Εθνικο Μετσοβιο Πολυτεχνειο

Σχολη Ηλεκτρολογων Μηχανικων και Μηχανικων Υπολογιστων

τομεας τεχνολογιας πληροφορικης και υπολογιστων

Εργαστηριο Μικροϋπολογιστων και Ψηφιακων Συστηματων

Microarchitectural Extension of CGRA Accelerator for Efficient LLM Code Mapping

Μικροαρχιτεκτονική Επέκταση Επιταχυντή Τύπου CGRA για Αποδοτική Απεικόνιση Εφαρμογών Τύπου LLM

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Διονυσίου Κεφαλληνού

Επιβλέπων: Σωτήριος Ξύδης
Επίκουρος Καθηγητής Ε.Μ.Π.

Αθήνα, Μάρτιος 2025

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

Microarchitectural Extension of CGRA Accelerator for Efficient LLM Code Mapping

Μικροαρχιτεκτονική Επέκταση Επιταχυντή Τύπου CGRA για Αποδοτική Απεικόνιση Εφαρμογών Τύπου LLM

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Διονυσίου Κεφαλληνού

Επιβλέπων: Σωτήριος Ξύδης
Επίκουρος Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 14η Μαρτίου, 2025.

.....	.....	.....
Σωτήριος Ξύδης	Δημήτριος Σούντρης	Γεώργιος Ζερβάκης
Επίκουρος Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Επίκουρος Καθηγητής Παν. Πατρών

Αθήνα, Μάρτιος 2025

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

στην οικογένεια μου

Περίληψη

Τα τελευταία χρόνια οι υπολογιστικές απαιτήσεις των Μεγάλων Γλωσσικών Μοντέλων (LLMs) ολοένα και αυξάνονται, καθώς το πεδίο εφαρμογών τους διευρύνεται και το πλήθος των παραμέτρων τους συνεχώς κλιμακώνεται. Η νεότερη ερευνητική τάση είναι η μετατόπιση του υπολογιστικού φόρτου για το inference όλο και πιο κοντά στον χρήστη, με τις edge συσκευές (ή agents). Στην δουλειά αυτή εξετάζουμε έναν συγκεκριμένο επιταχυντή τύπου CGRA, τον R-Blocks, ως πιθανή πλατφόρμα εκτέλεσης τέτοιων εφαρμογών. Αφενός επεκτείνουμε την μικροαρχιτεκτονική και τα εργαλεία του περιβάλλοντος μεταγλώττισης (OpenASIP) του R-Blocks για την υποστήριξη αριθμητικής κινητής υποδιαστολής, και αφετέρου απεικονίζουμε τα πρώτα πειραματικά benchmarks τύπου LLM στο επαναπρογραμματίσιμο υλικό, εξερευνώντας διαφορετικές αρχιτεκτονικές και παραμέτρους παραλληλοποίησης. Η τελική μας αξιολόγηση γίνεται σε ASIC τεχνολογία 22nm FD-SOI, και εξάγονται συμπεράσματα για την βιωσιμότητα της προσέγγισης μας ως προς την απόδοση, την ενέργεια και το εμβαδόν του κυκλώματος.

Λέξεις Κλειδιά — CGRA, μεγάλα γλωσσικά μοντέλα (LLM), μικροαρχιτεκτονική επέκταση, επαναπρογραμματίσιμο υλικό, υπολογισμοί κινητής υποδιαστολής, εξερεύνηση χώρου σχεδιασμού αρχιτεκτονικών

Abstract

In recent years, the computational demands of Large Language Models (LLMs) have been steadily increasing, driven by their expanding range of applications and the scaling of their parameter sizes. A key emerging trend is the shift of inference workloads closer to the user, leveraging edge devices and specialized agents. In this work, we explore the R-Blocks CGRA accelerator as a potential platform for running such workloads efficiently. Our contributions are twofold: first, we extend the microarchitecture and compilation toolchain (OpenASIP) of R-Blocks to support floating-point arithmetic, necessary for efficient LLM inference; second, we implement and benchmark LLM workloads on the reconfigurable hardware, investigating various architectural choices and parallelization strategies. Finally, we evaluate our design in a 22nm FD-SOI ASIC implementation, providing insights into its performance, energy efficiency, and area footprint, and assessing the viability of our approach for edge-based LLM inference.

Keywords — large language models (LLM), coarse-grained reconfigurable architectures (CGRA), edge computing, inference acceleration, floating-point arithmetic, reconfigurable hardware, architectural design space exploration

Ευχαριστίες

Για την καθοδήγηση και την επίβλεψη τους ευχαριστώ τους καθηγητές μου κύριο Δ. Σούντρη και Σ. Ξύδη. Για την βοήθεια τους ευχαριστώ του υποψήφιους διδάκτορες, μέλη της ομάδας του Convolve και μέλη του Microlab Α. Μάρα, Π. Χάιδο, Γ. Αναγνωστόπουλο, και ιδιαίτερα τον Γ. Αλεξανδρή για την ανεκτίμητη συμβολή τους που δεν μπορεί να επεξηγηθεί στα περιορισμένα όρια της μίας σελίδας. Για την συνεχή και ασταμάτητη υποστήριξή τους τα τελευταία 23 χρόνια αλλά ιδιαίτερα τους τελευταίους 6 μήνες ευχαριστώ τους γονείς μου, στους οποίους οφείλω τα πάντα.

Κεφαλληνός Διονύσιος
Μάρτιος 2025

"Drink this water of the spring, rest here awhile, we have a long way yet to go and I can't go without you."
— Ursula K. Le Guin, Always Coming Home

Contents

Figure List

Table List

Chapter 1

Εκτεταμένη Ελληνική Περίληψη

Στο πρώτο κεφάλαιο αυτής της διπλωματικής θα παρουσιάσουμε όλη μας τη δουλειά άμεσα και περιεκτικά για τον έλληνα αναγνώστη. Για περισσότερες λεπτομέρειες και τεχνικά ζητήματα αναφερθείτε στο αντίστοιχο αγγλικό κεφάλαιο του κειμένου, αν και οι πληροφορίες που θα παρουσιαστούν εδώ είναι επαρκείς για την πλήρη κατανόηση της εν λόγω δουλειάς.

1.1 Εισαγωγή

Η επανάσταση στον χώρο της τεχνητής νοημοσύνης τα τελευταία χρόνια οφείλεται σε μεγάλο βαθμό στην εισαγωγή των Μετασχηματιστών (Transformers), μοντέλων που βασίζονται στον μηχανισμό προσοχής (Attention). Από τη δημοσίευση του άρθρου "Attention Is All You Need" [1], τα μοντέλα αυτά κυριαρχούν στη δημιουργία φυσικής γλώσσας, με πρώτα γνωστά παραδείγματα τα GPT-3 [2] και LLaMA [3]. Τα σύγχρονα Μεγάλα Γλωσσικά Μοντέλα (Large Language Models, LLMs) δεν παράγουν απλώς κατανοητό κείμενο, αλλά μπορούν να απαντήσουν με ακρίβεια και δημιουργικότητα σε ερωτήσεις, ενώ χάρη σε τεχνικές όπως η ενισχυτική μάθηση (reinforcement learning) [4], αρχίζουν να επιδεικνύουν ακόμα και ικανότητες βαθιάς συλλογιστικής.

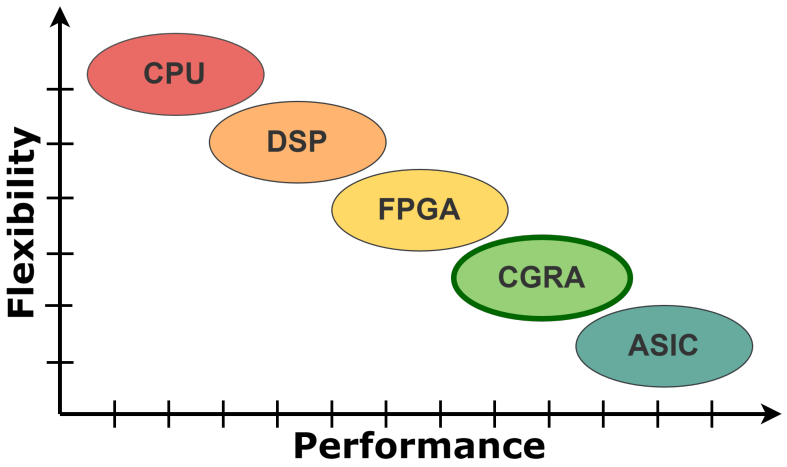


Figure 1.1.1: Συμβιβασμός μεταξύ ευελιξίας και απόδοσης.

Η εφαρμογή των Μετασχηματιστών δεν περιορίζεται στη φυσική γλώσσα, αλλά επηρεάζει πλήθος επιστημονικών πεδίων. Στην Υπολογιστική Όραση [5] και την Αναγνώριση Ομιλίας [6] έχουν ήδη φέρει σημαντικές βελτιώσεις, ενώ στη Βιολογία έχουν αλλάξει ριζικά τον τρόπο με τον οποίο μελετούμε τις πρωτεΐνες. Το μοντέλο AlphaFold [7], για παράδειγμα, χρησιμοποιεί μηχανισμούς προσοχής για να προβλέψει με

ακρίβεια τις τρισδιάστατες δομές πρωτεϊνών γνωρίζοντας μόνο την αλληλουχία των αμινοξέων τους, κάτι που έχει τεράστιες επιπτώσεις στη φαρμακευτική έρευνα και τη μοριακή επιστήμη.

Παρά τα οφέλη, η χρήση των LLMs συνοδεύεται από τεράστιες απαιτήσεις σε υπολογιστική ισχύ. Η εκπαίδευση και η εκτέλεσή τους βασίζονται σχεδόν αποκλειστικά σε ισχυρές συστοιχίες GPU, καθώς προσφέρουν τη βέλτιστη ισορροπία μεταξύ απόδοσης και προγραμματιστικής ευκολίας. Ωστόσο, οι ανάγκες για αποδοτικότερη εκτέλεση οδηγούν στην αναζήτηση εναλλακτικών αρχιτεκτονικών υλικού.

Μία από αυτές είναι οι Προγραμματιζόμενες Αρχιτεκτονικές Μικρού Επιπέδου Λεπτομέρειας (Coarse-Grained Reconfigurable Architectures, CGRAs) [8], που βρίσκονται ανάμεσα στις Συστοιχίες Επιτόπια Προγραμματιζόμενων Πυλών (Field-Programmable Gate Arrays, FPGAs) και τα Ολοκληρωμένα Κυκλώματα Ειδικού Σκοπού (Application-Specific Integrated Circuits, ASICs). Τα CGRAs προσφέρουν υψηλότερη απόδοση από τα FPGAs καθώς χρησιμοποιούν προσχεδιασμένα λειτουργικά στοιχεία που μπορούν να συνδεθούν δυναμικά με μικρότερο overhead, ενώ παραμένουν πιο ευέλικτα από τα ASICs, λόγω της επαναπρογραμματισιμότητάς τους (εικόνα 1.1.1).

Σε αυτό το πλαίσιο, το R-Blocks [9] αποτελεί μια πολλά υποσχόμενη πλατφόρμα CGRA. Πρόκειται για μια ενεργειακά αποδοτική αρχιτεκτονική που χρησιμοποιεί το μοντέλο εκτέλεσης VLIW-SIMD και αξιοποιεί τα εργαλεία της OpenASIP [10] που στοχεύουν στο μοντέλο αρχιτεκτονικών καθοδηγούμενων από τις μεταφορές δεδομένων (Transport Triggered Architectures, TTA). Ένα βασικό πλεονέκτημά του είναι η δυνατότητα να απεικονίζει υψηλού επιπέδου κώδικα απευθείας στο επαναπρογραμματίσιμο υλικό με τον υψηλού επιπέδου compiler της OpenASIP.

Στόχος της παρούσας διπλωματικής είναι η διερεύνηση της χρήσης του R-Blocks ως επιταχυντή για την εκτέλεση LLMs σε συσκευές αιχμής (edge devices). Ένα από τα μεγαλύτερα εμπόδια είναι η απουσία υποστήριξης αριθμητικής κινητής υποδιαστολής (floating point arithmetic), καθώς το R-Blocks, όπως και η πλατφόρμα Blocks [11] στην οποία βασίζεται, έχει σχεδιαστεί για απλές αριθμητικές πράξεις με ελάχιστη κατανάλωση ενέργειας.

Για την αντιμετώπιση αυτού του περιορισμού, σχεδιάσαμε και ενσωματώσαμε μια νέα Λειτουργική Μονάδα (Functional Unit, FU) που περιλαμβάνει μια ανοιχτού κώδικα Μονάδα Κινητής Υποδιαστολής (Floating Point Unit, FPU), συνοδευόμενη από το δικό της προσαρμοσμένο σύνολο εντολών. Με την προσθήκη υποστήριξης για τις νέες εντολές και στα εργαλεία μεταγλώττισης του R-Blocks, καθίσταται δυνατή η εκτέλεση όλων των δυνατών αριθμητικών πράξεων κινητής υποδιαστολής χρησιμοποιώντας βιβλιοθήκες υψηλού επιπέδου (π.χ. math.h).

Αυτή η επέκταση επέτρεψε, για πρώτη φορά, την απεικόνιση της διαδικασίας παραγωγής συμβόλων (token generation) ενός LLM στο R-Blocks. Επιπλέον, διερευνήθηκαν τεχνικές παραλληλισμού, όπως η διανυσματοποίηση/παραλληλοποίηση (vectorization), που επιτρέπει την εκτέλεση πολλαπλών αριθμητικών πράξεων ταυτόχρονα, αξιοποιώντας πολλές Λειτουγικές Μονάδες FPU συνδεδεμένες στον ίδιο αποκωδικοποιητή εντολών. Μέσα από αυτή τη διαδικασία, εξετάσαμε πώς διαφορετικές αρχιτεκτονικές επιδρούν στην απόδοση και ποιοι είναι οι περιορισμοί που προκύπτουν από την ίδια τη δομή του R-Blocks.

Οι προτεινόμενες αρχιτεκτονικές συντέθηκαν σε τεχνολογία 22nm FD-SOI χρησιμοποιώντας το Synopsys Design Compiler, με μέγιστη συχνότητα λειτουργίας 200MHz. Με αυτόν τον τρόπο συλλέξαμε δεδομένα σχετικά με την κατανάλωση ισχύος και την απόδοση των κυκλωμάτων.

Τα αποτελέσματα δείχνουν ότι το R-Blocks έχει σημαντικές δυνατότητες ως ενεργειακά αποδοτικός επιταχυντής LLMs. Ωστόσο, υπάρχουν περιορισμοί που αφορούν τη δυνατότητα κλιμάκωσης (scalability) και την πλήρη αξιοποίηση των δυνατοτήτων της αρχιτεκτονικής CGRA. Αναλυτικά αποτελέσματα παρατίθενται στα παρακάτω υποκεφάλαια.

Πριν από αυτά, όμως, είναι απαραίτητη μια εισαγωγή στις βασικές έννοιες των CGRAs και των Μεγάλων Γλωσσικών Μοντέλων, προκειμένου να γίνει καλύτερα κατανοητό το αντικείμενο της έρευνας.

1.2 Θεωρητικό Υπόβαθρο

1.2.1 Προγραμματιζόμενες Αρχιτεκτονικές Μικρού Επιπέδου Λεπτομέρειας (CGRAs)

Οι Προγραμματιζόμενες Αρχιτεκτονικές Μικρού Επιπέδου Λεπτομέρειας (Coarse-Grained Reconfigurable Arrays, CGRAs), όπως είπαμε, αποτελούν μια ενδιάμεση λύση μεταξύ της ευελιξίας των FPGAs και της απόδοσης των ASICs. Αντί να λειτουργούν σε επίπεδο μεμονωμένων λογικών πυλών, όπως τα FPGAs, τα CGRAs χρησιμοποιούν μεγαλύτερες λειτουργικές μονάδες (Functional Units, FUs), μειώνοντας το κόστος επαναδιαμόρφωσης και βελτιώνοντας την ενεργειακή αποδοτικότητα.

Σύμφωνα με το πίο ενδελεχές survey για τα σύγχρονα CGRAs [12], οι αρχιτεκτονικές αυτές βασίζονται σε τρεις βασικές αρχές:

- Ευελιξία σε συγκεκριμένες εφαρμογές, προσφέροντας επαναδιαμόρφωση μόνο όπου απαιτείται.
- Συνδυασμό χωρικής και χρονικής επεξεργασίας, αξιοποιώντας παραλληλισμό σε επίπεδο εντολών (VLIW) ή δεδομένων (SIMD).
- Μοντέλο εκτέλεσης που βασίζεται είτε στη συγκεκριμένη διαμόρφωση, είτε στη ροή δεδομένων, αντί για αυστηρά σειριακή εκτέλεση, όπως γίνεται στους επεξεργαστές γενικού σκοπού.

Η γενική μορφή μιας αρχιτεκτονικής τύπου CGRA φαίνεται στην εικόνα 1.2.1. Αυτή αποτελείται από τις Λειτουργικές Μονάδες ή Processing Elements (PEs) σε δομή πλέγματος διασυνδεδεμένες από καλώδια σε σχηματισμό πλέγματος (όπως εδώ) ή δικτύου (όπως στο R-Blocks που θα ακολουθήσει παρακάτω). Οι λειτουργικές μονάδες είναι ομογενή κομμάτια υλικού που επιτελούν συγκεκριμένες λειτουργίες και είναι προγραμματίσιμα ή όχι, ανάλογα την αρχιτεκτονική του εν λόγω CGRA. Το επαναπρογραμματίσιμο υλικό (στην εικόνα *PE array*) είναι συνδεδεμένο με εξωτερικές μνήμες εντολών, κύρια μνήμη και έναν κύριο επεξεργαστή για τον οποίο συνήθως λειτουργεί ως επιταχυντής.

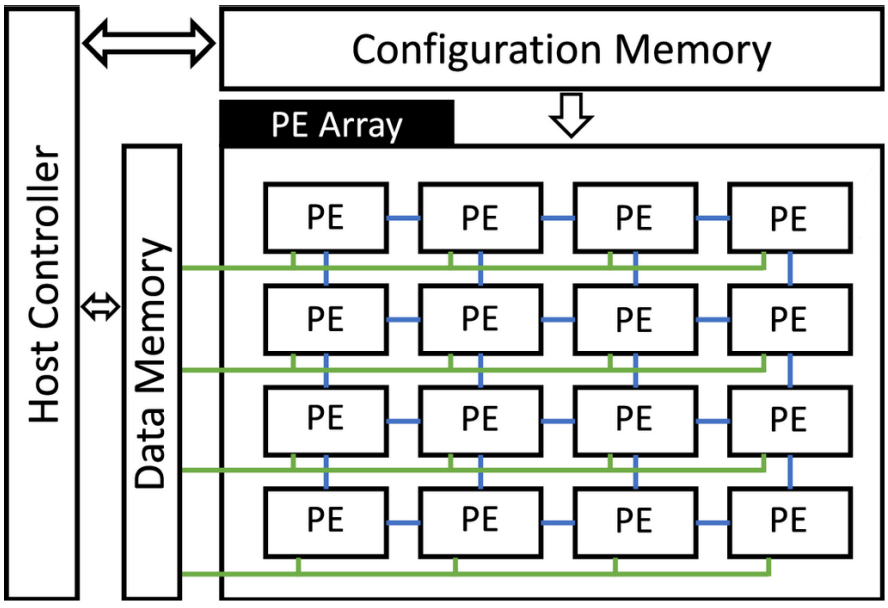


Figure 1.2.1: Η γενική μορφή μιας αρχιτεκτονικής CGRA

Ανάλογα με τον τρόπο που υλοποιείται η απεικόνιση του προγράμματος στο υλικό, τα CGRAs ταξινομούνται ως:

- SCSD (Single Configuration, Single Data) – Κάθε διαμόρφωση εκτελεί πράξεις με ένα συγκεκριμένο σύνολο δεδομένων.

- SCMD (Single Configuration, Multiple Data) – Στην ίδια διαμόρφωση εφαρμόζονται ταυτόχρονα σε πολλαπλά σύνολα δεδομένων, όπως στις SIMD αρχιτεκτονικές.
- MCMD (Multiple Configuration, Multiple Data) – Υποστηρίζει ταυτόχρονη εκτέλεση ξεχωριστών συνόλων δεδομένων σε διαφορετικές διαμορφώσεις, κατάλληλο για εφαρμογές που απαιτούν πολυνηματική εκτέλεση.

Ως διαμόρφωση εδώ εννοείται η συγκεκριμένη αρχιτεκτονική που έχει ορίσει ο προγραμματιστής, η οποία περιγράφει την εσωτερική διάταξη του CGRA, την δομή του και τις διασυνδέσεις του, και πάνω στην οποία απεικονίζονται και εκτελούνται οι εφαρμογές.

Σε επίπεδο εκτέλεσης, τα CGRAs μπορεί να ακολουθούν στατικό ή δυναμικό χρονοπρογραμματισμό, καθώς και σειριακή επεξεργασία ή επεξεργασία βασισμένη στη ροή δεδομένων (dataflow). Η επιλογή της κατάλληλης μικροαρχιτεκτονικής εξαρτάται από παράγοντες όπως τα χαρακτηριστικά ροής δεδομένων, η τοπολογία της διασύνδεσης και η ιεραρχία μνήμης.

Η ανάλυση ενός πραγματικού συστήματος CGRA θα μας βοηθήσει να κατανοήσουμε πώς αυτές οι αρχές εφαρμόζονται στην πράξη και πώς επηρεάζουν την απόδοση, την κλιμάκωση και την αποδοτικότητα της αρχιτεκτονικής.

1.2.2 R-Blocks

Το CGRA το οποίο θα μελετήσουμε, θα επεκτείνουμε και στο οποίο θα απεικονίσουμε εφαρμογές τύπου LLM ονομάζεται R-Blocks [9]. Δημιουργήθηκε από το Τεχνικό Πανεπιστήμιο του Αϊτχόφεν, και χρησιμοποιεί την σουίτα εργαλείων OpenASIP [10] του Πανεπιστημίου του Τάμπερε της Φινλανδίας.

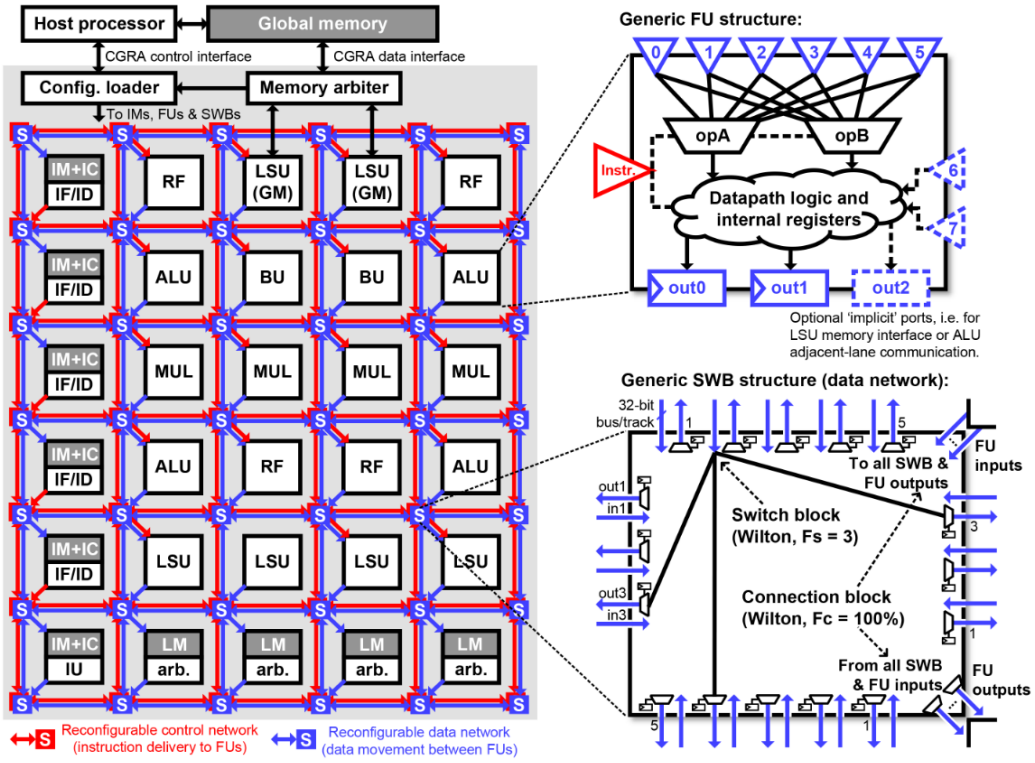


Figure 1.2.2: Η εσωτερική οργάνωση ενός πρότυπου συστήματος R-Blocks

Πρόκειται για ένα CGRA εξαιρετικά χαμηλής κατανάλωσης που ακολουθεί το μοντέλο εκτέλεσης εντολών VLIW-SIMD και υποστηρίζεται από ένα όλοκληρο περιβάλλον ταυτόχρονου σχεδιασμού υλικού και λογισμικού. Αυτό σημαίνει ότι ο προγραμματιστής έχει την δυνατότητα να καθορίσει τόσο το πρόγραμμα

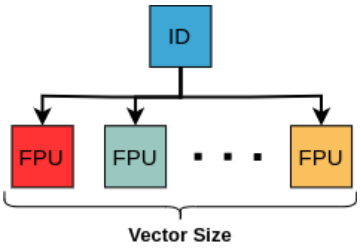


Figure 1.2.3: Συνδεσμολογία λειτουργικών μονάδων για παράλληλη επεξεργασία

που θα εκτελεστεί στην πλατφόρμα R-Blocks όσο και την αρχιτεκτονική στην οποία αυτό θα απεικονιστεί, καθώς και όλες τις λεπτομέρειες μεταξύ των δύο επιπέδων.

Από άποψη αρχιτεκτονικής το R-Blocks αποτελείται από ετερογενείς λειτουργικές μονάδες (Functional Units, FUs), σε αντίθεση με τα περισσότερα μοντέλα που έχουν προταθεί. Αυτά τα δομικά στοιχεία τοποθετούνται σε διάταξη πλέγματος, όπως φαίνεται στην εικόνα 1.2.2, και διασυνδέονται με δύο διαφορετικά δίκτυα καλωδίων σε σχηματισμό Network on Chip, χρησιμοποιώντας κουτιά διακοπτών (switchboxes). Το ένα δίκτυο μεταφέρει αποκλειστικά εντολές ενώ το άλλο τα δεδομένα μεταξύ των υπολογιστικών λειτουργικών μονάδων.

Το επαναπρογραμματίσιμο υλικό λειτουργεί ως επιταχυντής στον εξωτερικό επεξεργαστή, και συνδέεται μαζί του μέσω της κύριας μνήμης όπως φαίνεται και στην εικόνα 1.2.2.

Οι λειτουργικές μονάδες από τις οποίες αποτελείται το R-Blocks ανήκουν στους εξής διαφορετικούς τύπους:

- Αποκωδικοποιητής Εντολών (Instruction Decoder, ID)
- Μονάδα Άμεσων Τιμών (Immediate Unit, IU) για την άμεση δημιουργία τελεστών
- Μονάδα Φόρτωσης-Αποθήκευσης (Load-Store Unit, LSU) για τη διαχείριση της επικοινωνίας με την κύρια μνήμη
- Αρχείο Καταχωρητών (Register File, RF) για την αποθήκευση τελεστών
- Αριθμητική και Λογική Μονάδα (Arithmetic and Logical Unit, ALU)
- Μονάδα Συσσώρευσης και Διακλάδωσης (Accumulate and Branch Unit, ABU)
- Πολλαπλασιαστής (Multiplier, MUL)
- Μονάδα Δημιουργίας Διευθύνσεων (Address Generation Unit, AGU)
- Τοπικές Μνήμες (Local Memories, LM), οι οποίες αποθηκεύουν τους τελεστές για διανυσματικές πράξεις κοντά στις αντίστοιχες λειτουργικές μονάδες, ώστε να μπορούν να χρησιμοποιηθούν ταυτόχρονα.

Κάθε μία από αυτές τις λειτουργικές μονάδες έχει το δικό της σύνολο εντολών (εξού και η ετερογένεια τους) και χρησιμοποιείται για την απεικόνιση μέρους του προγράμματος. Στην ουσία όλη η δομή του R-Blocks θυμίζει αποσυναρμολογημένο επεξεργαστή, αλλά με την δυνατότητα να συμπεριλάβει κανείς περισσότερα λειτουργικά στοιχεία και σε οποιαδήποτε διάταξη επιθυμεί.

Η κάθε λειτουργική μονάδα (εκτός της IU) συνδέεται σε έναν ID για να λαμβάνει τις εντολές που πρέπει, κατα την εκτέλεση του προγράμματος. Υποστηρίζεται επίσης η σύνδεση πολλών ομοίων λειτουργικών μονάδων στον ίδιο ID έτσι ώστε να λειτουργούν ως μία SIMD μονάδα επεξεργασίας, όπως φαίνεται στο σχήμα 1.5.1.

Η μονάδα FPU που απεικονίζεται αποτελεί την δική μας συνεισφορά. Στην πρότερη κατάστασή του το R-Blocks δεν είχε την δυνατότητα επεξεργασίας αριθμών κινητής υποδιαστολής, και όπως θα αναλυθεί

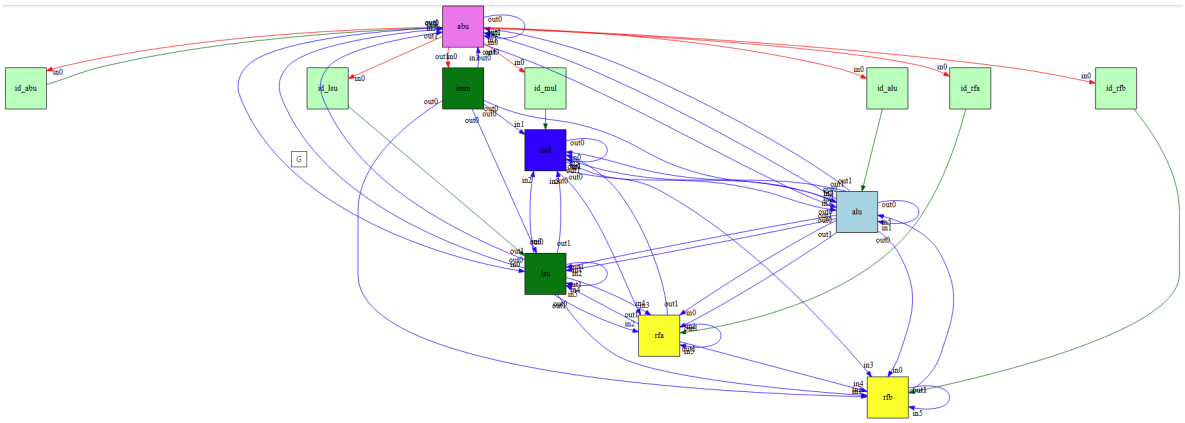


Figure 1.2.4: Αρχιτεκτονική Περιγραφή R-Blocks

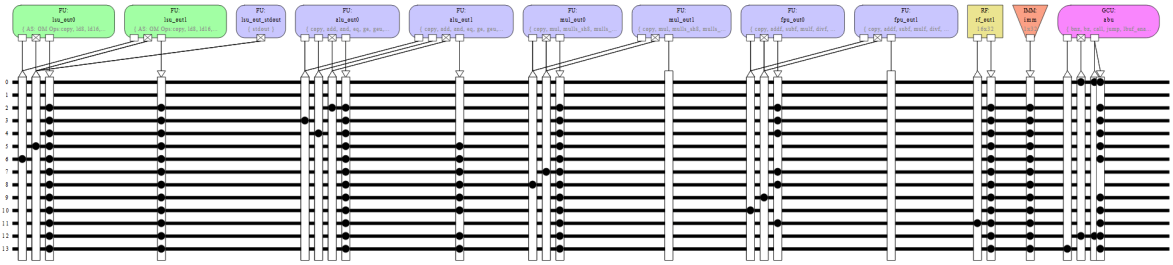


Figure 1.2.5: Αρχιτεκτονική Περιγραφή TTA από το GUI του εργαλείου ProDe της OpenASIP

παρακάτω, το επεκτείναμε με μια νέα λειτουργική μονάδα επεξεργασίας αριθμών κινητής υποδιαστολής (Floating Point Unit, FPU), ώστε να μπορέσουμε να εκτελέσουμε εφαρμογές LLM σε αυτό.

Το μοντέλο εκτέλεσης εντολών του R-Blocks βασίζεται στις αρχιτεκτονικές καθοδηγούμενες από τις μεταφορές δεδομένων (Transport Triggerred Architectures) [13]. Στο αρχιτεκτονικό αυτό μοντέλο το πρόγραμμα έχει τον έλεγχο των μεταφορών δεδομένων στους διαύλους και οι υπολογισμοί συμβαίνουν εξαιτίας αυτών των μεταφορών αντί να συμβαίνει το ανάποδο. Μια TTA αποτελείται από FUs, καταχωρητές και διαύλους δεδομένων. Είναι έτσι δυνατόν να απεικονίσουμε με απόλυτη ακρίβεια μια αρχιτεκτονική περιγραφή μιας οποιασδήποτε διάταξης R-Blocks ως TTA.

Η λειτουργία αυτή επιτελείται από το εργαλείο Blocks Translator που μεταφράζει μια αρχιτεκτονική R-Blocks που έχει περιγράψει ο προγραμματιστής (εικόνα 1.2.4) στην αντίστοιχη περιγραφή TTA (εικόνα 1.2.5).

Όλη αυτή η διαδικασία γίνεται για να μπορέσει να χρησιμοποιηθεί ο μεταγλωττιστής της σουίτας OpenASIP (που στοχεύει αρχιτεκτονικές TTA) για την απεικόνιση κώδικα υψηλού επιπέδου σε αρχιτεκτονικές R-Blocks. Ο μεταγλωττιστής αυτός λαμβάνει ως είσοδο την περιγραφή της αρχιτεκτονικής σε μορφή TTA και τον κώδικα C της εφαρμογής, και παράγει ένα εκτελέσιμο για την συγκεκριμένη αρχιτεκτονική (εικόνα 1.2.6).

Το εκτελέσιμο αυτό μπορεί να χρησιμοποιηθεί για την παραγωγή parallel assembly η οποία τελικά εκτελείται στο πραγματικό υλικό.

