ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

A Novel Reconfigurable Out-of-Order GPU Microarchitecture with Runtime Workload Characterization

---

Συγγραφέας:
Παναγιώτης-Ελευθέριος Ελευθεράκης

Επιβλέπων:
Δημήτριος Σούντρης
Καθηγητής

ΕΡΓΑΣΤΗΡΙΟ ΜΙΚΡΟΥΠΟΛΟΓΙΣΤΩΝ ΚΑΙ ΨΗΦΙΑΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

21 Μαρτίου 2023

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ΕΡΓΑΣΤΗΡΙΟ ΜΙΚΡΟΥΠΟΛΟΓΙΣΤΩΝ ΚΑΙ ΨΗΦΙΑΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

A Novel Reconfigurable Out-of-Order GPU Microarchitecture with Runtime Workload Characterization

---

Συγγραφέας:
Παναγιώτης-Ελευθέριος Ελευθεράκης

Επιβλέπων:
Δημήτριος Σούντρης
Καθηγητής

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 21η Μαρτίου 2023.

(Υπογραφή)	*(Υπογραφή)*	*(Υπογραφή)*
Δημήτριος Σούντρης Καθηγητής	Παναγιώτης Τσανάκας Καθηγητής	Σωτήριος Ξύδης Αναπληρωτής Καθηγητής

21 Μαρτίου 2023

Πνευματικά Δικαιώματα

Υπογραφή:

Ημερομηνία:

Εθνικό Μετσόβιο Πολυτεχνείο

Περίληψη

Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Διπλωματική Εργασία

A Novel Reconfigurable Out-of-Order GPU Microarchitecture with Runtime Workload Characterization

του Παναγιώτη-Ελευθέριου Ελευθεράκη

Στη σύγχρονη εποχή, δεδομένης της αποτελμάτωσης του νόμου του Moore, η αύξηση της υπολογιστικής επίδοσης προκύπτει από τη μαζική παραλληλία και την εξειδίκευση του υλικού. Με την κατάρρευση της κλιμάκωσης του Dennard και την έλευση της "Εποχής του σκοτεινού πυριτίου" καθίσταται αναγκαία η υψηλή ενεργειακή απόδοση στους επεξεργαστές. Σε αυτό το πλαίσιο, οι ετερογενείς και οι αναδιαμορφώσιμες αρχιτεκτονικές έχουν αναδειχθεί ως ευέλικτες προσεγγίσεις για την επίτευξη των παραπάνω στόχων.

Το ήδη προταθέν σχήμα εκτέλεσης Light-weight Out-of-Order GPU (LOOG) αντιμετωπίζει τη στασιμότητα σε επίδοση που χαρακτηρίζει μια κατηγορία εφαρμογών GPU γενικού σκοπού, συμπληρώνοντας την παραδοσιακή αξιοποίηση της παραλληλίας επιπέδου νήματος (TLP) και της γρήγορης εναλλαγής περιβάλλοντος της GPU, με την εκμετάλλευση της εγγενούς παραλληλίας επιπέδου εντολών (ILP) αυτών των εφαρμογών. Καθώς αποτελεί τη βάση της παρούσας εργασίας, το υλοποιούμε στην πιο πρόσφατη έκδοση του Accel-Sim, ενός πλαισίου προσομοίωσης GPU που βασίζεται στο μοντέλο επίδοσης του GPGPU-Sim, ενός προσομοιωτή επίδοσης GPU σε επίπεδο κύκλου.

Έχοντας προσαρμόσει το LOOG σε μία πλατφόρμα υψηλής υπολογιστικής επίδοσης (NVIDIA Quadro GV100, που τροφοδοτείται από τη μικροαρχιτεκτονική Volta) με τη σωστή διαστασιολόγηση των δομών του, την εφαρμογή ενός δυναμικού μηχανισμού αναδιαμόρφωσης του ρυθμιστικού διαύλου εντολών και τη βέλτιστη διαμόρφωση των στοιχείων front-end της GPU, συλλέγουμε λεπτομερή στατιστικά στοιχεία για τα σημεία συμφόρησης της αρχιτεκτονικής σε 7 σύνολα μετροπρογραμμάτων και 100 εφαρμογές (CUDA kernels).

Ο αναδυόμενος χαρακτηρισμός των εφαρμογών και η μελέτη των χαρακτηριστικών τους που προβλέπουν την επιτάχυνση στο LOOG, σε συνδυασμό με την ανάλυση της κλιμακωσιμότητας των στοιχείων LOOG από αρχιτεκτονική άποψη, παρακινεί την αξιολόγηση μιας κλιμακούμενης, αναδιαμορφώσιμης μικροαρχιτεκτονικής GPU εκτός σειράς, η οποία χειρίζεται κατάλληλα τόσο εφαρμογές που θεωρούνται ευαίσθητες στο LOOG όσο και γενικές εφαρμογές, για τη μεγιστοποίηση της επίδοσης ή της ενεργειακής απόδοσης.

Η αναδιαμορφώσιμη μικροαρχιτεκτονική αξιολογείται υπό διαφορετικά σχήματα και επίπεδα αναδιαμόρφωσης, συμπεριλαμβανομένου ενός ελεγκτή αναδιαμόρφωσης επιπέδου κλήσης συνάρτησης στο υλικό, χρησιμοποιώντας μετρητές επιδόσεων κατά τη διάρκεια της εκτέλεσης για την πρόβλεψη της βελτίωσης επίδοσης της εφαρμογής σε εκτέλεση εκτός σειράς. Μια στατική κλιμακωμένη διαμόρφωση LOOG παρέχει επιτάχυνση 1,48 για γενικές εφαρμογές και μείωση της εκλυόμενης ενέργειας κατά 13,7% σε σύγκριση με τη βασική αρχιτεκτονική (εντός σειράς). Η αναδιαμόρφωση με οδηγίες λογισμικού και η χρήση του ελεγκτή υλικού μπορούν να παρέχουν την ίδια επιτάχυνση όταν απαιτείται και έχουν τη δυνατότητα να βελτιώσουν την ενεργειακή απόδοση σε σχέση με τη βασική αρχιτεκτονική κατά 22,4% και 19,5% αντίστοιχα.

Λέξεις κλειδιά: GPU Γενικού Σκοπού, Υψηλή Υπολογιστική Επίδοση, Αναδιαμορφώσιμες Αρχιτεκτονικές, Παραλληλία επιπέδου εντολής, Εκτός Σειράς Αρχιτεκτονική, Παρλάλληλα Συστήματα, Ενεργειακή Απόδοση Υπολογισμού, Μοντελοποίηση και Προσομοίωση.

National Technical University of Athens

Abstract

Division of Computer Science
School of Electrical And Computer Engineering

Diploma Thesis

A Novel Reconfigurable Out-of-Order GPU Microarchitecture with Runtime Workload Characterization

by Panagiotis-Eleftherios Eleftherakis

Since the breakdown of Moore's law, high processor performance has been driven by Massively Parallel Processing and hardware specialization. The halt met by Dennard's scaling and the advent of the "Dark Silicon Era" necessitate energy-efficient computing. In this context, heterogeneous architectures and reconfigurable computing have emerged as flexible approaches for achieving the above goals.

Meanwhile, the previously proposed Light-weight Out-of-Order GPU (LOOG) execution scheme addresses the performance stagnation met by a class of general-purpose GPU workloads, by complementing the traditional TLP leveraging and fast context switching of the GPU, with exploitation of the inherent Instruction Level Parallelism (ILP) of these workloads. As it constitutes the backbone of this thesis, we implement it in the most recent version of Accel-Sim, a GPU simulation framework that provides modelling of recent high-end NVIDIA GPU architectures, built around the performance model of GPGPU-Sim 4.1.0, a cycle-level GPU performance simulator.

Having accommodated LOOG on an HPC-relevant platform (NVIDIA Quadro GV100, powered by the Volta microarchitecture) by right-sizing its structures, implementing a dynamic Instruction Buffer reconfiguration mechanism and optimally configuring GPU pipeline front-end components, we collect detailed architecture bottleneck statistics across 7 benchmark suites and 100 CUDA kernels.

The emerging application characterization and the study of workload characteristics that predict speedup on LOOG, paired with a scalability analysis of LOOG components from an architectural standpoint, motivates the assessment of a Scalable, Reconfigurable Out-of-Order GPU Microarchitecture that appropriately handles both kernels deemed LOOG-sensitive as well as generic kernels, to maximize performance or energy efficiency.

The reconfigurable microarchitecture is evaluated under different reconfiguration schemes and granularities, including a per-kernel-launch granularity hardware reconfiguration controller using runtime performance counters to predict application OOO performance improvement. A static scale-up LOOG configuration provides a speedup of 1.48 for generic kernels and a 13.7% reduction in energy dissipation, compared to the baseline (in-order) architecture. Reconfiguration under programmer-assisted directives and using the hardware controller can provide the same speedup when needed and have the potential to improve energy efficiency from baseline by 22.4% and 19.5% respectively.

Keywords: General-Purpose GPU, High Performance Computing, Reconfigurable computing, Instruction Level Parallelism, Out-of-Order, Parallel Systems, Energy Efficient computing, Modelling and Simulation.

Acknowledgements

The submission of this thesis marks the end of my undergraduate studies at the School of Electrical and Computer Engineering of the National Technical University of Athens.

First and foremost, I would like to express my profound and sincere thanks to Professor Dimitrios Soudris for giving me the opportunity to carry out my diploma thesis under his supervision as well as his inspiring approachability and guidance.

In addition, I would like to thank Professor Sotirios Xydis for his exceptional instruction and the vast comprehensive knowledge and expertise he shared with me regarding all aspects of my project and in all stages of it.

I extend my sincerest appreciation to Dr. Konstantinos Iliakis for being consistently available to provide me with invaluable advice and support me in every difficulty I faced. His innovative ideas set the track for my thesis and were instrumental in shaping its outcome.

Finally, I would like to thank those closest to me for their patience and support while carrying out this project.

Contents

List of Figures

List of Tables

Κεφάλαιο 1

Εκτεταμένη Ελληνική Περίληψη

1.1 Εισαγωγή

GPU Γενικού Σκοπού

Τις τελευταίες δεκαετίες λαμβάνει χώρα η εξέλιξη των Graphics Processing Units (GPU) από εξειδικευμένο υλικό για την απόδοση εφαρμογών γραφικών σε ευέλικτη χρήση για μη γραφικές υπολογιστικές διεργασίες, όπως επιστημονικές προσομοιώσεις, κρυπτογραφία και μηχανική μάθηση [1] . Οι GPU προσφέρουν σημαντικά κέρδη επίδοσης και ενεργειακής απόδοσης όταν εκτελούν υπολογιστικά εντατικά, μαζικά παράλληλα τμήματα μιας εφαρμογής.

Έτσι, έχουν καταστεί κυρίαρχες στο χώρο των επιταχυντών, ιδίως σε κέντρα υπερκλιμακωτής επεξεργασίας δεδομένων που επιταχύνουν εφαρμογές μηχανικής μάθησης με τεράστιο όγκο δεδομένων και παραλληλισμό [2, 3] .

Ετερογενείς και αναδιαμορφώσιμες αρχιτεκτονικές

Ο κορεσμός της επίδοσης ενός νήματος καθώς και το ευρύ φάσμα εξειδεικευμένων χαρακτηριστικών των εφαρμογών οδήγησε στην ανάπτυξη αναδιαμορφώσιμων και ετερογενών αρχιτεκτονικών για τη βελτιστοποίηση συγκεκριμένων εφαρμογών και τη μεγιστοποίηση της επίδοσης και της ενεργειακής αποδοτικότητας, συνδυάζοντας την ευελιξία των γενικού σκοπού επεξεργαστών με τις επιδόσεις των επεξεργαστών εξειδικευμένων εφαρμογών σε μια ενιαία συσκευή που μπορεί να προσαρμόζεται δυναμικά σε διαφορετικές εφαρμογές [4].

Οι αρχιτεκτονικές που περιλαμβάνουν πυρήνες με ετερογενή χαρακτηριστικά περιορίζονται από τον αμετάβλητο σχεδιασμό τους ανά πυρήνα. Οι αναδιαμορφώσιμες αρχιτεκτονικές λεπτομερούς (FPGA) και αδρομερούς (CGRA) αναδιαμόρφωσης [4, 5] αποτελούν μια εναλλακτική λύση, αλλά έχουν περιορισμούς, όπως η επιβάρυνση αναδιαμόρφωσης και οι περιορισμένοι πόροι [6].

Για την αντιμετώπιση αυτών των ζητημάτων έχουν προταθεί αναδιαμορφώσιμες αρχιτεκτονικές τσιπ-πολυεπεξεργαστών (reconfigurable CMP) [7] , οι οποίες επιτρέπουν τη δυναμική αναδιαμόρφωση με κατακόρυφη (scale-up) ή οριζόντια (scale-out) κλιμάκωση με τη χρήση λειτουργιών συγχώνευσης και διάσπασης πυρήνων.

Οι κλιμακούμενες, μερικώς αναδιαμορφώσιμες αρχιτεκτονικές, όπως ο MorphCore [8] , χρησιμοποιούν μεγάλους πυρήνες εκτός σειράς, βελτιστοποιημένους για ακολουθιακό κώδικα με ένα νήμα και αξιοποιούν τον παραλληλισμό σε επίπεδο εντολών, σε συνδυασμό με τη δυνατότητα μετάβασης σε ένα σχήμα εκτέλεσης SMT με υψηλό βαθμό παραλληλισμού σε επίπεδο νήματος.

΄Αλλες κλιμακούμενες αρχιτεκτονικές, όπως η Dynamic Core Boosting, Elastic core και το Flicker [9, 10, 11] , προσαρμόζονται στις απαιτήσεις της εφαρμογής κατά τη διάρκεια του χρόνου εκτέλεσης είτε με δυναμική κλιμάκωση είτε με δυναμικά διαμορφούμενη ετερογένεια ανάμεσα στους πυρήνες, χρησιμοποιώντας αποκοπή ρολογιού ή ισχύος και δυναμική κλιμάκωση τάσης και συχνότητας (Dynamic Frequency and Voltage scaling) για την ενίσχυση των επιδόσεων και την ελαχιστοποίηση της επιβάρυνσης ισχύος [8] .

Σχήμα εκτέλεσης LOOG

Οι GPU χρησιμοποιούν μαζικό παραλληλισμό σε επίπεδο νήματος και γρήγορη εναλλαγή περιβάλλοντος μεταξύ μεγάλων ομάδων νημάτων για να επιτύχουν υψηλή υπολογιστική επίδοση. Ωστόσο, ορισμένες εφαρμογές GPU δεν επωφελούνται από αυτές τις τεχνικές λόγω περιορισμένου παραλληλισμού σε επίπεδο δεδομένων, προκαλώντας συχνούς κύκλους αναμονής και μη βέλτιστη χρήση του υλικού. Για την αντιμετώπιση αυτού του ζητήματος, ο εγγενής παραλληλισμός επιπέδου εντολών αυτών των εφαρμογών μπορεί να αξιοποιηθεί με εκτέλεση εκτός σειράς (Out-of-Order, OOO). Ωστόσο, οι συμβατικές αρχιτεκτονικές GPU εκδίδουν πάντα εντολές εντός σειράς (in-order) για να αποφύγουν κινδύνους δεδομένων, γεγονός που περιορίζει την απόδοσή τους [12, 13] .

Το σχήμα εκτέλεσης Light-Weight Out-of-Order GPU (LOOG) [12, 13, 14] εφαρμόζει αναδιάταξη εντολών για την εκμετάλλευση της παραλληλίας επιπέδου εντολής (ILP). Οι τροποποιήσεις στη μικροαρχιτεκτονική περιλαμβάνουν την επαναχρησιμοποίηση των Operand Collector Units για να χρησιμεύσουν ως σταθμοί κράτησης στον αλγόριθμο Tomasulo [15] , την προσθήκη ενός Register Alias Table για την αντιμετώπιση των εξαρτήσεων ονόματος και την προσθήκη μονάδας αναδιάταξης των εντολών Load και Store για τη διευθέτηση των εξαρτήσεων από τις διευθύνσεις μνήμης.

Για την κλιμάκωση του LOOG και την εκμετάλλευση βαθύτερου παραλληλισμού σε επίπεδο εντολών, το μήκος του παραθύρου εντολών και ο αριθμός των μονάδων συλλογής (Collector Units) μπορούν να αυξηθούν, με το τελευταίο να είναι πολύ πιο σημαντικό σχετικά με την επίδοση. Ωστόσο, η αύξηση των CUs οδηγεί σε σημαντική επιβάρυνση ισχύος, οπότε στην παρούσα εργασία εισάγεται η δυναμική αναδιαμόρφωση του LOOG ώστε να προσαρμόζεται στη συμπεριφορά της εφαρμογής και να μπορεί να μεγιστοποιήσει την επίδοση και την ενεργειακή απόδοση [12, 13, 14] .

Επισκόπηση πρότασης

Παρατηρήσεις όπως αυτές που κινητοποίησαν το LOOG, προκάλεσαν το ενδιαφέρον μας τόσο για τη διερεύνηση συγκεκριμένων χαρακτηριστικών των εφαρμογών που δεν αντιμετωπίζονται επαρκώς από τις τρέχουσες αρχιτεκτονικές, όσο και για το πώς μπορεί να αντιμετωπιστεί βέλτιστα ειδικά η ILP, με τη σωστή διαστασιολόγηση των εφαρμοζόμενων τροποποιήσεων του LOOG σε αυτές τις αρχιτεκτονικές και την προσαρμογή τους στην εκάστοτε εφαρμογή.

Το LOOG υλοποιείται στη νέα έκδοση του GPGPU-sim (4.1.0), παρέχοντας πρόσβαση στον Accel-Sim και τη συνακόλουθη αύξηση της ακρίβειας προσομοίωσης σε πολλά μέτωπα. Η πρόσβαση στη διαμόρφωση (configuration) της GPU του σταθμού εργασίας NVIDIA Quadro GV100, η οποία τροφοδοτείται από την αρχιτεκτονική Volta, που παρέχεται από το Accel-sim και είναι συντονισμένη με μικροδείκτες, μας επιτρέπει να διερευνήσουμε την επιτάχυνση των εφαρμογών υπό LOOG σε ένα υπόστρωμα σχετικό με HPC.

Έχοντας φιλοξενήσει το LOOG σε αυτή την αρχιτεκτονική, συλλέγουμε στατιστικά στοιχεία χρόνου εκτέλεσης σε 100 εφαρμογές (CUDA kernels).
Κατηγοριοποιούμε τις εφαρμογές σε 5 συστάδες όσον αφορά τα αρχιτεκτονικά σημεία συμφόρησής τους και τις συσχετίζουμε με τη βελτίωσή τους στο LOOG. Αναδύεται μια κατηγορία εφαρμογών "ευαίσθητων στο LOOG", που περιλαμβάνει εφαρμογές οι οποίες επιταχύνονται περισσότερο από 100% στις πιο κλιμακούμενες διαμορφώσεις LOOG.

Έχοντας βέλτιστα διαστασιολογήσει τα στοιχεία που δεν σχετίζονται άμεσα με αυτήν (τα πρώτα στάδια της διοχέτευσης της GPU, (Fetch-Decode Bandwidth, Issue scheduling depth), υλοποίηση δυναμικής αναδιαμόρφωσης του ρυθμιστικού διάυλου εντολών) , εισάγουμε και αξιολογούμε μια κλιμακούμενη αναδιαμορφώσιμη αρχιτεκτονική εκτός σειράς με λεπτομερή αποκοπή ισχύος, αναδιαμορφώνοντας σε μια χρονική λεπτομέρεια ανά εκκίνηση CUDA kernel , είτε για τη βελτιστοποίηση των μέτρων επίδοσης είτε για τη βελτιστοποίηση της ενεργειακής αποδοτικότητας.

Η αξιολόγηση της αναδιαμορφώσιμης αρχιτεκτονικής OOO γίνεται αρχικά με βάση μια θεωρητική υλοποίηση λογισμικού που προϋποθέτει προηγουμένως τέλεια γνώση της συμπεριφοράς των εφαρμογών σε επίπεδο CUDA kernel launch και σε όλες τις διαθέσιμες διαμορφώσεις. Έτσι, αξιολογούμε αρχικά τη μέγιστη εφικτή βελτίωση στην επίδοση και την ενεργειακή αποδοτικότητα που παρέχει η δυνατότητα αναδιαμόρφωσης. Οι βέλτιστες διαμορφώσεις προσδιορίζονται αντίστοιχα είτε με βάση την επίδοση είτε με βάση τις ενεργειακές τιμές.

Για CUDA kernels πολλαπλών εκκινήσεων, προχωράμε στη σύγκριση υλοποιήσεων τόσο σε αδρή (στατική) όσο και σε λεπτομερή (ημιδυναμική) αναδιαμόρφωση, καθώς και στην αξιολόγηση ενός ελεγκτή αναδιαμόρφωσης που συμπεραίνει με ακρίβεια την καταλληλότερη διαμόρφωση για την πρώτη εκκίνηση, εκτελώντας σε μια διαμόρφωση που αντιστοιχεί στη βασική αρχιτεκτονική (in-order configuration) και αναδιαμορφώνει την αρχιτεκτονική σύμφωνα με αυτήν για τις υπόλοιπες εκκινήσεις της εφαρμογής.

Τα αποτελέσματα δείχνουν ελάχιστη επιδείνωση και για τις δύο μεταβάσεις (από λεπτομερή αναδιαμόρφωση ανά εκκίνηση σε αδρομερή αναδιαμόρφωση ανά εφαρμογή (CUDA kernel) και εξαγωγή συμπερασμάτων για τη βέλτιστη διαμόρφωση από το σύνολο της εκτέλεσης σε εξαγωγή συμπερασμάτων από την πρώτη εκκίνηση), γεγονός που μας παρακινεί να υλοποιήσουμε και να αξιολογήσουμε έναν ελεγκτή αναδιαμόρφωσης σε επίπεδο υλικού, ο οποίος προβλέπει βελτίωση των επιδόσεων στο LOOG από την πρώτη εκκίνηση μίας εφαρμογής, που εκτελείται με τη διαμόρφωση εντός σειράς (in-order) και αξιολογεί αν δικαιολογείται μια κλιμακωμένη, ενεργοβόρα διαμόρφωση εκτός σειράς για τις επόμενες εκκινήσεις του με βάση την επίδοση ή τις ενεργειακές μετρικές.

Κατά την αναδιαμόρφωση για τη βελτιστοποίηση της ενεργειακής απόδοσης, ο ελεγκτής αναδιαμόρφωσης που βασίζεται στο υλικό παρέχει κατά μέσο όρο 27,4% βελτίωση του χρόνου εκτέλεσης και 19,5% βελτίωση της ενέργειας σε σχέση με το βασικό μοντέλο εντός σειράς, σε όλες τις εφαρμογές που εξετάστηκαν, ενώ ο ελεγκτής αναδιαμόρφωσης που βασίζεται σε λογισμικό παρέχει 29% και 22,4% βελτίωση αντίστοιχα. Σε σύγκριση με μια στατική μικροαρχιτεκτονική LOOG, η ενεργειακή απόδοση βελτιστοποιείται κατά 6,7% και 10,1% για τη μέση εφαρμογή από τους αντίστοιχους ελεγκτές. Για CUDA kernels ευαίσθητους στο LOOG, ο χρόνος εκτέλεσης και η ενεργειακή απόδοση βελτιώνονται κατά 54% και 46%, από τον ελεγκτή λογισμικού.

Τέλος, εικάζουμε την κλιμάκωση του μεγέθους των Μονάδων Εκτέλεσης και της Κρυφής Μνήμης ως άλλους πιθανούς άξονες αναδιαμόρφωσης, ορίζοντας ομοίως εφαρμογές που δεσμεύονται από

τις αντίστοιχες δομές ως προς την επίδοση.

1.2 Θεωρητικό υπόβαθρο

1.2.1 CUDA

Η CUDA είναι μια πλατφόρμα προγραμματισμού που αναπτύχθηκε από την NVIDIA για την αξιοποίηση της ισχύος των GPU για υπολογιστικές εφαρμογές γενικού σκοπού [16, 17, 18] . Η CUDA παρέχει σημαντική επιτάχυνση σε σύγκριση με τους παραδοσιακούς υπολογισμούς που βασίζονται σε CPU και μπορεί να επεκταθεί ώστε να εκτελείται σε πολλαπλές GPU ή σε συστάδες GPU [19, 20] . Η CPU και η GPU συνεργάζονται σε μια ετερογενή εφαρμογή, με την CPU να χειρίζεται εργασίες εντατικές σε έλεγχο και την GPU να χειρίζεται υπολογιστικά εντατικές εργασίες με παραλληλισμό δεδομένων [1]

Ιεραρχία νημάτων

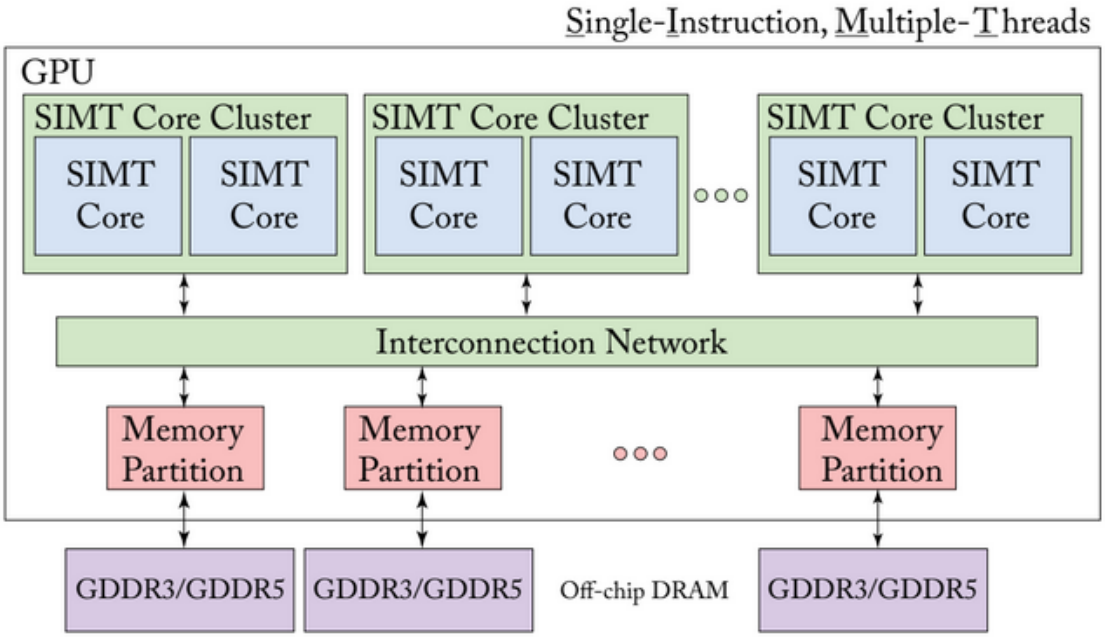
Η CUDA επεκτείνει τις γλώσσες στις οποίες υλοποιείται, επιτρέποντας στον προγραμματιστή να ορίζει συναρτήσεις στην αντίστοιχη γλώσσα, που ονομάζονται "CUDA kernels" (CUDA kernels) , οι οποίες μεταφέρονται στη GPU σε αντίθεση με τις κανονικές συναρτήσεις της γλώσσας. Οι CUDA kernels περιλαμβάνουν δεκάδες χιλιάδες νήματα, επιτρέποντας την επιτάχυνση μαζικά παράλληλων εφαρμογών. Τα νήματα αναγνωρίζονται με ένα πολυδιάστατο (1 έως 3) αναγνωριστικό νήματος, σχηματίζοντας ένα ¨μπλοκ νήματος¨. Υπάρχει όριο στον μέγιστο αριθμό νημάτων σε ένα μπλοκ, καθώς όλα βρίσκονται στον ίδιο πολυεπεξεργαστή ροής (SM) (επί του παρόντος 1024 νήματα), μοιράζοντας τους ίδιους πόρους μνήμης. Τα πολλαπλά μπλοκ είναι ομοίως οργανωμένα σε πολυδιάστατα πλέγματα. Έτσι, ένας CUDA kernel εκτελείται ως ένα πλέγμα από μπλοκ νημάτων.[21] Η βασική μονάδα εκτέλεσης σε μια GPU είναι ένα warp, μια ομάδα νημάτων (32 στις τρέχουσες υλοποιήσεις), που καταλαμβάνει αποκλειστικά ένα συγκεκριμένο στάδιο αγωγού ΣΜ σε κάθε δεδομένη στιγμή [22, 23].

Ιεραρχία μνήμης

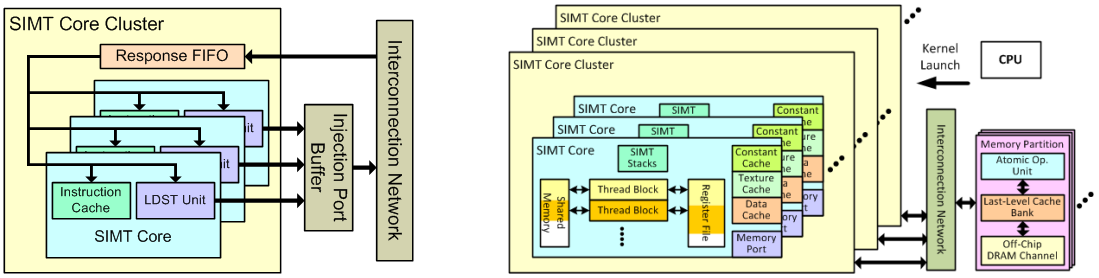
Τα νήματα CUDA έχουν πρόσβαση σε διάφορα επίπεδα της ιεραρχίας μνήμης κατά τη διάρκεια της εκτέλεσής τους. Η εν λόγω ιεραρχία αποτελείται από:

- Καταχωρητές ανά νήμα και τοπική μνήμη (χρησιμοποιείται κυρίως για τη διαρροή καταχωρητών).
- Παγκόσμια μνήμη ορατή από όλα τα νήματα ενός μπλοκ και με κοινή διάρκεια ζωής.
- Κοινή μνήμη που χρησιμοποιείται από όλα τα νήματα ενός μπλοκ ή μιας ομάδας μπλοκ.
- Σταθερή μνήμη, υποστηριζόμενη από την Constant Cache.
- Μνήμη υφής, υποστηριζόμενη από την Texture Cache, που παρέχει διαφορετικούς τρόπους διευθυνσιοδότησης που εξυπηρετούν 2d χωρική τοπικότητα.

[21, 24]



Σχήμα 1.1: Ένα μοντέλο των υψηλού επιπέδου δομών μίας σύγχρονης αρχιτεκτονικής GPU [1]



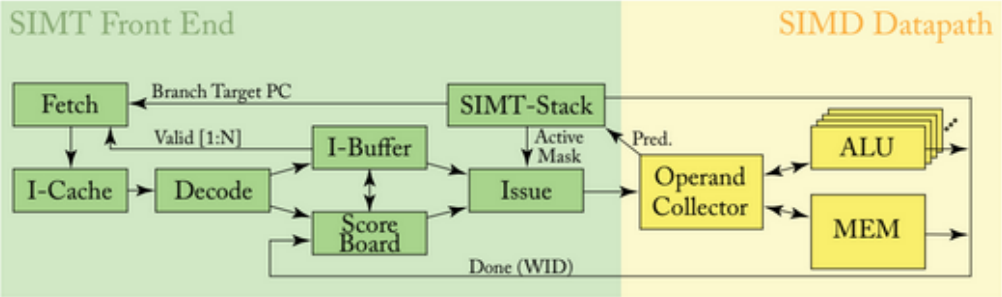
(α′) Αρχιτεκτονική GPU σε επίπεδο συστάδας SM [24] (β′) Η εσωτερική κατανομή και οργάνωση του SM [23]

Σχήμα 1.2: Μοντέλο της GPU από τον GPGPU-Sim [24]

Αρχιτεκτονική GPU

Μια σύγχρονη GPU διαθέτει πολλούς πυρήνες που ονομάζονται πολυεπεξεργαστές ροής ή υπολογιστικές μονάδες (SM) , όπως φαίνεται στην εικόνα 1.2α′. Κάθε SM είναι ένας επεξεργαστής SIMD που μπορεί να εκτελεί έως και χίλια νήματα ταυτόχρονα. Τα νήματα σε ένα SM μπορούν να επικοινωνούν μέσω κοινής μνήμης και συγχρονίζονται με γρήγορες λειτουργίες φραγμού.

Οι GPU πρέπει συχνά να έχουν πρόσβαση σε μεγάλα σύνολα δεδομένων που δεν μπορούν να αποθηκευτούν εξ ολοκλήρου στο τσιπ, οπότε απαιτούνται εξειδικευμένες κρυφές μνήμες και πρόσβαση στη μνήμη εκτός τσιπ με υψηλό εύρος ζώνης. Οι SM και τα διαμερίσματα μνήμης συνδέονται μέσω ενός δικτύου διασύνδεσης εντός του chip και η κίνηση μνήμης κατανέμεται στις μονάδες διαμερισμάτων μνήμης με τη χρήση διαχωρισμού διευθύνσεων. Τα SM είναι οργανωμένα σε ομάδες, καθεμία με μια FIFO ουρά απόκρισης που μπορεί να συγκρατήσει πακέτα που προέρχονται από το δίκτυο διασύνδεσης.



Σχήμα 1.3: Στάδια διοχέτευσης GPU [1]

Αρχιτεκτονική Κρυφής Μνήμης

Όπως απεικονίζεται στο Σχήμα 1.2β′, οι δομές μνήμης μέσα σε κάθε SM είτε χρησιμοποιούνται από κοινού από τα μπλοκ που το καταλαμβάνουν είτε είναι κατανεμημένες μεταξύ τους. Η μοιραζόμενη μνήμη και το Register File είναι κατανεμημένα μεταξύ των μπλοκ, και το Register File και τα SIMT Stacks συγκεκριμένα είναι ευρετηριασμένα με βάση το warp ID. Τα είδη κρυφής μνήμης περιλαμβάνουν:

1. Σταθερή κρυφή μνήμη: Μια κρυφή μνήμη μόνο για ανάγνωση που αποθηκεύει σταθερές στις οποίες γίνεται συχνή πρόσβαση, όπως πίνακες ή κλιμακωτές τιμές.

2. Κρυφή μνήμη υφής: Μια εξειδικευμένη κρυφή μνήμη που έχει βελτιστοποιηθεί για λειτουργίες χαρτογράφησης υφής.

3. Κρυφή μνήμη δεδομένων: Μια κρυφή μνήμη γενικής χρήσης που αποθηκεύει τμήματα της τοπικής και της παγκόσμιας μνήμης στα οποία γίνεται συχνή πρόσβαση.

4. Μοιραζόμενη μνήμη: Ένας χώρος μνήμης scratchpad υψηλού εύρους ζώνης και χαμηλής καθυστέρησης που μοιράζονται τα νήματα μέσα σε ένα μπλοκ.

5. Κρυφή μνήμη δεδομένων επιπέδου 2: Το πρώτο στάδιο του διαμερίσματος μνήμης εκτός τσιπ. Βελτιστοποιεί την απόδοση ανά μονάδα επιφάνειας για την GPU και διαχειρίζεται ταυτόχρονες εγγραφές από πολλαπλά νήματα σε διακλαδώσεις.

1.2.2 Στάδια διοχέτευσης των SM

Για να διατηρηθούν υψηλές επιδόσεις στην GPU, είναι απαραίτητο να εξισορροπηθεί το υψηλό εύρος ζώνης μνήμης με την υψηλή υπολογιστική επίδοση. Το μοντέλο του αγωγού της GPU παρέχει περαιτέρω πληροφορίες σχετικά με τους μηχανισμούς που σχετίζονται με αυτό το [1, 24, 23]. Το σχήμα 1.3 παρέχει μια οπτική αναπαράσταση της εσωτερικής αρχιτεκτονικής του αγωγού του Streaming Multiprocessor. Ο αγωγός αποτελείται από ένα SIMT front-end και ένα SIMD back-end. Παρόμοια με μια CPU που υλοποιεί πολυνηματικότητα, το SIMT front-end επιτρέπει την ταυτόχρονη λήψη, αποκωδικοποίηση και έκδοση (Fetch, Decode, Issue) των warps.

