are found in a central wiki page for sharing info Gists.

3 Caffe in Embedded SoC

3.1 Caffe in Zynq SoC

Embedded systems typically have very specific requirements and constraints such as limited power and energy budgets, small physical sizes resulting in high reliability requirements and hard real-time constraints [22]. This chapter gives a detailed information on how to port Caffe framework on embedded SoCs and specifically on ARM CPU of Xilinx Zynq 7000 SoC family with a presented novel scheme to easily migrate a DNN running in Caffe in the embedded system [23]. The first step in order to port the whole Caffe framework into the Zynq 7000 SoC to run on ARM (dual-core) was to cross-compile the required library dependencies of Caffe [1] and then the whole framework and link all the libraries using the ARM cross compiler that was included in the SDSoC Environment. The second step was to find a suitable DNN model architecture to fit into the limited memory of the embedded SoC. For the building of the framework, the interface was kept unchanged and the only part that had to be modified was the *Makefile* configuration of Caffe source code.

3.1.1 Creating the boot image

A light Linux image was selected for boot while having all the necessary system software needed for developing and configuring application. PetaLinux image which is created with Xilinx Software Development Kit was appropriate for the embedded Linux system. PetaLinux consists of three key elements: pre-configured binary bootable images, fully customizable Linux for the Xilinx device, and PetaLinux SDK which includes tools and utilities to automate complex tasks across configuration, build, and deployment.

[1] Caffe cannot run with the classic x64 libraries because we have a different architecture for the embedded system.

3.1.2 Cross-compiling 3d party libraries for ARM

The cross compilation of the libraries we have done was manual, meaning we downloaded the source files of each library (from github page) that Caffe needs and is dependent and then cross compiled them one by one using the ARM toolchain ”*arm-linux-gnueabihf*” of SDSoC (defining the appropriate configuration each time and overcoming several complications in some library installations) producing the header files and dynamic libraries at the end.
More specific, the Zynq SoC with the ARM processor supports NEON instrinsics so we set the arm cross-compiler with the *-mfpu=neon* directive to enable them during the compilation for all libraries in order to benefit from the SIMD architecture extension. Also, our ARM CPU had hardware support for floating point operations so we configured the toolchain to enable the calling Application Binary Interface (ABI) [3] to pass float variables across calls and thus use the Vector Floating Point (VFP) registers with *-mfloat-abi=hard* compiler instruction. It's also worth mentioning that some libraries required to modify a configuration file to bootstrap the code first before running the Makefile and install them while others required to control the software compilation with *CMake* tool.[4] The header files and dynamic libraries were installed in an appropriate build directory that we created for each library. Below we list the compilation procedure of all the required libraries of Caffe.

Gflags The Gflags package contains a C++ library that implements command-line flags processing. It includes built-in support for standard types such as string and the ability to define flags in the source file in which they are used [24]. For the compilation we created the following CMake file to use as a *toolchain* file for CMake with the following directives:

```
SET(CMAKE_SYSTEM_NAME Linux)
SET(CMAKE_C_COMPILER arm-linux-gnueabihf-gcc)
SET(CMAKE_CXX_COMPILER arm-linux-gnueabihf-g++)
SET(CMAKE_FIND_ROOT_PATH /opt/Xilinx/SDx/2016.4/SDK/gnu/aarch32/lin/gcc-arm-linux-
gnueabi/arm-linux-gnueabihf)
SET(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER)
SET(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
SET(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)
```

In the build folder then we run:

```
# cmake -DCMAKE_TOOLCHAIN_FILE=/home/jimakos/Desktop/gflags-
master/build/arm_make.cmake -DCMAKE_INSTALL_PREFIX=/home/jimakos/Desktop/gflags-
master/build -DBUILD_SHARED_LIBS=ON ..
# make
# sudo make install
```

OpenBlas It is an optimized BLAS library that provides standard building blocks for performing basic vector and matrix operations [25]. For the compilation we ran:

```
# make CC=arm-linux-gnueabihf-gcc HOSTCC=gcc ONLY_CBLAS=1 TARGET=ARMV7
# sudo make PREFIX=/home/jimakos/Desktop/openblas_libs/build/ install
```

[3] An ABI is an interface between two program modules that defines how data structures or computational routines are accessed in machine code, which is a low-level, hardware-dependent format [26].

[4] CMake is an open-source, cross-platform family of tools designed to build, test and package software using simple platform and compiler independent configuration files, and generate native makefiles [27].

Glog This library contains a C++ implementation of the Google logging module [28]. First we had to configure the Makefile appropriately before running the installation.

```
# ./autogen.sh
# ./configure CC=arm-linux-gnueabihf-gcc CXX=arm-linux-gnueabihf-g++ --host=arm-
linux-gnueabihf --prefix=/home/jimakos/Desktop/glog_libs/build/
# make
# make install
```

Boost It is a set of libraries for the C++ programming language that provide support for tasks and structures such as linear algebra, multithreading, image processing, regular expressions, and unit testing etc. [29]. First, we had to bootstrap the code:

```
# ./bootstrap.sh
```

Then edit the configuration file "*project-build.jam*" to use the ARM toolchain by replacing the line with "using gcc" by:

```
using gcc : arm : arm-linux-gnueabihf-g++ ;
```

Then in order to build and install the boost libraries we ran:

```
# ./bjam install toolset=gcc-arm --prefix=/home/jimakos/Desktop/boost/build/
```

Szip It is a compression library [30] essential for some libraries of Caffe.

```
# ./configure CC=arm-linux-gnueabihf-gcc CXX=arm-linux-gnueabihf-g++ --host=arm-
linux-gnueabihf --prefix=/home/jimakos/Desktop/szip-master/build/
# make
# make install
```

LMDB It is a library that provides a high-performance embedded transactional database in the form of a key-value store [31] and will be used for the training and validation datasets in Caffe. We modified Makefile changing only the following directives:

```
CC = arm-linux-gnueabihf-gcc
AR = arm-linux-gnueabihf-ar
Prefix = =/home/jimakos/Desktop/lmdb-master/build/
```

Zlib It is a general purpose data compression library [32] that is essential for some libraries of Caffe. For the compilation we configured the Makefile and continued with the installation.

```
# CC=arm-linux-gnueabihf-gcc ./configure --prefix=/home/jimakos/Desktop/zlib-
master/build/
# make
# make install
```

Protobuf Protocol Buffers (a.k.a., protobuf) are Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data [33]. Unfortunately, Protobuf libraries don't support cross-compilation well. The major problem here was to figure out the right way for *protoc* file to be built for the desktop machine, whereas the libraries needed to be built for the ARM CPU. So we installed the desktop version first and then cross-compiled using the x64 *protoc* bin file.

```
# ./configure --prefix=/home/jimakos/Desktop/protobuf/install/
# make
# make install
# make distclean
# ./configure --host=arm-linux --prefix=/home/jimakos/Desktop/protobuf/install/ARM
--with-protoc=/home/jimakos/Desktop/protobuf/install/bin/protoc
# make
# make install
```

Snappy It is a compression/decompression library that aims for very high speeds with reasonable compression [34]. For the compilation we created the following CMake file to use as a toolchain file for *cmake* command containing the following directives:

```
SET(CMAKE_SYSTEM_NAME Linux)
SET(CMAKE_C_COMPILER arm-linux-gnueabihf-gcc)
SET(CMAKE_CXX_COMPILER arm-linux-gnueabihf-g++)
SET(CMAKE_FIND_ROOT_PATH /opt/Xilinx/SDx/2016.4/SDK/gnu/aarch32/lin/gcc-arm-linux-
gnueabi/arm-linux-gnueabihf)
SET(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER)
SET(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
SET(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)
```

In the build folder then we run:

```
# cmake -DCMAKE_TOOLCHAIN_FILE=/home/jimakos/Desktop/snappy-
master/build/arm_make.cmake -DCMAKE_INSTALL_PREFIX=/home/jimakos/Desktop/snappy-
master/build -DBUILD_SHARED_LIBS=ON ..
# make
# sudo make install
```

LevelDB LevelDB is a fast key-value storage library written at Google that provides an ordered mapping from string keys to string values [35]. We modified *build_detect_platform* file adding the following directives just before “COMMON_FLAGS”:

```
CC = arm-linux-gnueabihf-gcc
CXX = arm-linux-gnueabihf-g++
TMPDIR = =/home/jimakos/Desktop/leveldb-master/temp/
```

Then link with the required *snappy* arm library that we compiled before because Caffe uses LevelDB library with Snappy data compression. So we edit the following line:

```
PLATFORM_LIBS = =″-L/home/jimakos/Desktop/snappy-master/build/ -lsnappy″
```

Also we had to enter the full path of the include file “snappy.h” from our snappy build folder. So we changed the include command at line 210 to:

```
#include “/home/jimakos/Desktop/snappy-master/snappy.h″
```

OpenCV This library is an Open Source Computer Vision Library (OpenCV) and has C++, Python and Java interfaces supporting Windows, Linux, Mac OS, iOS and Android. OpenCV was designed for computational efficiency with a strong focus on real-time applications taking advantage of multi-core processing systems [36]. For the compilation, we created again the following CMake file to use as a toolchain file for cmake command containing the following directives:

```
SET(CMAKE_SYSTEM_NAME Linux)
SET(CMAKE_C_COMPILER arm-linux-gnueabihf-gcc)
SET(CMAKE_CXX_COMPILER arm-linux-gnueabihf-g++)
SET(CMAKE_FIND_ROOT_PATH /opt/Xilinx/SDx/2016.4/SDK/gnu/aarch32/lin/gcc-arm-linux-
gnueabi/arm-linux-gnueabihf)
SET(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER)
SET(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
SET(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)
```

In the build folder then we run:

```
# cmake -DCMAKE_TOOLCHAIN_FILE=/home/jimakos/Desktop/opencv-
master/build/arm_make.cmake -DCMAKE_INSTALL_PREFIX=/home/jimakos/Desktop/opencv-
master/build -DBUILD_SHARED_LIBS=ON ..
```

Also we run *ccmake* command and enable jpeg support to read jpeg files and tbb to enable tbb threads and take advantage of multi-core ARM processor. Then we built with:

```
# make
# sudo make install
```

HDF5 This package is a data model, library, and file format for storing and managing data. It supports an unlimited variety of datatypes, and is designed for flexible and efficient I/O and for high volume and complex data [37].

There are two major problems with cross-compiling in HDF5. First, a macro in HDF5 tries to compile and run a test program, but this does not work for cross-compiling because it builds the program for the host system, and tries to run it on the build system. For example, HDF5 checks to see if the Fortran compiler supports the intrinsic function "SIZEOF" by running a test program. Based on the result, a Makefile conditional is set to toggle which source file to use when building H5test_kind. There are also many C++ compiler checks which can include checking if large files are supported, checking if SZIP compression can encode, checking if *gettimeofday* uses the timezone struct, and many more for checking conversion capabilities. The second problem is the generation of *H5Tinit.c* and, to a lesser extent, *H5libsettings.c* which are actually generated within *make* command, not by *configure*. The programs that generate them are C programs which are compiled to run on the target platform, but they are then run during *make* on the build platform, and thus fail (or, in some cases, simply produce incorrect results). HDF5 would need to generate these source files during configure without executing a machine-dependent program on the build system. Fortunately, we present a method to resolve these issues.

First, as with other library compilations, we created the following CMake file to use as a *toolchain* file for cmake command containing the following directives:

```
SET(CMAKE_SYSTEM_NAME Linux)
SET(CMAKE_C_COMPILER arm-linux-gnueabihf-gcc)
SET(CMAKE_CXX_COMPILER arm-linux-gnueabihf-g++)
SET(CMAKE_FIND_ROOT_PATH /opt/Xilinx/SDx/2016.4/SDK/gnu/aarch32/lin/gcc-arm-linux-
gnueabi/arm-linux-gnueabihf)
SET(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER)
SET(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
SET(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)
```

Then the following CMake command was run a couple of times until the configuring is done successfully. Also, Caffe uses HDF5 library demanding to be compiled with I/O external filters specifically *deflate* and *decode*. So we include and link the appropriate external ARM libraries *zlib* and *szip* that we previously compiled with the following options in cmake command in order to enable their support:

```
cmake -DCMAKE_TOOLCHAIN_FILE=arm_make.cmake -
DCMAKE_INSTALL_PREFIX=/home/jimakos/Desktop/hdf5-1.10.1/build -
DBUILD_SHARED_LIBS=ON -DHDF5_ENABLE_Z_LIB_SUPPORT:BOOL=ON -
DZLIB_INCLUDE_DIR:PATH="/home/jimakos/Desktop/zlib-master/build/include" -
DZLIB_LIBRARY:FILEPATH="/home/jimakos/Desktop/zlib-master/build/lib/libz.so" -
DHDF5_ENABLE_SZIP_SUPPORT:BOOL=ON -
DSZIP_INCLUDE_DIR:PATH="/home/jimakos/Desktop/szip-master/build/include" -
DSZIP_LIBRARY:FILEPATH="/home/jimakos/Desktop/szip-master/build/lib/libsz.so" ..
```

After that, the make command was run a couple of times to ensure that all files are created and we ignored the errors temporarily. Then we copy the files *libhdf5.settings, H5detect* and *H5make_libsettings* to our Zynq SoC board with the ARM processor. In order to generate the needed files we ran there the following commands on board:

```
# ./H5make_libsettings > ./tmp/H5lib_settings.c
# ./H5detect > ./tmp/H5T_init.c
```

Then all we had to do was transfer the two C files back to the desktop system from the device system and successfully run the make command and finish the build.

JPEG This package contains C software to implement JPEG[5] image encoding, decoding and transcoding [38]. This set of library is required for Caffe to read and write JPEG image files.

```
# ./configure --host=arm-linux-gnueabihf CC=arm-linux-gnueabihf-gcc --
prefix=/home/jimakos/Desktop/jpeg-master/build/
# make
# make install
```

3.1.3 Cross-compiling Caffe for ARM

Caffe provides a clear modular Makefile in order to adapt it seamlessly in many platforms. Though it does not officially support ARM compilation we provide a method to port it to the ARM processor of Zynq SoC using arm cross-compiler toolchain.

The first step was to place all the cross-compiled libraries of Caffe to a new custom folder of ours that will be linked during the build process and modify properly the configuration file of Makefile. The Zynq SoC does not have any GPU card embedded so we uncommented the directive *CPU_ONLY := 1* in order to enable the build only for the CPU processor. On the same principle we commented out the lines that had to do with settings for CUDA architecture. Then we set BLAS choice as *open* to use the OpenBlas library which is generally supported on ARM systems and set the include and library folders of BLAS to point on our new custom folder which contained all the cross-compiled libraries for Caffe. Also we set the include and library directories to point on that folder commenting out any other instruction that pointed on the original x86 library folders (ex. */usr/local/lib/ or /usr/local/include/*).

Furthermore, the modification of the Makefile had to be done. The appropriate directives were set at the beginning of the file (line 9) to trigger the ARM toolchain. It is worth mentioning that the rpath-link[6] folder had to be set to point in our library folder to make it the same path as the general libraries. This is used because some shared objects that we built, have encoded a wrong path to some executable arguments and thus the correct shared objects must be located at runtime. Lastly, any static linking directives were commented out and all dynamic libraries were included and linked with the appropriate name. The directives follow:

```
TOOLCHAIN := arm-linux-gnueabihf-
CC := $(TOOLCHAIN)gcc
CXX := $(TOOLCHAIN)g++ -Wl,-rpath-link=/home/jimakos/Desktop/third-party/lib/
AR := $(TOOLCHAIN)ar
LD := $(TOOLCHAIN)ld -Wl,-rpath-link=/home/jimakos/Desktop/third-party/lib/
CXXFLAGS += -Wno-unused-local-typedefs -DSDS
```

[5] JPEG (pronounced "jay-peg") is a standardized compression method for full-color and gray-scale images.

[6] rpath designates the run-time search path hard-coded in an executable file or library. Dynamic linking loaders use the rpath to find required libraries [39].

3.2 DNN architectures for embedded SoC

The tight constraints on employing DNNs on small embedded systems can include the limited on-chip memory of the device. FPGAs often have less than 10MB of on- chip memory. For inference for example, a sufficiently small model could be stored directly on the FPGA instead of being bottlenecked by memory band-width. In this section we try to find the “right" DNN, the one that offers acceptable accuracy but operates in real-time within power and energy constraints of its target embedded application.

3.2.1 Memory optimized networks

We tried to use small DNN architectures which will be more feasible to deploy on FPGAs (Xilinx ZC702 board was used) and other hardware with limited memory.

- SqueezeNet is proposed as a small 18-layer DNN alternative which achieves AlexNet-level accuracy on ImageNet with 50x fewer parameters. This parameter reduction has been accomplished by using fire modules and careful balancing of the architecture. One forward pass requires only 860 million MACC operations and 1.24 million parameters are enough to achieve 80.3% top-5 accuracy [40].
- BVLC GoogleNet Model with its 50MB weights was suitable for Caffe in the embedded system as well. This model is a replication of the model described in the GoogleNet publication [41]. The differences are:
    - not training with the relighting data-augmentation
    - not training with the scale or aspect-ratio data-augmentation
    - uses "xavier" to initialize the weights instead of "gaussian"
    - *quick_solver.prototxt* uses a different learning rate decay policy than the original *solver.prototxt*, that allows a much faster training

  The bundled model is the iteration 2,400,000 snapshot (60 epochs) using *quick_solver.prototxt*. This model obtains a top-1 accuracy 68.7% (31.3% error) and a top-5 accuracy 88.9% (11.1% error) on the validation set, using just the center crop.
- Larger neural networks with more parameters such as BVLC Reference Caffenet or VGG Net could not fully run on the device due to memory overhead. An error on the terminal was thrown every time sending this message:

  ```
  terminate called after throwing an instance of 'std::bad_alloc'
  what(): std::bad_alloc
  *** Aborted
  ```

  A way to get around this problem might was to build the linux image with an increased *rootfs*[7] by changing options in 'Root filesystem type' inside petalinux configuration.

[7] The root file system (rootfs) is the most basic component of Linux. A root file system contains everything needed to support a full Linux system: all the applications, configurations, devices, data, and more [42].

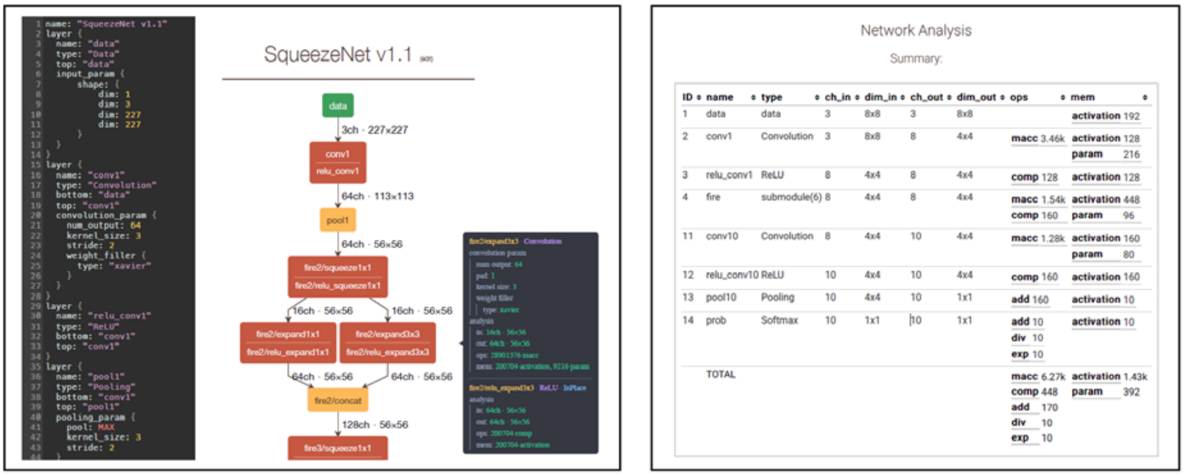
3.2.2 Accuracy optimized networks

In this section, we outline the design strategies for CNN architectures with few parameters in order to maintain competitive accuracy.

- SqueezeNet employs specific strategies to achieve significant accuracy. It replaces 3x3 filters with 1x1 filters in the convolution layers as a 1x1 filter has 9X fewer parameters than a 3x3 filter. Also downsample late in the network so that convolution layers have large activation maps. In a convolutional network, if layers have large strides, then most layers will have small activation maps. Conversely, if most layers in the network have a stride of 1, and the strides greater than 1 are concentrated toward the end of the network, then many layers in the network will have large activation maps. The idea is that large activation maps (due to delayed downsampling) can lead to higher classification accuracies [43].
- GoogleNet architecture leverages the network accuracy with its approach. It was found that a move from fully connected layers to average pooling improved the top-1 accuracy by about 0.6%, however the use of dropout remained essential even after removing the fully connected layers [41].

3.2.3 Final Verdict

The most suitable network for our embedded system was the SqueezeNet v1.1. As described in the previous sections, this DNN architecture offers many advantages to deploy in small devices as for the speed and efficiency as well as the accuracy. To analyze the insides of the network, we used Netscope CNN Analyzer, a web-based tool for visualizing and analyzing convolutional neural network architectures taking as inputs Caffe's prototxt format files [44]. Although Caffe includes a python script “draw_net.py” which draws the network structure, it doesn’t do any analysis, and excel tables tend to disintegrate after a short time.



```
name: "SqueezeNet v1.1"
layer {
  name: "data"
  type: "Data"
  top: "data"
  input_param {
      shape: {
          dim: 1
          dim: 3
          dim: 227
          dim: 227
      }
  }
}
layer {
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  convolution_param {
    num_output: 64
    kernel_size: 3
    stride: 2
    weight_filler {
      type: "xavier"
    }
  }
}
layer {
  name: "relu_conv1"
  type: "ReLU"
  bottom: "conv1"
  top: "conv1"
}
layer {
  name: "pool1"
  type: "Pooling"
  bottom: "conv1"
  top: "pool1"
  pooling_param {
    pool: MAX
    kernel_size: 3
    stride: 2
  }
```

fire2/expand3x3 · Convolution
convolution param
num output: 64
pad: 1
kernel size: 3
weight filler
type: xavier
analysis
in: 16ch · 56x56
out: 64ch · 56x56
ops: 28901376 macc
mem: 200704 activation, 9216 param

fire2/relu_expand3x3 · ReLU · InPlace
analysis
in: 64ch · 56x56
out: 64ch · 56x56
ops: 200704 comp
mem: 200704 activation

Network Analysis

Summary:

ID	name	type	ch_in	dim_in	ch_out	dim_out	ops	mem
1	data	data	3	8x8	3	8x8		**activation** 192
2	conv1	Convolution	3	8x8	8	4x4	**macc** 3.46k	**activation** 128 **param** 216
3	relu_conv1	ReLU	8	4x4	8	4x4	**comp** 128	**activation** 128
4	fire	submodule(6)	8	4x4	8	4x4	**macc** 1.54k **comp** 160	**activation** 448 **param** 96
11	conv10	Convolution	8	4x4	10	4x4	**macc** 1.28k	**activation** 160 **param** 80
12	relu_conv10	ReLU	10	4x4	10	4x4	**comp** 160	**activation** 160
13	pool10	Pooling	10	4x4	10	1x1	**add** 160	**activation** 10
14	prob	Softmax	10	1x1	10	1x1	**add** 10 **div** 10 **exp** 10	**activation** 10
	TOTAL						**macc** 6.27k **comp** 448 **add** 170 **div** 10 **exp** 10	**activation** 1.43k **param** 392

Figure 3.1: Visualization of the layer-level Network Graph (left), Analysis Summary Table (right) [6].

4 FPGA Accelerator Design

One of the main challenges for embedded system designers is to find a tradeoff between performance and power consumption of an application. In order to reach this goal, hardware accelerators have been used to offload specific tasks from the CPU, improving the global performance of the system and reducing its dynamic power consumption. FPGAs take the architectural approach even further by combining logic blocks and interconnects of traditional FPGAs with embedded microprocessors known as System on a Chip (SoC). While most software today is written so that instructions are executed in sequence, FPGAs have a reconfigurable architecture which enables the parallelization of specific computational intensive code in hardware. These algorithms are not controlled by instruction fetch[1] like traditional processor's and are usually repetitive and intensive.