Η αρχιτεκτονική που ορίζει ο προγραμματιστής αρχικά, συνήθως διασυνδέει όλες τις λειτουργικές μονάδες μεταξύ τους για να είναι βέβαιο ότι θα μπορεί να δρομολογηθεί κάθε πρόγραμμα στο υλικό. Στην πραγματικότητα, όμως, δεν απαιτούνται όλες αυτές οι διασυνδέσεις αλλά μόνο ένα μικρό ποσοστό τους. Επιπλέον, το πραγματικό υλικό έχει περιορισμούς που δεν αφήνουν πολύ μεγάλο πλήθος διασυνδέσεων να γίνει πραγματικότητα στο τελικό design. Για αυτούς τους λόγους, υπάρχει το εργαλείο Prunner το οποίο κάνοντας χρήση του εκτελέσιμου που παρήχθη για μια αρχιτεκτονική, "κουρεύει" τις διασυνδέσεις

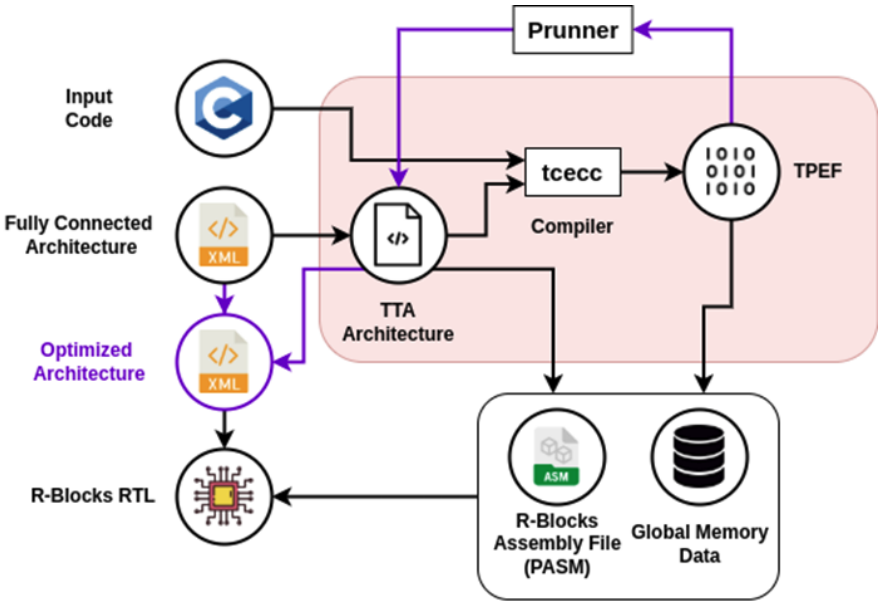


Figure 1.2.6: Η διαδικασία μεταγλώττισης στο περιβάλλον R-Blocks

μεταξύ λειτουργικών μονάδων που δεν είναι απαραίτητες ώστε να δημιουργήσει ένα πιο ρεαλιστικό και μικρό μοντέλο υλικού που μπορεί να περάσει στην φάση της σύνθεσης.

Το τελικό εκτελέσιμο που παράγεται από την νέα αυτή αρχιτεκτονική και η ίδια η περιγραφή της, δημιουργούν το τελικό bitstream που μπορεί να χρησιμοποιηθεί για προσομοιώσεις και εξαγωγή αποτελεσμάτων έκτασης και ισχύος του τελικού κυκλώματος.

Πολλά εργαλεία που συμμετέχουν σε αυτή τη διαδικασία χρειάστηκε να επεκταθούν για να υποστηρίζουν τις νέες δυνατότητες επεξεργασίας αριθμών κινητής υποδιαστολής του CGRA.

1.2.3 Μεγάλα Γλωσσικά Μοντέλα

Τα Μεγάλα Γλωσσικά Μοντέλα (LLM) βασίζονται στην λειτουργία του Μετασχηματιστή (Transformer) [1], ενσωματώνοντας, όμως, το κάθε ένα τις δικές του σχεδιαστικές αποφάσεις και καινοτομίες.

Η βασική λειτουργία ενός LLM είναι η πρόβλεψη της λέξης που θα ακολουθήσει με βάση μια ακολουθία λέξεων που δίνεται ως είσοδος. Με την αναδρομική χρήση των παραγόμενων λέξεων ως νέες είσοδοι, το LLM μπορεί να παράξει όλοκληρες προτάσεις, παραγράφους και κείμενα, βασισμένο σε ένα μικρό ερώτημα του χρήστη.

Πρώτο βήμα στην επίτευξη αυτού του σκοπού είναι η κωδικοποίηση των λέξεων (ή υπο-λέξεων, tokens) της ακολουθίας εισόδου σε διανύσματα, η οποία γίνεται από τον tokenizer. Στη συνέχεια τα διανύσματα αυτά περνάνε από συστάδες υπολογιστικών στρωμάτων στις οποίες συμμετέχουν τα βάρη ή παράμετροι που έχει εσωτερικεύσει το μοντέλο κατά την εκπαίδευσή του, και κωδικοποιούν ολόκληρη την κατεχόμενη γνώση του.

Μετά από τους υπολογισμούς αυτούς, προκύπτει ένας πίνακας πιθανοτήτων για το ποιές λέξεις (ή tokens) είναι πιο λογικό να ακολουθήσουν στην πρόταση, από τις οποίες επιλέγεται η έξοδος του LLM.

Το συγκεκριμένο μοντέλο που θα χρησιμοποιήσουμε σε αυτή τη δουλειά είναι το LLaMa2 [14], το οποίο ήταν από τα μεγαλύτερα και πιο ικανά όταν δημοσιεύθηκε, το 2023. Η εσωτερική του δομή όπως παρουσιάζεται στην εικόνα 1.2.7, θα μας βοηθήσει να καταλάβουμε καλύτερα την λειτουργία των LLM και ταυτόχρονα να κατανοήσουμε πως κώδικας τέτοιου τύπου μπορεί να εκτελεστεί σε αρχιτεκτονική CGRA.

Το πρώτο βήμα της κωδικοποίησης ακολουθείται από το κύριο μέρος "forward", το οποίο εσωτερικά αποτελείται από μια σειρά από ίδια υπολογιστικά επίπεδα (layers). Κάθε επίπεδο, για ευκολότερη

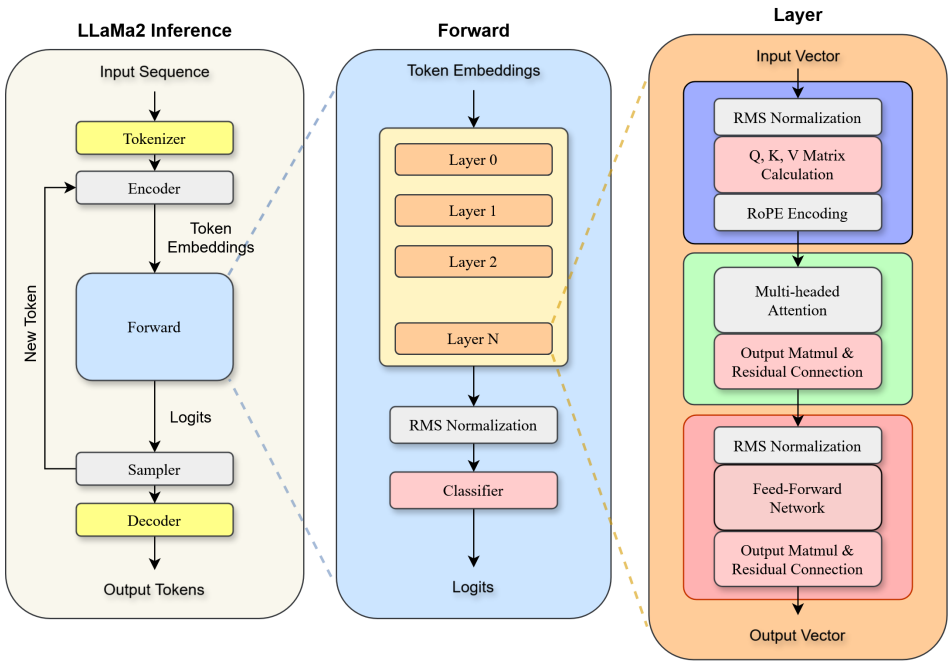


Figure 1.2.7: Αρχιτεκτονική του μοντέλου LLaMa2

κατανόηση το έχουμε χωρίσει σε 3 κομμάτια. Το πρώτο (μωβ) κομμάτι υπολογίζει τους πίνακες Query, Key και Value που χρειάζονται στον μηχανισμό προσοχής (attention) που εικονίζεται στο πράσινο κουτί. Αυτός ο μηχανισμός αναγνωρίζει τις συσχετίσεις μεταξύ των λέξεων της πρότασης και μεγεθύνει ή συρρικνώνει τις κωδικοποιήσεις της σημασίας τους στον διανυσματικό χώρο ανάλογα με το ποια λέξη αναφέρεται σε ποια άλλη μέσα στην πρόταση.

Το επόμενο κομμάτι αποτελεί το Feed Forward Network (FFN) του μετασχηματιστή, το οποίο είναι πανομοιότυπο με μοντέλα βαθιάς μάθησης που χρησιμοποιούνταν πριν την εφεύρεση του μετασχηματιστή. Αυτό το κομμάτι χρησιμοποιώντας τις συσχετίσεις που έφερε στην επιφάνεια ο μηχανισμός προσοχής, και το μεγαλύτερο ποσοστό των παραμέτρων του μοντέλου ερμηνεύει το νόημα της κωδικοποίησης και μαντεύει τι μπορεί να ακολουθήσει.

Τέλος, αφού η διαδικασία αυτή επαναληφθεί τόσες φορές όσα τα επίπεδα του μοντέλου, ένας απλός ταξινομητής (classifier) παράγει τον τελικό πίνακα πιθανοτήτων (logits), από τον οποίο θα προκύψει η επόμενη λέξη στην έξοδο.

Οι πράξεις που εκτελούνται στα διάφορα στάδια του LLM, είναι κατά κύριο λόγο, όπως θα δούμε και παρακάτω πολλαπλασιασμοί μεταξύ πίνακα και διανύσματος αριθμών κινητής υποδιαστολής. Τα κουτιά με κόκκινο χρώμα εκτελούν αποκλειστικά τέτοιου είδους πράξεις και καταλαμβάνουν πάνω από το 99% του συνολικού χρόνου εκτέλεσης.

Οι υπόλοιπες πράξεις (RMS κανονικοποίηση, Περιστροφική Κωδικοποίηση Θέσης-RoPE Encoding) κάνουν χρήση μη γραμμικών μαθηματικών συναρτήσεων που δεν θα αποτελέσουν μεγάλο πρόβλημα, κυρίως διότι δεν καταλαμβάνουν μεγάλο ποσοστό του υπολογιστικού χρόνου.

1.3 Σχετική Έρευνα

Παρακάτω θα παρουσιαστούν σύντομα μερικές έρευνες που έχουν δημοσιευθεί πάνω σε CGRA's και συγκεκριμένα σε απεικόνιση εφαρμογών τεχνητής νοημοσύνης σε αυτά, και θα συσχετιστούν με την δική μας δουλειά.

1.3.1 Plasticine

Το Plasticine [15] είναι μια αρχιτεκτονική CGRA που επιχειρεί να επιλύσει τα προβλήματα απόδοσης και κατανάλωσης ενέργειας που αντιμετωπίζουν τα FPGAs, εκμεταλλευόμενη επαναλαμβανόμενα πρότυπα παραλληλισμού (parallel patterns) στον κώδικα. Η βασική καινοτομία είναι η χρήση Μονάδων Υπολογισμού Προτύπων (Pattern Compute Units, PCUs) και Μονάδων Μνήμης Προτύπων (Pattern Memory Units, PMUs), οι οποίες βελτιστοποιούν την τοπικότητα των δεδομένων και την πρόσβαση στη μνήμη, επιταχύνοντας επαναλαμβανόμενα υπολογιστικά μοτίβα.

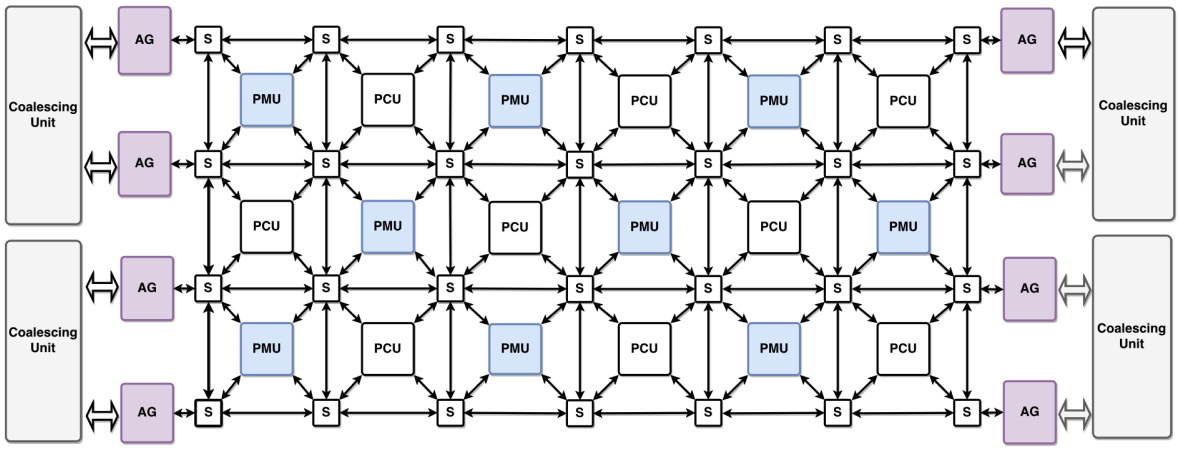


Figure 1.3.1: Αρχιτεκτονική του CGRA Plasticine

Η απεικόνιση εφαρμογών στο CGRA Plasticine βασίζεται αυστηρά στη μοντελοποίηση τους ως parallel patterns (παράλληλα μοτίβα). Οι χρήστες πρέπει να περιγράψουν την εφαρμογή τους σε μια εξειδικευμένη γλώσσα, την Delite Hardware Definition Language (DHDL) [16], η οποία επιβάλλει αυστηρή αντιστοίχιση σε συγκεκριμένα τέτοια παράλληλα μοτίβα. Αυτό επιτρέπει την υψηλή αποδοτικότητα σε όρους ενέργειας και απόδοσης, ξεπερνώντας σε επιδόσεις τα FPGAs κατά περισσότερο από 10 φορές σε πολλές περιπτώσεις.

Η δομή του Plasticine είναι ιδιαίτερα ομοιογενής, καθώς το πλέγμα αποτελείται αποκλειστικά από PCUs και PMUs που συνδέονται μέσω ενός στατικού, υβριδικού δικτύου διασύνδεσης. Τα PMUs περιλαμβάνουν μνήμη scratchpad με εξειδικευμένη λογική διευθυνσιοδότησης, ενώ η επικοινωνία με την εξωτερική DRAM πραγματοποιείται μέσω ειδικών γεννητριών διευθύνσεων και μονάδων συγχώνευσης αιτήσεων μνήμης.

Σε αντίθεση με το R-Blocks, το Plasticine εξαρτάται από την ύπαρξη επαναλαμβανόμενων παράλληλων μοτίβων για να απεικονίσει αποδοτικά τον κώδικα στο υλικό. Αν και αυτό το μοντέλο προσφέρει εξαιρετική ενεργειακή απόδοση, μειώνει την ευελιξία της αρχιτεκτονικής, καθιστώντας την λιγότερο κατάλληλη για εφαρμογές με πιο ετερογενή υπολογιστικά μοτίβα.

1.3.2 X-CGRA

Το X-CGRA [17] είναι μια αρχιτεκτονική CGRA που στοχεύει στην επιτάχυνση εφαρμογών ανθεκτικών σε σφάλματα, όπως η επεξεργασία πολυμέσων και σήματος, αξιοποιώντας τις προσεγγιστικές του δυνατότητες (approximate computing). Μέσω των Ποιοτικά Κλιμακούμενων Μονάδων Επεξεργασίας (Quality-Scalable Processing Elements, QSPEs), έχει τη δυνατότητα να αυξομειώνει δυναμικά το

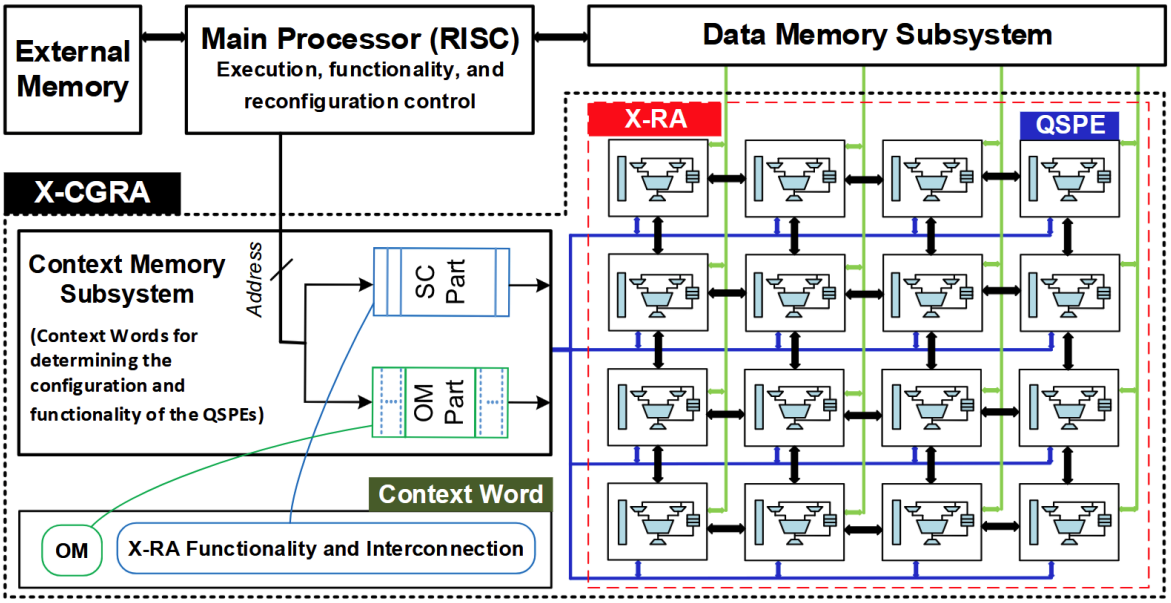


Figure 1.3.2: Αρχιτεκτονική του X-CGRA

επίπεδο ακρίβειας των υπολογισμών του. Παράλληλα, μπορεί να υποστηρίξει περιπτώσεις όπου η ποιότητα υπηρεσίας (Quality of Service, QoS) απαιτείται να είναι 100%.

Η αρχιτεκτονική του X-CGRA ακολουθεί τυπικές δομές CGRA, με τα QSPEs να είναι οργανωμένα σε πλέγμα και να διασυνδέονται μέσω ενός δικτύου 2D mesh. Η υπομονάδα μνήμης περιβάλλοντος είναι υπεύθυνη για τη ρύθμιση της δομής και της λειτουργικής κατάστασης του επιταχυντή, καθώς και για την επικοινωνία με τον εξωτερικό κεντρικό επεξεργαστή.

Η απεικόνιση εφαρμογών στο X-CGRA περιλαμβάνει δύο βασικά βήματα. Αρχικά, καθορίζεται το βέλτιστο επίπεδο ακρίβειας, και στη συνέχεια ο κώδικας προγραμματίζεται και αντιστοιχίζεται στις QSPE μονάδες χρησιμοποιώντας απεικόνιση πυρήνων (loop kernels) με DFG γράφους. Αυτός ο δυναμικός τρόπος προσαρμογής της ακρίβειας επιτρέπει μέγιστη ενεργειακή αποδοτικότητα, αλλά παράλληλα επιβαρύνει σημαντικά τη διαδικασία μεταγλώττισης. Εάν η ακρίβεια μπορούσε να ρυθμιστεί στατικά, το κόστος μεταγλώττισης θα μπορούσε να μειωθεί σημαντικά.

Παίρνοντας έμπνευση από το X-CGRA, το R-Blocks θα μπορούσε και αυτό να επεκταθεί ώστε να υποστηρίζει προσεγγιστικούς υπολογισμούς. Λόγω της ευέλικτης σχεδίασης του, η προσθήκη ενός στατικού προσεγγιστικού FU, όπως ενός προσεγγιστικού πολλαπλασιαστή, είναι τεχνικά εφικτή. Κάποιες ενδοεργαστηριακές προσπάθειες δείχνουν ήδη πολλά υποσχόμενα αποτελέσματα, ενώ μια τέτοια προσέγγιση θα μπορούσε να αξιοποιηθεί και σε εφαρμογές LLMs, αρκεί το σφάλμα ποιότητας να παραμένει αρκετά χαμηλό ώστε να μην επηρεάζει σημαντικά την ακρίβεια του μοντέλου.

1.3.3 CGRA-ME

Το CGRA-ME [18] είναι ένα framework που επιχειρεί να ενοποιήσει τον σχεδιασμό, τη μοντελοποίηση και τη προσομοίωση διαφόρων τύπων CGRA αρχιτεκτονικών. Το 2024, με την προσθήκη νέων δυνατοτήτων [19], το σύστημα επεκτάθηκε για να υποστηρίζει πιο ευέλικτες αρχιτεκτονικές CGRA, αριθμητική κινητής υποδιαστολής, predication και υβριδικά συστήματα που συνδυάζουν RISC-V επεξεργαστές με CGRAs.

Ο βασικός στόχος του CGRA-ME είναι η εξερεύνηση του χώρου αρχιτεκτονικών των CGRA. Λειτουργεί λαμβάνοντας ως είσοδο μια περιγραφή αρχιτεκτονικής σε XML και παρέχει εργαλεία για χρονοπρογραμματισμό (scheduling), απεικόνιση κώδικα (mapping), τοποθέτηση (placement) και δρομολόγηση (routing). Το σύστημα είναι ενσωματωμένο στο περιβάλλον του LLVM compiler, επιτρέποντας την αυτόματη προσαρμογή και μεταγλώττιση εφαρμογών γραμμένων σε C για διαφορετικές CGRA αρχιτεκτονικές, μέσα σε ένα

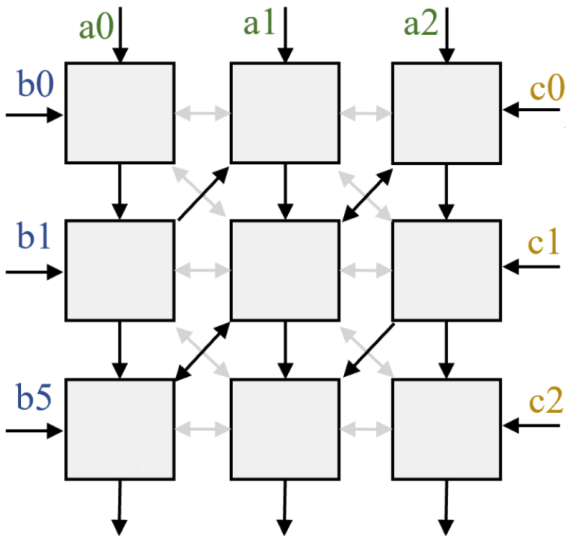


Figure 1.3.3: Τοπολογία διασυνδέσεων στο ML-CGRA

ενιαίο περιβάλλον ανάπτυξης. Αυτή η δυνατότητα επιτρέπει στους χρήστες να πειραματιστούν και να αξιολογήσουν διαφορετικές σχεδιαστικές επιλογές, ενώ τηρούνται οι εκάστοτε περιορισμοί του συστήματος.

Η ύπαρξη ενός τέτοιου ενοποιημένου πλαισίου είναι ιδιαίτερα χρήσιμη, καθώς οι τεχνολογίες CGRA κερδίζουν ολοένα και περισσότερο έδαφος. Ένα τέτοιο εργαλείο μπορεί να προσφέρει πολύτιμες πληροφορίες σχετικά με τους σχεδιαστικούς συμβιβασμούς και τις επιπτώσεις διαφορετικών αρχιτεκτονικών επιλογών. Παρόλα αυτά, η άμεση και ευρεία υιοθέτηση του από την ερευνητική κοινότητα θα μπορούσε να λειτουργήσει περιοριστικά, μειώνοντας το εύρος εξερεύνησης νέων αρχιτεκτονικών και περιορίζοντας τη δημιουργικότητα των ερευνητών.

1.3.4 ML-CGRA

Το ML-CGRA [20], επικεντρώνεται στην εκτέλεση εφαρμογών machine learning (ML). Η υπολογιστική δομή αυτών των εφαρμογών είναι παρόμοια με αυτή των μετασχηματιστών και των LLM, καθώς βασίζονται κυρίως στον γενικευμένο πολλαπλασιασμό πινάκων (generalized matrix multiplication) για τον υπολογισμό των ενεργοποιήσεων από τα βάρη και τις εισόδους του μοντέλου.

Το ML-CGRA αποτελεί ένα πλήρως open-source ενοποιημένο πλαίσιο μεταγλώττισης (end-to-end compilation framework), σχεδιασμένο για αποδοτική και ευέλικτη εκτέλεση ML αλγορίθμων σε CGRAs. Η αρχιτεκτονική του βασίζεται σε ένα τυπικό πρότυπο CGRA, το οποίο έχει βελτιωθεί με πρόσθετα εξειδικευμένα χαρακτηριστικά. Συγκεκριμένα, το συμβατικό on-chip δίκτυο αντικαταστάθηκε από ένα δίκτυο διασύνδεσης τύπου king mesh 8 κατευθύνσεων (εικόνα 1.3.3), που προσφέρει πιο αποδοτική δρομολόγηση δεδομένων. Επιπλέον, οι είσοδοι διαθέτουν ρυθμιζόμενη καθυστέρηση (configurable stalling) και υποστήριξη διπλής αποθήκευσης (double-buffering), ενώ οι αποκλειστικές μονάδες MAC και MAX επιτρέπουν την εκτέλεση βασικών ML πράξεων σε έναν μόνο κύκλο ρολογιού.

Με αυτές τις καινοτομίες, το σύστημα καταφέρνει σημαντικές βελτιώσεις απόδοσης, επιτυγχάνοντας επιτάχυνση 3.15× σε 4×4 CGRAs και 6.02× σε 8×8 CGRAs. Παρόλο που αυτά τα αποτελέσματα είναι εντυπωσιακά, πρέπει να σημειωθεί ότι οι τεχνικές επιτάχυνσης εφαρμογών ML που χρησιμοποιούνται δεν θα μπορούσαν να εφαρμοστούν χωρίς σοβαρές τροποποιήσεις για την επιτάχυνση LLM καθώς τους λείπει η ευελιξία που απαιτεί η απεικόνιση μη-γραμμικών συναρτήσεων.

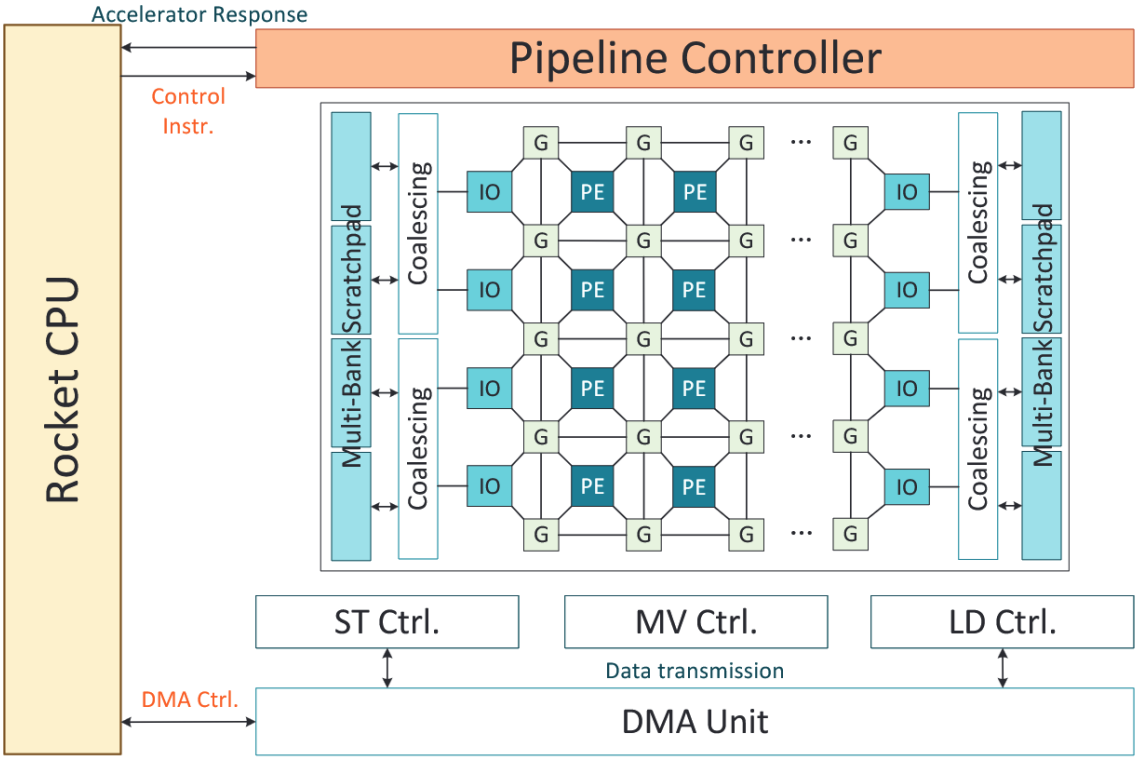


Figure 1.3.4: Η αρχιτεκτονική του CCGRA CFEACT

1.3.5 CGLA-IMAX3

Το IMAX3 [21] είναι ένα CGRA/CGLA (Coarse-Grained Linear Array Architecture), υλοποιημένο σε FPGA, σχεδιασμένο για LLM inference. Η απεικόνιση του κώδικα γίνεται μέσω της βιβλιοθήκης GGML, που επιτρέπει τη μεταφορά εφαρμογών Python στο IMAX3 χωρίς την ανάγκη περίπλοκης ανασχεδίασης.

Αρχιτεκτονικά, το IMAX3 διαφοροποιείται από τα παραδοσιακά 2D CGRAs υιοθετώντας μια διασύνδεση δακτυλίου (ring array interconnect), η οποία μειώνει τις ανισορροπίες προσπέλασης μνήμης και βελτιώνει το bandwidth συγκριτικά με τις GPUs. Οι υπολογιστικές μονάδες είναι βασισμένες σε CISC αριθμητικές μονάδες, μειώνοντας την πολυπλοκότητα της διασύνδεσης και επιταχύνοντας τη μεταγλώττιση.

Η δομή της διασύνδεσης αυτής εξαλείφει μεγάλο ποσοστό του overhead επαναδιαμορφώσεων, επιτρέποντας στους προγραμματιστές να παρέχουν οδηγίες σχετικά με multi-input υπολογισμούς, αναφορές σε cache και DMA transfers. Μετά από βελτιστοποιήσεις στο matrix multiplication, που αποτελεί τον κυρίαρχο υπολογισμό στα LLMs, το IMAX3 πέτυχε έως και 20% βελτίωση απόδοσης έναντι εμπορικών CPUs, καθιστώντας το το πρώτο CGRA, από όσα τουλάχιστον καταφέραμε να βρούμε, που δοκιμάστηκε επιτυχώς με πραγματικές εφαρμογές LLM inference. Το ξεχωριστό του αρχιτεκτονικό μοντέλο, όμως, το καθιστά ακατάλληλο για άμεσες συγκρίσεις με πιο κλασσικού τύπου CGRA.

1.3.6 CFEACT

Το CFEACT [22] είναι ένα πλαίσιο για την υλοποίηση και αξιολόγηση προσαρμοσμένων accelerator SoCs βασισμένο σε CGRA, με έμφαση στην επιτάχυνση transformers και CNNs. Περιλαμβάνει έναν hardware generator που υποστηρίζει πολλαπλές διαμορφώσεις και παραμέτρους σχεδίασης, ενώ ο front-end compiler αναλύει τους πυρήνες εκτέλεσης της εφαρμογής και δημιουργεί αποδοτικά το data flow graph (DFG) για τη συγκεκριμένη CGRA διαμόρφωση.

Η αρχιτεκτονική του CFEACT ακολουθεί τη συνηθισμένη δομή των CGRAs, με ένα ομοιογενές πλέγμα

CGRA	Publication Date	Architecture	Platform	Mapping Technique	Architectural DSE	LLM Mapped
Plasticine	June 2017	Homogeneous PEs and Memory Elements	ASIC	Utilizing parallel patterns (DHDL)	Manual	No
X-CGRA	October 2020	Homogeneous QSPEs	ASIC	DFG kernel mapping	Manual	No
CGRA-ME	May 2024	3 homogeneous PE architectures	Simulated	C kernel mapping. Differs by architecture	Manual	No
ML-CGRA	July 2023	Homogeneous PEs	Simulated	Python MLIR into kernel mapping	Manual	No
CGLA	November 2024	IMAX3 in memory CISC	FPGA, ASIC	Manual, utilizing GGML	Manual	Yes
CFEACT	September 2024	Homogeneous PEs	ASIC	Linearization and kernel mapping	Scalable Template	Yes
R-Blocks (our work)	-	Disaggregated PEs (op. level)	ASIC	High-level architecture targeting compiler	Semi-Automated	Yes

Figure 1.3.5: Πίνακας συγκρίσεων μεταξύ των διαφόρων CGRA που παρουσιάστηκαν

Processing Elements (PEs) συνδεδεμένο μέσω ενός δικτύου καλωδίωσης. Περιλαμβάνει επίσης μονάδες εισόδου-εξόδου και memory controllers που περιβάλλουν το αναδιαμορφώσιμο τμήμα, επικοινωνώντας με έναν εξωτερικό κύριο επεξεργαστή και χρησιμοποιεί DMA για προσπέλαση μνήμης.

Για εφαρμογές transformers, η ενσωμάτωση εκτεταμένων βελτιστοποιήσεων στον front-end compiler επιτρέπει στο CFEACT να επιτυγχάνει 58% υψηλότερη απόδοση από το ML-CGRA ίδιου μεγέθους. Αυτό ήταν αναμενόμενο, δεδομένου ότι η αρχιτεκτονική είναι ειδικά προσαρμοσμένη για transformers. Σύμφωνα με τους ερευνητές, με καλύτερη αξιοποίηση των λειτουργικών μονάδων υπολογισμού, η βελτίωση στο Area-Delay Product (ADP) μπορεί να ξεπεράσει το 2× σε σχέση με το ML-CGRA.

1.3.7 Σύνοψη

Στον πίνακα της εικόνας 1.3.5 συνοψίζονται όσες αρχιτεκτονικές CGRA αναφέρθηκαν παραπάνω. Αυτό που αξίζει να παρατηρηθεί είναι ότι το R-Blocks είναι το μοναδικό από αυτά που χρησιμοποιεί ετερογενείς λειτουργικές μονάδες και υψηλού επιπέδου μεταγλωττιστή που στοχεύει συγκεκριμένες αρχιτεκτονικές.

Επιπλέον, οι μοναδικές προσπάθειες απεικόνισης LLM σε αρχιτεκτονικές CGRA, σύμφωνα με όσα γνωρίζουμε, δημοσιεύθηκαν στα τέλη του 2024, δηλαδή όταν η δική μας δουλειά είχε ήδη ξεκινήσει, κάτι που δείχνει πόσο καινούριο είναι αυτό το ερευνητικό πεδίο.

1.4 Επέκταση

Περνώντας στο στάδιο της επέκτασης του περιβάλλοντος R-Blocks υπενθυμίζουμε ότι η κινητήρια δύναμη και ο λόγος πίσω από αυτή την επέκταση είναι η ενσωμάτωση δυνατοτήτων χειρισμού αριθμών και πράξεων κινητής υποδιαστολής από το hardware, έτσι ώστε να είναι σε θέση να εκτελεί αποδοτικά εφαρμογές LLM που βασίζονται κατά κύριο λόγο σε αυτές.

1.4.1 Επέκταση Υλικού

Το πρότυπο IEEE754 καθορίζει τους κανόνες για αριθμητικές πράξεις με αριθμούς κινητής υποδιαστολής, τις μορφές αποθήκευσης τους, τις μετατροπές, τους τρόπους στρογγυλοποίησης, καθώς και τις εξαιρέσεις και τις ειδικές τιμές (π.χ. άπειρο ή NaN) που μπορεί να πάρουν. Μία μονάδα αριθμών κινητής υποδιαστολής (Floating Point Unit, FPU) πρέπει να παράγει αποτελέσματα σύμφωνα με αυτό το πρότυπο για να εξασφαλίζεται η ομαλή συνεργασία με τα υπόλοιπα στοιχεία υλικού και λογισμικού.