Ο χρονοπρογραμματισμός του αγωγού πραγματοποιείται σε τρεις συνεχείς "βρόχους": τον βρόχο ανάκτησης εντολών, τον βρόχο έκδοσης εντολών και τον βρόχο προγραμματισμού προσπέλασης καταχωρητών. Ο βρόχος ανάκτησης εντολών περιλαμβάνει τα μπλοκ Fetch, I-Cache, Decode και

I-Buffer που φαίνονται στο Σχήμα 1.3 . Ο βρόχος έκδοσης εντολών περιλαμβάνει τα μπλοκ I-Buffer, Scoreboard, Issue και SIMT Stack. Ο βρόχος προγραμματισμού πρόσβασης σε καταχωρητές περιλαμβάνει τα μπλοκ Operand Collector, ALU και Memory. Το συνολικό μοντέλο της διοχέτευσης του SM περιλαμβάνει τα ακόλουθα στάδια:

Fetch Ένα warp επιλέγεται για χρονοπρογραμματισμό και ο μετρητής προγράμματος προσπελαύνει την κρυφή μνήμη εντολών για να αντλήσει την επόμενη εντολή.

Decode Η εντολή αποκωδικοποιείται και τοποθετείται σε ένα buffer εντολών μέχρι να διαπιστωθεί ότι δεν υπάρχουν κίνδυνοι.

Οι τιμές της μάσκας εκτέλεσης SIMT για τη στοίβα SIMT καθορίζονται παράλληλα με την άντληση των αρχικών τελεστών από το αρχείο καταχωρητών.

Issue Ο χρονοπρογραμματιστής έκδοσης αποφασίζει ποιες εντολές θα εκδοθούν στον αγωγό και ποια warps θα έχουν προτεραιότητα. Η στοίβα SIMT ενημερώνεται και η απόκλιση των warp αντιμετωπίζεται μέσω της σειριοποίησης της εκτέλεσης νημάτων εντός ενός warp. Ένας πίνακας αποτελεσμάτων χρησιμοποιείται για την αποτροπή κινδύνων και την εκμετάλλευση κενών θέσεων στο backend.

Operand Collect Οι εντολές τοποθετούνται σε Collector Units στη δομή Operand Collector για να αποκρύψουν τις καθυστερήσεις μνήμης. Ο Operand Collector προσομοιώνει ένα αρχείο καταχωρητών πολλαπλών θυρίδων μέσω παραλληλισμού σε επίπεδο τράπεζας.

Dispatch Οι εντολές τοποθετούνται σε μια δεξαμενή που επιλέγεται από τον χρονοπρογραμματιστή αποστολής για εκτέλεση στις λωρίδες της κατάλληλης μονάδας εκτέλεσης.

Execute Η εκτέλεση των εντολών γίνεται στις αντίστοιχες μονάδες. Οι GPU της NVIDIA διαθέτουν ετερογενείς μονάδες εκτέλεσης, όπως μονάδες φόρτωσης/αποθήκευσης, λειτουργικές μονάδες ακέραιων αριθμών, λειτουργικές μονάδες κινητής υποδιαστολής, μονάδες ειδικών λειτουργιών και μονάδες Tensor Core.

Writeback Οι εντολές εκδίδουν ένα Register File write για τους τελεστές προορισμού τους και ενημερώνουν τον πίνακα αποτελεσμάτων, απελευθερώνοντας τυχόν εξαρτημένες εντολές.

1.2.3 Αναδιαμορφώσιμες αρχιτεκτονικές

Δεδομένης της αποτελμάτωσης των νόμων κλιμάκωσης, έχουν χρησιμοποιηθεί διάφορες προσεγγίσεις σχεδιασμού τσιπ και αρχιτεκτονικής στην προσπάθεια να ικανοποιηθούν οι απαιτήσεις επιδόσεων των σύγχρονων εξειδικευμένων εφαρμογών.

Αρχιτεκτονικές συγκεκριμένων εφαρμογών και τομέων ASIC, DSA

Οι αρχιτεκτονικές ASIC και οι αρχιτεκτονικές ειδικών τομέων σε επίπεδο μονάδων εκτέλεσης (DSA) είναι δύο παραδείγματα αυτής της τάσης, όπου οι ASIC σχεδιάζονται για συγκεκριμένες περιπτώσεις χρήσης με τη χρήση γλωσσών περιγραφής υλικού, ενώ οι DSA προσαρμόζουν τις μονάδες εκτέλεσης για συγκεκριμένα πλαίσια λογισμικού [25, 26, 27] .

Υπολογιστικά συστήματα λεπτομερούς αναδιαμόρφωσης

Οι αναδιαμορφώσιμες αρχιτεκτονικές αναδύονται ως εναλλακτική λύση έναντι των ASIC για την ελαχιστοποίηση του μη επαναλαμβανόμενου κόστους σχεδιασμού (NRE) για τη σχεδίαση προσαρμοσμένου υλικού. Οι συστοιχίες προγραμματιζόμενων πυλών πεδίου (FPGAs) παρέχουν ευελιξία και απόδοση υλικού με τη δυνατότητα επαναπρογραμματισμού και πρωτοτυποποίησης διαφορετικών σχεδίων. Ωστόσο, συνοδεύονται επίσης από υψηλό κόστος σχετικά με την ισχύ, τη χρήση της περιοχής και την επιβάρυνση αναδιαμόρφωσης [28, 29, 30, 4, 31] .

Υπολογιστικά συστήματα αδρομερούς αναδιαμόρφωσης

Οι επαναδιαμορφώσιμες αρχιτεκτονικές αδρομερούς αναδιαμόρφωσης (CGRAs) ενσωματώνουν προγραμματιζόμενους λογικούς πόρους με εξειδικευμένες λειτουργικές μονάδες, παρέχοντας μια ισορροπία μεταξύ ευελιξίας και απόδοσης. Σε σχέση με τα FPGA, προσφέρουν υψηλότερες επιδόσεις ανά μονάδα, μειωμένη επιβάρυνση επαναδιαμόρφωσης, καλύτερη αξιοποίηση του χώρου και χαμηλότερη κατανάλωση ενέργειας [32, 30] .

Μαλακοί πυρήνες (Soft Cores)

Οι μαλακοί πυρήνες επιτρέπουν την προσαρμογή των συνόλων εντολών για την αντιμετώπιση συγκεκριμένων εφαρμογών, ελαχιστοποιώντας την ανάγκη για ASICs ή προσαρμοσμένα σχέδια υλικού. Ο σχεδιασμός των soft cores μπορεί να γίνει με τη χρήση γλωσσών υψηλού επιπέδου, όπως η C, γεγονός που καθιστά ευκολότερη την ανάπτυξη και τη δοκιμή τους [33, 34] .

1.2.4 Ετερογενείς αρχιτεκτονικές

Τα ετερογενή υπολογιστικά συστήματα αναφέρονται σε συστήματα που χρησιμοποιούν πολλαπλές μονάδες επεξεργασίας με διαφορετικές αρχιτεκτονικές, δυνατότητες και λειτουργίες, οι οποίες λειτουργούν στην ίδια ροή εργασίας για την εκτέλεση ενός ή περισσότερων υπολογισμών και αναθέτουν κάθε έναν από αυτούς στο στοιχείο επεξεργασίας που του ταιριάζει καλύτερα [35].

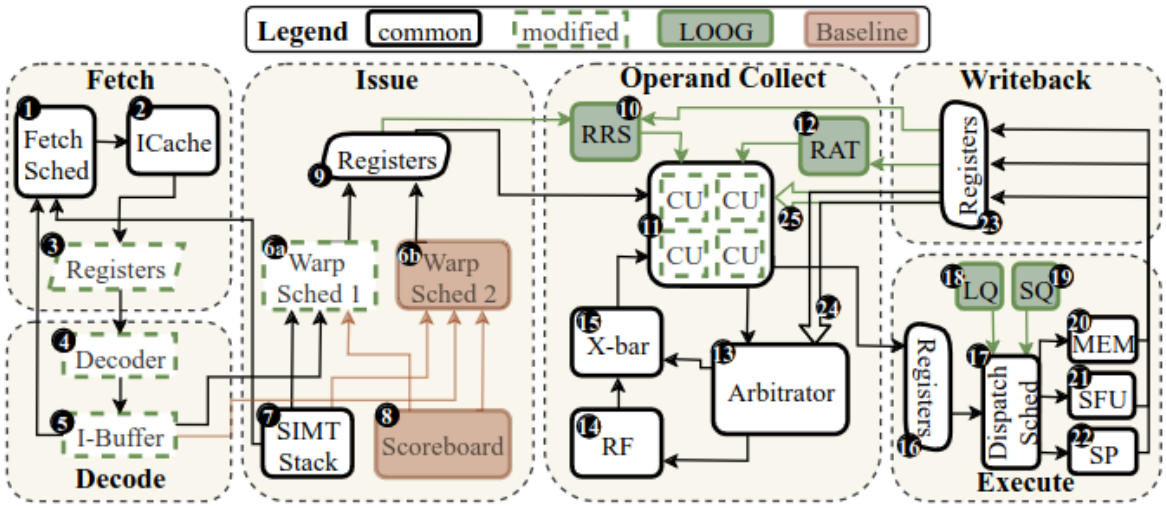
Παραδείγματα αποτελούν:

Υβριδικό σύστημα CPU-GPU Τα δεδομένα ελέγχου του προγράμματος σε τέτοια συστήματα μπορούν να υποβληθούν σε επεξεργασία από την CPU, ενώ οι πράξεις κινητής υποδιαστολής μπορούν να μεταφερθούν στην GPU.

Συστάδα ετερογενών επεξεργαστών Μια συστάδα που αποτελείται από πολλούς επεξεργαστές με διαφορετικές αρχιτεκτονικές, όπως CPU, GPU, FPGA και DSP.

Σύστημα στοιχείων υλικού και λογισμικού Ένα σύστημα που περιλαμβάνει τόσο στοιχεία υλικού όσο και στοιχεία λογισμικού, όπως πλατφόρμες καθορισμένες από λογισμικό (SDR) που περιλαμβάνουν FPGA και DSP.

Ασύμμετροι πολυεπεξεργαστές τσιπ (ACMP) Ασύμμετροι πολυεπεξεργαστές τσιπ (ACMP): Πολλαπλοί διαφορετικοί επεξεργαστές που βρίσκονται στο ίδιο chip, οι οποίοι συνήθως διαφέρουν ως προς τον υπερκλιμάκωση και το μήκος του παραθύρου εντολών.



Σχήμα 1.4: Τροποποιήσεις του LOOG στη βασική μικροαρχιτεκτονική [14]

1.2.5 Accel-Sim

Η υλοποίηση του LOOG και άλλων τροποποιήσεων της μικροαρχιτεκτονικής που έγιναν στην έκδοση 4.1.0 του GPGPU-sim παρεχουν πρόσβαση σε όλες τις δυνατότητες του Accelsim [36] . Ο Accel-sim εισάγει ένα ευέλικτο frontend που παρέχει τη δυνατότητα αξιοποίησης της ISA μηχανής (mISA) σε λειτουργία με βάση ίχνη (traces). Για προσομοίωση νέων καρτών, ένα αυτοματοποιημένο πλαίσιο ρύθμισης τροποποιεί το αρχείο διαμόρφωσης της αρχιτεκτονικής που προσομοιώνεται, παράγοντας ένα ακριβές μοντέλο επιδόσεων.

Ο Accel-sim παρέχει ακριβέστερη μοντελοποίηση και διαμόρφωση επιδόσεων των δοκιμασμένων GPU, συμπεριλαμβανομένης της NVIDIA Quadro GV100.

Στο πλάισιο της εργασίας προστίθενται νέα λεπτομερή στατιστικά στοιχεία, σχετικά με το LOOG (η συνολική ετοιμότητα ανά warp, η συνολική κατανομή ετοιμότητας warp, η παρακολούθηση της απόδοσης αποκωδικοποιητή, η κατανομή πληρότητας RRS, οι καταχωρήσεις RAT συνολικά ανά warp και οι καταχωρήσεις RAT που χρησιμοποιούνται ανά warp για την παρακολούθηση του ILP.

1.2.6 Οι δομές του LOOG και οι τροποποιήσεις στον GPGPU-Sim

Το Light-Weight Out-of-Order GPU execution scheme (LOOG) [12, 13, 14] περιλαμβάνει την επαναχρησιμοποίηση τυπικών μικροαρχιτεκτονικών GPU για την εκμετάλλευση της ILP και την καλύτερη διαχείριση ενός συνόλου εφαρμογών-στόχων που χαρακτηρίζονται από χαμηλή επιτυγχανόμενη αξιοποίηση και συνακόλουθο χαμηλό IPC (εντολές ανα κύκλο). Οι αλλαγές και οι προσθήκες στα στοιχεία της τυπικής μικροαρχιτεκτονικής GPU για την παραγωγή του LOOG φαίνονται στο Σχήμα 1.4 και είναι οι ακόλουθες:

- Οι Μονάδες Συλλογής (CUs) τροποποιούνται ώστε να χρησιμεύουν ως σταθμοί κράτησης του αλγορίθμου Tomasulo, ο οποίος επιτρέπει την άμεση έκδοση οδηγιών χωρίς να περάσει πρώτα από έλεγχο του πίνακα αποτελεσμάτων.

- Προστίθεται ένας πίνακας Register Alias Table (RAT) για την εξάλειψη των εξαρτήσεων ψευδών ονομάτων. Όταν μια εντολή κατανέμει μια CU/RS, διαβάζει τον RAT ανά warp για κάθε αρχικό τελεστή.
- Το παράθυρο εντολών τροποποιείται ώστε να εξυπηρετεί την εκμετάλλευση του παραλληλισμού επιπέδου εντολών (ILP) με την ανάκτηση 16 έως 128 bytes δεδομένων από την ICache σε κάθε ανάκληση και οι εντολές εκδίδονται απευθείας στις CUs για αναδιάταξη.
- Το LOOG επωφελείται από τον χρονοπρογραμματισμό των warp σε βάθος, ο οποίος αποδεικνύεται ότι βελτιώνει σημαντικά την απόδοση.
- Η αναδιάταξη φόρτωσης-αποθήκευσης στο LOOG περιλαμβάνει τη δημιουργία μιας ουράς φόρτωσης και μιας ουράς αποθήκευσης και την εκχώρηση μιας εγγραφής για τις αντίστοιχες εντολές μνήμης για την αντιμετώπιση των εξαρτήσεων δεδομένων μνήμης πέρα από τις εξαρτήσεις καταχωρητών.
- Ο δίαυλος μετάδοσης αποτελεσμάτων μεταδίδει τα αποτελέσματα μετά την ολοκλήρωση της εκτέλεσης και στέλνει ένα αίτημα εγγραφής RF στον διαιτητή. Οι CUs απελευθερώνονται μόνο μετά την εγγραφή της αντίστοιχης εντολής.
- Εισάγεται η Στοίβα Μετονομασίας Καταχωρητών (Register Renaming Stack - RRS) για την αποθήκευση ενός καταλόγου μοναδικών αναγνωριστικών που θα χρησιμοποιηθούν στο RAT αντί του αναγνωριστικού CU, γεγονός που μειώνει σημαντικά τη συμφόρηση CU και αυξάνει την εκμετάλλευση του ILP.

1.2.7 Ανασκόπηση ετερογενών και αναδιαμορφώσιμων αρχιτεκτονικών

Αναδιαμορφώσιμοι πολυεπεξεργαστές τσιπ (reconfigurable CMPs)

Οι αναδιαμορφώσιμες αρχιτεκτονικές CMP [37, 38] προσφέρουν μια εναλλακτική λύση στις αρχιτεκτονικές ασύμμετρων πολυεπεξεργαστών τσιπ για τον χειρισμό της ποικιλομορφίας του φόρτου εργασίας. Η συγχώνευση πυρήνων (Core Fusion) [37] είναι μια επαναδιαμορφώσιμη αρχιτεκτονική CMP που χρησιμοποιεί ISA RISC ή CISC και πρόσθετα στοιχεία για την αποτελεσματική εκτέλεση λειτουργιών FUSE και SPLIT μεταξύ υποσυνόλων πυρήνων κατά τη διάρκεια του χρόνου εκτέλεσης.

Κλιμακούμενοι πυρήνες

Η αρχιτεκτονική Elastic Core [10] κλιμακώνει δυναμικά τους πόρους, την τάση λειτουργίας και τη συχνότητα ώστε να προσαρμόζονται στη συμπεριφορά της εφαρμογής, χρησιμοποιώντας ένα μοντέλο γραμμικής παλινδρόμησης για την πρόβλεψη ισχύος και απόδοσης. Χρησιμοποιεί αποκοπή ρολογιού (clock gating) σε επίπεδο μονάδας για τον περιορισμό της δυναμικής διάχυσης ισχύος σε ανενεργά στοιχεία.

Ετερογενείς αρχιτεκτονικές

Οι ετερογενείς αρχιτεκτονικές σε αυτό το πλαίσιο αναφέρονται στους ασύμμετρους πολυεπεξεργαστές τσιπ (ACMP), επίσης γνωστούς ως ετερογενή συστήματα πολυεπεξεργαστών σε ένα τσιπ (MPSoC). Το Big.Little [39] είναι μια αρχιτεκτονική που συνδυάζει πυρήνες υψηλής απόδοσης και πυρήνες χαμηλής κατανάλωσης ενέργειας για βελτιστοποιημένη απόδοση και ενεργειακή αποδοτικότητα. Η ετερογενής αρχιτεκτονική μπλοκ (HBA) [40] χωρίζει τον κώδικα σε ατομικά μπλοκ που εκτελούνται ανεξάρτητα σε διαφορετικές μικροαρχιτεκτονικές, επιτρέποντας καλύτερο χειρισμό φάσεων μνήμης και έντασης υπολογισμού. Το Dynamic Core Boosting (DCB) [9] μετριάζει την ανισορροπία του φόρτου εργασίας aCMPs, ενισχύοντας τα κρίσιμα νήματα μέσω μεμονωμένου DVFS σε επίπεδο πυρήνα.

"Μαλακοί πυρήνες " (Soft cores)

Το TRIPS [41, 42] είναι ένα παράδειγμα αναδιαμορφώσιμου υπολογισμού που χρησιμοποιεί ένα Explicit Data Graph Execution (EDGE) ISA για την αποτελεσματική εκτέλεση φορτίων εργασίας έντασης δεδομένων, λειτουργώντας απευθείας στον γράφο ροής δεδομένων. Η κύρια καινοτομία του έγκειται στη χρήση ζωνών προστασίας για την ατομική εκτέλεση ομάδων εντολών. Η απαιτούμενη υποστήριξη μεταγλωττιστή είναι το σημαντικότερο μειονέκτημά της.

Αναδιαμορφώσιμες αρχιτεκτονικές GPU

Η αρχιτεκτονική Bahurupi [43] είναι ένα πολυμορφικό ομοιογενές πολυπύρηνο σύστημα που μπορεί να μετατραπεί σε ετερογενή πολυπύρηνη αρχιτεκτονική κατά την εκτέλεση με οδηγίες λογισμικού, επιτρέποντάς του να εκμεταλλεύεται τόσο τον παραλληλισμό με νήματα όσο και τον παραλληλισμό με εντολές. Υλοποιεί τη συγχώνευση πυρήνων (Core Fusion) σε GPU.

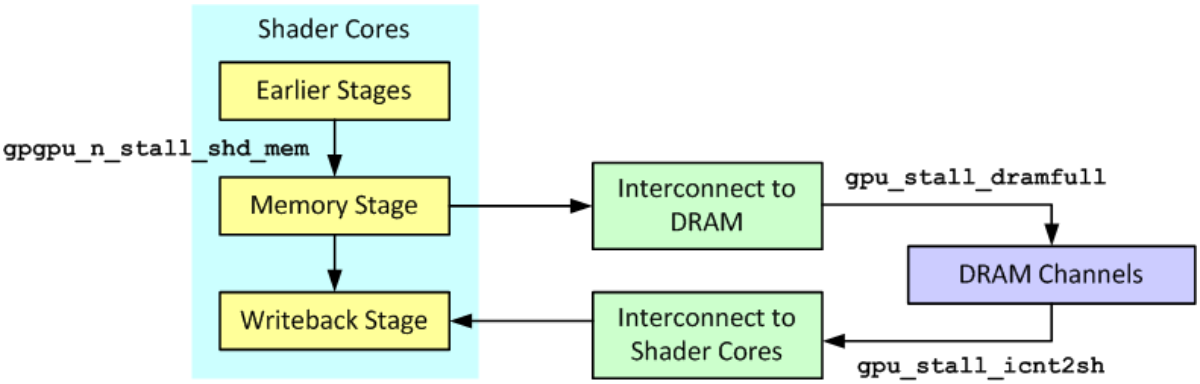
Το Equalizer [44] είναι ένα σύστημα διαχείρισης υλικού που έχει σχεδιαστεί για να παρακολουθεί δυναμικά τις απαιτήσεις πόρων ενός πυρήνα με σκοπό τη βελτιστοποίηση της απόδοσης και της ενεργειακής αποδοτικότητας. Προσαρμόζει το υλικό ώστε να ταιριάζει με τις ανάγκες του εκτελούμενου CUDA kernel, διαχειριζόμενο την ταυτόχρονη χρήση στο chip, τη συχνότητα των πυρήνων και τη συχνότητα της μνήμης.

Το Amoeba [45] παρακολουθεί δυναμικά την επεκτασιμότητα των εφαρμογών και προσαρμόζει ανάλογα τη διαμόρφωση των πολυεπεξεργαστών ροής (SM), συγχωνεύοντάς τα σε αδρομερή βάση ως προς τη χρήση του Network-on-Chip, τη συνένωση υπο-warp, τη μνήμη, την απόκλιση ελέγχου και τη διεκδίκηση της κρυφής μνήμης L1.

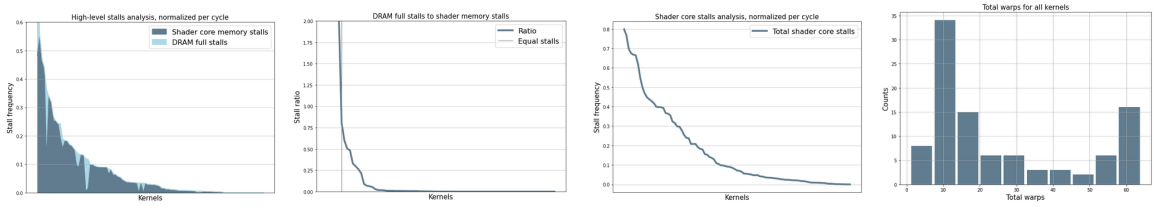
1.3 Λεπτομέρειες υλοποίησης

1.3.1 Ανάλυση κύκλων αναμονής

Η ανάλυση των κύκλων αναμονής σε αυτό το κεφάλαιο, μας δίνει μια εικόνα για τα χαρακτηριστικά των εφαρμογών που μπορούν να αξιοποιηθούν από τη σκοπιά της αναδιαμόρφωσης της αρχιτεκτονικής (όσον αφορά τα αρχιτεκτονικά σημεία συμφόρησης) όσο και για εκείνα που συσχετίζονται



Σχήμα 1.5: Ροή αιτημάτων μνήμης στον GPGPU-Sim [24]



(α′) Σημεία συμφόρησης υψηλού επιπέδου (β′) Αναλογία αναστολών DRAM, μνήμης SM (γ′) Αναστολές SM (δ′) Ιστόγραμμα αριθμού warps

Σχήμα 1.6:

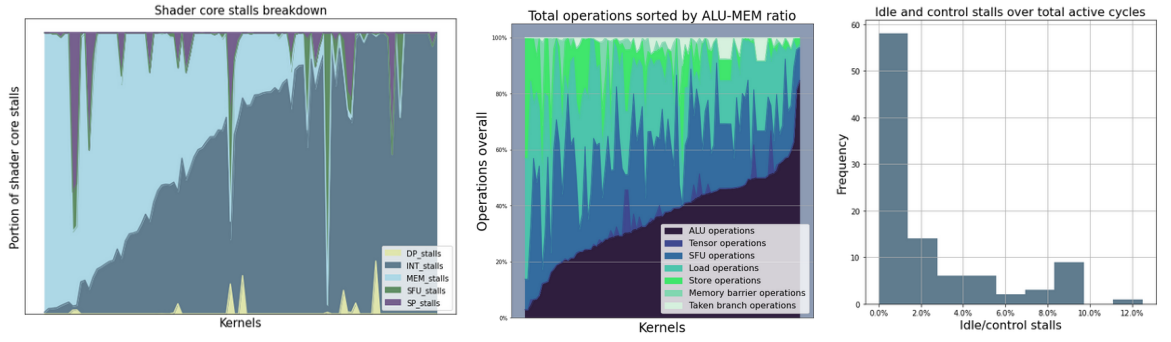
σημαντικά με τη βελτίωση υπό LOOG. Όπως φαίνεται στο Σχήμα 1.5, καταγράφονται μετρικές αναστολής μνήμης.

Στο σχήμα 1.6α′, φαίνεται η κατανομή των αναστολών μνήμης σε υψηλό επίπεδο. Επιπλέον, στο Σχήμα 1.6β′ φαίνεται ότι μόνο το 12% των πυρήνων αναστέλλεται περισσότερο στην DRAM από ό,τι στη διοχέτευση του SM.

Όπως απεικονίζεται στο Σχήμα 1.6γ′, οι αναστολές στο SM ακολουθούν την αναμενόμενη παρόμοια κατανομή με τις αναστολές υψηλού επιπέδου. Στο Σχήμα 1.6δ′, φαίνεται ότι για τη συντριπτική πλειονότητα των εφαρμογών GPGPU που προσομοιώθηκαν, το μέγιστο ποσό των διαθέσιμων warp slots είναι υποαπασχολημένο, γεγονός που εμποδίζει τη GPU να κρύψει τις εν λόγω αναστολές με εναλλαγή περιβάλλοντος μεταξύ των ενεργών warps.

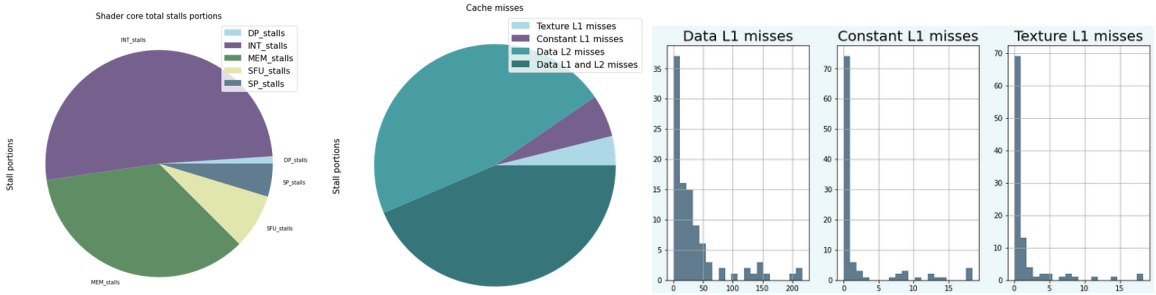
Οι αναστολές αδράνειας ή ελέγχου κατά τη διάρκεια ενεργών κύκλων έκδοσης αντιπροσωπεύουν διάμεσο 0,91% και μέσο όρο 2,3% επί του συνόλου των ενεργών κύκλων, όπως φαίνεται στο Σχήμα 1.7γ′ Όλες οι αναστολές του προβλήματος προκαλούνται ουσιαστικά από αναστολές λειτουργικών μονάδων, η κατανομή των οποίων απεικονίζεται στο Σχήμα 1.7α′. Μία σημαντική κατηγορία πυρήνων (περίπου 25%) που δεν παρουσιάζουν σχεδόν καθόλου αναστολές μνήμης (compute intensive) στον αγωγό. Ωστόσο, όπως φαίνεται στο Σχήμα 1.7β′, ακόμη και αυτοί οι CUDA kernels έχουν σημαντικό αριθμό λειτουργιών μνήμης συνολικά, επομένως όλα τα αδιέξοδα μνήμης τους συμβαίνουν στην DRAM.

Μια ανάλυση των ακινητοποιήσεων της κρυφής μνήμης του SM που προκαλούν ακινητοποιήσεις μνήμης φαίνεται στο Σχήμα 1.8β′. Εμφανώς, οι μισές από τις εντολές που αστοχούν στην κρυφή μνήμη δεδομένων L1 αστοχούν και στην εκτός chip, ανά τμήμα μνήμης L2. Δεδομένου ότι οι καθυστερήσεις που προκαλούνται από αυτές τις αστοχίες δεν μπορούν να αντιμετωπιστούν με



(α′) Αναστολές SM backend (β′) Δυναμικές εντολές (γ′) Αναστολές ελέγχου και αδράνειας

Σχήμα 1.7:



(α′) Αναστολές λειτουργικών μονάδων (β′) Αστοχίες κρυφής μνήμης (γ′) Κατανομή αστοχιών κρυφής μνήμης

Σχήμα 1.8:

αναδιάταξη των εντολών και αλλαγές στον χρονοπρογραμματισμό, αυτό αποτελεί κίνητρο για την ιδέα της δυνητικής αύξησης της απόδοσης της κρυφής μνήμης και της μελέτης του συμβιβασμού μεγέθους κρυφής μνήμης / Area-Power overhead, στο κεφάλαιο 5.9 .

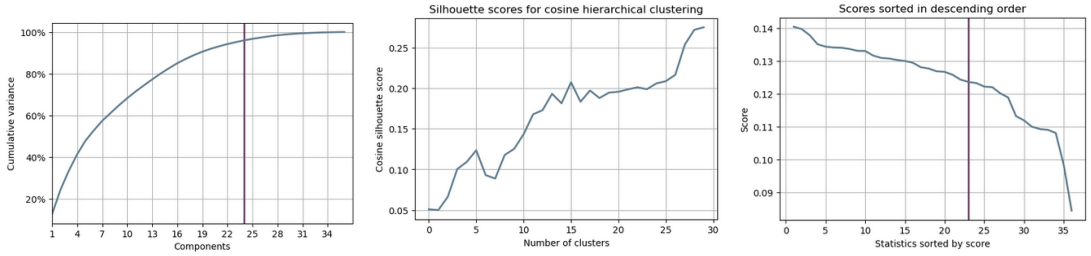
1.3.2 Χαρακτηρισμός των εφαρμογών και συσχετίσεις με την ILP

Ο σκοπός αυτής της μελέτης είναι τόσο ο προσδιορισμός αναδυόμενων κατηγοριών εφαρμογών GPGPU βάσει αρχιτεκτονικών σημείων συμφόρησης που θα υπαγορεύουν αντίστοιχες κατηγορίες διαμόρφωσης υλικού, όσο και η συσχέτιση των εν λόγω κατηγοριών με τη βελτίωση της απόδοσης σε σχήματα εκτέλεσης OOO, Long Instruction Window, όπως το LOOG [12, 13]. Συλλέξαμε στατιστικά στοιχεία σε πολλαπλές διαμορφώσεις, in-order , για τα τρία επίπεδα της ανάλυσης όπως και στην "Whole Picture Analysis" [46, 47]. Τα πιο σημαντικά χαρακτηριστικά που παράγουν ποικιλομορφία φόρτου εργασίας παρουσιάζονται στο Σχήμα 1.10.

$$importance(feature) = \sum_{x \in S} \lambda(x) \cdot coeff(x, feature) \tag{1.1}$$

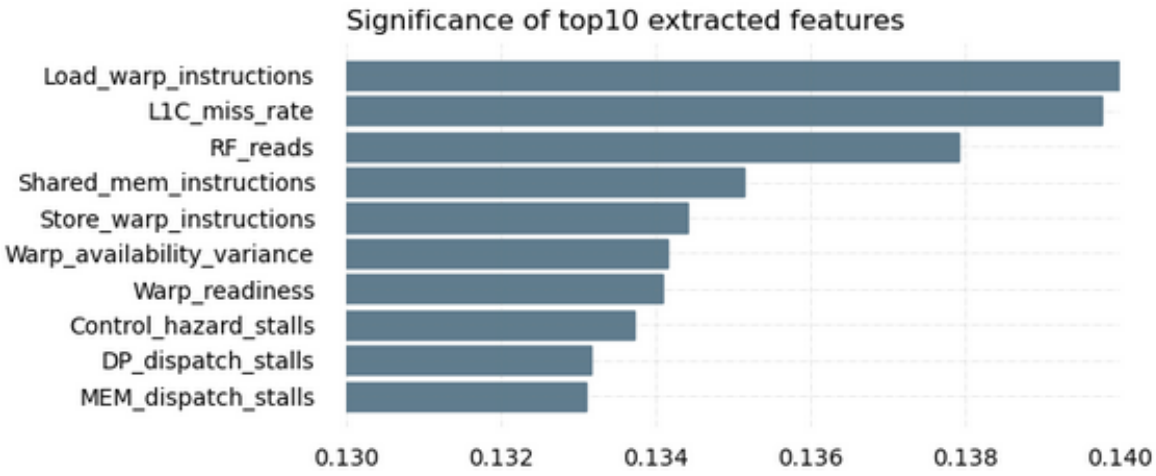
S: *Το σύνολο όλων των ιδιοδιανυσμάτων*
$\lambda(x)$: *Η ιδιοτιμή για το ιδιοδιάνυσμα x*
Οι βαθμολογίες σιλουέττας μεγιστοποιούνται στην τιμή 5 και επιλέξαμε τόσες συστάδες, όπως

φαίνεται στο Σχήμα 1.12 . Οι συστάδες που δημιουργήθηκαν, καθώς και οι κορεσμένες βελτιώσεις LOOG IPC και οι χρήσεις GPU απεικονίζονται στο Σχήμα 1.11.