In this section, we analyze the hardware accelerator that we developed in Xilinx SDSoC to run on FPGA, aiming to accelerate and outperform the image classification performance on Caffe framework when running on ARM CPU of the Zynq SoC. In order to define the function for acceleration we first executed a profiling on Caffe framework to identify the memory or computation bottlenecks. Then, Xilinx SDSoC Development Environment was used to implement, validate and optimize the hardware function and integrate it with the whole Caffe framework. The function is called GEMM (General Matrix Multiplication), a function similar to BLAS SGEMM which is already used by Caffe and many DNN frameworks and is responsible for the most workload within the layers of a network. This is the heart of the DNNs which involves dense float matrix multiplications. The accelerated function was implemented, tested for correctness and was optimized for the PL of the Xilinx ZC702 board but can be modified with ease for other FPGAs The result was a CPU-FPGA-based system, a highly heterogeneous all-programmable SoC that supports the Caffe framework and utilizes the hardware accelerator achieving significant speed and power efficiency compared to the ARM Zynq processor.

[1] Instruction fetch is a stage in a pipeline that loads the next instruction referred to by the program counter.

4.1 Selecting function for acceleration

The challenge before designing the hardware accelerator is to find the code within the current application that can be parallelized, and then how to parallelize that code so that it can be executed on an array of computational elements configured in the FPGA fabric. One good starting point is to first profile the code to find the computationally-intensive portions of the code and then find ways to isolate the code so that it does not have many data dependencies. Once this code is isolated, we must find ways to optimize the code so that it can be executed on the resources available. Caffe rather than providing a layer abstraction, provides a lower-level computational primitive, in order to simplify parallelization on high performance systems. After profiling and careful analysis of the source code of the framework, we found that the core algorithm responsible for most computations is the function GEMM. Next we describe the steps taken before we arrive at this result.

Caffe profiling For Caffe profiling we will use *caffe time* command which benchmarks model execution layer-by-layer through timing and synchronization. This is useful to check system performance and measure relative execution times for models. Given the BVLC Reference CaffeNet model for benchmarking we acquired the following results showing a large computation overhead over the convolution layers of the networks and a significant overhead to fully connected layers as well. Convolution is one of the most basic computations in a network and as seen in Figure 4.1, it is in Caffe framework as well. Though the basic code of the computation is not clear yet and we need to define a repetitive algorithm which can be accelerated though the programmable logic of the FPGA.

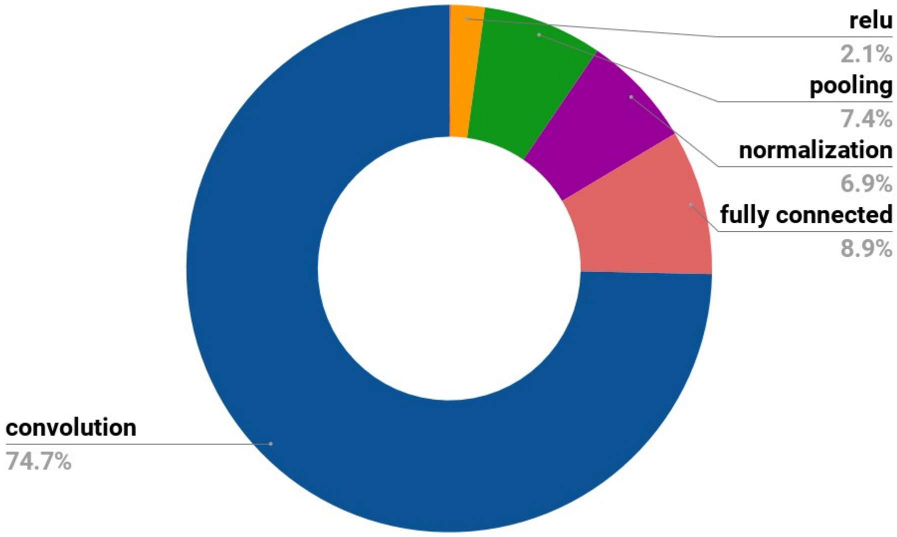


Figure 4.1: Execution time per layer category.

Caffe's convolution The implementation is not as obvious and while there are several ways to implement it efficiently (ex. Winograd, FFT etc.), Caffe tries to provide a method for a highly computer optimizable operation. The convolution layer treats its input as a two dimensional image, with a number of channels for each pixel, much like a classical image with width, height, and depth. The number of channels can be in hundreds inside the inner layers of a network rather than just RGB which may be at the first layer only. The convolution produces its output by taking a number of *kernels* of weights and applying them across the image [45].

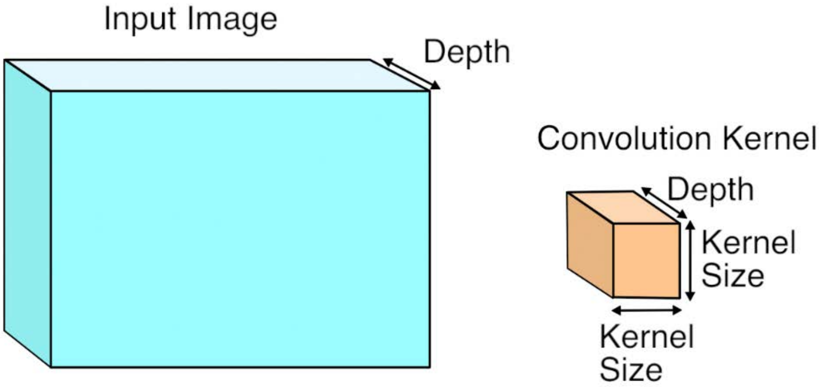


Figure 4.2: Illustration of an input image and a single kernel [45].

Each kernel is another three-dimensional array of numbers, with the depth the same as the input image, but with a much smaller width and height. To produce a result, a kernel is applied to a grid of points across the input image. At each point where it's applied, all of the corresponding input values and weights are multiplied together, and then summed to produce a single output value at that point. The kernel contains a pattern of weights and when the part of the input image it applies on has a similar pattern it outputs a high value. When the input doesn't match the pattern, the result is a low number in that position [45].

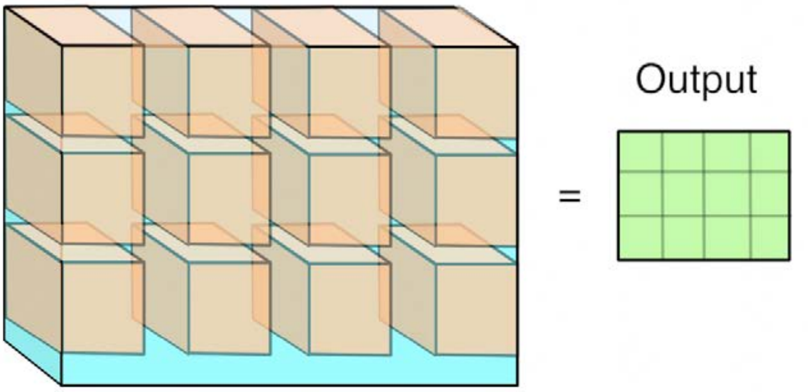


Figure 4.3: Output of a single kernel's application on the input image [45].

Convolution computation Caffe's strategy is to lower the convolutions into matrix multiplications. The first step is to turn the input from an image, which is effectively a 3D array, into a 2D array that we can treat like a matrix; this is known as *im2col*. This can be done by reshaping the filter tensor F into a matrix F_m with dimensions $K \times CRS$ and gathering a data matrix by duplicating the original input data into a matrix D_m with dimensions $CRS \times N$. The computation can then be performed with a matrix multiply to form an output matrix O_m with dimension $K \times N$.[2]

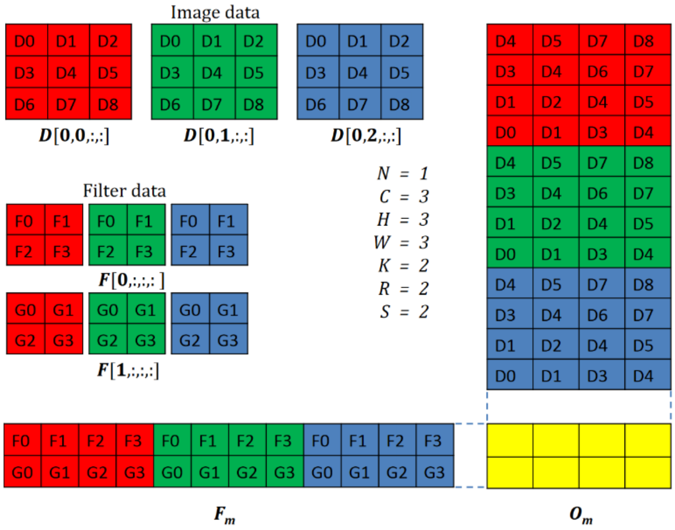


Figure 4.4: Convolution lowering to matrix multiplication [3].

Parameter	*Meaning*
N	Number of images in mini-batch
C	Number of input feature maps
H	Height of input image
W	Width of input image
K	Number of output feature maps
R	Height of filter kernel
S	Width of filter kernel

Table 4.1: Convolutional parameters

As seen above, pixels that are included in overlapping kernel sites will be duplicated in the matrix, which seems inefficient. Though this wastage is outweighed by the advantages though.

GEMM The matrix to matrix multiplication in Figure 4.4 is implemented in Caffe using a function called *GEMM* (General Matrix Multiply) which is basically the simple matrix equation $C = \alpha AB + \beta C$ where α, β scalars. That may include millions floating point operations and there can be dozens of layers in a modern architecture that have GEMM calls, so very often these networks need several billion FLOPs to calculate a single frame. Fortunately, it turns out that large matrix to matrix multiplications are highly optimizable and Caffe utilizes the commonly available, highly optimized BLAS (CUBLAS on NVidia GPUs) libraries for dense matrix computation. Also modern architectures such as FPGAs benefit from the very regular patterns of memory access outweighing the wasteful storage costs. Thus, in the next sections we will analyze the hardware accelerator that was designed to perform the GEMM function call and eventually outperform the BLAS GEMM call that Caffe uses by default which runs on CPU.

[2] Convolution may involve some other parameters such as stride and padding which is filling the input volume with zeros in such way that the convolution layer does not alter the spatial dimensions of the input.

4.2 Software function implementation

We considered many approaches to accelerate the convolution of Caffe like using the FFT (Fast Fourier Transform) based approach but it uses a significant amount of temporary memory (for example filters must be padded to be the same size as the inputs). So the software function that we implemented will behave similar to GEMM function which is the core of the convolution algorithm as shown previously, aiming to give the same result as the BLAS level-3 function *SGEMM* which will be replaced. GEMM has a straightforward implementation with inputs two matrices and scalars and output one matrix, computing the matrix equation $C = \alpha AB + \beta C$. In this section, we will outline the motivation and reasoning behind our design choices as for the software code of the function.

As stated in the previous sections, lowering convolutions to matrix multiplications can be efficient, since matrix multiplication is highly optimized. Matrix multiplication is fast because it has a high ratio of floating-point operations per byte of data transferred. This ratio increases as the matrices get larger, meaning that matrix multiplication is less efficient on small matrices. Accordingly, this approach to convolution is most effective when it creates large matrices for multiplication. The sizes of the matrices in the equation depend on products of the parameters to the convolution, not the parameters themselves. This means that performance using this approach can be very consistent, since the algorithm does not care if one of the parameters is small, as long as the product is large enough. For example, it is often true that in early layers of a convolutional network, input channels are few but the width and height of the filters are large, while at the end of the network the opposite happens. However, the product which results the input matrices is usually fairly large for all layers, so performance can be consistently good. The disadvantage of this approach is that forming these matrices involves duplicating the input data up many times, which can require a prohibitively large temporary allocation. To work around this, implementations sometimes materialize matrices piece by piece, for example, by calling matrix multiplication iteratively for each element of the mini-batch [3].

In this section, we describe the *wrapper* function and optimization techniques of GEMM at software level.[3] The basic approach is to use a block matrix multiplication where careful analysis was made to define the block sizes. The result of our custom function was tested on multiple inputs with irregular dimension matrices producing correct values.

[3] The wrapper function in our case is the routine function of GEMM whose main purpose is to call a second subroutine which is the hardware kernel that will be discussed in the next section.

4.2.1 Algorithm Design

As we discussed earlier, convolutions can be lowered into matrix multiplication. This approach provides simplicity of implementation as well as consistency of performance across the parameter space, although materializing the lowered matrix in memory can be costly. The optimizations in the CPU code can be several depending on the use case of the GEMM function.

Function declaration First the GEMM function besides the input matrices, dimensions and scalars, it must accept the *transpose* values (0 or 1) for each array of whether to transpose the matrices or not. This is needed depending on the major order type in which the arrays will be stored and read. In a row-major order, the consecutive elements of a row reside next to each other, whereas the same holds true for consecutive elements of a column in a column-major order. This is important for performance when traversing an array because modern systems process sequential data more efficiently than nonsequential data. Also, along with that the leading dimensions lda, ldb and ldc of arrays A, B and C are given as declared in the calling program. The definition of the function called *my_gemm* is shown below:

```
void my_gemm(int TA, int TB, int M, int N, int K, data_t alpha, data_t * A, int lda,
             data_t * B, int ldb, data_t beta, data_t * C, int ldc)
```

Algorithm complexity Ideally, performance should be limited by the arithmetic throughput of the processor. Indeed, for large square matrices where *M*=*N*=*K*, the number of math operations in classical algorithm of a product of matrices is $\mathrm{O}(\mathrm{N}^3)$ while the amount of data needed is $\mathrm{O}(\mathrm{N}^2)$, yielding a compute intensity on the order of *N*. However, taking advantage of the theoretical compute intensity requires reusing every element $O(N)$ times. The algorithm must hold a relatively small working set in fast on-chip caches, which results in good overall performance despite the fact that *M*, *N*, and *K* can grow.

Other approaches We considered using other algorithms with better complexities, such as Strassen's algorithm [46] which has a $O(n^{2.807})$ complexity, would yield better performance by replacing expensive multiplies with less expensive additions. While relatively limited on an FPGA, multipliers are quite cheap when compared to the routing costs of complicated multiplexing logic needed to implement these algorithms.[4] So the GEMM function with the classic matrix implementation was therefore chosen and we focused exclusively on accelerating simple $\mathrm{O}(\mathrm{N}^3)$ algorithms.

[4] A multiplexer (or mux) is a device that selects one of several analog or digital input signals and forwards the selected input into a single line.

Implementation The proposed algorithm is a variation on normal matrix multiplication where you divide the matrix into smaller sub-matrices and then calculate those matrices individually. The advantage of this method is that values needed for the matrix calculation are kept in the cache longer. Fixed sized submatrices of the input matrices A and B are successively read into on-chip memory and are then used to compute a submatrix of the output matrix C which is then stored in the appropriate location of C array. We compute on tiles of A and B while fetching the next tiles of A and B from DDR memory into on-chip caches and other memories. This technique hides the memory latency associated with the data transfer, allowing the matrix multiplication computation to be limited mainly by the time it takes to perform the arithmetic. Also padding with zeros is used for matrices whose dimensions are not multiplies of the block sizes so the block outer values are filled with zeros. The pseudocode of the algorithm is shown below:

Listing 1. *GEMM function pseudocode*

```
for (i = 0; i < Mtile; i+=Mtile) {
    for (j = 0; j < Ntile; j+=Ntile) {
        zero();
        for (k = 0; k < Ktile; k+=Ktile) {
            loadIntoFU(A, i, k);
            loadIntoFU(B, k, j);
            MulAccumulate();
        }
        store(C, i, j);
    }
}
```

- The *zero()* pseudo-function is used to clear the C result block before the computation.
- The loadIntoFU() pseudo-functions just copy each time the blocks needed for computation of array A and B from the external memory to fast CPU cache.
- The *MulAccumulate()* pseudo-function is the core computation of this algorithm as it is the multiplication and addition of pair of blocks that computes the result block each time. We discuss it in the next section how the implementation occured in hardware.
- The *store()* pseudo-function stores in the appropriate memory location the computed block of C to the output array C.

4.2.2 Software optimizations

First for the fast copying and clearing the blocks each time we used *memcpy()* and *memset()* functions respectively which are probably the fastest functions for handling these memory computations in software. Also, these need contiguous memory so we allocated the blocks using *sds_alloc()* instead of *malloc()* (this is also a requirement for ensuring fast communication with our hardware kernels which we discuss in the next section). Moreover, one slight optimization we made was to copy the B matrix before the corresponding A matrix. This ultimately hides part of the total load latency since the functional unit's access of A naturally matches the order in which it is loaded, allowing the computation to begin as soon as the first element of A returns from memory. Lastly, we attempted to use 2 threads for the parallelization of the copy functions *copy_tile_A* and *copy_tile_B* which copy the tiles each time. Though, the performance benefit was minimal because of the multiple instantiation and termination of the threads happening in the inner loop of our program.

The multiplication and addiction of the blocks are left for the *mmult_accel()* and *madd_accel()* functions which perform all the MAC operations and were implemented in the PL of the FPGA after validating them in software first. Our intention was to attempt to reduce the number of sub-matrix loads by defining large blocks while at the same time they could fit in the on-chip memory and be appropriate for the matrix operations used in the neural networks. The best block size happened to be 192 × 192 for the ZC702 FPGA SoC. The blocking technique is visualized in the following figure:

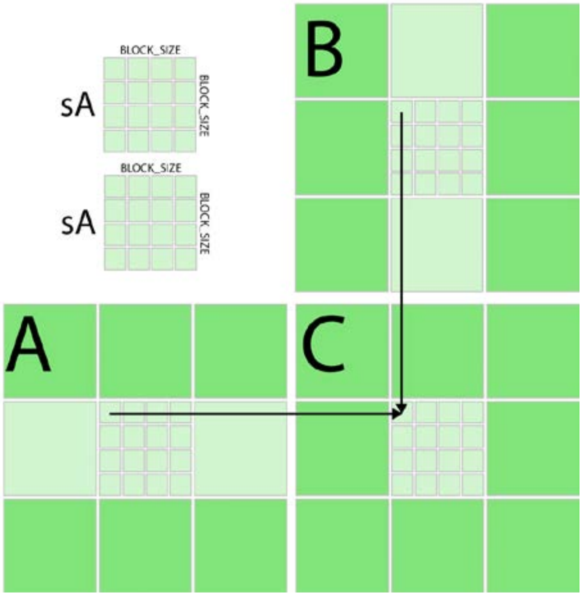


Figure 4.5: Visualization of blocked matrix multiplication [4].

With all these optimizations in mind, the full C++ software code of the GEMM wrapper function was constructed and tested on ZC702 SoC and it is as follows:

Listing 2. *GEMM wrapper software function*

```c
static data_t * A_tile, * B_tile, * C_tile, * R, * T;

void my_gemm(int TA, int TB, int M, int N, int K, data_t alpha, data_t * A, int lda, data_t * B, int ldb, data_t beta, data_t * C, int ldc) {
    int i, j, k, x, y, i_temp, j_temp;
    A_tile = (data_t * ) sds_alloc(MB * KB * sizeof(data_t));
    B_tile = (data_t * ) sds_alloc(KB * NB * sizeof(data_t));
    C_tile = (data_t * ) sds_alloc(MB * NB * sizeof(data_t));
    R = (data_t * ) sds_alloc(MB * NB * sizeof(data_t));
    T = (data_t * ) sds_alloc(MB * NB * sizeof(data_t));

    if (!A_tile || !B_tile || !C_tile || !T || !R) {
        fprintf(stderr, "Error buffer allocation\n")
        exit(1);
    }
    // initialize the blocks with zeros
    memset(A_tile, 0, sizeof(data_t) * MB * KB);
    memset(B_tile, 0, sizeof(data_t) * KB * NB);
    memset(C_tile, 0, sizeof(data_t) * MB * NB);
    memset(T, 0, sizeof(data_t) * MB * NB);

    for (i = 0; i < M; i += MB) {
        for (j = 0; j < N; j += NB) {
            // initializes C_tile for computation
            for (x = 0; x < MB; x++) {
                for (y = 0; y < NB; y++) {
                    i_temp = i + x;
                    j_temp = j + y;
                    if (i_temp < M && j_temp < N) C_tile[x * NB + y] = beta * C[i_temp * ldc + j_temp];
                    else C_tile[x * NB + y] = 0;  // padding with zeros
                }
            }

            // C_tile computation
            for (k = 0; k < K; k += KB) {
                copy_tile_B(TB, K, N, k, j, ldb, B);          // loadIntoFU for tile B
                copy_tile_A(TA, M, K, i, k, lda, A, alpha);   // loadIntoFU for tile A
                mmult_accel(A_tile, B_tile, T, alpha);        // tile multiplication
                madd_accel(T, C_tile, R);                     // tile addition
                memset(A_tile, 0, sizeof(data_t) * MB * KB);  // clear tile A
                memset(B_tile, 0, sizeof(data_t) * KB * NB);  // clear tile B
                memcpy(C_tile, R, sizeof(data_t) * MB * NB);  // store tile R
            }
            // store C_tile to output array C
            for (x = 0; x < MB; x++) {
                for (y = 0; y < NB; y++) {
                    i_temp = i + x;
                    j_temp = j + y;
                    if (i_temp < M && j_temp < N) C[i_temp * ldc + j_temp] = C_tile[x * NB + y];
                }
            }

        }
    }
    sds_free(A_tile);
    sds_free(B_tile);
    sds_free(C_tile);
    sds_free(R);
    sds_free(T);
    return;
}
```

Listing 3. *Software functions for copying tiles A and B*

```
// copy block A from DDR to cache
void copy_tile_A(int TA, int M, int K, int i, int k, int lda, data_t * A, data_t alpha
) {
    int i_temp, j_temp, x, y;
    if (TA == 0) {  // we don’t need transpose case for A
        for (i_temp = 0; i_temp < MB && i + i_temp < M; ++i_temp) {
            for (j_temp = 0; j_temp < KB && k + j_temp < K; ++j_temp) {
                x = i + i_temp;
                y = k + j_temp;
                A_tile[i_temp * KB + j_temp] = A[x * lda + y];
            }
        }
    }
}
// copy block B from DDR to cache
void copy_tile_B(int TB, int K, int N, int k, int j, int ldb, data_t * B) {

    int i_temp, j_temp, x, y;
    if (TB == 0) {  // We don’t need transpose case for B
        for (i_temp = 0; i_temp < KB && k + i_temp < K; ++i_temp) {
            for (j_temp = 0; j_temp < NB && j + j_temp < N; ++j_temp) {
                x = k + i_temp;
                y = j + j_temp;
                B_tile[i_temp * NB + j_temp] = B[x * ldb + y];
            }
        }
    }
}
```

4.3 Hardware function implementation

As described in the previous section it is necessary to split the problem into a number of overlapping tiles and later stitch the results back together. On FPGAs and in ASICs as well, matrix multiplications can be efficiently implemented with a systolic architecture. A suitable systolic array consists of a regular grid of simple, locally-connected processing units. Each of them performs one multiplication and one addition, before pushing the operands on to their neighbors. Thanks to the locality of computation, communication and memory, these architectures are very hardware-friendly [47]. Also, they are especially efficient for large problem sizes and batched computation which is important for batched image inference.

The two functions *mmult_accel* and *madd_accel* in the GEMM wrapper were placed so that they perform all the MAC operations of the matrix multiplication. The first is a simple matrix multiplication of each pair of blocks including the use of the alpha scalar and the second one is a block addition for the current result tile. Hence, they were synthesized in the programmable fabric of the FPGA in order to achieve acceleration compared to running in the ARM processor. The programs were written in high level C++ language that SDSoC analyzed, optimized and translated to RTL code. The FPGA architecture increases the computing speed by using the concept of parallel processing and pipelining into a single concept. We propose a systolic algorithm that relies on data from different directions arriving at cells in the array at regular intervals and being combined. By this pipelining processing it may proceed concurrently with input and output and consequently overall execution time is minimized [5]. The data must be stored on the fast BRAMs of the hardware which are near the computation and are fetched in a streaming manner to the PE (processing elements) of the systolic array. The theoretical 2- dimensional systolic architecture is visualized in the following figure.

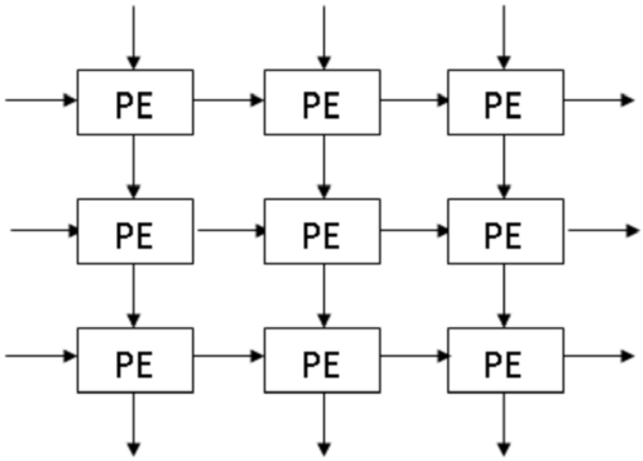


Figure 4.6: Systolic array architecture [47].