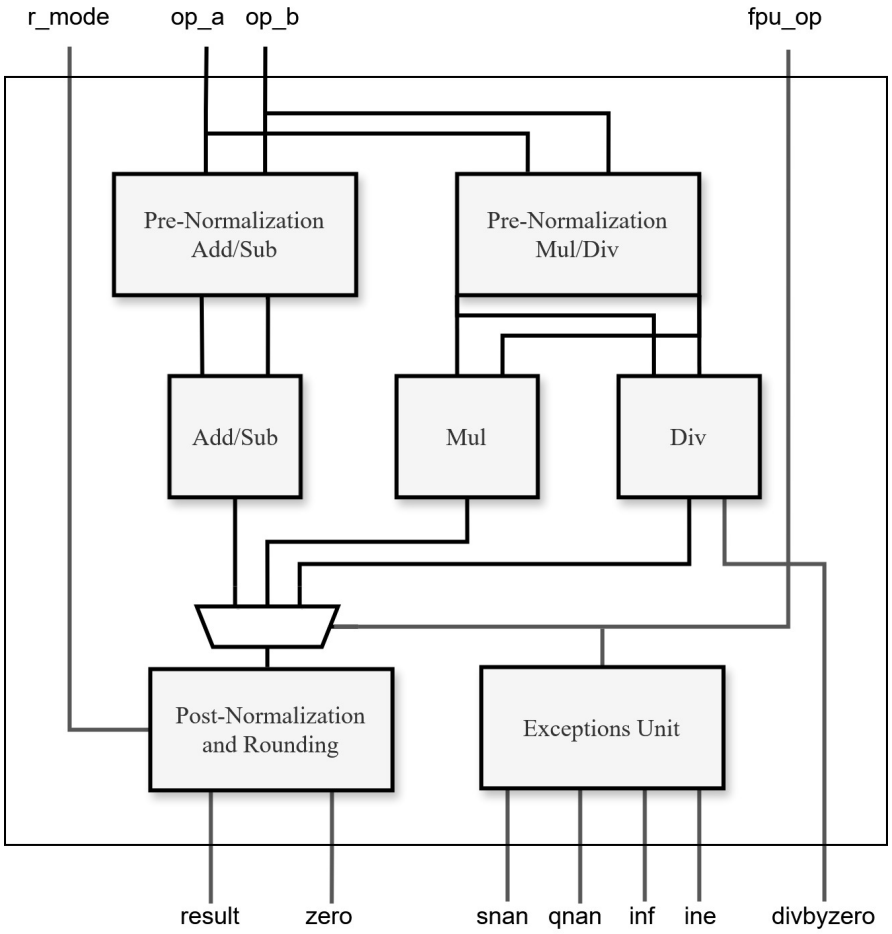


Figure 1.4.1: Η μικροαρχιτεκτονική δομή της μονάδας κινητής υποδιαστολής

Για τον σκοπό αυτό, χρησιμοποιούμε την open-source FPU από το OpenCores [23], η οποία είναι πλήρως συμβατή με το πρότυπο IEEE754 για αριθμούς κινητής υποδιαστολής single-precision, δηλαδή 32-bit, είναι γρήγορη και απλή, και παρέχει όλες τις δυνατότητες που μας είναι απαραίτητες.

Η αξιοπιστία του υλικού αξιολογήθηκε ξεχωριστά, με πάνω από 14 εκατομμύρια διανύσματα ελέγχου που παρήχθησαν από τη βιβλιοθήκη SoftFloat του Berkeley [24].

Η FPU υποστηρίζει τις εξής βασικές λειτουργίες με FP32 εισόδους και εξόδους:

Table 1.1: Τα σήματα εισόδου/εξόδου της FPU

Όνομα Σήματος	**Πλάτος**	**Κατεύθυνση**	**Περιγραφή**
clk	1	Είσοδος	Ρολόι Συστήματος
r_mode	1	Είσοδος	Λειτουργία Στρογγυλοποίησης
fpu_op	1	Είσοδος	Επιλογέας Πράξης Κινητής Υποδιαστολής
op_a, op_b	32	Είσοδος	Τελεστέος A και B
out	32	Έξοδος	Έξοδος Αποτελέσματος
snan	1	Έξοδος	Κάποιος τελεστέος είναι SNAN
qnan	1	Έξοδος	Η έξοδος είναι QNAN
inf	1	Έξοδος	Η έξοδος είναι INF
ine	1	Έξοδος	Το αποτέλεσμα είναι Inexact
divbyzero	1	Έξοδος	Η πράξη είναι διαίρεση και το op_b είναι μηδέν
zero	1	Έξοδος	Η έξοδος είναι αριθμητικό μηδέν

Table 1.2: Τα σήματα του συγκριτή

Όνομα Σήματος	**Πλάτος**	**Κατεύθυνση**	**Περιγραφή**
clk	1	Είσοδος	Ρολόι Συστήματος
opa, opb	32	Είσοδος	Τελεστέος A και B
unordered	1	Έξοδος	Το opa ή το opb είναι NAN
altb	1	Έξοδος	Το opa είναι μεγαλύτερο από το opb
blta	1	Έξοδος	Το opb είναι μεγαλύτερο από το opa
aeqb	1	Έξοδος	Το opa είναι ίσο με το opb
inf	1	Έξοδος	Το opa ή το opb είναι INF
zero	1	Έξοδος	Το opa είναι αριθμητικό μηδέν

- Πρόσθεση
- Αφαίρεση
- Πολλαπλασιασμό
- Διαίρεση
- Μετατροπές (από INT32 σε FP32 και αντίστροφα)
- Συγκρίσεις

Με αυτές τις στοιχειώδεις αριθμητικές πράξεις και τη χρήση υψηλού επιπέδου βιβλιοθηκών μαθηματικών (π.χ. math.h της C), μπορούμε να υλοποιήσουμε κάθε δυνατή πράξη, όπως τετραγωνική ρίζα, εκθετικές και τριγωνομετρικές συναρτήσεις. Επιπρόσθετα, το υλικό υποστηρίζει τέσσερις διαφορετικές λειτουργίες στρογγυλοποίησης, αλλά εμείς θα χρησιμοποιήσουμε μόνο τη συνηθισμένη μέθοδο "στρογγυλοποίηση στον πλησιέστερο πλήρη αριθμό", καθώς οι υπόλοιπες δεν συνάδουν με τη δομή του R-Blocks και δεν είναι απαραίτητες για την εφαρμογή μας.

Αρχιτεκτονικά, η FPU διαθέτει 4 στάδια pipeline και μπορεί να εκτελεί μία πράξη ανά κύκλο: Οι είσοδοι –όπως ο τύπος της πράξης, ο τρόπος στρογγυλοποίησης και οι τελεστέοι– αποθηκεύονται σε buffers για όσο χρειάζονται, και το αποτέλεσμα παράγεται μετά από τέσσερις κύκλους ρολογιού.

Η εσωτερική δομή της FPU (εικόνα 1.4.1) περιλαμβάνει δύο μονάδες pre-normalization: μία για πρόσθεση/αφαίρεση και μία για πολλαπλασιασμό/διαίρεση. Οι αντίστοιχες βασικές μονάδες (Add/Sub, Mul, Div) εκτελούν τις αριθμητικές πράξεις, ενώ μια κοινή μονάδα post-normalization φροντίζει για την τελική κανονικοποίηση και στρογγυλοποίηση πριν το αποτέλεσμα κωδικοποιηθεί σε μορφή single-precision FP.

Επίσης, το σύστημα περιλαμβάνει μια ξεχωριστή μονάδα σύγκρισης, πλήρως συμβατή με το πρότυπο

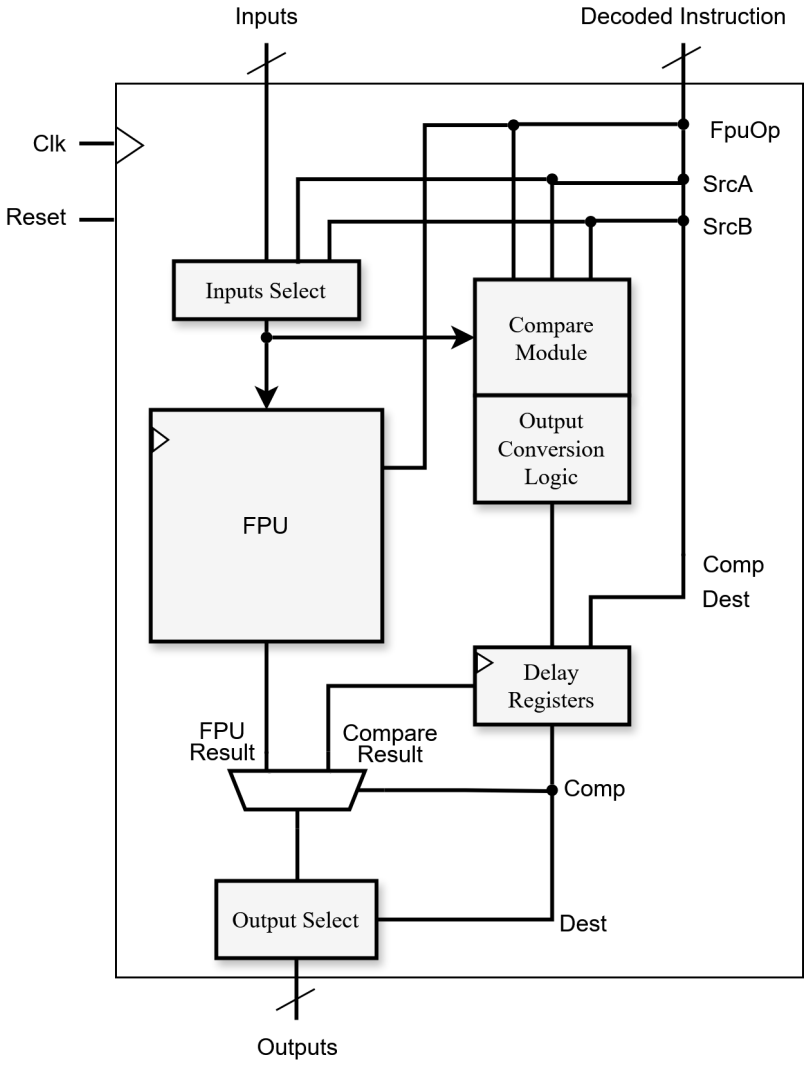


Figure 1.4.2: Η τελική μορφή της νέας λειτουργικής μονάδας

IEEE754 για single-precision floating-point αριθμούς. Αυτή η μονάδα, η οποία έχει σχεδιαστεί και επαληθευτεί ανεξάρτητα, επίσης με χρήση της βιβλιοθήκης SoftFloat, είναι πιο απλή και παράγει αποτελέσματα έναν κύκλο μετά την ενεργοποίηση των εισόδων.

Τόσο η FPU όσο και η μονάδα σύγκρισης έχουν περάσει από σύνθεση σε τεχνολογία FD-SOI 22nm, με μέγιστη συχνότητα λειτουργίας τα 415MHz προσδιορισμένη από τον Synopsys Design Compiler. Οι μετρικές επιφάνειας και κατανάλωσης ενέργειας είναι συγκρίσιμες με άλλες λειτουργικές μονάδες του περιβάλλοντος R-Blocks, επιτρέποντας ομαλή ενσωμάτωση της FPU στο σύστημα.

Οι πίνακες 1.1 και 1.2 εξηγούν τα σήματα εισόδου και εξόδου των δύο μονάδων.

Το νέο υλικό πρέπει να έχει συγκεκριμένη δομή ώστε να ενσωματωθεί ως tile στο οικοσύστημα του R-Blocks. Ενώ τα περισσότερα tiles (όπως ALU, RF, και MUL) έχουν καθυστέρηση ενός μόνο κύκλου (για την εγγραφή στον καταχωρητή εξόδου), η FPU χρησιμοποιεί pipeline 4 σταδίων, κάτι που οδηγεί σε καθυστέρηση εξόδου 5 κυκλών (4 για επεξεργασία και 1 για καταχώρηση).

Τα tiles διαθέτουν συγκεκριμένες εισόδους και εξόδους (8 εισόδους και 2 εξόδους των 32 bits όπως φαίνεται και στον πίνακα 1.3), συνεπώς χρειάζεται ένα hardware wrapper για τη μετατροπή των σημάτων του CGRA σε σήματα κατανοητά από την FPU και το αντίστροφο. Η πληροφορία για τις θύρες εισό-

Table 1.3: Τα σήματα ενός Tile του R-Blocks

Όνομα Σήματος	**Πλάτος**	**Κατεύθυνση**	**Περιγραφή**
Clk	1	Είσοδος	Ρολόι Συστήματος
Reset	1	Είσοδος	Επαναφορά Συστήματος
Inputs	8x32	Είσοδος	8 διαφορετικές είσοδοι των 32-bit ενωμένες
DecodedInstruction	12 έως 33	Είσοδος	Το πλάτος εξαρτάται από το tile
Outputs	2x32	Έξοδος	2 διαφορετικές έξοδοι των 32-bit ενωμένες

Table 1.4: Output Conversion Logic

Τελεστής C	**Όνομα Εντολής Assembly**	**Λογική Μετατροπής**
==	eqf	NOT unordered AND aeqb
!=	nef	NOT unordered AND NOT aeqb
>	gtf	NOT unordered AND altb
>=	gef	NOT unordered AND (altb OR aeqb)
<	ltf	NOT unordered AND blta
<=	lef	NOT unordered AND (blta OR aeqb)

δου και εξόδου που θα χρησιμοποιηθούν προκύπτει από το σήμα *DecodedInstruction*, το οποίο περιέχει στοιχεία όπως:

- *OutputValid* (ενεργοποίηση εξόδου),
- Επιλογείς θυρών πηγής και προορισμού,
- Τύπο πράξης
- Σήματα ελέγχου (π.χ. για υποστήριξη λειτουργίας διπλών εξόδων για μεγαλύτερους τελεστέους σε συγκεκριμένα FU).

Για την FPU, το σήμα αυτό διαχωρίζεται σε ειδικά σήματα ελέγχου, όπως `CompareFlag`, `FPU opcode`, `RoundingMode`, καθώς και σε σήματα επιλογής θυρών εισόδου/εξόδου. Επιπλέον, λόγω του pipelining πολλών σταδίων, τα σήματα ελέγχου και τα αποτελέσματα της μονάδας σύγκρισης πρέπει να καθυστερήσουν κατά 4 κύκλους περνώντας μέσω ενός μπλοκ καταχωρητών καθυστέρησης για λόγους συγχρονισμού.

Τέλος, η λογική μετατροπής εξόδου (Output Conversion Logic) διαμορφώνει τα αποτελέσματα του συγκριτή ώστε να είναι σύμφωνα με το πρότυπο IEEE754 και να απαντούν στις ερωτήσεις των υψηλού επιπέδου εντολών (π.χ. "Is A > B?") αντί για την σύμβαση που χρησιμοποιεί η λογική εξόδου του συγκριτή που χρησιμοποιήσαμε (πίνακας 1.4).

Η νέα ολοκληρωμένη λειτουργική μονάδα που ενσωματώνουμε απεικονίζεται στην εικόνα 1.4.2.

Με αυτόν τον τρόπο επιτυγχάνεται η πλήρης ενσωμάτωση της FPU στο υλικό του R-Blocks, και το μόνο που μένει είναι η υποστήριξή της και από τα εργαλεία.

1.4.2 Επέκταση Λογισμικού και Εργαλείων

Το R-Blocks, όπως αναφέραμε και εισαγωγικά, βασίζεται στο μοντέλο TTA για Exposed Datapath Architectures (EDPAs), πράγμα που σημαίνει ότι στον πυρήνα του είναι μια TTA μηχανή με συγκεκριμένους περιορισμούς που καθορίζουν τα αρχιτεκτονικά του χαρακτηριστικά. Για την απεικόνιση υψηλού επιπέδου κώδικα στο αναδιαμορφώσιμο υλικό, απαιτείται μια διαδικασία πολλαπλών επιπέδων.

Αρχιτεκτονική Συνόλου Εντολών (ISA)

Η επικοινωνία μεταξύ λογισμικού και υλικού ξεκινά από το Instruction Decoder (ID), μια ειδική λειτουργική μονάδα (Functional Unit, FU) που αποκωδικοποιεί εντολές και τις προωθεί στον αντίστοιχο υπολογιστικό tile. Η πλήρης περιγραφή του συνόλου εντολών γίνεται σε ένα αρχείο XML, όπου ορίζονται όλοι οι διαφοερτικοί τύποι FUs, οι εντολές τους και η αντιστοίχιση των αποκωδικοποιημένων bit τους σε συγκεκριμένα σημεία του opcode.

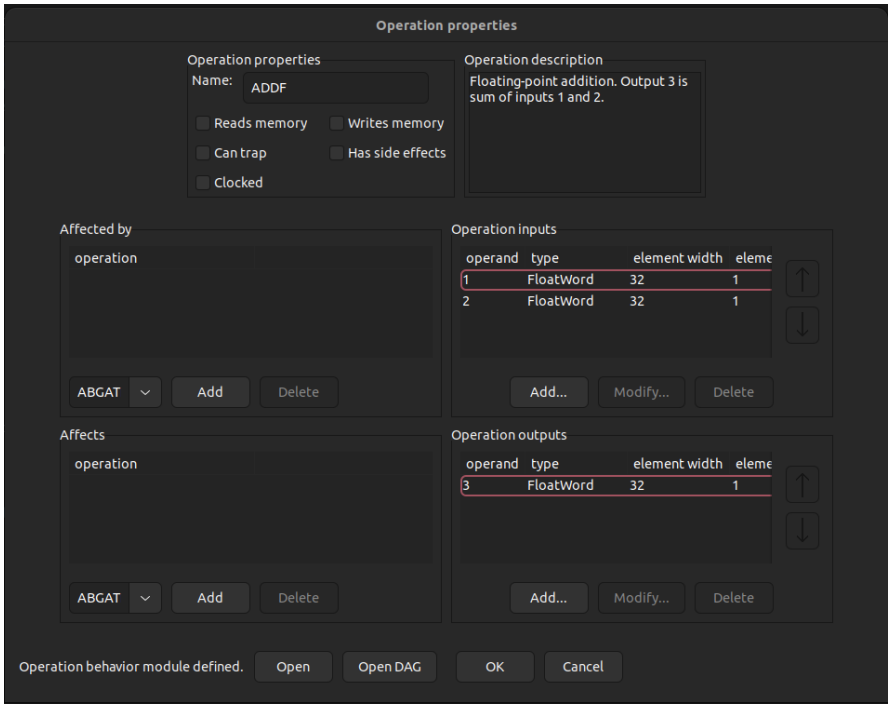


Figure 1.4.3: Το γραφικό περιβάλλον επεξεργασίας του OSAL

Το ISA της FPU αποτελείται από 16-bit αποκωδικοποιημένων εντολών και περιλαμβάνει βασικές αριθμητικές πράξεις κινητής υποδιαστολής, μετατροπές μεταξύ FP32 και INT32 και πράξεις σύγκρισης, οι οποίες φαίνονται στο παρακάτω απόσπασμα από το εν λόγω αρχείο.

```
<FPU decoded_width="16">
    <nop        decoded_instr="0_0_000_00_00_?_???_???"/>
    <pass       decoded_instr="0_0_000_00_01_D_???_AAA"/>

    <addf       decoded_instr="1_0_000_00_00_D_BBB_AAA"/>
    <subf       decoded_instr="1_0_001_00_00_D_BBB_AAA"/>
    <mulf       decoded_instr="1_0_010_00_00_D_BBB_AAA"/>
    <divf       decoded_instr="1_0_011_00_00_D_BBB_AAA"/>

    <cif        decoded_instr="1_0_100_00_00_D_???_AAA"/>
    <cfi        decoded_instr="1_0_101_00_00_D_???_AAA"/>

    <eqf        decoded_instr="1_1_000_00_00_D_BBB_AAA"/>
    <nef        decoded_instr="1_1_001_00_00_D_BBB_AAA"/>
    <gtf        decoded_instr="1_1_010_00_00_D_BBB_AAA"/>
    <gef        decoded_instr="1_1_011_00_00_D_BBB_AAA"/>
    <ltf        decoded_instr="1_1_100_00_00_D_BBB_AAA"/>
    <lef        decoded_instr="1_1_101_00_00_D_BBB_AAA"/>
</FPU>
```

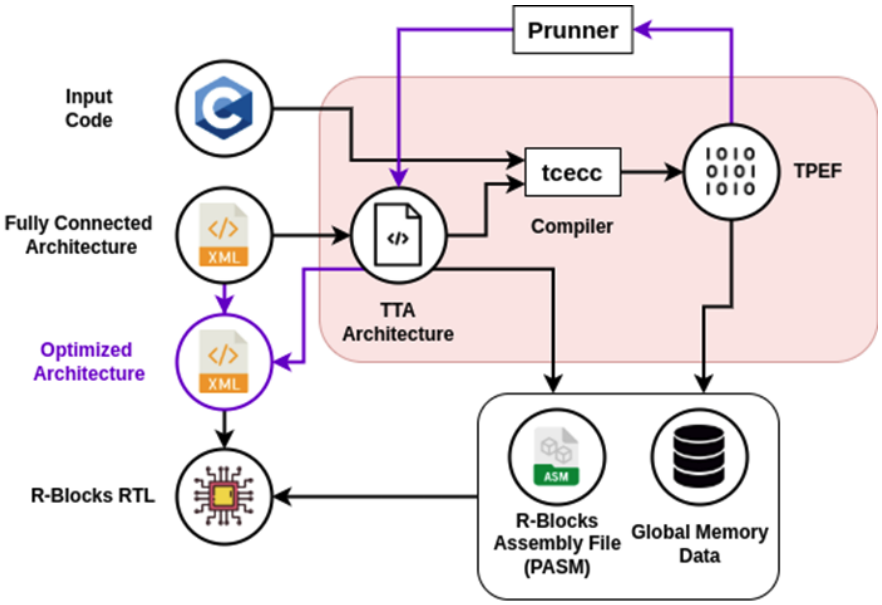


Figure 1.4.4: Η διαδικασία μεταγλώττισης του περιβάλλοντος R-Blocks

Οι ονομασίες των εντολών δεν είναι τυχαίες, αλλά προέρχονται από το Operation Set Abstraction Layer (OSAL) του OpenASIP, το οποίο αποθηκεύει τις σημασιολογικές ιδιότητες των εντολών, αλλά όχι τον χρονισμό τους (εικόνα 1.4.3), εξασφαλίζοντας τη σωστή αντιστοιχία μεταξύ της προσαρμοσμένης μας αρχιτεκτονικής και του LLVM compiler back-end.

Εργαλείο Blocks Translator

Ο μεταγλωττιστής tcecc δέχεται ως είσοδο τον high-level κώδικα και την περιγραφή TTA της αρχιτεκτονικής στην οποία στοχεύουμε να εκτελέσουμε τον κώδικα, παράγοντας ένα TPEF executable για την αντίστοιχη TTA το οποίο τελικά μετατρέπεται σε R-Blocks εκτελέσιμο όπως έχει ήδη αναλυθεί (Εικ. 1.4.4).

Η μετατροπή της αρχιτεκτονικής περιγραφής του R-Blocks σε μορφή TTA πραγματοποιείται από το εργαλείο Blocks Translator, το οποίο περιλαμβάνει μια λεπτομερή περιγραφή συμπεριφοράς για κάθε FU. Στην περιγραφή αυτή δηλώνονται οι υπάρχουσες εντολές, ο χρονισμός του pipeline τους και οι διανυσματικές εντολές (vectorization) που υποστηρίζονται όταν πολλές FUs συνδέονται στον ίδιο ID ως vector units.

Για την υποστήριξη της FPU, ο Blocks Translator επεκτάθηκε με την κλάση BlocksFPU σε κώδικα C++, η οποία κληρονομεί από την γενική κλάση BlocksFU και περιέχει τις νέες εντολές και τα pipeline τους.

Μετά τη μεταγλώττιση του εκτελέσιμου αρχείου, ακολουθούν τα στάδια του pruning και της δημιουργίας του RTL επιπέδου περιγραφής του τελικού υλικού. Το εργαλείο prunner μειώνει τις συνδέσεις μεταξύ των FUs ώστε η αρχιτεκτονική να μπορεί να υλοποιηθεί σε πραγματικό πλέγμα R-Blocks. Στη συνέχεια, τα εργαλεία για το mapping, το routing και τη σύνθεση του τελικού bitstream μας βοηθούν στην τελική προσομοίωση, και παρέχουν στοιχεία για την κατανάλωση ισχύος και το εμβαδόν του ολοκληρωμένου κυκλώματος.

Μετά την ενσωμάτωση της νέας μονάδας, πραγματοποιήθηκαν προσομοιώσεις σε επίπεδο RTL για να διασφαλιστεί η σωστή λειτουργία των FUs και των διανυσματικών μονάδων. Επιπλέον, η απεικόνιση δοκιμαστικών συναρτήσεων από τη βιβλιοθήκη math.h επαλήθευσε ότι οι FP πράξεις εκτελούνται σωστά στην FPU, με τα αποτελέσματα να συμφωνούν με αυτά του GCC compiled κώδικα. Η αξιολόγηση της απόδοσης της FPU θα παρουσιαστεί αναλυτικά και στο επόμενο μέρος, με την απεικόνιση των LLM benchmarks.

1.5 Απεικόνιση Εφαρμογών

Μετά την επιτυχή επέκταση του υλικού και της ροής εργαλείων ώστε να υποστηρίζεται αριθμητική κινητής υποδιαστολής, παρουσιάζονται και αναλύονται τα αποτελέσματα της απεικόνισης των πρώτων LLM benchmarks σε επιλεγμένες αρχιτεκτονικές.

1.5.1 Ορισμός Αρχιτεκτονικών

Το πρώτο βήμα στη διαδικασία SW/HW co-design είναι ο καθορισμός των αρχιτεκτονικών στις οποίες θα απεικονιστεί ο κώδικας. Ως αφετηρία στον αχανή σχεδιαστικό χώρο του R-Blocks, ορίστηκαν τέσσερις αρχιτεκτονικές, οι οποίες αξιολογήθηκαν σε LLM benchmarks. Αυτές χρησιμοποιούν διαφορετικά επίπεδα παραλληλισμού, ενσωματώνοντας διαφορετικά μεγέθη διανυσματικών μονάδων. Τα χαρακτηριστικά τους φαίνονται στον Πίνακα 1.5.

Table 1.5: Παράμετροι των Αρχιτεκτονικών που εξερευνήθηκαν

Όνομα Αρχιτεκτονικής	Μέγεθος Διανύσματος	Τοπικές Μνήμες	Μέγεθος Πλέγματος	IDs	FPUs	Σύνολο FUs
Scalar	-	1	4x6	6	1	7
Vector4	4	4	6x11	12	5	29
Vector8	8	8	8x10	12	9	41
Vector16	16	16	8x18	12	17	73

Οι διανυσματικές αρχιτεκτονικές βασίζονται στην Scalar αρχιτεκτονική, περιλαμβάνοντας τις ίδιες βασικές μονάδες συστήματος (LSU, ABU, RFs), αλλά ενισχυμένες με διανυσματικές μονάδες που καθορίζουν τον βαθμό παραλληλισμού. Για παράδειγμα, η Vector8 διαθέτει 8 FPUs συνδεδεμένες ως διανυσματική μονάδα, ελεγχόμενες από τον ίδιο Instruction Decoder, αλλά μόνο μία επιπλέον FPU για μη διανυσματικές FP πράξεις (Εικ. 1.5.1). Αυτό σημαίνει ότι μπορεί να εκτελεί μέχρι και 9 (συνήθως 8) πράξεις κινητής υποδιαστολής σε κάθε κύκλο με βέλτιστο scheduling.

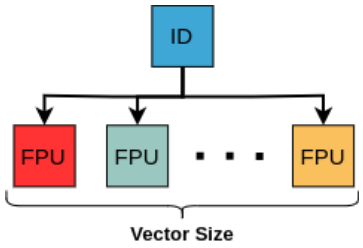


Figure 1.5.1: Απεικόνιση πολλαπλών FPU ελεγχόμενων από τον ίδιο ID

Μετά τον αρχικό καθορισμό, οι αρχιτεκτονικές πέρασαν από το εργαλείο R-Blocks Prunner, το οποίο αφαιρεί περιττές συνδέσεις ώστε να μπορεί να μπορεί να γίνει το routing της αρχιτεκτονικής στο πραγματικό υλικό. Τα τελικά πλέγματα παρουσιάζονται στις Εικ. 1.5.2-1.5.5. Σημειώνεται ότι η Vector16 δεν μπόρεσε τελικά να γίνει route με τον υπάρχοντα αλγόριθμο, λόγω του μεγάλου όγκου διασυνδέσεων, οπότε δεν ήταν δυνατή η σύνθεσή της.

1.5.2 Προετοιμασία Κώδικα

Το επόμενο βήμα είναι η επιλογή των τμημάτων του κώδικα που θα απεικονιστούν στο CGRA. Αρχικά, πραγματοποιήθηκε ανάλυση χρόνου εκτέλεσης του κώδικα inference του LLaMa2, επιβεβαιώνοντας ότι ο περισσότερος υπολογιστικός χρόνος κατανέμεται σε πολλαπλασιασμούς πινάκων. Στην εικόνα 1.5.6 φαίνεται πόσο χρόνο καταλαμβάνουν συγκριτικά τα blocks που αποτελούνται από τέτοιους πολλαπλασιασμούς και πόσο τα υπόλοιπα μέρη της εκτέλεσης (Sampling και Rest).

Τα benchmarks που επιλέχθηκαν για απεικόνιση είναι:

- Υπολογισμός πινάκων Q, K, V

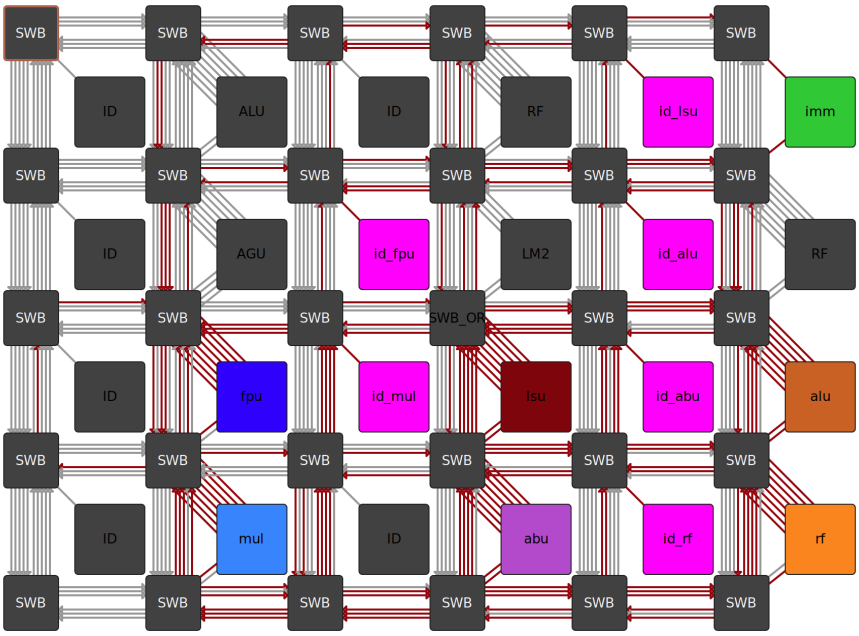


Figure 1.5.2: Αρχιτεκτονική Scalar

- Μπλοκ Attention
- Πολλαπλασιασμοί πινάκων-διανυσμάτων διαφορετικών διαστάσεων

Η επιλογή αυτών των συγκεκριμένων benchmarks βασίστηκε στην ανάγκη για συνδυασμό απαιτητικών υπολογισμών, όπως οι πολλαπλασιασμοί πίνακα-διανύσματος, και δύσκολα απεικονίσιμων μη γραμμικών συναρτήσεων, όπως εκθετικές και τριγωνομετρικές. Στόχος ήταν η αξιολόγηση της απόδοσης του CGRA σε επαναλαμβανόμενες αριθμητικές πράξεις, καθώς και η μελέτη του κόστους απόδοσης που επιφέρει η εισαγωγή πιο πολύπλοκων μαθηματικών πράξεων στο ενδιάμεσο των βασικών υπολογισμών.

Ως proof of concept της δυνατότητας απεικόνισης οποιασδήποτε λειτουργίας του κώδικα LLM, επιτεύχθηκε πλήρης απεικόνιση της συνάρτησης forward του LLaMa2, ωστόσο η υλοποίησή της σε πραγματικό υλικό απαιτεί αρχιτεκτονικές μεγαλύτερης κλίμακας. Επιπλέον, λόγω περιορισμών της μνήμης και των εργαλείων, οι κύκλοι εκτέλεσης για αυτή τη συνάρτηση εξήχθησαν αποκλειστικά μέσω του TTA simulator.

Η διαδικασία απεικόνισης περιλάμβανε αρχικά την αφαίρεση εξαρτήσεων που δεν υποστηρίζονται από βιβλιοθήκες του R-Blocks, όπως η ανάγνωση των παραμέτρων του μοντέλου, τα οποία φορτώθηκαν απευθείας στις κύριες μνήμες του CGRA. Παράλληλα, συναρτήσεις όπως printf και memcpy αντικαταστάθηκαν με αντίστοιχες του R-Blocks, οι οποίες εκτελούνται αποκλειστικά με TTA assembly. Μετά από αυτές τις προσαρμογές, τα benchmarks ήταν έτοιμα για απεικόνιση στο αναδιαμορφώσιμο υλικό με τη χρήση του compiler, αν και χωρίς δυνατότητες παράλληλης εκτέλεσης.

1.5.3 Παραλληλοποίηση Πολλαπλασιασμού Πινάκων

Για να αξιοποιηθούν οι διανυσματικές λειτουργικές μονάδες των διανυσματικών/παράλληλων αρχιτεκτονικών, πρέπει πρώτα να μεταφερθούν οι απαραίτητες τιμές στις τοπικές μνήμες (LM). Αυτό επιτυγχάνεται με συναρτήσεις από τις βιβλιοθήκες TTA, παρόμοιες με τη memcpy της C, αλλά με επιπλέον ορίσματα για το μέγεθος διανύσματος.