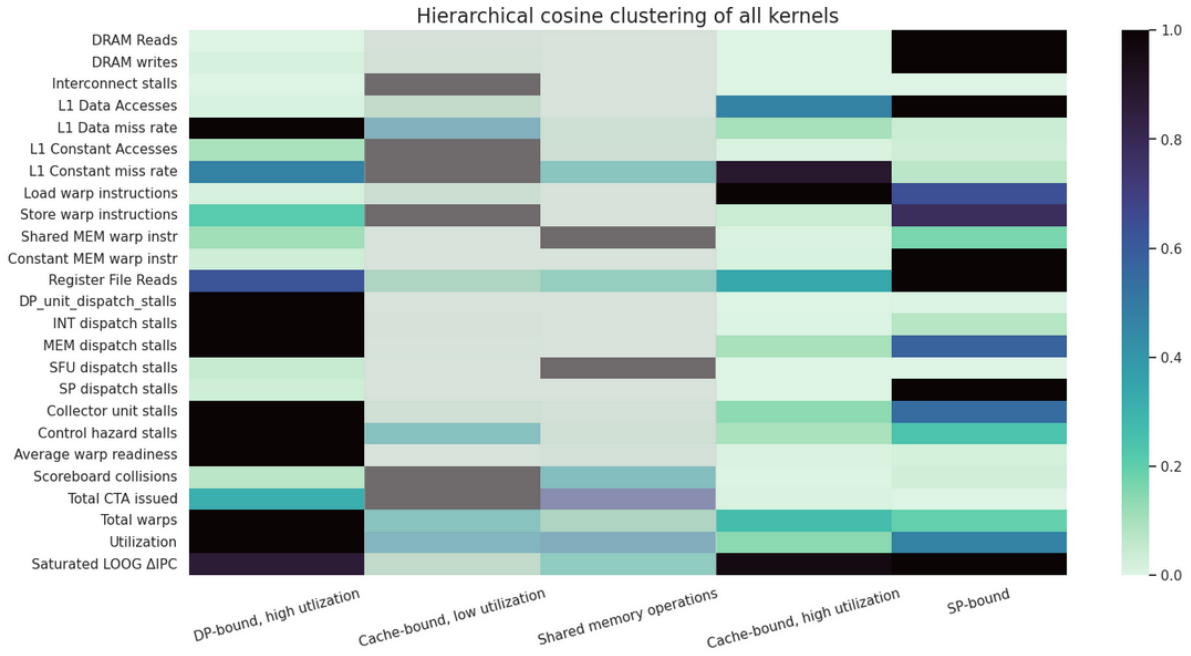


(α′) Αναλογία αθροιστικής απόκλισης (β′) Σκορ σιλουέττας ανά αριθμό συστάδων (γ′) Ταξινομημένα σκορ σημαντικότητας

Σχήμα 1.9: Εξαγωγή χαρακτηριστικών και ομαδοποίηση

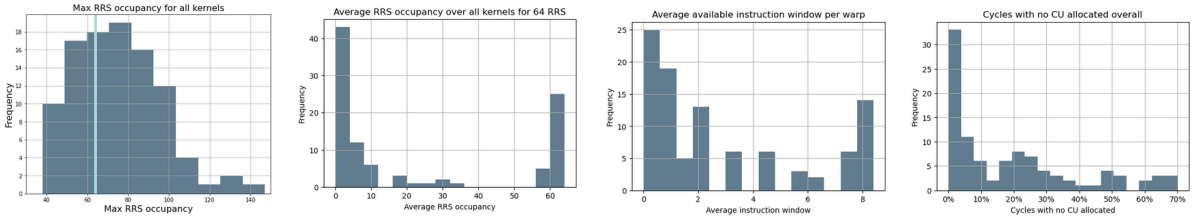


Σχήμα 1.10: Σχετική σημαντικότητα εξαγόμενων χαρακτηριστικών



Σχήμα 1.11: Σχετικές τιμές συστάδων

Σχήμα 1.12: Οπτικοποίηση των συστάδων με ιεραρχική ομαδοποίηση



(α′) Μέγιστες καταχωρήσεις RRS (β′) Μέση πληρότητα RRS (γ′) Μέσο παράθυρο εντολών ανά warp στα για RRS 64 (δ′) Κύκλοι χωρίς διατεθειμένα CUs

Σχήμα 1.13: Διαστασιολόγηση RRS

1.3.3 Διαστασιολόγηση του LOOG στην NVIDIA Quadro GV100

Μετά την υλοποίηση του LOOG στην έκδοση 4.1.0 του GPGPU-sim [23], κρίθηκε απαραίτητη η επανεκτίμηση του σωστού μεγέθους των σχετικών δομών, καθώς η αρχική υλοποίηση μελετήθηκε στο μοντέλο GeForce GTX1080Ti [48, 14], μια GPU βιντεοπαιχνιδιών, που υλοποιεί την Pascal, την προηγούμενη γενιά αρχιτεκτονικής της Volta. Δεδομένου ότι η νέα βασική μας πλατφόρμα GPU είναι μια GPU σταθμού εργασίας, η συμπεριφορά του LOOG αναμένεται να διαφέρει όσον αφορά το σωστό μέγεθος της στοίβας μετονομασίας καταχωρητών [12, 13] , του συλλέκτη τελεστών και του παραθύρου εντολών.

Στοίβα Μετονομασίας Καταχωρητών (RRS)

΄Οπως φαίνεται στον Πίνακα 1.1, η επιβάρυνση ισχύος και επιφάνειας της κλιμάκωσης RRS είναι ελάχιστη, ενώ η μέση απόδοση ανά CUDA kernel βελτιώνεται σημαντικά μόνο από 32 σε 64 RRS, που αποτελεί την τελική τιμή. Στο Σχήμα 1.13α′το απόλυτο μέγιστο είναι 146 και το απόλυτο ελάχιστο 41. Στο Σχήμα 1.13β′, παρουσιάζεται η μέση κατάληψη RRS σε όλους τους CUDA kernels με RRS ρυθμισμένο σε 64. Προφανώς, μια μειοψηφία (30%) των πυρήνων έχει μέση πληρότητα RRS για την οποία τα 64 RRS δεν επαρκούν, κάτι που επηρεάζει αμελητέα την απόδοση του backend. Οι τιμές κοντά στο 64 στο Σχήμα 1.13β′ παράγονται από υψηλή απόδοση εύρους ζώνης, πιθανώς σε CUDA kernels έντασης υπολογισμών.Το μέσο διαθέσιμο μήκος παραθύρου I ανά ωαρπ για μια διαμόρφωση RRS 64 απεικονίζεται στο Σχήμα 1.13γ′ (μέση τιμή 2,93).

RRS size	64	128	256
Power overhead	0.07	0.018%	0.047%
Area overhead	0.002%	0.004%	0.008%
DeltaIPC	7.52%	7.67%	7.67%

1.1: Βελτίωση επίδοσης και επιβαρύνσεις για κλιμακούμενα RRS

Παράθυρο εντολών

Οι επιβαρύνσεις ισχύος και επιφάνειας για την κλιμάκωση του παραθύρου εντολών με βασική τιμή $IWindow = 1$ (οι τιμές IWindow στον πίνακα είναι κανονικοποιημένες σε 1 εντολή Fetched, Decoded και Issued ανά κύκλο ανά μπλοκ επεξεργασίας SM) παρουσιάζονται στον πίνακα 5.4. Στη νέα αρχιτεκτονική με 4 μπλοκ επεξεργασίας ανά SM [49] (sub-cores[36, 24]) δεν είναι συμφέρον να αυξηθεί το Instruction Window, αλλά μπορεί να μειωθεί σε $\frac{1\_insn}{processing\_block \cdot cycle}$ χωρίς σημαντική επιδείνωση της επίδοσης.

IWindow	2	4	6	8	10	12	14	16
Power	4.79%	10.68%	17.58%	24.98%	34.06%	47.01%	60.47%	74.77%
Area	0.22%	0.95%	1.80%	2.71%	3.70%	4.85%	5.99%	7.23%

1.2: Επιβαρύνσεις κλιμακούμενου παράθυρου εντολών (Fetch - Decode - Issue scheduler throughput)

Μονάδες Συλλογής

Εϊναι εμφανές ότι οι Μονάδες Συλλογής, που χρησιμεύουν ως σταθμοί κράτησης του αλγορίθμου Tomasulo [15] είναι το πιο κρίσιμο τμήμα του, τόσο από την άποψη της αύξησης της απόδοσης όσο και από την άποψη των επιβαρύνσεων ισχύος και επιφάνειας. Η μέση αύξηση της ταχύτητας για όλους τους CUDA kernels που δοκιμάστηκαν καθώς και οι επιβαρύνσεις Area και Power παρουσιάζονται στον Πίνακα 1.3. Προφανώς, η βελτίωση του IPC σε LOOG κοραίνεται πέραν των 48 CUs για τον μέσο CUDA kernel, αποτελώντας το όριο σχεδιασμού για τις περαιτέρω δοκιμές.

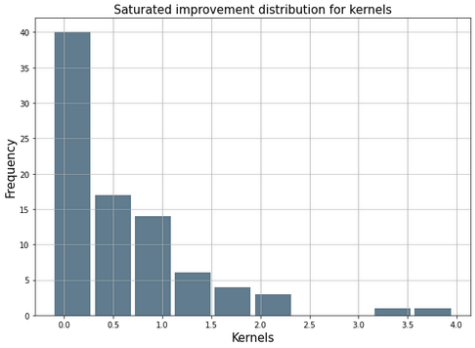
Ορίζουμε την κορεσμένη βελτίωση LOOG ως τη μέγιστη εκατοστιαία βελτίωση της απόδοσης που μπορεί να επιτευχθεί σε σχέση με τη βασική μικροαρχιτεκτονική. Είναι περίπου ίση με τη βελτίωση στις 48 CUs και ακριβώς ίση με τη βελτίωση στις 64 CUs. Το ιστόγραμμα της κορεσμένης βελτίωσης για όλους τους CUDA kernels που εκτελέστηκαν απεικονίζεται στο Σχήμα 1.14. Οι καμπύλες βελτίωσης για την κλιμάκωση CU ανά εκατοστημόριο CUDA kernel κορεσμένης βελτίωσης από την 50η έως την 90η παρουσιάζονται στο Σχήμα 1.15α′. Ορίζουμε αυθαίρετα την κλάση "LOOG-sensitive" ως την κλάση των εφαρμογών των οποίων η κορεσμένη βελτίωση LOOG είναι μεγαλύτερη από 100%.

Θέσαμε ως στόχο να σχεδιάσουμε έναν μηχανισμό αναδιαμόρφωσης στην λεπτομέρεια που βλέπουμε στον πίνακα 1.3, προκειμένου να βελτιστοποιήσουμε την επίδοση ή την ενεργειακή απόδοση για κάθε φόρτο εργασίας κατά την εκτέλεση.

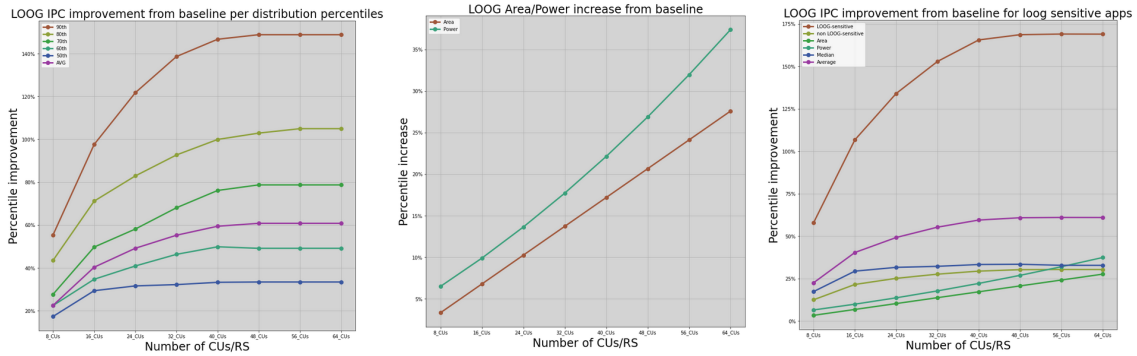
Στο Σχήμα 1.15β′, οι επιβαρύνσεις περιοχής και ισχύος συνδιαγραμμίζονται με τη βελτίωση του IPC ανά κατηγορία ευαισθησίας LOOG. Εμφανώς, ο διάμεσος CUDA kernels όσον αφορά την ευαισθησία LOOG είναι ουσιαστικά μη LOOG-ευαίσθητος. Η μέση βελτίωση IPC για τους CUDA kernels με ευαισθησία LOOG ξεπερνά το 160%.

Collector Units	8_CUs	16_CUs	24_CUs	32_CUs	40_CUs	48_CUs	56_CUs	64_CUs
Area overhead	3.33%	6.80%	10.26%	13.72%	17.19%	20.65%	24.12%	27.58%
Power overhead	6.50%	9.90%	13.64%	17.72%	22.13%	26.87%	31.95%	37.37%
Average DeltaIPC	22.29%	40.76%	48.57%	52%	59.66%	61.14%	61.16%	61.16%

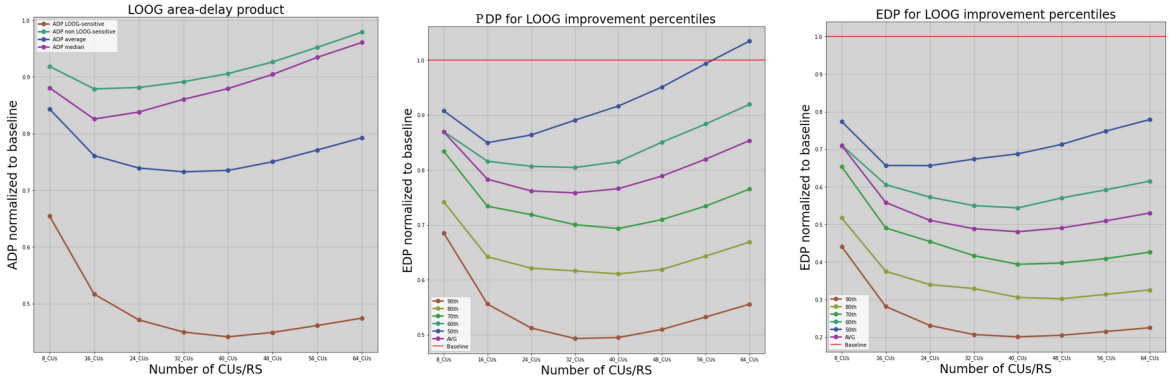
1.3: Επίδοση και επιβαρύνσεις κλιμακούμενων Σταθμών Συλλογής



Σχήμα 1.14: Ιστόγραμμα κορεσμένου IPC στο LOOG



(α′) Βελτίωση επίδοσης ανά εκατοστημόριο στο LOOG (β′) Επιβαρύνσεις κατά την κλιμάκωση του LOOG (γ′) Βελτίωση επίδοσης και επιβαρύνσεις



(δ′) Κλιμακούμενο γινόμενο Εμβαδού-Χρόνου εκτέλεσης (ε′) Κλιμακούμενο γινόμενο ισχύος-χρόνου εκτέλεσης (ς′) Κλιμακούμενο γινόμενο Ενέργειας-χρόνου εκτέλεσης

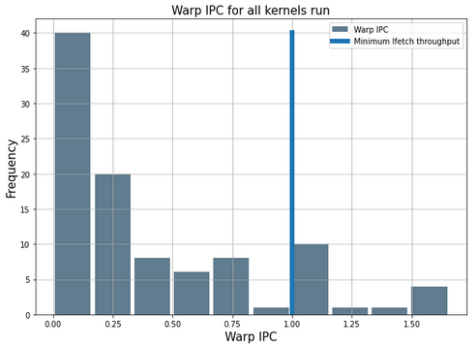
Σχήμα 1.15:

Προσαρμογή του front-end βάσει του LOOG στην NVIDIA Quadro GV100

Το LOOG επιβάλλει άμεσα μόνο τροποποιήσεις στο backend ενώ το frontend παραμένει ουσιαστικά άθικτο. Ωστόσο, με την αυξημένη απόδοση του backend ενδέχεται να δημιουργούνται νέα σημεία συμφόρησης, ή υποβέλτιστες διαμορφώσεις στο frontend. Έτσι, εξετάζουμε τη βέλτιστη συμπεριφορά των στοιχείων του frontend: Decoder, Issue scheduler και Instruction Buffer.

Benchmark	Suite	Kernel	Sat- ΔIPC
MST	Lonestar	Find Minimum	405%
LPS	Ispass-2009	3D Laplace calculation	396%
LIB	Ispass-2009	Path calculation	319%
Gramschmidt	Polybench	Gramschmidt	245%
Correlation	Polybench	Mean calculation	232%
MST	Lonestar	Verify minimum element	226%
MST	Lonestar	Find minimum element	194%
MST	Lonestar	Find minimum element 2	185%
QTC	Shoc	Compute degrees	180%
testAmr	Dragon	Refinement kernel	157%
Hotspot	Rodinia-3.1	Calculate temperature	153%
Backprop	Rodinia-3.1	Weights adjustment	145%
NN	Ispass-2009	Execute second layer	140%
Reduction	Shoc	Reduce	127%
Histo	Parboil	Input image	125%
Gramschmidt	Polybench	Initialization	119%
S3D	Shoc	Find Minmum	116%
Histo	Parboil	Intermediate kernel	115%
Histo	Parboil	Histogram calculation	112%
CFD	Rodinia-3.1	Compute	110%
Stencil2D	Parboil	Stencil kernel	106%
FFT	Shoc	FFT kernel	102%

1.4: Εφαρμογές " ευάισθητες στο LOOG" και οι κορεσμένες βελτιώσεις επίδοσής τους.



Σχήμα 1.16: Εντολές warp ανά κύκλο

Προσαρμογή Fetch-Decode

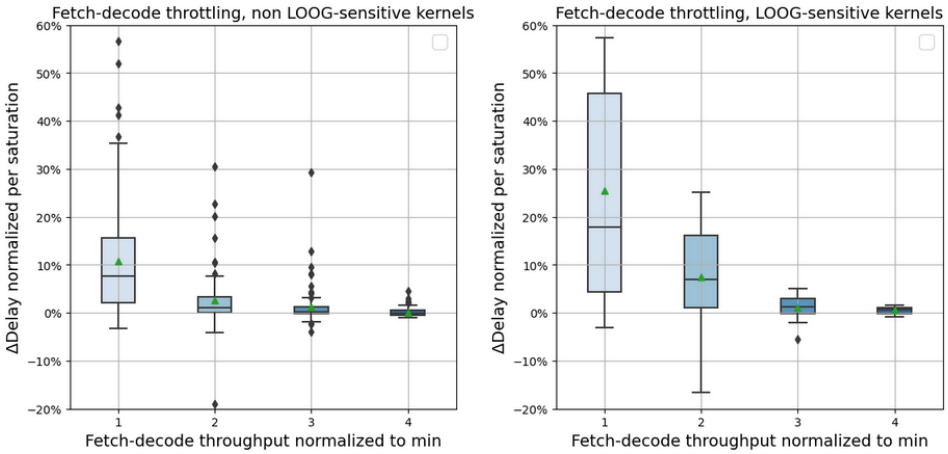
Η επέκταση του ορισμού των Instructions Per Cycle (Εντολές ανά κύκλο) αναφέρεται συνήθως σε εντολές νήματος ανά κύκλο, χωρίς να λαμβάνει υπόψη τις εντολές warp, που είναι η πραγματική μονάδα που εκτελείται στο backend:

$$IPC = \frac{scalar\_thread\_instructions}{total\_cycles} \tag{1.2}$$

Δεδομένου του ενιαίου μετρητή προγράμματος της αρχιτεκτονικής Pascal και του μοντέλου απόκλισης ενεργής μάσκας ανά warp (όπου τα νήματα μεταξύ των warp δεν μπορούν να συνενωθούν), επαναπροσδιορίζουμε τη μετρική IPC ως προς την απόδοση εντολών του frontend ως εξής:

$$\begin{aligned} W\_IPC = \frac{warp\_insn}{total\_cycles} &= \frac{scalar\_thread\_insn}{total\_cycles} \cdot \frac{warp\_instructions}{scalar\_thread\_insn} \\ &= \frac{IPC}{warp\_size \cdot warp\_occupancy} \end{aligned} \tag{1.3}$$

Έτσι μπορούμε να συγκρίνουμε το Warp IPC, το οποίο είναι μια μετρική του backend, με την αντίστοιχη του frontend. Όπως απεικονίζεται στο Σχήμα 1.16, όταν εκτελείται με κορεσμένο frontend, η απόδοση του backend σε επίπεδο SM και ο μέσος όρος σε χρονική βάση είναι κάτω



(α′) (β′)

Σχήμα 1.17: Περιορισμός απόδοσης Fetch-Decode

από την ελάχιστη απόδοση του frontend (που ισοδυναμεί με 1 εντολή που λαμβάνεται ανά SM-wide L1 Icache ανά κύκλο) για το 83% των εφαρμογών. Ως εκ τούτου, η έννοια των front-end bound και back-end bound εφαρμογών δεν είναι εφαρμόσιμη στη GPU. Στο Σχήμα 1.17α′ και στο Σχήμα 1.17β′, εμφανίζεται η επιβάρυνση καθυστέρησης για κάθε ένα από τα μειωμένα εύρη ζώνης του αποκωδικοποιητή και για κάθε κατηγορία ευαισθησίας LOOG.

Στον Πίνακα 1.5, παρουσιάζεται το IPC, κανονικοποιημένο ως προς τη βασική απόδοση frontend (8 ανά SM ανά κύκλο ή 2 ανά μπλοκ επεξεργασίας ανά κύκλο) για κάθε μία από τις περιορισμένες τιμές εύρους ζώνης Fetch-Decode. Οι τιμές αυτές κανονικοποιούνται όπως εξηγήθηκε παραπάνω. Η εξοικονόμηση ισχύος και εμβαδού λόγω περιορισμού απόδοσης σε αυτά τα στάδια και στο στάδιο έκδοσης είναι 4,57% και 0,22% αντίστοιχα.

Decoder BW	1	2	3	4
LOOG-sensitive IPC	89,22%	97,43%	98,78%	99,88%
LOOG-insensitive IPC	74,43%	92,54%	98,84%	99,48%

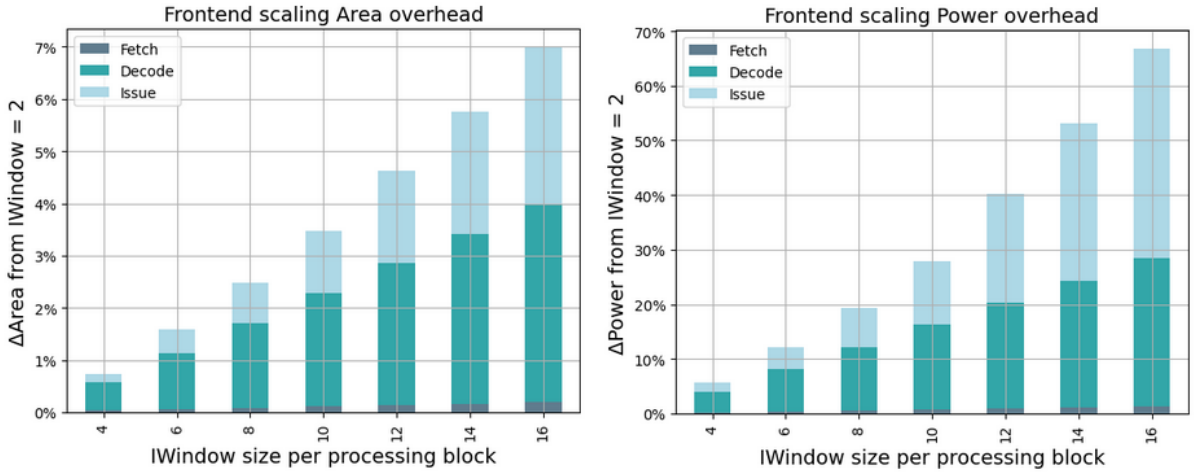
1.5: Εντολή ανά κύκλο για απόδοση Fetch-Decode κανονικοποιημένη στο $\frac{1\_insn}{SM \cdot cycle}$.

IWindow	4	6	8	10	12	14	16
Fetch	0.03%	0.05%	0.08%	0.11%	0.14%	0.16%	0.19%
Decode	0.54%	1.09%	1.63%	2.17%	2.72%	3.26%	3.80%
Issue	0.16%	0.44%	0.77%	1.19%	1.77%	2.33%	3.00%
Total	0.73%	1.58%	2.48%	3.47%	4.62%	5.76%	6.99%

1.6: Επιβαρύνσεις εμβαδού κάθε σταδίου σε σχέση με παράθυρο εντολών 2

IWindow	4	6	8	10	12	14	16
Fetch	0.20%	0.39%	0.59%	0.79%	0.98%	1.18%	1.38%
Decode	3.86%	7.72%	11.58%	15.44%	19.30%	23.17%	27.03%
Issue	1.56%	4.09%	7.09%	11.70%	20.00%	28.79%	38.38%
Total	5.62%	12.21%	19.27%	27.93%	40.29%	53.13%	66.78%

1.7: Επιβαρύνσεις ισχύος κάθε σταδίου σε σχέση με παράθυρο εντολών 2



(α′) (β′)

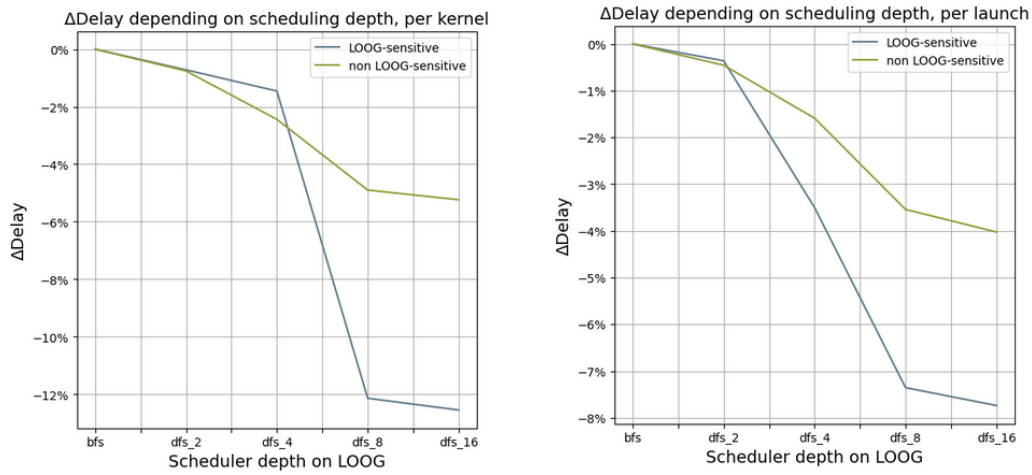
Σχήμα 1.18: Επιβαρύνσεις για κλιμάκωση του front-end (σε σχέση με παράθυρο εντολών 2)

Προσαρμογή Issue scheduling

Όπως απεικονίζεται στα Σχήματα 1.19α′ και 1.19β′ , ένας χρονοπρογραμματιστής με τη δυνατότητα να χρησιμοποιεί παράθυρα εντολών μεγαλύτερου βάθους από μεμονωμένα warps, παρέχει σημαντικά καλύτερα αποτελέσματα με το LOOG, ειδικά για εφαρμογές ευαίσθητες στο LOOG. Η βελτίωση αυτή είναι κορεσμένη πέραν των 8 εντολών. Συγκεκριμένα στο Σχήμα 1.19α′ βλέπουμε μια προκατάληψη των πυρήνων single-launch προς την LOOG-ευαισθησία, η οποία επανεξετάζεται εκτενώς στην ενότητα 5.8.2. Το βάθος του scheduling αντιστοιχεί στο issue_depth στον αλγόριθμο 1.

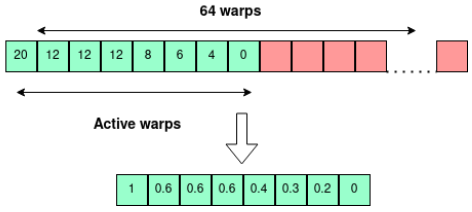
1.3.4 Αναδιαμόρφωση Instruction Buffer

Δεδομένου ότι οι CUDA kernels που είναι ευαίσθητοι στο LOOG επωφελούνται περισσότερο από βαθύτερα παράθυρα χρονοπρογραμματισμού Issue, παρουσία ετερογένειας ILP μεταξύ των warps περισσότερη ταυτόχρονη πρόσβαση στον Operand Collector θα πρέπει να δίνεται σε warps με υψηλό ILP. Δεδομένης της ευρείας κατανομής των προαναφερθέντων μετρικών μεταξύ των warps για τους περισσότερους CUDA kernels, καθιστούμε την κατάτμηση Ibuffer ανά warp παραμετροποιήσιμη αντί για στατική (όπως στο βασικό μοντέλο [24]) και εξασφαλίζουμε ότι προσαρμόζεται στο εκμεταλλεύσιμο ILP του αντίστοιχου warp κατά την εκτέλεση.



(α′) (β′)

Σχήμα 1.19: Delay improvement across Issue scheduling depths



Σχήμα 1.20: Τα δεδομένα που παρουσιάζονται παρακάτω

Μετρική αναδιαμόρφωσης χρησιμοποιημένων καταχωρήσεων RAT

Η αξιοοποίηση των χρησιμοποιούμενων καταχωρήσεων RAT για την απο-προτεραιοποίηση των warps, παρέχει έναν ομαλό προγραμματισμό εντολών στο backend για warps με ανεξάρτητες εντολές. Τα warps με υψηλές εξαρτήσεις μπορούν να καταλαμβάνουν τις CUs όταν οι πρώτες έχουν εξαντλήσει το IWindow τους ή έχουν αντιμετωπίσει ακινητοποιήσεις της Icache.

Μετρική αναδιαμόρφωσης ετοιμότητας warp

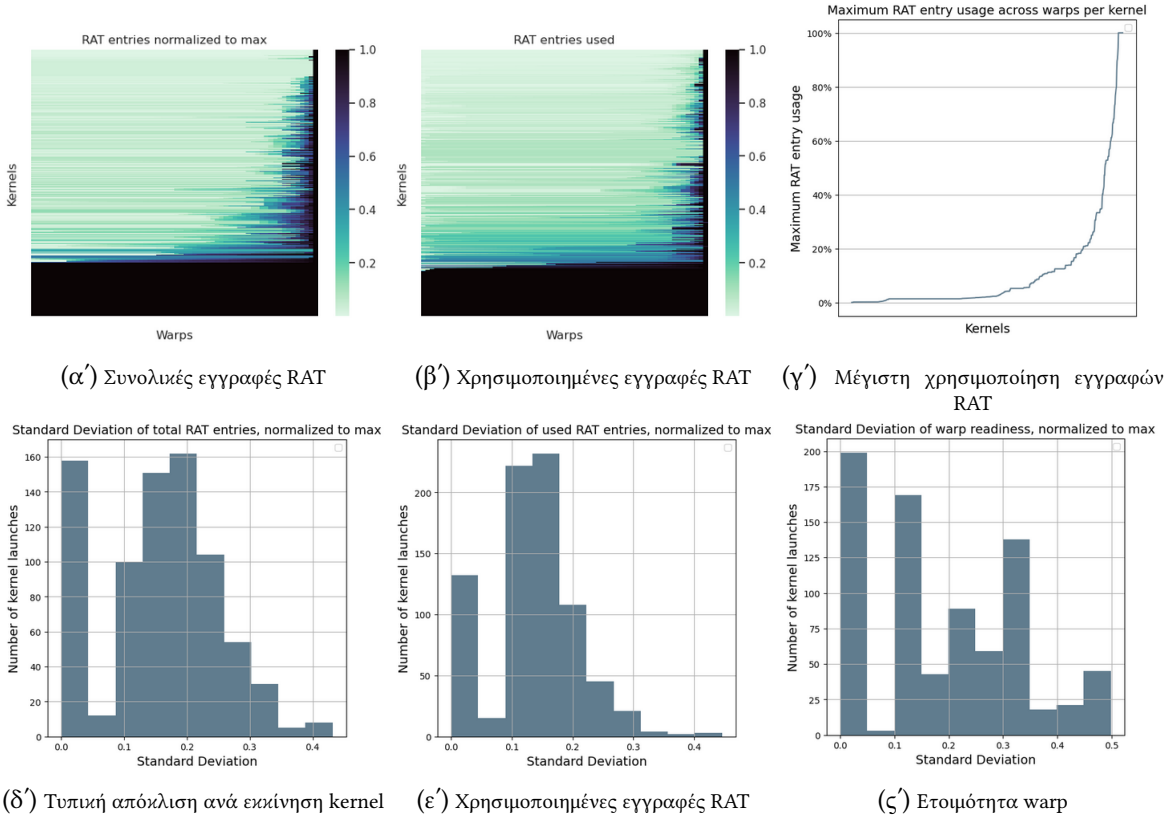
Η ετοιμότητα των warps ποικίλλει μεταξύ των warps παρουσία κινδύνων ελέγχου λόγω βρόχων, που προκαλούν περισσότερα χτυπήματα στην Icache και λόγω της προτίμησης χρονοπρογραμματισμού έκδοσης των αντίστοιχων warps, που προκαλεί συχνό άδειασμα των καταχωρίσεων IBuffer τους. Όταν ένα warp έχει προτεραιότητα για Issue, αυξάνεται η μετρική ετοιμότητάς του, παρέχοντάς του ακόμη μεγαλύτερη προτεραιότητα κ.ο.κ.

Πολιτική αναδιαμόρφωσης δίκαιου διαχωρισμού (Split)

Όταν χρησιμοποιείται η πολιτική Split που παρουσιάζεται στην εξίσωση 5.4, οι διαθέσιμες εγγραφές Ibuffer κατανέμονται δίκαια μεταξύ των warps.

Algorithm 1
ά μό warp

```
order_warps_RR()
total_issued = 0
while total_issued < max_issue_per_cycle do
  get_next_warp()
  issued = 0
  while issued < issue_depth && total_issued < max_issue_per_cycle do
    if !ibuffer_empty() && !warp_waiting() && pipeline_avail() then
      issue_instruction()
      issued += 1
      total_issued += 1
    else
      break
    end if
  end while
end while
```



(α′) Συνολικές εγγραφές RAT (β′) Χρησιμοποιημένες εγγραφές RAT (γ′) Μέγιστη χρησιμοποίηση εγγραφών RAT

(δ′) Τυπική απόκλιση ανά εκκίνηση kernel των συνολικών εγγραφών RAT, καν. στο μέγιστο (ε′) Χρησιμοποιημένες εγγραφές RAT (ς′) Ετοιμότητα warp

Σχήμα 1.21

Πολιτική αναδιαμόρφωσης winner-take-all

Η πολιτική "winner-take-all" ανακατανέμει άπληστα τις διαθέσιμες καταχωρήσεις μεταξύ των warps με τις υψηλότερες βαθμολογίες.

$$
\begin{aligned}
warp\_score &= \frac{1}{1 + reverse\_reconf\_metric} \\
warp\_score &= reconf\_metric \\
total\_score &= \sum_{warp \in Ready\_Warps} warp\_score(warp) \\
warp\_entry\_pool &= \sum_{warp \in Ready\_Warps} current\_entries(warp) \\
entries(warp) &= floor(warp\_entry\_pool \cdot \frac{warp\_score}{total\_score}) \\
remaining\_entries &= warp\_entry\_pool - \sum_{warp \in Ready\_Warps} entries(warp) \\
remaining\_score(warp) &= \frac{warp\_score}{total\_score} - floor(\frac{warp\_score}{total\_score})
\end{aligned}
\tag{1.4}
$$

Algorithm 2
Winner-take-all IBuffer reconfiguration policy

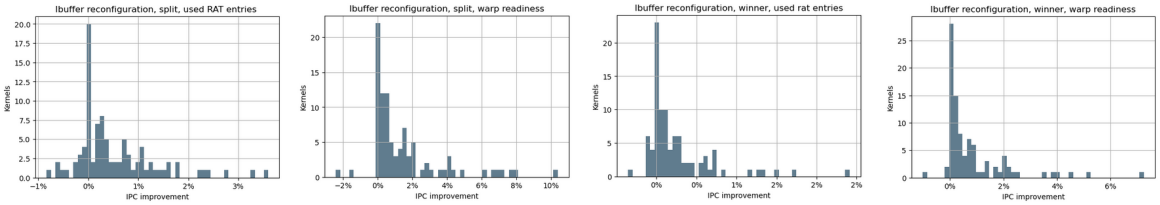
```
sort_warps_by_score()
while warp_entry_pool do
  warp = get_next_warp()
  entries[warp] = min(max_entries_per_warp, warp_entry_pool)
  warp_entry_pool -= entries[warp]
end while
```

Αποτελέσματα αναδιαμόρφωσης

Τα αποτελέσματα που προκύπτουν από το συνδυασμό των δύο πολιτικών αναδιαμόρφωσης με τις δύο μετρικές, σε όλους τους CUDA kernels, απεικονίζονται στο Σχήμα 1.22α′, στα Σχήματα 1.22β′, 1.22γ′ και 1.22δ′. Η πολιτική διαχωρισμού με τη χρήση της μετρικής ετοιμότητας warp παρέχει σημαντικά αποτελέσματα, με μέση επιτάχυνση 4,4% για CUDA kernels ευαίσθητους σε LOOG, έως και 10,2%.