4.3.1 Communication interface

In the SDSoC environment where we design our hardware accelerator, the system generation process is controlled by structuring hardware functions and in order to balance communication and computation we inserted certain pragmas into the source code to guide the system compiler. The SDSoC compiler automatically chooses the best possible system port to use for any data transfer, but allows to override this selection by using pragmas. Specific pragmas select different data movers for the hardware function arguments and also control the number of data elements that are transferred to/from the hardware function. All pragmas related to the SDSoC environment are prefixed with *#pragma SDS* and were inserted into our C++ source code, either immediately prior to function declarations or at a function call site for optimization of a specific function call [48]. This project uses the AXI SIMPLE_DMA interface with ACP (Accelerator Coherence Port) system ports to connect to the main memory via the AXI4 bus in the Zynq ZC702. Initialization, status, and management registers are accessed through an AXI4-Lite slave interface.[5]

Data access `#pragma SDS data access_pattern(array:SEQUENTIAL)` pragma was used to specify the data access pattern in the hardware functions. SDSoC checks the value of this pragma to determine the hardware interface to synthesize. The access pattern was set as *SEQUENTIAL* and a streaming interface (*ap_fifo*) was generated. Otherwise, with *RANDOM* access pattern, a BRAM interface would be generated where the accelerator would access data from and could result in slower results. So after we guaranteeing in software level that the array arguments are located in physically contiguous memory (*using sds_alloc() instead of malloc()*), the most efficient data movers were used to fetch each matrix block in a fast streaming manner. Also these arrays had to be one-dimensional which is the general rule in simple DMAs and the access pattern would be A[0], A[1], A[2], ... , A[1023] with all elements accessed exactly once to minimize data input reads.

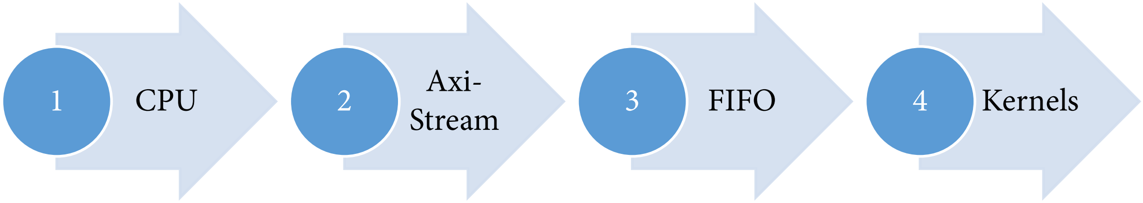


Figure 4.7: Sequential AXI Stream interface for hardware kernels.

[5] AXI is a DMA (Direct Memory Access) IP that provides high-bandwidth direct memory access between memory peripherals and

When the software running on the ARM A9 processor "calls" our hardware kernels, it actually calls an SDSoC environment generated stub function that calls underlying drivers to send data from the processor memories to the hardware function and to get data back from the hardware function to the processor memories over the system port ACP. This port allowed the processor and our hardware kernels to access the same fast cache memory which was the matrix blocks from GEMM wrapper, as shared memory [49].

Below is the header file of the GEMM wrapper function which includes all the necessary declarations of functions and communication interfaces. SDSoC data access pattern is guided through the pragmas that we inserted before the hardware function declarations. `#pragma SDS data access_pattern(A: SEQUENTIAL, B: SEQUENTIAL, C: SEQUENTIAL)` was used before *mmult_accel* and *madd_accel* hardware functions to enable the streaming FIFO interfaces for all the three arrays A, B and C. These arrays are the tile blocks of the matrix multiplication which are defined to a specific size because SDSoC needs to know the exact array length during compilation in order to synthesize the appropriate number of DSPs, LUTs etc. Also, the scalar *alpha* on *mmult_accel* requires very few hardware resource so it is transferred by the AXI_LITE data mover. The code is as follows:

Listing 4. *Header file of GEMM wrapper function*

```
#define MB 64
#define KB 64
#define NB 64

typedef float data_t;  //type of precision used

void my_gemm(int TA, int TB, int M, int N, int K, data_t alpha, data_t * A, int lda,
data_t * B, int ldb, data_t beta, data_t * C, int ldc);

#pragma SDS data access_pattern(A: SEQUENTIAL, B: SEQUENTIAL, C: SEQUENTIAL)
void mmult_accel(data_t A[MB * KB], data_t B[KB * NB], data_t C[MB * NB], data_t alpha
);

#pragma SDS data access_pattern(A: SEQUENTIAL, B: SEQUENTIAL, C: SEQUENTIAL)
void madd_accel(data_t A[MB * NB], data_t B[MB * NB], data_t C[MB * NB]);
```

4.3.2 Hardware optimizations

In order to achieve large throughput we had to enable a high degree of fine-grained parallelism in application execution within the PL (Programmable Logic) fabric.[6] Vivado HLS provides through SDSoC specific pragmas that can be used to optimize the design: reduce latency, improve throughput performance, and reduce area and device resource utilization of the resulting RTL code. So the next step to the optimization methodology in hardware was to place these pragmas directly to the source code for the kernels. Using SDSoC Area Estimation we always checked the details of how many resources are required in the PL to implement the hardware function and never exceeded 100% for any of the resources of the ZC702 FPGA board.

Function inlining Similar to function inlining of software functions, it was beneficial to inline the hardware functions. Function inlining replaces a function call by substituting a copy of the function body after resolving the actual and formal arguments. After that, the inlined function is dissolved and no longer appears as a separate level of hierarchy. We inserted the directive `#pragma HLS inline` at the beginning of the body of the hardware function in order to direct the Vivado HLS to inline the kernel. However, we did manual function inlining without using the directive which allowed operations within our inlined function to be optimized more effectively with surrounding operations.

Metrics optimization A common issue when our hardware functions are first compiled is sometimes report files showed the latency and interval as a question mark "?" rather than as numerical values. If the design has loops with variable loop bounds, the compiler cannot determine the latency and uses the "?" to indicate this condition. To resolve this condition many times we had to locate the lowest level loop which fails to report a numerical value and use the directive `#pragma HLS loop_tripcount` to apply an estimated tripcount. This allowed values for latency and interval to be reported and allows implementations with different optimizations to be compared [48].

Memory allocation The issue here is to use temporary arrays near the computation in hardware in order to reduce the memory communication as much as possible. So we copied the arrays in the hardware functions to temporary arrays which are implemented using the efficient block RAM resources in the PL fabric. In order to guide the Xilinx tool to use the BRAMs we used the directive `#pragma HLS RESOURCE variable=<array> core=RAM_2P_BRAM` for the arrays A and B. This resulted in a small cost-efficient fast design. The disadvantage of block RAM though is that, like other memories such as DDR or SRAM, they have a limited number of data ports, typically a maximum of two, and also they have limited size.

[6] In fine-grained parallelism, the program is broken down to a large number of small tasks. These tasks are assigned individually to many computing elements or processors [50].

Loop Parallelization Loop pipelining and loop unrolling improve the hardware function's performance by exploiting the parallelism between loop iterations. In sequential languages such as C/C++, the operations in a loop are executed sequentially and the next iteration of the loop can only begin when the last operation in the current loop iteration is complete. Loop pipelining allows the operations in a loop to be implemented in a concurrent manner as shown in the following figure.

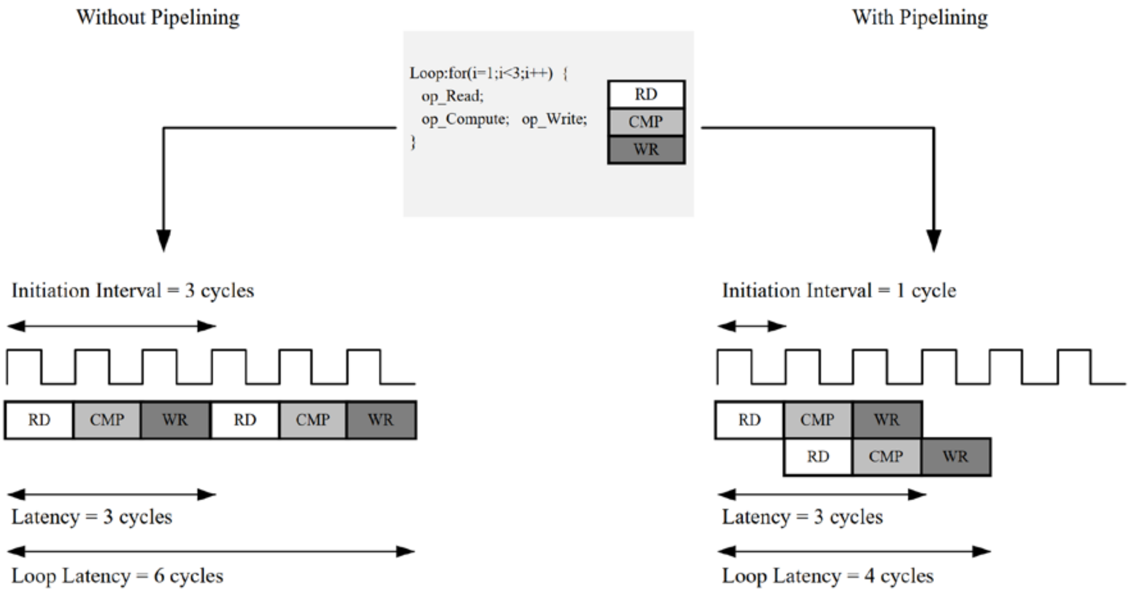


Figure 4.8: The effect of loop pipelining in latency [48].

An important term for loop pipelining is called Initiation Interval (II), which is the number of clock cycles between the start times of consecutive loop iterations. In the above figure, the Initiation Interval (II) is one because there is only one clock cycle between the start times of Consecutive loop iterations. To pipeline our loops, we put `#pragma HLS pipeline II=1` at the beginning of every loop body to ensure Vivado HLS will try to pipeline the loop with minimum Initiation Interval. We avoided data dependencies, as matrix multiplication for example is by default highly parallelizable and thus constructed a highly parallel and pipelined architecture with minimum latency that performed the MAC operations very efficiently. In our 3-level loop of matrix multiplication kernel we could only pipeline the two outer loops which were data independent so we placed the directive after second loop.

Furthermore, in the inner loop we tried to apply an adder tree of the related additions which occur for each element in order to calculate each value of the output array. We used `#pragma HLS unroll factor=<K>` directive to unroll the additions on the rows of array A and columns of array B while playing with the unroll factor for optimal solution. Though, the results showed increased resource allocation and the increase in concurrency was limited (about 4%).

Function pipelining The next stage in creating a high-performance design is to pipeline the functions. These technique optimize the parallelism between the functions where data flow pipelining exploits the "coarse grain" parallelism at the level of functions. SDSoC chained together our hardware functions because the data flow between them does not require transferring arguments out of programmable logic and back to system memory so the first function did not have to complete before starting the next function. The following example figure compares the latency with function data flow pipelining which the results are fetched into the other functions as soon as they are ready.

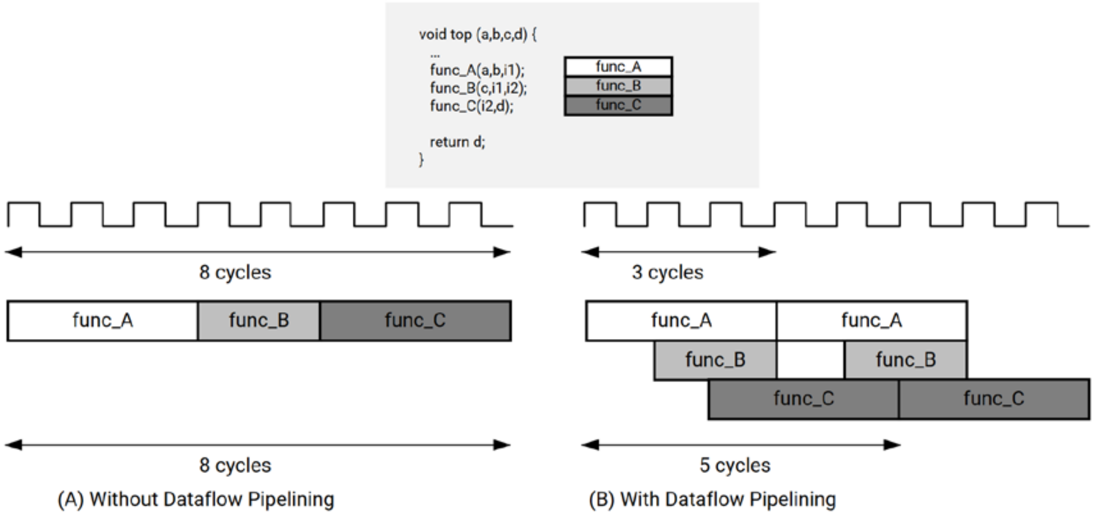


Figure 4.9: The effect of function pipelining in latency [48].

We implemented this technique along with placing the two hardware functions one after another. The output of the matrix multiplication function is the input to the matrix addiction function. So, Vivado HLS automatically inserted channels between the functions either as mutli-buffers or FIFOs and the data was directly fetched from the one function to the other without having to be retrieved back to the processor.

Memory Bandwidth With the previous directives, Vivado HLS creates logic cells for the memory allocated at the BRAM where we copied our arrays. This bottlenecks our acceleration because data access is limited to only a few BRAMs slices so DSPs are created only for the specific memory ports resulting in limited number of DSPs cells. This issue can be solved by inserting `#pragma HLS array_partition variable=<array> block factor=<N> dim=<dim>` directive after the declaration of the temporary arrays. With this pragma, we partitioned the temporary arrays into smaller equal blocks on individual registers allocating more BRAMs enabling more access ports and so Vivado created more RTL resulting in improved performance. The *factor* value for the number of the blocks was set as half the dimension of each array enabling the dual port read of BRAMs and *dim* value was set to “2” for the array A and “1” for the array B because we traverse them in rows and columns respectively.

Resource allocation of operations In the hardware kernels we sometimes needed to insert instance restrictions to limit or specify the type of resource allocation on specific variables in the implemented kernel. Vivado HLS implements the operations in the code using hardware cores. As we specified the BRAM type resource for the allocation of the arrays as mentioned previously, we also had to guide Vivado HLS to implement some arithmetic operations using specific cores. For example we used the directive `#pragma HLS RESOURCE variable=<name> core=FMul_fulldsp` on our multiplier variable to ensure only DSPs will be allocated for the multiplication operation. Also we used the directive `#pragma HLS RESOURCE variable=<name> core=FAddSub_fulldsp` on our accumulator variable to ensure the same for the addition operation. Thus, DSPs were mapped during the synthesis in order to achieve a highly efficient and low latency architecture.

Other design techniques We generally avoided to use close to 100% resource utilization because that could result in lower kernel clock speeds because the synchronization of the PL would be harder due to complex design. Especially when using LUTs, placement and routing can become very hard[7], specifically if we needed to meet aggressive timing constraints. So we preferred using DSPs and BRAMs as the latency was limited compared with the LUTs. Furthermore, we attempted to load as larger matrices as we could on the BRAM memory of the PL while maintaining a resource utilization below 100%. This resulted in greater performance in GFLOPs because the communication overhead was covered by the large computation of MAC operations. Of course, the ZC702 board could not fit enough logic to fully parallelize the matrices but the performance improvement remained substancial even for the array partitions we implemented. The two source files of the hardware kernels follow on the next page.

[7] In a smaller FPGA, this percentage may be larger because possibly there is not that large routing distance in smaller fabrics.

Listing 5. *Hardware kernel of block multiplication*

```
void mmult_accel(data_t A[MB * KB], data_t B[KB * NB], data_t C[MB * NB], data_t alpha ) {

    data_t tA[MB][KB], tB[KB][NB], tC[MB][NB];

    #pragma HLS array_partition variable = tA block factor = 16 dim = 2
    #pragma HLS array_partition variable = tB block factor = 16 dim = 1
    #pragma HLS RESOURCE variable = tA core = RAM_2P_BRAM
    #pragma HLS RESOURCE variable = tB core = RAM_2P_BRAM

    for (int i = 0; i < MB; i++) {
        for (int j = 0; j < KB; j++) {
            #pragma HLS PIPELINE
            tA[i][j] = alpha * A[i * KB + j];
            tB[i][j] = B[i * NB + j];
        }
    }

    for (int i = 0; i < MB; i++) {
        for (int j = 0; j < NB; j++) {
            #pragma HLS PIPELINE
            data_t result = 0;
            for (int k = 0; k < KB; k++) {
                data_t term = tA[i][k] * tB[k][j];
                result += term;
            }
        tC[i][j] = result;
    }

    for (int i = 0; i < MB; i++) {
        for (int j = 0; j < NB; j++) {
            #pragma HLS PIPELINE
            C[i * NB + j] = tC[i][j];
        }
    }
}
```

Listing 6. *Hardware kernel of block addition*

```
void madd_accel(data_t A[MB * NB], data_t B[MB * NB], data_t C[MB * NB]) {
    int i, j;
    for (i = 0; i < MB; i++) {
        for (j = 0; j < NB; j++) {
        #pragma HLS PIPELINE
        C[i * NB + j] = A[i * NB + j] + B[i * NB + j];
        }
    }
}
```

4.4 Integration with Caffe

After porting Caffe into Zynq SoC on ARM and designing the FPGA accelerator (GEMM) that would be used to accelerate the image classification algorithm, we then had to integrate the accelerator function into Caffe's framework. This procedure, involves many steps before running Caffe on board and actually using the FPGA kernel to speed-up image classification. In order to integrate the hardware accelerator with the Caffe framework and port it into the Zynq SoC we had to create it as a shared library instead of an application binary and then link it with the rest of the Caffe framework. Through the SDSoC Development Environment we were able not only to create the dynamic library but also the SD card and boot image that our board would boot from. The shared library was suitable for linking and our accelerator function was integrated into Caffe framework with success, communicating with the FPGA kernels when needed. The process comprises some steps which are described on the next sections.

4.4.1 Creating the booting system

Boot Image On this step we describe how we created the boot image that our board would boot from. The creation of the boot image was done by SDK tools from SDSoC Environment. In order to produce the specific file (*boot.bin*), SDSoC created the necessary files automatically; FSBL (First Stage Boot Loader), U-Boot, uImage, Rootfs and Device Tree Blob. Though what components are part of the boot image and what not, cannot be answered in a generic way. It heavily depends on the use-case and requirements. At a bare minimum, it must contain an FSBL without being enough. Hence a common boot image consists of an FSBL and U-Boot. In cases the FSBL also takes over programming the PL, a bitstream would be added as well. In our case, the bitstream from out hardware accelerator was taken into account, with a long process of place and route where the output bitstream was added to the boot image. Also, higher level OS components can be processed by U-Boot from various sources. U-Boot can load those images from flash, via Ethernet or assume they have been pre-loaded by other means (e.g. JTAG or the FSBL). We used the JTAG method.

Prepare Boot Medium The next step of the process is the preparation of a medium as boot device. This of course assumes a Linux system. Xilinx Software Development Kit can automatically create an SD card with preinstalled Linux system, called PetaLinux, which is appropriate for the embedded system that we created that needs a Linux image. We also configured SDSoC to create a shared library instead of an application binary in order to link with Caffe build. After, the creation of the SD card along with the boot image *boot.bin* and the dynamic/shared library (.so file), we followed with the building of Caffe framework and linking it with our library.

4.4.2 Linking the hardware function with Caffe

After the creation of the boot files and the shared library of our accelerator, we then had to link the library with Caffe so that Caffe could communicate with our function and the function could communicate with the PL. The shared library was suitable for linking with the ARM toolchain through the modified Makefile of the Caffe framework. We just added the compiler flag *–lgemm* in order to link our shared library *libgemm.so* with the rest of the Caffe framework and also added the appropriate header file for our accelerator function which is listed below.

Listing 7. *Header file of GEMM wrapper function needed for Caffe*

```
typedef float data_t;

void my_gemm(int TA, int TB, int M, int N, int K, data_t alpha,
     data_t * A, int lda,
     data_t * B, int ldb, data_t beta,
     data_t * C, int ldc);
```

Then all we had to do was replace the GEMM function call that runs on CPU which is called by BLAS with our GEMM wrapper function that is accelerated though the PL of the FPGA and run the Makefile with the new linked shared library. Specifically we modified *caffe_cpu_gemm* function which is found on *math_functions.cpp* file of Caffe source files. We replaced *cblas_sgemm* function with our custom function *my_gemm* and did some other modification which are shown below. Also, we took into account the matrix dimensions that our accelerator performed well and excluded the other dimensions (i.e. very small or linear dimensions).

Listing 8. *GEMM replacement in Caffe source code*

```
void caffe_cpu_gemm<float>(const CBLAS_TRANSPOSE TransA, const CBLAS_TRANSPOSE TransB,
                           const int M, const int N, const int K,
                           const float alpha, const float * A,
                           const float * B, const float beta, float * C) {

    int lda = (TransA == CblasNoTrans) ? K : M;
    int ldb = (TransB == CblasNoTrans) ? N : K;
    int TA = (TransA == CblasNoTrans) ? 0 : 1;
    int TB = (TransB == CblasNoTrans) ? 0 : 1;

    if (M >= 32 && N >= 32 && K >= 32 && TA == 0 && TB == 0)
       my_gemm(TA, TB, M, N, K, alpha, A, lda, B, ldb, beta, C, N);  //accelerated
    else
       cblas_sgemm(CblasRowMajor, TransA, TransB, M, N, K, alpha,    //default
                   A, lda, B, ldb, beta, C, N);
}
```

4.5 Setting up and running the final system

In order to run the whole accelerated system on board we had to set up the FPGA board and the PC which would be connected through UART. First, in order to boot our generated SD card image, the boot pins of the board had to be configured accordingly, as shown in the image below.

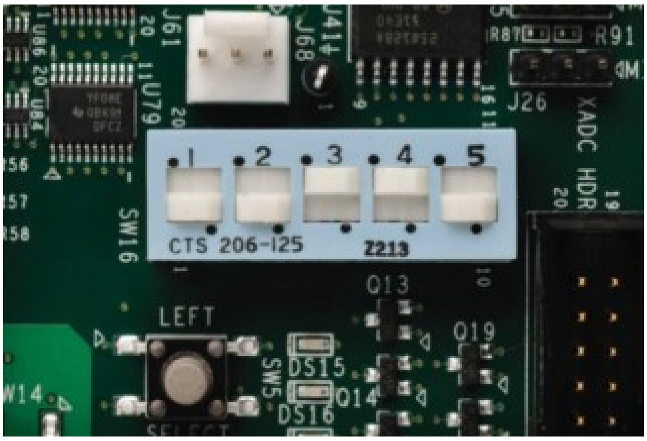


Figure 4.10: Switch setup for SD card boot mode on ZC702

Additionally on the Xilinx Development Board we used the USB UART port connected with a mini-B USB cable to connect the USB UART port on the board to our PC. The correct JTAG mode had to be selected, according to the used interface. The JTAG mode is controlled by switch SW4 on the zc706 (or SW10 on ZC702). The settings are listed in the following table.