Η παραλληλοποιημένη συνάρτηση πολλαπλασιασμού πίνακα με διάνυσμα της μορφής $B[Y : X] \cdot A[X]$ οργανώνει τα στοιχεία της κάθε γραμμής του πίνακα (και του διανύσματος A) σε ομάδες των N (το μέγεθος των παράλληλων λειτουργικών μονάδων, συγκεκριμένα των vectorized FPU), μειώνοντας έτσι

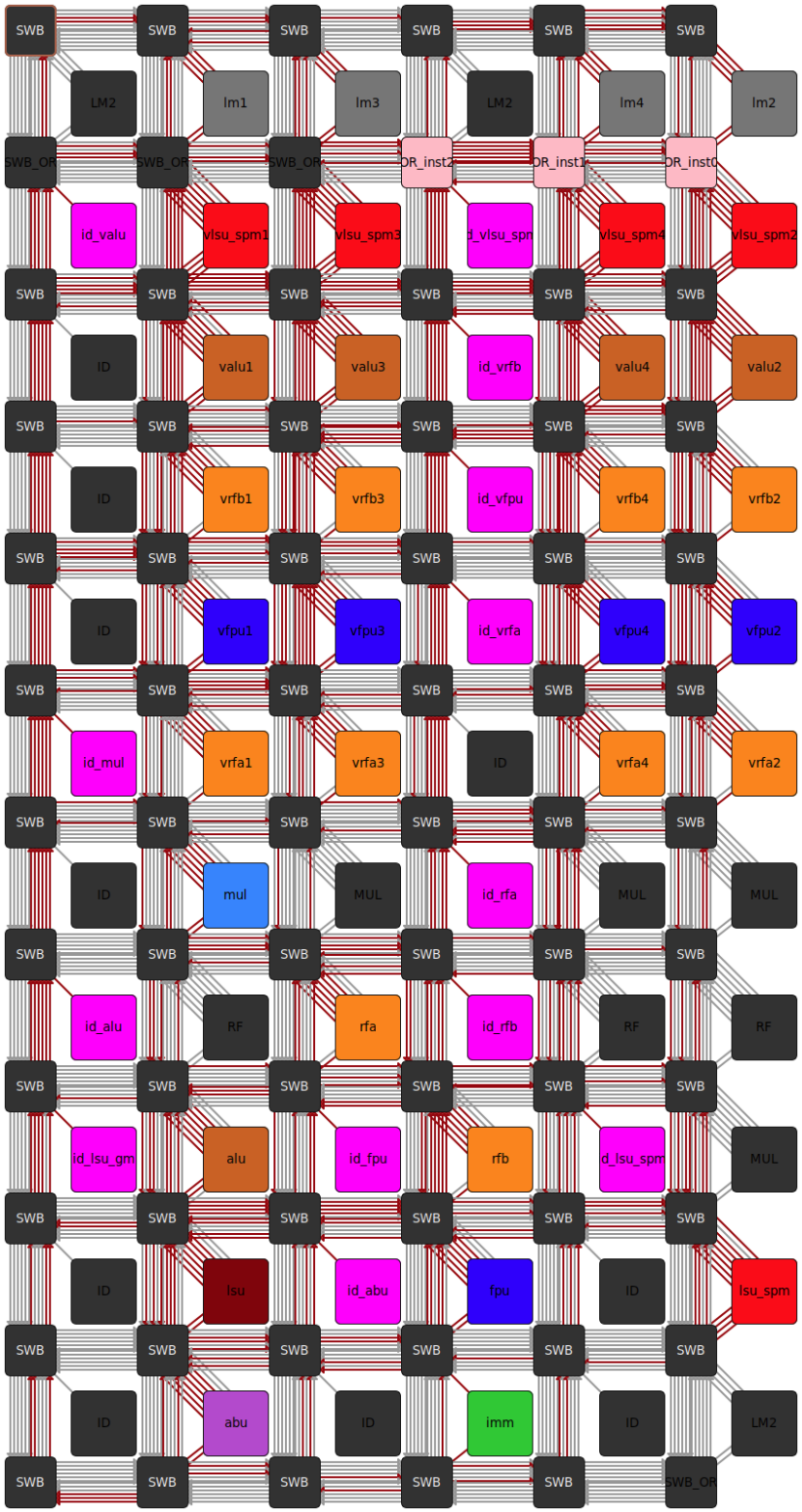


Figure 1.5.3: Αρχιτεκτονική Vector4

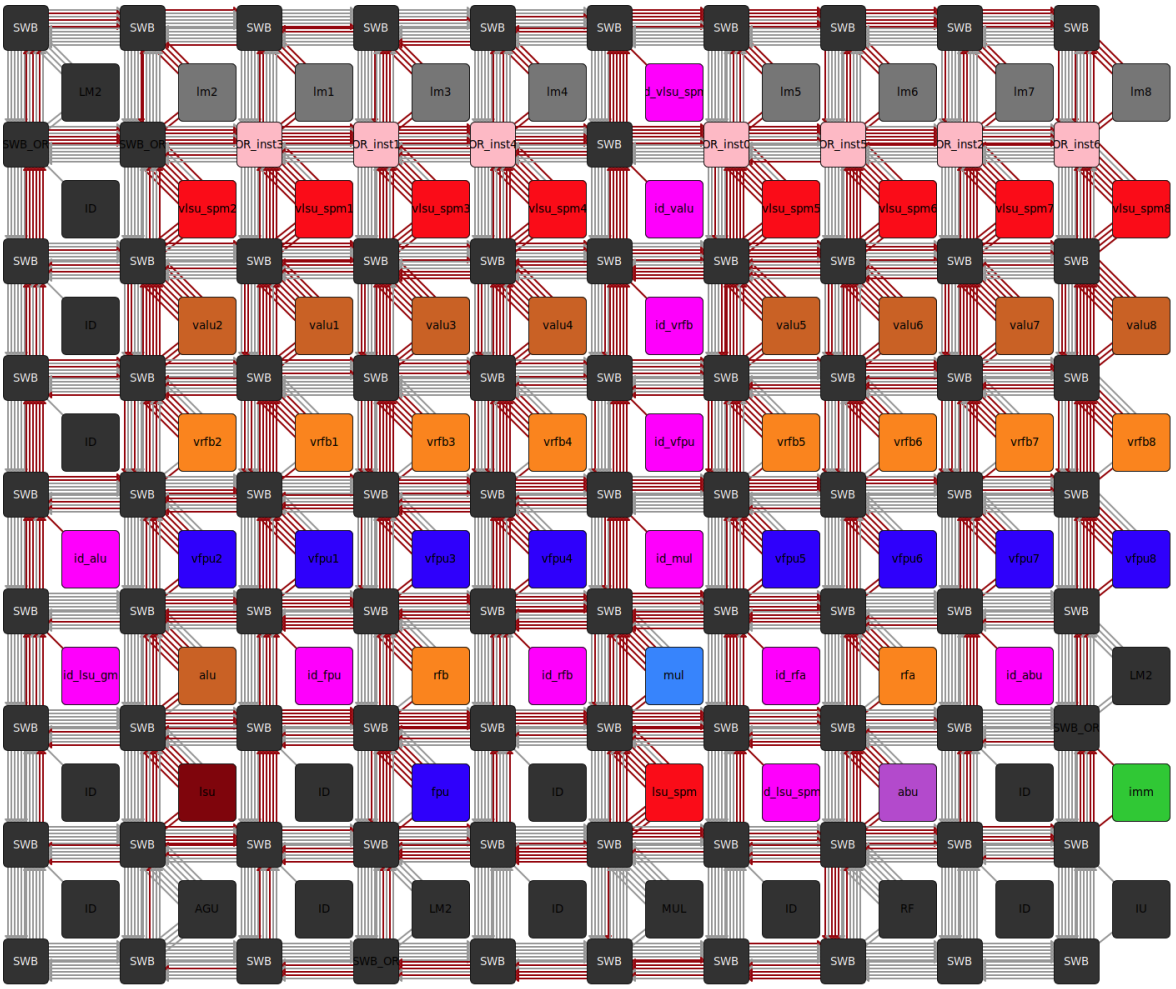


Figure 1.5.4: Αρχιτεκτονική Vector8

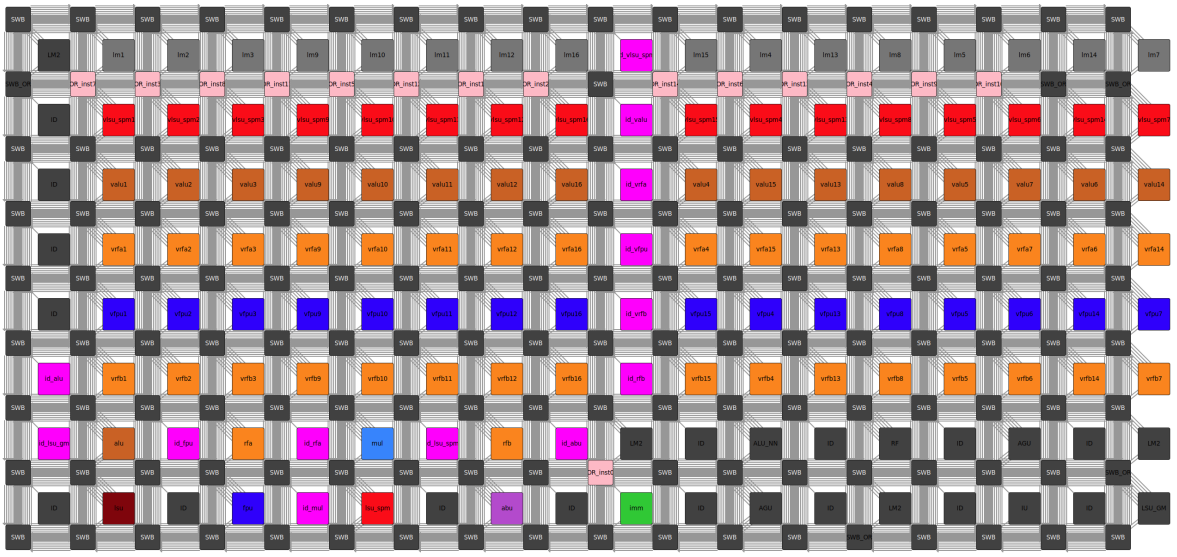


Figure 1.5.5: Αρχιτεκτονική Vector16

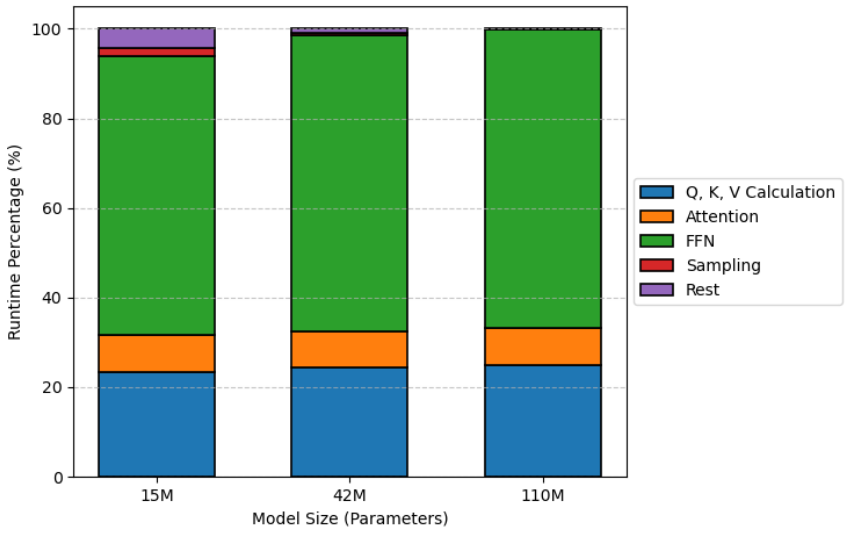


Figure 1.5.6: Ποσοστό χρόνου εκτέλεσης των υπολογιστικών μπλοκ του LLaMa2 σε CPU

το πλήθος των επαναλήψεων του εσωτερικού loop από X σε X/N φορές (Εικ. 1.5.7). Οι στοιχειώδεις πράξεις γίνονται παράλληλα και τα αποτελέσματα συσσωρεύονται τοπικά πριν τον τελικό υπολογισμό της γραμμής. Ο αλγόριθμος περιγράφεται και με ψευδοκώδικα παρακάτω:

```
// vector is stored in local memory address A
// matrix is stored in global memory address B
// performing the multiplication B[y:x]*A[x] with N vector lanes
// result is stored in global memory address res

parallel_matmul(int x, int y, int N) {
    for(i from 0 to y) {                          // for every row of matrix B
        memcopy(row, B + i*x, size n, GM_TO_LM);  // copy x elements from global
            // memory to local memory, storing them in address "row"

        vector row_buffer = {0.0};                // initialize buffer to 0
        for(j from 0 to x/N) {
            val += row[j] * A[j];                 // row[j] is a vector of N floats
        }

        float val = 0.0;
        for(k from 0 to N) {
            val += vector_extract(row_buffer,k);  // add up the final elements
                // of the buffer
        }
        res[i] = val;
    }
}
```

Για να διατηρηθεί το μέγεθος των LM χαμηλό, τα δεδομένα του πίνακα αποθηκεύονται στην LM γραμμή-γραμμή, αντί να φορτωθεί ολόκληρος ο πίνακας. Η βελτιστοποίηση αυτή είναι κρίσιμη, καθώς οι LM καταλαμβάνουν μεγάλο ποσοστό του τελικού εμβαδού του CGRA layout, και η μείωση του μεγέθους τους αποτελεί βασική προτεραιότητα στον σχεδιασμό.

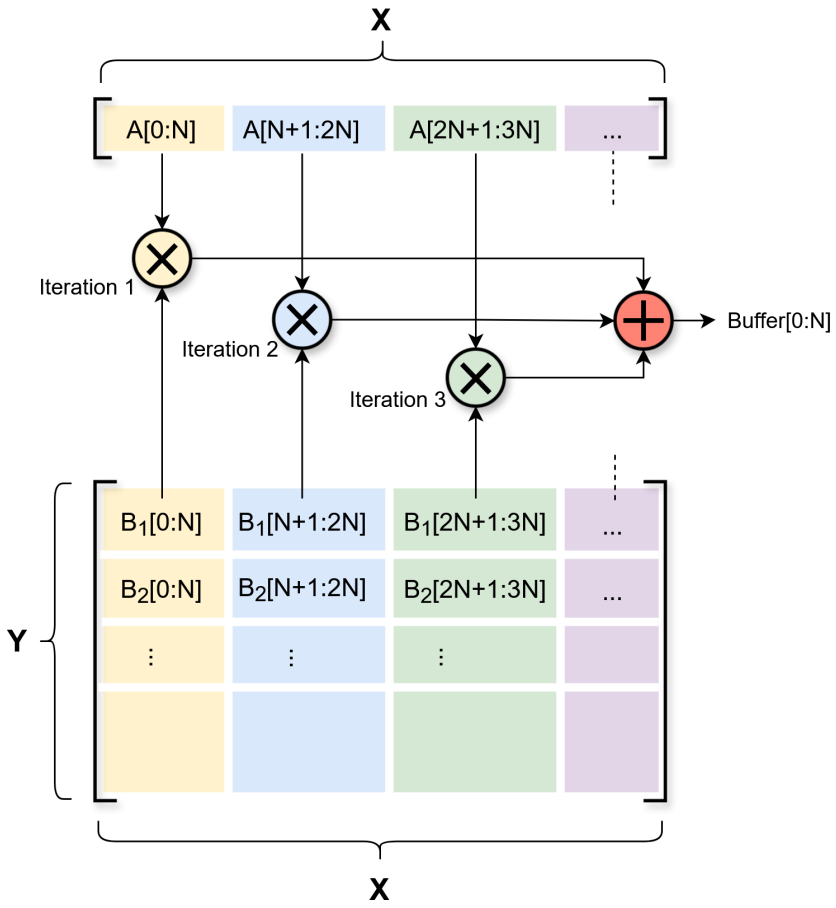


Figure 1.5.7: Οπτικοποίηση της συνάρτησης παράλληλου πολλαπλασιασμού πινάκων

1.6 Αποτελέσματα

Έχοντας εξηγήσει τη μεθοδολογία απεικόνισης κώδικα, παρουσιάζουμε τώρα τα αποτελέσματα από την εκτέλεση των πρώτων LLM benchmarks στο R-Blocks. Συγκεκριμένα, μετράμε τους κύκλους ρολογιού ανά εκτέλεση, καθώς και το εμβαδόν και την κατανάλωση ισχύος των κυκλωμάτων που προέκυψαν από τις καθορισμένες αρχιτεκτονικές. Τέλος, επιχειρούμε την ερμηνεία των αποτελεσμάτων μας, και την μοντελοποίηση της συμπεριφοράς του συγκεκριμένου συστήματος.

1.6.1 Ανάλυση Απόδοσης

Ο Πίνακας 1.6 παρουσιάζει τους κύκλους εκτέλεσης για τα τρία επιλεγμένα benchmarks σε κάθε μία από τις τρεις αρχιτεκτονικές. Οι πράξεις πολλαπλασιασμού πινάκων-διανυσμάτων κυριαρχούν χρονικά σε όλα τα benchmarks, και έτσι επιλέχθηκε να παρουσιαστεί και ένα καθαρό (χωρίς μη-γραμμικές συναρτήσεις ενδιάμεσα) matrix-vector multiplication, για σύγκριση. Τα αποτελέσματα προέκυψαν μέσω προσομοίωσης RTL, εκτός από την Vector16, όπου χρησιμοποιήθηκε ο cycle-accurate προσομοιωτής της OpenASIP, λόγω αδυναμίας δρομολόγησης (routing) της αρχιτεκτονικής.

Table 1.6: Ανάλυση Απόδοσης σε Κύκλους

Benchmark	Αρχιτεκτονική	Κύκλοι Ρολογιού (Χιλιάδες)	% Βελτίωση σε σχέση με Scalar
Υπολογισμός Q, K, V Πινάκων	Scalar	12.258	-
	Vector4	4.587	62,5
	Vector8	3.537	71,1
	Vector16	3.943	67,8
Μπλοκ Attention	Scalar	4.175	-
	Vector4	1.606	61,5
	Vector8	1.273	69,5
	Vector16	1.391	66,7
Πολλαπλασιασμός Πίνακα-Διανύσματος (1376x512)	Scalar	13.691	-
	Vector4	3.622	73,5
	Vector8	2.498	81,7
	Vector16	2.229	83,7

Όπως φαίνεται και στο διάγραμμα της εικόνας 1.6.1, η χρήση παραλληλίας μειώνει δραστικά τους κύκλους εκτέλεσης, ξεπερνώντας το 60% σε όλα τα benchmarks και το 70% στο καθαρό matrix-vector multiplication.

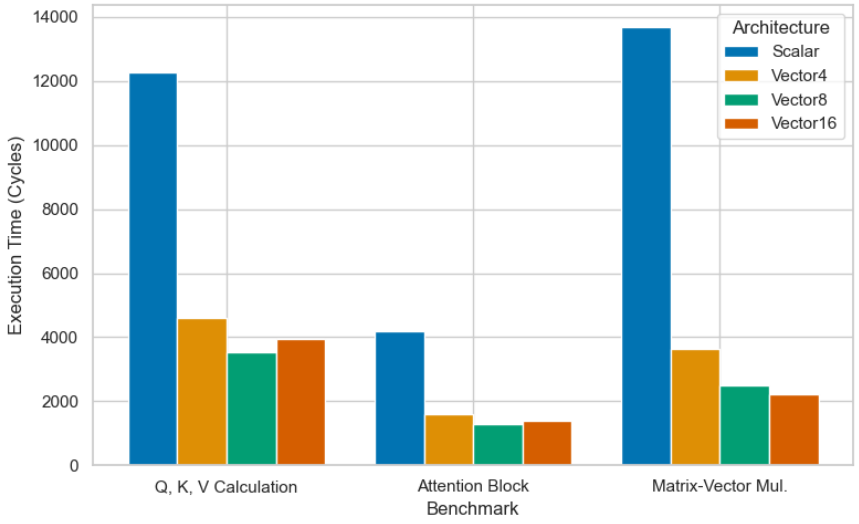


Figure 1.6.1: Κύκλοι εκτέλεσης ανά αρχιτεκτονική

Παρατηρείται όμως ότι ο μεγαλύτερος βαθμός παραλληλίας δεν οδηγεί πάντα στη βέλτιστη επιτάχυνση, κάτι που οφείλεται στην αναποτελεσματική διαχείριση της τοπικής μνήμης στις βιβλιοθήκες TTA, όπως

θα αναλυθεί παρακάτω.

1.6.2 Μοντελοποίηση Κύκλων Εκτέλεσης

Για τη μοντελοποίηση της χρονιστικής συμπεριφοράς του παράλληλου πολλαπλασιασμού, απεικονίστηκαν πολλαπλασιασμοί πινάκων διαφορετικών διαστάσεων στις διανυσματικές/παράλληλες αρχιτεκτονικές. Τα αποτελέσματα φαίνονται στην εικόνα 1.6.2, όπου αποτυπώνεται η μη γραμμική βελτίωση που προκύπτει από την αύξηση του μεγέθους του διανύσματος.

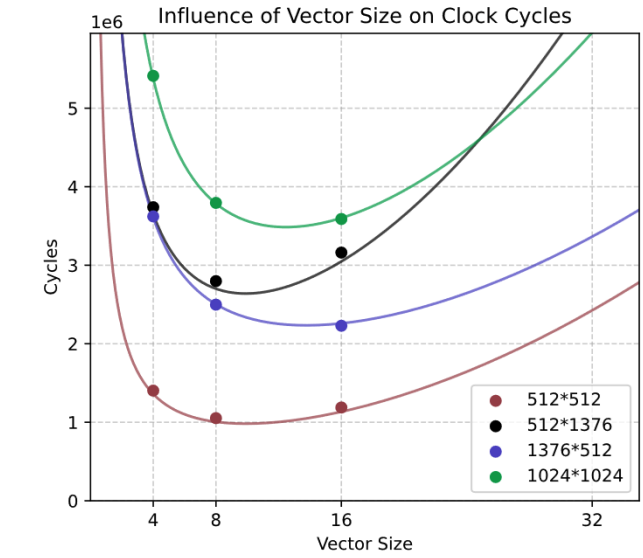


Figure 1.6.2: Επίδραση των διαστάσεων του πίνακα στην παραλληλοποίηση

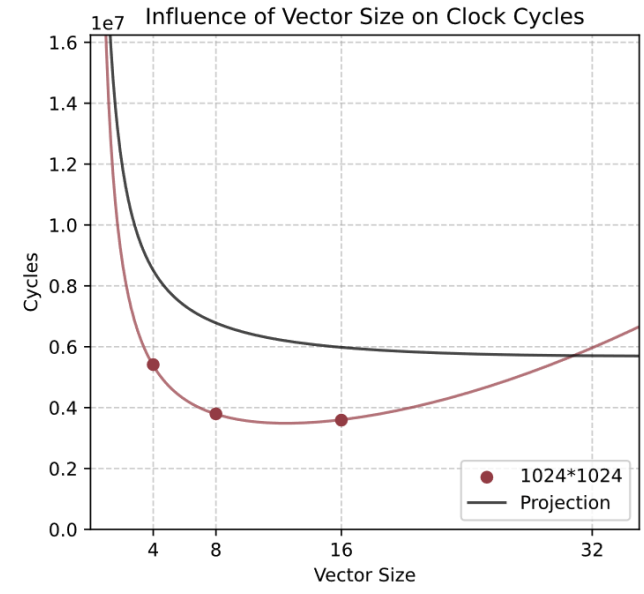


Figure 1.6.3: Προβλεπόμενη συμπεριφορά ιδανικού συστήματος

Από ανάλυση της συμπεριφοράς της συνάρτησης παράλληλου πολλαπλασιασμού που κατασκευάσαμε,

καταλήγουμε στην παρακάτω συνάρτηση που μοντελοποιεί τον αριθμό κύκλων, βασισμένη στις παραμέτρους M, G και E που αντιστοιχούν στον χρόνο πρόσβασης στην κύρια μνήμη, το χρονικό κόστος πολλαπλασιασμού και συσσώρευσης από τις vectorized FU, και στην τελική εξαγωγή των αποτελεσμάτων από την τοπική μνήμη για πρόσθεση:

$$Cost = Y \cdot (X \cdot M + \frac{X}{N} \cdot G + \frac{N^2}{2} \cdot E) \tag{1.6.1}$$

Η ανάλυση δείχνει ότι το βασικό πρόβλημα βρίσκεται στις προσπελάσεις στοιχείων στις τοπικές μνήμες. Η εν λόγω συνάρτηση έχει πολυπλοκότητα $O(N)$ (δημιουργώντας τελικά τετραγωνική εξάρτηση αφού καλείται για κάθε στοιχείο του vector) σε σχέση με τον βαθμό παραλληλίας N, κάτι που προκαλεί σημαντική επιβράδυνση.

Σε μία μελλοντική πιο αποδοτική υλοποίηση, η εξάρτηση της προσπέλασης από το N θα ήταν γραμμική, όπως φαίνεται στην Εικ. 1.6.3, όπου τα προβλεπόμενα αποτελέσματα δείχνουν σταδιακή βελτίωση χωρίς ξαφνική αύξηση κόστους, αλλά με σταδιακή μείωση του ρυθμού βελτίωσης όπως είναι λογικό.

Μετρικές Απόδοσης

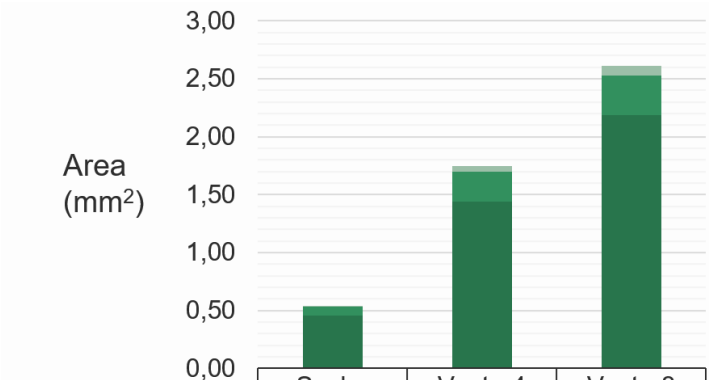
Οι αρχιτεκτονικές που μπόρεσαν να δρομολογηθούν σε επίπεδο RTL συντέθηκαν σε τεχνολογία 22nm FD-SOI με μέγιστη συχνότητα ρολογιού 200MHz.

Όπως φαίνεται στην εικόνα 1.6.4, μεγάλο μέρος της επιφάνειας καταλαμβάνουν οι μνήμες, συγκεκριμένα και οι μνήμες εντολών, αλλά κυρίως οι τοπικές, παρά τις προσπάθειες περιορισμού τους κατά την διαδικασία σχεδιασμού και απεικόνισης του κώδικα. Τα υπόλοιπα FUs κλιμακώνονται γραμμικά, με την FPU να μην επιφέρει σημαντικό overhead.

Στην κατανάλωση ισχύος, η εικόνα 1.6.5 δείχνει ότι η χρήση περισσότερων τοπικών μνημών αυξάνει δραματικά την κατανάλωση, ξεπερνώντας το 50% της συνολικής ισχύος στις διανυσματικές αρχιτεκτονικές. Αυτό είναι ένα πρόβλημα που μελλοντικά πρέπει να επιλυθεί αν το R-Blocks στοχεύει να γίνει πιο ενεργειακά αποδοτικό.

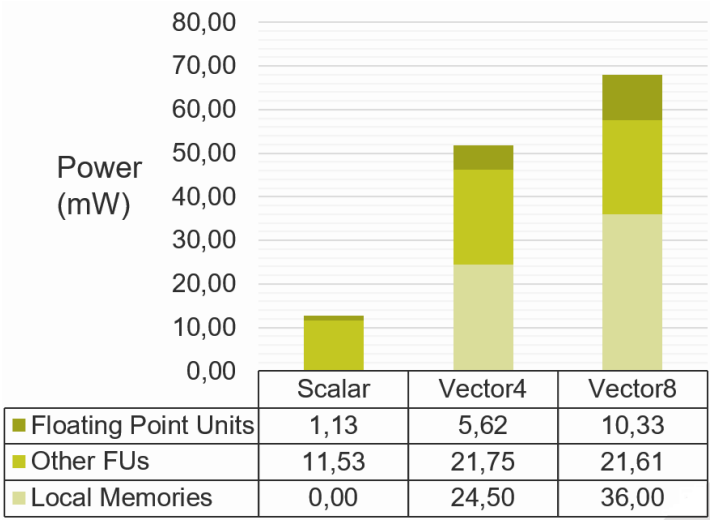
Από άποψη υπολογιστικών λειτουργικών μονάδων, όμως, παρατηρούμε ότι η FPU δεν επιφέρει σημαντικό overhead ούτε στην κατανάλωση ισχύος, τουλάχιστον σε σχέση με τις υπόλοιπες FU.

Δεδομένου, ακόμα, ότι ενδιαφερόμαστε για την ενεργειακή αποδοτικότητα του σχεδιασμού, παρουσιάζουμε και το Energy-Delay Product (EDP) ως σημαντική μετρική. Η εικόνα 1.6.6 δείχνει την τάση του EDP καθώς αυξάνεται ο βαθμός παραλληλίας.



	Scalar	Vector4	Vector8
Floating Point Units	0,01	0,05	0,08
Other FUs	0,0756	0,2587	0,3393
Memories	0,45	1,4383	2,187

Figure 1.6.4: Ανάλυση εμβαδού του τελικού κυκλώματος



	Scalar	Vector4	Vector8
Floating Point Units	1,13	5,62	10,33
Other FUs	11,53	21,75	21,61
Local Memories	0,00	24,50	36,00

Figure 1.6.5: Ανάλυση κατανάλωσης ισχύος του τελικού κυκλώματος

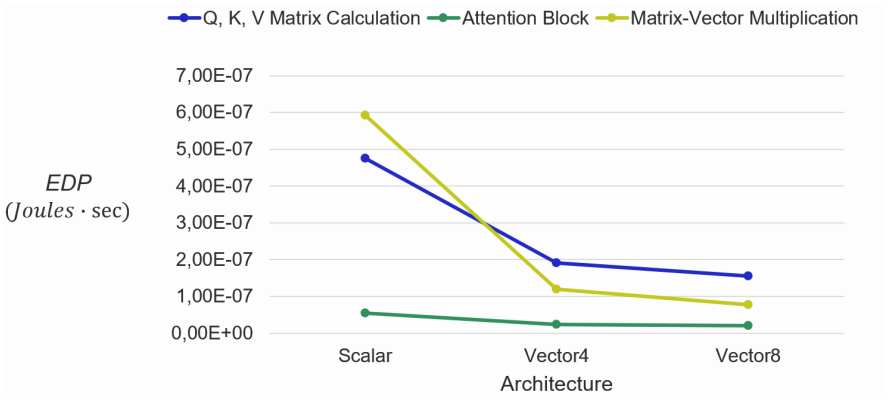


Figure 1.6.6: Ανάλυση του EDP ανά benchmark

Η μείωση των κύκλων είναι σημαντικά μεγαλύτερη από την αύξηση της κατανάλωσης ισχύος, οδηγώντας σε μείωση του EDP όσο αυξάνεται ο βαθμός παραλληλίας. Ωστόσο, μετά από ένα σημείο (Vector4), ο ρυθμός μείωσης πέφτει πολύ, καθιστώντας την περαιτέρω αύξηση της παραλληλίας μη αποδοτική, καθώς κοστίζει και σε εμβαδόν.

1.7 Επίλογος

Κλείνοντας την ελληνική σύνοψη αυτής της διπλωματικής, συνοψίζω τα βασικά ευρήματα για την απεικόνιση των LLMs στην πλατφόρμα R-Blocks, παρουσιάζοντας ξανά τις κυριότερες συνεισφορές μας και προτείνοντας μελλοντικές ερευνητικές κατευθύνσεις.

1.7.1 Επέκταση του περιβάλλοντος R-Blocks

Παρά το γεγονός ότι η δυνατότητα μεταγλωττισμού και εκτέλεσης LLMs (και άλλων εφαρμογών που βασίζονται σε αριθμητική FP) στο περιβάλλον R-Blocks αποτελεί σημαντικό ορόσημο, αναδεικνύονται και ορισμένα ζητήματα, όπως οι αναποτελεσματικότητες στις προσβάσεις στην τοπική μνήμη, οι περιορισμοί στο μέγεθος και την εκμετάλλευση των εξωτερικών μνημών και η ανάγκη βελτίωσης πολλών εργαλείων του περιβάλλοντος OpenASIP-R-Blocks (π.χ. του αλγορίθμου δρομολόγησης, του prunner κλπ).

Σε κάθε περίπτωση η προσπάθεια που έχει γίνει μέχρι εδω, έχει ρίξει φως στο ανεξερεύνητο πεδίο της επέκτασης του περιβάλλοντος R-Blocks, παρέχοντας μια μεθοδολογία που θα διευκολύνει μελλοντικές προσπάθειες στον τομέα.

1.7.2 LLMs σε CGRAs

Ο βασικός στόχος ήταν να εξεταστεί αν μια πλατφόρμα CGRA, όπως η R-Blocks, μπορεί να χρησιμοποιηθεί αποδοτικά ως επιταχυντής για εφαρμογές LLMs στο edge. Αν και σημειώθηκαν σημαντικές πρόοδοι, δεν μπορεί να δοθεί μια οριστική απάντηση. Τα CGRAs διαπρέπουν σε εφαρμογές ειδικού σκοπού που απαιτούν παραμετροποίηση κυκλωμάτων, αλλά οι GPUs παραμένουν η εύκολη επιλογή όσον αφορά τα LLM λόγω της εκτεταμένης έρευνας γύρω από αυτά και της υποστήριξης από την κοινότητα. Ωστόσο, η προσέγγιση του R-Blocks, που δίνει έμφαση στην εξαιρετικά χαμηλή κατανάλωση ενέργειας και στο HW/SW co-design, την καθιστά μια ενδιαφέρουσα εναλλακτική για εκτέλεση edge και για διάφορες πειραματικές εφαρμογές.

Η περαιτέρω έρευνα πάνω στην απεικόνιση LLMs στο R-Blocks προϋποθέτει την αντιμετώπιση βασικών περιορισμών του υλικού και των εργαλείων ανάπτυξης. Ιδιαίτερα κρίσιμη είναι η βελτιστοποίηση των προσβάσεων στις τοπικές μνήμες και η εξερεύνηση νέων αρχιτεκτονικών παραμέτρων που θα μειώσουν τα σημεία συμφόρησης στον υπολογισμό. Με την ενίσχυση των δυνατοτήτων προσομοίωσης και την άρση ορισμένων υφιστάμενων περιορισμών, μπορεί να επιτευχθεί καλύτερη διαχείριση του datapath και αύξηση της ενεργειακής αποδοτικότητας του συστήματος.

1.7.3 Approximate Computing

Μία ενδιαφέρουσα κατεύθυνση για μελλοντική έρευνα είναι η ενσωμάτωση approximate computing τεχνικών στην αρχιτεκτονική του CGRA. Αυτό θα μπορούσε να επιτευχθεί είτε με την εισαγωγή προσεγγιστικών λειτουργικών μονάδων (approximate functional units), όπως μειωμένης ακρίβειας πολλαπλασιαστές, είτε μέσω αλγοριθμικών τεχνικών που επιτρέπουν χαμηλότερη κατανάλωση ενέργειας με αποδεκτή επίπτωση στην ακρίβεια των υπολογισμών. Προκαταρκτικά αποτελέσματα από εσωτερικές έρευνες στο Microlab, NTUA δείχνουν ότι τέτοιες τεχνικές έχουν σοβαρές προοπτικές για εφαρμογές όπου ένα μικρό σφάλμα δεν επηρεάζει τη συνολική λειτουργικότητα.

1.7.4 Εξερεύνηση Αρχιτεκτονικών

Η ευελιξία του R-Blocks επιτρέπει την εξερεύνηση διαφορετικών αρχιτεκτονικών επιλογών. Στην παρούσα εργασία, η έμφαση δόθηκε κυρίως στην αύξηση του βαθμού παραλληλίας μέσω διανυσματικών μονάδων, αλλά παραμένουν αναπάντητα ερωτήματα σχετικά με την βελτίωση του δικτύου διασύνδεσης (interconnect) και την κατανομή των scalar και vector μονάδων για τη μέγιστη απόδοση. Η αυτοματοποίηση αυτής της διαδικασίας, μέσω εργαλείων όπως το prunner, θα μπορούσε να οδηγήσει σε καλύτερα ισορροπημένες αρχιτεκτονικές που λαμβάνουν υπόψη ακόμα και περιορισμούς σε κατανάλωση ενέργειας, εμβαδόν και συνολική αποδοτικότητα του τελικού κυκλώματος.

1.7.5 Τελικές Σκέψεις

Παρά τις υπάρχουσες δυνατότητες μεταγλώττισης του TTA compiler του R-Blocks, υπάρχουν σημαντικά περιθώρια βελτίωσης. Πειραματικά δεδομένα έδειξαν ότι χειροκίνητα βελτιστοποιημένος κώδικας assembly μπορεί να επιτύχει έως και 2 φορές επιτάχυνση σε σχέση με τον αυτόματα παραγόμενο κώδικα. Οι μελλοντικές έρευνες θα πρέπει να επικεντρωθούν στη βελτίωση του scheduler και των βιβλιοθηκών χειρισμού του dataflow της OpenASIP, ώστε να αξιοποιηθούν πλήρως οι δυνατότητες παραλληλίας της αρχιτεκτονικής και να μειωθούν οι περιττές καθυστερήσεις κατά τη μεταφορά δεδομένων.