1.3.5 Εκτός σειράς αναδιαμόρφωση

Η βελτιστοποίηση της επίδοσης ή των αριθμητικών μεγεθών που λαμβάνουν υπόψη τη διάχυση ισχύος (PDP, EDP) απαιτεί πολύ διαφορετικές διαμορφώσεις μονάδων συλλέκτη για διαφορετικούς CUDA kernels. Συγκεκριμένα, η κλιμάκωση των CUs παράγει αρκετή αύξηση των επιδόσεων για ορισμένα φορτία εργασίας (ευαίσθητα στο LOOG) που δικαιολογεί την επιβάρυνση ισχύος και επιφάνειας των μεγαλύτερων διαμορφώσεων, ενώ για άλλα, καμία διαμόρφωση LOOG δεν παρέχει αρκετά μεγάλη επιτάχυνση ώστε να δικαιολογούνται οι επιβαρύνσεις. Οι βελτιώσεις του LOOG σε



(α′) Πολιτική διαχωρισμού, μετρική χρησιμοποιημένων εγγραφών RAT (β′) Πολιτική διαχωρισμού, μετρική ετοιμότητας warp (γ′) Πολιτική Winner-Take-All policy, μετρική χρησιμοποιημένων εγγραφών RAT (δ′) Πολιτική Winner-Take-All policy, μετρική ετοιμότητας warp

Σχήμα 1.22: Βελτίωση IPC ανά μετρική και πολιτική αναδιαμόρφωσης

	8_CUs	16_CUs	24_CUs	32_CUs	40_CUs	48_CUs	56_CUs	64_CUs
90th	55.39%	97.63%	121.77%	138.65%	146.71%	148.84%	148.85%	148.89%
80th	43.50%	71.17%	82.92%	89.10%	99.94%	104.97%	105.10%	105.40%
70th	27.65%	49.72%	58.13%	68.07%	76.13%	78.74%	79.66%	79.50%
60th	22.42%	34.69%	40.89%	46.32%	49.84%	49.12%	49.31%	49.41%
50th	17.32%	29.35%	31.58%	32.18%	33.26%	33.39%	32.79%	32.77%
AVG	22.50%	40.32%	49.15%	55.24%	59.44%	60.79%	60.96%	60.93%

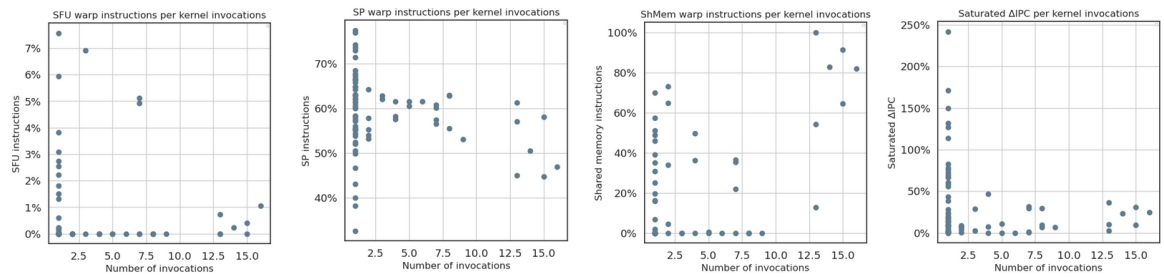
1.8: Βελτίωση IPC ανά διαμόρφωση και εκατοστημόριο

σχέση με τη βασική μικροαρχιτεκτονική για τα διάφορα εκατοστημόρια της κορεσμένης βελτίωσης παρουσιάζονται με μεγαλύτερη ακρίβεια στον πίνακα ;;

Συμπεριφορά εκκινήσεων kernels με το LOOG

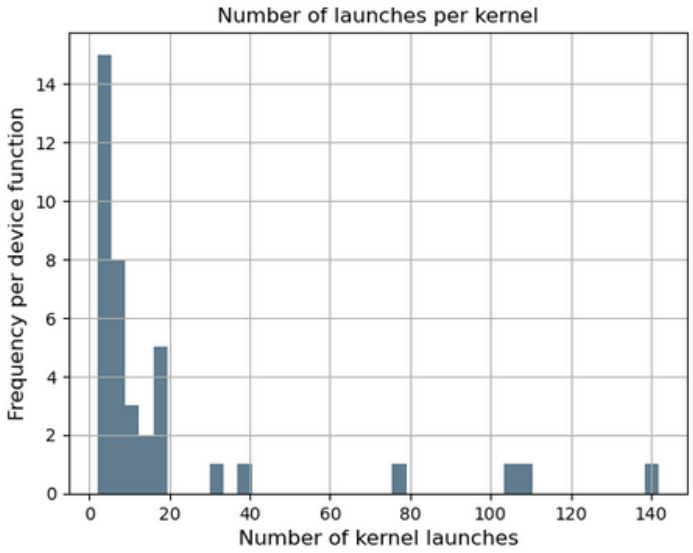
Όσον αφορά τον αριθμό των εκκινήσεων για τους εξεταζόμενους CUDA kernels, στον πίνακα 3.1 42% των πυρήνων είχαν συνολικά μία εκκίνηση. Η κατανομή του αριθμού των εκκινήσεων για τους υπόλοιπους CUDA kernels απεικονίζεται στο Σχήμα ;;.

Όπως απεικονίζεται στο σχήμα 1.23β′, 1.23γ′ και 1.23α′, υπάρχει σαφής συσχέτιση μεταξύ των υπολογιστικών εντολών και του αριθμού των κλήσεων. Όπως φαίνεται στο Σχήμα 1.25β′, οι προσπελάσεις σταθερής μνήμης των φόρτων εργασίας μας συσχετίζονται κυρίως με εντολές μνήμης παραμέτρων. Πράγματι, αυτό οφείλεται στους χαμηλούς συνολικούς χρόνους εκτέλεσης. Η βελτίωση του κορεσμένου IPC στο LOOG συσχετίζεται έντονα αρνητικά με (Σχήμα 1.25α′). Επομένως, ο χαμηλός χρόνος εκτέλεσης εκκίνησης είναι μια σημαντική παράμετρος κατά τη σκιαγράφηση πυρήνων σύμφωνα με τη βελτίωση του LOOG.

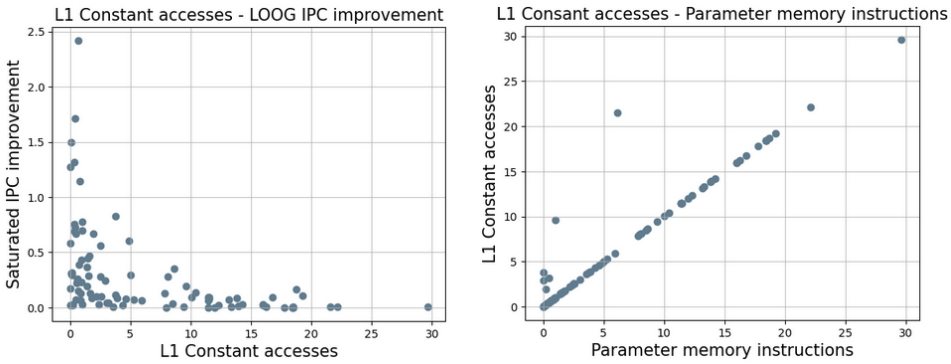


(α′) Δυναμικές εντολές SFU (β′) Δυναμικές εντολές SP (γ′) Δυναμικές εντομές μοιραζόμενης μνήμης (δ′) Κορεσμένη βελτίωση επίδοσης - εκκινήσεις

Σχήμα 1.23: Τύποι εντολών - εκκινήσεις kernels. Εμφανής συσχέτιση κλήσεων - εντολών μνήμης



Σχήμα 1.24: Αριθμός κλήσεων ανά kernel



(α′) Η κορεσμένη βελτίωση επίδοσης είναι αντιστρόφως ανάλογη των προσβάσεων L1C

(β′) Οι προσβάσεις L1C εξυπηρετούν κυρίως εντολές μνήμης παραμέτρων

Σχήμα 1.25:

Χρησιμοποιούμενες μετρικές

Ο καθορισμός της βέλτιστης αναδιαμόρφωσης είναι σχετικός με τον απαιτούμενο δείκτη αξίας, ο οποίος αντιπροσωπεύει κάποιο είδος τομέα αποδοτικότητας που πρέπει να βελτιστοποιήσουμε. Μαζί με αυτό χρησιμοποιούνται και αριθμοί αξίας που λαμβάνουν υπόψη την ισχύ:

- *Κορεσμένη βελτίωση IPC 98%*
  Θεωρείται ότι είναι βελτιστοποιημένη όταν η τρέχουσα διαμόρφωση παρέχει απόδοση εντός ενός αυθαίρετου 2% της κορεσμένης απόδοσης βελτίωσης (μέγιστη επιτεύξιμη βελτίωση IPC στην πιο κλιμακούμενη διαμόρφωση LOOG).

- *Κορεσμένη βελτίωση IPC 95%*
  Το IPC θα πρέπει να είναι εντός του 5% της μέγιστης επιτεύξιμης βελτίωσης IPC.

- *Power-Delay Product*
  Ισοδυναμεί με τη συνολική διάχυση ενέργειας της εφαρμογής.

- *Energy-Delay Product*
  Ισοδύναμο με το γινόμενο $Power - Delay^2$, λαμβάνοντας περισσότερο υπόψη το χρόνο εκτέλεσης.

Αυτές είναι οι μετρικές με βάση τις οποίες θα αξιολογηθεί και το μοντέλο αναδιαμόρφωσης, σε σχέση με τις βέλτιστες τιμές τους. Συχνά θα αναφερόμαστε σε αυτές ως ¨μετρικές αναδιαμόρφωσης¨.

Είδη αναδιαμόρφωσης

- Βάσει του επιπέδου υλοποίησης του ελεγκτή αναδιαμόρφωσης:
  Επίπεδο λογισμικού/ υλικού,

- Βάσει της εισόδου του ελεγκτή αναδιαμόρφωσης:
  Βέλτιση αναδιαμόρωση (όπου οι βέλτιστες διαμορφώσεις είναι γνωστές για κάθε εφαρμογή και μετρική αναδιαμόρφωσης) / Αναδιαμόρφωση κατά την εκτέλεση (όπου οι βέλτιστες διαμορφώσεις προβλέπονται από μετρητές υλικού)

- Βάσει του χρονικού βαθμού λεπτομέρειας:
  Στατική (αναδιαμόρφωση ανά εφαρμογή) / Ημι-δυναμική (αναδιαμόρφωση ανά εκκίνηση kernel) / Πρώτης εκκίνησης (πρώτη εκκίνηση εντός σειράς και σταθερή διαμόρφωση για τις υπόλοιπες εκκινήσεις)

Προσδιορισμός βέλτιστων διαμορφώσεων

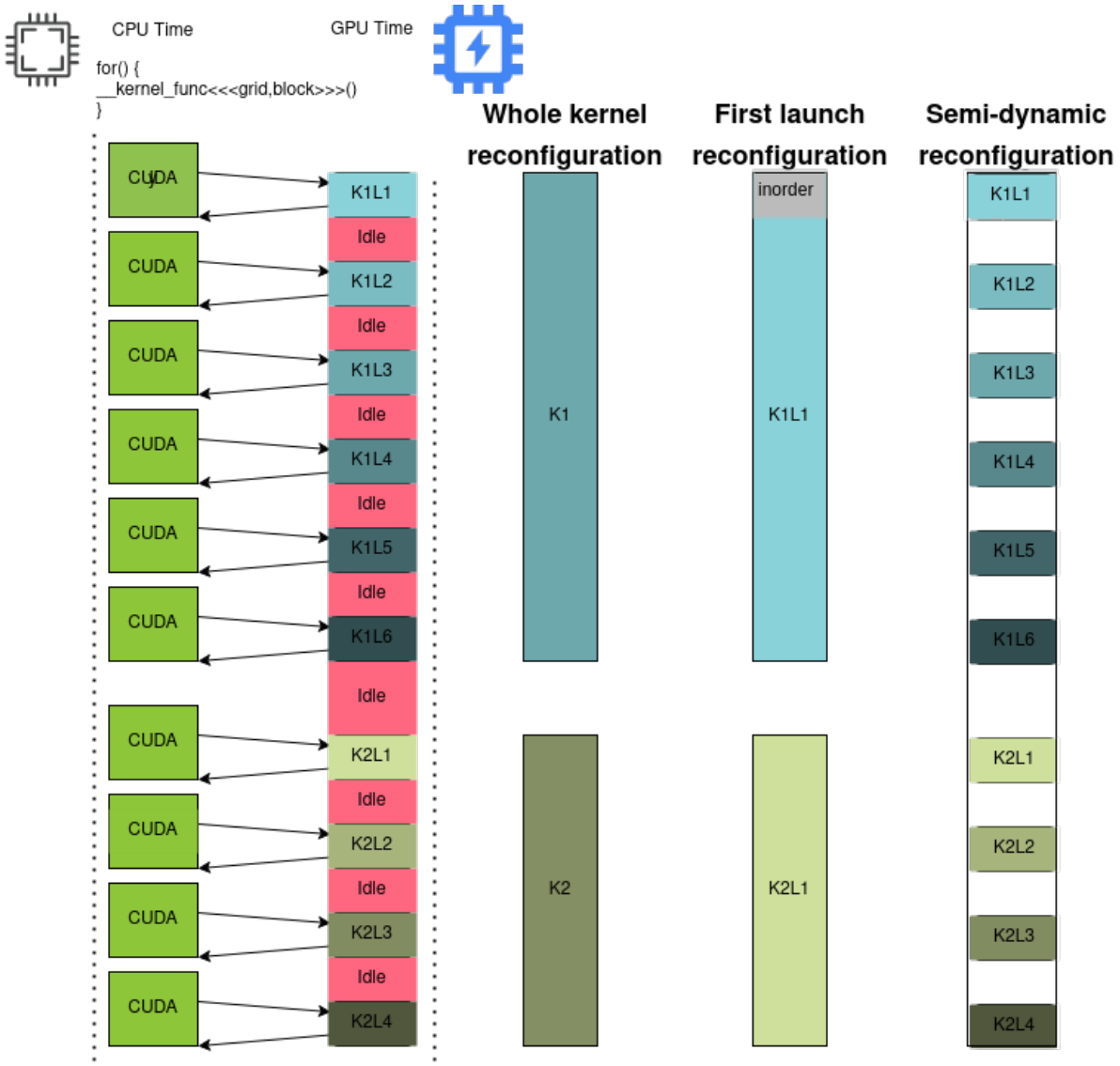
Values	98% saturation	PDP	EDP
LOOG-sensitive			
min	24	16	24
max	48	48	48
median	40	32	40
LOOG-insensitive			
min	inorder	inorder	inorder
max	48	48	48
median	32	16	16

1.9: Βέλτιστες διαμορφώσεις ανά kernel και μετρική

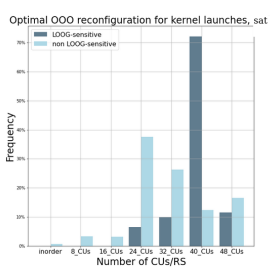
Βέλτιστη διαμόρφωση για CUDA kernels μίας εκκίνησης

Για τη βέλτιστη αναδιαμόρφωση για CUDA kernels μονής εκκίνησης με βάση τη βελτιστοποίηση όλων των μετρικών που ορίζονται, λαμβάνουμε αποτελέσματα για τη μέση βελτίωση του χρόνου εκτέλεσης και τη μέση ενεργειακή απόδοση σε όλους τους CUDA kernels και τις κατηγορίες ευαισθησίας LOOG στον πίνακα 1.10.

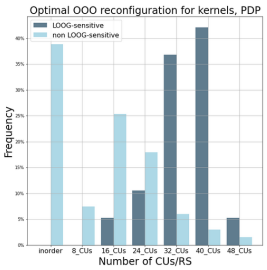
Τα διαγράμματα 1.28α′ και 1.28β′ απεικονίζουν σαφώς την παρατήρηση ότι οι CUDA kernels πολλαπλών εκκινήσεων τείνουν να μην είναι ευαίσθητοι στο LOOG.



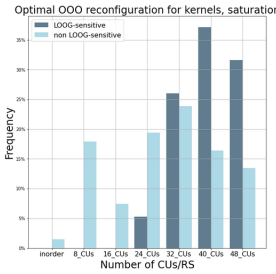
Σχήμα 1.26: Τύποι αναδιαμόρφωσης βάσει χρονικής λεπτομέρειας



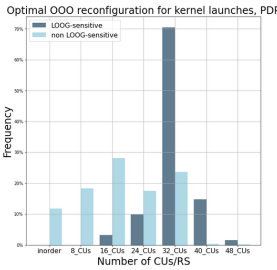
(A) sat98,launches, LOOG-sensitivity



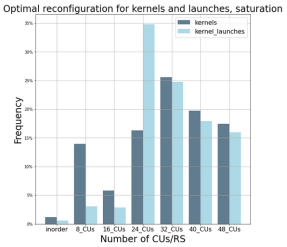
(B) PDP, kernels, LOOG-sensitivity



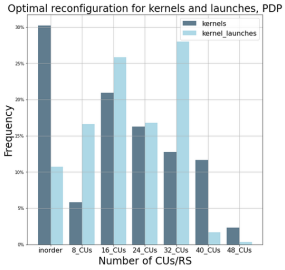
(C) sat98, kernels, LOOG-sensitivity



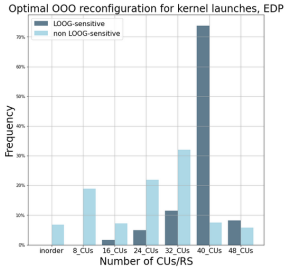
(D) PDP, launches, LOOG-sensitivity



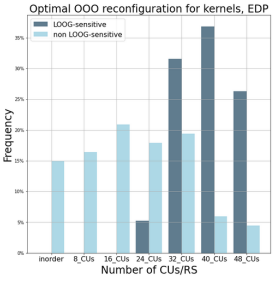
(E) sat98, kernels and launches



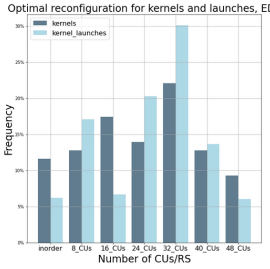
(F) PDP, kernels and launches



(G) EDP, launches, LOOG-sens

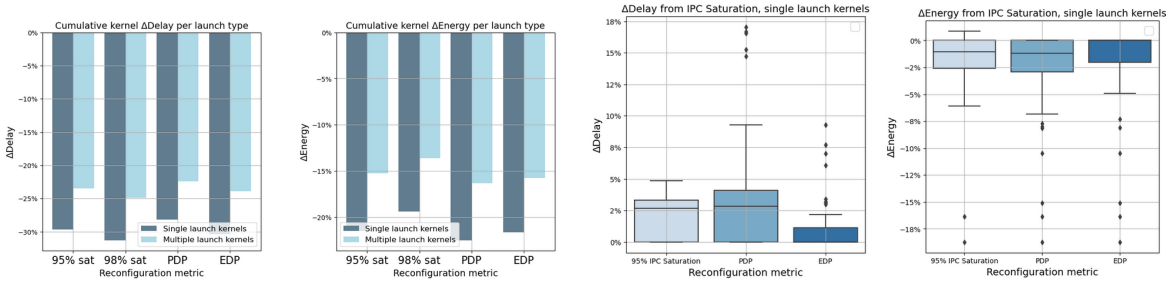


(H) EDP, kernels, LOOG-sens



(I) EDP, kernels and launches

μ 1.27: Βέλτιστες διαμορφώσεις CU ανά CUDA kernels, εκκινήσεις, μετρικές αναδιαμόρφωσης και ευαισθησία στο LOOG



(α′) Βελτίωση χρόνου εκτέλεσης (β′) Βελτίωση ενέργειας (γ′) Επιδείνωση χρόνου εκτέλεσης από κορεσμό (δ′) Βελτίωση ενέργειας από κορεσμό

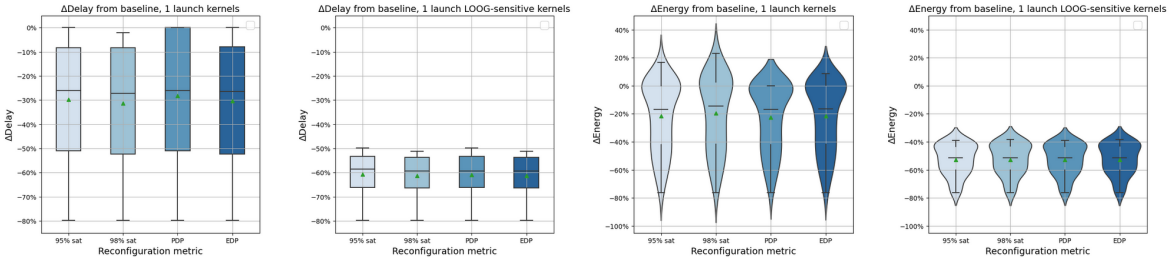
Σχήμα 1.28:

Όπως φαίνεται στο Σχήμα 1.28γ′, η βελτιστοποίηση για τις άλλες μετρικές προκαλεί μια αναμενόμενη μικρή επιδείνωση της καθυστέρησης σε σύγκριση με το 98%sat. Όσον αφορά τη μεταβολή ενέργειας, στο Σχήμα 1.28 παρατηρούμε ότι κατά μέσο όρο - 1,1%, -1,4% και -0,9% μτεαβολή ενέργειας παράγεται από τις μετρικές 95% IPC saturation, PDP και EDP αντίστοιχα.

Σημειώνεται ότι, όπως φαίνεται στις κατανομές ενέργειας στο Σχήμα 1.29γ′, εκτός από την αναμενόμενη βελτιστοποίηση του PDP, ένα σύνολο πυρήνων δεν κερδίζει αρκετά σημαντική επιτάχυνση ώστε να αντισταθμίσει την επιβάρυνση ισχύος των μεγαλύτερων διαμορφώσεων LOOG. Αυτοί οι CUDA kernels αντιπροσωπεύουν 25% για τις μετρικές κορεσμού 95% και EDP και 27% για τη μετρική κορεσμού 98%. Όπως φαίνεται στο Σχήμα 1.29δ′, όλοι οι CUDA kernels που είναι ευαίσθητοι στο LOOG μειώνουν τη διάχυση ενέργειας κατά τουλάχιστον 38% σε βέλτιστες διαμορφώσεις.

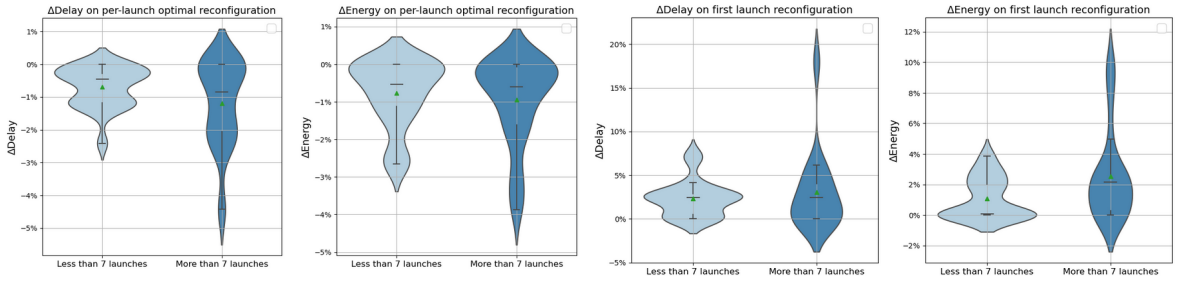
	95% sat	98% sat	PDP	EDP
Delay				
Total	-29.4%	-30.1%	-28.2%	-30%
LOOG-sensitive	-59.8%	-60%	-60%	-60%
Energy				
Total	-20%	-19%	-21%	-20%
LOOG-sensitive	-53%	-52.9%	-52.8%	-52.8%

1.10: Βέλτιστες διαμορφώσεις για kernels μονής εκκίνησης



(α′) Χρόνος εκτέλεσης, γενικοί kernels (β′) Χρόνος εκτέλεσης, kernels ευαίσθητοι στο LOOG (γ′) Ενέργεια, γενικοί kernels (δ′) Ενέργεια, kernels ευαίσθητοι στο LOOG

Σχήμα 1.29: Χρόνος εκτέλεσης και ενέργεια για τις βέλτιστες διαμορφώσεις σε kernels μονής εκκίνησης



(α′) Βελτίωση χρόνου εκτέλεσης σε ημι-δυναμική διαμόρφωση (β′) Βελτίωση ενέργειας σε ημι-δυναμική διαμόρφωση (γ′) Χειροτέρευση χρόνου εκτέλεσης σε διαμόρφωση πρώτης εκκίνησης (δ′) Χειροτέρευση ενέργειας σε διαμόρφωση πρώτης εκκίνησης

Σχήμα 1.30:

CUDA kernels πολλαπλών εκκινήσεων

Ημι-δυναμική αναδιαμόρφωση

Όπως φαίνεται στα σχήματα 1.30α′ και 1.30β′, οι μέσοι όροι κατανομής των δέλτα για την καθυστέρηση και την ενέργεια κατά τη διαμόρφωση ανά εκκίνηση και τη χρήση της κατάλληλης μετρικής (κορεσμός IPC για την καθυστέρηση και PDP για την ενέργεια) δεν διαφέρουν σημαντικά από τη στατική αναδιαμόρφωση.

Εκτός από μερικές ακραίες τιμές με υπερβολικό αριθμό εκκινήσεων CUDA kernel που φαίνονται στον πίνακα 1.11 , των οποίων οι είσοδοι και οι παράμετροι ενδεχομένως αλλάζουν με την πάροδο του χρόνου, τα αποτελέσματα είναι ίδια για τις δύο ομάδες με βάση τον αριθμό των εκκινήσεων.

Benchmark suite	Name	Description	Characteristics	Kernels	Launches	deltaDelay	deltaEnergy
Rodinia-3.1	NW	Sequence alignment	Compute intensive	1	104	-2.4%	-4.3%
Rodinia-3.1	CFD	Fluid Dynamics	Compute intensive	3	15	-2%	-3.5%
Polybench	3DConvolution	3D filtering	Compute & BW	1	32	-1.8%	-3%
SHOC	Spmv	Sparse Vector Mul	Sparse lin Algebra	1	129	-2.1%	-4.1%
SHOC	Scan	Parallel Scan	Memory BW	3	45	-1.8%	-3.1%

1.11: Kernels πολλών εκκινήσεων με ανομοιόμορφη κλιμάκωση εκτός σειράς

Στατική αναδιαμόρφωση πρώτης εκκίνησης

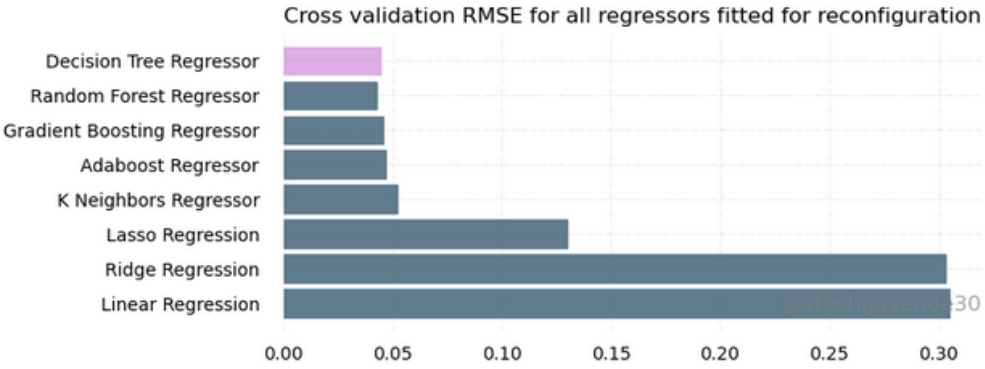
Λαμβάνοντας υπόψη αυτά τα αποτελέσματα, είναι λογικό να εξετάσουμε αν η βέλτιστη σκιαγράφηση του προφίλ της πρώτης εκκίνησης παρέχει μια αξιοπρεπή διαμόρφωση για την υπόλοιπη εκτέλεση του CUDA kernel. Όπως απεικονίζεται στο Σχήμα 1.30γ′, για CUDA kernels πολλαπλών εκκινήσεων με λιγότερες από 7 εκκινήσεις και περισσότερες από 7 εκκινήσεις αντίστοιχα, όταν εφαρμόζεται η βέλτιστη αναδιαμόρφωση πρώτης εκκίνησης σε σύγκριση με τη βέλτιστη στατική αναδιαμόρφωση, εισάγεται μια μέση επιβάρυνση καθυστέρησης 2,6% και 3,9%. Οι αντίστοιχες επιβαρύνσεις ενέργειας είναι 1,2% και 2,6%, όπως φαίνεται στο Σχήμα 1.30δ′. Οι μικρές αυτές επιβαρύνσεις κινητοποιούν την πρόβλεψη της κλιμάκωσης εκτός σειράς στο υλικό, για την εύρεση των βέλτιστων διαμορφώσεων κατά την εκτέλεση.

Πρόβλεψη κλιμάκωσης εκτός σειράς

Δοκιμάστηκαν πολλαπλά μοντέλα παλινδρόμησης για την πρόβλεψη της κορεσμένης βελτίωσης LOOG (ουσιαστικά ένα μέτρο της κλιμάκωσης OOO) από τις μετρήσεις που συλλέχθηκαν σε όλα τα επίπεδα χαρακτηρισμού των εφαρμογών. Η ρίζα μέσου τετραγωνικού λάθους διασταυρούμενης επικύρωσης για όλα τα μοντέλα που προσαρμόστηκαν απεικονίζονται στο Σχήμα 1.31.

Αρχικά δοκιμάστηκε ένα μοντέλο γραμμικής παλινδρόμησης, με τα σημαντικότερα χαρακτηριστικά του να απεικονίζονται στο Σχήμα 1.32. Τα χαρακτηριστικά με αρνητικούς συντελεστές είναι χρωματισμένα με κόκκινο χρώμα. Όπως συζητήθηκε εκτενώς στην ενότητα 5.3, τα στατιστικά στοιχεία χρόνου εκτέλεσης που σχετίζονται με προσπελάσεις μνήμης (Global accesses, MEM dispatch stalls, L1 Data pending hits -απαιτήσεις για τρέχοντα misses-) συσχετίζονται αρνητικά με την επιτάχυνση σε διαμορφώσεις LOOG, ενώ εκείνα που σηματοδοτούν υψηλή απόδοση εντολών συσχετίζονται θετικά.

Το σύνολο εκπαίδευσης περιλάμβανε και τους 42 CUDA kernels μονής εκκίνησης καθώς και 28 CUDA kernels πολλαπλών εκκινήσεων και το σύνολο δοκιμής περιλάμβανε τους υπόλοιπους 40 CUDA kernels πολλαπλών εκκινήσεων. Η προσαρμογή πραγματοποιήθηκε με βάση τα αθροιστικά στατιστικά στοιχεία ολόκληρου του CUDA kernel (μέσος όρος για όλες τις εκκινήσεις για τους CUDA kernels πολλαπλών εκκινήσεων). Για να αποφευχθεί η υπερβολική προσαρμογή και λόγω του απαγορευτικού μεγέθους των δεδομένων εκπαίδευσης για διασταυρούμενη επικύρωση, το μοντέλο που επιλέχθηκε παρείχε RMSE εντός 5% της διαμέσου των 100 προσαρμογών (τελικό RMSE 0,12 και MAPE 27%).

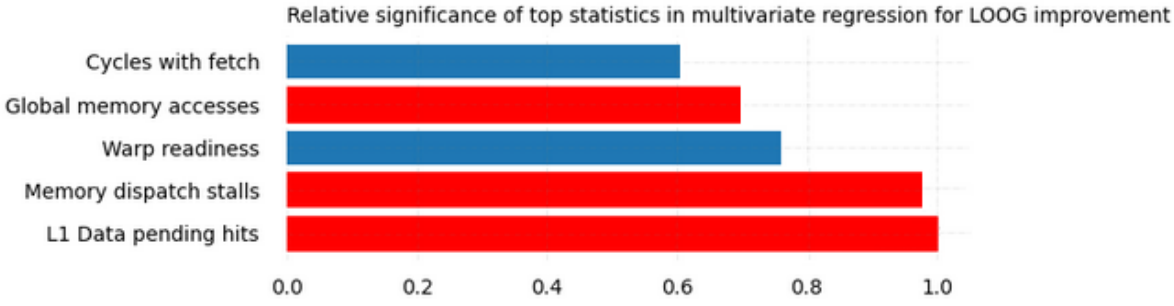


Σχήμα 1.31: Αποτελέσματα RMSE διασταυρούμενης επικύρωσης για όλους τους regressors που προσαρμόστηκαν στα δεδομένα

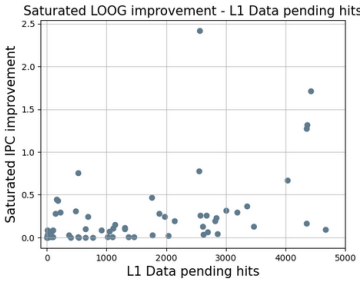
Η συσχέτιση της μεταβλητής-στόχου με το σημαντικότερο χαρακτηριστικό του Regressor, που φαίνεται στο Σχήμα 1.33 δεν είναι γραμμική.

Πρόβλεψη επίδοσης στις ενδιάμεσες διαμορφώσεις

Προσπαθούμε να προβλέψουμε τη βελτίωση των επιδόσεων στις ενδάμεσες διαμορφώσεις δεδομένης της κορεσμένης βελτίωσης IPC και της μέσης κανονικοποιημένης καμπύλης βελτίωσης IPC (κουκίδες που απεικονίζονται στο Σχήμα 1.34α').



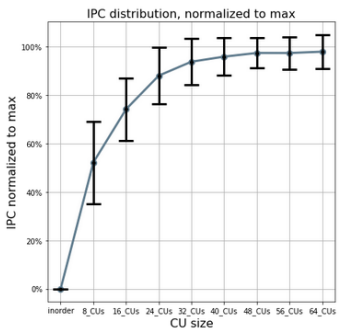
Σχήμα 1.32: Τα πιο σημαντικά χαρακτηριστικά για την πολυμεταβλητή γραμμική παλινδρόμηση που προσαρμόζεται στα δεδομένα



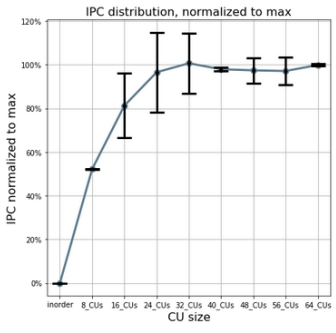
Σχήμα 1.33: Συσχέτιση L1 Data pending hits με κορεσμένη βελτίωση επίδοσης

Η βελτίωση του IPC στη διαμόρφωση 64 CU έχει οριστεί στη γνωστή τιμή κορεσμού. Οι ενδιάμεσες τιμές υπολογίζονται πολλαπλασιάζοντας τη μέση κανονικοποιημένη βελτίωση IPC σε κάθε διαμόρφωση με την τιμή κορεσμού. Τα αποτελέσματα απεικονίζονται στο Σχήμα 1.34γ΄.

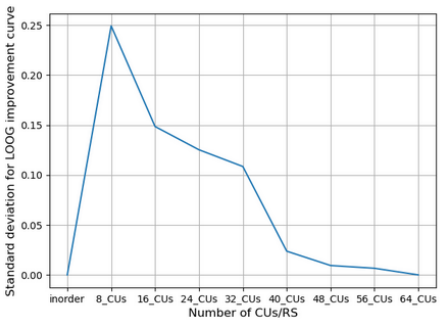
Οι επιταχύνσεις σε ενδιάμεσες διαμορφώσεις υπολογίζονται όπως περιγράφηκε προηγουμένως, χρησιμοποιώντας τη μέση καμπύλη επιτάχυνσης στο εύρος [8_CUs, 48_CUs]. Τα αποτελέσματα απεικονίζονται στο Σχήμα 1.34β΄.