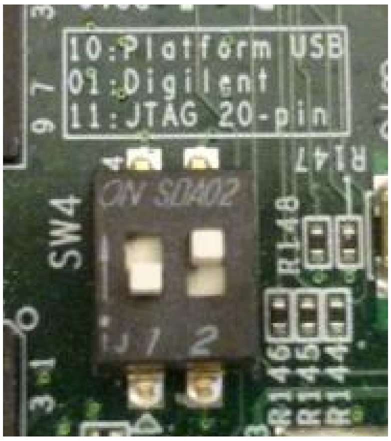
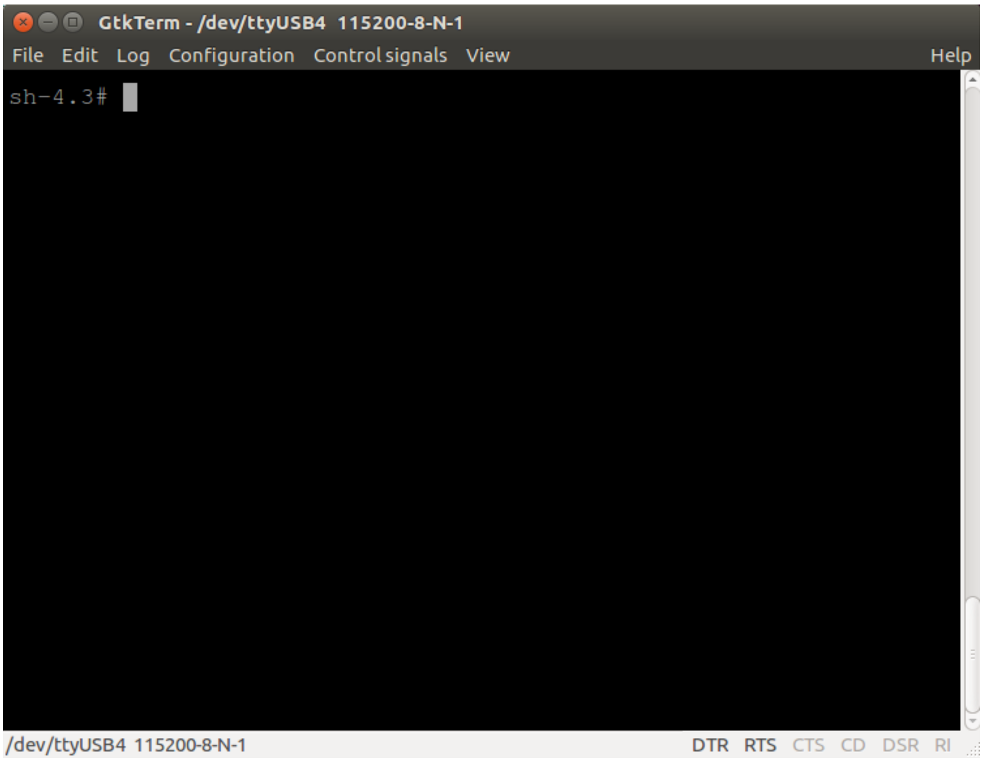


Figure 4.11: Switch selection for Digilent USB JTAG connection

Also, we had to install on the PC a terminal emulator which allows the access to FPGA terminal and all its applications running via the USB cable. We used *TeraTerm* [51] which is a terminal tool for connecting with remote or local hosts.
The settings for the serial connection may differ from board to board, but the following settings surely work for Zynq platforms:

baud rate = 115200
data bits = 8
stop bits = 1
flow control = none
parity = none

The serial device depended on our operating system and cable connection. On our Linux PC we found the serial devices in the /dev directory and precisely was *ttyUSB4* serial device port. After configuring the following settings on the TeraTerm terminal we successfully made a connection and booted the board accessing all files in the SD card.


GtkTerm - /dev/ttyUSB4 115200-8-N-1
File Edit Log Configuration Control signals View Help
/dev/ttyUSB4 115200-8-N-1
DTR RTS CTS CD DSR RI

```
sh-4.3#
```

Figure 4.12: Communication with the FPGA board via TeraTerm terminal

Operation of the final system The final point of this thesis work was to link the designed hardware accelerator with the rest of the Caffe framework and run it on board. As shown in Figure 4.13, the whole process starts from the Caffe framework where the user runs the classification command on the host CPU of Zynq SoC. Then, whenever Caffe needs to make a GEMM call, it does not load the previous BLAS GEMM call which would run on CPU but instead it loads our new GEMM function (*my_gemm*). Next, whenever our function needs to speed up MAC operations, it communicates through AXI stream with our kernels (*mmult_accel, madd_accel*) passing each time the blocked matrices on the PL and sends back the result blocks. So, we efficiently send back and forth information any time needed within the PS and PL in a streaming manner. At the end, Caffe retrieves all the information from our GEMM function which returns the entire calculated array (array C).

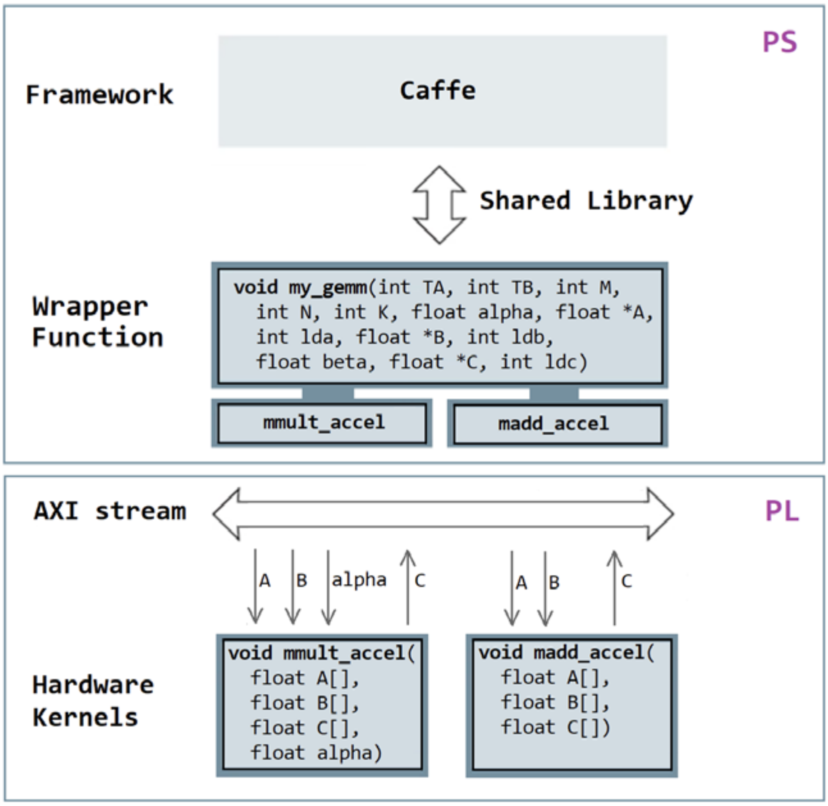

PS
Framework
Caffe
Shared Library
Wrapper Function
void my_gemm(int TA, int TB, int M, int N, int K, float alpha, float *A, int lda, float *B, int ldb, float beta, float *C, int ldc)
mmult_accel
madd_accel
AXI stream
PL
A
B
alpha
C
A
B
C
Hardware Kernels
void mmult_accel(float A[], float B[], float C[], float alpha)
void madd_accel(float A[], float B[], float C[])

Figure 4.13: Visualization of the accelerator communicating with Caffe framework

5 Evaluation and Results

The last three chapters gave a detailed information of porting Caffe into embedded SoCs, the design of the GEMM hardware accelerator and the heterogeneous CPU-FPGA accelerated system which supports Caffe framework. All of these components have been completed successfully, and together constitute the fully operable system which can make image classification run more efficiently on FPGA SoC. This chapter is concerned with an in-depth evaluation of this system regarding different aspects. First, we take some metrics and make an analysis of some neural network models with Caffe running solely on ARM CPU of the embedded SoC (section 5.1). Then, we make a hardware analysis and assess the performance of the designed hardware accelerator (section 5.2). The final section brings both components together and investigates the overall system performance of the Caffe framework running on CPU-FPGA SoC while proposing several other potential improvements (section 5.3).

5.1 Caffe on embedded CPU performance

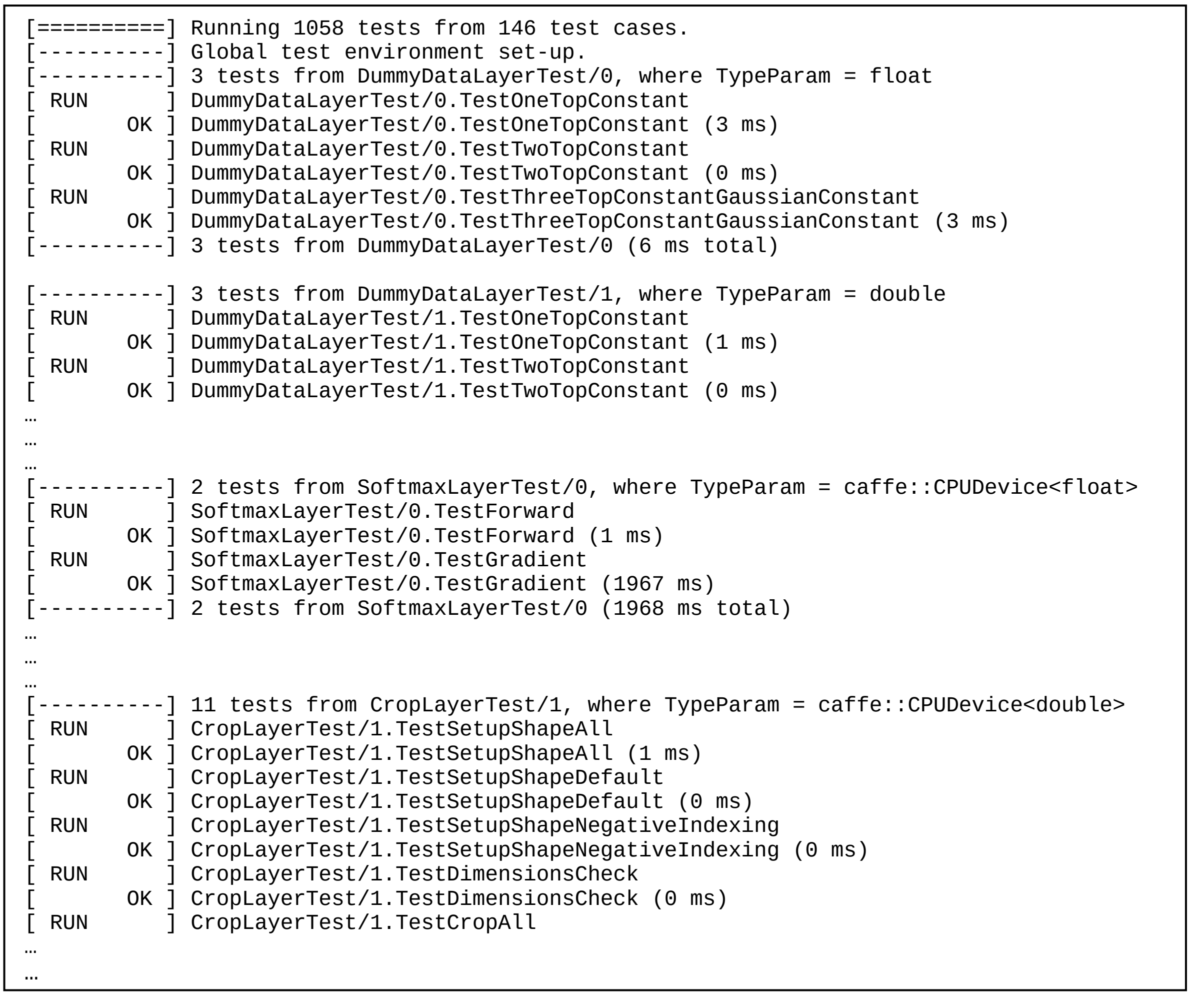
In chapter 3, a detailed method has been described of how to port Caffe into ARM CPU of Zynq SoC. Therefore, in this section we confine ourselves to a summary of the most important characteristics of the operation of the framework in ARM CPU. To start with, section 5.1.1 gives several metrics on Caffe framework regarding its operation (accuracy, execution times, etc.). Then in section 5.1.2 we analyze some characteristics of several neural network models and present what lies behind the operation of the framework and the different layers.[1]

[1] For this task we used the python API of Caffe framework because it can visualize networks while all model data, derivatives, and parameters are exposed for reading and writing.

5.1.1 Caffe metrics on ARM

After successfully porting Caffe to run on ARM CPU of Zynq SoC we did several tests utilizing the internal tools that Caffe provides, making the necessary tweaks when applicable in order to give a comparison between different scenarios (i.e. different network models etc.)

Framework functionality The first step after porting Caffe into ARM was to test the framework via the test binaries that are produced when running "*make test*" in Caffe directory. These binaries assess each layer with different inputs and checks if the results are correct which is essential for the proper operation of the whole framework. The results were correct and all the tests passed except those that had to do with the gpu device which is irrelevant and not used in the FPGA SoC. Some of the tests are presented below:



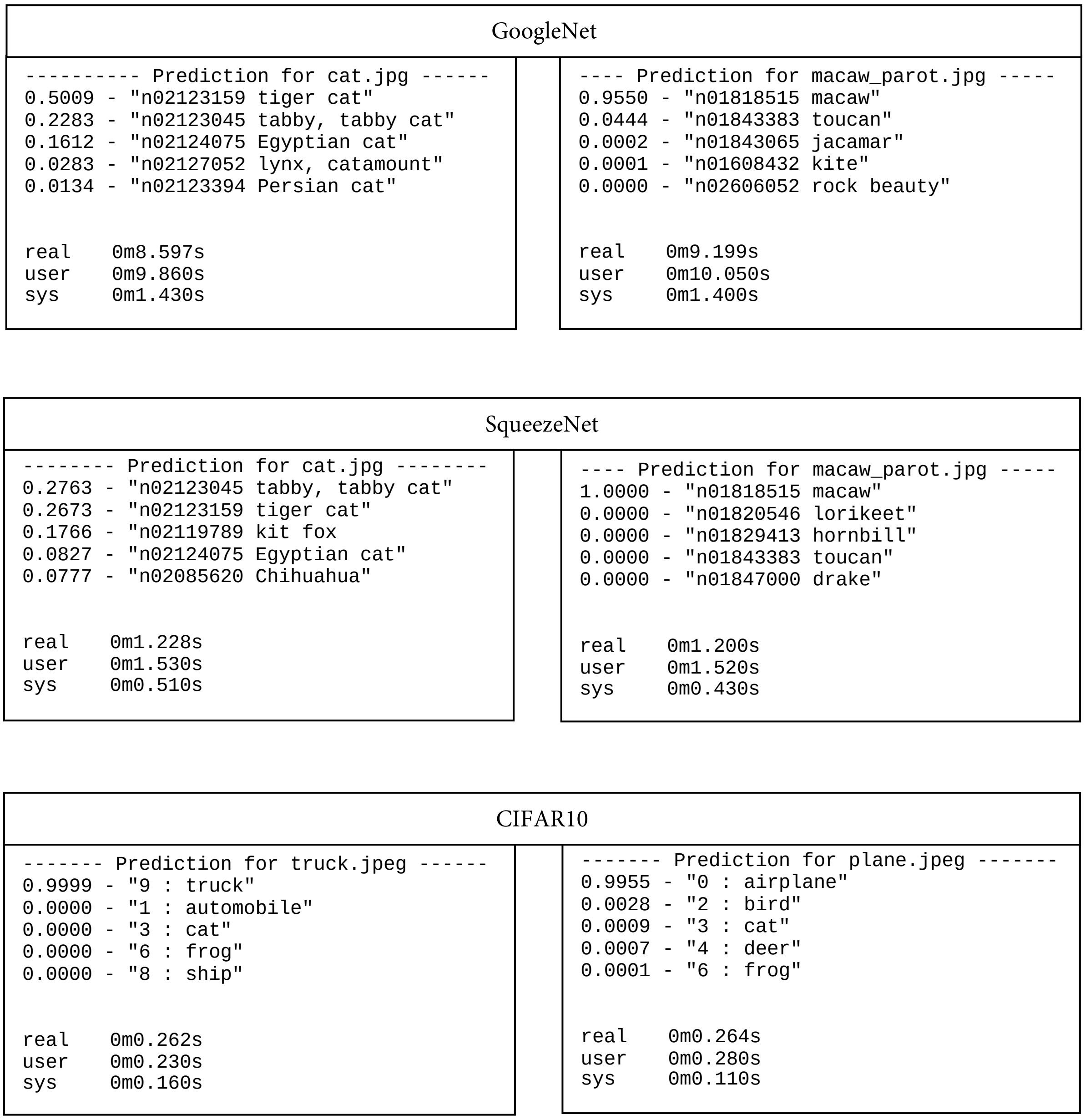
```
[==========] Running 1058 tests from 146 test cases.
[----------] Global test environment set-up.
[----------] 3 tests from DummyDataLayerTest/0, where TypeParam = float
[ RUN      ] DummyDataLayerTest/0.TestOneTopConstant
[       OK ] DummyDataLayerTest/0.TestOneTopConstant (3 ms)
[ RUN      ] DummyDataLayerTest/0.TestTwoTopConstant
[       OK ] DummyDataLayerTest/0.TestTwoTopConstant (0 ms)
[ RUN      ] DummyDataLayerTest/0.TestThreeTopConstantGaussianConstant
[       OK ] DummyDataLayerTest/0.TestThreeTopConstantGaussianConstant (3 ms)
[----------] 3 tests from DummyDataLayerTest/0 (6 ms total)

[----------] 3 tests from DummyDataLayerTest/1, where TypeParam = double
[ RUN      ] DummyDataLayerTest/1.TestOneTopConstant
[       OK ] DummyDataLayerTest/1.TestOneTopConstant (1 ms)
[ RUN      ] DummyDataLayerTest/1.TestTwoTopConstant
[       OK ] DummyDataLayerTest/1.TestTwoTopConstant (0 ms)
…
…
…
[----------] 2 tests from SoftmaxLayerTest/0, where TypeParam = caffe::CPUDevice<float>
[ RUN      ] SoftmaxLayerTest/0.TestForward
[       OK ] SoftmaxLayerTest/0.TestForward (1 ms)
[ RUN      ] SoftmaxLayerTest/0.TestGradient
[       OK ] SoftmaxLayerTest/0.TestGradient (1967 ms)
[----------] 2 tests from SoftmaxLayerTest/0 (1968 ms total)
…
…
…
[----------] 11 tests from CropLayerTest/1, where TypeParam = caffe::CPUDevice<double>
[ RUN      ] CropLayerTest/1.TestSetupShapeAll
[       OK ] CropLayerTest/1.TestSetupShapeAll (1 ms)
[ RUN      ] CropLayerTest/1.TestSetupShapeDefault
[       OK ] CropLayerTest/1.TestSetupShapeDefault (0 ms)
[ RUN      ] CropLayerTest/1.TestSetupShapeNegativeIndexing
[       OK ] CropLayerTest/1.TestSetupShapeNegativeIndexing (0 ms)
[ RUN      ] CropLayerTest/1.TestDimensionsCheck
[       OK ] CropLayerTest/1.TestDimensionsCheck (0 ms)
[ RUN      ] CropLayerTest/1.TestCropAll
…
…
```

Figure 5.1: Partial output from Caffe framework testing on ARM for proper functionality.

Image Classification Caffe provides an example C++ binary (*classification.bin*) for image classification. In order to classify several images, we successfully imported some pretrained models (*caffemodels*) on Caffe such BVLC GoogleNet and Squeezenet which are based on ImageNet dataset and can classify 1000 categories. Moreover, we also used a small pretrained model with 10 categories which is based on CIFAR10 dataset. The command we executed each time had the following format :

```
./classification.bin  deploy.prototxt  network.caffemodel  mean.binaryproto
labels.txt image.jpg
```

Some examples are presented below showing the execution times on different models:



GoogleNet

```
--------- Prediction for cat.jpg ------
0.5009 - "n02123159 tiger cat"
0.2283 - "n02123045 tabby, tabby cat"
0.1612 - "n02124075 Egyptian cat"
0.0283 - "n02127052 lynx, catamount"
0.0134 - "n02123394 Persian cat"


real    0m8.597s
user    0m9.860s
sys     0m1.430s
```

```
---- Prediction for macaw_parot.jpg -----
0.9550 - "n01818515 macaw"
0.0444 - "n01843383 toucan"
0.0002 - "n01843065 jacamar"
0.0001 - "n01608432 kite"
0.0000 - "n02606052 rock beauty"


real    0m9.199s
user    0m10.050s
sys     0m1.400s
```

SqueezeNet

```
-------- Prediction for cat.jpg --------
0.2763 - "n02123045 tabby, tabby cat"
0.2673 - "n02123159 tiger cat"
0.1766 - "n02119789 kit fox
0.0827 - "n02124075 Egyptian cat"
0.0777 - "n02085620 Chihuahua"


real    0m1.228s
user    0m1.530s
sys     0m0.510s
```

```
---- Prediction for macaw_parot.jpg -----
1.0000 - "n01818515 macaw"
0.0000 - "n01820546 lorikeet"
0.0000 - "n01829413 hornbill"
0.0000 - "n01843383 toucan"
0.0000 - "n01847000 drake"


real    0m1.200s
user    0m1.520s
sys     0m0.430s
```

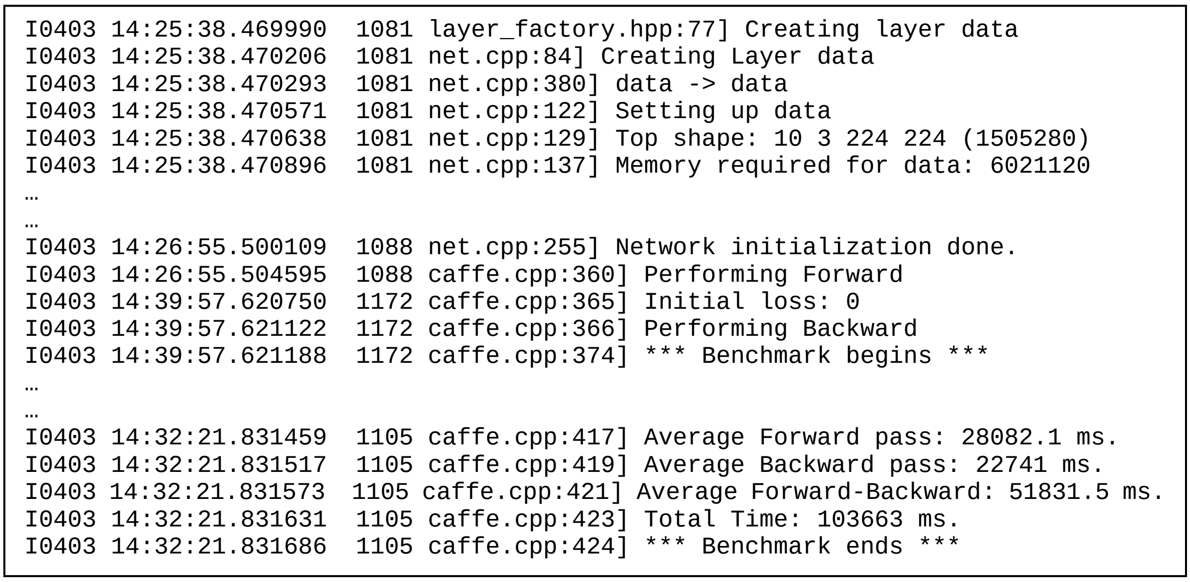
CIFAR10

```
------- Prediction for truck.jpeg ------
0.9999 - "9 : truck"
0.0000 - "1 : automobile"
0.0000 - "3 : cat"
0.0000 - "6 : frog"
0.0000 - "8 : ship"


real    0m0.262s
user    0m0.230s
sys     0m0.160s
```

```
------- Prediction for plane.jpeg -------
0.9955 - "0 : airplane"
0.0028 - "2 : bird"
0.0009 - "3 : cat"
0.0007 - "4 : deer"
0.0001 - "6 : frog"


real    0m0.264s
user    0m0.280s
sys     0m0.110s
```

Figure 5.2: Image Classifications and execution times on different models on ARM.

Inference-Learning Next, we wanted to evaluate only the inference time for the prediction of an image and ignore the layer initializations. Caffe provides a *time* tool which benchmarks model execution layer-by-layer through timing and synchronization. This is useful to check system performance and measure relative execution times for models. In the next figures, a partial example output for GoogleNet benchmark is shown in the first figure and the clean inference comparison between different models is shown in the second figure. All models were tested in batching of 10 except CIFAR10 which is on a different dataset and used smaller images with no batching.



```
I0403 14:25:38.469990  1081 layer_factory.hpp:77] Creating layer data
I0403 14:25:38.470206  1081 net.cpp:84] Creating Layer data
I0403 14:25:38.470293  1081 net.cpp:380] data -> data
I0403 14:25:38.470571  1081 net.cpp:122] Setting up data
I0403 14:25:38.470638  1081 net.cpp:129] Top shape: 10 3 224 224 (1505280)
I0403 14:25:38.470896  1081 net.cpp:137] Memory required for data: 6021120
…
…
I0403 14:26:55.500109  1088 net.cpp:255] Network initialization done.
I0403 14:26:55.504595  1088 caffe.cpp:360] Performing Forward
I0403 14:39:57.620750  1172 caffe.cpp:365] Initial loss: 0
I0403 14:39:57.621122  1172 caffe.cpp:366] Performing Backward
I0403 14:39:57.621188  1172 caffe.cpp:374] *** Benchmark begins ***
…
…
I0403 14:32:21.831459  1105 caffe.cpp:417] Average Forward pass: 28082.1 ms.
I0403 14:32:21.831517  1105 caffe.cpp:419] Average Backward pass: 22741 ms.
I0403 14:32:21.831573 1105 caffe.cpp:421] Average Forward-Backward: 51831.5 ms.
I0403 14:32:21.831631  1105 caffe.cpp:423] Total Time: 103663 ms.
I0403 14:32:21.831686  1105 caffe.cpp:424] *** Benchmark ends ***
```

Figure 5.3: Example partial output for BVLC GoogleNet benchmark. The top shape dimensions “10 3 224 224” means it has a batch of 10 images, with 3 channels (RGB) and 224×224 image dimensions. The inference per image is the “Average Forward pass” divided by the number of batches, so $\frac{28082.1\ ms}{10} = 2{,}8s$.