Η παρούσα εργασία αποτελεί μόνο ένα πρώτο βήμα στην προσπάθεια κατανόησης των δυνατοτήτων των CGRAs για επιτάχυνση LLMs. Όσοι επιθυμούν να συνεχίσουν την έρευνα σε αυτό το πεδίο θα βρουν ένα εξαιρετικά ενδιαφέρον αντικείμενο με πολλές ανοιχτές προκλήσεις. Κλείνοντας, θα ήθελα να εκφράσω και πάλι τις θερμές μου ευχαριστίες προς τα μέλη του Microlab, NTUA για την υποστήριξη και τις πολύτιμες γνώσεις που μοιράστηκαν μαζί μου καθ' όλη τη διάρκεια αυτής της εργασίας.

Chapter 2

Introduction

Ever since the publication of the groundbreaking paper "Attention Is All You Need" [1], generative transformers utilizing the attention mechanism have taken over the field of artificial intelligence and more specifically natural language generation. Initially we saw models such as GPT3 [2] and LLaMA [3] dominate the field, with more competitors rapidly entering the scene as the technologies surrounding large language models (LLMs) advanced. These models can answer user questions about seemingly anything while being not only coherent but also informative, intelligent and creative. The latest language models, additionally utilizing reinforcement learning [4] can even perform complex reasoning tasks at a near-human level.

Transformers utilizing the attention mechanism have not only propelled advances in the field of computer science such as in computer vision [5], speech recognition [6] and natural language processing [25], but in many other scientific domains too. In structural biology, for instance, transformer-based models have been instrumental in the development of methods for 3D protein structure prediction. AlphaFold [7], an attention based model, leverages this mechanism to infer three-dimensional structures of proteins from their amino acid sequences, and has profound implications for drug discovery and our understanding of molecular function. By integrating such sophisticated architectures, researchers are now better equipped to explore the complex interplay between sequence and structure, paving the way for innovations in bioinformatics and beyond.

More effort is currently being put into leveraging this newfound technology to facilitate advances in the fields of material science [26], physics [27] and molecular science [28]. It is therefore reasonable to assume that transformer-based models are going to be around for a long time and are going to shape the way we think about many scientific fields.

Large generative transformer models, however, require vast amounts of computational resources. Traditionally this refers to large GPU arrays as they are commercially available, optimized in terms of hardware and software support and can accelerate various model architectures with minimal programmer effort. This unprecedented demand for training and inference hardware is fueling research into alternative hardware architectures fitted for such applications.

At the same time, a resurfacing hardware architecture is being evaluated in different application domains. Coarse-grained reconfigurable architectures (CGRAs) are, in essence, a middle ground between the flexibility of FPGAs and the performance and efficiency of ASICs [8]. Instead of catering to the programmer's general needs, as FPGAs do, they instead focus on application domain-specific tasks by being reconfigurable in a more coarse-grained fashion. Instead of bit-level flexibility they traditionally provide pre-designed, efficient components that can be flexibly interconnected and used to execute specific tasks with higher performance, reduced reconfiguration overhead, lower power consumption and better area utilization compared to their fine-grained counterparts. There are also other reconfigurable architectures that can support the execution of virtually any program, while maintaining their flexibility and high energy efficiency.

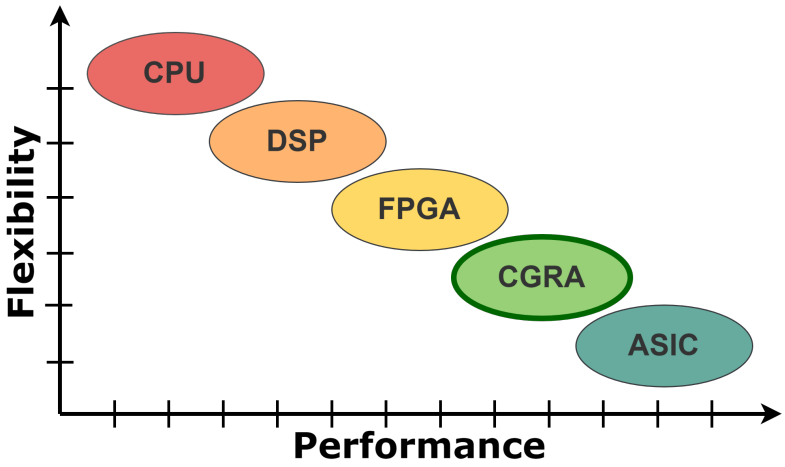


Figure 2.0.1: Flexibility and Performance

One such architecture is R-Blocks [9], the successor of the Blocks CGRA [11]. R-Blocks is an ultra low power CGRA using the VLIW-SIMD execution model and leveraging the OpenASIP toolset [10] for transport triggered architectures (TTA) to provide the programmer a complete software/hardware co-design flow. This means that high-level code can easily be compiled and mapped to the reconfigurable hardware, and that multiple architectures can be explored relatively quickly.

The basic idea and motivation behind this thesis, is to evaluate R-Blocks as an LLM inference acceleration edge device. The scope of this research, as anyone can understand is very wide and requires a vast array of theoretical and practical knowledge, as well as access to computational resources and tools beyond a single student's reach. This is why I'm extremely grateful to have the full cooperation of the Convolve project team of Microlab, NTUA who helped me through every step of the process.

The starting point for this endeavor was the basic R-Blocks architecture, with its accompanying toolchain for compilation, high-level simulation, architectural DSE and much more. What was missing from this full-stack workflow is a method for handling floating point operations. R-Blocks as a platform has been designed with simple instruction sets and low power in mind, and based on the earlier Blocks iterations, it completely ignores floating point operations. But given that LLMs mostly perform multiplications involving matrices of floating point numbers, we have to find a way to compute them in R-Blocks.

We make the decision to expand R-Blocks and the entire end-to-end compilation toolchain to support single precision floating point numbers. We do that by creating a new Functional Unit according to the R-Blocks architecture, that incorporates an open source floating point unit (FPU) [23] and its own custom instruction set. With the instructions included in this small instruction set, after expanding the compilation tools to support it, we manage to compile any floating point operation for execution on the reconfigurable accelerator, utilizing high-level libraries (e.g. math.h).

After successfully implementing this expansion, we manage to map the entire LLM token generation function for the first time on R-Blocks. However, our contribution doesn't stop there. We looked into vectorization options -connecting Functional Units to the same instruction decoder for parallel execution- and evaluated how they impact performance. We assigned different sections of the target LLM inference code to programmer-defined architectures that use vectorized FPUs. This allowed us to gather experimental data on their efficiency and model the effects of various parallelization strategies on the system.

These architectures were synthesized on a 22nm FD-SOI technology using the Synopsys Design Compiler, with a maximum clock frequency of 200MHz, to gather data on the area and power of the resulting circuits.

A brief overview of the results, points towards promising capabilities in terms of power efficiency and parallelization capabilities, however the issues surrounding the CGRA as technology and R-Blocks as a platform, enforce limitations around scaling and unlocking the true potential of such an architecture.

More in depth analysis of the results follows in chapter 7, while the expansion of the platform and the mapping of the first LLM benchmarks will be discussed in chapters 5 and 6. However, we have to start with a foundational theory background regarding both CGRAs as an architectural paradigm and LLMs as an emerging technology.

Chapter 3

Theoretical Background

In this chapter, we present the fundamental concepts underpinning this work. We introduce the principles of coarse-grained reconfigurable arrays (CGRAs), and provide an overview of the current state of this research domain. Following this, comes a more detailed explanation of R-Blocks, the specific reconfigurable architecture used in this work. Next we delve on transformer architectures and their role in modern artificial intelligence, focusing particularly on large language models (LLMs). Finally, we discuss Meta's Llama2 as a concrete example of a modern commercial LLM, as this is the application we will focus on accelerating.

3.1 Coarse-Grained Reconfigurable Architectures (CGRAs)

3.1.1 General Theory on CGRAs

Coarse-grained reconfigurable arrays (CGRAs) represent a pivotal evolution in the field of reconfigurable computing. Their development emerged from the need to bridge the gap between the fine-grained flexibility of field-programmable gate arrays (FPGAs) and the high performance and energy efficiency characteristic of application-specific integrated circuits (ASICs). In the early explorations of reconfigurable computing during the late 1980s and early 1990s, FPGAs were widely adopted due to their unparalleled adaptability at the gate level. However, the inherent granularity of FPGAs often resulted in significant performance overheads and power inefficiencies, particularly when scaling to complex computational tasks [8].

In response to these challenges, early research in the 1990s and 2000s [29, 30] began to investigate architectures that operated at a higher level of abstraction. By grouping several basic processing elements into larger, more efficient functional units, researchers laid the groundwork for CGRAs. These architectures were designed to offer a compromise: they maintained a degree of reconfigurability while simultaneously achieving higher computational density and better energy efficiency compared to traditional FPGAs.

Over the subsequent decades, CGRAs have resurfaced as a new contender that could tackle the world's increasing hardware demands. Advances in design methodologies, interconnect strategies, and reconfiguration techniques, as well as software compatibility, some of which will be explored later, have transformed these early experimental systems into versatile platforms capable of accelerating a wide spectrum of computational tasks.

Definition

The definition of a CGRA, according to the most comprehensive survey done on the topic, by Liu et al. [12] is: A computing fabric that has the following characteristics:

- Domain-specific flexibility
- Combining spatial and temporal computation
- Configuration- or data-driven execution

Unpacking this definition it becomes apparent that CGRAs are not designed to accelerate -or in some cases even compute- a solution for every possible programming problem. Instead, these architectures deliberately sacrifice universal flexibility, in order to achieve high efficiency within a **specific application domain**. This specialization is the source of their energy efficiency, as they can incorporate highly optimized, pre-designed hardware units that outperform the fine-grained complexity of their counterparts. However, this also implies that the advantage of CGRAs in general purpose computing is typically limited.

Efficient data utilization in both spatial and temporal dimensions is another defining characteristic of CGRAs. By offloading much of the scheduling responsibility to the programmer, scheduler, and compiler, CGRAs can effectively leverage VLIW and SIMD computation (i.e., spatial concurrency) -capabilities that are often beyond the reach of conventional general-purpose processors (GPPs). Additionally, the coarse granularity of CGRAs enhances area efficiency compared to the spatial computation observed in FPGAs.

Finally, the configuration- or data-driven execution model employed by CGRAs signifies that they do not adhere to the static sequential execution models dictated by compilers, as seen in GPPs. Instead, they operate according to their configuration and the availability of data at each processing element (PE). This approach, however, places increased demands on the scheduler and compiler, which must be capable of managing the complex and intricate structures inherent in these architectures.

Classification

A multidimensional taxonomy proposed by Liu et al. [12] classifies CGRAs according to their:

- Programming model
- Computation model
- Execution model

At the programming model level, CGRA systems can be broadly classified into three categories:

1. **Imperative Programming Models:** In this category, computations are expressed as a fixed, ordered sequence of statements—as seen in languages such as C/C++. Although this model does not inherently expose parallelism, custom compiler support (or programmer efforts) can enable explicit parallel constructs. Imperative programming is widely used because of its familiarity and seamless integration with general-purpose processors. For example, many CGRAs are programmed imperatively, where the device's operations are controlled via a predetermined configuration sequence [31, 32].
2. **Parallel Programming Models:** This category encompasses models that allow the explicit or implicit expression of parallelism. On one end, declarative models (e.g., functional or dataflow languages, and hardware description languages) let programmers specify computation logic without dictating control flow, thereby implicitly capturing parallelism. On the other end, concurrent imperative models -such as those using OpenMP, MPI, or CUDA C- employ directives to explicitly expose parallelism. These models reduce the burden on compilers and hardware by letting the programmer indicate which parts of the algorithm can be executed concurrently [15].
3. **Transparent Programming Models:** Here, no static compilation is performed for a specific CGRA architecture. Instead, dynamic compilation techniques generate configurations at runtime, based on common program representations like instruction streams. Systems such as DORA [33] and CCA [34] illustrate this approach, where runtime optimization allows the hardware to adapt dynamically. Although this improves programmer productivity and can optimize

based on live data, it typically incurs a performance and energy overhead due to the additional hardware required for runtime parallelism management.

Moving to a lower level, the computation model characterizes how an application's computational tasks are structured and executed on a CGRA. While all CGRAs fundamentally follow a Multiple Instruction, Multiple Data (MIMD) paradigm [35], a configuration-based classification provides a more nuanced view:

1. **Single Configuration, Single Data (SCSD):** In the SCSD model, a single configuration is executed on a single dataset. This approach is a pure spatial computation method that extracts maximum instruction-level parallelism by mapping all operations of a kernel onto the hardware. For instance, the Pegasus intermediate representation [36] paired with the application specific hardware (ASH) microarchitecture [37] template exemplifies this model—though its scalability is inherently limited by the hardware size.

2. **Single Configuration, Multiple Data (SCMD):** The SCMD model extends spatial computation by applying one configuration concurrently across multiple data sets, akin to SIMD or SIMT paradigms. This model is especially suited for stream-oriented or vector applications (e.g., multimedia or digital signal processing, and more recently AI applications), where data-level parallelism is dominant. All threads execute the same configuration simultaneously, each working on a different portion of the data.

3. **Multiple Configuration, Multiple Data (MCMD):** The most flexible model, MCMD, allows multiple configurations to be executed concurrently on different datasets. This model supports both simultaneous multithreading (SMT) and temporal multithreading (TMT), and is well-suited for scenarios requiring thread-level parallelism. Communication between threads is typically managed via message passing or shared memory. Sub-categories such as the Communicating Sequential Processes (CSP) model [38] -where processes synchronize through blocking message channels- and the Kahn Process Network (KPN) model [39] -using asynchronous FIFO buffers- illustrate different approaches to handling inter-thread communication.

The execution model is the lowest layer, and it focuses on the mechanisms for scheduling and executing the configurations on a CGRA. Two primary aspects characterize this model:

1. **Configuration Scheduling:** This aspect pertains to how configurations are fetched from memory and mapped onto the hardware. For example, FPGAs often rely on a static scheduling approach where compilers determine the fetching order and placement of configurations. In contrast, superscalar processors might employ dynamic scheduling based on real-time conditions, such as resource availability or predicate evaluation.

2. **Operation Execution:** Once scheduled, the manner in which operations within a configuration are executed can follow either a sequential or a dataflow paradigm. In sequential execution, operations are performed in a fixed order as determined at compile time. Alternatively, dataflow execution allows operations to be executed as soon as their operands become available. This can be implemented as either:

    - **Static Dataflow Execution:** Operations execute when data tokens with matching tags are ready, but only a single instance of an operation may run at a time.

    - **Dynamic Dataflow Execution:** Multiple instances of the same routine can execute concurrently as soon as their data dependencies are met, offering greater flexibility at the cost of increased hardware complexity.

Applied to a specific CGRA use case, this taxonomy can help the reader understand and appreciate the design decisions made during both the architecture design phase and the code mapping process in the application implementation.

One huge aspect of CGRAs is also their microarchitecture, but since it can differ so much in terms of data path granularity, reconfigurable logic function, network/interconnect topology, memory hierarchy,

operation scheduling, reconfiguration mechanism, custom operations, coupling and resource sharing with the host, it might be better examined in a practical scenario rather than in abstract theory.

The following analysis of a real CGRA system relies upon the previously proposed taxonomy to position the specific architecture within the broader design space, and to set the foundation for a systematic evaluation of its performance, scalability, and overall suitability for targeted applications.

3.1.2 R-Blocks

R-Blocks [9] is a CGRA developed by the Eindhoven University of Technology in a joint effort with Tampere University, as an ultra low power (ULP), reconfigurable and fully programmable hardware solution for application acceleration and also general purpose computing. It comes with a comprehensive software toolchain built upon the OpenASIP tool suite [10], which encompasses compilation tools, operation scheduling support, architectural design space exploration frameworks, and software simulators, among other components. This full-stack approach empowers programmers to map high-level code onto custom-designed architectures, thereby facilitating the efficient exploration of a wide range of configurations.

Following is a concise presentation of the entire R-Blocks architecture, but to understand the full scope of the work, the reader is encouraged to refer to the cited paper.

Architecture

The R-Blocks' physical architecture is a grid of programmable functional units (FUs) and a reconfigurable interconnect with two switchbox networks, one for data and one for control. The interconnect remains static during program execution for minimal recofiguration overhead.

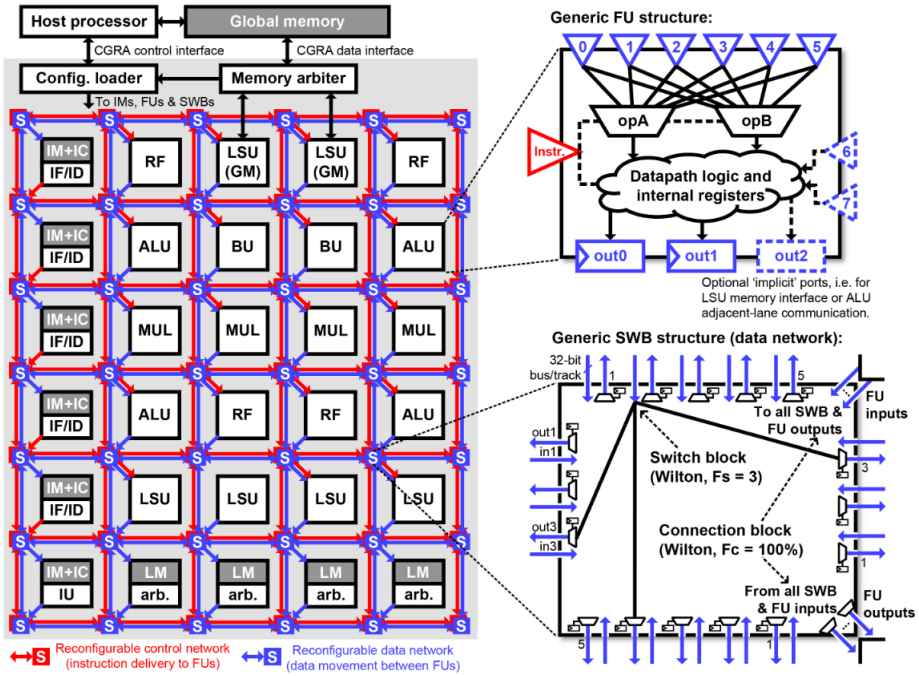


Figure 3.1.1: An example R-Blocks system consisting of a reconfigurable grid, global memory and a host processor. On the right, the internal structure of a functional unit and a Wilton switchbox.

The FUs are programmed using instruction decode (ID) units that can connect to one or multiple FUs of the same type, therefore enabling the SIMD execution model. The hardware of these FUs is designed separately but uniformly, aiming at maximum power efficiency. The types of functional units included in the original R-Blocks work are:

- Instruction Decoder (ID)
- Immediate Unit (IU) for immediate operand generation
- Load-Store Unit (LSU) handling the main memory transactions
- Register File (RF) for storing operands
- Arithmetic and Logical Unit (ALU)
- Accumulate and Branch Unit (ABU)
- Multiplier (MUL)
- Address Generation Unit (AGU)
- Local Memories (LM) that store the operands for vectorized operations close to the respective FUs, so they can be fetched simultaneously

Having heterogeneous types of FUs (and therefore IDs) allows for opcode reuse, keeping instruction words small across the entire device. R-Blocks is designed with the goal of allowing the processor designer to add functional units and operations as they see fit. This process is semi-automated, as long as the added instructions are supported by the LLVM compilation back-end.

The CGRA is also connected to the main memory outside of the reconfigurable fabric, with the LSU tile facilitating their communication. A typical R-Blocks system is depicted in Figure 3.1.1 as an example.

It should finally be mentioned that while R-Blocks is designed primarily as an accelerator tethered to a host processor, its capabilities in terms of flexibility and code adaptability allow it to sometimes act as a standalone core in specific applications or application domains.

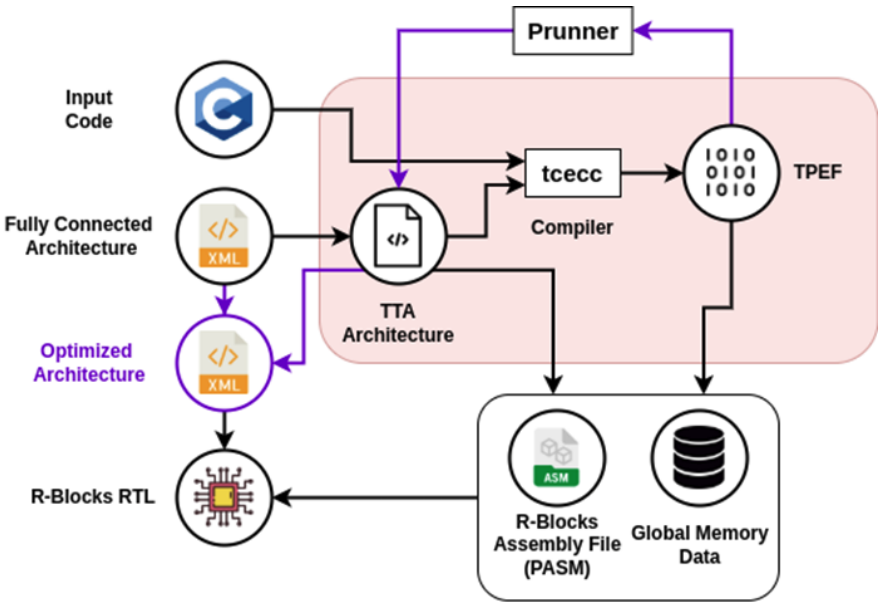


Figure 3.1.2: R-Blocks compilation flow

Software Tool Support

Modeled as a Transport Triggered Architecture (TTA) [13], R-Blocks is capable of leveraging the full suite of OpenASIP tools [10] in addition to custom utilities developed specifically for R-Blocks (Fig. 3.1.2.

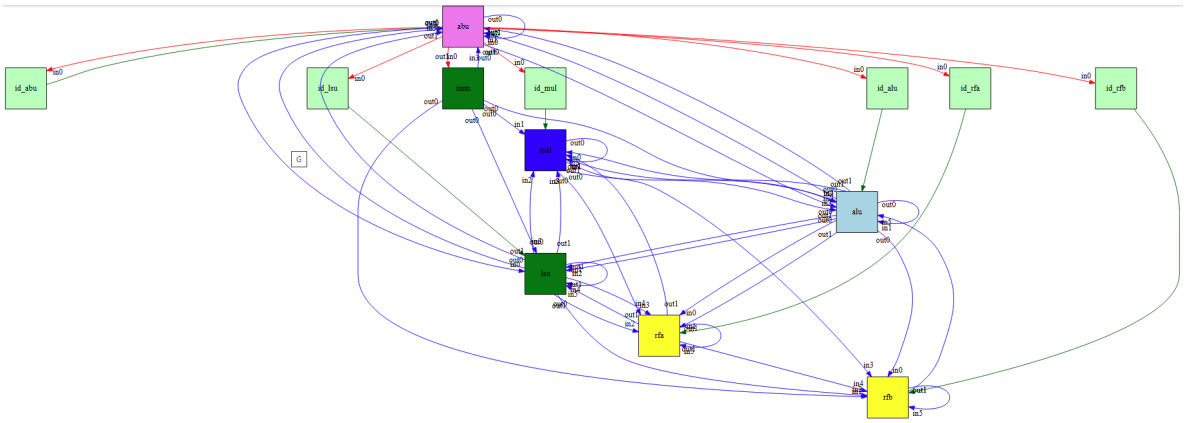


Figure 3.1.3: Example of a virtual R-Blocks architecture

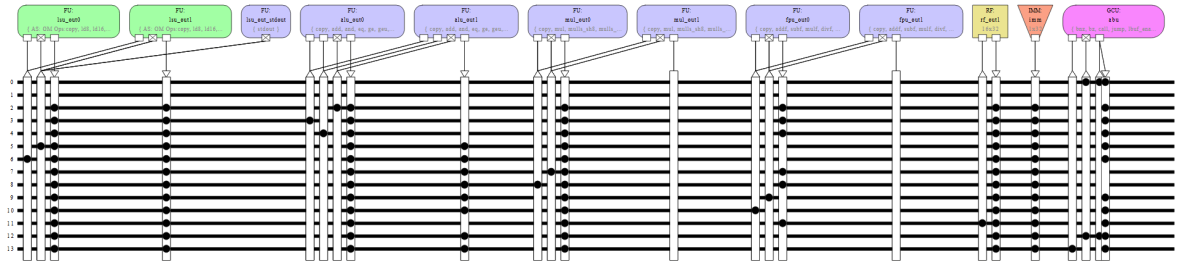


Figure 3.1.4: Virtual architecture after being translated to TTA, and viewed using the ProDe GUI

In general, TTA is a processor design paradigm in which programs exert direct control over the internal transport buses, and computation occurs as a side effect of data transfers: writing data into a functional unit's triggering port initiates the corresponding computation. In R-Blocks, the instruction memories contained within the instruction decoders (IDs) are transmitted to the FUs via the instruction interconnect network, while operands and computation results are routed over the data interconnect network rather than going through a centralized memory or register file. Although there is, as previously noted, a dedicated register file FU, it is not utilized for every data transfer.

To integrate a virtual R-Blocks architecture (Fig. 3.1.3) within the OpenASIP ecosystem, it must be described as a TTA architecture. The architecture translator tool performs the necessary adjustments, converting an R-Blocks virtual architecture file (XML) into a TTA architecture description file (ADF) (Fig. 3.1.4).

Subsequently, the OpenASIP TTA compiler -leveraging its complete LLVM back-end- compiles and schedules the programmer's high-level code for this specific TTA. At this stage, the TTA device can be simulated using tools such as OpenASIP's ttasim (CLI) and proxim (GUI), which provide a comprehensive debugging interface that includes memory content views, instruction stepping, and detailed overviews of functional units and bus activity.

Before the executable is ready for deployment on a target virtual architecture, the programmer has the option to optimize the architecture, particularly if it is deemed overly complex for direct hardware mapping. This optimization is achieved using the pruner tool, which iteratively prunes connections between tiles in the original architecture description to identify an energy-optimal configuration based on specified constraints and data transfer power estimations.

Once the TTA executable is compiled, the parallel assembly for the R-Blocks virtual architecture can be extracted, and assembler utilities produce the corresponding binaries for program execution.

Finally, the blocks instantiator scripts automatically generate the synthesizable hardware of the virtual

architecture -based on the architecture description XML file- allowing the binaries to be loaded and executed.

It is important to note that we extended many of these tools and utilities to include support for floating-point arithmetic, an enhancement that was not part of the originally published work.

Classification

Refering back to the taxonomy section, and starting from the lowest level, the execution model of R-Blocks consists of VLIW-SIMD cores (mainly FUs) that support software bypassing [40]. This feature allows the compiler, in certain cases, to bypass the use of register files by directly transferring operands between FUs. Additionally, the virtual architecture designer has the flexibility to instantiate an arbitrary number of FUs connected by a custom datapath. Such a processor is classified as an exposed datapath architecture (EDPA) [41].

This means that, in terms of configuration scheduling, R-Blocks adheres to a conventional, static approach, while introducing innovations in the domain of operation execution,by employing dynamic dataflow execution with specific optimizations.

With respect to the computation model, R-Blocks follows the single configuration multiple data (SCMD) paradigm. It leverages the SIMD execution model to achieve data-level parallelism without introducing multiple concurrent configurations, thereby preserving both flexibility and ease of programming.

At the highest level, R-Blocks utilizes the parallel programming model, more specifically a concurrent imperative model. The target application is developed in high-level C code and subsequently compiled for the custom architecture using the aforementioned tool-flow. Simultaneously, the designer retains the option to define the target virtual architecture, while also being able to rely on a default or domain-specific configuration, and can fine-tune it down to the dataflow level according to the application's requirements.

Concluding the section on R-Blocks, it is important to emphasize that this architecture was designed with power efficiency as its primary goal, while also offering remarkable flexibility in both in terms of architectural design and application domain. Its inherent SIMD execution model excels at leveraging data-level parallelism, which makes it well-suited for handling computationally demanding tasks. These attributes are what inspired us to explore the feasibility of running large language model inference on R-Blocks. LLMs are inherently resource-intensive and stand to benefit from an architecture that can manage extensive parallelism without compromising on energy consumption. Ultimately, our investigation aims to reveal the potential of R-Blocks as an efficient platform for emerging AI workloads, expanding its relevance beyond traditional, power-hungry computing systems.

There is, however, one big problem when porting LLM code to R-Blocks: The hardware has zero support for floating point arithmetic. Because floating point numbers are at the core of transformers (as will be discussed below) and simple FP emulation methods could not be employed efficiently at a hardware level, the decision was made to expand the entire tool-flow and hardware in unison, with a floating point unit (FPU) and full end-to-end support for FP arithmetic. This expansion led us to a deep exploration and understanding of the R-Blocks ecosystem and what it has to offer, and will be the object of chapter 5 of this thesis.

3.2 Transformer Architectures and Large Language Models

In this section we give the theoretical background about transformer models, LLMs and finally Meta's Llama2, required to understand the methodology section of our work.

3.2.1 Transformer

Transformer architectures have emerged as a paradigm shift in deep learning, largely replacing traditional recurrent and convolutional models in many natural language processing tasks. Introduced by [1], the key innovation of transformers is the *self-attention mechanism*, which allows the model to weigh the importance and correlation of different input tokens dynamically. This is in contrast to sequential models that process input data strictly in order, thereby limiting their ability to capture long-range dependencies.

The input of the transformer is a sequence of tokens. A token in the case of natural language processing (NLP) can be a word (or, more prominently, a subword), in the case of images or video can represent sub-images or objects, and in the case of audio can represent a sound signature that corresponds to a letter or a particular sound. This sequence is passed through a variation of multiple layers of the Encoder-Decoder model. The output is simply a probability table for which token is more likely to come next in the sequence (logits).

A transformer model can be trained in a variety of ways, with or without significant human intervention, and usually contains billions of learnt weight parameters influencing the output result. The training capacity, efficiency, and the challenges researchers face in this front, is beyond the scope of this work, as we will only focus on inference of pre-trained models.

Immediately, it becomes apparent that a model like this -one that can be trained to retain vast amounts of knowledge, can understand correlation between tokens in a sequence and has a simple and repetitive inferencing model- is the closest computers can come to mimicking the human function of language, among other mechanisms.

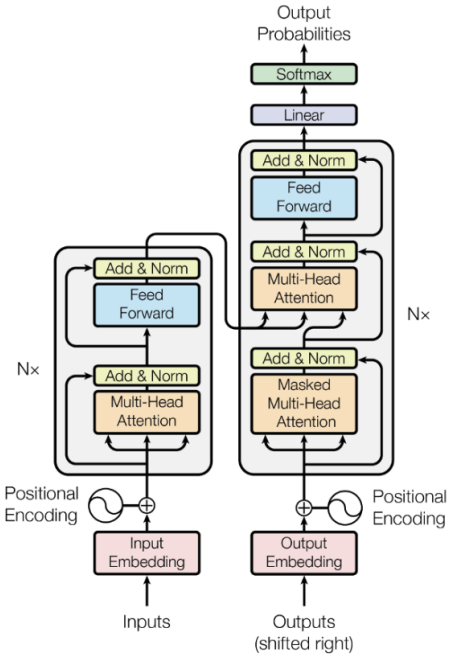
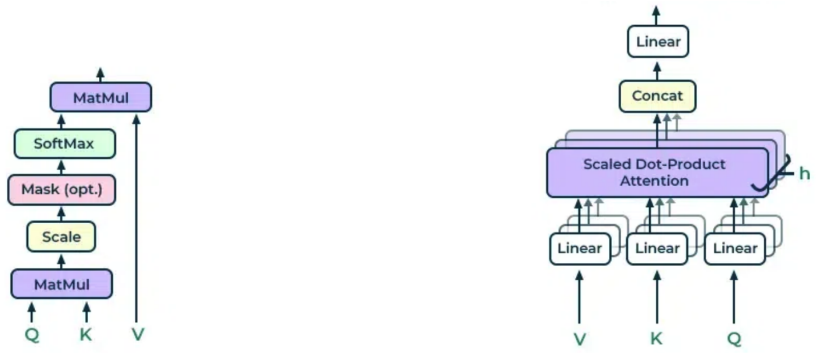


Figure 3.2.1: The Original Transformer Architecture from (Vaswani et al., 2017)

Encoder-Decoder Architecture

A transformer, as proposed in the original paper, is a specific instance of the encoder-decoder model (Fig. 3.2.1). A generic encoder–decoder architecture consists of two main components. The encoder processes an input sequence and encodes it into a fixed-length vector representation, while the decoder takes this vector and generates the corresponding output sequence. Both components are jointly trained



(a) Self-attention

(b) Multi-head attention

Figure 3.2.2: Attention in Transformers

to maximize the conditional log-likelihood of the output given the input, enabling the model to either produce an output from a given input sequence or evaluate the compatibility of an input/output pair.

A transformer can be composed of arbitrarily many identical encoder and decoder structures, called layers. Each encoder layer includes two sublayers; a multi-head self-attention mechanism, which computes token representations by attending to all tokens in the input, and a feed-forward network—augmented with residual connections and layer normalization. The decoder, while similar, employs a masked multi-head self-attention sublayer to prevent future tokens from influencing the current prediction, and adds an extra multi-head attention layer to attend to the encoder's outputs. Subsequent transformer model variants, have modified these components by utilizing either encoder-only or decoder-only architectures, respectively. Additionally, more revolutionary ideas and modifications have been introduced to the transformer, to create this AI race that we are currently witnessing [42].

Attention

The concept of attention is to enable the storage of context information in the range of a specific sequence. The breakthrough in the original transformer paper was that attention was the only such mechanism necessary to achieve well performing models, as there were many others being researched up to that point.

The way this is achieved is by calculating the Key, Query, and Value vectors for each token, from the model's attention weights, which are also learned through training. The Key and Query vectors for each token are multiplied for each position in the sequence, resulting in a vector for each token that, in a very intuitive way, represents the correlation between itself and the other tokens in this sequence. Afterwards, this result is normalized using softmax, or some other technique, by dividing it by some factor (usually the root of the dimensionality). Next, it is multiplied by the Value vector for each token, in order to "drown out" the Values of tokens with small attention scores. After adding up the Value vectors for a specific position, and for a specific token, this result (self-attention score) can be passed on to the feed forward network.

Multi-headed attention is mostly the same thing, except replicated more times for each token, with different initial attention weights, in order to encode more correlation information in total. These calculations are primarily carried out in the form of matrices for efficiency. This way, the method described above can be written simply as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V, \tag{3.2.1}$$

where Q, K, and V represent the query, key, and value matrices, respectively, and d_k is the dimensionality of the keys.

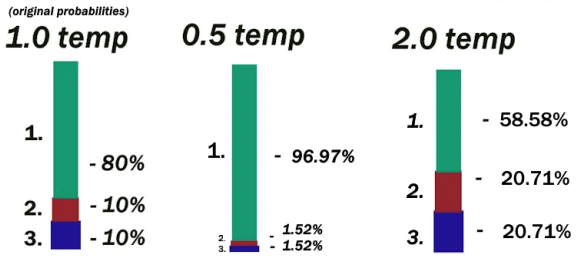


Figure 3.2.3: Example of Temperature on a Sampler. We assume that only the top 3 tokens/words can be sampled. Notice how the breakpoints don't scale linearly, as the Temperature multiplier is applied to the raw logit scores and not the probability.