(α΄) Καμπύλη βελτίωσης IPC με την κλιμάκωση του LOOG



(β΄) Τυπική απόκλιση των τιμών IPC που προβλέπονται από τον ελεγκτή σε όλες τις διαμορφώσεις CU



(γ΄) Τυπική απόκλιση για τις τελικές προβλεπόμενες τιμές

Σχεδιασμός ελεγκτή αναδιαμόρφωσης στο υλικό

Τα δέντρα αποφάσεων για την πρόβλεψη ελάχιστης βελτίωσης του IPC LOOG και την πρόβλεψη κορεσμένης βελτίωσης του IPC απεικονίζονται στα σχήματα 1.35α΄ και 1.35. Οι τιμές εξόδου για κάθε

δέντρο απόφασης του ελεγκτή αναδιαμόρφωσης παρουσιάζονται στον πίνακα 1.12. Ο σχεδιασμός του ελεγκτή αναδιαμόρφωσης υλικού απεικονίζεται στο Σχήμα 1.36.

Predictor	RMSE	MAPE	Leaf values								Test set mean	Test set 90th
8_CU regressor	0.08	21%	0.08	0.21	0.24	0.71	1.19	1.85	3.96	-	0.18	0.57
64_CU regressor	0.12	27%	0.23	0.49	0.51	0.87	1.33	1.81	2.45	4.1	0.61	1.52

1.12: Σφάλματα και προβλεπόμενες τιμές IPC για τους regressors δέντρων απόφασης

1.4 Αξιολόγηση της αναδιαμορφώσιμης αρχιτεκτονικής

1.4.1 Διαστασιολόγηση

Στο Σχήμα 1.38α′ παρέχεται η μετρική ADP, κανονικοποιημένη ως προς τη βασική μικροαρχιτεκτονική, για όλες τις εξεταζόμενες μικροαρχιτεκτονικές και με τη χρήση των μετρικών αναδιαμόρφωσης PDP και EDP. Ελαχιστοποιείται στην πιο χαμηλής κλίμακας μικροαρχιτεκτονική των 16 CUs με τις 32 CUs να ακολουθούν αμέσως μετά. Στα σχήματα 1.38γ′, 1.38 και 1.38β′, είναι προφανές ότι όλες οι άλλες μετρικές βελτιστοποιούνται στην πιο κλιμακούμενη διαμόρφωση κλίμακας, 48 CU. Ωστόσο, η πιο σημαντική πτώση συμβαίνει από 16 σε 32 CUs, με ελάχιστη βελτίωση από 32 σε 48.

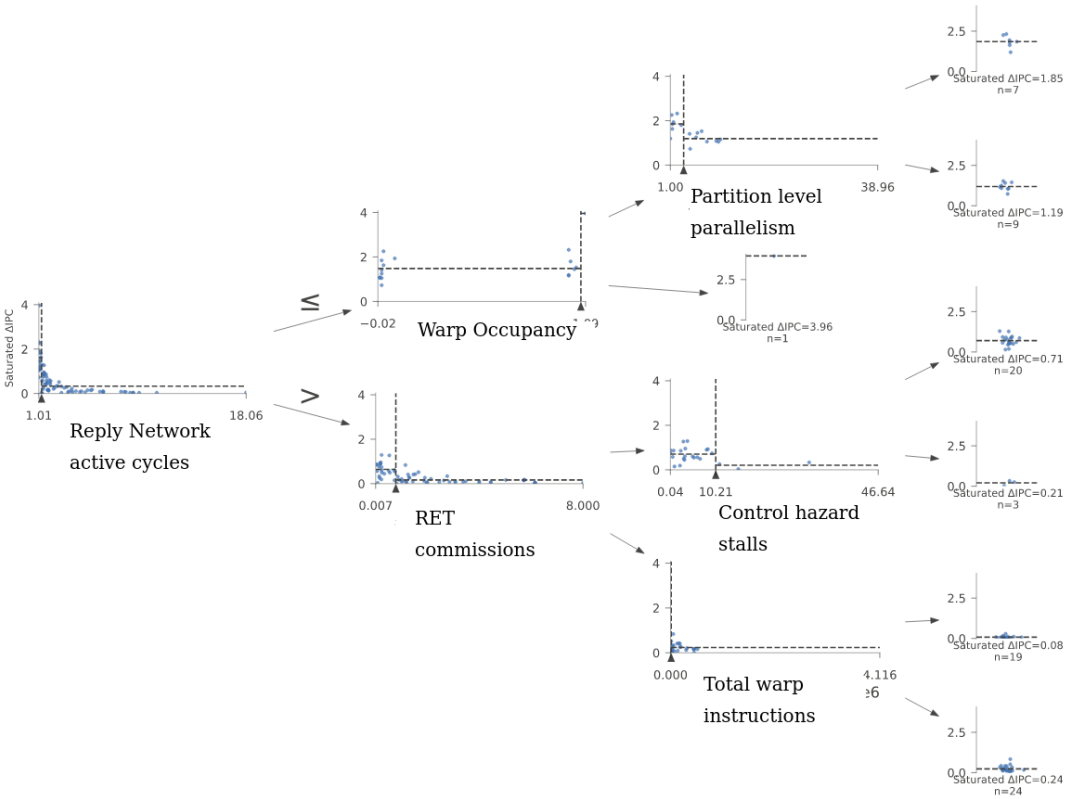
Οι μέσες τιμές των βέλτιστων αποτελεσμάτων αναδιαμόρφωσης για μια δεδομένη αρχιτεκτονική ανά μετρική αναδιαμόρφωσης συνοψίζονται στον πίνακα 1.37α , κανονικοποιημένες ως προς τη βασική μικροαρχιτεκτονική (π.χ. Μια μικροαρχιτεκτονική με 16 CUs, παράγει μια μέση καθυστέρηση 0,749 κανονικοποιημένη ως προς τη βασική μικροαρχιτεκτονική όταν αναδιαμορφώνεται βέλτιστα με μια μετρική PDP). Όπως φαίνεται επίσης στον πίνακα, όλες οι μετρικές εκτός από την ADP ελαχιστοποιούνται για μέγιστες CUs. Για όλα τα μεγέθη αξίας, η διαμόρφωση 32 CU παρέχει τιμές που προσεγγίζουν τη βέλτιστη. Όπως φαίνεται στον πίνακα 1.37β, για όλα τα μεγέθη αξίας που χρησιμοποιήθηκαν (Delay, Energy, ADP, PDP) η μικροαρχιτεκτονική 32 CU παρέχει επιβαρύνσεις εντός 2% του βέλτιστου.

Προφανώς, για τον μέσο CUDA kernel, η μικροαρχιτεκτονική 32 CU παρέχει τον πιο λογικό συμβιβασμό μεταξύ της βελτιστοποίησης του ADP και των άλλων σύνθετων μετρήσεων. Όπως φαίνεται στους πίνακες 1.37ς και 1.37β, για τους ευαίσθητους στο LOOG kernels, η τιμή αυτή είναι 48.

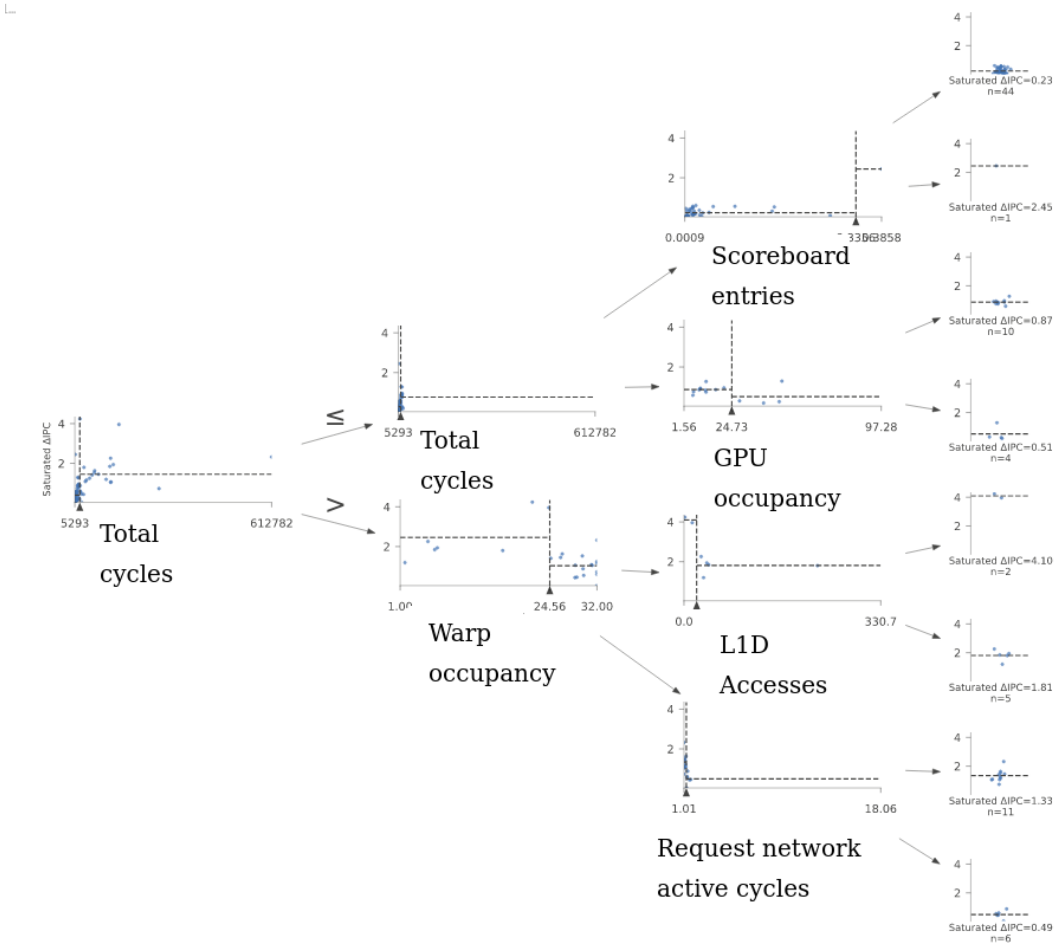
1.4.2 Ελεγκτής στο λογισμικό

Στατική αναδιαμόρφωση

Στο Σχήμα 1.39α′, όσον αφορά τις στατικές (set-in-stone) μικροαρχιτεκτονικές, η αντιστάθμιση ενεργειακής απόδοσης-καθυστέρησης απεικονίζεται για 16 έως 32 CUs, με το EDP να ελαχιστοποιείται στις 32 CUs. Η επίδοση, η ενεργειακή απόδοση και το EDP βελτιστοποιούνται σε 48, 16 και 32 CUs αντίστοιχα. Είναι ενδιαφέρον ότι, ακόμη και για γενικούς CUDA kernels, η αναδιαμόρφωση με οποιαδήποτε μετρική είναι ενεργειακά αποδοτικότερη από οποιαδήποτε στατική διαμόρφωση.

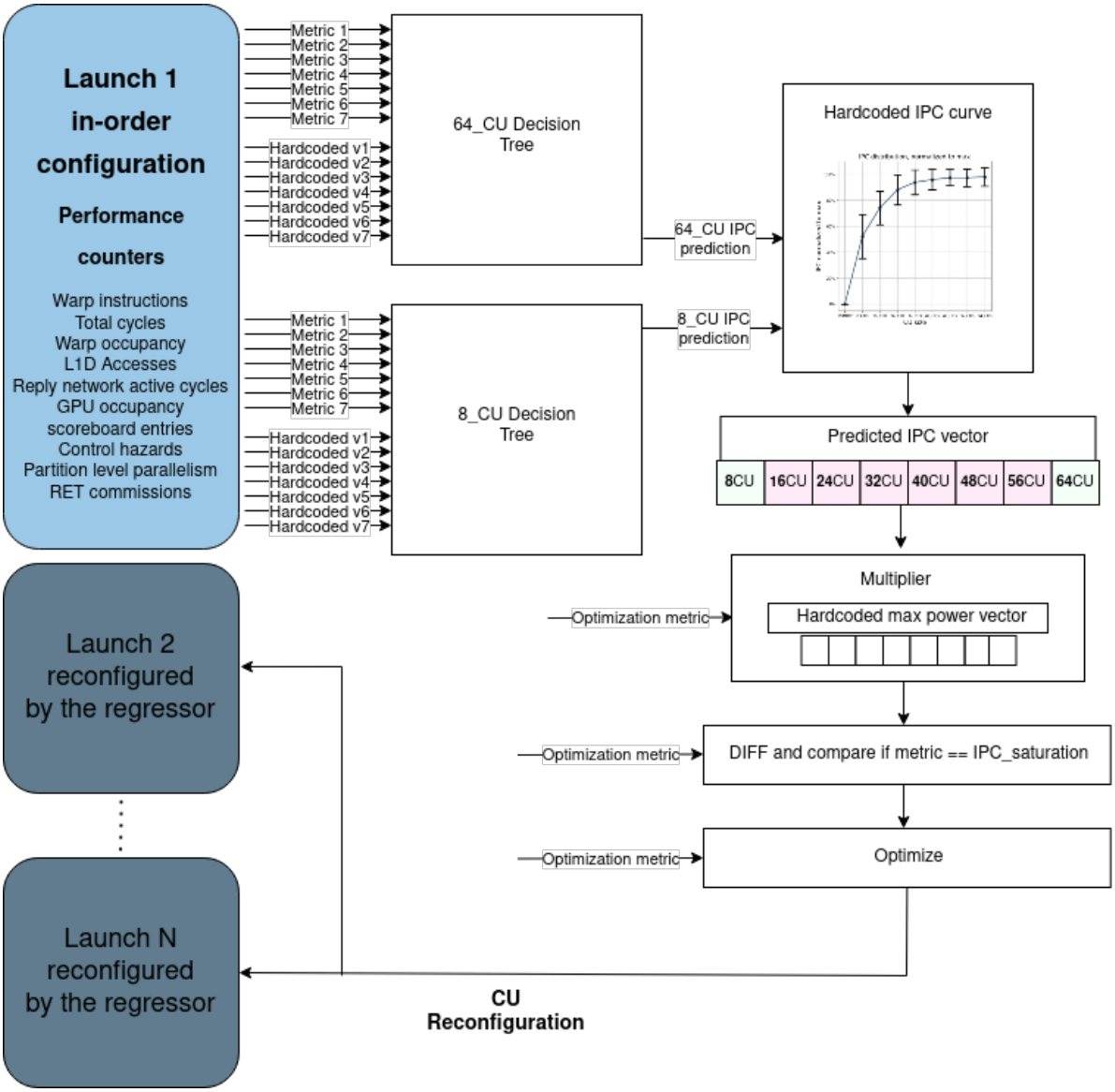


(α′) Πρόβλεψη βελτίωσης IPC στην ελάχιστη διαμόρφωση LOOG



(β′) Πρόβλεψη βελτίωσης IPC στη μέγιστη διαμόρφωση LOOG

Σχήμα 1.35: Προσαρμοσμένα δέντρα απόφασης για την πρόβλεψη της βελτίωσης επίδοσης στο LOOG



Σχήμα 1.36: Σχέδιο του ελεγκτή αναδοιαμόρφωσης στο υλικό

CU	16	32	48
ADP			
PDP	0,800	0,815	0,857
EDP	0,790	0,796	0,833
Delay			
PDP	0,749	0,716	0,710
EDP	0,739	0,699	0,690
Energy			
PDP	0,800	0,780	0,776
EDP	0,803	0,786	0,784
EDP			
PDP	0,599	0,558	0,551
EDP	0,593	0,549	0,541

(Α) Σύγκριση διαφορετικών μικροαρχιτεκτονικών για γενικούς kernels

CU	16	32	48
ADP			
PDP	0	1.87%	7.12%
EDP	0	0.76%	5.44%
Delay			
PDP	5.49%	0.85%	0
EDP	7.1%	1.3%	0
Energy			
PDP	3.09%	0.52%	0
EDP	2.42%	0.26%	0
EDP			
PDP	8.71%	1.27%	0
EDP	9,61%	1,48%	0

(Β) Επιβαρύνσεις σε σύγκριση με το βέλτιστο για γενικούς kernels

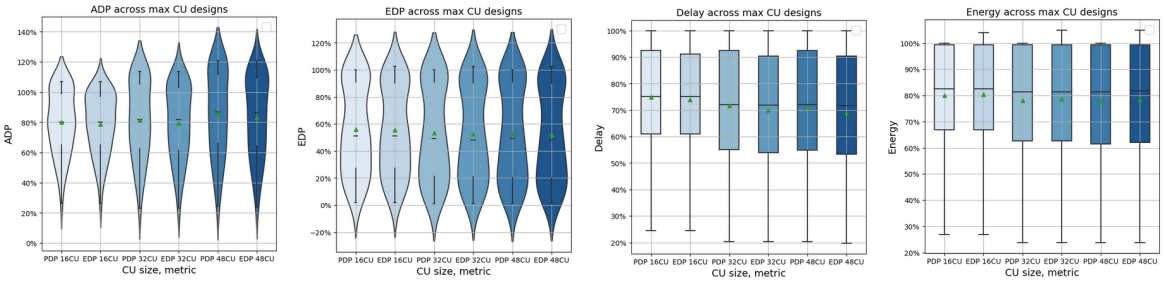
CU	16	32	48
ADP			
PDP	0,582	0,533	0,537
EDP	0,582	0,525	0,531
Delay			
PDP	0,545	0,462	0,445
EDP	0,545	0,462	0,440
Energy			
PDP	0,599	0,536	0,528
EDP	0,599	0,536	0,530
EDP			
PDP	0,346	0,270	0,258
EDP	0,346	0,270	0,256

(C) Σύγκριση διαφορετικών μικροαρχιτεκτονικών για ευαίσθητους στο LOOG kernels

CU	16	32	48
ADP			
PDP	9,13%	0	0,70%
EDP	10,78%	0	1,15%
Delay			
PDP	22,45%	3,79%	0
EDP	23,73%	4,88%	0
Energy			
PDP	13,45%	1,58%	0
EDP	13,03%	1,20%	0
EDP			
PDP	34,21%	4,66%	0
EDP	35,10%	5,36%	0

(D) Επιβαρύνσεις σε σύγκριση με το βέλτιστο για ευαίσθητους στο LOOG kernels

μ 1.37: Σύγκριση αναδιαμορφώσιμων μικροαρχιτεκτονικών διαφορετικού μεγέθους κατά τη βέλτιστη αναδιαμόρφωση με τις μετρικές αναδιαμόρφωσης PDP και EDP



(α′) (β′) (γ′) (δ′)

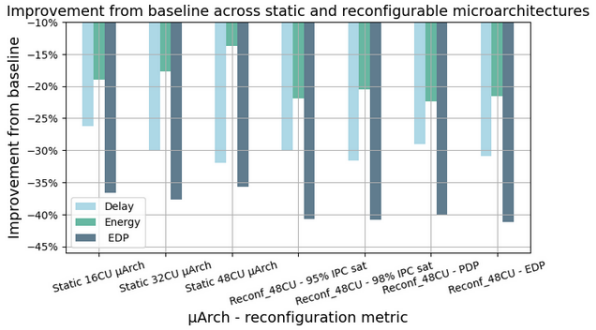
Σχήμα 1.38: Σύγκριση μικροαρχιτεκτονικών διαφορετικής κλιμάκωσης εκτός σειράς

Ημι-δυναμική αναδιαμόρφωση

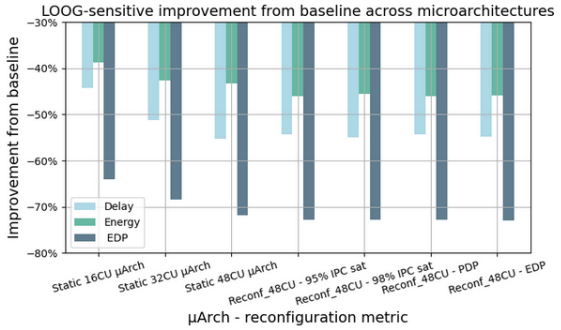
Σε αυτό το υποτμήμα, η επίδραση μιας ημι-δυναμικής αναδιαμόρφωσης εξετάζεται αποκλειστικά για CUDA kernels πολλαπλών εκκινήσεων, σε δύο κατηγορίες kernels, πάνω και κάτω από τον διάμεσο αριθμό εκκινήσεων. Όπως παρουσιάστηκε στην ενότητα 5.8.5, η στατική αναδιαμόρφωση δεν δημιουργεί σημαντικές επιβαρύνσεις σε σύγκριση με την ημιδυναμική αναδιαμόρφωση που φαίνεται στο Σχήμα 1.26. Πιο συγκεκριμένα, όπως φαίνεται στον πίνακα 1.13, παρατηρούνται ελάχιστες μέσες τιμές επιβάρυνσης και για τις δύο κατηγορίες εφαρμογών (σύμφωνα με τον αριθμό των κλήσεων), με την κατανομή να είναι αρκετά ευρεία. Επομένως, περιθωριακές περιπτώσεις χαμηλής συνοχής κλιμάκωσης εντός του ίδιου kernel (χρονική αλλαγή της συμπεριφοράς κλιμάκωσης εκτός σειράς) που παράγουν σημαντικές επιβαρύνσεις παρατηρούνται στους πίνακες 1.15 και 1.14.

Launches	Less than 7 launches		7 or more launches	
	Delay	Energy	Delay	Energy
Min	0	0	0	0
Median	-0.45%	-0.52%	-0.82%	-0.58%
Mean	-0.52%	-0.80%	-1.10%	-0.86%
Max	-2.41%	-2.65%	-4.42%	-4.89%

1.13: Κατανομή βελτίωσης χρόνου εκτέλεσης και ενέργειας από στατική σε ημι-δυναμική αναδιαμόρφωση



(α′) Γενικοί kernels



(β′) kernels ευαίσθητοι στο LOOG

Σχήμα 1.39: Βελτίωση των στατικών μικροαρχιτεκτονικών και των αναδιαμορφώσιμων ανά μετρική από τη βασική μικροαρχιτεκτονική.

Kernel	DEnergy	Description	static configuration	inorder	8_CUs	16_CUs	24_CUs	32_CUs	40_CUs	48_CUs
testSssp	-4,89%	Reg. expand kernel	8_CUs	3	3	0	0	0	0	0
ispass-2009-BFS	-3,26%	BFS	16_CUs	1	0	8	0	0	0	0
shoc-Reduction	-2,85%	Reduce operation	32_CUs	0	0	0	0	3	1	0
shoc-Sort	-1,77%	Scan	16_CUs	2	0	2	0	0	0	0
test-Amr	-1,49%	Reg refine kernel	24_Cus	0	0	1	2	1	0	0

1.14: kernels με τη μεγαλύτερη χειροτέρευση σε ενέργεια από την ημι-δυναμική στη στατική αναδιαμόρφωση

Kernel	DDelay	Description	static configuration	inorder	8_CUs	16_CUs	24_CUs	32_CUs	40_CUs	48_CUs
dwt2d-rodinia-3.1	-4,42%	DWT2D kernel	32_CUs	0	0	0	0	3	4	0
lonestar-sssp-wln	-4,27%	RelaxGraphWorklist	16_CUs	0	0	4	2	1	1	0
cfd-rodinia	-3,87%	Initialization	16_CUs	0	2	0	1	0	0	0
shoc-Spmv	-3,69%	spmv_scalar	48_CUs	0	0	0	0	13	4	82
test-Amr	-2,55%	Reg refine kernel	32_CUs	0	0	0	3	1	0	0

1.15: kernels με τη μεγαλύτερη χειροτέρευση σε χρόνο εκτέλεσης από την ημι-δυναμική στη στατική αναδιαμόρφωση

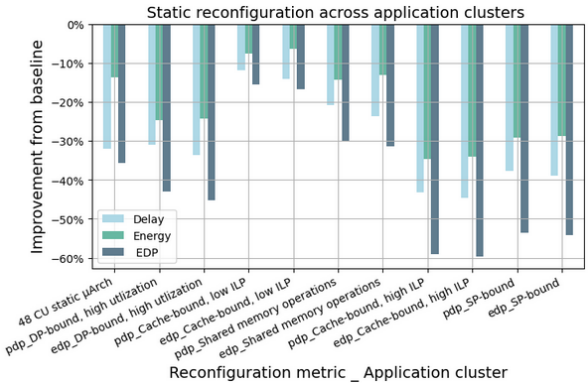
Συσχέτιση με συστάδες εφαρμογών

Στο Σχήμα 1.40α′, το βέλτιστο σχήμα στατικής αναδιαμόρφωσης αξιολογείται σε σχέση με τη στατική 48 CU μικροαρχιτεκτονική στις συστάδες εφαρμογών που ορίζονται στην ενότητα 5.3 .

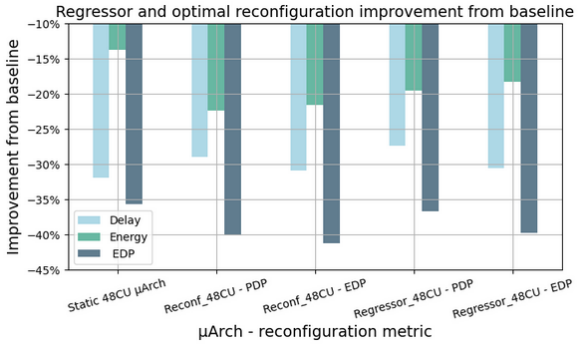
1.4.3 Ελεγκτής στο υλικό

Αναδιαμόρφωση πρώτης εκκίνησης

Τα αποτελέσματα για τον μέσο CUDA kernel συνοψίζονται στον πίνακα 1.16. Σε αυτή την ενότητα, ο ελεγκτής αναδιαμόρφωσης υλικού αξιολογείται σε σχέση με τη βέλτιστη στατική αναδιαμόρφωση που υλοποιείται με λογισμικό, όπως περιγράφεται στην ενότητα 5.8.4, καθώς και με μια στατική (set-in-stone) μικροαρχιτεκτονική με 48 CUs.Οι μετρητές επιδόσεων υλικού συλλέγονται για όλες τις κλήσεις του CUDA kernel, κατά τη διάρκεια μιας εκτέλεσής του και χρησιμοποιούνται για την εξαγωγή συμπερασμάτων σχετικά με μια βέλτιστη διαμόρφωση, η οποία εφαρμόζεται στη συνέχεια σε επόμενες εκτελέσεις του CUDA kernel σε όλες τις εκκινήσεις του. Στο Σχήμα 1.40β′, παρουσιάζεται η προαναφερθείσα σύγκριση. Ο ελεγκτής υλικού είναι λιγότερο αποδοτικός από τον ελεγκτή στατικής αναδιαμόρφωσης λογισμικού σε όλα τα μεγέθη αξίας και τις μετρικές αναδιαμόρφωσης.



(α′) Αξιολόγηση της αναδιαμορφώσιμης αρχιτεκτονικής ανά συστάδα εφαρμογών



(β′) Αξιολόγηση του ελεγκτή αναδιαμόρφωσης στο υλικό έναντι της βέλτιστης αναδιάμορφωσης και της στατικής μικροαρχιτεκτονικής με 48 CUs

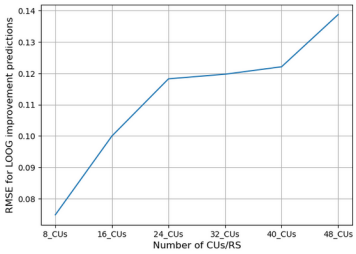
Static 48CU uArch		
Delay	0,680	
Energy	0,863	
EDP	0,643	
Optimal static reconf		
Metric	PDP	EDP
Delay	0,710	0,690
Energy	0,776	0,784
EDP	0,599	0,587
Regressor		
Metric	PDP	EDP
Delay	0,726	0,694
Energy	0,805	0,817
EDP	0,633	0,602

1.16: Ενέργειες και χρόνοι εκτέλεσης για τα σχήματα αναδιαμόρφωσης και τη στατική μικροαρχιτεκτονική, κανονικοποιημένα στη βασική.

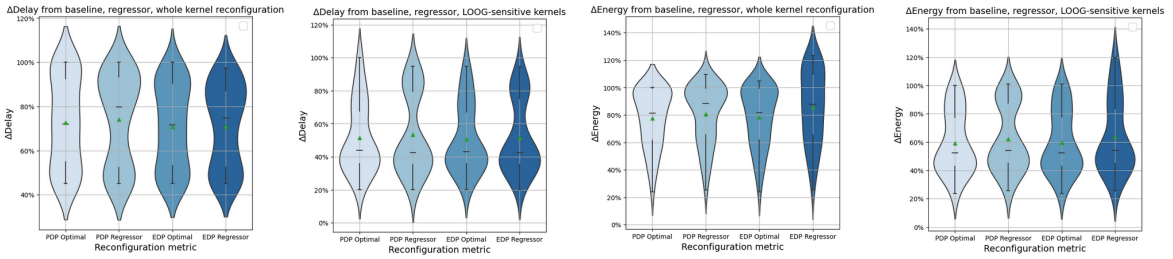
Το RMSE για το προβλεπόμενο κανονικοποιημένο διάνυσμα βελτίωσης του IPC στο σύνολο δοκιμών παρουσιάζεται στο Σχήμα 1.41.

Αξιολόγηση του ελεγκτή σε στατική αναδιαμόρφωση

Σε αυτό το υποτμήμα, ο ελεγκτής αξιολογείται σε στατική (ολόκληροι CUDA kernels) αντί για αναδιαμόρφωση κατά την πρώτη εκκίνηση. Ο σκοπός αυτής της αξιολόγησης είναι να επεκτείνει το περιορισμένο σύνολο δεδομένων πυρήνων πολλαπλών εκκινήσεων ώστε να συμπεριλάβει όλους τους CUDA kernels και να καθορίσει τη συμπεριφορά του ελεγκτή όσον αφορά CUDA kernels ευαίσθητους σε LOOG.



Σχήμα 1.41: RMSE για τις προβλέψεις του ελεγκτή και τις υπολογισμένες ενδιάμεσες τιμές IPC



(α′) Χρόνος εκτέλεσης, γενικοί kernels (β′) Χρόνος εκτέλεσης, ευαίσθητοι στο LOOG kernels (γ′) Ενέργεια, γενικοί kernels (δ′) Ενέργεια, ευαίσθητοι στο LOOG kernels

Σχήμα 1.42: Βελτίωση χρόνου εκτέλεσης και ενέργειας για βέλτιστη στατική αναδιαμόρφωση με τον ελεγκτή

1.5 Συμπεράσματα και μελλοντικές επεκτάσεις

1.5.1 Συμπεράσματα

Με την εφαρμογή του σχήματος εκτέλεσης LOOG στον Accel-Sim, αποκτούμε τη δυνατότητα να αυξήσουμε την ακρίβεια της προσομοίωσης σε πολλά μέτωπα. Αποκτάμε επίσης πρόσβαση σε διαμορφώσεις συντονισμένες με ακρίβεια για GPU κέντρων δεδομένων, συμπεριλαμβανομένης της Quadro GV100 που χρησιμοποιήθηκε στην ανάλυσή μας.

Έχοντας συλλέξει στατιστικά στοιχεία χρόνου εκτέλεσης της μικροαρχιτεκτονικής σε 7 σύνολα μετροπρογραμμάτων και 100 CUDA kernels, τα χρησιμοποιούμε για να χαρακτηρίσουμε τις εφαρμογές σε ομάδες όσον αφορά τα σημεία συμφόρησης της αρχιτεκτονικής, καθώς και για να συσχετίσουμε συγκεκριμένα χαρακτηριστικά που διαθέτουν με την επιτάχυνση και την υψηλή χρήση της GPU σε κλιμακωτές διαμορφώσεις LOOG. Μια επιλεγμένη κατηγορία εφαρμογών που κερδίζουν σημαντική επιτάχυνση σε τέτοιες διαμορφώσεις ονοματίζεται "LOOG-ευαίσθητη¨. Αυτή η ανάλυση παρέχει τη ραχοκοκαλιά για τον περαιτέρω χαρακτηρισμό του φόρτου εργασίας σε όλη τη διάρκεια αυτής της εργασίας.

Τα στοιχεία του front-end του αγωγού διαστασιολογούνται και διαμορφώνονται βέλτιστα σε συνδυασμό με το LOOG, οδηγώντας στα συμπεράσματα ότι το εύρος ζώνης του αποκωδικοποιητή μπορεί να περιοριστεί (-4,57% Power, -0,22% Area) και ότι ο χρονοπρογραμματισμός Depth-First instruction Issue παρέχει βέλτιστη απόδοση (επιτάχυνση 1,14 για CUDA kernels ευαίσθητους στο LOOG). Το τελευταίο, σε συνδυασμό με τη συνειδητοποίηση ότι το εκμεταλλεύσιμο ILP ποικίλλει ευρέως μεταξύ των warps, οδηγεί στην υλοποίηση ενός ελεγκτή αναδιαμόρφωσης διαμερισμού του

ρυθμιστικού διαύλου εντολών που παρέχει μέση αύξηση της απόδοσης κατά 4,4% για CUDA kernels ευαίσθητους στο LOOG, έως και 10,2%.

Οι δομές που σχετίζονται με το LOOG (Collector Units, Register Renaming Stack και Instruction Buffer) μελετώνται όσον αφορά την κλιμάκωση, οδηγώντας στο συμπέρασμα ότι οι Collector Units είναι το κύριο στοιχείο που οδηγεί στην αύξηση της ταχύτητας, αλλά είναι επίσης απαιτητικές σε ισχύ και επιφάνεια. Η δυνατότητα να επωφεληθεί κανείς από τη σημαντική επιτάχυνση που επιτυγχάνεται σε διαμορφώσεις LOOG με κλιμάκωση όσον αφορά CUDA kernels ευαίσθητους στο LOOG, διατηρώντας παράλληλα την ενεργειακή απόδοση όταν είναι απαραίτητο, παρέχεται από μια κλιμακούμενη αναδιαμορφώσιμη μικροαρχιτεκτονική με λεπτομερή αποκοπή ισχύος στους Σταθμούς Συλλογής (Collector Units). Η αναδιαμορφώσιμη αρχιτεκτονική αξιολογείται αρχικά με βάση έναν βέλτιστο ελεγκτή λογισμικού που εκτελεί ημι-δυναμική επαναδιαμόρφωση σε επίπεδο ανά εκκίνηση. Συνειδητοποιώντας ότι η ταυτοποίηση (profiling) κατά την πρώτη εκκίνηση του CUDA kernel είναι επαρκής για την αναδιαμόρφωση σε επόμενες κλήσεις, υλοποιούμε έναν ελεγκτή αναδιαμόρφωσης υλικού που χρησιμοποιεί μετρητές επιδόσεων κατά τη διάρκεια εκτέλεσης, βασισμένους σε δέντρα απόφασης.

Μια στατική διαμόρφωση LOOG με κλιμάκωση παρέχει επιτάχυνση 1,48 για γενικούς CUDA kernels και 13,7 % μείωση στην απώλεια ενέργειας, σε σύγκριση με τη βασική αρχιτεκτονική. Η αναδιαμόρφωση με οδηγίες λογισμικού και η χρήση του ελεγκτή υλικού μπορούν να παρέχουν την ίδια επιτάχυνση όταν χρειάζεται και έχουν τη δυνατότητα να βελτιώσουν την ενεργειακή απόδοση σε σχέση με τη βασική μικροαρχιτεκτονική κατά 22,4% και 19,5% αντίστοιχα.