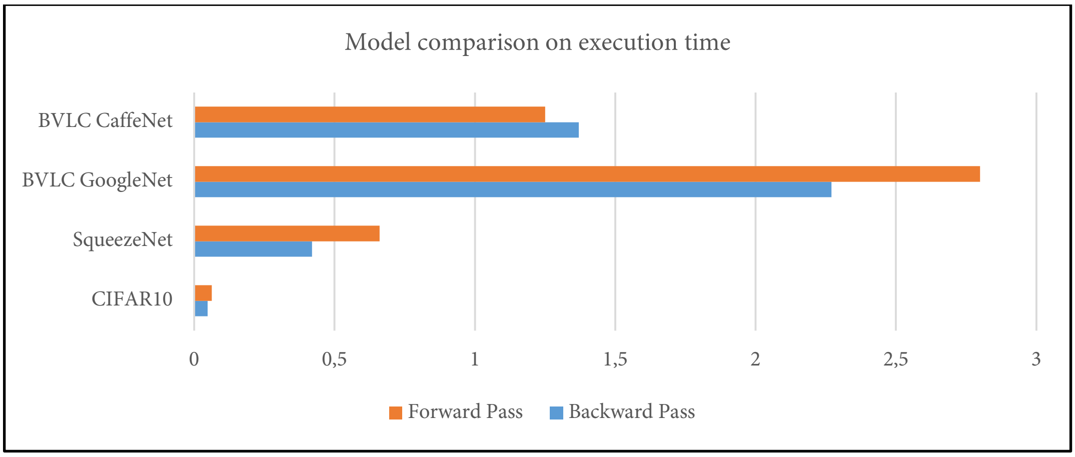


Figure 5.4: Comparison in execution time (seconds) between different models for inference (forward pass) and learning (backward pass) network computations on ARM.

Accuracy on CPU Caffe runs in our embedded system which has limited memory. Due to large memory size of ImageNet dataset we could not use it for our validation data in order to evaluate the accuracy. So we used the much smaller CIFAR10 dataset (50000 32x32 images), creating the lmdb files for training and testing first and then computing the image mean binaryproto of the dataset. For measuring accuracy, Caffe provides a *test* tool that scores models by running them in the test phase and reports the net output as its score. The net architecture must be properly defined to output an accuracy measure or loss as its output. The per-batch score is reported and then the grand average is reported last. Figure 5.4 shows the partial output of creating the lmdb, image mean and then running the *test* Caffe tool to measure the accuracy and loss.

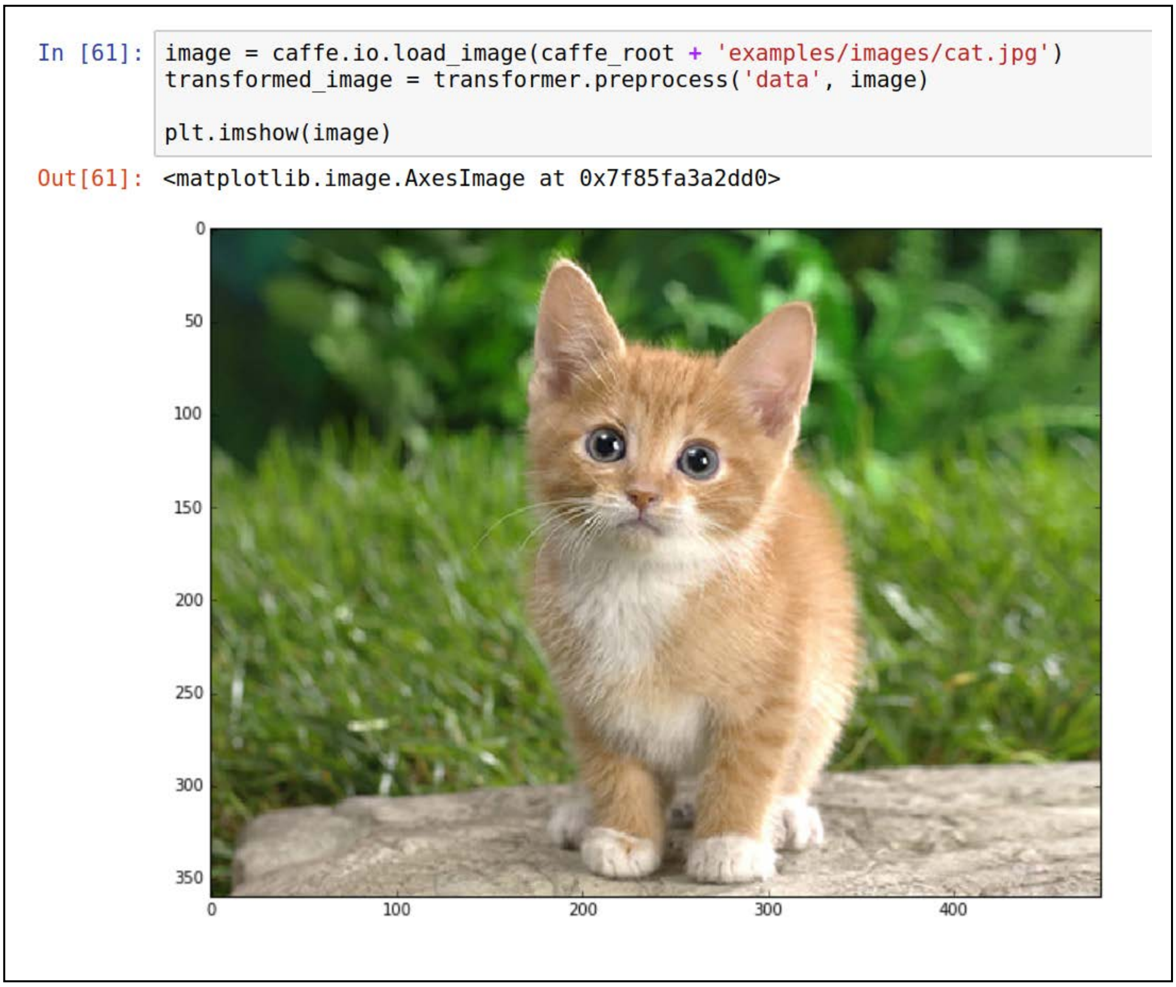
```
Creating lmdb...
I0306   13:48:48.762825      1019   db_lmdb.cpp:35]   Opened   lmdb   /mnt/caffe-
cpu/examples/cifar10/cifar10_train_lmdb
I0306 13:48:48.764823  1019 convert_cifar_data.cpp:52] Writing Training data
I0306 13:48:48.765002  1019 convert_cifar_data.cpp:55] Training Batch 1
I0306 13:48:50.477787  1019 convert_cifar_data.cpp:55] Training Batch 2
I0306 13:48:52.161989  1019 convert_cifar_data.cpp:55] Training Batch 3
I0306 13:48:53.653244  1019 convert_cifar_data.cpp:55] Training Batch 4
I0306 13:48:55.608544  1019 convert_cifar_data.cpp:55] Training Batch 5
I0306 13:49:59.846719  1019 convert_cifar_data.cpp:73] Writing Testing data
I0306   13:49:59.847904      1019   db_lmdb.cpp:35]   Opened   lmdb   /mnt/caffe-
cpu/examples/cifar10/cifar10_test_lmdb

Computing image mean...
I0306   13:53:50.209133      1074   db_lmdb.cpp:35]   Opened   lmdb   /mnt/caffe-
cpu/examples/cifar10/cifar10_train_lmdb
I0306 13:53:50.225416  1074 compute_image_mean.cpp:70] Starting iteration
I0306 13:53:51.304004  1074 compute_image_mean.cpp:95] Processed 10000 files.
I0306 13:53:53.283586  1074 compute_image_mean.cpp:95] Processed 20000 files.
I0306 13:53:55.271657  1074 compute_image_mean.cpp:95] Processed 30000 files.
I0306 13:53:56.383890  1074 compute_image_mean.cpp:95] Processed 40000 files.
I0306 13:53:57.580297  1074 compute_image_mean.cpp:95] Processed 50000 files.
I0306  13:53:57.580576   1074  compute_image_mean.cpp:108]  Write  to  /mnt/caffe-
cpu/examples/cifar10/mean.binaryproto
I0306 13:53:57.582489  1074 compute_image_mean.cpp:114] Number of channels: 3
I0306 13:53:57.582587  1074 compute_image_mean.cpp:119] mean_value channel [0]:
125.307
I0306 13:53:57.582872  1074 compute_image_mean.cpp:119] mean_value channel [1]:
122.95
I0306 13:53:57.582953  1074 compute_image_mean.cpp:119] mean_value channel [2]:
113.865

#./caffe.bin    test    -model    cifar10_quick_train_test.prototxt    -weights
cifar10_quick_iter_5000.caffemodel
…
…
I0306 13:55:38.627992  1116 caffe.cpp:313] Batch 48, accuracy = 0.5
I0306 13:55:38.628142  1116 caffe.cpp:313] Batch 48, loss = 0.569972
I0306 13:55:38.674666  1116 caffe.cpp:313] Batch 49, accuracy = 0.5
I0306 13:55:38.674819  1116 caffe.cpp:313] Batch 49, loss = 0.840962
I0306 13:55:38.674933  1116 caffe.cpp:318] Loss: 0.707087
I0306 13:55:38.675132  1116 caffe.cpp:330] accuracy = 0.77
I0306 13:55:38.675346  1116 caffe.cpp:330] loss = 0.707087 (* 1 = 0.707087 loss)
```

Figure 5.5: Preparing the CIFAR10 dataset and measuring accuracy on ARM. The model achieved ~71% accuracy very close to the original baseline (75%).

5.1.2 Neural Network Analysis

For this task we utilized the Caffe's Python interface "pycaffe". With the Caffe python module we could load models, do forward and backward computations, handle IO, visualize networks, and even instrument model solving. All model data, derivatives, and parameters are exposed for reading and writing and that is essential to break into pieces every model. For this task we used the *IPython Jupyter Notebook* found in Caffe source files which is browser based interactive and scripting interpreter tool, great for visualizations and sharing collaborative work. Next we present some visualizations and layer data for different network models through the *Jupyter Notebook*.

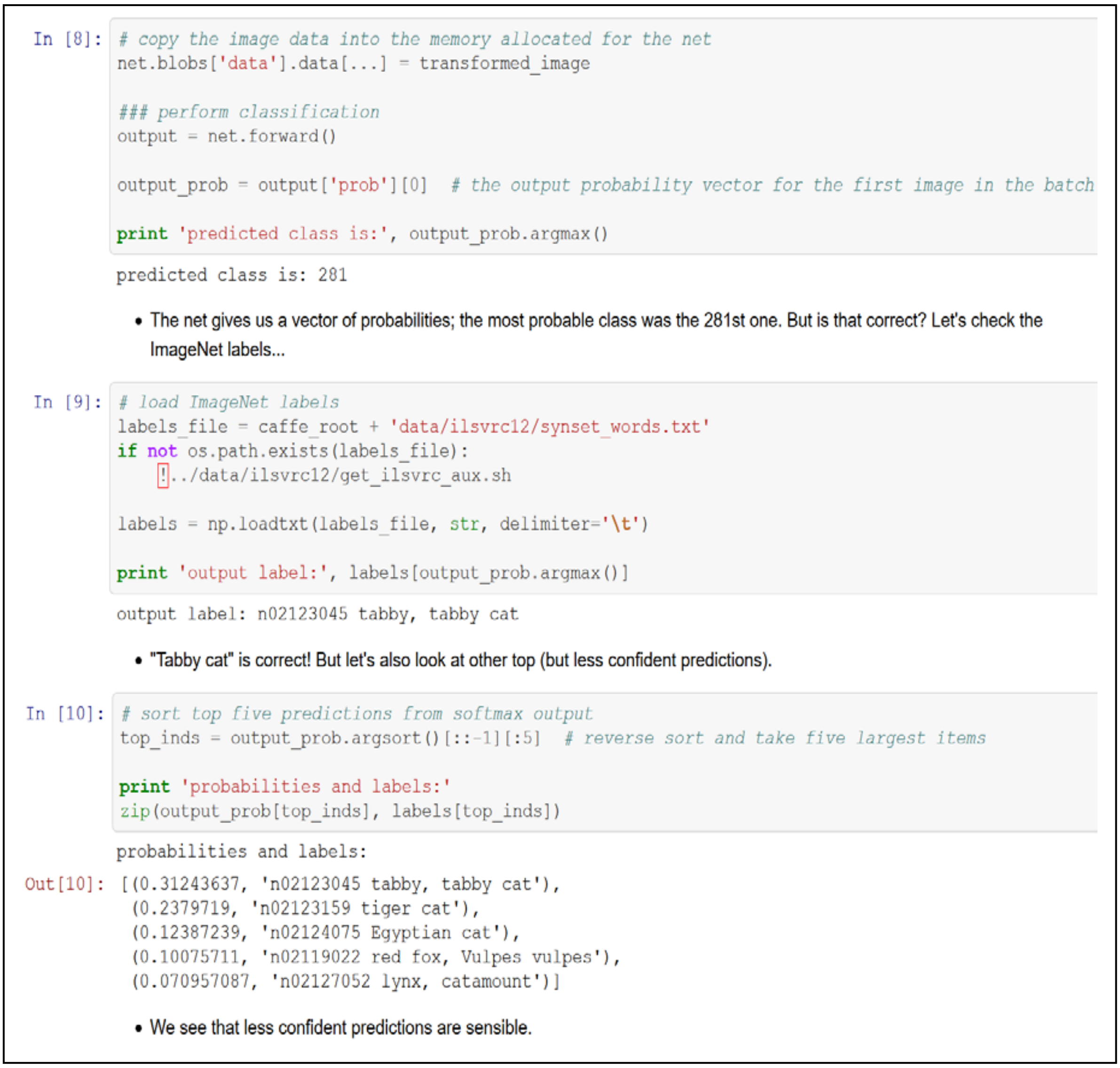
Preparation of input After setting up python, numpy and matplotlib we loaded Caffe module into the notebook. Next, we set up Caffe's input preprocessing configuration for reading images (configured in BGR format as opposed to RGB) and loaded the example image of a cat.



```
In [61]: image = caffe.io.load_image(caffe_root + 'examples/images/cat.jpg')
         transformed_image = transformer.preprocess('data', image)

         plt.imshow(image)
Out[61]: <matplotlib.image.AxesImage at 0x7f85fa3a2dd0>
```

Figure 5.6: Image input in Caffe with IPython Notebook.

Python Image Classification After configuring Caffe module and loading the input image, we then fed it into the network and performed the forward pass to compute the probability vector that will output the predicted class. As it is shown below the predicted class has the number 281 of the 1000 ImageNet classes which corresponds to the label “tabby cat”. The probability for this predicted class is ~31% but we output the top 5 predictions which are all very similar to the input image.[2]



```
In [8]: # copy the image data into the memory allocated for the net
        net.blobs['data'].data[...] = transformed_image

        ### perform classification
        output = net.forward()

        output_prob = output['prob'][0]  # the output probability vector for the first image in the batch

        print 'predicted class is:', output_prob.argmax()

        predicted class is: 281
```

- The net gives us a vector of probabilities; the most probable class was the 281st one. But is that correct? Let's check the ImageNet labels...

```
In [9]: # load ImageNet labels
        labels_file = caffe_root + 'data/ilsvrc12/synset_words.txt'
        if not os.path.exists(labels_file):
            !../data/ilsvrc12/get_ilsvrc_aux.sh

        labels = np.loadtxt(labels_file, str, delimiter='\t')

        print 'output label:', labels[output_prob.argmax()]

        output label: n02123045 tabby, tabby cat
```

- "Tabby cat" is correct! But let's also look at other top (but less confident predictions).

```
In [10]: # sort top five predictions from softmax output
         top_inds = output_prob.argsort()[::-1][:5]  # reverse sort and take five largest items

         print 'probabilities and labels:'
         zip(output_prob[top_inds], labels[top_inds])

         probabilities and labels:
Out[10]: [(0.31243637, 'n02123045 tabby, tabby cat'),
          (0.2379719, 'n02123159 tiger cat'),
          (0.12387239, 'n02124075 Egyptian cat'),
          (0.10075711, 'n02119022 red fox, Vulpes vulpes'),
          (0.070957087, 'n02127052 lynx, catamount')]
```

- We see that less confident predictions are sensible.

Figure 5.7: Image classification with IPython Notebook.

[2] The probability values for this image are produced from loading the BVLC Reference CaffeNet model weights and differ from model to model.

Examining Layer Parameters A net acts just a black box but through IPython Notebook we can take examine some of the parameters and intermediate activations of a network structure. First we read out the structure of the net in terms of parameter shapes. The parameters are exposed as `net.params` in the python code. We need to index the resulting values with either [0] for weights or [1] for biases. Also, the *param* shapes typically have the form (*output_channels, input_channels, filter_height, filter_width*) (for the weights) and the one-dimensional shape (*output_channels*) (for the biases).

BVLC CaffeNet

```
conv1  (96, 3, 11, 11) (96)
conv2  (256, 48, 5, 5) (256)
conv3  (384, 256, 3, 3) (384)
conv4  (384, 192, 3, 3) (384)
conv5  (256, 192, 3, 3) (256)
fc6    (4096, 9216) (4096)
fc7    (4096, 4096) (4096)
fc8    (1000, 4096) (1000)
```

GoogleNet

```
conv1/7x7_52 (64, 3, 7, 7) (64)
conv2/3x3_reduce (64, 64, 1, 1) (64)
conv2/3x3 (192, 64, 3, 3) (192)
inception_3a/1x1 (64, 192, 1, 1) (64)
inception_3a/3x3_reduce (96, 192, 1, 1) (96)
inception_3a/3x3 (128, 96, 3, 3) (128)
inception_3a/5x5_reduce (16, 192, 1, 1) (16)
inception_3a/5x5 (32, 16, 5, 5) (32)
inception_3a/pool_proj (32, 192, 1, 1) (32)
…
…
inception_5b/5x5_reduce (48, 832, 1, 1) (48)
inception_5b/5x5 (128, 48, 5, 5) (128)
inception_5b/pool_proj (128, 832, 1, 1) (128)
loss3/c1assifier (1000, 1024) (1000)
```

SqueezeNet

```
convl (64, 3, 3, 3) (64)
fire2/squeeze1x1 (16, 64, 1, 1) (16)
fire2/expand1x1 (64, 16, 1, 1) (64)
fire2/expand3x3 (64, 16, 3, 3) (64)
fire3/squeeze1x1 (16, 128, 1, 1) (16)
fire3/expand1x1 (64, 16, 1, 1) (64)
fire3/expand3x3 (64, 16, 3, 3) (64)
fire4/squeeze1x1 (32, 128, 1, 1) (32)
fire4/expand1x1 (128, 32, 1, 1) (128)
fire4/expand3x3 (128, 32, 3, 3) (128)
fire5/squeezelxl (32, 256, 1, 1) (32)
fire5/expandlxl (128, 32, 1, 1) (128)
fire5/expand3x3 (128, 32, 3, 3) (128)
fire6/squeeze1x1 (48, 256, 1, 1) (48)
fire6/expand1x1 (192, 48, 1, 1) (192)
fire6/expand3x3 (192, 48, 3, 3) (192)
fire7/squeeze1x1 (48, 384, 1, 1) (48)
fire7/expand1x1 (192, 48, 1, 1) (192)
fire7/expand3x3 (192, 48, 3, 3) (192)
fire8/squeeze1x1 (64, 384, 1, 1) (64)
fire8/expand1x1 (256, 64, 1, 1) (256)
fire8/expand3x3 (256, 64, 3, 3) (256)
fire9/squeeze1x1 (64, 512, 1, 1) (64)
fire9/expand1x1 (256, 64, 1, 1) (256)
fire9/expand3x3 (256, 64, 3, 3) (256)
conv10 (1000, 512, 1, 1) (1000)
```

CIFAR10

```
conv1  (32, 3, 5, 5) (32)
conv2  (32, 32, 5, 5) (32)
conv3  (64, 32, 5, 5) (64)
ip1    (64, 1024) (64)
ip2    (10, 64) (10)
```

Useful information can be extracted from the different model parameters. For example we can see that in the first conv layer of each network the output channels are the number of model filters that are used for the first layer (i.e. CaffeNet has 96). Also, the last layer parameters that calculate the probability for each label have 1000 output channels because it is the number of ImageNet labels whereas on the CIFAR10 dataset where we have 10 labels, the channels are 10.

Figure 5.8: Comparison of weight and bias parameters in every layer between network models.

Examining First Layer Filters Caffe IPython Notebook example provides a method to visualize the filters of a network layer in a grid of multiple small rectangular heatmaps that contain each filter. In this task we visualized the learned filters on the first layer only and compared them between different models. For example, it's worth mentioning that BVLC CaffeNet initializes the filters in every layer with a Gaussian distribution. Also, the net paramateres are exposed again as `net.params` in the code so we access the first convolution layer with *`net.params['conv1'][0].data`*. These filters as seen below, express the low level reasoning of the neural network as they try to find specific edges or textures in the image.