3.2.2 Large Language Models

Large language models (LLMs) are a direct application of transformer architectures. They leverage vast amounts of text data (therefore "large") to learn representations of language that are both rich and versatile. These models are capable of tasks ranging from translation and text summarization to more complex applications such as dialogue generation and code synthesis.

In large language models, the transformer tokens are subwords. This is useful because it solves the out-of-vocabulary issue that is inherent in a word-based system. The models are trained on broad data (usually large data-banks from the world wide web), using self-supervision at scale. These "foundation models" can then be futher optimized for specific tasks through a process called fine-tuning. By fine-tuning already knowledgeable models to answer user questions, as an assistant, and enforcing some restrictions to the generated content, AI assistants such as the famous ChatGPT were born. Additionally, by utilizing models fine-tuned for different tasks and combining them into an assistant model that chooses which of the "experts" to direct the question towards, a new generation of models called "mixture of experts" was created.

The transformer at the heart of LLMs, should be noted, only calculates the most likely words to appear after each specific sequence. The actual choice of which word will be generated next, falls to the sampler, which can be configured to be more or less elastic with its choice of words. This is the reason LLMs don't always generate the same output for the same input sequence. A good temperature can make or break a model, as it scales the probability breakpoints for the word selection, as shown in Fig. 3.2.3.

The size of these models, being billions of parameters with a training dataset in the petabytes, begs the question of when does a language model start to speak coherent english? This question has a rather interesting answer as explored in the Tiny Stories paper [43]. It turns out that with a relatively small size, even 10 million parameters, if the domain is narrow enough (such as generating stories understandable by a 5 year old kid), language models can generate coherent english. This is important because we will be using the Tiny Stories dataset for our own experiments, since working with a large language model and a custom hardware architecture can prove to be difficult.

Finally, regarding the parallelization of LLMs, we note that because at the lowest level they consist of mostly independent matrix multiplication operations, and the attention heads are also entirely parallelizable, we expect that they will make good use of the SIMD capabilities of the host CGRA device.

3.2.3 Meta's Llama2

At this section we will take a closer look at the structure of the target LLM, Meta's Llama2 [14]. Published in 2023, the Llama2 collection of models ranging from 7 to 70 billion parameters was state of the art compared to the best competitors of that era. The model was pretrained on an extensive corpus of self-supervised data, followed by alignment with human preferences via techniques such as

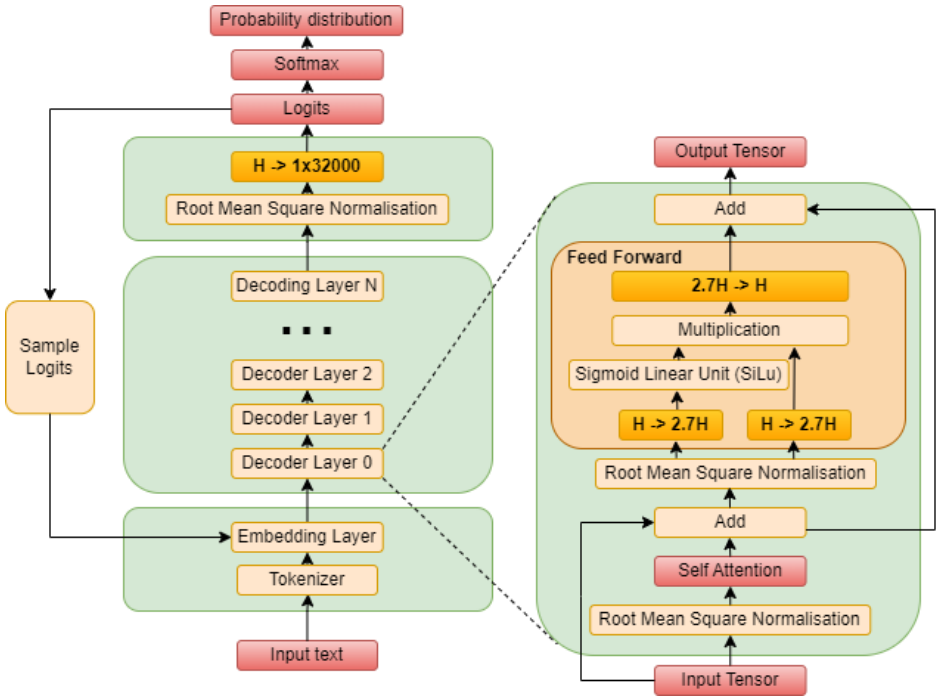


Figure 3.2.4: Overview of LLaMA 2's Model Architecture

Reinforcement Learning with Human Feedback (RLHF).

Llama 2 (Fig. 3.2.4) is composed of a series of decoder layers, where each layer consists of a self-attention mechanism followed by a feed-forward multi-layer perceptron. Unlike the original transformer, Llama models employ different projection sizes in their feed-forward layers; both Llama1 and Llama2 use a projection size of 2.7 times the hidden size, in contrast to the standard 4 times. A notable architectural difference between Llama1 and Llama2 is found in the attention mechanism: Llama 2 incorporates a Grouped Query Attention (GQA) mechanism, which enhances efficiency by grouping the Key and Value vectors across groups of attention heads, interpolating between multi-head and multi-query attention.

Some other not-so-notable modifications are the use of Rotary Positional Encoding (RoPE) for the token embeddings and the use of the sigmoid linear unit as a non-linearity, in place of ReLu or other commonly used functions.

For the purposes of our experiments we will be using a Baby Llama model of 15 million parameters trained on the Tiny Stories dataset. The model can produce coherent english text at a rate of approximately 30 tokens per second when inferenced with a custom script in C [44], modified for our needs, on our Ryzen 3 Series private work laptop, or 110 tokens per second on an M1 MacBook Air. More information about our custom setup will be presented at chapter 6 of this thesis.

Chapter 4

Related Work

Accelerating transformer architectures has become a prominent research area, with significant advancements being pursued at both software and hardware levels. As algorithmic improvements and novel techniques continue to emerge to build faster, more intelligent models, the underlying hardware must evolve to implement these innovations seamlessly. During the course of our work for this thesis, extensive research has been published on accelerating transformers using CGRAs. To justify our approach, we begin by comparing R-Blocks to other CGRA designs before exploring their applications in accelerating transformer architectures.

4.1 CGRA Architectures

4.1.1 Plasticine

Starting with the project that pointed out the potential of structured computational units for exploiting parallel patterns in software, Plasticine [45] attempts to be the first (in this recent CGRA resurgence) to solve the problem of performance and power inefficiencies that FPGAs struggle with.

Utilizing Pattern Compute Units (PCU) and Pattern Memory Units (PMU), Plasticine aims to exploit data locality, repeating memory access patterns and control flow to accelerate specific repeating parallel patterns in programs. The Pattern Compute Units are reconfigurable pipelined units, based on the SIMD execution model, while the Pattern Memory Units are composed of a banked scratchpad memory

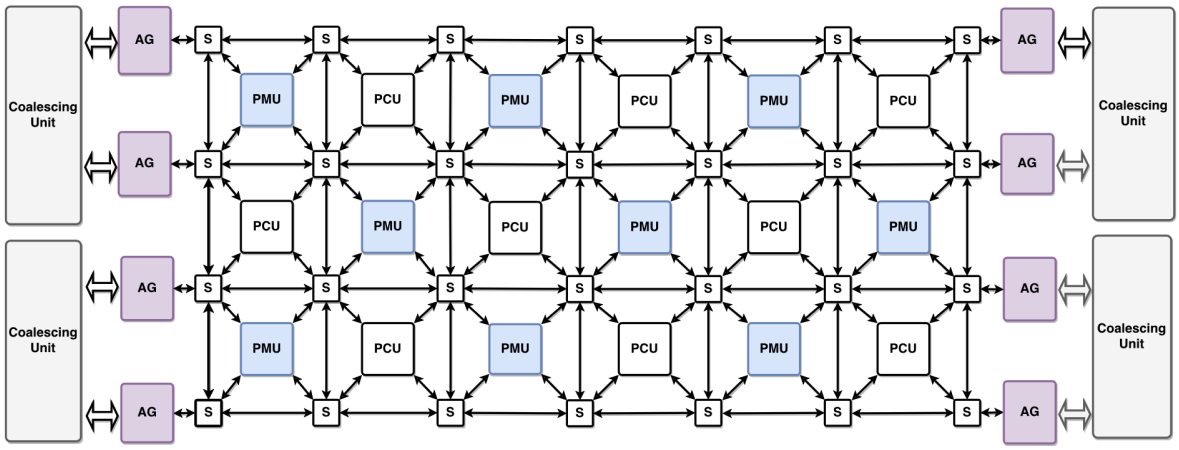


Figure 4.1.1: Plasticine chip-level scaled-down architecture (3x6 grid)

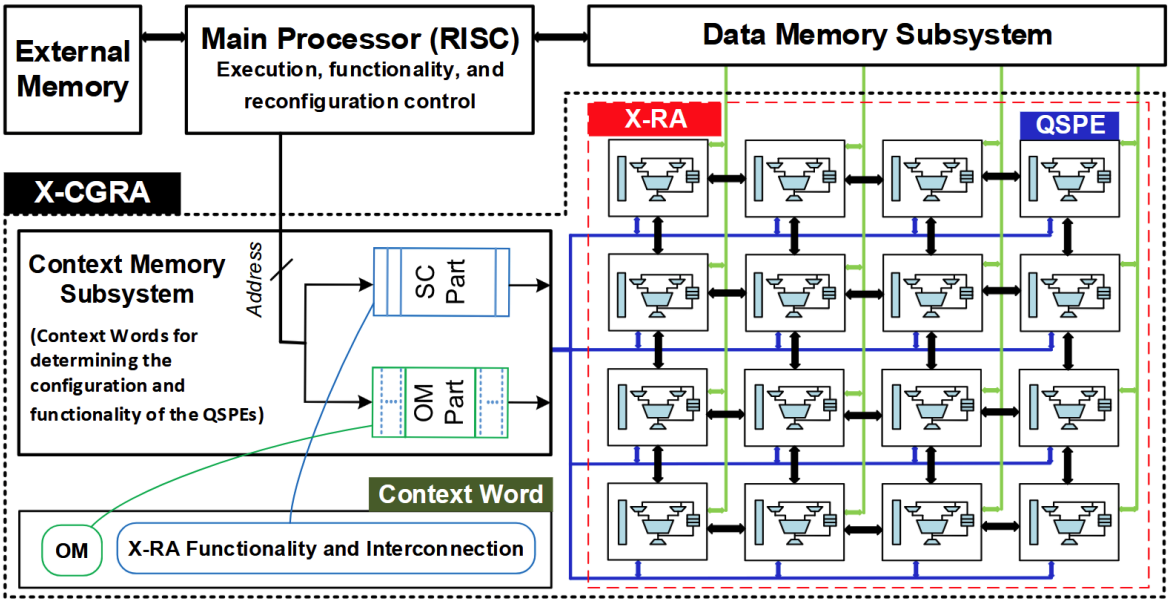


Figure 4.1.2: Architecture of X-CGRA in a reconfigurable system. X-RA refers to the Approximate Reconfigurable Array, SC refers to Structural Configuration part and OM refers to the Operation Mode part (in terms of accuracy)

with dedicated addressing logic and address decoders.

A typical Plasticine grid (Fig. 4.1.1 is composed of PMUs and PCUs that communicate with each other by a static hybrid interconnect grid. Communication with the off-chip DRAM is achieved through several address generators, supported by coalescing units to handle dense memory requests.

The main weakness of this architecture compared to R-Blocks is its reliance on parallel patterns in order to map software to the chip. The application has to be represented as a hierarchy of parallelizable dataflow pipelines written in a parallel pattern-based language called the Delite Hardware Definition Language (DHDL) [16]. Even tho this might be an excellent model in terms of hardware efficiency and power for a reconfigurable architecture -which it clearly is, as it achieves over 10 times the performance and performance per watt of an FPGA in most benchmarks-, it limits the hardware to a very narrow functional domain.

There are many other design innovations made in this project, and the reader is encouraged to look at the cited paper in order to understand the full scope of this work.

4.1.2 X-CGRA

Leveraging its approximate capabilities, X-CGRA [17] attempts to accelerate applications that are inherently error resilient, such as multimedia and signal processing. Having the ability to dynamically configure the computational accuracy, utilizing its Quality-Scalable Processing Elements (QSPEs), X-CGRA can additionally support use cases where the quality of service (QOS) requirement is 100%.

The architecture of X-CGRA is sufficiently commonplace among CGRA designs, with the QSPEs situated in a grid, interconnected by a 2D mesh as illustrated in Fig. 4.1.2. The context memory subsystem is responsible for the structural and operation mode configuration as well as communication with the off-chip main processor.

The novel mapping technique employed in X-CGRA first determines the optimal accuracy level settings and subsequently schedules and binds the software to the QSPEs. Although this technique is essential for dynamic quality calibration, it introduces significant compilation overheads that could be mitigated

by using a static approximation degree.

Inspired by the X-CGRA approach, one potential enhancement for R-Blocks is the incorporation of approximate computing capabilities. Given that the R-Blocks fabric can be readily expanded with new tiles and instructions, integrating a static approximate computing tile -such as an approximate multiplier- appears feasible. Preliminary internal research in this direction has yielded promising results, and this strategy might also be applicable in LLM applications, provided that the quality error remains sufficiently low to avoid adversely affecting model accuracy.

4.1.3 CGRA-ME

CGRA-ME [18], is a proposed CGRA unification framework that includes design, simulation and modelling capabilities for various types of CGRA architectures. In 2024, with [19], it became even more capable, including support for more elastic CGRAs, floating point arithmetic, predication and hybrid RISC-V systems with CGRA.

The main contribution of this work is an architecture exploration framework that takes an XML description of a CGRA as input and provides a flexible scheduling, mapping, placement and routing tool, built within the popular LLVM compiler infrastructure, that maps an optimized C-language benchmark onto the specified CGRA. This allows the user to rapidly test and evaluate different CGRA architectures, that adhere to the systems specific restrictions.

A unification framework is something that will become necessary in the future, as CGRA technologies get more traction, while it can already provide valuable insights into the tradeoffs specific to CGRA design. However, the immediate adaptation of this system by the academic community could stunt the growth of the field by narrowing the exploration domain and restricting the creative freedom of researchers.

4.1.4 Conclusion

There are numerous published CGRA architectures that are worth mentioning, but this is not the point of this thesis. We expect that, up to this point, and with all the approaches presented so far, the reader has accumulated the necessary knowledge to understand and evaluate the methodologies that will follow. We will continue by evaluating attempts at transformer acceleration using CGRA systems that have not been presented thus far.

4.2 Transformer Acceleration Using CGRAs

4.2.1 ML-CGRA

ML-CGRA [20], published in 2023, does not explicitly aim on accelerating transformers, but instead machine learning applications which at a low level are very similar, relying on generalized matrix multiplication to compute activations from weights and inputs.

ML-CGRA is an open-source integrated end-to-end compilation framework that enables efficient and flexible ML acceleration on CGRAs. In this work, a standard CGRA template is also enhanced with additional hardware components. The conventional on-chip mesh network is replaced by an 8-directional king mesh topology for improved dataflow routing (Fig. 4.2.1). Furthermore, the input buffers support configurable stalling and double-buffering, while dedicated MAC and MAX functional units enable single-cycle execution of common ML operations.

With all these innovations the framework, on average, improves the execution performance of a diverse set of ML models with a 3.15× and 6.02× speedup on 4×4 and 8×8 CGRAs respectively. This speedup is certainly impressive but it should be noted that ML application acceleration has been a topic of research for a long time and the framework ecosystem surrounding this system is very user friendly and standardized.

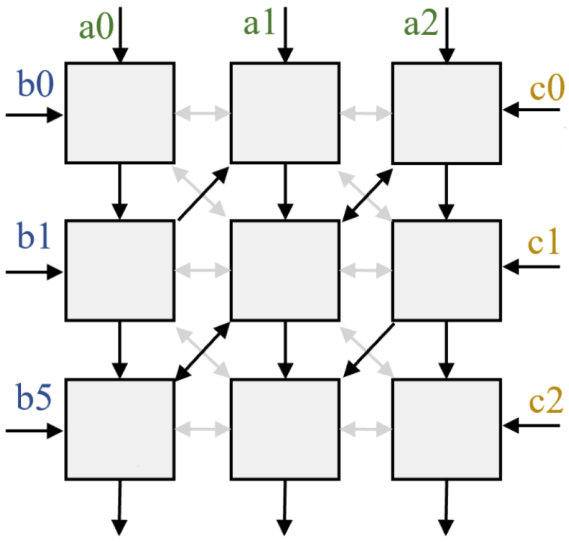


Figure 4.2.1: Mesh topology in ML-CGRA architecture

4.2.2 IMAX3

IMAX3 [21] is a proposed Linear Array Coarse-Grained Reconfigurable Architecture (CGLA), implemented on FPGA, used for LLM inference and result evaluation. During this thorough and very illuminating work, improvements were made to the high level code which was adapted from Python for IMAX3 using GGML, a library for running LLMs on CPUs.

Architecturally IMAX3 distinguishes itself from traditional 2D CGRAs, which often experience uneven memory access distances, by employing a ring array structure. This design choice improves memory bandwidth utilization relative to GPUs and simplifies the programming process, thereby contributing to enhanced power efficiency.

Another notable advantage of IMAX3 is its ability to generate computation networks directly from conventional program code without the lengthy compilation times typically required for CGRA exploration. The ring array structure removes the need for such exploration, while allowing programmers to provide hints for location-free operations, multi-input computations, cache referencing, and DMA transfers between main memory and cache. Optimizations across the compiler, library, and hardware components additionally minimize data transfer overheads. Moreover, equipping each unit with CISC-based multifunctional arithmetic units reduces the number of required units and simplifies network routing, further decreasing compilation time. Again we encourage the reader to check out the original paper for a deeper understanding of the architecture, as it was developed over a series of steps, based on IMAX and IMAX2, that are documented extensively and each introduce its own innovations.

At a high level, the researchers noticed the domination of the matrix multiplication function both in terms of time and resources utilization. After optimizing it for the CGLA and testing the system on real hardware they reported up to 20% improvement compared to commercial CPUs. This is extremely impressive performace considering this is the first LLM application to be tested on an actual CGRA hardware environment.

4.2.3 CFEACT

CFEACT [22] is another CGRA-based framework for implementation and evaluation of custom accelerator SoCs. This work, however, focuses on acceleration of transformer and CNN benchmarks. It includes a hardware generator that supports multiple template configurations and design parameters, while a front-end compiler processes the application's execution kernels efficiently creating the data flow graph (DFG) for the specific CGRA configuration.

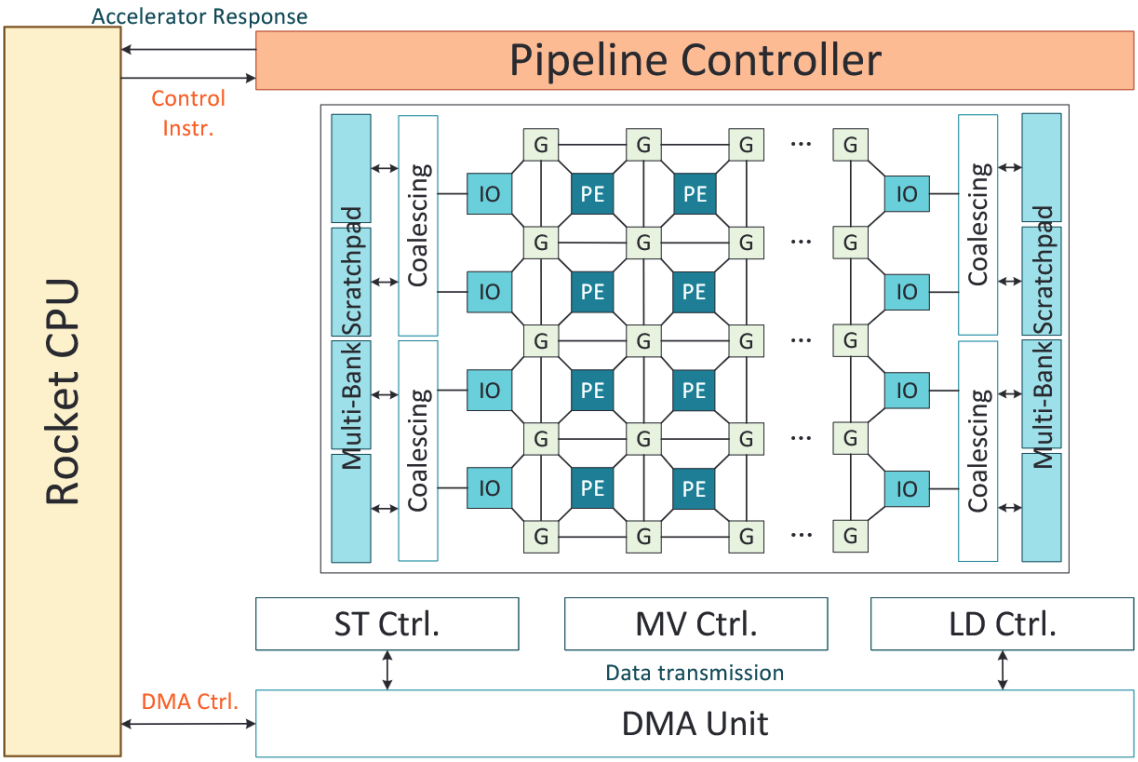


Figure 4.2.2: CFEACT Architecture Overview

The CGRA's SoC organization is conventional, featuring the same PEs in a grid with an interconnect network, IO units and memory controllers surrounding the reconfigurable fabric, communicating with an off-chip main processing core and utilizing DMA for memory accessing (Fig. 4.2.2).

For transformer applications specifically, this setup, with its extensive front-end compiler optimizations and innovations, achieves on average 58% better performance compared to the same size ML-CGRA setup, explored previously. This is not surprising considering it is specialized for transformer applications, and, as the researchers comment, with better PE utilization it should exhibit an ADP gain of more than 2x over ML-CGRA.

4.2.4 ULP CGRA for Transformer Acceleration at the Edge

Perhaps the most similar work to what we are trying to achive with R-Blocks is [46]. This CGRA fitted specifically for accelerating General Matrix Multiplication (GEMM), tackles the fundamental computational time-sink of transformers head-on. It presents an analysis of how CGRA architectures can be tailored to these workloads, highlighting recent innovations in PE design, memory management, and interconnect strategies.

While bearing obvious organizational similarities with other CGRA structures (Fig.4.2.3, the innovation of this work is the inclusion of PEs designed to specifically perform low-latency arithmetic operations required by GEMM, and placing them in a heterogeneous grid as shown in Fig. 4.2.4 so that every row has direct access to a Memory Operation Block (MOB).

Additionally, a unique feature of this CGRA is its switchless mesh torus interconnect, which facilitates data transfers between the PEs and MOBs without a conventional switching network. This reduces the power consumption while making for a predictable, fixed data flow pattern, aligning with what is required for GEMM.

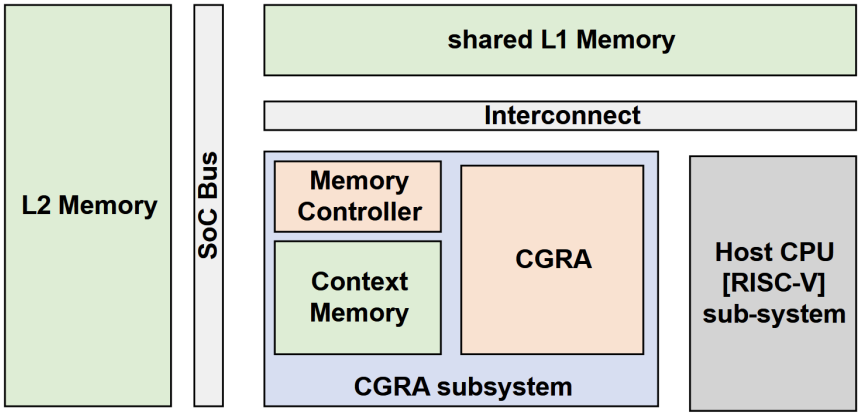


Figure 4.2.3: ULP CGRA SoC Overview

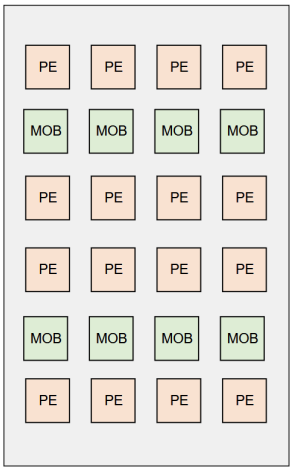


Figure 4.2.4: ULP CGRA Reconfigurable Fabric

Although no results from comparisons to baseline systems are provided in this work, the techniques used to parallelize the software of the transformer, such as loop tiling, and data reuse can be applied in our case as well, although in a different context.

No further research has been published on transformer or LLM acceleration utilizing CGRAs this far, and even this cited work has been published in the year leading up to this thesis. However there is one more dimension relevant to this field of research that might be worth discussing.

4.2.5 TransMap

A different approach to transformers in CGRA is presented in [47]. Here, instead of using the CGRA to accelerate transformers, the transformer model is utilized to encode global attention into the CGRAs mapping state. Furthermore, a deep reinforcement learning agent is devised to decode the state, providing guidance for the mapping process. Experimental results show that TransMap offers superior mapping quality and reduced compilation time compared to current state-of-the-art approaches, even in complex mapping scenarios.

This opens the door to other avenues of future research, such as incorporating transformers to all CGRAs, running on native hardware, for even better mapping of applications, thus revolutionizing the field of CGRAs completely.

4.2.6 Summary

Comparing R-Blocks to other proposed CGRAs, the most important differences, as shown in the table of Fig. 4.2.5, are the operation level disaggregated approach of the architecture as well as the retargetable compiler that allows R-Blocks to map high-level code directly to a specific user defined virtual architecture. These differences will play a big role in our approach of expanding and mapping code on R-Blocks that will be presented in the next chapters.

CGRA	Publication Date	Architecture	Platform	Mapping Technique	Architectural DSE	LLM Mapped
Plasticine	June 2017	Homogeneous PEs and Memory Elements	ASIC	Utilizing parallel patterns (DHDL)	Manual	No
X-CGRA	October 2020	Homogeneous QSPEs	ASIC	DFG kernel mapping	Manual	No
CGRA-ME	May 2024	3 homogeneous PE architectures	Simulated	C kernel mapping. Differs by architecture	Manual	No
ML-CGRA	July 2023	Homogeneous PEs	Simulated	Python MLIR into kernel mapping	Manual	No
CGLA	November 2024	IMAX3 in memory CISC	FPGA, ASIC	Manual, utilizing GGML	Manual	Yes
CFEACT	September 2024	Homogeneous PEs	ASIC	Linearization and kernel mapping	Scalable Template	Yes
R-Blocks (our work)	-	Disaggregated PEs (op. level)	ASIC	High-level architecture targeting compiler	Semi-Automated	Yes

Figure 4.2.5: Summary of publications relating to CGRA architectures and LLM code mapping

What is also worth noting, is how recently the first examples of research regarding LLM mapping in this type of architectures were published. This reveals the experimental character of this work and how emerging the research field of combining those two technologies really is.

Chapter 5

R-Blocks Expansion Methodology

As discussed in previous chapters, LLMs make heavy use of Floating Point (FP) arithmetic in their calculations of attention and activation values. However, in its previously available state, the R-Blocks architecture did not include support for FP operations. This is why we expanded the entire CGRA flow with a Floating Point Unit tile, and tool support to be able to compile, translate, build and execute high level code that includes FP operations.

5.1 Hardware Expansion

5.1.1 The Floating Point Unit

IEEE754 is the technical standard for Floating Point arithmetic, which we will follow. It defines the operations between different precision FP numbers, the rules about formats, conversion and rounding modes, as well as rules about exceptions and special values (such as infinity or NaN). A floating point unit (FPU) needs to produce results that adhere to this standard, so they can be utilized without conflicts by the other hardware and software components of the system.

For this purpose we use the OpenCores FPU [23], which is fully IEEE754 compliant, light, simple and provides all the capabilities we require. This is a single-precision FPU, meaning it can only support 32-bit operands, which is, currently, also a limitation of the R-Blocks architecture.

The functionality of the hardware was verified independently by, testing it with over 14 million case vectors, generated by Berkeley's SoftFloat library [24].

The FPU supports the following operations with FP32 operands and output:

- Addition
- Subtraction
- Multiplication
- Division
- Conversions (INT32 to FP32, and backwards)
- Comparisons

With these simple arithmetic operations and using high level math libraries such as math.h we can perform any other operation we might require, such as square root, exponentiation and trigonometric functions. The hardware also supports 4 different rounding modes, but we will only make use of the "rounding to nearest even" mode, since the rest of the rounding modes neither correspond to OpenASIP instructions bound to the LLVM back-end, nor are they required for our LLM application.

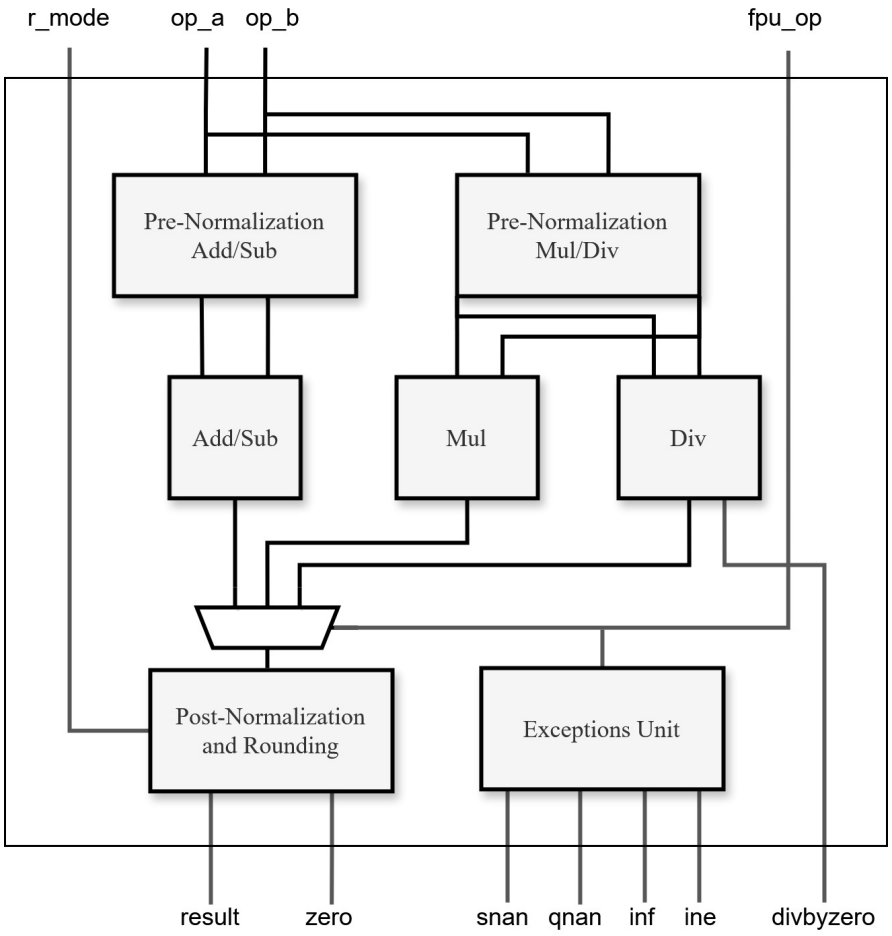


Figure 5.1.1: Internal structure of the OpenCores FPU

The FPU is 4 stage pipelined. It can perform a floating point operation every cycle. It will latch the operation type, rounding mode and operands and deliver a result four cycles later.

In terms of microarchitecture, two pre-normalization units adjust the fractions (mantissas) and exponents, one dedicated to addition and subtraction operations and the other to multiplication and division. The Add/Sub, Mul, and Div primitive blocks then execute the corresponding arithmetic operations. A shared post-normalization block subsequently normalizes and rounds the fraction before the final result is packed into a valid single-precision floating-point format. Figure 5.1.1 illustrates the internal implementation (pipeline not shown).

The input and output signals are explained in Table 5.1.

The compare module is a standalone IEEE754 compliant single-precision FP unit, that was designed and verified separately, again, using the SoftFloat library. Table 5.2 lists the signals of the compare module and their respective functionality. This module is not pipelined since it is more compact than the rest of the FPU, which means it outputs the result one clock cycle after the inputs are asserted.

Both of these modules were synthesized at 22nm in FD-SOI technology, with a 415MHz maximum clock frequency determined by the Synopsys Design Compiler. The individual area and power metrics were not far off other R-Blocks tiles, which means that the FPU can be smoothly incorporated.

Table 5.1: FPU Signals

Signal Name	Width	Direction	Description
clk	1	Input	System Clock
r_mode	1	Input	Rounding Mode
fpu_op	1	Input	Floating Point Operation Select
op_a, op_b	32	Input	Operand A and B
out	32	Output	Result Output
snan	1	Output	Asserted when either operand is a SNAN
qnan	1	Output	Asserted when output is a QNAN
inf	1	Output	Asserted when output is an INF
ine	1	Output	Asserted when the Result is Inexact
divbyzero	1	Output	Asserted when fpu_op is set to divide and op_b is zero
zero	1	Output	Asserted when the output is a numeric zero

Table 5.2: Compare Module Signals

Signal Name	Width	Direction	Description
clk	1	Input	System Clock
opa, opb	32	Input	Operand a and B
unordered	1	Output	Asserted when opa or opb is a NAN
altb	1	Output	Asserted when opa is larger than opb
blta	1	Output	Asserted when opb is larger than opa
aeqb	1	Output	Asserted when opa is equal to opb
inf	1	Output	Asserted when opa or opb is a INF
zero	1	Output	Asserted when opa is a numeric zero

5.1.2 The FPU R-Blocks Tile

A hardware module must have a specific structure in order to be incorporated to the R-Blocks ecosystem as a tile. All tiles up to this point, such as ALUs, RFs and MULs have been operating at a single cycle latency. They all perform the calculations during the clock cycle, and latch the result at the output registers in the next rising edge. Our FPU is the first tile aiming to break that convention, by being 4 stage pipelined, something that is thankfully supported by the OpenASIP compiler and toolflow, as well as the hardware environment.

Additionally, tiles must have the same input and output ports, which creates the need for a hardware wrapper module that unpacks the CGRA inputs into FPU inputs and signals, and packs the FPU results back into CGRA outputs. All generic tiles (generic meaning not with special functionality and connections, such as the Immediate Unit or the Instruction Decoder), have the inputs and outputs shown in Table 5.3.

Each tile has 8 inputs and 2 outputs, each one of size 32 bits. The *Inputs* and *Outputs* signals pack

Table 5.3: Generic Tile Ports

Signal Name	Width	Direction	Description
Clk	1	Input	System Clock
Reset	1	Input	System Reset
Inputs	8x32	Input	8 different 32-bit Inputs packed into one
DecodedInstruction	12 up to 33	Input	Width depends on the tile
Outputs	2x32	Output	2 different 32-bit Outputs packed into one

these values in one data bus and connect to the switchbox interconnect network. The input ports that will be used for the calculation, and the output register where the output will be stored, are specified in the *DecodedInstruction* signal. This signal originates from each tile's respective *Instruction Decoder* tile and contains all the information required to perform its operations, besides the input values. Specifically, it contains:

- The *OutputValid* signal (Enable).
- Source and Destination ports (3-bits to distinguish between the 8 inputs and 1-bit to distinguish between the 2 outputs).
- The Operation Type, relevant to each specific tile.
- Configuration and Mode selection signals specific to each tile. Some tiles for example can support dual output mode for large operands.