1.5.2 Μελλοντικές επεκτάσεις

- Μικρή αναδιαμόρφωση του πηγαίου κώδικα του Accel-Sim, όπου χρειάζεται (Παράρτημα A.1), καθώς και κατάλληλες εγκαταστάσεις για την αξιοποίηση της λειτουργικότητας προσομοίωσης με βάση το ίχνος (mISA) που παρέχει το Accel-Sim είναι η άμεση προτεραιότητά μας για την αύξηση της ακρίβειας της προσομοίωσης επίδοσης.

- Μοντελοποίηση ισχύος με ακρίβεια κύκλου, με την εισαγωγή μετρητών επιδόσεων κατά τη διάρκεια εκτέλεσης σχετικών με το LOOG από το μοντέλο επιδόσεων (όπως λειτουργεί το Accelwattch για το βασικό μοντέλο) είναι απαραίτητη για να διαπιστωθεί η ποσοτική αξιοπιστία των αποτελεσμάτων μας.

- Για την επιβεβαίωση των παραδοχών μας απαιτείται μια ακριβέστερη αξιολόγηση της επιφάνειας των τρανζίστορ ύπνου (sleep transistors) της αποκοπής ισχύος, της ενέργειας και του χρόνου αφύπνισης. Λόγω των χρησιμοποιούμενων σχημάτων αναδιαμόρφωσης, η Ενέργεια και η καθυστέρηση αφύπνισης είναι ανταλλάξιμες και έχουν μεγάλο περιθώριο σφάλματος, αλλά το Εμβαδόν είναι ζωτικής σημασίας.

- Με μεγάλο βαθμό βεβαιότητας, οι παλινδρομήσεις του ελεγκτή αναδιαμόρφωσης υλικού μπορούν να προσαρμοστούν σε διαμορφώσεις LOOG ενδιάμεσης κλίμακας, ώστε να αποφευχθεί η επιβάρυνση διαμόρφωσης της πρώτης εκκίνησης σε σειρά.

- Μπορεί να πραγματοποιηθεί μια πιο σχολαστική μελέτη της ετερογένειας μεταξύ των warps όσον αφορά την εκμεταλλεύσιμη ILP και να δοκιμαστούν νέες πολιτικές ή μετρικές αναδιαμόρφωσης του ρυθμιστή εντολών. Δεδομένου ότι η αναδιαμόρφωση του Ibuffer δεν αποτελούσε την κύρια εστίαση αυτής της εργασίας, εικάζουμε ότι τα σχετικά αποτελέσματα είναι υποβέλτιστα.

Chapter 2

Introduction

2.1 The modern hardware accelerator landscape

2.1.1 The end of the scaling laws

There have almost been two decades since the start of what is commonly known as "the breakdown of Moore's Law" (which states that transistors on a chip approximately double every 18 months) and the concomitant cease of the exponential growth in the computing power in single-core systems. Up to that point, the semiconductor industry has enjoyed the benefits of Dennard's scaling, another transistor "scaling law" which states that chip power density stays constant with transistor scaling-down, thus enabling the design of more, faster and more energy-efficient transistors and justifying the cost required to develop new process nodes. This, coupled with advances in device technology, microarchitecture and compilers, had made the single-core path an affordable solution to the ever increasing demands in software [50].

As Dennard's scaling came to a halt due to supply voltage limits as well as increased transistor manufacturing complexity at atomic scale technology nodes, power densities rapidly increased in contemporary chips [51]. The growing transistor count that still follows an exponential trend owing to Moore's law is not accompanied by an increased power budget, leading to the advent of "The Dark Silicon Era" [50], where more transistors occupy an Integrated Circuit than can be turned on at any given time. A reevaluation of power overheads, subject to critical thermal limits and area overheads, being associated with increased manufacturing costs and data communication overheads is necessary [10, 52].

To overcome this obstacle for applications that rely on the traditionally ever-increasing computing performance such as Artificial Intelligence and Data Analytics, meticulous software optimization and fine tuning are required as well as advanced microarchitectures that leverage software diversity with hardware heterogeneity or can even dynamically reconfigure to suit applications' specific characteristics at runtime [14].

2.1.2 High Performance Computing

High-performance computing (HPC) involves utilizing parallel computing technology, where multiple powerful processors collaborate to handle enormous multi-dimensional datasets, also known as big data, and solve complex problems at incredibly high speeds, that would be impossible to tackle with traditional computing resources [53].

HPC systems operate at speeds over a million times faster than traditional desktop, laptop, or server systems. While supercomputers were once the preferred HPC system architecture, many organizations now use HPC solutions on clusters and grids (promoted by the use of a collapsed network backbone, which is easier to troubleshoot and upgrade) of high-speed computer servers hosted either on-premises or in the cloud [54, 55].

HPC has recently seen remarkable progress in both the hardware and software optimization domains. It has become essential in various fields, including physics, chemistry, biology, engineering, and finance. HPC integrates system administration, network, and security knowledge with parallel programming into a multidisciplinary field that combines digital electronics, computer architecture, system software, programming languages, algorithms, and computational techniques [55].

2.1.3 General-Purpose GPU

The GPU (Graphics Processing Unit) is a specialized hardware accelerator originally designed for graphics rendering applications. In the early 2000s, researchers started experimenting with using GPUs for non-graphics computing tasks, typically handled by the CPU, such as scientific simulations, cryptography, and machine learning [1].

Given the halt met by the scaling laws, hardware specialization and the turn to more efficient hardware architectures emerged as an alternative source of speedup.

The adoption of GPU usage for such applications lead to significant gains in performance and energy efficiency, with sequential operations being handled by the CPU and compute-intensive, massively parallel portions of the application being offloaded to the GPU. A challenge for computer architects is balancing the need for efficiency with the need for flexibility to support a range of programs. GPUs are attractive because they support a Turing Complete programming model, making them flexible for a wide range of applications. Hence, they can be an order of magnitude more efficient than CPUs when software is optimized to make full use of the hardware [56, 2].

Initially, these general-purpose applications required developers to write low-level code in specialized languages, to take advantage of the GPU's parallel architecture. However, as the demand for GPGPUs grew, companies like NVIDIA and AMD started developing GPGPU platforms that allowed developers to program the GPU using high-level languages such as C++ or Python, with the inclusion of compiler directives and enabling the use of heterogeneous platform frameworks (such as OpenCL).

Nowadays, computing systems have made a rapid transition towards parallel multi-core architectures. Thus, the accelerator space has been more than ever dominated by GPUs, owing to the aforementioned performance improvement they provide. This has facilitated their widespread adoption in many application domains. Enterprise and hyperscale data-centers are increasingly being built around workloads using Artificial Intelligence (AI) and computationally intensive Deep Neural Networks (DNNs) with massive amounts of data and exploitable parallelism. Striving for the described performance gains and energy efficiency, they employ GPU servers with specialized GPUs arranged in clusters and scrupulously designed network and cooling system architectures [3].

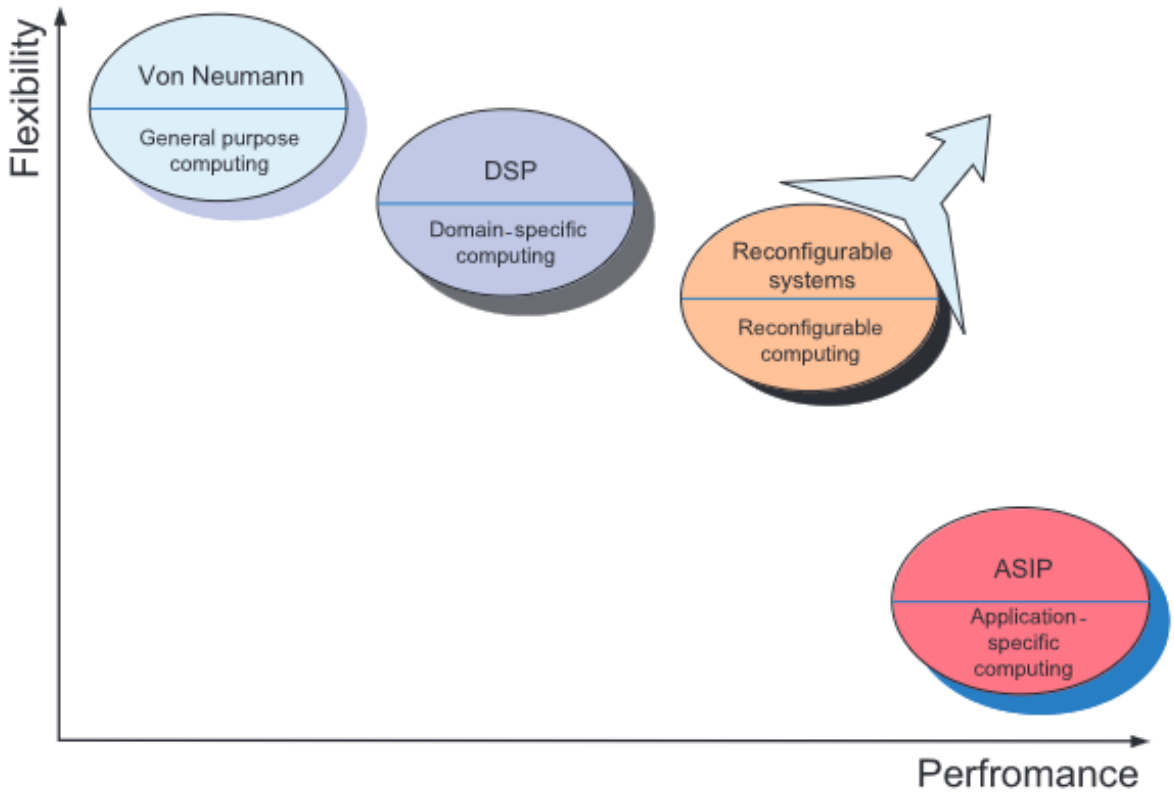
The history of GPUs designed for general-purpose computing began with Nvidia's GeForce 3, which was the first GPU to have programmable shaders. Initially, this technology was used to create more realistic 3D graphics by allowing for 3D transform, bump mapping, specular mapping, and lighting computations. Later, ATI's 9700 GPU became the first card that was capable of DirectX 9 and offered some programming flexibility similar to CPUs. With the release of Windows Vista and DirectX 10, unified shader cores were included as a part of the standard, which made it possible for GPUs to perform general-purpose computations.

Despite being originally developed to speed up rasterized 3D graphics, GPUs have surpassed CPUs in terms of performance for raytraced pre-rendered graphics. Although real-time demonstrations of raytracing are not yet seen in games, the advances in GPGPUs show that computer graphics may be capable of rendering intensive geometry and lighting similar to those in 3D movies in the near future [2].

2.2 Reconfigurable and heterogeneous architectures

Hardware specialization resulting from the halt of the scaling laws and the consequent saturation of single-thread performance has also given rise to reconfigurable and hetereogeneous architectures, providing flexibility to tune the architecture to the requirements of the application at hand,thus maximizing performance and energy efficiency, by leveraging software diversity. In [4], a distinction between Von Neumann (VN) and Application-Specific Processors (ASIP) is made to categorize any architectures in between according to flexibility and performance.

VN computers are highly flexible but lack performance as their general-purpose design is not adapted to any particular application domain. ASIPs offer high performance as they are optimized for a specific sets of applications and can adapt the hardware to them. Between these two extremes, lie a large number of processors with varying degrees of performance and flexibility, as seen in Figure 2.1a. For applications with a wide range of uses, a General-Purpose Processor (GPP) is suitable, while designing a new ASIP optimized for a specific application is best for embedded systems. Reconfigurable computing aims to combine the



(A) Flexibility and performance of processor classes [4]

flexibility of GPPs with the performance of ASIPs in a single device that can adapt to different applications on the fly. This is achieved by changing the structure of the hardware at compile-time or run-time, usually by downloading a bitstream into the device. The Field Programmable Gate Array (FPGA) is currently the most widely used reconfigurable device [4].

Traditional set-in stone heterogeneous architectures like Asymmetric chip Multiprocessors (Asymmetric CMPs), falling between DSPs and Reconfigurable systems in Figure 2.1a have handled this diversity to an extent, typically by using thread migration between cores of different characteristics to adapt to the workload currently executing. The most prominent limitation of such architectures is the immutable design per core, that limits the flexibility required by definition. Fine-grain reconfigurable fabrics implementing "soft cores" (digital circuits representing functional paradigms) [34] and Coarse Grain Reconfigurable Architectures (with larger functional units and coarser-grained interconnects) are an established alternative [5, 4]. However, they share limitations such as reconfiguration overhead, limited support for control-intensive applications and limited resources, hindering the implementation of more complex designs [6].

Reconfigurable chip-multiprocessor architectures [7] have been proposed to tackle this issue, through dynamic scale-up or scale-out reconfiguration by using core fusion and split operations respectively on a multi-core substrate. Thus, multiple independent small in-order

cores that provide high throughput when executing multi-threaded programs can coalesce to form fewer large superscalar Out-of-Order cores when executing single-threaded programs with exploitable Instruction Level Parallelism. Latencies introduced in the pipeline stages of fused cores as well as cache data migration overheads motivated the design of scalable, partially reconfigurable architectures, such as MorphCore [8].
These architectures use large, OOO cores as their base substrate, optimized for single-threaded sequential code and leveraging ILP, paired with the capability to switch to a highly-threaded in-order SMT execution scheme in the presence of Thread-Level-Parallelism. Other scalable architectures such as Dynamic Core boosting [9] , Elastic Core [10] and Flicker [11] , adapt to application performance during runtime, on a core, microarchitecture component and pipeline lane granularity respectively, tailoring the architecture to executing workloads by either dynamic scalability or heterogeneity. As opposed to multi-core scale-up/scale-out techniques used in previously mentioned architectures, these scalable architectures used clock or power gating at various levels as well as Dynamic Voltage and Frequency Scaling. The latter is used to boost performance in critical bottleneck components or individual cores and the former is used to minimize their power overhead in the greatest possible degree [8].

2.3 Light-Weight Out-of-Order GPU (LOOG) execution scheme

GPU architectures leverage massive Thread Level Parallelism to achieve high computational throughput, paired with fast context switching between large groups of threads (warps), in a similar fashion to simultaneous multithreading in CPUs. When long latency operations stall the pipeline, available warps are selected from a large pool to occupy it with computation. As observed in the relevant paper, given the rise of General Purpose GPU computing, a large group of kernels (functions that are offloaded to and accelerated by the GPU) do not sufficiently benefit from the GPU resources, as expressed from frequent stalling and suboptimal hardware utilization, due to their limited data-level parallelism. These kernels bring insufficiencies of the underlying hardware to the forefront and call for exploitaiton of their inherent Instruction-Level Parallelism with Out-of-Order (OOO) execution [12, 13].

Conventional GPU architectures always issue instructions in-order. A scoreboard is used to make sure RAW and WAW data hazards are avoided, paired with a mechanism to safeguard again WAR hazards. In the recent Volta architecture (2017-2018) examined in this thesis, instructions from the same warp can be issued in the same cycle for a small window of 2 and can be generally dispatched and executed Out-of-Order save for register name dependencies. This virtually sets the limit of the industry regarding OOO execution on the GPU.

A Light-Weight Out-of-Order GPU (LOOG) [12, 13, 14] execution scheme has been proposed, exploiting ILP by implementing instruction reordering. The main modifications to the microarchitecture include repurposing Operand Collector Units (reserved by instructions until their operands are collected from the Register File) to serve as the Reservation Stations

in the Tomasulo [15] algorithm, adding a Register Alias Table (RAT) to address name dependencies, and adding Load and Store instruction reordering to settle address dependencies. Scaling-up LOOG to exploit deeper Instruction Level Parallelism, therefore translates to increasing the Instruction Window Length and the number of Collector Units (Reservation Stations, in essence), with the latter being far more significant in the microarchitecture we studied, as will be demonstrated. Due to power overheads associated with increasing CUs and given the wide diversity in performance increase of applications under LOOG, we speculate on dynamically reconfiguring LOOG to match workload behavior and maximize performance and energy efficiency [12, 13, 14].

2.4 Proposal Overview

Our thorough study of modern reconfigurable computing and our simulation-aided analysis of the diverse characteristics of GPU workloads that can be accomodated by it are both driven by the aforementioned observations that motivated the conception and implementation of LOOG. From early on, said observations sparked our interest about both exploring specific workload attributes that are not sufficiently addressed by the current architectures, as well as how ILP specifically can be optimally addressed, by right-sizing the applied modifications on these architectures and tailoring them to the workload at hand.

LOOG is implemented in the new version of GPGPU-sim (4.1.0), granting access to Accel-Sim and the concomitant increase in simulation accuracy on many fronts. Access to the Accel-sim-provided microbenchmarks-tuned configuration of the workstation GPU, Quadro GV100, powered by the Volta architecture, allows us to explore the acceleration of workloads under LOOG on an HPC-relevant substrate. Having accommodated LOOG in this architecture, we collect runtime statistics on 100 kernels.

We proceed to thoroughly study the diverse characteristics of these workloads, categorize them into 5 classes regarding their architectural bottleneck resources and correlate them with improvement on LOOG. A "LOOG-sensitive" class of applications emerges, comprising kernels that are accelerated by more than 100% in the most scale-up LOOG configurations. Having optimally right-sized and configured components that are not directly relevant to it, we theorize and evaluate a scalable OOO reconfigurable architecture with fine-grain power gating, reconfiguring on a per-kernel-launch temporal granularity, to either optimize measures of performance or energy efficiency.
Component optimization is done on the front-end of the architecture (Fetch to Issue stage), regarding Fetch-Decode throughput, instruction Issue scheduling depth and Instruction buffer runtime reconfiguration.

Evaluation of the OOO reconfigurable architecture is initially done on the basis of a theoretical software implementation that assumes previously gained perfect knowledge on workload behavior on a per-kernel-launch granularity and on all the available configurations.

Thus, we initially assess the maximum attainable improvement in performance and energy efficiency that is provided by the ability to reconfigure. Optimal configurations are respectively determined by either performance or energy figures of merit.

For multiple-launch kernels, we proceed to compare implementations on both a coarse (static) and fine (semi-dynamic) reconfiguration granularity, as well as evaluate an oracle reconfiguration controller that accurately infers the most suitable configuration for the first launch, executing on a configuration that corresponds to the original architecture (in-order configuration) and reconfigures the architecture according to it for the rest of the kernel's launches.

Results show minimal deterioration for both transitions (fine-grain per-launch reconfiguration to coarse-grain per-kernel reconfiguration and inferring the optimal configuration from the whole execution to inferring it from the first launch), which motivates us to implement and evaluate a hardware-level reconfiguration controller that predicts performance improvement on LOOG from the first launch of a kernel, executing on the in-order configuration and assesses whether a scale-up, power hungry out-of-order configuration is justified for its subsequent launches based on performance or energy figures of merit.

When reconfiguring to optimize energy efficiency, the hardware-based reconfiguration controller provides a 27.4% delay improvement and a 19.5% energy improvement from the inorder baseline model on average, across all kernels examined, while the software-based reconfiguration controller provides a 29% and 22.4% improvement respectively. Compared to a static, scale-up LOOG microarchitecture, energy efficiency for the average kernel is optimized by 6.7% and 10.1% by the respective controllers. For LOOG-sensitive kernels, delay and energy efficiency are improved by 54% and 46%, by the software controller.

Finally, we speculate on Execution Unit and Cache size scaling as other potential axes of reconfiguration, similarly defining EXU-bound and Cache-bound kernels.

2.5 Contributions

- Implementation of the complete version of LOOG in the new GPGPU-sim version (4.1.0) granting access to:
  - Access to Accel-sim, providing the potential for accurate trace-based simulation using the trace-generation tool provided
  - New microarchitectures simulated, including Volta, providing configurations for datacenter and workstation GPUs, including the Quadro GV100 simulated in our experiments
  - More accurate microarchiecture configurations for the microarchitectures simulated, produced by the automated microbenchmarks-based Accel-sim configuration tuner.

  - New architectural features, including the Volta architecture sub-core model, memory coalescing unit, unified L1 Data cache and shared memory, Register File Cache and Execution Unit customization.
  - New detailed runtime statistics

- New LOOG-dependent runtime statistics and metrics collected used for the reconfiguration model, namely:
  - Fetch-Decode throughput tracking
  - Warp availability distribution across runtime
  - Total and used RAT entries for each active warp
  - Total readiness per warp

- New runtime frontend reconfiguration controllers implemented (decoder online throttling, online Instruction Buffer partitioning reconfiguration controller).
- LOOG-dependent issue scheduling depth study
- Enhancing Accel-sim job launching and stat collection scripts with per-configuration option design space exploration.
- Workload-aware LOOG right-sizing for the Quadro GV100
- Workloads clustering based on microarchitecture bottlenecks and brief LOOG-improvement correlation study.
- Workload characterization (with a kernel granularity) on the three potential reconfiguration axes - OoO scalability, Cache-bound, Execution Unit-bound -.
- Implementation and evaluation of an OoO reconfiguration controller
- Various LOOG-based reconfiguration schemes:
  - Software-based optimal semi-dynamic reconfiguration (kernel launch granularity)
  - Software-based optimal static reconfiguration (whole kernel granularity)
  - Online profiling reconfiguration based on the first launch

- Speculation on the reconfiguration potential on Cache, Execution Unit scaling axes.

2.6 Thesis structure

The following chapters of thesis are organized as follows:

- In Chapter 3, the taxonomies of parallel and reconfigurable architectures are discussed, to ensure comprehension on relevant Prior Art. The architecture of the GPU is elaborated upon and the LOOG execution scheme is presented. A brief introduction to the simulation framework used in out analysis is performed.
- In Chapter 4, Prior Art is presented regarding workload characterization on various fronts and heterogeneous as well as reconfigurable architectures.
- In Chapter 5 we elaborate on the workload categorization and motivational analysis that lead to our proposed microarchitecture optimizations and reconfigurable architecture. The accommodation of LOOG in the workstation architecture is presented and all types of reconfiguration performed are thoroughly analyzed.
- The right-sizing process of the reconfigurable architecture is presented in Chapter 6, along with evaluation of all the aforementioned types of reconfiguration.
- In Chapter 7, we conclude this thesis and summarize potential avenues for further research and testing.

Chapter 3

Background

3.1 Introduction

In this chapter, we will examine the various types of application parallelism as well as how they can be exploited by the underlying hardware, software or their cooperative design in more detail.
We will provide a brief overview of the research GPU simulators (Accelsim and Accelwattch) on which the proposed architectures were evaluated [57]. Per-generation GPU architectural features will be shortly analyzed, as well as the model used in Accelsim, mentioning various component configuration tradeoffs. The LOOG architecture will be more thoroughly discussed, as it provides the basis for the OOO axis of reconfiguration introduced in Chapter 5 .

3.2 Parallel computing

3.2.1 Fundamentals of parallel computing

Parallelism is defined as the potential to execute multiple tasks or operations simultaneously across multiple processors. It can characterize both applications and the underlying architecture. The concept of parallelism is fundamental to high-performance computing, and is widely encountered in scientific computing, data analytics, and machine learning. Parallelism can be achieved at various levels, including instruction-level parallelism, bit-level-parallelism, thread-level parallelism, and task-level parallelism, with the latter being closely tied to and exploiting the underlying data parallelism [58].

Concurrency is the ability to make progress on multiple tasks or computations simultaneously. Concurrency is closely related to parallelism, but is a more general concept that can include situations where tasks are executed sequentially on a single processor, but interleaved with other tasks, remaining active and making progress simultaneously. Concurrent programming is particularly important in modern computing systems, where multiple tasks are executed concurrently to maximize resource utilization,

nevertheless, in the context of a single algorithm, concurrency usually characterizes the algorithm itself while parallelism characterizes the implementation [59].

Scalability is the ability to increase the number of processors or computing resources to handle larger problems or increase performance. Scalability is a critical concern in parallel computing, as it determines the maximum size of problems that can be solved using parallel methods. Achieving scalability often requires careful design and optimization of parallel algorithms, as well as the use of appropriate parallel programming models and hardware architectures. In terms of the problem itself, scalability analysis refers to the selection of the optimal algorithm-architecture combination under certain constraints (problem size or number of processors) [60].

Load balancing is the distribution of computational load evenly across multiple processors to ensure maximum utilization of resources. Load balancing is essential for achieving high performance in parallel systems, as it ensures that all processors are utilized as efficiently as possible, minimizing idle time and maximizing performance. There are many load balancing techniques, including static load balancing, dynamic load balancing, and work stealing. The concept of Load balancing frequently emerges in heterogeneous and scalable architectures [61].

Synchronization Synchronization is the coordination of activities among different processors to ensure correct execution of the program. Synchronization is necessary when multiple threads or processes access shared resources, to avoid race conditions and other forms of interference. Common synchronization techniques include locks, semaphores, and barriers. It is particularly important to the GPU, as it needs to take place on a coarser than individual thread basis. [62].

Communication Communication refers to the transfer of data and information between different processors or nodes in a parallel system. Communication is essential for coordinating activities among different processors, exchanging data between different tasks, and distributing work across multiple processors. There are many communication models, including message passing, shared memory, and remote procedure calls. Axes of communication pattern categorization include local/global communication (regarding task communication scope), structured/unstructured (where a task and its neighbors either form a regular structure like a tree or a random graph), static/dynamic (entailing constant or changing communication patterns) and synchronous/asynchronous (depending on whether the producer of the data participates in the communication process) [63].

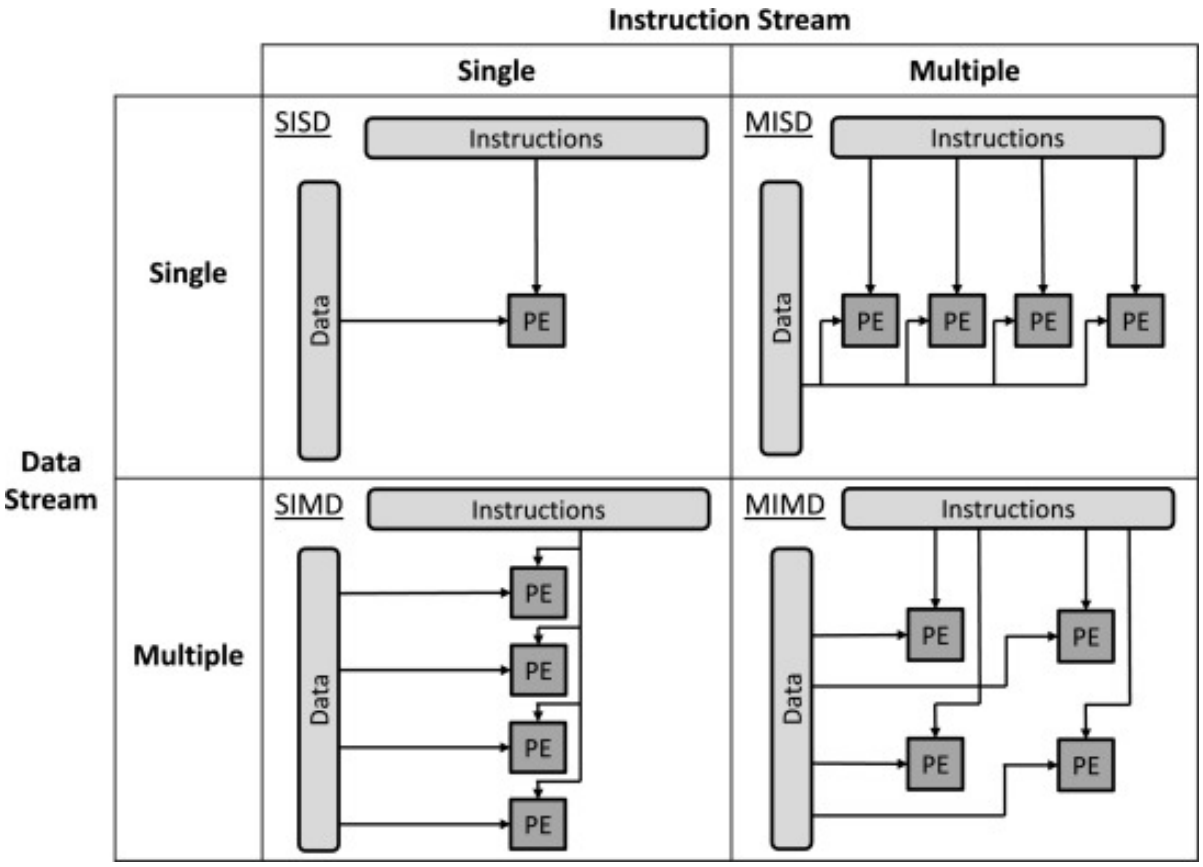


FIGURE 3.1: Flynn's taxonomy [64]

3.2.2 Taxonomy of parallel computing architectures

Constrained by power dissipation and cost, parallelism is the main driving force of contemporary computer design.

- Data-Level Parallelism (DLP) consists in the ability to concurrently operate on multiple data elements.
- Task-Level Parallelism (TLP) consists in the independence of multiple tasks of work, which can largely operate in parallel.

Although these elementary types of parallelism typically characterize applications, their presence can be leveraged in various forms by certain types of architectures, as seen below. Flynn's taxonomy [65] is a classification system proposed by Michael J.Flynn in 1960, used to categorize computer architectures based on the number of instruction streams concurrently operating on the most constrained component of the multiprocessor and the number of data streams being operated on. Although the classification is generally crude, its abbreviations are still used today, as all multiprocessors are hybrids of the models it comprises, as seen in Figure 3.1 [66, 67].

- SISD (Single instruction stream, single data stream) represents the uniprocessor. Instruction Level Parallelism (see below) is the only type of parallelism that can be exploited by this architecture.
- SIMD (Single instruction stream, multiple data stream) involves the execution of the same instruction by a unique instruction stream in multiple processors, each operating in different data items in parallel.
- MISD has been implemented by no commercial multiprocessor to date.
- MIMD comprises the concurrent execution of multiple instruction streams by multiple cores operating on multiple data streams. As an implementation of task-level parallelism, it is more flexible than SIMD, thus more costly. Multi-threading represents its most tightly-coupled form.

[65]

3.2.3 Composite types of parallelism in applications

The elementary types of parallelism mentioned in Section 3.2.2 can form other, composite types that can be directly exploited by the underlying architecture.

- Thread-level parallelism (TLP), generally exploits both data-level parallelism and task-level parallelism in tightly coupled architectures that allow for concurrent execution and communication among multiple threads.
- Data-level parallelism (DLP) is also exploited by vector architectures and GPUs by applying a sequence of instructions to a collection of data in parallel.
- Task-level parallelism, exploits parallelism among tasks which comprise independent transactions and are logically decoupled explicitly by the programmer or the operating system. Such parallelism can be exploited in various levels, from separate processors down to concurrent threads.
- Instruction-level parallelism (ILP), expressing the finest grain of data-level parallelism and task-level parallelism in essence, refers to the overlap in execution of instructions from the same instruction sequence. This can moderately be achieved by compiler optimizations implementing instruction pipelining as well as superscalar execution (multiple execution units per core), out of order execution, usually coupled with register renaming or speculative execution, usually coupled with branch prediction.
- Bit-level parallelism (BLP) is leveraged by a type of parallel computing that involves increasing the size of the processor word. This approach reduces the number of instructions required by the processor to perform an operation on variables larger than the word size [68].

[69]

3.3 GPGPU Programming model

CUDA

CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model developed by NVIDIA for their graphics processing units (GPUs) [16]. It allows developers to leverage the computational power of GPUs for general-purpose computing applications, making it a popular choice for accelerating scientific simulations, machine learning, and data analytics workloads. The CUDA platform includes a software development kit (SDK) that provides a set of programming tools and libraries for developing parallel applications that can be executed on NVIDIA GPUs. These tools and libraries include compilers, debuggers, performance profiling tools and mathematical libraries optimized for GPU processing. The underlying hardware architecture includes a GPU and a concomitant large number of processing cores that can be utilized simultaneously to perform computations in parallel. The programming model includes a set of compiler directives available in C, C++, Fortran and Python that allows developers to write high-level code that can be compiled and executed on the GPU. [18, 17]. The software tools include a debugger, profiler, and performance analysis tool that can be used to optimize the performance of CUDA applications [19].

Offloading computationally intensive workloads to the GPU can result in significant speedup compared to traditional CPU-based computing. Additionally, CUDA provides a flexibility and scalability in its platform for parallel computing, allowing developers to write programs that can run on a single GPU or scale up to run on multiple GPUs or even clusters of GPUs [20]. Another advantage of CUDA is its wide adoption in both academia and industry. Many researchers and companies have utilized CUDA to accelerate their workloads, resulting in a large community of developers and a wealth of resources available for learning and developing with CUDA [17]. A closer, case-dependent inspection of the CUDA programming model is performed in Subsection 5.3.1, motivated by the workload characterization in Section 5.3 and providing insight into high-level diverse workload characteristics.

Working in conjunction with the CPU and connected to it through a PCI-Express bus, the CPU is called the host and the GPU is called the device, in GPU computing terms. A heterogeneous application comprises discrete parts, the host code and the device code, running on the respective devices. Computationally intensive applications usually exhibit a great amount of data parallelism, exploited by the GPU to gain performance. Thus, the CPU handles control-intensive tasks, while the GPU handles highly data-parallel, computationally intensive tasks. [1]

Thread hierarchy

CUDA extends the languages it is implemented in, allowing the programmer to define functions in the respective language, called kernels, that are offloaded to the GPU as opposed to regular functions of the language. Kernels comprise tens of thousands of threads, enabling acceleration of massively parallel applications. Threads are identified with a multi-dimensional (1 through 3) thread id, forming a "thread block". There is a limit to the maximum number of threads in a block, as they all reside on the same streaming multiprocessor (currently 1024 threads), sharing the same memory resources. Mutliple blocks are likewise organized in multi-dimensional grids. Thus, a kernel is executed as a grid of blocks of threads.[21] The basic unit of execution in a GPU is a warp, a group of threads (32 in current implementations), that solely occupies a given SM pipeline stage at any given moment. From compute capability 9.0 onward (11.0 is utilized in our implementation), blocks can be further organized into clusters. Within said clusters, hardware-supported synchronization can be opted for, as well as R/W and atomic operations on a common shared memory [22, 23].

Memory hierarchy

CUDA threads may access various levels of the memory hierarchy during their execution. Said hierarchy consists of:

- Per-thread registers and local memory (mostly used for register spilling).
- Global memory visible by all threads withing a block, and sharing its lifetime.
- Shared memory operated on by all threads withing a block or cluster of blocks.
- Constant memory, backed by the Constant Cache.
- Texture memory, backed by the Texture Cache, providing different addressing modes that service 2d spatial locality.