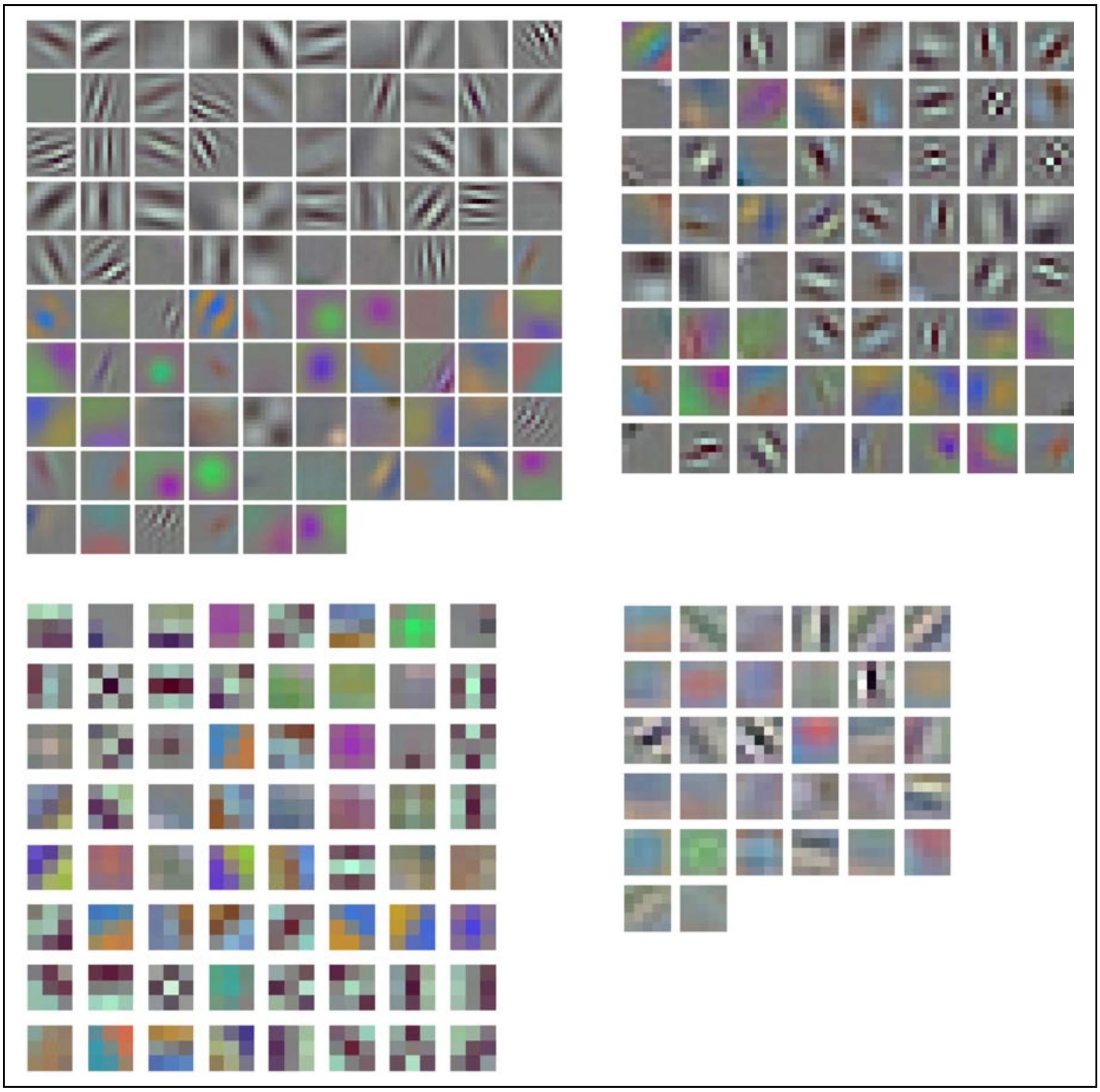
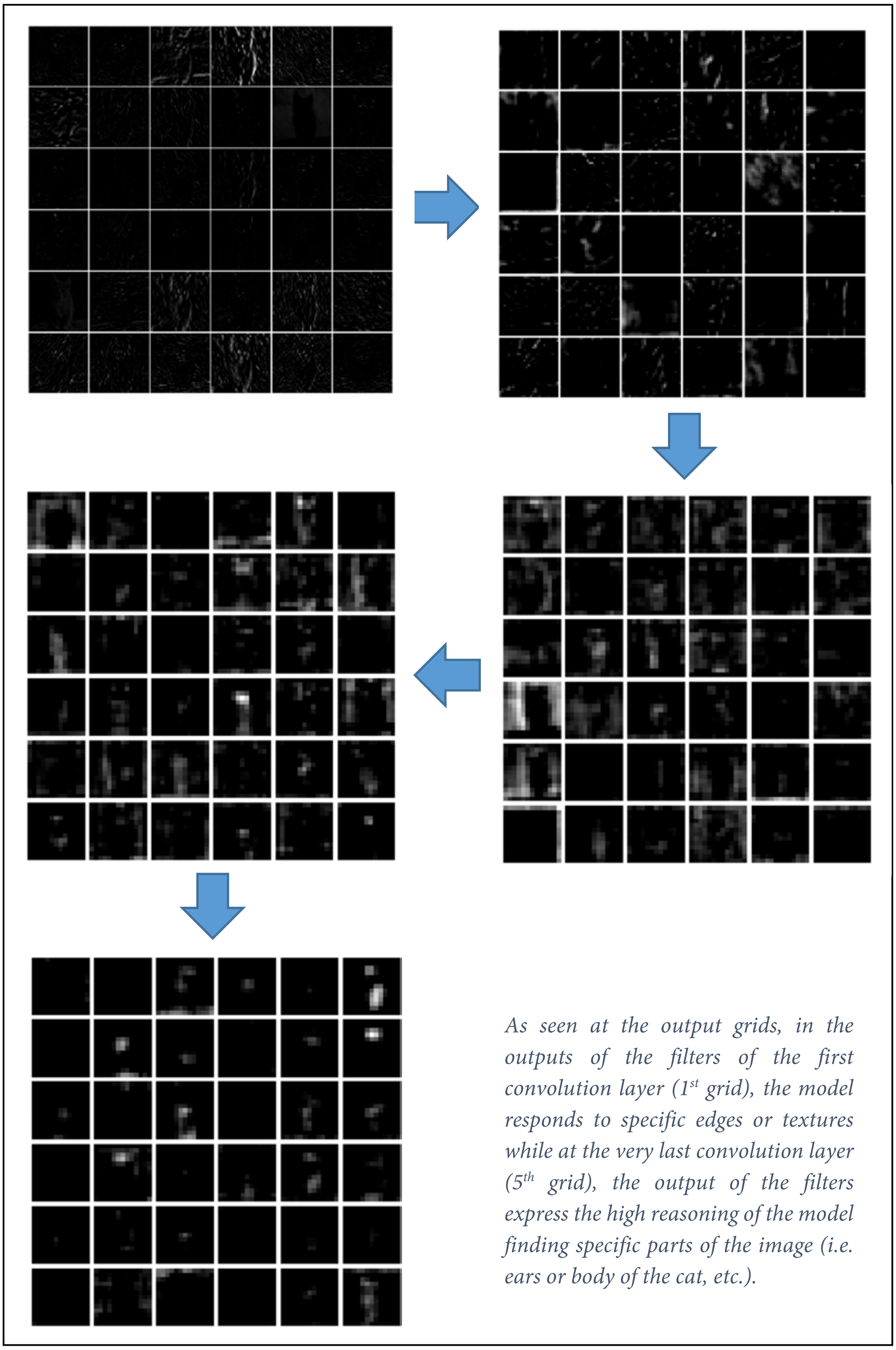


Figure 5.9: Comparison of first layer filters between models.
CaffeNet (top-left), GoogleNet (top-right),
SqueezeNet (bottom-left), CIFAR10 (bottom-right)

Examining Layer Outputs Now we analyze the output of the rectified responses of the filters from the first convolution layer up to the fifth (showing only 36 in each layer). We used the BVLC CaffeNet model with input the example image of a cat, as already shown previously.



As seen at the output grids, in the outputs of the filters of the first convolution layer (1st grid), the model responds to specific edges or textures while at the very last convolution layer (5th grid), the output of the filters express the high reasoning of the model finding specific parts of the image (i.e. ears or body of the cat, etc.).

Figure 5.10: Rectified Responses from the first to the fifth convolution layer on BVLC CaffeNet.

5.2 Hardware Accelerator Performance

In this section we analyze the performance of our designed accelerator function GEMM and the two hardware kernels (*mmult_accel, madd_accel*) on ZC702 FPGA. We start (section 5.2.1) with the kernel function analysis regarding latency, software tracing, data motion analysis, resource utilization etc. Next (section 5.2.2), we give several metrics on the hardware accelerator regarding speed-up from ARM and other architectures using a C++ test case program as a test-bench to determine performance on different matrices.

5.2.1 Hardware Kernel Analysis

After selecting the two kernels *mmult_accel* and *mmad_accel* of our wrapper function GEMM for hardware acceleration we ran the Estimate Performance from SDSoC Project Overview in order to analyze the hardware functions. SDSoC Environment instead of synthesizing the system to bitstream, it computes an estimate of the performance based on estimated latencies for the hardware functions and data transfer time estimates for the callers of hardware functions.

We generally tried to optimize the block sizes of the kernels to be multiples of the matrix dimensions while being able to fit into the FPGA's BRAMs. We found that 192 × 192 block size is optimal in terms of resources and overall performance. Also, we implemented an 8-bit fixed point implementation using *ap_fixed* datatype and achieved significant results in terms of latency, acceleration and resource utilization. For the same block size, we could partition the arrays much more due to fixed point numbers as the FPGA utilizes more efficiently the resources with fixed point precision. Thus, we parallelized more the code and achieved ~6,5 × more GFLOPs on the kernels. Moreover, the latency was reduced to 15% and the clock speed of the ZC702 FPGA was increased to the max frequency of 200MHz (instead of 100MHz in the float implementation). This is possible because the architecture fully distributes the computation as well as all the required data onto the different computational units. There are no dependencies between the individual computational units, even their results are accumulated separately.

In this section we analyze in detail the hardware accelerator with float datatype that was used. However, we also make a comparison in performance with the fixed point implementation in many results of the analysis.

Data Motion Analysis Every transfer between the software program and our hardware kernels requires a data mover, which consists of a hardware component that moves the data, and an operating system-specific library function. The following tables list all the data movers and various properties for each used in each kernel. As seen below, the declared size for the float arrays is $192 \times 192 \times 4 = 36864 \times 4$, the system port used is the ACP and the DMA is the AXI DMA_SIMPLE.

Table 5.1: Data Motion Network for hardware kernels (float datatype)

Accelerator	Argument	IP Port	Direction	Declared Size(bytes)	Connection
madd_accel_1	A	A	IN	36864*4	mmult_accel_1:C
	B	B	IN	36864*4	ps7_S_AXI_ACP:AXIDMA_SIMPLE
	C	C	OUT	36864*4	ps7_S_AXI_ACP:AXIDMA_SIMPLE
mmult_accel_1	A	A	IN	36864*4	ps7_S_AXI_ACP:AXIDMA_SIMPLE
	B	B	IN	36864*4	ps7_S_AXI_ACP:AXIDMA_SIMPLE
	C	C	OUT	36864*4	madd_accel_1:A
	alpha	alpha	IN	4	ps7_M_AXI_GP0:AXILITE:0xC

Table 5.2: Accelerator Callsites (float datatype)

Accelerator	Callsite	IP Port	Transfer Size(bytes)	Paged or Contiguous	Datamover Setup Time(CPU cycles)	Transfer Time(CPU cycles)
madd_accel_1	my_gemm.cpp:97:5	A	147456	contiguous		
		B	147456	contiguous	1123	246849
		C	147456	contiguous	1123	246849
mmult_accel_1	my_gemm.cpp:96:5	A	147456	contiguous	1123	246849
		B	147456	contiguous	1123	246849
		C	147456	contiguous		
		alpha	4	paged	0	13

Latency Analysis We will now analyze the latency in the *mmult_accel* kernel because it is responsible for the most computations of the matrix multiplication and possesses the largest portion of the execution time. The architecture that we implemented fully distributes the computation as well as all the required data onto the different computational units without having dependencies between the individual computational units. This resulted in low latency on the inner loops of the kernel function while having low initiation interval on the pipelining. The estimated clock speed (in ns) and the latency analysis are shown in the next figures comparing the float and fixed point datatype implementation.

Table 5.3: Summary of the clock timing (ns).
Floating Point (left), Fixed Point (right)

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	9.58	1.25

Clock	Target	Estimated	Uncertainty
ap_clk	5.00	5.09	0.63

Table 5.4: Summary of Latency (clock cycles).
Floating Point (left), Fixed Point (right)

Latency		Interval		Pipeline Type
min	max	min	max	
259022	259022	259023	259023	none

Latency		Interval		Pipeline Type
min	max	min	max	
74128	74128	74129	74129	none

Table 5.5: Summary of Loop Latency (clock cycles).
Floating Point (up), Fixed Point (down)

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
-Loop1	36875	36875	13	1	1	36864	yes
-Loop2	222143	222143	966	6	1	36864	yes

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
-Loop1	36870	36870	8	1	1	36864	yes
-Loop2	37254	37254	392	1	1	36864	yes

Resource Utilization The performance estimation output shows a detailed resource utilization on all the hardware functions used in the program. All the instances and modules created as well as the memory banks, LUTs, etc. are clearly shown in the Utilization Estimate report. Each expression (i.e. multiplication, addition, compare) in the report, shows the exact resources needed to accomplish the operation. In the following table we summarize all the resources of the FPGA fabric used for the hardware kernels comparing the float and fixed point datatype implementation.

Table 5.6: Resource Utilization on ZC702 FPGA.
Floating Point (left), Fixed Point (right)

Resource	Used	Total	%Utilization
DSP	164	220	75,55
BRAM	128	140	91,43
LUT	45557	53200	85,63
FF	24445	106400	22,97

Resource	Used	Total	%Utilization
DSP	193	220	87,73
BRAM	96	140	68,57
LUT	36681	53200	68,95
FF	26683	106400	25,08

5.2.2 Hardware function Acceleration

In order to measure the acceleration of the GEMM function on ZC702 FPGA SoC we constructed a C++ test bench that tests our custom function with multiple matrices with different and irregular dimensions checking if the result is correct compared with the *golden* version which is the plain simple GEMM function on CPU. Also, we wanted to compare our function performance with the simple GEMM on ARM CPU and with BLAS GEMM which is used in Caffe by default and is an optimized library for GEMM operations. The SDSoC Environment provided a simple, source code annotation based time-stamping API (*sds_clock_counter()*) that was used to measure application performance. With this API we measured the exact time (in CPU cycles) that each function needed to operate by running the application on target FPGA SoC and printing the actual run-time.

In this section we present the actual performance results when the application is running on board comparing it with different implementations. Moreover, we provide a comparison of the hardware accelerator performance with other architectures such as CPU or GPU.

Kernel Performance The most important part of the GEMM function is the two hardware kernels (*mmult_accel, madd_accel*) that are responsible for the most computations (esp. *mmult_accel*). So in the next figure we compare different kernel approaches (i.e. different block sizes, different datatype etc.) and their performance (in GFLOPs). All the implementations perform the maximum parallelization that is possible (as described in Section 4.3) with the available resources. It's worth mentioning that in the final approach of the fixed point datatype we were able to push the clock to 200MHz without problems due to the low latency of the fixed point implementation. Also, as seen in the figure, in this approach we achieved almost half the maximum theoretical GFLOPs the ZC702 FPGA SoC can achieve. This is because the board consists of 220 DSPs each capable of doing 2 MACCs per cycle so on the maximum clock of 200MHz we have $220 \cdot 2 \cdot 200 \cdot 10^6 = 88\ GFLOPs$.

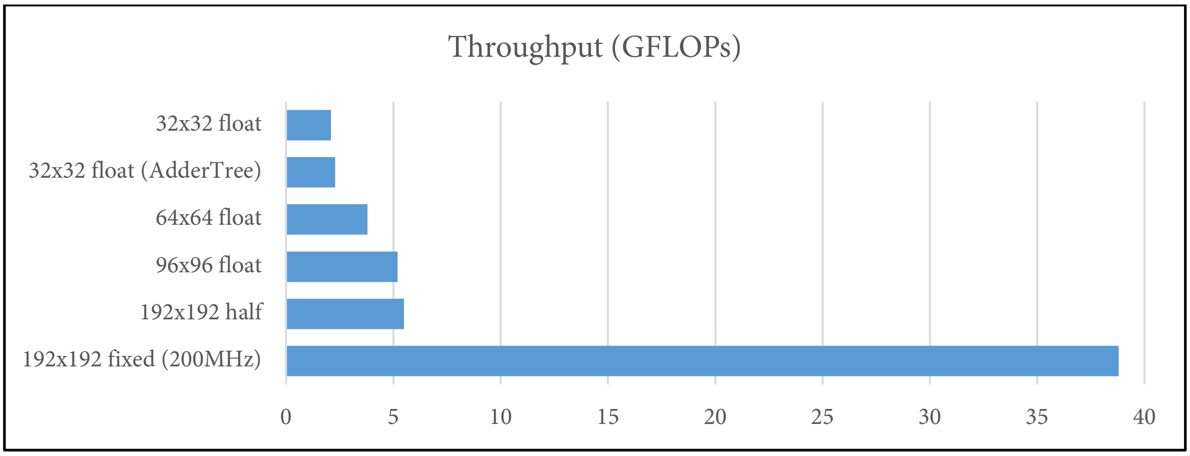


Figure 5.11: Performance of hardware kernels on different approaches.

Function Performance In order to have a general performance measurement of the function GEMM, we used the C++ test-bench that we described previously. This is the top-level function performance and it is crucial as it express the actual time our function produces the output of the final matrix C of the GEMM operation. The first figure presents the acceleration for different approaches on multiple matrix inputs while the second figure presents the acceleration of GEMM on different architectures. All the speed-ups are calculated when compared with the SW-only version of GEMM (naïve) on ARM as a baseline.

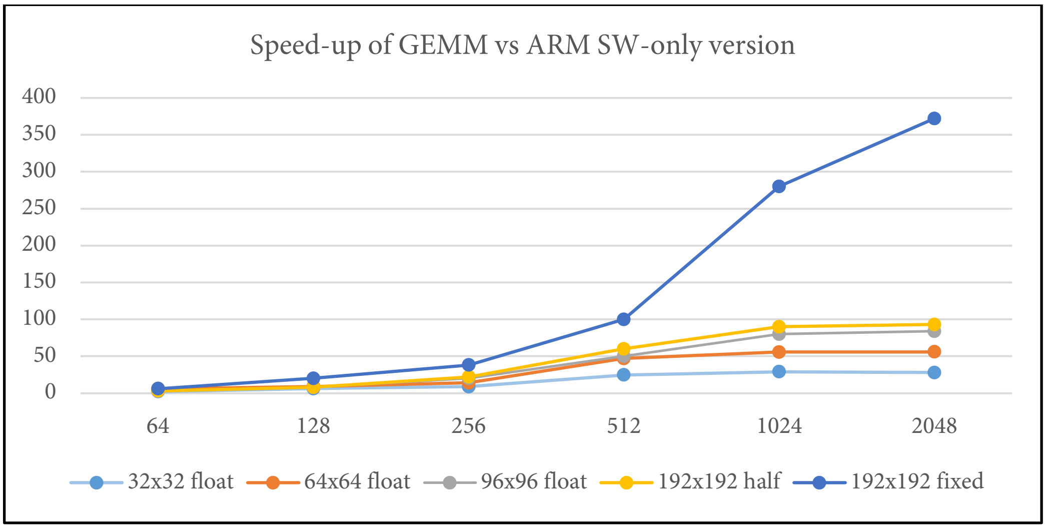


Figure 5.12: Comparison of GEMM acceleration for different approaches on multiple matrix inputs.

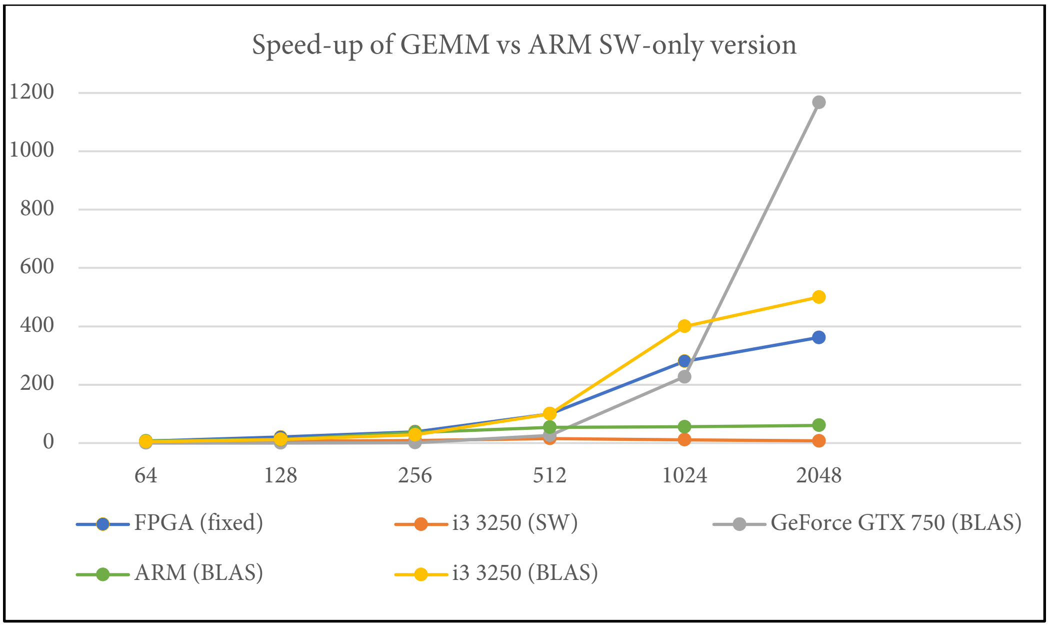


Figure 5.13: Comparison of GEMM acceleration for different architectures on multiple matrix inputs.

5.3 Caffe in CPU-FPGA performance

After integrating the hardware accelerator that we designed with the Caffe framework, we wanted to measure the overall performance of Caffe on the heterogeneous CPU-FPGA SoC (ZC702 board) and compare it with the float and fixed point implementation of the hardware accelerator (fixed point Caffe performance is an estimate based on kernel performance). So, in this section we measure the classification performance and accuracy on the new modified Caffe framework on the FPGA that utilizes our custom *GEMM* accelerator and we evaluate the power and energy consumption of our system as well.

Inference-Learning Performance In order to have a performance measurement of the image inference/learning, that is the time it takes for the model to perform a full forward/backward pass, we used Caffe's *time* tool. In the following figure, we present the network propagation times for different implementations and models (the fixed approach is simulated).

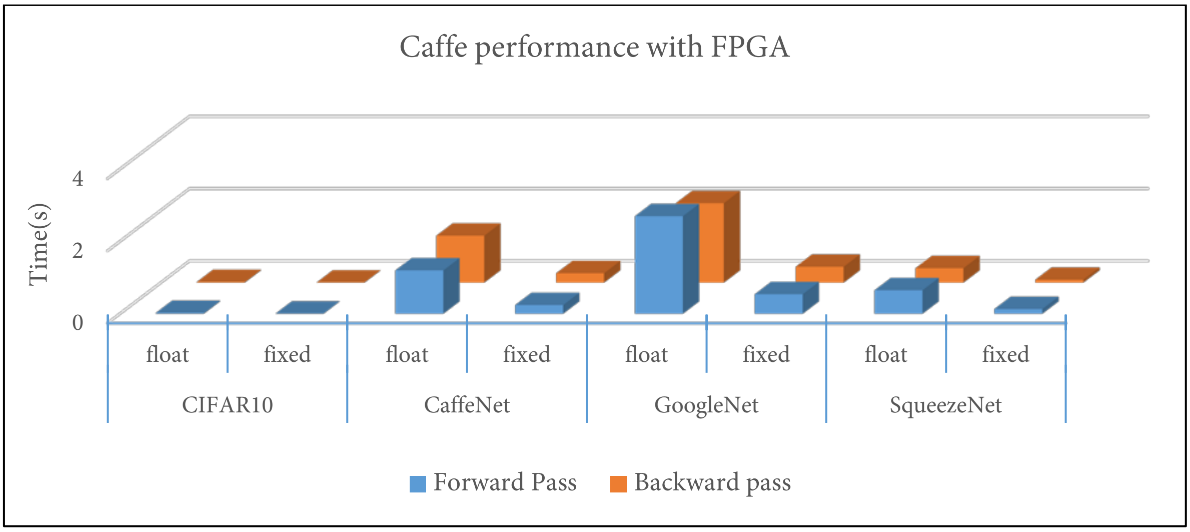


Figure 5.14: Comparison of execution time (s) in Forward/Backward Pass (Inference/Learning) between different GEMM implementation on different models.

Accuracy Using Caffe's *test* tool we were able to measure the accuracy on the FPGA-based Caffe. In the following figure we present the accuracy using the small CIFAR10 validation set between different implementations (we ran the whole data with 1000 iterations and batch 10).

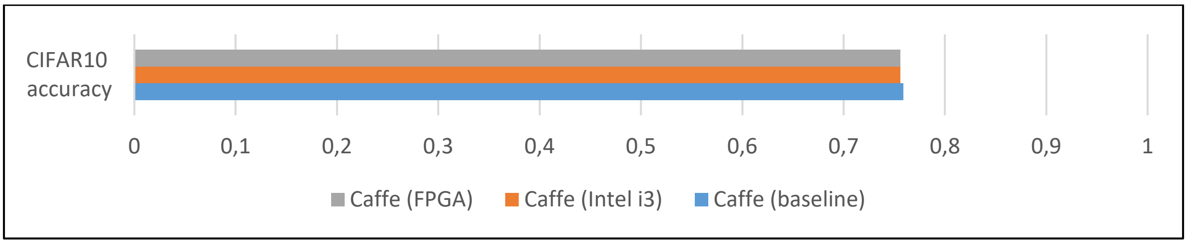


Figure 5.15: Prediction accuracy on CIFAR10 dataset between different implementations.

Power consumption The energy consumption of the complete CPU-FPGA system running Caffe has been evaluated using the Xilinx Power Estimator (XPE) for the ZC702 board. The power measurements include all conversion losses, peripheral devices, as well as the system fan. The system has not been optimized for low-power operation due to the advanced project time, and a significant amount of energy is already consumed in the idle state.

All measurements regarding the ZC702 board's power dissipation were considered with caution. With all known improvements (better memory communication, fixed-point arithmetic, ideal GEMM dimensions etc.) applied, the power efficiency could possibly be boosted to a respectable improved point.

The average power consumption of the FPGA SoC (both the AP SoC and the DRAM) during the Caffe forward passes (inference) is about 3.9 Watt. In this case, we can achieve up to 7x better energy efficiency compared with an Intel i3 running at 3,5GHz due to the lower power consumption.

Table 5.7: Power consumption on ZC702 @ 12V.

Devices	Power dissipation (Watt)
Transceiver	0,0 W
I/O	0,674 W
PS+FPGA	3,020 W
Device Static	0,211 W
Off-chip devices	0,0 W
Total	3,905 W

The next figure presents the performance per Watt usage ($\frac{Images}{s}$/W) for 3 different architectures and compares the energy efficiency between these approaches. The FPGA implementation achieves 7x energy efficiency compared with the Intel i3 CPU (simulated with fixed point arithmetic implementation).