Because each tile has its own Instruction Decoder FU, the decoded operation codes between different tiles can overlap, decreasing the bit-width of the signal necessary to program and configure each one of them, compared to requiring a unique opcode for every instruction in the ISA. The bits of the *DecodedInstruction* signal are highly customized for each tile, and are used directly as control signals for the underlying hardware.

To connect our FPU for instance, considering the operations it must be able to support, we break up the *DecodedInstruction* signal into the following control signals:

- The *OutputValid* enable signal.
- The *CompareFlag* which identifies comparison operations.
- The FPU opcode, which is directly fed into the FPU module's *fpu_ op* input.
- The *RoundingMode* signal, which, in this implementation, is tied to 00 for all instructions, since only the round-to-nearest-even mode is used. However, by including this hardware connection, new instructions that use different rounding modes can be added to the ISA with minimal effort in the future.
- The *Dest* and *Src* signals which specify the inputs and outputs that will be used by each operation.

Figure 5.1.2 illustrates in the necessary detail the specifics of the hardware implementation. The source and destination selection signals, originating from the decoded instruction, are used as select signals for the input and output select multiplexers, and the *CompareFlag* denoted **Comp** in Fig. 5.1.2 is used as the selector between the FPU result and the comparator result.

What should be commented on is the timing of the circuit. As mentioned before, the FPU produces the correct output 4 clock cycles after the input changes and a new *DecodedInstruction* signal is asserted, with the *OutputValid* flag set. Therefore, the rest of the control signals and also the output of the Compare Module must be delayed too, in order for the result to reach the output register at the correct time. This is done by the Delay Registers block which introduces a 4 cycle delay and subsequently sends the control signals and the calculated comparison result to the output multiplexers.

In total, because the output result is also stored in a register, the circuit has a 5 cycle output latency. This will be taken into consideration during the tool-flow expansion.

One more component of the tile is the Output Conversion Logic, which aims to make the comparator adhere to the IEEE754 standard. Out of the box, besides the exception logic, the comparator answer the question; "What is the relation between operands A and B?". This question has 3 possible answers assuming valid comparison; operand A is either greater, equal or less than B. This is encoded in the 3 output signals of the original comparator, but needs to be converted to a different question, the one that the high level (C) instruction asks. These 6 possible questions of relations between two operands (for instance "Is a $>$ b?", or "Is a $<=$ b?") always have a binary answer, which can be extracted, with some logic from the outputs of the Compare Module, as shown in Table 5.4.

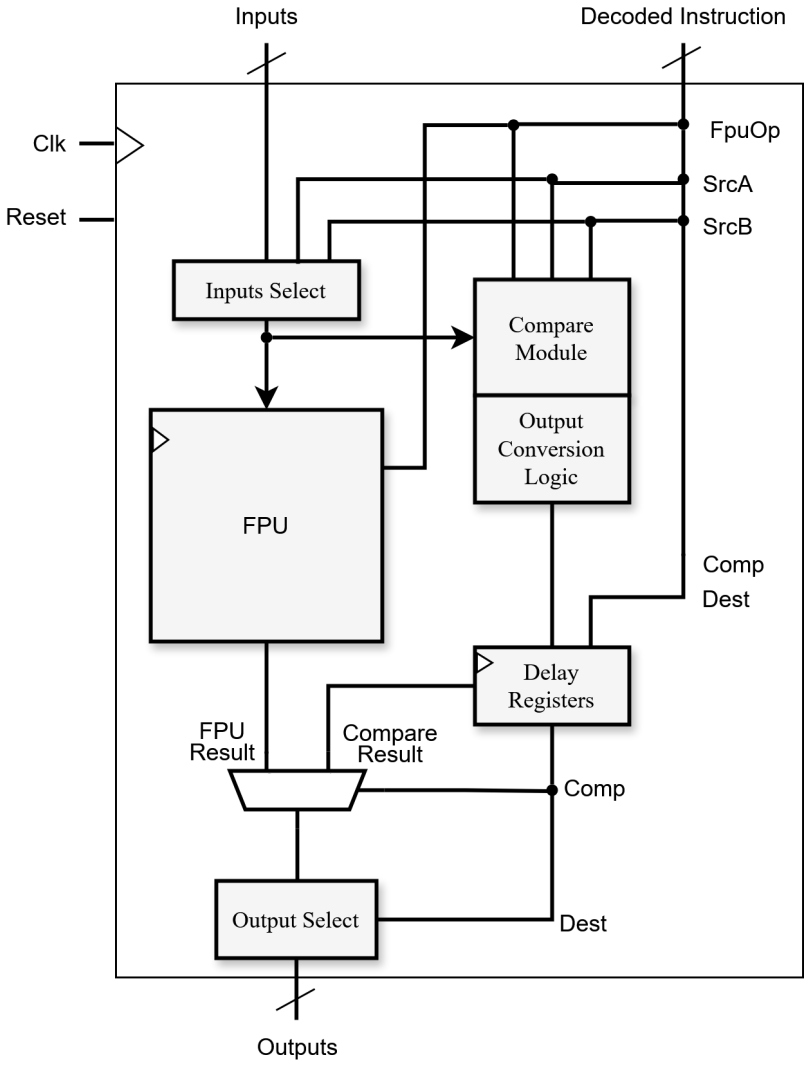


Figure 5.1.2: The FPU Functional Unit

5.2 Toolset Expansion

The R-Blocks compilation and execution framework was based upon the TTA model for Exposed Datapath Architectures (EDPAs). This means that R-Blocks at its core is a TTA machine with some restrictions that define its architectural features.

In order to understand how to compile and map high-level code to the reconfigurable hardware we must take a tour through the many abstraction layers utilized in this work.

5.2.1 Instruction Set Architecture

The connection between software and hardware is initially made in the Instruction Decoder (ID) tile. This is a special FU with a specific functionality. Each computational (or generic) FU is connected to its own ID, which has the responsibility of decoding and sending the instructions stored in the instruction memory to its respective FU every cycle.

The ID tile is basically a dictionary of encoded bit sequences to decoded meaningful signals that correspond to the control signals in each FU. The generation of the ID is done using Huffman Trees to

Table 5.4: Output Conversion Logic

C Operator	Assembly Instruction Name	Conversion Logic
==	eqf	NOT unordered AND aeqb
!=	nef	NOT unordered AND NOT aeqb
>	gtf	NOT unordered AND altb
>=	gef	NOT unordered AND (altb OR aeqb)
<	ltf	NOT unordered AND blta
<=	lef	NOT unordered AND (blta OR aeqb)

minimize the instruction width, which means that when the ISA changes, the ID must be regenerated to account for the new instructions in the tree.

The entire ISA is defined in a single XML file. This is where the Functional Units and their instructions are declared, and the specific decoded instruction bits are bound in a specific position of the decoded instruction.

```
<FPU decoded_width="16">
    <nop        decoded_instr="0_0_000_00_00_?_???_???"/>
    <pass       decoded_instr="0_0_000_00_01_D_???_AAA"/>

    <addf       decoded_instr="1_0_000_00_00_D_BBB_AAA"/>
    <subf       decoded_instr="1_0_001_00_00_D_BBB_AAA"/>
    <mulf       decoded_instr="1_0_010_00_00_D_BBB_AAA"/>
    <divf       decoded_instr="1_0_011_00_00_D_BBB_AAA"/>

    <cif        decoded_instr="1_0_100_00_00_D_???_AAA"/>
    <cfi        decoded_instr="1_0_101_00_00_D_???_AAA"/>

    <eqf        decoded_instr="1_1_000_00_00_D_BBB_AAA"/>
    <nef        decoded_instr="1_1_001_00_00_D_BBB_AAA"/>
    <gtf        decoded_instr="1_1_010_00_00_D_BBB_AAA"/>
    <gef        decoded_instr="1_1_011_00_00_D_BBB_AAA"/>
    <ltf        decoded_instr="1_1_100_00_00_D_BBB_AAA"/>
    <lef        decoded_instr="1_1_101_00_00_D_BBB_AAA"/>
</FPU>
```

As shown in the above code excerpt directly from the FPU ISA definition file, the current implementation of the FPU ISA has 16-bits decoded instruction width and supports the basic arithmetic operations, conversions between FP32 and INT32 numbers, as well as the comparison operations explained in Table 5.4.

The names of the instructions are noteworthy. They were not chosen at random, but instead correspond to instructions in the Operation Set Abstraction Layer (OSAL) of OpenASIP. The OSAL stores the semantic properties of the operation, which includes the simulation behavior, operand count, memory accesses, interactions with other operations, but not the latency (Fig. 5.2.1). It is also possible to add custom operations to the OSAL, but the behavioral model for the new instruction has to be defined strictly.

Thankfully, the behaviors and parameters of the basic single-precision FP operations we are implementing are already in the OSAL, so we only have to use the designated names to bind an instruction from the ID's ISA to the TTA OSAL, so it can be understood by the LLVM compiler.

At this stage, we have already converted the hardware control signals to instructions and now defined their behavior as operations. However, to complete the software-hardware co-design process, and map

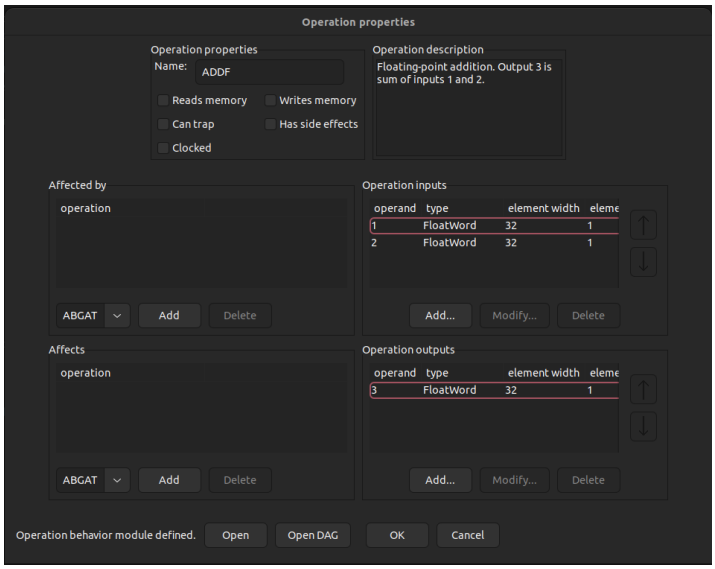


Figure 5.2.1: The Operation Set Editor GUI

high-level code to the reconfigurable fabric, it has to be compiled with the retargetable TTA compiler *tcecc*.

5.2.2 Blocks Translator

The tcecc compiler takes both the high-level driver code and the target TTA description as inputs, and produces a TPEF executable file for this specific instance of R-Blocks (Fig. 5.2.2. The conversion from an R-Blocks architecture description to a TTA is done by the Blocks Translator tool.

To achieve this translation, which works both ways, the blocks translator must include a behavioral description of every R-Blocks FU, containing its instructions, their pipeline timings and operands as well as the vectorized instructions that can be used when connecting multiple FUs in parallel as a vector unit.

Of course, the Blocks Translator tool had to be expanded with the description of the FPU tile and its instructions in order to support the new operations. This was done simply by implementing the BlocksFPU class in C++, inheriting from the generic BlocksFU class, and adding the different instructions in its execution pipeline.

With the new additions, the Blocks Translator successfully translates a user defined R-Blocks architecture to a TTA description and the compiler can follow up by generating the targeted executable for this architecture. At this stage the newly incorporated FPU tile was independently tested to verify its functionality as a FU and ensure consistency with the R-Blocks environment.

5.2.3 Hardware Generation

The next stages after compiling an executable for a specific architecture are architecture pruning and synthesis. The prunning is performed by the prunner utility provided in the R-Blocks tool suite, and it is necessary, in order to convert a fully connected virtual architecture to a new architecture with fewer connections between the FUs, so that they can be routed in the physical R-Blocks grid. Finally, the mapping, routing and synthesis of the generated bitstream are performed by the corresponding scripts and tools, in order to extract results about the area and power of the IC.

At this stage in the R-Blocks development, the CGRA does not actually communicate with an external controller, so the required instruction and data initialization memories are generated in the form of

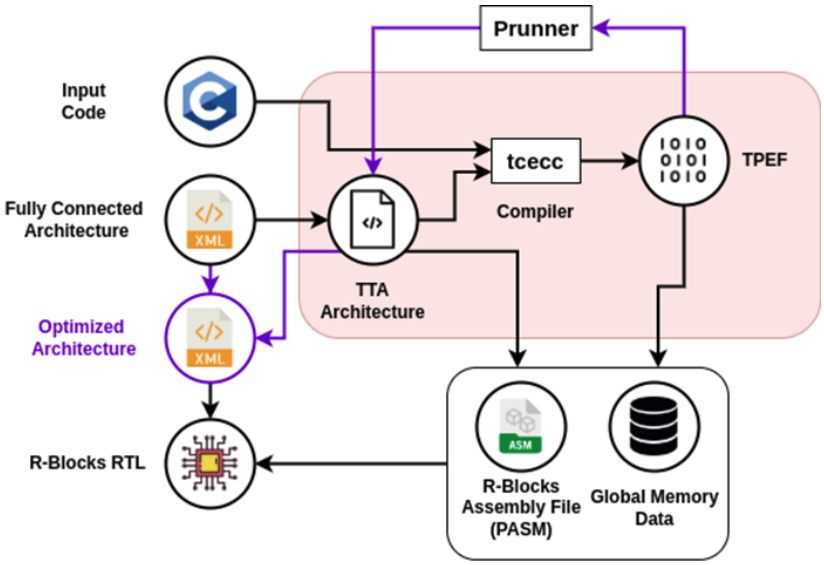


Figure 5.2.2: The R-Blocks Hardware-Software co-design flow

binaries and loaded to the hardware in order to simulate and evaluate the application. However, progress towards implementing an AXI protocol connection with a host RISC-V core is underway.

The hardware generation scripts also had to be slightly expanded to include the FPU tile in all of their parameters and restrictions. One problem encountered in this step was the limitation of the instantiator scrip regarding the number of different FU types. The restriction stemmed from the use of only 3 bits to encode the FU type, leading to only 8 different FU types being able to generate in a specific instantiation. Since there were already 8 different FU types, there was no room for the inclusion of our FPU. This constraint was lifted by increasing the configuration signal width to 4 bits, both in the instantiator scripts and in the internal Instruction Decoder configuration signals.

Having incorporated the new FU from end-to-end, we evaluated the new capabilities of R-Blocks. RTL level simulations validate the correct functionality of all the FUs and vector unit functionality checks ensure the operational stability of the new tile. Early mappings of "math.h" function testcases to the CGRA ensure that the FP operations are utilizing the FPU, and the results are consisten with those of the GCC compiled code. The performance of the new tile, however, will be better evaluated in the following chapter, with the LLM benchmark mapping methodology.

Chapter 6

LLM Mapping Methodology

In this chapter, we will present, in an understandable and concise manner, the methodology followed for mapping high-level LLM code to the R-Blocks CGRA, after the hardware and tool-flow expansion has been successfully completed and R-Blocks can support floating point arithmetic. Afterwards, the results from the mapping of the first benchmarks to targeted architectures will be presented and analyzed.

6.1 Architecture Definitions

The first step to the SW/HW co-design process is defining the architectures on which the code will be mapped. As a starting point in the endless R-Blocks architectural design space, four architectures were described, and evaluated for LLM benchmarks. These architectures make use of varying degrees of parallelism, by including different sizes of vector units. The specifics of each architecture are presented in Table 6.1.

The Scalar architecture contains the fewest possible FUs on which a full LLM benchmark can be mapped. These include the FPU tile that handles the floating point operations, MUL and ALU tiles for integer operations, one register file (RF), the load-store unit (LSU), the accumulate-branch unit (ABU), and the immediate (operand generation) unit (IM) that are necessary for system operation, together with their corresponding instruction decoders. Note that the immediate unit contains its own instruction decoder internally. This is why the number of IDs is lower than the number of FUs in the Scalar architecture.

The vector architectures are simply extensions of the Scalar architecture, containing the same system operation units (LSU, ABU, RFs), but augmented with vector units, designated by their vector size, which corresponds to the degree of parallelization that each can achieve. For example, Vector8 contains 8 FPUs connected as a vectorized unit of size 8, controlled by the same Instruction Decoder (as shown in Fig. 6.1.1), plus an extra scalar FPU for FP operations that have not been vectorized in the mapped code. This architecture can achieve 8 concurrent FP operations per cycle -at least in theory, with optimal operation scheduling.

Table 6.1: Architectural DSE Parameters

Architecture Name	Vector Size	Local Memories	Grid Size	IDs	FPUs	Total Active Computational Units
Scalar	-	1	4x6	6	1	7
Vector4	4	4	6x11	12	5	29
Vector8	8	8	8x10	12	9	41
Vector16	16	16	8x18	12	17	73

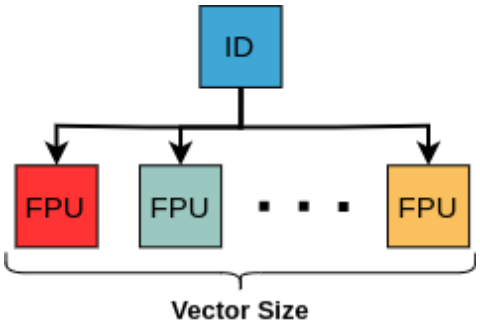


Figure 6.1.1: Illustration of multiple FPU tiles controlled by the same ID

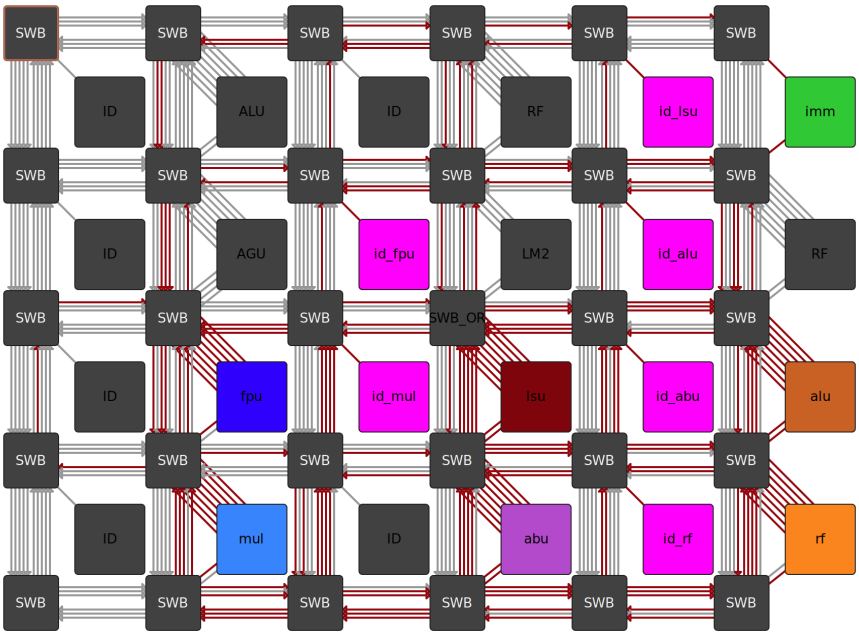


Figure 6.1.2: Scalar Architecture

The architectures were initially defined with their FUs fully connected, meaning each FU could communicate with any other FU directly (with a single cycle delay). This, however, is not realistic and could not be realized in an R-Blocks ASIC, since each FU has a maximum of 8 input wires originating from the switchbox network. To remedy this issue the architecture descriptions had to be passed through the R-Blocks Prunner tool which iteratively identifies useless connections and eliminates them to create a "legal" architecture interconnect network.

The final R-Blocks grids with the defined FUs colored and the unused FUs greyed out are shown in Figures 6.1.2-6.1.5. These visualizations are produced after mapping and routing a specific architecture in a specific R-Blocks instance grid. Note that Vector16 architecture could not be routed (this is why no red/active connection wires appear) by the current R-Blocks routing algorithm because of the complexity and density of connections between the FUs. Therefore, we could not extract measurements about the Area and Power of this specific architecture as it couldn't be synthesized. The cycle measurements presented in the next sections are derived from the TTA simulator, which is adequately accurate according to the rigorous testing conducted.

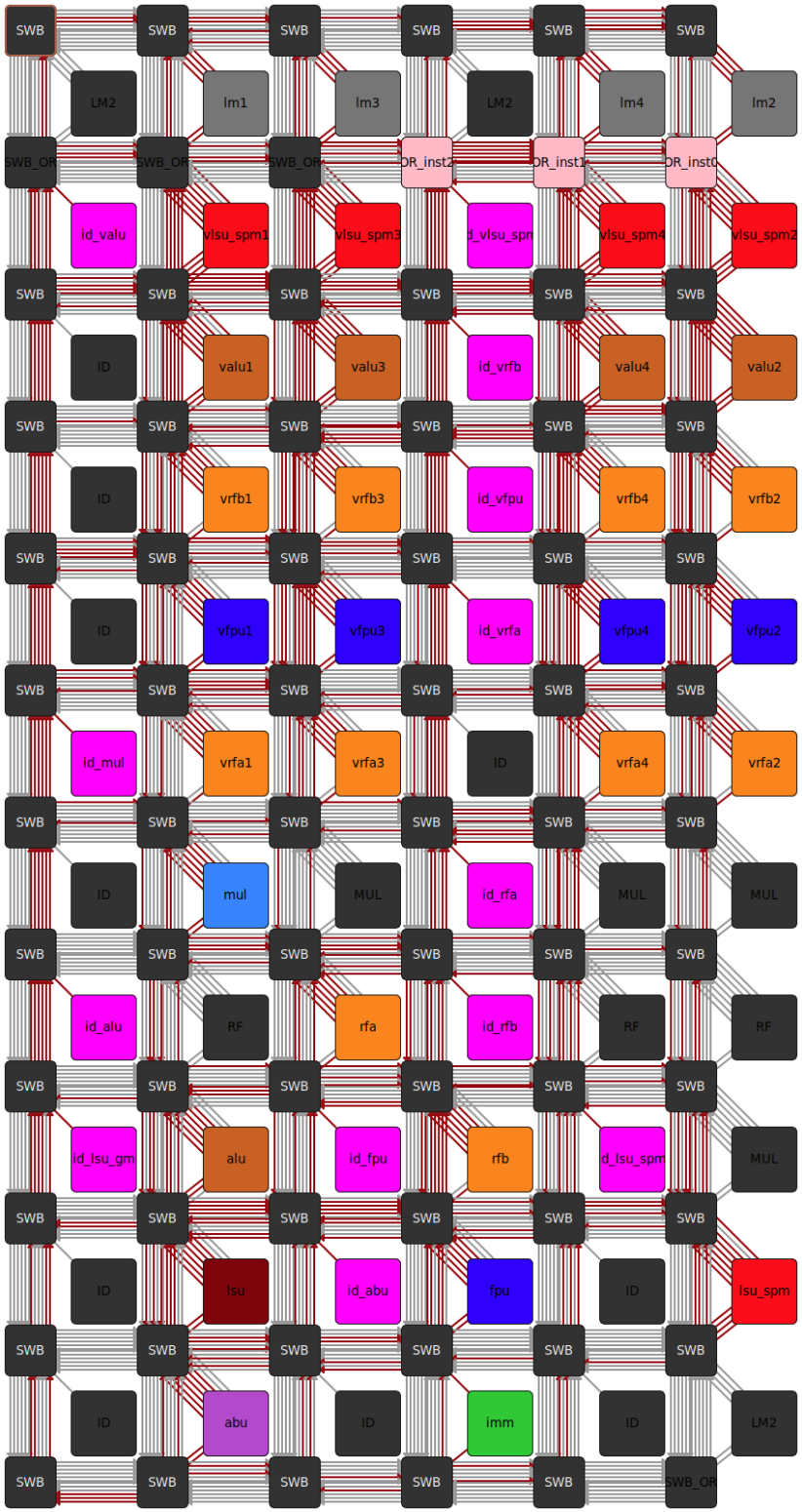


Figure 6.1.3: Vector4 Architecture

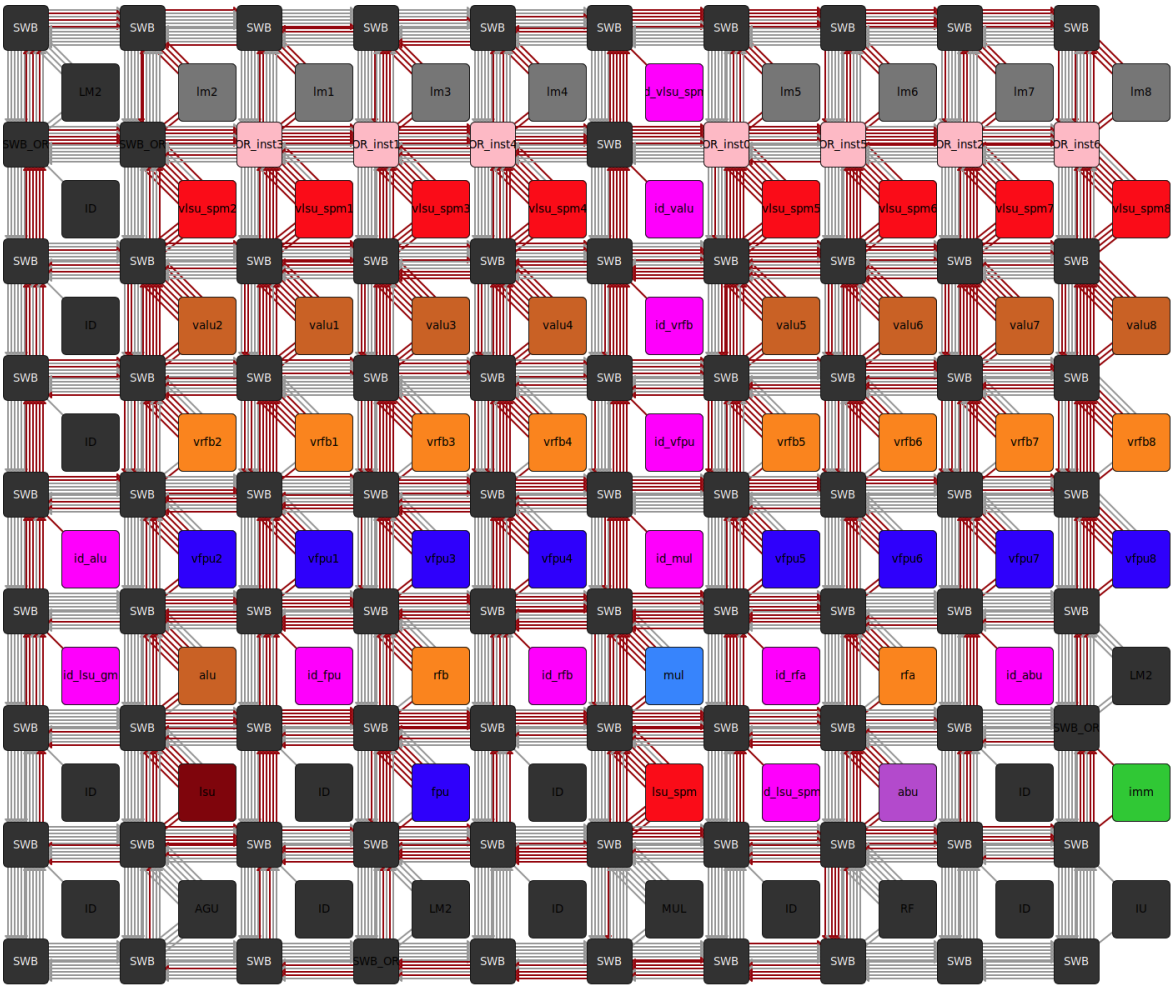


Figure 6.1.4: Vector8 Architecture

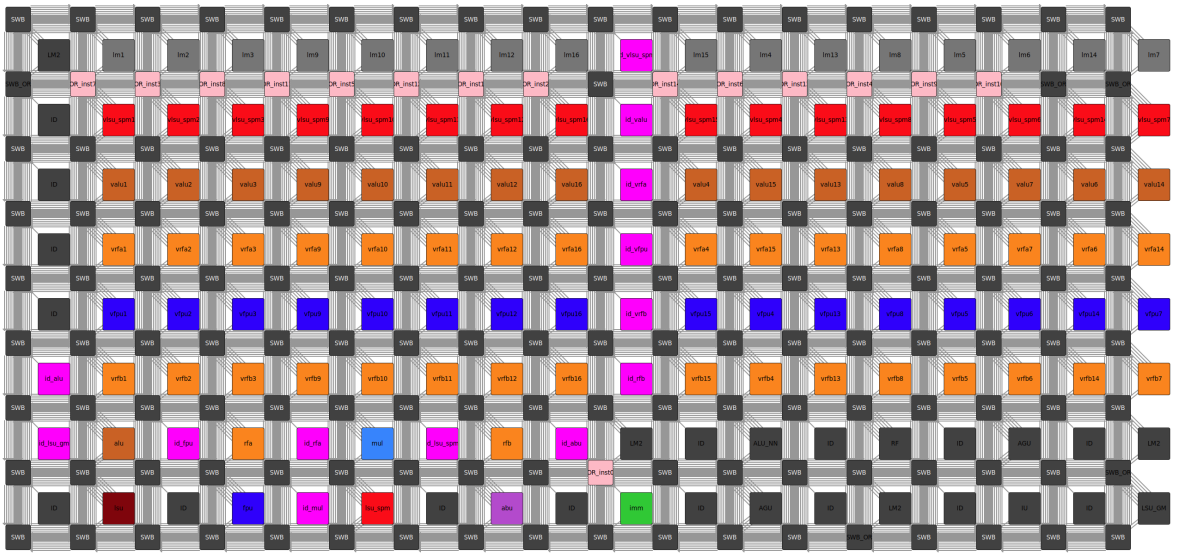


Figure 6.1.5: Vector16 Architecture

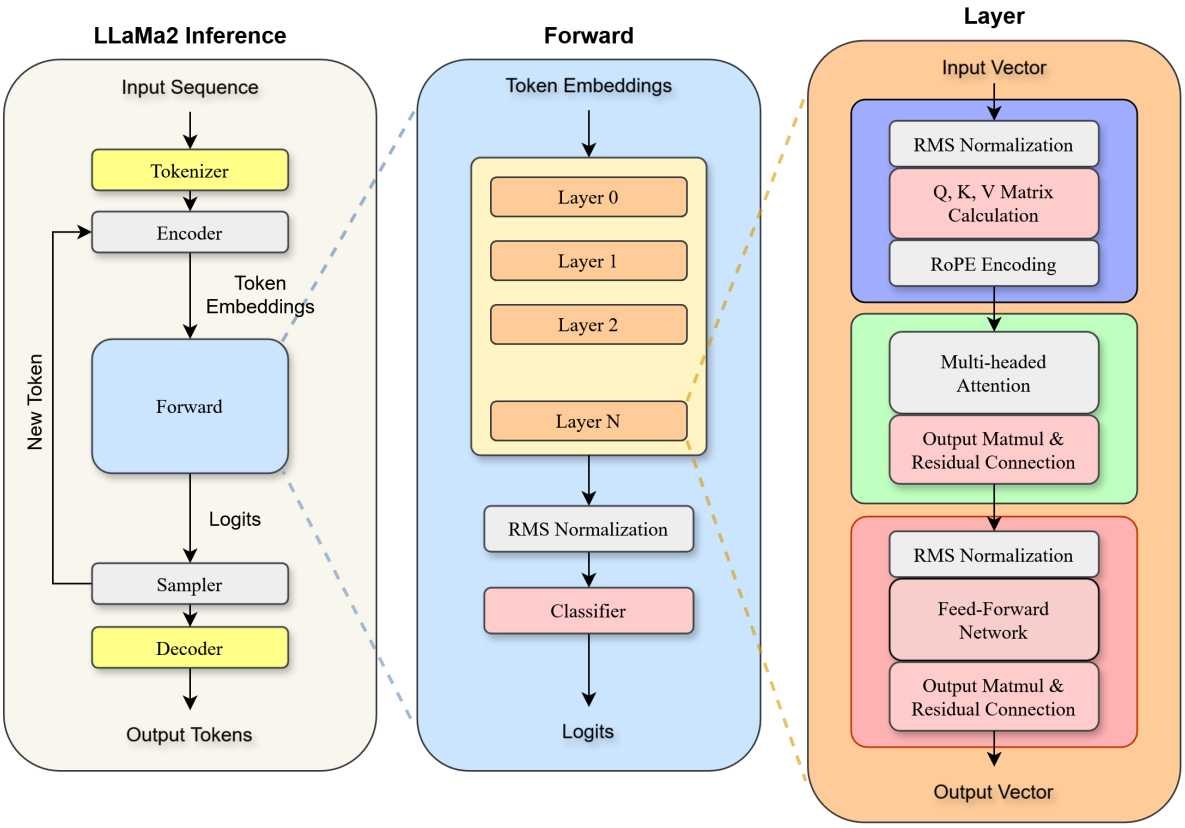


Figure 6.2.1: The LLaMa2 Inference Model

6.2 Benchmark Code Preparation

The next step in the R-Blocks mapping methodology is selecting the parts of the LLM that will be mapped to the reconfigurable fabric. In reality, this decision is made in tandem with the architecture definition process, so the HW/SW co-design character of this platform can be utilized at a maximum extent. Before selecting which sections will be mapped to the CGRA, a preliminary timing analysis of the LLaMa2 inference code was performed on 3 pre-trained models of different sizes.

Taking a closer look at the internal structure of the LLaMa2 model (Fig. 6.2.1), we notice that most of the calculations performed are matrix-vector multiplications (blocks colored in red). The exact percentages of runtime attributed to the matrix-vector multiplications in each of the blocks are shown in Figure 6.2.2. There, we notice that as the model size increases, the matrix-vector multiplication blocks dominate over the sampler and the rest of the constituent blocks of LLaMa2.

The benchmarks we ultimately decide to map to the CGRA, as a starting point of reference, are the following:

- Q, K, V matrix calculation (blue block)
- Attention block (green block)
- Pure matrix-vector multiplications of different dimensions

The reason for the selection of these specific benchmarks to be initially mapped is their combination of time consuming calculations, in the form of matrix-vector multiplications, and difficult to map non-linear functions, such as exponentials and trigonometric functions in the first two benchmarks. We wanted to extract good measurements about the performance of the CGRA when handling mostly

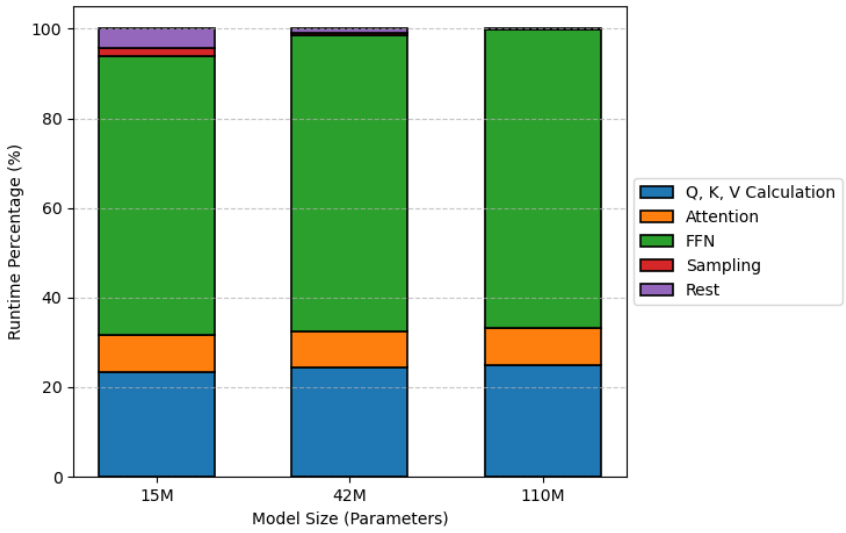


Figure 6.2.2: Runtime percentage of the LLaMa2 computational blocks on a CPU execution

repetitive loads of computations (matmul), and at the same time examine if mapping different non-linear functions in between these calculations would induce a significant performance overhead.