[21, 24]

3.4 Architecture of the GPU

3.4.1 High-level architecture

A modern GPU comprises many cores, called streaming multiprocessors (SMs) by NVIDIA or compute units by AMD. Each SM is a Single Instruction Multiple Data (SIMD) processor running up to the order of a thousand threads concurrently (it can be considered MPMD if the definition is extended to include simultaneous multithreading). Threads running on an SM can communicate via a per-core scratchpad memory (shared memory) that is often unified with the L1 cache. Concurrent threads on an SM are synchronized with fast barrier

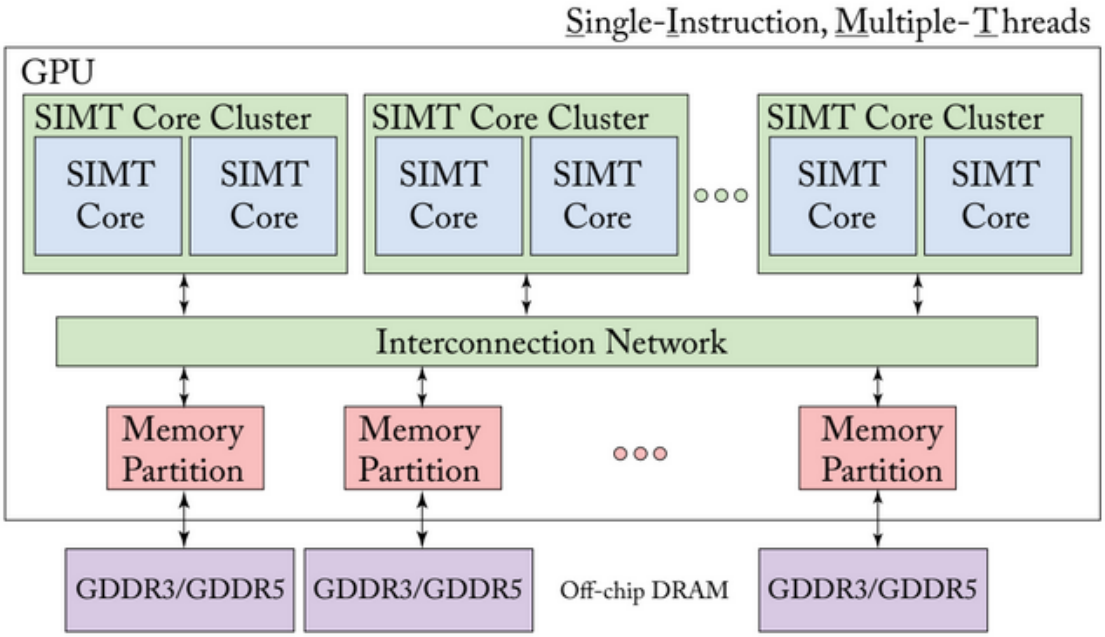


FIGURE 3.2: A model of the high-level parts of a modern GPU architecture [1]

operations. A per-core Instruction cache is also used together with L1 cache to reduce traffic sent off-chip to lower levels of the memory system.

In the context of graphics rendering but not limited to it, GPUs often need to access data sets that are too large to be stored entirely on the chip. To achieve high performance and enable programmability for graphics rendering and game design, specialized on-chip caches and high bandwidth off-chip memory access are required to support frequent memory access. The memory parallelism required to provide the high memory bandwidth is provided by partitioning of the last level memory system (multiple DRAM chips) via multiple memory controllers, each corresponding to a memory partition unit. Often, each memory partition comprises both a memory controller and a respective part of the last-level cache [23, 24].

The SMs and memory partitions are connected via an on-chip interconnection network such as a crossbar (though other NoC organizations are possible). Memory traffic is distributed across the memory partition units using address interleaving. As seen in Figure 3.3, SMs (equivalent to SIMT cores in GPGPU-Sim terminology) are organized in clusters each one of which has a response FIFO that can hold packets coming from the interconnection network. These packets are then directed to either the instruction cache of an SM (in case of a memory response related to an instruction fetch miss) or its memory pipeline (LDST unit).

3.4.2 Cache architecture

As depicted in Figure 3.5, the memory structures within each SM are either shared by the blocks occupying it or partitioned among them. Shared memory and the Register File are

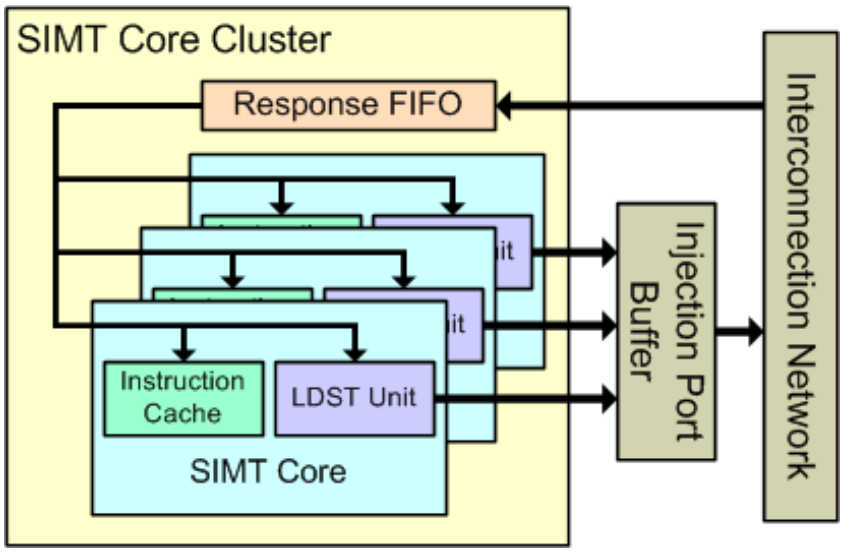


FIGURE 3.3: GPU architecture at the SM cluster level [23]

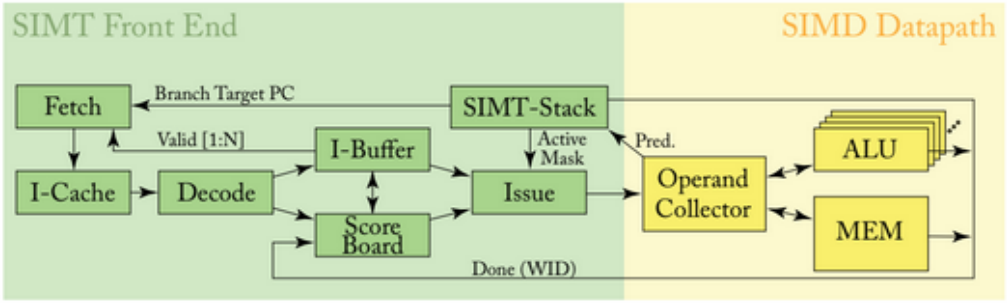


FIGURE 3.4: GPU pipeline stages [1]

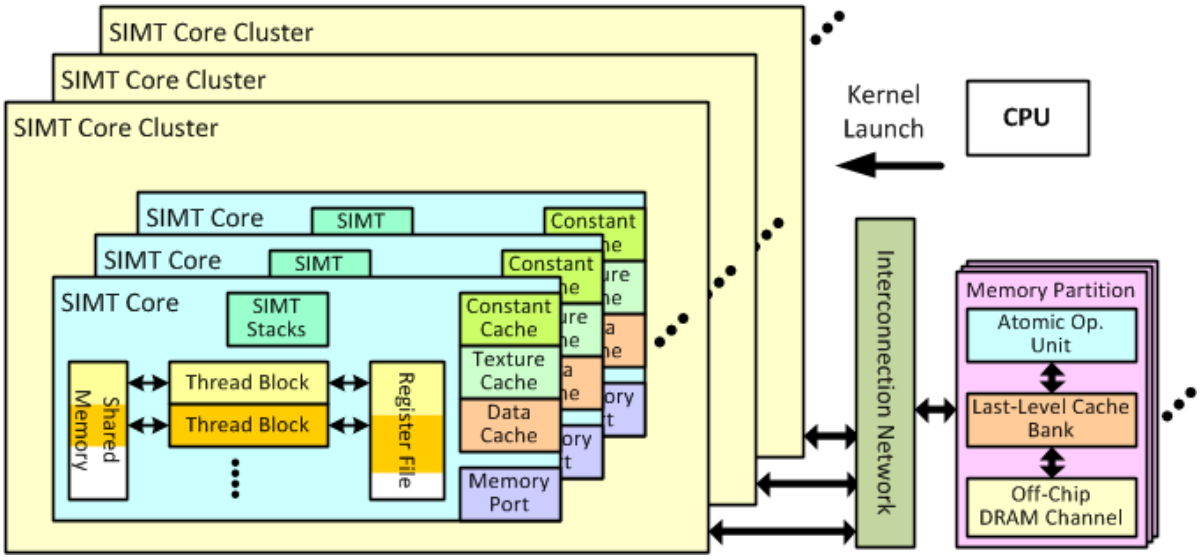


FIGURE 3.5: A model of the internal SM organization, with components both shared and partitioned between blocks. [23]

divided between blocks, and the Register File and SIMT Stacks specifically are indexed by warp ID. Cache types include:

1. The constant cache is a read-only cache that is used to store constants that are accessed frequently by GPU kernels. It is typically small in size and optimized for low-latency access to frequently accessed data, such as constant memory arrays or scalar values. It is typically highly banked. In the Fermi architecture it is organized as a 64 KB cache divided into 32 banks, with each bank having a 4-byte data path [70].

2. The texture cache is a specialized cache used for texture mapping operations, which involve accessing texture data using a coordinate system to map the texture onto a 3D surface. The texture cache is optimized for spatial locality and uses sophisticated filtering algorithms to perform texture filtering operations such as bilinear or trilinear filtering. In texture mapping, an image, called a texture, is applied to a surface in a 3D model to make the surface look more realistic. To render an image, the rendering pipeline identifies the position of one or more texels within a texture image. Texels are the individual points within a texture that contain color and other information. The pipeline then uses the texel coordinates to locate the memory addresses where the corresponding texels are stored. Because neighboring screen pixels usually map to adjacent texels, and it's often necessary to use the values of nearby texels, there is a high degree of spatial locality in texture memory accesses that can be leveraged by caches. It is organized as a hierarchy of texture units, each consisting of multi-banked texture caches, allowing for parallel processing of multiple texture operations. The texture cache is typically larger than the constant cache and can be configured in various modes to optimize performance [71].

3. The data cache is a general-purpose cache that is used to store portions of the local and global memory address spaces that are accessed frequently during GPU kernels. It is typically larger than the constant and texture caches and is organized as a set-associative cache to minimize conflicts and improve performance. The data cache can also use different cache replacement policies, such as Least Recently Used (LRU) or Random, to optimize performance. Additionally, it can be configured to support different data types and access patterns, such as strided or unaligned accesses [71, 72].

4. The shared memory (or "Global Register File") is a memory space that is shared between threads within a single block. It is typically used to improve performance by reducing the number of accesses to global memory and improving inter-thread communication. Shared memory is organized as a high-bandwidth, low-latency scratchpad memory (with a latency comparable to the Register File) that is physically located on the GPU chip. It is typically faster than the data cache and can be used to store intermediate results or to communicate between threads. Shared memory is accessed using load and

store instructions and can be configured in various modes to optimize performance. Shared memory and L1 data cache are often found in a unified configuration. [71, 73].

5. As previously mentioned, the L2 cache is usually a part and the first stage of the off-chip memory partition. Its design includes several optimizations to improve overall throughput per unit area for the GPU [1]. The L2 cache portion inside each memory partition is composed of multiple slices (2 in the Fermi architecture and 40 in Ampere) [74]. Each slice contains separate tag and data arrays and processes incoming requests in order [75]. To match the DRAM atom size of 32 bytes in GDDR5, each cache line inside the slice has four 32-byte sectors [76, 77]. Cache lines are allocated for use either by store instructions or load instructions. To optimize throughput in the common case of coalesced writes (concurrent writes from multiple threads across warps) that completely overwrite each sector on a write miss, no data is first read from memory. This is quite different from how CPU caches are commonly described in standard computer architecture textbooks. How uncoalesced writes, which do not completely cover a sector, are handled is not described in the relevant patents, but two solutions are storing byte-level valid bits and bypassing the L2 entirely. To reduce the area of the memory access scheduler, data that is being written to memory is buffered in cache lines in the L2 while writes await scheduling [1].

3.4.3 Pipeline stages

To maintain high performance on the GPU, it is essential to balance high memory bandwidth with high computational throughput. The pipeline model of the GPU provides further insight on the mechanisms associated with this [1, 24, 23]. Figure 3.4 provides a visual representation of the internal architecture of the Streaming Multiprocessor pipeline. The pipeline consists of a SIMT front-end and a SIMD back-end. Similar to a CPU implementing multithreading, the SIMT front-end enables concurrent fetch, decode, and issue of warps. The pipeline scheduling occurs in three continuous "loops": the instruction fetch loop, the instruction issue loop, and the register access scheduling loop. The instruction fetch loop includes the Fetch, I-Cache, Decode, and I-Buffer blocks seen in Figure 3.4 . The instruction issue loop includes the I-Buffer, Scoreboard, Issue, and SIMT Stack blocks. The register access scheduling loop includes the Operand Collector, ALU, and Memory blocks. A per-stage view of the pipeline model is presented in 3.7.

1. In each cycle, a warp is selected for scheduling. Its program counter is used to access an instruction cache and find the next instruction to execute. After the instruction is fetched, it is decoded and source operand registers are fetched from the register file.

2. In parallel with fetching source operands from the register file, the SIMT execution mask values for the SIMT stack (elaborated on below and in Section 3.9) are determined.

3. The Fetch and Decode stages are followed by placing the instruction in an instruction buffer, where it remains until it is determined that no structural or data hazards exist and it is scheduled to the backend. This dependency check that is traditionally done via scoreboarding allows the GPU to exploit vacancies in the backend and schedule instructions before others from the same warp have committed, essentially hiding long latency memory operations as well. The traditional scoreboard paradigm that is usually implemented in the GPU is fairly simple. Each register is represented with a single bit that is set when a write is still pending to it. This prevents Read-After-Write and Write-After-Write hazards. Combined with in-order instruction issue that is usually employed in GPUs, Write-After-Read hazards are also prevented, provided Register File accesses are also bound to occur in-order.

4. An issue scheduler is used to decide which of the several instructions to be issued to the rest of the pipeline and which warps to prioritize. Structural hazards encountered in this stage are handled with instruction replay, as if speculative execution was taking place. The SIMT stack is updated in this stage. One of the key characteristics of contemporary GPUs is the SIMT execution model. From a functional standpoint, this model allows individual threads to execute independently, although not necessarily with improved performance. While it is possible to achieve this programming model using predication alone, current GPUs employ a combination of traditional predication and this stack of predicate masks [1] . The approach used in current GPUs is to serialize execution of threads following different paths within a given warp [49] . The SIMT stack has certain drawbacks, namely area cost (proportional to $in\_flight\_warps \cdot warp\_size \cdot max\_warps$), lower SIMD efficiency and needless serialization, inadequate MIMD abstraction due to the forced reconvergence and not accounting for user-implemented synchronization mechanisms and system-level interrupts, hence the alternative modern approached to handling warp divergence.

5. Instructions selected by the Issue scheduler are placed in Collector Units in a structure called the Operand Collector. As already explained, to hide long memory latencies it is essential for many warps to be concurrently executing on the GPU. And to support such concurrent execution and fast context switching it is necessary to have a large Register File that contains separate physical registers for each warp that is executing. This area has been upwards of 256KB already in pre-Pascal architectures. The area of an SRAM memory used to implement the RF is proportional to the number of its ports. A naive implementation requires one port per operand per instruction issued per cycle, which would equate to large area and power overheads. In reality, this large number of ports is simulated using multiple banks of single ported memories that can be concurrently accessed (rarely exposed to the ISA), and a structure called the Operand Collector, seen in Figure 3.6a. The operand collector comprises a set of collector units that are either

generic to the backend execution pipelines or specialized to them. They contain buffering space for all source operands required to execute an instruction. Given the large number of source operands and the many concurrently scheduled instructions in the pipeline (potentially many from the same warp), "bank-level parallelism" is achieved, allowing simultaneous bank access and essentially simulating a multi-ported Register File. Specific scheduling policies are employed from the RF arbitrator to minimize bank conflicts. Other techniques, such as swizzled bank register layout, where registers of adjacent warps are offset with regards to the bank they are assigned to, are also widely implemented. Techniques to improve RF performance will be further elaborated on in Section 3.4.5.

6. When all of the source operands have been collected for an instruction, it is placed in a pool of instructions available to be selected by the dispatch scheduler for execution in the lanes of the appropriate Execution Unit. The Execution Units are typically heterogeneous meaning a given function unit supports only a subset of instructions. They contain as many lanes as there are threads within a warp. In many implementations, fewer lanes exist and a single warp is executed over several clock cycles. NVIDIA GPUs are equipped with various units including a Load/Store unit, Integer functional units, Floating-point functional units, Special Function Units (SFU), and most recently, Tensor Core units with the release of Volta.

7. In the Writeback stage, instructions issue a Register File write for all their destination operands and update the scoreboard freeing any instructions dependent upon them.

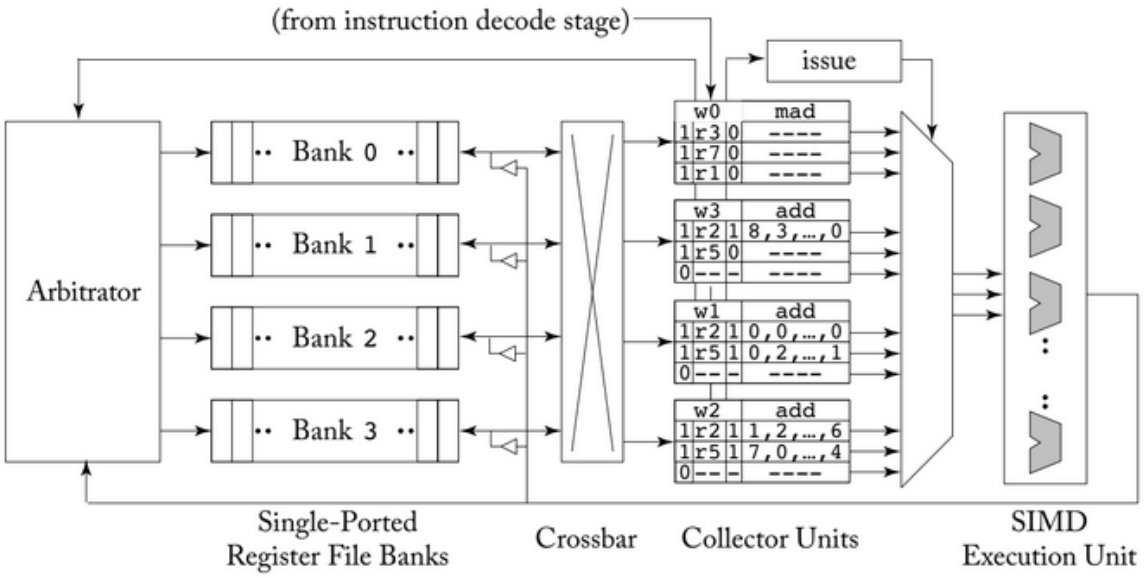
3.4.4 Parallelism exploited by the GPU

GPUs, owing to the potential to greatly exploit both levels of inherent application parallelism, have become popular for media applications. This, coupled with the availability of CUDA and OpenCL has opened up the option of general purpose computing. They exploit every type of parallelism that can be captured by the programming environment, and are essentially a hybrid of all the models in Flynn's taxonomy [78]:

- *Task-level parallelism:* Multiple divergent instruction streams can be represented by the concurrent execution of multiple warps (brfanch divergence) and multiple blocks (different points of synchronization) on an SM, which in CPU terms is equivalent to multi-threading (MIMD).

- *Data-level parallelism:* Individual warps implement SIMD by simultaneously operating on thread-id-indexed data using the same instruction, thus, leveraging the DLP of the applications.

- *Thread-level parallelism:* Warps are equivalent to threads in CPU terms, and concurrently execute and communicate within an SM.



(A) GPU Operand Collector [1]

- *Instruction-level parallelism:* Instructions from the same are often not inter-dependent. Typical GPU architectures account for data hazards by scoreboarding and stall the issue stage for warps that exhibit them. In LOOG, register renaming is implemented to eliminate WAW and WAR hazards, as well as concurrently execute independent instructions. This exploits the manifested ILP of the kernels run.

3.4.5 Kernel execution sequence

The GPGPU compute architecture comprises two units of execution: a host program, and a set of kernels (denoted by the "__global__" CUDA C keyword, or the "__device__" keyword when invoked by the host or the device respectively) typically consisting of hundreds of thousands of scalar threads, executing the same instruction stream on different data in an SIMT fashion [1, 21]. GPGPU applications begin execution on the host, which then offloads kernels to the device (GPU). The previously described thread hierarchy is initialized as follows:

A grid of thread blocks is initialized based on the parameters specified in the kernel launch, and using a GPU-wide block scheduling mechanism. Restrictions in block scheduling include total number of registers (binding threads) and available warp slots.

Each thread block is assigned to a Streaming Multiprocessor. More than one blocks can be assigned to the same SM concurrently, within parameter limits. During a kernel launch, the programmer specifies the number of warps per block and shared memory allocated per block.

The warps of each block are assigned their respective execution resources (Ibuffer, RF as well as dispatch and issue schedulers in architectures with "separate processing blocks" -in NVIDIA terminology [49] - or sub-core model -in GPGPU-sim terminology [23] -)

Warps execute in lockstep, using a single Program Counter (PC) for all their threads (despite uArch optimizations being applied to this execution model, such as independent thread scheduling and sub-warp coalescing). The order in which warps execute within a block, or blocks relative to each other is not specified and hidden from the programmer.

Each thread has a separate thread ID denoting the data assigned to it for execution exclusive access to a limited number of registers. Having a separate set of registers per thread allows for fast and efficient context switching. Due to the very large number of active contexts (warps), the capacity of the register file tends to be substantial and a lot of effort has been put into optimising it in terms of efficiency, area and power consumption

Synchronization is achieved on three levels

- The aforementioned exertion of threads within a warp in lockstep.
- Block-level synchronization barrier instructions called by the host ("__syncthreads" in current CUDA C implementations).
- Grid-level synchronization can be achieved by using consecutive kernel calls (launches), ensuring that all threads begin and end their execution simultaneously (GPU synchronization or implicit synchronization) [79] .

As was mentioned in the memory hierarchy section, each thread has exclusive access to a lifetime-defined set of registers in the register file, to enable the required thread-level and warp-level concurrency, enforcing a cache-like capacity on the Register File (256KB for Quadro GV100). Great effort has been put into optimizing it in terms of efficiency, area and power consumption. Namely:

- A Register File Cache (RFC), implementing the concept of a hierarchical RF and exploiting the fact that most registers are only read once. This technique is frequently paired with a two-level active warp scheduler, which restricts execution to just a pool of active warps, allowing for significant reduction in the RFC size. When said warps encounter a long-latency memory operation they are removed from the RFC and relevant contents are flushed. This technique is used in the Volta GV100 architecture and modelled by increasing the number of banks in the respective Accel-sim configuration [80, 36].
- Register file partitioning exploits the same concepts, using a Fast Register File and a Slow Register File and implementing the latter with Near-Threshold Voltage (NTV) SRAMs, providing lower access energy and less leakage power [81].
- Drowsy State Register File is a power-aware technique where a trimodal entry register file is proposed, with entries that can switch between {ON,OFF,Drowsy} modes, where the latter retains the value of the register but needs to be awaken to ON a reasonable

time before the register is accessed. Setting each register to Drowsy mode after each access allows most registers to spend most time in this mode, reducing the leakage power of the RF [81].

- Register file virtualization, where the physical size of the RF is reduced, and renaming to virtual physical registers is utilized [82].
- In Regless, the register file is eliminated and replaced with an operand staging buffer. At given timeframes, only a small portion of the RF is accessed. Therefore a compiler optimization is implemented, where the kernel is divided up into contiguous regions, each with a specific set of live registers. Registers are normally stored in a backing storage area in global memory and potentially backed by the L1 data cache. When a warp begins executing, the appropriate registers are brought into a cache called the Operand Staging Unit [83].
- In the LOOG microarchitecture, instruction reordering that takes place due to instruction stream ILP significantly reduces Register File traffic. Instructions allocating CUs and waiting for one or more source operands provided by in-flight instructions, can immediately capture said operands without issuing an RF read. Similarly, instructions whose destination registers were renamed while in-flight do not have to issue Register File writes, as that would cause WAW hazards for instructions to come. In the original baseline microarchitecture modelled, and for the given set of workloads, 34.8% of RF reads and 20% of all RF writes are avoided due to writeback stage result broadcast [12, 13, 14].

Unlike the number of warps per block and the portion of the shared memory assigned to it, which are programmer specified, the number of registers per warp are defined by the compiler and depend on kernel code. All three of these parameters limit the Achieved Occupancy (defined as the ratio of warps concurrently executing on the GPU to the maximum number of warps that can be supported). This term is often encountered as "Warp Occupancy" but in the context of this thesis the latter will be used to describe the ratio of average active scalar threads per warp to maximum threads. High Achieved Occupancy usually equates to better performance, until the limit set by inter-warp cache contention is encountered [84].

3.5 Reconfigurable architectures

Given the halt met by the scaling laws, several chip and architecture design approaches have been used in the effort to meet performance requirements of modern specialized workloads. High performance microprocessors encapsulate all high performance and energy-efficient single core and multi-core hardware accelerators, including the GPU.

Application-Specific and Domain-Specific Architectures

The ASIC (Application-Specific Integrated Circuit) is a prime example of this trend, designed for specific applications and use cases, and providing a high performance energy efficient design. It does not suffer the slow and power-hungry of serial instruction fetch, decode and execution phases often met in microprocessors. It is designed with hardware description languages (HDL), allowing designers to precisely describe the functionality of the circuits just above Register-Transfer Level (RTL) abstraction [25, 26].

Execution-Unit level Domain Specific Architectures (DSA) are another promising solution to the adaptability issue. They entail specialized Execution Units such as the TPU or the Pixel Visual Core (designed for TensorFlow and Halide respectively), following domain-specific design guidelines and being tailored to the computational requirements of respective software frameworks [27].

Fine-grain reconfigurable computing systems

Given the fact that the Non-recurring engineering (NRE) cost of ASICs needs to be amortized over large production volumes, the still maturing field of reconfigurable architectures has emerged [28]. They comprise one or more programmable processing units called "reconfigurable logic" and programmable interconnections called "reconfigurable fabric". Custom functional units can be built on the former, performing a specific data-dominated task, interconnected by the latter. They combine some of the flexibility of software with the high performance of hardware. Great efforts have been put into various relevant domains such as system-level architecture optimization, reconfigurable fabric design and reconfigurable logic design [29]. The FPGA is a prime example of the aforementioned reconfigurable fabric. Introduced in 1984 [85], it is an IC that contains an array of programmable logic blocks attempting to bridge the customization gap between set-in-stone single core architectures (mentioned as VN earlier) and ASICs. Its configuration is performed with a hardware description language much like the ASICs, as well as block diagramming techniques. They can be reprogrammed to perform different functions and suit a variety of applications and their flexibility also allows designers to prototype and test secondary designs. Said flexibility allows for hardware to be reused in multiple applications, reducing manufacturing cost and time-to-market [30].

However their high configurability comes at high costs (power and area utilization as well as reconfiguration overhead), especially for high performance applications, and their limited resources such as logic blocks and memory limits the complexity of the designs that can be implemented. Considering the above, has given rise to specific contemporary trends in reconfigurable architectures [4, 31]. Coarse-grain fabrics, which alleviate the overheads set by large interconnects by increasing the granularity of logic units, are setting the expectation to see a migration to more complex logic blocks, even

stand alone FPGAs.

Coarse-grain reconfigurable architectures

Coarse-grained reconfigurable architectures (CGRAs) are a class of reconfigurable computing systems that provide a balance between performance and flexibility by integrating programmable logic resources with specialized functional units [32]. In the classification presented in Figure 2.1a, they occupy the reconfigurable systems area, along with FPGA, but closer to VN architectures in comparison to them. CGRAs are composed of a large number of processing elements (PEs) operating on word-level and interconnected by a special-purpose routing fabric, allowing for efficient data movement and parallel processing. Compared to fine-grain reconfigurable architectures they offer higher performance, reduced reconfiguration overhead, better area utilization, and lower power consumption [30].

The PEs are specialized functional units optimized for specific tasks, such as arithmetic operations, memory access, and data movement. The flexibility of the routing fabric allows the CGRA to be reconfigured for different applications or even within the same application to optimize performance [32].

CGRAs have been used in a variety of application areas, including image and signal processing, scientific computing, and machine learning. For example, the Trimaran compiler framework [86] has been used to develop custom CGRAs for multimedia processing, achieving up to 8 times the performance of a traditional processor for video decoding tasks.

While CGRAs offer high performance and flexibility (potentially more on the former than FPGA but less on the latter), they also come with challenges. One challenge is the development of programming models and tools that can efficiently utilize the specialized functional units and routing fabric. Another challenge is the optimization of the routing fabric, which can have a significant impact on performance and area. Flexibility must be taken into account and the architecture should be characterized by generality (the PE mix should be able to address all classes of applications) and regularity (the PEs and interconnects should be organized in regular structures), so that the system can adapt to the requirements of applications that were not known at design time. Hence, the number of the reconfigurable units, their implemented operations and their organization are all critical design parameters [30]. Despite these challenges, CGRAs remain a promising approach for accelerating a wide range of applications.

Soft cores

With the aforementioned advancements in technology and the availability of more transistors in both reconfigurable logic and fabric, it is possible to include complex non-programmable or semi-programmable functions in heterogeneous architectures that have both general-purpose logic resources and fixed-function embedded blocks.

The term "Soft Core" is used to describe specific processors that have a synthesisable version provided by the vendor in a hardware description language (HDL) and can be either implemented by the user using a reconfigurable fabric, or etched as an ASIC. In many applications, soft-core processors provide several advantages over custom designed processors such as reduced cost, flexibility, platform independence and greater immunity to obsolescence. However, as explained, reconfigurable computing comes at the cost of larger area and performance overheads due to flexible routing on the bit level when compared to ASIC technology [33, 34].

3.6 Heterogeneous architectures

Heterogeneous computing systems refer to systems that employ multiple processing units with different architectures, capabilities, and functions, operating on the same workflow to perform a single or multiple computations and assign each one of them to the processing element that suits it better. Heterogeneous systems are becoming increasingly popular in scientific computing and other application areas due to their ability to provide high performance, energy efficiency, and scalability [35]. One example of a heterogeneous computing system is a CPU-GPU hybrid system. Such systems consist of a CPU (Central Processing Unit) and a GPU (Graphics Processing Unit) connected by a high-speed bus, allowing for efficient communication between the two processors. Program control data can be processed by the CPU, while floating point operations can be offloaded to the GPU. Robust orchestration of hardware resources and the inherent software complexity is required [35]. Another example of a heterogeneous computing system is a cluster of heterogeneous processors. These clusters consist of multiple processors with different architectures, such as CPUs, GPUs, FPGAs (Field Programmable Gate Arrays), and DSPs (Digital Signal Processors). The Modular Supercomputer Architecture (MSA) is one such approach for integrating heterogeneous systems suitable for various application portfolios. It is a cluster-based architecture that combines multiple clusters, or "modules," with each module tailored to meet the needs of a specific class of applications. For example, one cluster may consist of CPUs designed for high single-thread performance for low to medium scalable applications. There are new architecture designs that are attempting to reverse the conventional method of building systems around computing elements. Instead, they place a pool of memory that is globally accessible at the center of the system [87].

Another type of heterogeneous computing system is a system that includes both hardware and software components. For example, a system that includes a FPGA and a DSP, such as a software-defined radio (SDR) platform can be used to perform real-time signal processing tasks. The FPGA can be used to perform low-level signal processing

tasks such as filtering and modulation, while the SDR platform can be used to perform high-level tasks such as demodulation and decoding [88].

Finally, the type of heterogeneous architecture that was more thoroughly studied was Asymmetric Chip Multiprocessors (ACMPs). ACMPs are architectures consisting of multiple diverse processors residing on the same chip, typically varying in superscalar degree and Instruction Window length, just like core fusion architectures, subsequently elaborated on in Section 3.5. In the relevant paper by the Intel corporation [7], the performance of applications running in isolation is improved by a heterogeneous archirecture with a single high-performance OOO core and multiple in-order cores, hence targeting desktop PC environments. This paper, however, focuses on programmer effort pertaining to code parallelization, which is alleviated by more efficient execution of the serial portion of applications. Several other similar ACMP architectures have been proposed such as a powerful core and an array of small cores, focusing on energy efficiency [89] and a spectrum of cores varying in Issue Width, cache sizes, branch predictors and number of MSHRs, examining energy efficiency and targeting multi-programmed workloads [90]. Balakrishnan et al. [91] have studied the performance and scalability of commercial workloads on ACMPs and other work has been done to improve scheduling of threads. The two biggest challenges with ACMPs are lifecycle management (LCM) and inter-processor communication (IPC) [92].

3.7 GPGPU-Sim pipeline model

All modelling throughout this thesis is performed with version 4.1.0 of GPGPU-sim [23], the state of-the-art, open-source research GPU simulator, which in this version interfaces with Accelsim [36].
GPGPU-Sim models massively parallel architectures with generalized components, providing enough flexibility and to accurately model modern GPU architectures with appropriate component configuration. The SIMT Core clusters shown in Figure 3.5 represent groups of SMs with a unified Interconnect Injection Port Buffer and Response FIFO. Since these structures are independent in the GV100 architecture even for processing block partitions, it is configured in the simulator with one SM per partition. The per-SM instruction cache in our configuration has a port throughput of 4 instead of 1, to model the per-partition L0 Icache in GV100 which is not implemented in the simulator. The per-SM shared unified L1 data cache/ shared memory is configured as 32KB DL1 and 96KB shared memory. The shader core pipeline model comprises six stages (Fetch, Decode, Issue, Dispatch, Execute, Writeback)

Fetch and Decode are executed sequentially as many times as the value of the instruction_fetch_throughput option defined in the configuration file, representing as many partitions within an SM. m_inst_fetch_buffer acts as a pipeline register between Fetch and Decode. Whenever it is free, he decoder places its contents to the respective

warp's Instruction Buffer. In order for fetch to happen for a warp, its Instruction Buffer needs to be empty. By default, two instructions are fetched from the Icache every cycle, and placed to the respective warp's Instruction Buffer. Warps eligible for fetch are active warps with empty Ibuffer entries and selection is performed in a Round-Robin fashion.

In the Issue stage , a configurable number of issue schedulers, each with their own scheduling policy, select one warp from which to issue up to two instructions per cycle, by default. For a warp instruction to issue, is must not be blocked by synchronization barriers, have no data hazards as verified by the scoreboard, and have free pipeline registers leading to the Collector Units. The scoreboard contains a vector of ready bits for each warp, equal to the maximum number of registers that can be assigned to a warp. Upon issue, the ready bits of the corresponding destination registers are set on the scoreboard and all source operand registers are checked for in-flight pending writes, thus avoiding RAW and WAW hazards. WAR hazards are not accounted for in the simulator by default, but the impact is presumably not significant. After the instruction issues, the SIMT stack and the scoreboard are updated. In GPGPU-Sim [23] version 4.1.0, each warp scheduler has a dedicated register file (RF) and its own execution units (EUs). A warp scheduler, together with the above structures is called a sub-core (the equivalent of processing blocks in the Volta architecture). Sub-cores are isolated, sharing only the instruction cache and the memory subsystem.

In the Operand Collect stage instructions freshly scheduled by the Issue scheduler occupy a Collector Unit (CU). The Issue-OC pipeline registers lead to per-Execution Unit specialized Collector Units, unless the latter are configured as generic. In the GV100 default configuration, 8 generic Collector units are used per core leading to 4 SP, 4 DP, 4 SFU, 4 INT and 4 Tensor core Execution Units. These numbers may differ in the actual microarchitecture, but they are fine tuned given GPGPU-sim's Execution Unit configuration. Instructions scheduled to the OC, submit a read request for each source operand to the RF arbitrator. The instructions remain in their respective CUs until the RF has read the source operands and they are ready to be sent to the Execution Units. A mechanism for handling RF bank conflicts is employed to ensure that the maximum possible RF read throughput is achieved. To interconnect the CUs with the RF ports, a crossbar (X-bar) is utilized.

In the Execute and Writeback stages there are multiple types of SIMD-vectorized (to the width of a warp) and usually pipelined Execution Units (EXUs: Single-precision, Double-precision, Special-Function Units, Load-Store Unit, and Tensor cores in the Volta architecture [49]). Memory operations are directed to the MEM unit. Unlike the predecessor Pascal and Fermi architectures (and as implemented from GPGPU-sim 4.0 onward), in

Volta, transcendental instructions are processed by the SFUs and the SPs are responsible for handling the remaining arithmetic operations. To select instructions for execution, the dispatch scheduler chooses up to one instruction per EXU from a pool of ready-to-execute instructions, giving priority to the oldest. Once executed, the instructions and the values of the destination registers are stored in a set of Execute-Writeback registers. These registers then issue a write request to the RF arbitrator before retiring from the pipeline. It is possible for multiple instructions to perform Writeback operations in a single cycle.