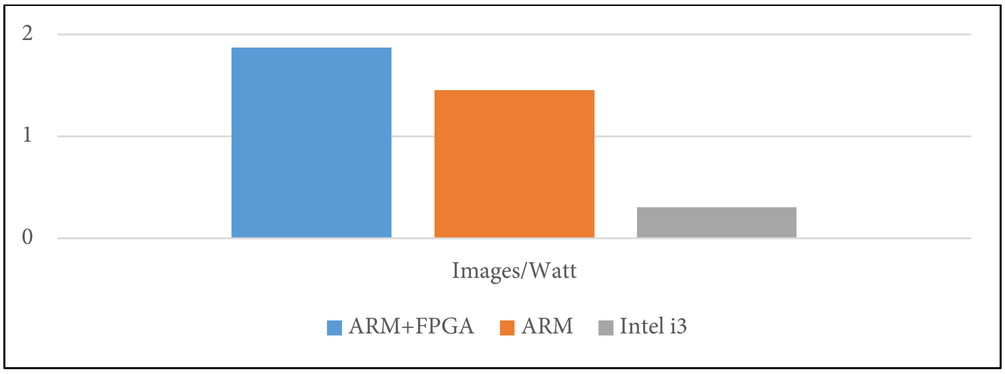


Figure 5.16: Energy efficiency of the final system for different architectures (SqueezeNet model used).

5.4 Caffe in AWS cloud performance

The Amazon Elastic Compute Cloud (Amazon EC2) provides machine learning practitioners and researchers with the infrastructure and tools to accelerate deep learning in the cloud, at any scale. With this service we were able to quickly launch Amazon EC2 instances, specifically the FPGA instance (f1.2xlarge), and experiment with the pre-installed Caffe on the cloud using the Machine Learning Development Stack from Xilinx. After creating the virtual server and selecting the Amazon Machine Image (AMI), we specified the instance settings and successfully launched and evaluated the FPGA-accelerated inference using the ready-to-run included network models for Caffe. In this paragraph, we analyze the steps to configure and launch these instances and finally evaluate the performance of Caffe on the cloud using this Development Stack.

Instance settings In the following tables we present the most important instance settings in order to successfully launch the Machine Learning Development Stack from Xilinx using the f1.2xlarge FPGA instance. The basic services required for this Stack are the EBS volumes and of course the EC2 instances.[3]

Table 5.7: Storage settings.

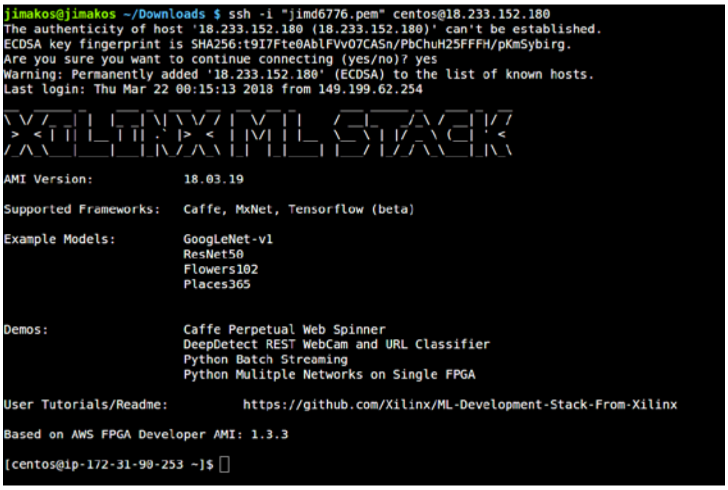
Volume Type	Device	Size (GB)	Volume Type
Root	/dev/sda1	70	gp2
ebs	/dev/sdb	8	gp2

Table 5.8: Security Groups settings

Type	Protocol	Port Range	Source
SSH	TCP	22	0.0.0.0/0
HTTP	TCP	88	0.0.0.0/0
Custom TCP rule	TCP	8080	0.0.0.0/0
Custom TCP rule	TCP	8888-8900	0.0.0.0/0
Custom TCP rule	TCP	8998-8999	0.0.0.0/0

[3] Note the security groups have port 22 for SSH and if desired, open ports 8080, 8888-8900, and 8998-8999 for web based demos through the Caffe web API.

Connecting to the instance For connecting via ‘ssh’ we utilized for encryption an ‘ssh’ private key and made connection using the Public DNS in the EC2 dashboard.



```
jimakos@jimakos ~/Downloads $ ssh -i "jimd6776.pem" centos@18.233.152.180
The authenticity of host '18.233.152.180 (18.233.152.180)' can't be established.
ECDSA key fingerprint is SHA256:t9I7Fte0AblFVvO7CASn/PbChuH25FFFH/pKmSybirg.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '18.233.152.180' (ECDSA) to the list of known hosts.
Last login: Thu Mar 22 00:15:13 2018 from 149.199.62.254

AMI Version:             18.03.19

Supported Frameworks:    Caffe, MxNet, Tensorflow (beta)

Example Models:          GoogLeNet-v1
                         ResNet50
                         Flowers102
                         Places365


Demos:                   Caffe Perpetual Web Spinner
                         DeepDetect REST WebCam and URL Classifier
                         Python Batch Streaming
                         Python Mulitple Networks on Single FPGA

User Tutorials/Readme:           https://github.com/Xilinx/ML-Development-Stack-From-Xilinx

Based on AWS FPGA Developer AMI: 1.3.3

[centos@ip-172-31-90-253 ~]$ 
```

Figure 5.17: Successfully established connection on the AWS instance.

Caffe simulation First, we executed the 8/16 bit networks through Caffe with the included GoogLeNet-v1, ResNet-50, Flowers-102 and Places-365 models. We navigated to `/home/centos/xfdnn_18_03_19/caffe/` and started the docker. The results of running the network models on the FPGA are presented in the following figure. This is not the peak FPGA performance because this includes loading the weights and instructions on the FPGA and this version of Caffe does not have a multi-process pipeline to fully utilize the FPGA. [4]

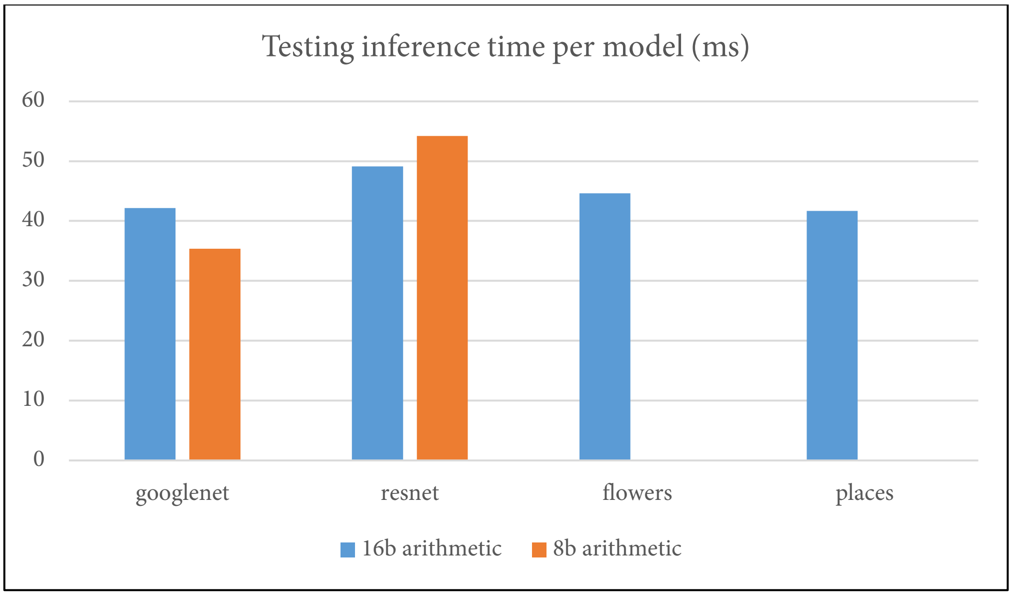
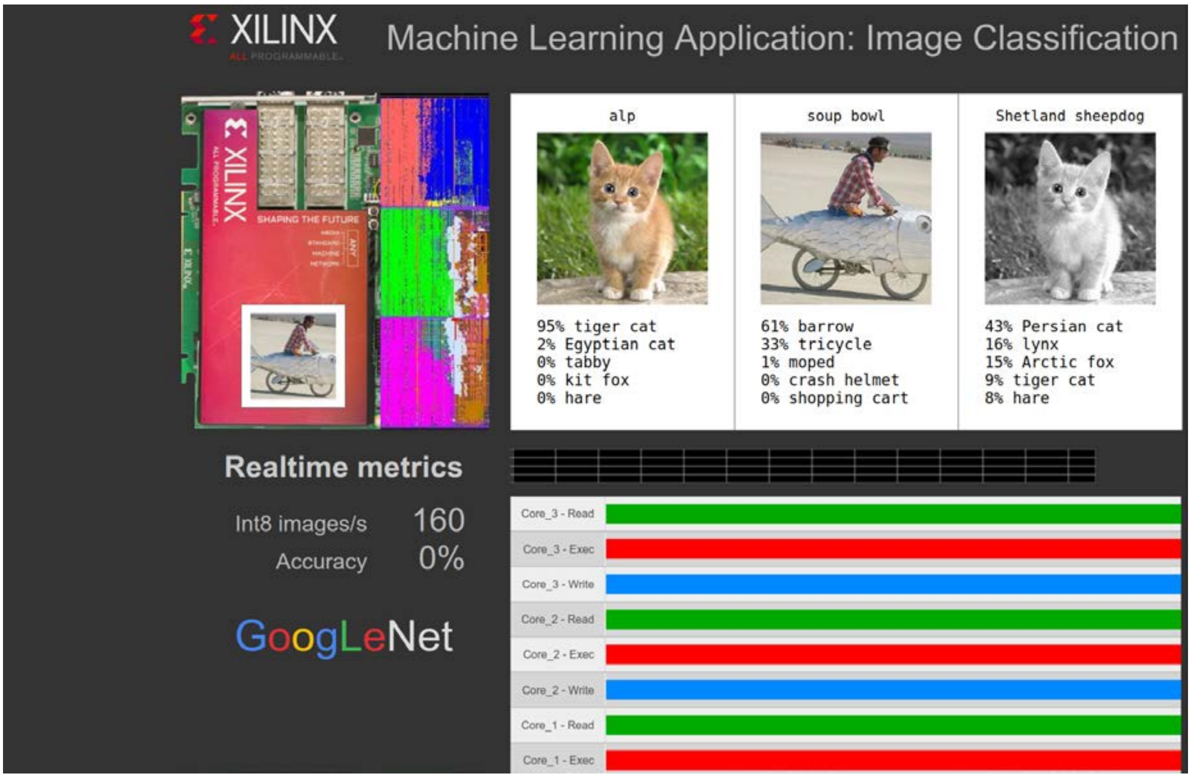


Figure 5.18: Inference time on different models using 8/16 bit precision using the AWS FPGA instance.

[4] Note that the 8b support for flowers and places model was not supported in order to test it.

Caffe web demo This demo on the GoogLeNet v1 8-bit network is meant to use ImageNet ILSVRC2012 validation files in order to test the full performance of the framework. We monitored the image classifications from our internet web based browser using the address provided by the tutorial and the results are presented below. The maximum int8 images/s that we achieved reached 160. For the input validation images we selected only a few instead of inserting the whole ImageNet dataset which is very large and that's why the accuracy measure is not showing correctly.



XILINX ALL PROGRAMMABLE. Machine Learning Application: Image Classification

alp

95% tiger cat
2% Egyptian cat
0% tabby
0% kit fox
0% hare

soup bowl

61% barrow
33% tricycle
1% moped
0% crash helmet
0% shopping cart

Shetland sheepdog

43% Persian cat
16% lynx
15% Arctic fox
9% tiger cat
8% hare

Realtime metrics

Int8 images/s	160
Accuracy	0%

GoogLeNet

Figure 5.19: Image classification performance on the web based Caffe demo using the AWS FPGA instance.

6 Conclusion

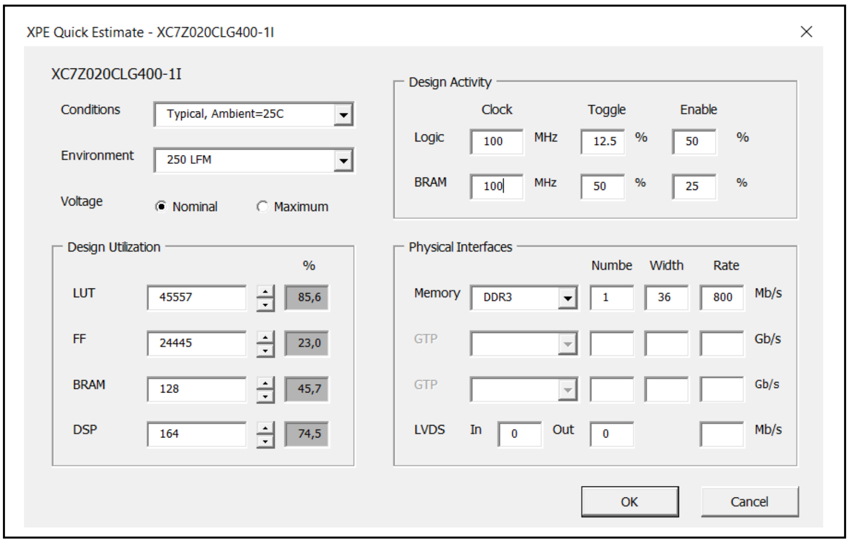
Deep Neural Networks represent a universal model, which can empower specific applications to solve a great variety of tasks. In this field, FPGAs can improve significantly the performance and the energy of these applications by using hardware accelerators. However, DNN frameworks such as Caffe do not officially support the transparent utilization of such acceleration modules.

This work clearly presents an end-to-end DNN deployment on Xilinx FPGA SoC using Caffe framework that utilizes hardware accelerators in order to achieve better performance and efficiency. We first ported the whole Caffe framework and all its library dependencies to work and run on the embedded ARM CPU with the cross-compilation method. Then, through full profiling on the framework, we determined the computational bottlenecks in order to specify the right function for hardware acceleration which was the GEMM function (General Matrix Multiplication). Next we described the implementation and optimization strategies on software and hardware of the accelerator so as to utilize the massive parallelism and high bandwidth memory access patterns of the FPGA design. The FPGA accelerator has been synthesized with High-Level Synthesis using Xilinx SDSoC Development Environment on the Xilinx Zynq ZC702 SoC and reaches a $380x$ speed-up with the maximum clock frequency of 200MHz and device utilization of ~80%. The evaluation and validation of the hardware function was successful producing correct results for different matrix sizes and dimensions. Finally, we integrated the hardware accelerator with Caffe framework and ran it successfully on the heterogeneous CPU-FPGA system which leverages the FPGA architecture with the hardware accelerator that we designed. For the final system evaluation we measured the performance of image classification and the results showed that the proposed system can reduce inference time up to ~10% compared with the ARM CPU and also reduce the energy consumption having less than 0.4% accuracy drop.

The CPU-FPGA based system with support of Caffe framework has been assembled into a fully working proof-of-concept system on the Xilinx Zynq-7000 All Programmable platform. This project clearly demonstrates the feasibility of FPGA-based embedded DNN implementations. The current solution already exhibits a reasonable performance with a number of techniques for substantial gains in throughput and power efficiency which have been pointed out.

Power estimation with Xilinx XPE

A.



XPE Quick Estimate - XC7Z020CLG400-1I

XC7Z020CLG400-1I

Conditions	Typical, Ambient=25C
Environment	250 LFM
Voltage	● Nominal ○ Maximum

Design Activity

	Clock		Toggle		Enable	
Logic	100	MHz	12.5	%	50	%
BRAM	100	MHz	50	%	25	%

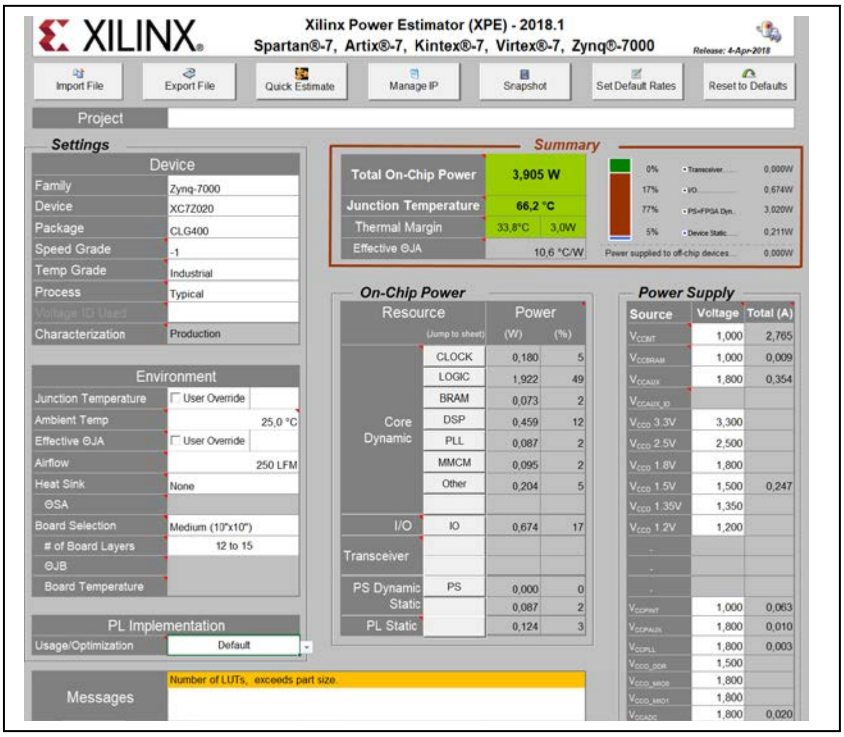
Design Utilization

		%
LUT	45557	85,6
FF	24445	23,0
BRAM	128	45,7
DSP	164	74,5

Physical Interfaces

		Numbe	Width	Rate	
Memory	DDR3	1	36	800	Mb/s
GTP					Gb/s
GTP					Gb/s
LVDS	In 0	Out 0			Mb/s

Figure A.1: FPGA logic settings for power estimation.



Xilinx Power Estimator (XPE) - 2018.1
Spartan®-7, Artix®-7, Kintex®-7, Virtex®-7, Zynq®-7000
Release: 4-Apr-2018

Project

Settings

Device	
Family	Zynq-7000
Device	XC7Z020
Package	CLG400
Speed Grade	-1
Temp Grade	Industrial
Process	Typical
Voltage ID Used	
Characterization	Production

Environment		
Junction Temperature	□ User Override	
Ambient Temp	25,0 °C	
Effective ΘJA	□ User Override	
Airflow	250 LFM	
Heat Sink	None	
ΘSA		
Board Selection	Medium (10"x10")	
# of Board Layers	12 to 15	
ΘJB		
Board Temperature		

PL Implementation	
Usage/Optimization	Default

Summary

Total On-Chip Power	3,905 W	
Junction Temperature	66,2 °C	
Thermal Margin	33,8°C	3,0W
Effective ΘJA	10,6 °C/W	

0%	Transceiver	0,000W
17%	I/O	0,674W
77%	PS+FPGA Dyn.	3,020W
5%	Device Static	0,211W
Power supplied to off-chip devices		0,000W

On-Chip Power

Resource	(Jump to sheet)	Power (W)	(%)
Core Dynamic	CLOCK	0,180	5
	LOGIC	1,922	49
	BRAM	0,073	2
	DSP	0,459	12
	PLL	0,087	2
	MMCM	0,095	2
	Other	0,204	5
I/O	IO	0,674	17
Transceiver			
PS Dynamic	PS	0,000	0
Static		0,087	2
PL Static		0,124	3

Power Supply

Source	Voltage	Total (A)
V_{CCINT}	1,000	2,765
V_{CCBRAM}	1,000	0,009
V_{CCAUX}	1,800	0,354
V_{CCAUX_IO}		
V_{CCO} 3.3V	3,300	
V_{CCO} 2.5V	2,500	
V_{CCO} 1.8V	1,800	
V_{CCO} 1.5V	1,500	0,247
V_{CCO} 1.35V	1,350	
V_{CCO} 1.2V	1,200	
-		
-		
-		
V_{CCPINT}	1,000	0,063
V_{CCPAUX}	1,800	0,010
V_{CCPLL}	1,800	0,003
V_{CCO_DDR}	1,500	
V_{CCO_MIO0}	1,800	
V_{CCO_MIO1}	1,800	
V_{CCADC}	1,800	0,020

Messages

Number of LUTs, exceeds part size.

Figure A.2: Final power estimation overview.

FPGA accelerator Library

B.