It should be noted that, as a proof of concept, a full mapping of the *forward* function of the LLaMa2 inference model has been achieved, but the architectures required to map it to real hardware must be scaled up to be able to execute it efficiently. Because of this reason and also limitations of the hardware and the tool-flow regarding memory sizing, we were only able to extract cycle results for this mapping from the TTA simulator.

The way these parts of the inference code were mapped, was by first stripping them of any dependencies not included in the R-Blocks source library files. This includes the weights of the models, which were directly loaded to the CGRA main memories from header files. From there, the weights can be accessed with simple pointers in the main code. Other luxuries such as the *printf* or *memcpy* functions had to be replaced with the native R-Blocks library functions that perform the same operations utilizing strictly TTA assembly operations.

After these changes, the benchmarks are ready to be mapped to the reconfigurable hardware, although without parallel execution capabilities.

6.3 Matrix Multiplication Vectorization

In order to be able to utilize the instantiated vector units in the Vector4, Vector8 and Vector16 architectures we must use the designated execution model of R-Blocks, which works by only mapping operations to the vector units if the operands have already been transferred to the local memory (LM) tiles.

In order to transfer data to the LM, we have to utilize the address spaces defined in the high-level code, and the TTA functions that copy memory blocks from one to the other. This is as simple as using the *memcpy* function in pure C, but with some additional arguments regarding the vector size of the used architecture.

After this process of fetching the data, and allocating them to vector buffers in the local memory, performing any operations between them will result in them being mapped to the corresponding vector

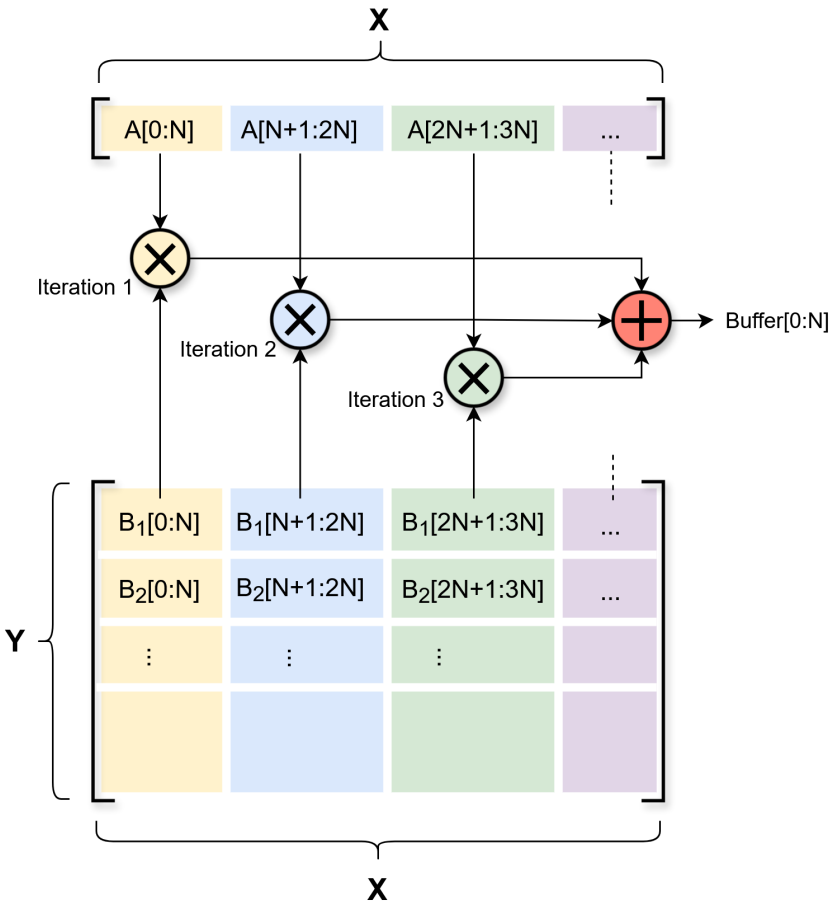


Figure 6.3.1: Matrix Multiplication Vectorization Scheme

FUs.

Since most of the calculations happen inside the MatMul (matrix-vector multiplication) function of the LLM benchmarks mapped, we create a new vectorized MatMul function that always takes advantage of the available vector FPU units.

This is done, simply by grouping the elements of lines to be multiplied and accumulated, in groups of N, where N is the vector size of the architecture, as illustrated in Figure 6.3.1. The inner loop of a classic matrix-vector multiplication $B[Y : X] \cdot A[X]$, which would normally be repeated X times, is now repeated X/N times, and in each of these iterations two vectors of N single-precision floating point numbers are multiplied element-wise and afterwards accumulated into a row buffer (of the same size N). After all the row elements have been multiplied with the corresponding elements of vector A, the N buffer elements are added up to form the final result of the row, and the same process is repeated for the next row of the matrix.

In each outer loop iteration, so once for each matrix row, the X elements of the matrix row have to be fetched to the LM tiles, where the elements of the A vector are also stored during the entire process to be reused. We choose not to fetch and store the entire matrix all at once, in order to keep the required LM size small. As will be discussed later, local memories take up a big percentage of the CGRA layout area and minimizing their size should be a prime concern for any designer.

Chapter 7

Results

Following the explanation of the mapping methodology, we will present the experimental results from the execution of the first LLM benchmarks on R-Blocks. Specifically, we will measure the clock cycles of each execution and the area and power of the circuits generated by our defined architectures. Additionally we will attempt to interpret and model our observations so more general conclusions can be drawn from this work.

7.1 Performance

Table 7.1 shows the execution cycles for the three chosen benchmarks when executed on each of the 3 architectures. The Q, K, V Matrix Calculation and the Attention Block contain 512x512 matrix-vector multiplications, so we chose to also present a pure matrix-vector multiplication with different dimensions (1376x512). These results were extracted at the stage of the RTL simulation, except for the Vector16 architecture for which they were extracted from the TTA cycle-accurate simulator, since the physical routing was impossible with the current routing algorithm, as discussed in chapter 6.

As can be observed in Fig. 7.1.1 which better visualizes the results, there is a steep decrease in cycles when utilizing parallel execution with vector units. Specifically, in all benchmarks we notice a reduction upwards of 60%, and on the case of pure matrix-vector multiplication (which is the benchmark fully utilizing parallel execution) upwards of 70%.

What is, perhaps, surprising, and will be analyzed in the next section is that the optimal speedup isn't always the result of the greatest vector size. This is counter-intuitive, since we would expect more operations happening in parallel to always reduce the number of cycles, even with diminishing returns.

Table 7.1: Performance Analysis in terms of Cycles

Benchmark	Architecture	Clock Cycles (Thousands)	% Improvement over Scalar
Q, K, V Matrix Calculation	Scalar	12.258	-
	Vector4	4.587	62,5
	Vector8	3.537	71,1
	Vector16	3.943	67,8
Attention Block	Scalar	4.175	-
	Vector4	1.606	61,5
	Vector8	1.273	69,5
	Vector16	1.391	66,7
Matrix-Vector Multiplication (1376x512)	Scalar	13.691	-
	Vector4	3.622	73,5
	Vector8	2.498	81,7
	Vector16	2.229	83,7

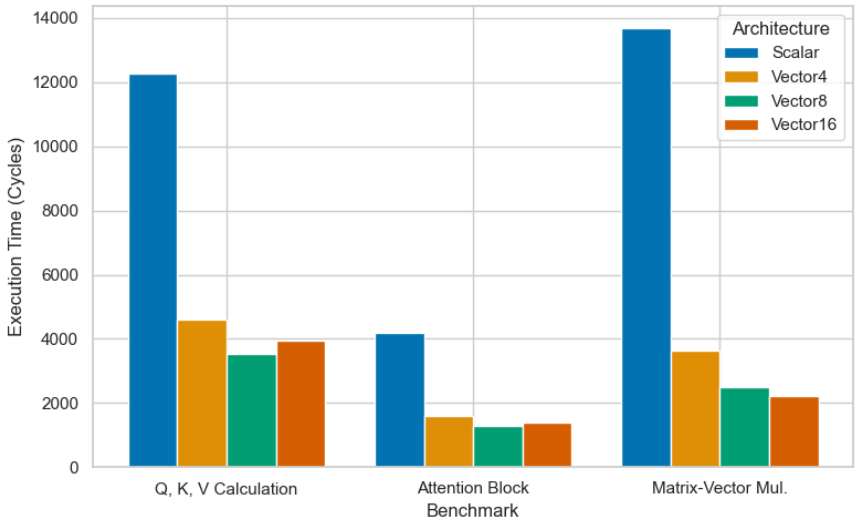


Figure 7.1.1: Cycles per benchmark for each tested architecture

This behavior is a result of inefficiencies in the TTA libraries for local memory access of elements and will be analyzed in the following section.

7.2 Cycle Modeling

In an attempt to model the timing behavior of the parallel multiplication function that we created for the mapping of the benchmarks onto the vector units, we mapped pure matrix-vector multiplication benchmarks with various dimension sizes to the previously presented vectorized architectures. The results are presented in Figure 7.2.1, with trendlines, showcasing the worsening that would result from further increasing the vector size.

These trendlines were not generated at random. They are the result of careful modeling that takes into account the dataflow of our *parallel matmul* function as showcased in the pseudocode below:

```
// vector is stored in local memory address A
// matrix is stored in global memory address B
// performing the multiplication B[y:x]*A[x] with N vector lanes
// result is stored in global memory address res

parallel_matmul(int x, int y, int N) {
    for(i from 0 to y) {                          // for every row of matrix B
        memcopy(row, B + i*x, size n, GM_TO_LM);  // copy x elements from global
            // memory to local memory, storing them in address "row"

        vector row_buffer = {0.0};                // initialize buffer to 0
        for(j from 0 to x/N) {
            val += row[j] * A[j];                 // row[j] is a vector of N floats
        }

        float val = 0.0;
        for(k from 0 to N) {
            val += vector_extract(row_buffer,k);  // add up the final elements of the
                // buffer
        }
```

```
        res[i] = val;
    }
}
```

In this function, it becomes clear which operations cost cycles to perform. There is a time cost M for copying elements from the R-Blocks global memory to the local memory tiles, so they can be used for parallel execution, another cost G for multiplying and accumulating the vector's elements, and a different cost E for extracting an element from the local memory to be accumulated in the final buffer and transferred back to the global memory.

All of these costs are modeled by the total cost function:

$$Cost = Y \cdot (X \cdot M + \frac{X}{N} \cdot G + \frac{N^2}{2} \cdot E) \tag{7.2.1}$$

This function, when we set the parameters M, G and E to values representing the real timing costs of our system, and by only changing the X and Y dimensions, closely models all of our experimental measurements from mapping different sizes of matrix multiplications on different architectures (Fig. 7.2.1). Therefore, we conclude that it is a good model of how the parallelization factor influences clock cycles in our benchmarks.

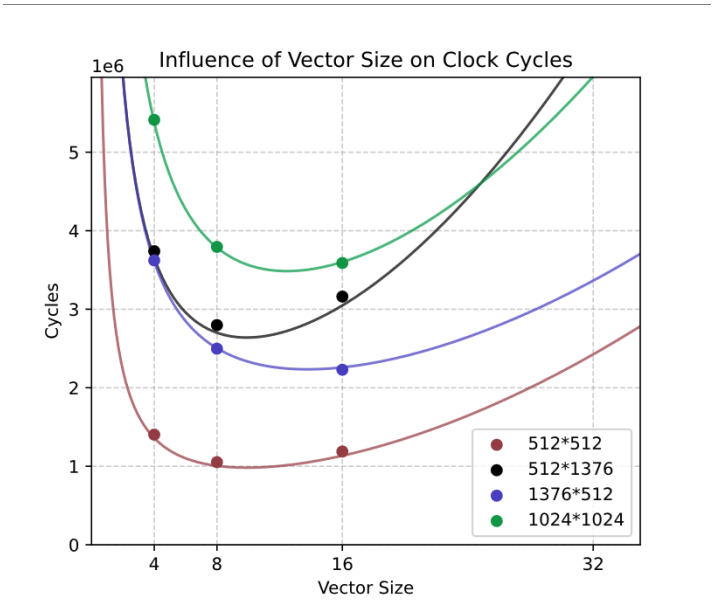


Figure 7.2.1: Illustration of how different matrix dimensions make use of parallelization

The problem is, that despite *parallel matmul* not being completely optimized in terms of dataflow, the main slowdown lies in the extraction of elements from the local memories. The current library functions in the R-Blocks development environment have glaring inefficiencies which cause them to have an $O(N)$ complexity relative to the vector size N, resulting in the $\frac{N^2}{2} \cdot E$ term in the cost function.

Additionally, the current implementation of R-Blocks (at least at a simulation level) has a much smaller global memory access time than it should, which also linearly skews the data. By adjusting both of those factors, in an attempt to model the cycles in a properly working system (regarding memory transfers) we increase the value of M and remove the quadratic dependency in the last term of the cost function. Now the cost of accessing a local memory element is linear and the projection in Figure 7.2.2 has a completely different behavior than the actual observations.

The projected cycles steadily decrease with the increase of vector size, but there is an elbow point, after which the diminishing returns cause the increase in hardware (parallel units) to not be justified.

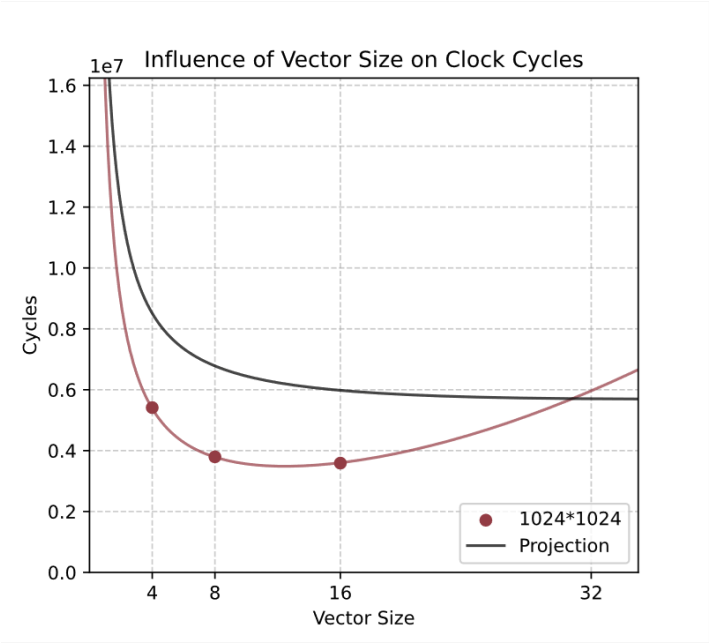
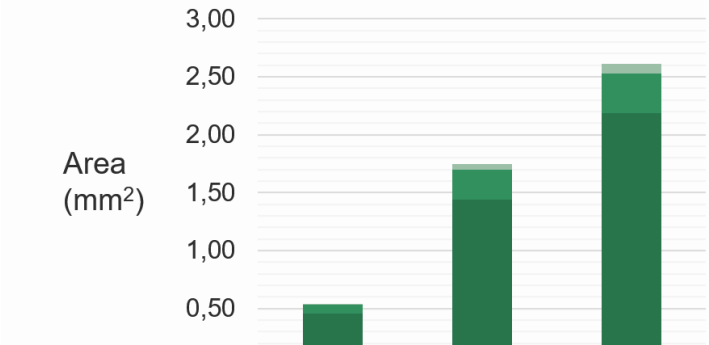


Figure 7.2.2: The projected cycles if the issues regarding memory accesses are dealt with

Actually fixing these inefficiencies is in the scope of future work, with research already being carried out in this direction.

7.3 Area and Power Analysis

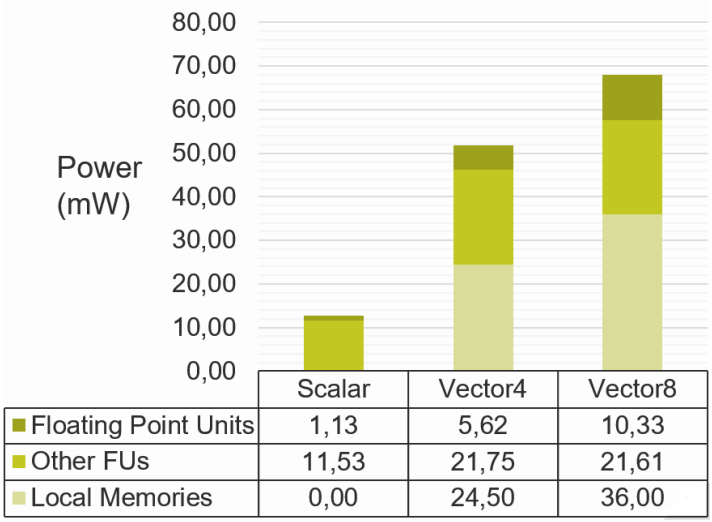
The bitstreams of the three architectures that were able to get mapped and routed in an RTL level grid, generated from the process described in chapter 5, were synthesized in 22nm FD-SOI technology with a maximum clock frequency of 200MHz determined experimentally.



	Scalar	Vector4	Vector8
Floating Point Units	0,01	0,05	0,08
Other FUs	0,0756	0,2587	0,3393
Memories	0,45	1,4383	2,187

Figure 7.3.1: Area analysis of the final IC

The resulting area and power numbers produced by the Synopsys Design Compiler are summarized in Figures 7.3.1 and 7.3.2. In Figure 7.3.1 it can be observed that most of the chip area is occupied



	Scalar	Vector4	Vector8
Floating Point Units	1,13	5,62	10,33
Other FUs	11,53	21,75	21,61
Local Memories	0,00	24,50	36,00

Figure 7.3.2: Power analysis of the final IC

by memories. These consist of instruction memories contained in each of the CGRA's fetch units but, at a larger extent, they are the Local Memory tiles themselves. During design we strove to keep the required local memory address space small, in order to prevent the memories from dominating spatially. However, even with the strict constraints enforced, this is still an issue on our future work radar.

The rest of the FUs including the FPUs scale linearly in terms of area, which is expected, and the FPU does not induce a big area overhead, which deems the integration successful, at least by this metric.

In terms of power consumption, Figure 7.3.2 tells the story of how including and utilizing more local memory tiles costs more than operating more FPUs (or more FUs for that matter). In the Scalar architecture which makes no use of local memories, their power consumption is naturally zero, whereas by including as many LM tiles as the vector unit size, in architectures Vector4 and Vector8, the power rises dramatically. We can see that in both architectures more than 50% of the power is attributed to local memory transfers, while the functional units' consumption scales linearly.

The FPU tile accounts for approximately 10-15% of power usage, which is reasonable, considering most of the operations happening are FPU dependent.

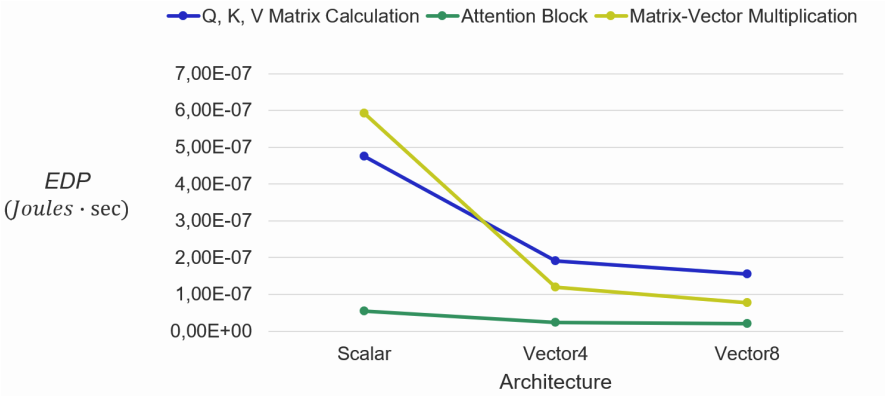


Figure 7.3.3: Energy-Delay Product per Benchmark

Since we are interested in the energy efficiency of our design, and the power consumption of the IC, the clock frequency and the total number of clock cycles for our specific benchmarks are all known, we

can use better metrics such as the Energy-Delay Product (EDP) to evaluate it.

Figure 7.3.3 illustrates the EDP trends for our 3 benchmarks as the degree of parallelization increases.

Interpreting these results is straightforward. The reduction in clock cycles due to parallelization is significantly steeper than the corresponding increase in power consumption. As a result, the Energy-Delay Product (EDP) decreases with higher degrees of parallelization, though with diminishing returns beyond a certain point. This inflection point suggests that increasing the parallelization degree beyond four offers little benefit in terms of efficiency and, given the current approach to accessing local memory, may even be detrimental. Additionally, the increase in area cost when scaling from a vector unit size of four to eight further reinforces that Vector4 represents the optimal trade-off for our current workloads.

A final observation is the significant impact of computation type on cycle reduction. Benchmarks composed entirely of matrix-vector multiplications, particularly those with larger dimensions that fully leverage parallel execution, exhibit a more substantial decrease in EDP. In contrast, benchmarks that include both matrix-vector multiplications and nonlinear functions, which cannot be optimized within this setup, experience a comparatively smaller improvement.

Chapter 8

Conclusion

Closing up this thesis, I would like to summarize the most valuable insights about the inner workings of LLMs on the R-Blocks platform, restate this work's contributions, but mostly provide guidelines on future research avenues relating to this field.

8.1 R-Blocks Development

While being able to compile and run LLMs (or any other FP-dependent benchmarks) on the R-Blocks co-design environment is a significant milestone, it also brings to light many of the platform's shortcomings and quirks.

Some of them we touched on, during the explanation of this work, such as the inefficiencies relating to local memory accesses, the limitations regarding memory sizing and area, or the tool-flow optimizations that are due (e.g. routing algorithm). Other issues either remain for future researchers to address or were resolved in this work, but mentioning them could lead to unnecessary tangents that might detract from a clear and focused presentation.

One of the best things to come from this work, I personally believe, is the illumination of the uncharted territory that is the expansion of the R-Blocks co-design environment. In order to expand the tools and hardware, one must first fully understand them. This understanding, and the methodologies and procedures developed in this work will, hopefully, help guide other researchers to bigger and better accomplishments.

8.2 LLMs on CGRAs

The main question we set out to answer was whether a CGRA platform such as R-Blocks could be considered as an edge accelerator for LLM inference. Although a lot of progress was made towards answering it, a final conclusive answer can not be given. Looking at the bigger picture, it is hard to see CGRAs being a commercially viable candidate for this specific task in the near future.

CGRAs shine as domain specific accelerators for tasks that require some degree of circuit parameterization to be executed efficiently. Although one could exploit some of the characteristics of LLMs to make use of this property to some extent, there are still more viable alternatives with a head start in terms of research and community support. GPUs have been heralded as the premier competitor in this scene for years, with continuous hardware and software development backed by industry giants.

Despite all these, CGRAs, and specifically R-Blocks, offer a completely different approach; one that focuses on ultra low power and efficient utilization of HW/SW co-design. As such, nobody can rule them out as an edge alternative with certainty. And even then, trying to map this kind of applications gives us valuable insights into their limitations, constraints and opportunities.

8.3 Future Work

8.3.1 Continuing the Research on LLM mapping

This research opens up a lot of possible directions for future work. In order to expand this work specifically towards the direction it is headed, one would first have to iron out a lot of the hardware and tool-flow issues previously mentioned. Afterwards, with better simulation support, less constrictive memory limits and the potential for bigger architectures, a researcher would have all the tools required to optimize the mapping of LLMs on this platform. Many aspects of the dataflow can still be improved upon, such as data reuse, redundancy elimination and memory management.

These optimizations will perhaps achieve energy efficiency metrics that push R-Blocks close to the state-of-the-art, or more realistically provide even greater insights to what the bottleneck is on LLM inference and how it could be overcome in similar architectures.

8.3.2 Approximate Computing

Another avenue of exploration, paved by the knowledge gained about expanding the R-Blocks environment, is infusing the co-design process with approximate computing elements. This could come in the form of new Functional Units with approximate capabilities and their own ISA, which is a direction already explored by the Convolve project team at Microlab, NTUA, with promising preliminary results. It could also come in the form of approximate algorithmic techniques that would be incorporated and efficiently mapped to the reconfigurable hardware, perhaps in conjunction with the approximate units.

The possibilities are truly endless with a roadmap pointing the way towards the expansion and implementation of new R-Blocks tiles.

8.3.3 Architectural DSE

Additionally, a very underrated asset of the R-Blocks co-design environment is the ease with which a programmer can define and experiment with different architectures. Being able to design and program the reconfigurable tiles in any way imaginable gives the engineer the freedom to create hardware truly fit for the software mapped to it.

In our exploration, which was mostly a proof of concept for LLM architectural exploration, we only looked into the differences in vector unit sizes, while keeping the basic structure of the scalar architecture untouched. What if more scalar FPU tiles could deliver a bigger speedup with less power consumption? What if the interconnect network could be relieved of the congestion by including more register file FUs?

All these questions can be answered relatively easily with the manual architectural design approach we followed in this work, but this search could also be augmented with methodical automation. The prunner tool mentioned in chapter 5 already provides some automation capabilities, and the entire flow could also be expanded to include automatic architecture generation, which could perhaps also be software- and resource-aware.

This would mean that the architecture generator would take into consideration all of the design limitations we wrestled with and would produce an architecture optimized for a specific benchmark (perhaps with help from the programmer, with high-level directives), and with area and power constraints in mind.

8.3.4 Compiler Support

The exploration principles described above, of course, rely on the high-level architecture targeting the compiler to work. This compiler, as explained before, was designed for Transport Triggered Architectures, and R-Blocks is only one of the platforms making use of it. This means that it is not entirely fitted for compiling code on R-Blocks architectures efficiently, and could use a lot of optimizations.

Experimental observations suggest that manually optimized assembly, directly executed on R-Blocks, could achieve a 2x speedup compared to compiler-generated code. This leaves a lot of performance on the table since multi-stage pipelining and parallel execution can't be fully taken into consideration by the scheduler and end up sub-utilized in the final code.

Since fully swapping out the compiler is time costing, future research could focus on improving the current library support and underlying scheduling decisions underpinning all of this development.

8.3.5 Final Words

For anyone interested in continuing this promising research, know that a treasure of knowledge awaits, and I hope that you recieve as much help and guidance as I did from the members of Microlab, NTUA, who I would like to, wholeheartedly, thank again for their invaluable contribution.

Bibliography

[1] A. Vaswani and N. S. et al., "Attention is all you need," 2017.

[2] T. B. Brown, B. Mann, and N. R. et al., "Language models are few-shot learners," 2020.

[3] H. Touvron and T. L. et al., "Llama: Open and efficient foundation language models," 2023.

[4] DeepSeek-AI, D. Guo, and D. Y. et al., "Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning," 2025.

[5] S. Jamil, M. J. Piran, and O.-J. Kwon, "A comprehensive survey of transformers for computer vision," 2022.

[6] S. Latif, A. Zaidi, H. Cuayahuitl, F. Shamshad, M. Shoukat, and J. Qadir, "Transformers in speech processing: A survey," 2023.

[7] J. Abramson, J. Adler, J. Dunger, R. Evans, T. Green, A. Pritzel, O. Ronneberger, L. Willmore, A. J. Ballard, J. Bambrick, *et al.*, "Accurate structure prediction of biomolecular interactions with alphafold 3," *Nature*, pp. 1–3, 2024.

[8] G. Theodoridis, D. Soudris, and S. Vassiliadis, "A survey of coarse-grain reconfigurable architectures and cad tools: Basic definitions, critical design issues and existing coarse-grain reocnfigurable systems," *Fine-and Coarse-Grain Reconfigurable Computing*, pp. 89–149, 2007.

[9] B. de Bruin, K. Vadivel, M. Wijtvliet, P. Jääskeläinen, and H. Corporaal, "R-blocks: an energy-efficient, flexible, and programmable cgra," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 17, May 2024.

[10] P. Jääskeläinen, T. Viitanen, J. Takala, and H. Berg, *HW/SW Co-design Toolset for Customization of Exposed Datapath Processors*, pp. 147–164. Springer International Publishing, 2017.

[11] M. Wijtvliet, J. Huisken, L. Waeijen, and H. Corporaal, "Blocks: Redesigning coarse grained reconfigurable architectures for energy efficiency," in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 17–23, 2019.

[12] L. Liu, J. Zhu, Z. Li, Y. Lu, Y. Deng, J. Han, S. Yin, and S. Wei, "A survey of coarse-grained reconfigurable architecture and design: Taxonomy, challenges, and applications," *ACM Comput. Surv.*, vol. 52, Oct. 2019.

[13] H. Corporaal, *Microprocessor Architectures: From VLIW to Tta.* USA: John Wiley & Sons, Inc., 1997.

[14] H. Touvron and L. M. et al., "Llama 2: Open foundation and fine-tuned chat models," 2023.

[15] R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao, S. Hadjis, A. Pedram, C. Kozyrakis, and K. Olukotun, "Plasticine: A reconfigurable architecture for parallel paterns," *SIGARCH Comput. Archit. News*, vol. 45, p. 389–402, june 2017.

[16] D. Koeplinger, R. Prabhakar, Y. Zhang, C. Delimitrou, C. Kozyrakis, and K. Olukotun, "Automatic generation of efficient accelerators for reconfigurable hardware," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 115–127, 2016.

[17] O. Akbari, M. Kamal, A. Afzali-Kusha, M. Pedram, and M. Shafique, "X-cgra: An energy-efficient approximate coarse-grained reconfigurable architecture," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 10, pp. 2558–2571, 2020.

[18] S. A. Chin, N. Sakamoto, A. Rui, J. Zhao, J. H. Kim, Y. Hara-Azumi, and J. Anderson, "Cgra-me: A unified framework for cgra modelling and exploration," in *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pp. 184–189, 2017.

[19] O. Ragheb, S. Wicklund, M. Walker, R. Beidas, A. Ragab, T. Yu, and J. Anderson, "Cgra-me 2.0: A research framework for next-generation cgra architectures and cad," in *2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 642–649, 2024.

[20] Y. Luo, C. Tan, N. B. Agostini, A. Li, A. Tumeo, N. Dave, and T. Geng, "Ml-cgra: An integrated compilation framework to enable efficient machine learning acceleration on cgras," in *2023 60th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2023.

[21] H. Uetani and Y. Nakashima, "Implementation and evaluation of llm on a cgla," in *2024 Twelfth International Symposium on Computing and Networking (CANDAR)*, pp. 252–258, 2024.

[22] Y. Mao, X. Gao, J. Lou, Y. Qiu, W. Yin, W.-S. Luk, and L. Wang, "Cfeact: A cgra-based framework enabling agile cnn and transformer accelerator design," in *2024 34th International Conference on Field-Programmable Logic and Applications (FPL)*, pp. 213–219, 2024.

[23] R. Usselmann, "Opencores fpu." https://opencores.org/projects/fpu, 2009. [Online; accessed 4-March-2025].

[24] J. Hauser, "Berkeley softfloat." http://www.jhauser.us/arithmetic/SoftFloat.html, 2017. [Online; accessed 5-March-2025].

[25] K. S. Kalyan, A. Rajasekharan, and S. Sangeetha, "Ammus : A survey of transformer-based pretrained models in natural language processing," 2021.

[26] N. Rane, "Transformers in material science: Roles, challenges, and future scope," *Challenges, and Future Scope (March 26, 2023)*, 2023.

[27] N. Geneva and N. Zabaras, "Transformers for modeling physical systems," *Neural Networks*, vol. 146, pp. 272–289, 2022.

[28] J. Jiang, L. Ke, L. Chen, B. Dou, Y. Zhu, J. Liu, B. Zhang, T. Zhou, and G.-W. Wei, "Transformer technology in molecular science," *WIREs Computational Molecular Science*, vol. 14, no. 4, p. e1725, 2024.

[29] R. Hartenstein, A. Hirschbiel, M. Riedmuller, K. Schmidt, and M. Weber, "A novel asic design approach based on a new machine paradigm," *Solid-State Circuits, IEEE Journal of*, vol. 26, pp. 975 – 989, 08 1991.

[30] S. H. Gerez, S. M. H. de Groot, E. R. Bonsma, and M. J. M. Heijligers, *Overlapped Scheduling Techniques for High-Level Synthesis and Multiprocessor Realizations of DSP Algorithms*, pp. 125–150. Boston, MA: Springer US, 1998.

[31] B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins, "Adres: An architecture with tightly coupled vliw processor and coarse-grained reconfigurable matrix," in *International Conference on Field-Programmable Logic and Applications*, 2003.

[32] V. Baumgarte, G. Ehlers, F. May, A. Nückel, M. Vorbach, and M. Weinhardt, "Pact xpp—a self-reconfigurable data processing architecture," *J. Supercomput.*, vol. 26, p. 167–184, Sept. 2003.

[33] M. A. Watkins, T. Nowatzki, and A. Carno, "Software transparent dynamic binary translation for coarse-grain reconfigurable architectures," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 138–150, 2016.

[34] N. Clark, M. Kudlur, H. Park, S. Mahlke, and K. Flautner, "Application-specific processing on a general-purpose core via transparent instruction set customization," in *37th International Symposium on Microarchitecture (MICRO-37'04)*, pp. 30–40, 2004.

[35] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Transactions on Computers*, vol. C-21, no. 9, pp. 948–960, 1972.

[36] M. Budiu and S. C. Goldstein, "Pegasus: An efficient intermediate representation," 2002.

[37] M. Budiu and S. C. Goldstein, *Spatial computation*. PhD thesis, USA, 2003. AAI3126922.

[38] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, p. 666–677, Aug. 1978.

[39] G. Kahn, "The semantics of a simple language for parallel programming," in *IFIP Congress*, 1974.

[40] K. Vadivel, B. De Bruin, R. Jordans, H. Corporaal, and P. Jääskeläinen, "Prebypass: Software register file bypassing for reduced interconnection architectures," in *2022 25th Euromicro Conference on Digital System Design (DSD)*, pp. 157–164, 2022.

[41] K. Vadivel, R. Jordans, S. Stujik, H. Corporaal, P. Jääskeläinen, and H. Kultala, "Towards efficient code generation for exposed datapath architectures," in *Proceedings of the 22nd International Workshop on Software and Compilers for Embedded Systems*, SCOPES '19, (New York, NY, USA), p. 86–89, Association for Computing Machinery, 2019.

[42] X. Amatriain, A. Sankar, J. Bing, P. K. Bodigutla, T. J. Hazen, and M. Kazi, "Transformer models: an introduction and catalog," 2024.

[43] R. Eldan and Y. Li, "Tinystories: How small can language models be and still speak coherent english?," *arXiv preprint arXiv:2305.07759*, 2023.

[44] A. Karpathy, "karpathy/llama2.c." https://github.com/karpathy/llama2.c, 2023. [Online; accessed 27-February-2025].

[45] R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao, S. Hadjis, A. Pedram, C. Kozyrakis, and K. Olukotun, "Plasticine: A reconfigurable architecture for parallel paterns," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, (New York, NY, USA), p. 389–402, Association for Computing Machinery, 2017.

[46] R. Prasad, "An ultra-low-power CGRA for accelerating Transformers at the edge." working paper or preprint, Jan. 2025.

[47] J. Li, Y. Dai, Y. Hu, J. Li, W. Yin, J. Tao, and L. Wang, "Transmap: An efficient cgra mapping framework via transformer and deep reinforcement learning," in *2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 626–633, 2024.