Listing 1. *main_test.cpp*

```
#include < iostream >
#include < stdlib.h >
#include < stdint.h >
#include < math.h >
#include < stdio.h >
#include < getopt.h >
#include < time.h >
#include "sds_lib.h"

//custom and blas libraries header files
#include "gemm_accel.h"
#include "gemm_golden.h"
#include "/home/jimakos/Desktop/caffe_libs_hardf/openblas_libs/include/cblas.h"

int NUM_TESTS = 1;

#define PIPELINED 1
#define USE_FIXED 0
#define USE_BRAMS 1

static int Mi, Ni, Ki;

class perf_counter {
    public: uint64_t tot, cnt, calls;
    perf_counter(): tot(0), cnt(0), calls(0) {};
    inline void reset() {
        tot = cnt = calls = 0;
    }
    inline void start() {
        cnt = sds_clock_counter();
        calls++;
    };
    inline void stop() {
        tot += (sds_clock_counter() - cnt);
    };
    inline uint64_t avg_cpu_cycles() {
        return (tot / calls);
    };
};

static void init_arrays(float * A, float * B, float * C_sw, float * C, float * C_blas)
 {
    int temp;
    for (int i = 0; i < Mi; i++) {
        for (int j = 0; j < Ki; j++) {
            A[i * Ki + j] = rand() % 5;
```

```
      if (A[i * Ki + j] > 2) A[i * Ki + j] = 0.01; //weights simulation
    }
  }
  for (int i = 0; i < Ki; i++) {
    for (int j = 0; j < Ni; j++) {
      B[i * Ni + j] = rand() % 2;
    }
  }
  for (int i = 0; i < Mi; i++) {
    for (int j = 0; j < Ni; j++) {
      temp = rand() % 2;
      C_sw[i * Ni + j] = temp;
      C[i * Ni + j] = temp;
      C_blas[i * Ni + j] = temp;
    }
  }
}

static int result_check(float * C, float * C_sw) {
  int flag = 0;
  for (int i = 0; i < Mi * Ni; i++) {
    if (fabs(C_sw[i] - C[i]) > 1) { //good overall error for accuracy
      printf("%f <> %f\n", (float) C_sw[i], (float) C[i]);
      flag = 1;
    }
  }
  return flag;
}

int mmult_test(float * A, float * B, float * C_sw, float * C, float * C_blas) {
  std::cout << "Testing " << NUM_TESTS << " iterations of SGEMM..." << std::endl;
  perf_counter hw_ctr, sw_ctr, sw_blas_ctr;

  for (int i = 0; i < NUM_TESTS; i++) {
    init_arrays(A, B, C_sw, C, C_blas);
    sw_ctr.start();
    gemm_golden(0, 0, Mi, Ni, Ki, 1.0, 0, A, B, C_sw);
    sw_ctr.stop();
    sw_blas_ctr.start();
    cblas_sgemm (CblasRowMajor,CblasNoTrans, CblasNoTrans, Mi, Ni, Ki, 1.0, A, Ki,
    B, Ni, 0, C_blas, Ni);
    sw_blas_ctr.stop();
    hw_ctr.start();
    my_gemm(0, 0, Mi, Ni, Ki, 1.0, A, Ki, B, Ni, 0, C, Ni);
    hw_ctr.stop();

if (result_check(C, C_sw)) return 1;
  }

  double flop = ((double) Mi) * Ni * Ki * NUM_TESTS;
  double gflop = flop * 1e-9;

  uint64_t sw_cycles = sw_ctr.avg_cpu_cycles();
  uint64_t hw_cycles = hw_ctr.avg_cpu_cycles();
  uint64_t sw_blas_cycles = sw_blas_ctr.avg_cpu_cycles();
      double speedup1 = (double) sw_cycles / (double) hw_cycles;
      double speedup2 = (double) sw_blas_cycles / (double) hw_cycles;
      double sw_sec = (double)((double) sw_cycles) / 667000000;
      double hw_sec = (double)((double) hw_cycles) / 667000000;
      double sw_blas_sec = (double)((double) sw_blas_cycles) / 667000000;
      printf("SW %lf s, %lf GFlops\n", sw_sec, gflop / sw_sec);
      printf("HW %lf s, %lf GFlops\n", hw_sec, gflop / hw_sec);
      printf("Increase = %lf\n", (sw_sec - hw_sec) / sw_sec);
      printf("RESULTS FOR INPUT: M=%d K=%d N=%d\n", Mi, Ki, Ni);
      printf("BLOCK_MATRIX: M=%d K=%d N=%d\n", MB, KB, NB);
      if (USE_FIXED) printf("USE_FIXED = 1\n");
    else printf("USE_FIXED = 0\n");
    if (USE_BRAMS) printf("USE_BRAMS = 1\n");
    else printf("USE_BRAMS = 0\n");
    printf("__________________________________________________\n");
    printf("SW seconds: = %lf\n", sw_sec);
    printf("HW seconds: = %lf\n", sw_blas_sec);
    printf("HW seconds: = %lf\n", hw_sec);
    std::cout << "Average number of CPU cycles running GEMM in SW: " << sw_cycles << std::endl;
    std::cout << "Average number of CPU cycles running GEMM in BLAS: " << sw_blas_cycles << std::endl;
    std::cout << "Average number of CPU cycles running GEMM in HW: " <<
    hw_cycles << std::endl;
    printf("__________________________________________________\n");
    std::cout << "Speed up from SW: " << speedup1 << std::endl;
    std::cout << "Speed up from BLAS: " << speedup2 << std::endl;
    return 0;
}

int main(int argc, char * argv[]) {
    int test_passed = 0;
    float * A, * B, * C_sw, * C, * C_blas;
    Mi = 2048;
    Ki = 2048;
    Ni = 2048;
    A = (float * ) malloc(Mi * Ki * sizeof(float));
    B = (float * ) malloc(Ki * Ni * sizeof(float));
    C = (float * ) malloc(Mi * Ni * sizeof(float));
    C_sw = (float * ) malloc(Mi * Ni * sizeof(float));
    C_blas = (float * ) malloc(Mi * Ni * sizeof(float));
    if (!A || !B || !C || !C_sw) {
        if (A) free(A);
        if (B) free(B);
        if (C) free(C);
        if (C_sw) free(C_sw);
        if (C_blas) free(C_blas);
        return 2;
    }
    test_passed = mmult_test(A, B, C_sw, C, C_blas);
    std::cout << "TEST " << (test_passed ? "FAILED" : "PASSED") << std::endl;
    printf("__________________________________________________\n");
    printf("__________________________________________________\n");

    NUM_TESTS = 2;
    Mi = 1024;
    Ki = 1024;
    Ni = 1024;
    test_passed = mmult_test(A, B, C_sw, C, C_blas);
    std::cout << "TEST " << (test_passed ? "FAILED" : "PASSED") << std::endl;
    printf("__________________________________________________\n");
    printf("__________________________________________________\n");
    Mi = 192; //
    Ki = 1152; // sample googlenet calculation
    Ni = 784; //

    test_passed = mmult_test(A, B, C_sw, C, C_blas);
    std::cout << "TEST " << (test_passed ? "FAILED" : "PASSED") << std::endl;
    printf("__________________________________________________\n");
    printf("__________________________________________________\n");


    free(A);
    free(B);
    free(C);
    free(C_sw);
    free(C_blas);
```

```
        Mi = 64; //
        Ki = 147; // sample googlenet calculation
        Ni = 12544; //
        A = (float * ) malloc(Mi * Ki * sizeof(float));
        B = (float * ) malloc(Ki * Ni * sizeof(float)); //re-malloc in order to
        C = (float * ) malloc(Mi * Ni * sizeof(float)); // use bigger arrays
        C_sw = (float * ) malloc(Mi * Ni * sizeof(float));
        C_blas = (float * ) malloc(Mi * Ni * sizeof(float));
        test_passed = mmult_test(A, B, C_sw, C, C_blas);
        std::cout << "TEST " << (test_passed ? "FAILED" : "PASSED") << std::endl;
        printf("_________________________________________________________\n");
        printf("_________________________________________________________\n");

        free(A);
        free(B);
        free(C);
        free(C_sw);
        free(C_blas);
        Mi = 192; //
        Ki = 576; //sample googlenet calculation
        Ni = 3136; //
        A = (float * ) malloc(Mi * Ki * sizeof(float));
        B = (float * ) malloc(Ki * Ni * sizeof(float));
        C = (float * ) malloc(Mi * Ni * sizeof(float));
        C_sw = (float * ) malloc(Mi * Ni * sizeof(float));
        C_blas = (float * ) malloc(Mi * Ni * sizeof(float));
        test_passed = mmult_test(A, B, C_sw, C, C_blas);
        std::cout << "TEST " << (test_passed ? "FAILED" : "PASSED") << std::endl;
        printf("_________________________________________________________\n");
        printf("_________________________________________________________\n");

        free(A);
        free(B);
        free(C);
        free(C_sw);
        free(C_blas);
        Mi = 64; //
        Ki = 27; //sample squeezenet calculation
        Ni = 12769; //
        A = (float * ) malloc(Mi * Ki * sizeof(float));
        B = (float * ) malloc(Ki * Ni * sizeof(float));
        C = (float * ) malloc(Mi * Ni * sizeof(float));
        C_sw = (float * ) malloc(Mi * Ni * sizeof(float));
        C_blas = (float * ) malloc(Mi * Ni * sizeof(float));
        test_passed = mmult_test(A, B, C_sw, C, C_blas);
        std::cout << "TEST " << (test_passed ? "FAILED" : "PASSED") << std::endl;
        printf("_________________________________________________________\n");
        printf("_________________________________________________________\n");

        free(A);
        free(B);
        free(C);
        free(C_sw);
        free(C_blas);
        Mi = 64; //
        Ki = 144; // sample squeezenet calculation
        Ni = 3136; //
        A = (float * ) malloc(Mi * Ki * sizeof(float));
        B = (float * ) malloc(Ki * Ni * sizeof(float));
        C = (float * ) malloc(Mi * Ni * sizeof(float));
        C_sw = (float * ) malloc(Mi * Ni * sizeof(float));
        C_blas = (float * ) malloc(Mi * Ni * sizeof(float));
        test_passed = mmult_test(A, B, C_sw, C, C_blas);
        std::cout << "TEST " << (test_passed ? "FAILED" : "PASSED") << std::endl;
        printf("_________________________________________________________\n");
        printf("_________________________________________________________\n");

        free(A);
        free(B);
        free(C);
        free(C_sw);
        free(C_blas);
        Mi = 128; //
        Ki = 288; // sample squeezenet calculation
        Ni = 784; //
        A = (float * ) malloc(Mi * Ki * sizeof(float));
        B = (float * ) malloc(Ki * Ni * sizeof(float));
        C = (float * ) malloc(Mi * Ni * sizeof(float));
        C_sw = (float * ) malloc(Mi * Ni * sizeof(float));
        C_blas = (float * ) malloc(Mi * Ni * sizeof(float));
        test_passed = mmult_test(A, B, C_sw, C, C_blas);
        std::cout << "TEST " << (test_passed ? "FAILED" : "PASSED") << std::endl;
        printf("_________________________________________________________\n");
        printf("_________________________________________________________\n");

        free(A);
        free(B);
        free(C);
        free(C_sw);
        free(C_blas);
        return (test_passed ? -1 : 0);
    }
```

Listing 2. *gemm_golden.cpp*

```

#include "gemm_golden.h"

// Normal A(M,K) X Normal B(K,N)
void mmult_nn(int M, int N, int K, float alpha, float * A, int lda, float * B, int ldb,          float * C, int ldc) {
        int i, j, k;
        float count;
        for (i = 0; i < M; ++i) {
            for (j = 0; j < N; j++) {
                count = 0;
                for (k = 0; k < K; k++) {
                    count += alpha * A[i * lda + k] * B[k * ldb + j];
                }
                C[i * ldc + j] += count;
            }
        }
    }

// Normal A(M,K) X Transport B(N,K)
void mmult_nt(int M, int N, int K, float alpha, float * A, int lda, float * B, int ldb,                 float * C, int ldc) {
        int i, j, k;
        float count;
        for (i = 0; i < M; ++i) {
            for (j = 0; j < N; j++) {
                count = 0;
                for (k = 0; k < K; k++) {
                    count += alpha * A[i * lda + k] * B[j * ldb + k];
                }
                C[i * ldc + j] += count;
            }
        }
    }
```

```

 // Transport A(K,M) X Normal B(K,N)
void mmult_tn(int M, int N, int K, float alpha, float * A, int lda, float * B, int ldb, float * C, int ldc) {
        int i, j, k;
        float count;
        for (i = 0; i < M; ++i) {
            for (j = 0; j < N; j++) {
                count = 0;
                for (k = 0; k < K; k++) {
                    count += alpha * A[k * lda + i] * B[k * ldb + j];
                }
                C[i * ldc + j] += count;
            }
        }
    }

// Transport A(K,M) X Transport B(N,K)
void mmult_tt(int M, int N, int K, float alpha, float * A, int lda, float * B, int ldb, float * C, int ldc) {
    int i, j, k;
    float count;
    for (i = 0; i < M; ++i) {
        for (j = 0; j < N; j++) {
            count = 0;
            for (k = 0; k < K; k++) {
                count += alpha * A[k * lda + i] * B[j * ldb + k];
            }
            C[i * ldc + j] += count;
        }
    }
}

void gemm_golden(int TA, int TB, int M, int N, int K, float alpha, float beta,
                 float * A, float * B, float * C) {
    int lda = (!TA) ? K : M;
    int ldb = (!TB) ? N : K;
    int ldc = N;
    int i, j;
    for (i = 0; i < M; i++) {
        for (j = 0; j < N; j++) {
            C[i * ldc + j] *= beta;
        }
    } // choose transport flag
    if (!TA && !TB) mmult_nn(M, N, K, alpha, A, lda, B, ldb, C, ldc);
    else if (TA && !TB) mmult_tn(M, N, K, alpha, A, lda, B, ldb, C, ldc);
    else if (!TA && TB) mmult_nt(M, N, K, alpha, A, lda, B, ldb, C, ldc);
    else mmult_tt(M, N, K, alpha, A, lda, B, ldb, C, ldc);
}
```

Listing 3. *gemm_golden.h*

```

#ifndef SRC_GEMM_GOLDEN_H_
#define SRC_GEMM_GOLDEN_H_

void gemm_golden(int TA, int TB, int M, int N, int K, float alpha, float beta,
                 float * A, float * B, float * C);

#endif /* SRC_GEMM_GOLDEN_H_ */
```

Listing 4. my_gemm.cpp

```

#include < stdio.h >
#include < string.h >
#include < stdlib.h >
#include < iostream >
#include < stdint.h >
#include "sds_lib.h"
#include "gemm_accel.h"

static data_t1 * A_tile, * B_tile, * C_tile, * R, * T;

void copy_tile_A(int TA, int M, int K, int i, int k, int lda,
    const float * A, data_t1 alpha) {
    int i_temp, j_temp, x, y;
    if (TA == 0) {
        for (i_temp = 0; i_temp < MB && i + i_temp < M; ++i_temp) {
            for (j_temp = 0; j_temp < KB && k + j_temp < K; ++j_temp) {
                x = i + i_temp;
                y = k + j_temp;
                A_tile[i_temp * KB + j_temp] = A[x * lda + y];
            }
        }
    } //TODO TA == 1
}

void copy_tile_B(int TB, int K, int N, int k, int j, int ldb,
    const float * B) {
    int i_temp, j_temp, x, y;
    if (TB == 0) {
        for (i_temp = 0; i_temp < KB && k + i_temp < K; ++i_temp) {
            for (j_temp = 0; j_temp < NB && j + j_temp < N; ++j_temp) {
                x = k + i_temp;
                y = j + j_temp;
                B_tile[i_temp * NB + j_temp] = B[x * ldb + y];
            }
        }
    } //TODO TB == 1
}

void my_gemm(int TA, int TB, int M, int N, int K, float alpha,
    const float * A, int lda,
    const float * B, int ldb, float beta,
    float * C, int ldc) {

    int i, j, k, x, y, i_temp, j_temp;

    A_tile = (data_t1 * ) sds_alloc(MB * KB * sizeof(data_t1));
    B_tile = (data_t1 * ) sds_alloc(KB * NB * sizeof(data_t1));
    C_tile = (data_t1 * ) sds_alloc(MB * NB * sizeof(data_t1));
    R = (data_t1 * ) sds_alloc(MB * NB * sizeof(data_t1));
    T = (data_t1 * ) sds_alloc(MB * NB * sizeof(data_t1));
    data_t1 a = alpha;

    if (!A_tile || !B_tile || !C_tile || !T || !R) {
        fprintf(stderr, "Error buffer allocation\n");
        exit(1);
    }

    memset(A_tile, 0, sizeof(data_t1) * MB * KB);
    memset(B_tile, 0, sizeof(data_t1) * KB * NB);
    memset(C_tile, 0, sizeof(data_t1) * MB * NB);
    memset(T, 0, sizeof(data_t1) * MB * NB);


    for (i = 0; i < M; i += MB) {
         for (j = 0; j < N; j += NB) {                           //set C_tile
              memset(C_tile, 0, sizeof(data_t1) * MB * NB);      //is beta not needed
            /*for(x = 0; x < MB; x++) {                          //if beta needed
             for(y = 0; y < NB; y++) {
                          i_temp = i+x;
                   j_temp = j+y;
                   if(i_temp < M && j_temp < N)
                          C_tile[x*NB+y] = beta*C[i_temp*ldc+j_temp];
                          else
                                 C_tile[x*NB+y] = 0;
                          }
         }*/
              for (k = 0; k < K; k += KB) { //compute C tile from A and B tiles
                   copy_tile_B(TB, K, N, k, j, ldb, B);
                   copy_tile_A(TA, M, K, i, k, lda, A, a);
                   mmult_accel(A_tile, B_tile, T, a);
                   madd_accel(T, C_tile, R);
                   memset(A_tile, 0, sizeof(data_t1) * MB * KB);
                   memset(B_tile, 0, sizeof(data_t1) * KB * NB);
                   memcpy(C_tile, R, sizeof(data_t1) * MB * NB);
              }
              for (x = 0; x < MB; x++) {  //copy C_tile back to main output of array C
                   for (y = 0; y < NB; y++) {
                        i_temp = i + x;
                        j_temp = j + y;
                        if (i_temp < M && j_temp < N)
                            C[i_temp * ldc + j_temp] = C_tile[x * NB + y];

                   }
              }
         }
    }
    sds_free(A_tile);
    sds_free(B_tile);
    sds_free(C_tile);
    sds_free(R);
    sds_free(T);
    return;
    }
```

Listing 5. *gemm_accel.h*

```
#define MB 192
#define KB 192
#define NB 192
//#include <ap_fixed.h>  //uncomment for fixed support
//#include <hls_half.h>  //uncomment for half float support
typedef float data_t;
typedef float data_t1;
//typedef ap_fixed<16,15,AP_TRN,AP_SAT_SYM> data_t1; //uncomment for fixed support

void my_gemm(int TA, int TB, int M, int N, int K, float alpha,
          const float * A, int lda,
          const float * B, int ldb, float beta,
          float * C, int ldc);

#pragma SDS data access_pattern(A: SEQUENTIAL, B: SEQUENTIAL, C: SEQUENTIAL)
void mmult_accel(data_t1 A[MB * KB], data_t1 B[KB * NB], data_t1 C[MB * NB],
                     data_t1 alpha);

#pragma SDS data access_pattern(A: SEQUENTIAL, B: SEQUENTIAL, C: SEQUENTIAL)
void madd_accel(data_t1 A[MB * NB], data_t1 B[MB * NB], data_t1 C[MB * NB]);
```

Listing 6. *mmult_accel.cpp*

```
#include "gemm_accel.h"
void mmult_accel(data_t1 A[MB * KB], data_t1 B[KB * NB], data_t1 C[MB * NB],
    data_t1 alpha) {

    data_t1 tA[MB][KB], tB[KB][NB];

    #pragma HLS array_partition variable = tA block factor = 16 dim = 2
    #pragma HLS array_partition variable = tB block factor = 16 dim = 1
    #pragma HLS RESOURCE variable = tA core = RAM_2P_BRAM
    #pragma HLS RESOURCE variable = tB core = RAM_2P_BRAM

    for (int i = 0; i < MB; i++) {
        for (int j = 0; j < KB; j++) {#
             #pragma HLS PIPELINE
             tA[i][j] = alpha * A[i * KB + j];
             tB[i][j] = B[i * NB + j];
        }
      }

    for (int i = 0; i < MB; i++) {
        for (int j = 0; j < NB; j++) {
             #pragma HLS PIPELINE
             data_t1 result = 0;
             for (int k = 0; k < KB; k++) {
                  data_t1 term = tA[i][k] * tB[k][j];
                  #pragma HLS RESOURCE variable=term core=FMul_fulldsp
//#pragma HLS RESOURCE variable=term core=HMul_fulldsp       //half float DSP support
                  result += term;
                  #pragma HLS RESOURCE variable=result core=FAddSub_fulldsp
             }
             C[i * NB + j] = result;
        }
    }
}
```

Listing 7. *madd_accel.cpp*

```
#include < stdlib.h >
#include "gemm_accel.h"

void madd_accel(data_t1 A[MB * NB], data_t1 B[MB * NB], data_t1 C[MB * NB]) {
    int i, j;
    for (i = 0; i < MB; i++) {
        for (j = 0; j < NB; j++) {
             #pragma HLS PIPELINE
          C[i * NB + j] = A[i * NB + j] + B[i * NB + j];
        }
    }
}
```

Listing 8. my_gemm.*h*

```

typedef float data_t;

 void my_gemm(int TA, int TB, int M, int N, int K, data_t alpha,
    const data_t * A, int lda,
    const data_t * B, int ldb, data_t beta,
    data_t * C, int ldc);
```

Listing 8. *GEMM replacement in Caffe source code*

```
void caffe_cpu_gemm<float>(const CBLAS_TRANSPOSE TransA, const CBLAS_TRANSPOSE TransB,
                           const int M, const int N, const int K,
                           const float alpha, const float * A,
                           const float * B, const float beta, float * C) {

    int lda = (TransA == CblasNoTrans) ? K : M;
    int ldb = (TransB == CblasNoTrans) ? N : K;
    int TA = (TransA == CblasNoTrans) ? 0 : 1;
    int TB = (TransB == CblasNoTrans) ? 0 : 1;

    if (M >= 32 && N >= 32 && K >= 32 && TA == 0 && TB == 0)
       my_gemm(TA, TB, M, N, K, alpha, A, lda, B, ldb, beta, C, N);  //accelerated
    else
       cblas_sgemm(CblasRowMajor, TransA, TransB, M, N, K, alpha,    //default
                   A, lda, B, ldb, beta, C, N);
}
```

Bibliography

[1] "Convolutional Neural Networks for Visual Recognition," Stanford, [Online]. Available: http://cs231n.github.io/neural-networks-1/.

[2] A. Angel, "Towards Distortion-Predictable Embedding of Neural Networks," *ArXiv,* vol. 1508.00102, 2015.

[3] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro and E. Shelhamer, "cuDNN: Efficient Primitives for Deep Learning," *arXiv:1410.0759.*

[4] H. Y. Badgujar, "Introduction to CUDA 5.0," [Online]. Available: https://hemprasad.wordpress.com/2013/03/03/introduction-to-cuda-5-0/.

[5] R. Birle and L. Bandil, "Design and FPGA Implementation of Systolic Array Architecture for Matrix Multiplication," *International Journal of Engineering and Advanced Technology (IJEAT),* August 2012.

[6] D. Gschwend, "ZynqNet: An FPGA-Accelerated Embedded Convolutional Neural Network," 2016.

[7] S. Behnke, "Hierarchical Neural Networks for Image Interpretation," 2003, pp. 35-63.

[8] Z. Kolter, " Neural Networks and Deep Learning: A Quick Overview," Carnegie Mellon University, [Online]. Available: http://i-systems.github.io/HSE545/machine%20learning%20all/16%20Deep%20learning/Overview_Deep_Learning.html.

[9] "Field Programmable Gate Arrays (FPGA)," University of Miskolc, [Online]. Available: http://mazsola.iit.uni-miskolc.hu/cae/docs/pld1.en.html.

[10] "Logic Block," [Online]. Available: https://en.wikipedia.org/wiki/Logic_block.

[11] "Field-programmable gate array," [Online]. Available: https://en.wikipedia.org/wiki/Field-programmable_gate_array.

[12] "GPU vs FPGA Performance Comparison," [Online]. Available: http://www.bertendsp.com/company/blog/.

[13] H. Kaeslin, "Digital Integrated Circuit Design: From VLSI Architectures to CMOS Fabrication," Cambridge University Press, 2008, pp. (cit. on pp. 13, 14)..

[14] "Heterogeneous computing," [Online]. Available: https://en.wikipedia.org/wiki/Heterogeneous_computing.

[15] SDSoC, [Online]. Available: https://www.xilinx.com/products/design-tools/software-zone/sdsoc.html.

[16] "Cloud computing," Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Cloud_computing.

[17] "Amazon web service," Amazon, [Online]. Available: https://aws.amazon.com/.

[18] "Amazon EC2 F1 Instances," Amazon, [Online]. Available: https://aws.amazon.com/ec2/instance-types/f1/.

[19] U. Farooq, Z. Marrakchi and H. Mehrez, Tree-based Heterogeneous FPGA Architectures, Springer.

[20] L. H. Crockett, R. A. Elliot, M. A. Enderwitz and R. W. Stewart, The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc, Academic Media, 2014.

[21] Caffe. [Online]. Available: http://caffe.berkeleyvision.org.

[22] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama and T. Darrell, "Caffe: Convolutional Architecture for Fast Embedding".

[23] "Caffe to Zynq: State-of-the-Art Machine Learning Inference," Xilinx, [Online]. Available: https://www.youtube.com/watch?v=kgA2fpK3xpo.

[24] "Gflags," Google, [Online]. Available: https://github.com/gflags/gflags.

[25] "OpenBlas," Texas, University of; Austin, [Online]. Available: https://github.com/xianyi/OpenBLAS.

[26] Wikipedia, "Application binary interface," [Online]. Available: https://en.wikipedia.org/wiki/Application_binary_interface.

[27] "CMake," [Online]. Available: https://cmake.org/.

[28] "Glog," Google, [Online]. Available: //github.com/google/glog.

[29] "Boost," [Online]. Available: https://github.com/boostorg/boost.

[30] "Szip," [Online]. Available: https://github.com/erdc/szip.

[31] "LMDB," [Online]. Available: https://github.com/LMDB/lmdb.

[32] M. Adler, "Zlib," [Online]. Available: https://github.com/madler/zlib.

[33] "Protobuf," Google, [Online]. Available: https://github.com/google/protobuf.

[34] "Snappy," Google, [Online]. Available: https://github.com/google/snappy.

[35] "LevelDB," [Online]. Available: https://github.com/google/leveldb.

[36] "OpenCV," [Online]. Available: https://github.com/opencv/opencv.

[37] HDF5, NCSA , [Online]. Available: https://support.hdfgroup.org/HDF5/.

[38] T. G. Lane and G. Vollbeding., "JPEG library," [Online]. Available: https://github.com/LuaDist/libjpeg.

[39] "Rpath," Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Rpath.

[40] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally and K. Keutzer, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size," *arXiv:1602.07360.*

[41] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke and A. Rabinovich, "Going Deeper with Convolutions," *arXiv:1409.4842.*

[42] L. Jiang, "Finding rootfs during boot," 11 March 2009. [Online]. Available: https://www.ibm.com/developerworks/linux/library/l-boot-rootfs/.

[43] K. He and J. Sun, " Convolutional Neural Networks at Constrained Time Cost," *arXiv:1412.1710.*

[44] "Neural network visualizer," MIT, [Online]. Available: https://github.com/ethereon/netscope.

[45] P. Warden, "Why GEMM is at the heart of deep learning," [Online]. Available: https://petewarden.com/2015/04/20/why-gemm-is-at-the-heart-of-deep-learning/.

[46] V. Strassen, "Gaussian elimination is not optimal," 1969, pp. 354-356.

[47] C. R. Wan and D. J. Evans, "Nineteen ways of systolic matrix multiplication," 1998 , pp. vol. 68, no. 1-2, pp. 39–69.

[48] "SDSoC Optimization Guide," Xilinx, [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug1235-sdsoc-optimization-guide.pdf.

[49] "SDSoC User Guide," Xilinx, [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_1/ug1027-sdsoc-user-guide.pdf.

[50] "Granularity (parallel computing)," Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Granularity_(parallel_computing).

[51] "TeraTerm," Ayera Technologies, Inc, [Online]. Available: http://www.ayera.com/teraterm/.