3.8 Accel-Sim

With the implementation of LOOG and the rest of our microarchitecture modifications in the most recent GPGPU-sim performance model (version 4.1.0), access to all the features of Accelsim [36] is granted. A fundamental problem of emulation-based execution-driven simulation is keeping up with the industry. With the rapid scaling of parallelism and introduction of new processing pipelines (i.e. the Tensor cores in Volta), undocumented changes in both the microarchitecture and the ISA happen often and the research baseline assumptions lag behind. GPU architectures widely use a vISA (virtual ISA), a more general form of an ISA that provides flexibility to make machine ISA (mISA) changes that are hidden even in binary level. Accel-sim introduces a flexible frontend that provides the option operate in trace-driven mode. A trace generation tool is provided as a wrapper around NVBit, that produces mISA traces from any CUDA binary, including those that use exotic hand-tuned closed source libraries like cuDNN and CUBLAS. NVbit works by intercepting the machine code of a GPU application as it is executed, and then modifying or replacing certain instructions in real-time. This allows developers and researchers to perform a wide range of tasks, such as tracing memory accesses, profiling performance, and even injecting faults for testing purposes. These traces are converted into an intermediate representation that is input to the performance model. Therefore, implementation of the ISA's functional model is not required and the accuracy of the simulator is improved over using vISA. However, it is not possible to evaluate designs that rely on global synchronization mechanisms or data values stored in registers or memory without execution-driven simulation. Integration of LOOG and our concomitant architectural optimizations in the new GPGPU-Sim version enable trace-based simulation, but emulation-based simulation with NVIDIA's stable, well documented vISA, PTX was used in all of our experiments. Trace-based simulation can potentially increase accuracy through accounting for register allocation and other compiler optimizations, however, in the case of LOOG some degree of inaccuracy would be added as address dependence resolution of reordered instructions takes place in the LDST pipeline. The performance model in Accel-sim is extensively modified and therefore more detailed, configurable and accurate. Configuration validation is enabled by through performance counters output by the model

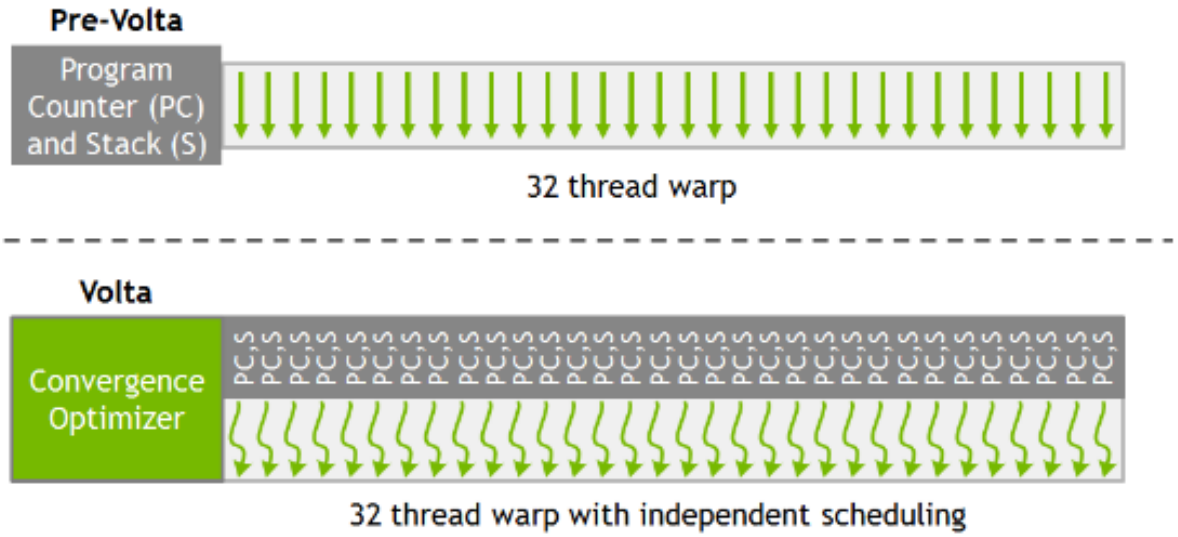
that have a 1:1 correlation with hardware data produced by NVIDIA profilers. These counters are fed into an automated tuning framework that operates in a feedback loop, modifying the configuration file of the architecture being simulated, and producing an accurate performance model by running GPU microbenchmarks targeting various parts of the microarchitecture. Therefore modelling and performance configuration of the tested GPUs provided by Accel-sim, including the GV100 we use in our experiments is ever more accurate. New detailed statistics are added, to which we further contributed a wide range, many of which are LOOG-custom (namely total readiness per warp and overall warp readiness distribution, decoder throughput tracking, RRS occupancy distribution, RAT entries total per warp and RAT entries used per warp, to track ILP).

3.9 Nvidia Quadro GV100 key features

The Quadro GV100 is a professional graphics card manufactured by Nvidia [93]. The term "Quadro" refers to the series of professional graphics cards produced by NVIDIA that have been in production for around two decades are specifically designed for use in high-performance workstations and servers for professional applications such as computer-aided design (CAD), video editing, and scientific visualization [94], as opposed to the equally long-lived GeForce lineup that are meant for use in gaming and the consumer market. "RTX", when present refers to the functionality of real-time ray tracing. The "Quadro" prefix originates from the fact that they support quad-buffered stereo, a type of three-dimensional display used in scientific enterprises like molecular biology responsible for performing complex calculations related to 3D rendering and other graphic-intensive tasks [95]. The "GV" in "Quadro GV100" stands for "Volta", which is the microarchitecture used in the GPU [93]. As a high-performance GPGPU architecture designed for deep learning, scientific simulations, and other HPC applications, the NVIDIA Volta V100 was selected as the substrate of the implementation proposed in this thesis. The architecture was first introduced in May 2017 and has since become a popular choice for data centers and supercomputing clusters. The Volta V100 architecture features several key improvements over its predecessor, the Pascal architecture [49]:

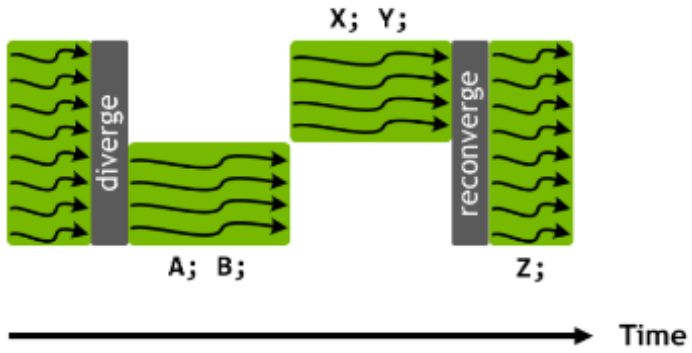
Streaming multiprocessor architecture optimized for Deep Learning

Its new SM architecture is 50% more energy efficient than the previous generation, enabling major boosts in FP32 (SP) and FP64 (DP) pipeline performance in the same power envelope [49]. Its mixed precision computing with independent parallel INT and FP data paths allow for more efficient handling of mixed workloads with computation and addressing calculations. It introduces independent thread scheduling, seen in Figure 3.7a , where each thread has an independent Program Counter and Call Stack, instead of the traditional SIMT stack model depicted in Figure 3.7c , where compatible threads can be coalesced on a sub-warp basis. One of the most significant improvements is the use of Tensor Cores, which are specialized processing units designed to



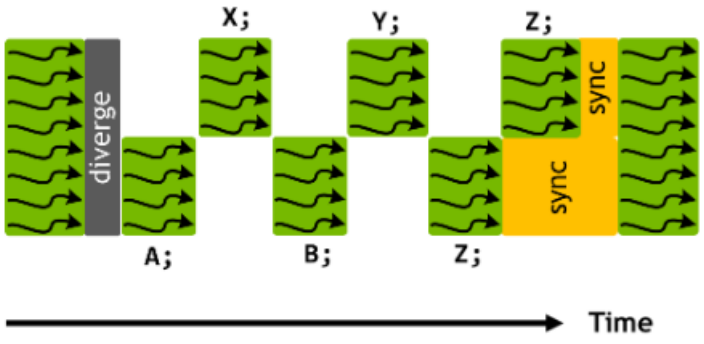
(A) Independent thread scheduling [49]

```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```



(B) SIMT stack [1]

```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
__syncwarp()
```



(C) From CUDA 9.0 onward, __syncwarp() can provide reconvergence on demand [49]

FIGURE 3.7: Methods used for handling warp deivergence

accelerate deep learning computations. Tensor Cores can perform matrix-matrix multiplication and accumulate operations at a much faster rate than traditional CPUs or GPUs.

High-bandwidth memory

The Volta V100 architecture features a highly tuned 16 GB HBM2 (high-bandwidth memory) subsystem which delivers 900 GB/sec peak memory bandwidth, providing significantly more memory bandwidth than previous generations . This enables faster access to data and improved performance in memory-intensive applications.

Unified Shared memory - L1 Data Cache

The Unified Memory technology introduced, includes new access counters to allow more accurate migration of memory pages to the processor that accesses them most frequently, improving efficiency for memory ranges shared between processors. Furthermore, partitioning of the Unified Memory is configurable.

Optimized software

Volta-optimized versions of GPU accelerated libraries such as cuDNN, cuBLAS, and TensorRT leverage the new features of the Volta GV100 architecture to deliver higher performance for both DL inference and HPC applications built with frameworks such as Caffe2, MXNet, CNTK and TensorFlow. As will be mentioned in later sections, the simulator on which our microarchitectural modifications were implemented supports trace-based simulation, permitting the use of these hand-tuned closed source binary libraries.

Maximum Performance and Maximum Efficiency Modes

In Maximum Performance mode, the V100 accelerator will operate up to its TDP (Thermal Design Power) level of 300 W to accelerate applications that require the fastest computational speed and highest data throughput. Maximum Efficiency Mode allows data center managers to tune power usage of their V100 accelerators (by setting a not to exceed power limit on a per-GPU-rack basis) to operate with optimal performance per Watt.

[49]

3.10 Workloads

3.10.1 Benchmark suites used

A total of 100 kernels , impartially selected from 7 benchmark suites have been used over all our testing. Selection was mainly done due to reasonable time constraints of simulations

regarding data that should be collected over multiple design space points. The specialization of each suite is presented below:

The Scalable Heterogeneous Computing (SHOC) [96] benchmark suite is a collection of benchmark programs that are used to evaluate the performance of GPUs and other accelerators for high-performance computing workloads. It consists of a variety of tests that exercise different aspects of a system's performance, including memory bandwidth, floating-point operations, and communication between CPUs and accelerators. At the lowest level, SHOC uses microbenchmarks to assess architectural features of the system. At higher levels, SHOC uses kernels to determine system-wide performance including many system features such as intranode and internode communication among devices. Therefore, this suite addresses issues such as architecture heterogeneity, power throttling performance tradeoffs, and fine-tuned machine-code level to directive-based code level performance tradeoffs. Benchmarks used in this thesis - due to overlap with other suites- include: BFS (Breadth-First Search), FFT(Fast Fourier Transform), GEMM (Matrix Multiplication), NeuralNet (Image recognition),QTC (quality threshold gene clustering), Reduction (data set reduction), S3D(combustion particle simulation) ,Scan (parallel scan), Sort, Spmv, Stencil2D, Triad, spmv-modified.

Lonestar [97] includes applications from several domains deemed "irregular". Said domains comprise meshing, clustering, simulation, and machine learning, and very different algorithmic foundations: they require building, computing with, and modifying large sparse graphs. Applications used from this suite include: bfs-atomic, bfs-wlw, bfs-wla, sssp, sssp-wlc, sssp-wln, bfs-wlc, mst,bh.

Rodinia-3.1 [98] contains a range of scientific applications, including bioinformatics, molecular dynamics, and fluid dynamics, from which we select: Backpropagation, DWT2D, Heartwall, HotSpot, LUD, NW,NN, CFD, Gaussian and K-means. These applications are designed to stress different aspects of a heterogeneous computing system, such as data transfer, memory usage, and computation. This suite will be further elaborated on and its Backprop benchmark used as a CUDA-level example of our workload characterization in Section 5.3.

The Parboil benchmarks [99] are collected from different scientific and commercial fields and are pre-selected to implement scalable algorithms with fine-grained parallel tasks. They include several implementations, some of which are provided as base implementations for new optimization efforts, while others represent the current state-of-the-art targeting specific CPU and GPU architectures. The benchmarks offer opportunities to demonstrate tools and architectures that help programmers get the most out of their parallel hardware. Less optimized versions are presented as challenges to the research

communities to develop technology that automatically raises the performance of simpler implementations to the level of sophisticated programmer-optimized implementations. The benchmarks are continuously optimized for new and existing architectures and the developers welcome new implementations and benchmark contributions from other developers. We use two statistical, compute-intensive (Sum of Absolute Differences, Histogram) and two memory-intensive (BFS, MRI-Q image processing) applications from this suite.

The Dragon Benchmark Suite is a collection of synthetic benchmarks used to evaluate the performance of high-performance computing systems, particularly in the areas of computer architecture, compiler optimization, and parallel processing. It includes applications such as Bitonic Sort,FFT,LU Factorization,Monte Carlo Pi,Reduction,Sparse Matrix Vector Multiplication and Stream Benchmark

ispass-2009 [23] comprises a set of workloads used in the original GPGPU-Sim Ispass paper, meticulously selected to stress implemented architectural features. Workloads we selected due to the aforementioned time constraints include: AES (encryption and decryption for files using CUDA. Constants are stored in constant memory, the expanded key in texture memory, and input data is processed in shared memory). LIB (The study uses Carlo simulations based on a model, with constant memory variables stored in 8KB cache per core. Memory bandwidth is a bottleneck due to local memory accesses), LPS (3D Laplace solver, a finance application optimized for parallel processing that uses shared memory and coalesced global memory accesses. However, performance loss occurs due to branch divergence), NN (convolutional Neural Network for image recognition), NQU (N-Queen solver, which solves a classic puzzle of placing N = 10 queens on a chess board, using a simple backtracking algorithm. One thread does the computation, causing low IPC)

A summary of the above is provided in Table 3.1

3.10.2 Elaborating on the Rodinia benchmark suite

The Rodinia benchmark suite [98] was selected to be elaborated on, as benchmarks belonging to it will be used as examples regarding workload categorization in later sections. The Rodinia benchmark suite [98] was developed according to Berkeley's "Dwarves" to cover a wide range of massively parallel applications, run on GPUs and multicore CPUs using CUDA and OpenMP. Berkeley's "dwarves" (initially seven in Phil Colella's work) constitute classes of applications where coexistence in a class equates to similar computation and data movement. The "dwarves" are specified at a high enough level of abstraction to cover a wide enough range of applications while still containing specialized and future-proof underlying patterns. Initially they were defined as:

- Dense linear algebra (vector-vector, matrix-vector, matrix-matrix operations corresponding to the three BLAS levels respectively). They are characterized by unit-stride accesses to read data from rows and strided accesses to read data from columns
- Sparse linear algebra (operations on elements vectors or matrices with a few non zero elements). Accesses are indexed.
- Spectral methods (such as FFT). These methods are in the frequency domain, and comprise "butterfly" stages with consecutive mutiply-add operations. Data permutations are all-to-all or strictly local, depending on the stage.
- N-body methods (interactions between several particles). Complexities range from $O(N^2)$ in brute force to $O(N)$ or $O(NlogN)$ when aggregating forces for many particles in some implementations.
- Structured grids (data belongs on a regular grid, areas of which are updated together, therefore exhibiting great spatial locality).
- Unstructured grids (data belongs on an irregular grid, and their associations involve multiple levels of memory reference indirection).
- Monte Carlo (computations depend on statistical results of repeated random operations).

EEMB from embedded computing and SPEC2006 for desktop and server computing added more classes to the list:

- Combinational logic (simple operations on large amounts of data, employing bit-level parallelism)
- Graph traversal (indirect table lookup and little computation)
- Probabilistic graphical models (random variables as nodes and conditional dependencies as edges). Access patterns involve indirect table lookup based on these patterns.
- Finite state machines (an interconnected set of states, some of which can decompose into multiple simultaneously active state machines).

As for machine learning, two more "dwarves" were added:

- Dynamic programming
- Backtrack and Branch-and-Bound

Support Vector Machines, Principal Component Analysis, Decision Trees and Hashing are all covered by the above. Backpropagation from the rodinia suite is the first application that exhibits all behaviors manifested

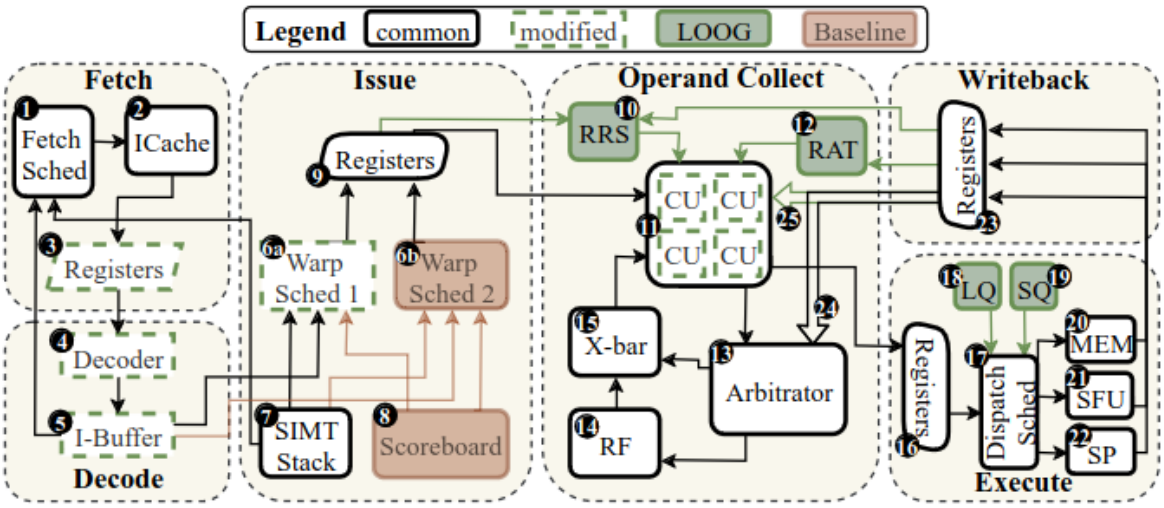


FIGURE 3.8: LOOG modifications on top of the baseline architecture [14]

3.11 LOOG components and modifications implemented

LOOG stands for Light-Weight Out-of-Order GPU execution scheme [12, 13, 14] . It entails re-purposing of typical GPU microarchitectures to exploit ILP and better handle a set of target applications that are characterized by low Achieved Utilization, and concomitant low IPC and frequent stalling. The changes and additions to components of the typical GPU microarchitecture in order to produce LOOG can be seen in Figure 3.8 and are the following:

- The main alteration implemented is modifying the **Collector Units (CUs) with the necessary fields required to serve as the Reservation Stations** of the Tomasulo algorithm [15] .
- **Instructions can therefore be immediately issued from the Instruction Buffer to the CUs, without passing a scoreboard check having first updated a Register Alias Table (RAT)** that is added to eliminate false name dependencies. When an instruction allocates a CU/RS, it reads the per-warp RAT for every source operand. The RAT contains the ID of the CU containing the instruction bound to broadcast the result, if said instruction is still in flight, or a null value if all previous instructions writing on the source operand have committed and the value should be read from the RF. In the former case, the appropriate CU ID is copied to the respective register value field of the CU, until the instruction it describes commits and broadcasts its result on a *result broadcast bus.*
- **The Instruction Window is modified to accommodate exploitation of ILP**. By default 16 to 128 bytes -equivalent to 2 to 16 instructions- are fetched from the ICache on every fetch (4 times per cycle in Quadro GV100), unless the requested address is approaching the end of the cache line. As explained in the baseline model, this IWindow size dictates the size of the Fetch-Decode pipeline registers, as well as the Decoder

throughput and the size of the Instruction Buffers for all warps. As discussed in later sections, Fetch-Decode throughput can be effectively throttled without significantly affecting performance, and without hurting ILP exploitation potential, since instructions are directly issued to the CUs for reordering, provided that issue scheduling is performed at an appropriate depth (examined as well).

- Instruction issue in typical GPU architectures is performed by at least two issue schedulers utilizing different scheduling policies, so as to not cause starvation for warps that would be excluded by a single policy. As examined in [14], unlike the baseline, where due to the short Operand Collect stage issue and dispatch are closely temporally coupled, LOOG is insensitive to scheduling policies, and therefore in the original implementation, a Round-Robin issue scheduler is used. As demonstrated in section , without implementing a more sophisticated scheduling policy and just by configuring the scheduling depth, we uncover great workload diversity. **It is demonstrated that LOOG benefits the most from depth-first warp scheduling**, saturating for all kernels beyond 8 instructions.

- Load-store reordering in LOOG entails **creating a Load Queue and a Store Queue, upon which an entry is allocated for the respective memory instructions to tackle memory data dependencies** beyond just register dependencies. Thus addresses of memory instructions are compared in all cases, between all active scalar threads and the following scenarios arise:

  - In case of a memory barrier operation, check that it is the earliest instruction issued before dispatch.
  - In all other cases, check that a memory barrier operation has not issued earlier than the instruction at hand.
  - In case of a store check that no stores have issued earlier , to avoid WAW hazards, then check all the loads that issued earlier to avoid WAR hazards.
  - In case of a load check all stores that issued earlier to eliminate RAW hazards.

  In the original LOOG implementation, a mere median value of 1.67 separate memory requests per memory warp instruction was documented, despite the potential for 32 requests.

- Result broadcast bus. After finishing execution, an instruction accesses the warp-ID-indexed RAT for all its destination registers. If the CU ID corresponding to the current instruction is found, an RF write request is sent to the arbitrator, and the entry is cleared, **while the result is broadcasted over the broadcast bus**. Otherwise, the register has been renamed while in-flight and the RF does not need to be updated (as a WAW hazard would occur). Comparative logic is added to the CUs for this purpose, and

in the initial implementation the CUs are freed only after writeback of the respective instruction.

- RRS. In order to ensure that effective instruction reordering and ILP exploitation takes place, CU stalls due to CU congestion must be eradicated. When long latency operations enter the CU, other instructions are blocked from allocating them, and potentially reading register values broadcasted by inflight instructions, thus reducing Register File traffic. Increasing the number of CUs is area and power sensitive due to the crossbars that connect them both to the RF and to the pipeline registers leading to the Execution Units. To efficiently partially alleviate this problem, **the Register Renaming Stack (RRS) is introduced, which holds a list of unique IDs to be used in the RAT instead of the CU ID**. Upon instruction issue, the RRS ID is read instead of the CU ID, which is held until the writeback stage. Therefore Collector units can be freed after dispatch, significantly reducing CU congestion, and increasing ILP exploitation. The size of the RRS in bits is $\#RRS \cdot log2(\#RRS)$ [14] .

Name	Description	Characteristics	Kernels	Launches
Shoc				
FFT	Fast Fourier Transform	Spectral methods	3	25
QTC	Data Clustering	Dynamic Programming	1	1
Reduction	Parallel Reduction	Memory BW	1	4
S3D	Combustion Simulation	Dense linear algebra	4	4
Scan	Parallel Scan	Memory BW	3	45
Sort	Radix Sort	Memory BW	5	19
Spmv	Sparse Matrix Vector Mul	Sparse linear Algebra	1	129
Stencil2D	2D Stencil	Structured Grids	1	2
Triad	Streaming Bandwidth	Memory BW	1	124
Lonestar				
bfs-atomic	BFS, atomic ops	Graph traversal atomic	1	1
bfs-wlw	BFS, worklist workload	Graph traversal	3	9
bfs-wla	BFS, worklist aware	Graph traversal	1	4
sssp	shortest path	Graph traversal	1	1
sssp-wln	Weight longest next	Graph traversal	3	9
bfs-wlc	bfs weighted-least cost	Graph traversal	3	3
mst	MST-Kruskal	Disjoint-set data, sorting	6	9
bh	Barnes-Hut particle sim	Parallel task-based model	3	3
Dragon				
testAmr	Adaptive mesh refinement	Memory BW	2	6
testBfs	Breadth-First search	Graph traversal	4	12
testSssp	shortest path	Graph traversal	3	7
testJoin	Database join	Memory BW	4	4
Polybench				
3DConvolution	3D filtering and processing	Compute & BW	1	32
BICG	Biconjugate Gradient	Dense linear algebra	1	1
Correlation	Signal Processing	Compute & BW	2	2
Gramschmidt	Vector Orthogonization	Dense linear algebra	2	2
Ispass-2009				
AES	Adv. Encryption Standard	Compute intensive	1	1
LIB	Monte Carlo simulations	Local memory BW	1	1
LPS	3D Laplace	Compute, divergence	1	1
NN	Neural Network	Graphical model	2	2
NQU	N-Queens	Compute, low IPC	1	1
Rodinia-3.1				
Backprop	NN Backpropagation	Compute intensive	2	2
CFD	Fluid Dynamics	Compute intensive	3	15
DWT2D	2D discrete wave transform	Compute intensive	2	2
Gaussian	Gaussian elimination	Dense linear algebra	6	12
Heartwall	Image processing	Graphical model	1	1
Hotspot	Diffusion simulation	Compute intensive	1	1
Kmeans	K-means clustering	Compute intensive	1	1
LUD	LU decomposition	Dense linear algebra	3	46
NN	Neural Network	Graphical model	1	1
NW	Sequence alignment	Compute intensive	1	104
Parboil				
sad	Sum Absolute Differences	Compute intensive	1	1
mri-q	Image processing	Graphical model	1	1
histo	Histogram calculation	INT compute intensive	2	2
bfs	Breadth-First Search	Graph traversal	2	3

Table 3.1: Workloads used in our simulations

Chapter 4

Prior Art

4.1 Introduction

In this chapter, the literature examined on workload characterization as well as reconfigurable architectures will be presented, providing insight into the concepts and methods utilized.

4.2 Characterization of workloads

4.2.1 "Whole Picture Characterization"

In Whole-Picture Analysis [47, 46] workload characterization (WPC) it is demonstrated that for accurate CPU workload characterization results to be obtained, the methodology used has to take three levels of workload characteristics into account:[46, 47]

- *ISA-independent:* To obtain ISA-independent characteristics, an Intermediate Representation (IR) is used to provide both a static (IR code) and dynamic (IR stream) analysis using IR tools such as LLVM. The IR stream is fed to a presumed processor model with infinite registers but without cache or pipeline modeling. Thus, instruction mix, branch behaviors and instruction and data locality metrics are obtained. Our methodology includes the dynamic instruction mix analysis by instruction type statistics provided by gpgpu-sim for all instructions executed. Data locality metrics are obtained via averaging out cache statistics over multiple architectural configurations including caches of varying sizes (collection of instruction locality statistics is not sensible on the GPU).

- *Microarchitecture-independent:* ISA-level analysis includes all of the previous level characteristics as well as perfect cache behaviors and parallelism behaviors, by modeling execution of the binary stream on a perfect processor without a pipeline model. Such statistics are obtained in our implementation by running workloads on configurations with exaggerated cache sizes and LOOG instruction windows.

- *Microarchitecture-dependent*: This level encompasses collecting runtime hardware metrics produced by actual workload execution. The most comprehensive characteristics

included at this level (from a CPU perspective) include instruction mix, branch predictor behavior, cache behavior, TLB behavior and pipeline system behavior. Our approach includes all of the above that are applicable on the GPU.

Thus, "Whole picture analysis" [47] provides the following metrics for program characterization (those that are sensibly applicable on the GPU are colored green):

- Instruction mix
- Instruction locality
- Data locality
- Branch predictability
- Parallelism

Instruction locality and branch predictability are omitted, since branch instructions are scarce in PTX, and always produce control hazards on the GPU.

4.3 Prior Art regarding reconfigurable and heterogeneous architectures

A selection of modern reconfigurable architectures proposed in literature was studied in the context of this thesis, belonging to the categories previously described.

4.3.1 Further classification of reconfigurable architectures

Apart from the aforementioned classification of reconfigurable architectures regarding fine or coarse granularity and homogeneity or heterogeneity, two other distinct categories are worth mentioning:

Degree of reconfiguration

Partial reconfigurable architectures and full reconfigurable architectures are two types of reconfigurable architectures used in FPGA-based systems, nevertheless, the concept can be extended to any sensible architecture discussed below such as reconfigurable ACMPs or scalable processors. Partial reconfiguration allows for dynamic reconfiguration of only a portion of the hardware, while keeping the rest of the design intact. This has several advantages over full reconfiguration, including shorter reconfiguration times, lower power consumption, and increased design flexibility and reusability of hardware resources [100]. In contrast, full reconfiguration entails complete dynamic reconfiguration of the entire hardware platform. While full reconfiguration can provide greater flexibility in certain applications, it can also

be more time-consuming and resource-intensive. Overall, the choice between partial and full reconfiguration depends on the specific application requirements, and designers must carefully consider the trade-offs between flexibility, performance, and resource usage [101].

Temporal granularity of reconfiguration

Static reconfigurable architectures typically use fixed circuits that can be reconfigured at design time to optimize performance for specific applications. **The definition is often extended (as is in this thesis) to tailored reconfiguration on a per-workload granularity.** These architectures are typically implemented using FPGAs, but this concept can also be extended as mentioned in the previous paragraph. Once the FPGA has been programmed, its configuration remains fixed until the next reprogramming cycle. Dynamic reconfigurable architectures, also known as run-time reconfigurable architectures, allow for reconfiguration of the hardware during runtime. This allows for the system to dynamically adapt to changes in the workload and optimize performance for specific tasks. Dynamic reconfigurable architectures can be implemented using FPGAs or other types of programmable logic devices that can be reconfigured on the fly. Dynamic reconfigurable architectures are used in applications where the workload is unpredictable or changes frequently, such as in data centers or cloud computing environments. They are also commonly used in applications where real-time responsiveness is critical. Semi-dynamic reconfiguration describes architectures that can be reconfigured only at specified time intervals [102].

4.3.2 Related work on heterogeneous and reconfigurable architectures

Reconfigurable Chip Multiprocessor (CMP) architecture

Reconfigurable CMP architectures provide an alternative to Asymmetric Chip Multiprocessor architectures, which are set-in stone heterogeneous core architectures and is the traditional design approach to handling workload diversity.

Core fusion [37] is a reconfigurable CMP architecture where groups of fundamentally independent processors can either be used as distinct processing elements (scaling-out) or be dynamically morphed into a single large CPU (scale-up). It heavily relies on design reuse by exploiting RISC or CISC ISAs and added components that enable reconfiguration to support heterogeneous and massively parallel workloads by efficiently performing the FUSE and SPLIT operations between subsets of cores during runtime. It allows multiple dynamically allocated processors to share a single contiguous instruction window. Due to the use of said ISAs, some structures (e.g. register renaming) must be physically shared, limiting its scalability to 8-wide issue.

Composable Lightweight Processors (CLP) [38] are typically based on a modular design, where individual processing elements can be combined or "composed" to create custom computing systems that are tailored to specific workloads. They address the aforementioned problem by using an EDGE ISA.

Scalable cores

In the Elastic Core architecture [**elastic**], resources along with operating voltage and frequency are dynamically scaled to match application behavior. Elastic Core utilizes a linear regression model for power and performance prediction to guide the scaling of the core size and the operating voltage and frequency to maximize efficiency, essentially implementing Dynamically Frequency and Voltage scaling (DVFS), a widespread technique in modern power-aware designs.

Morph Core [8] outperforms several previously mentioned designs by starting with a high performance, long instruction-window OOO core and dynamically making the minimum necessary changes to transform it into a highly threaded in-order SMT core when necessary. Being able to operate both in in-order and Out-of-Order mode, shutting off power hungry instruction reordering structures such as the renaming logic, OOO scheduling and the load queue in the former. It essentially performs unit-level clock gating to restrict the dynamic power dissipation in inactive components. Leakage power is still consumed (sub-threshold leakage, gate leakage and band-to-band tunneling). Complete power gating (also known as sleep-transistor technique) is not performed as according to McPAT it does not justify the unit-level gating overhead. The reconfigurable architecture we propose in Chapter 5 is closest to Morph Core, regarding the literature studied in the context of this thesis.

The Flicker architecture [11] addresses the issues of coarse grain power control in the respective gating and uniform power allocation in core-level gating as well as the complexity of having multiple voltage domains within a reconfigurable core. Given that applications widely vary in the pipeline width that best balances performance and power consumption, it utilizes deconfigurable lanes –horizontal slices through the pipeline– that permit tailoring an individual core to the running application with a lower overhead than microarchitecture-level adaptation, and greater flexibility than core-level power gating. While cores are homogeneous in design, they can be dynamically reconfigured into a heterogeneous multicore system that meets power constraints.

Heterogeneous architectures

In the context under examination, all heterogeneous architectures presented here represent Asymmetric Chip Multiprocessors (ACMPs), also known as heterogeneous multiprocessor systems on a chip (MPSoC).

Big.Little [39] is a heterogeneous architecture for mobile devices, developed by ARM Holdings. The architecture consists of a combination of high-performance cores and low-power cores, which support the same ISA and work together to optimize performance and energy efficiency. It introduces a solution to optimize power consumption by selecting the core type most suitable for a level of processing load along with high performance.

In the Heterogeneous Block Architecture (HBA) paper [40], two observations are made. Firstly, most serial code exhibits fine-grained heterogeneity. At the scale of tens or hundreds of instructions, regions of code fit different microarchitectures better due to separate memory-intensive and compute-intensive phases (this holds true regarding the GPU as examined in Section 5.3) , which is exploited by migrating threads to "big" and "little" cores respectively. Secondly, this fine-grained heterogeneity allows to split the code into atomic blocks that execute independently in their respective backends using a well defined communication interface (liveins - liveouts). The heterogeneous backends that are conbined into one in HBA include Out-of-Order, VLIW and smaller Instruction Window in-order backends.

In Dynamic Core Boosting [9], the observation is made that in SMT architectures, overall latency is dictated by the execution time of the longest running thread. DCB implements a software-hardware cooperative system that mitigates this workload imbalance in performance assymetric CMPs by leveraging individual CMP-level DVFS to boost ctritical threads. DCB coordinates its compiler and runtime to enable asymmetric CMPs to achieve near-optimal utilization of core boosting. The compiler instruments the program with instructions to give progress hints and the runtime monitors their execution, enabling DCB to intelligently accelerate selected threads within a total core boosting budget for better performance.

Soft cores

The TRIPS (Tera-op, Reliable, Intelligently adaptive Processing System) is a scalable and energy efficient reconfigurable computing paradigm that addresses the lag between Instruction Set Architecture (ISA) design and microfabrication technology as well as the drained scaling of chip resourcess, which is manifested as the exhaustion of pipeline scaling in microprocessors. It implements a Very Long Isntruction Word (VLIW) Explicit Data Graph Execution (EDGE) ISA, a type of Instruction Set Architecture (ISA) that is designed for efficient execution of data-intensive workloads. EDGE is a dataflow architecture that relies on a directed acyclic graph (DAG) to represent the dependencies between instructions [41].
Unlike traditional von Neumann architectures, where instructions are executed sequentially and data is stored in memory, the EDGE architecture directly operates on the dataflow graph, and performs computation as soon as all the input data for a particular instruction is available. This eliminates the need for explicit instruction sequencing and allows for parallelism at the instruction level. It also addresses the pipeline-depth limit with fine-grain concurrency mechanisms. The main benefit of an EDGE IDA is the direct-instruction communication

which entails direct delivery of the producer instructions output to the consumer instruction by the hardware. Thus, instructions execute in dataflow order. Its en mass speculative instruction scheduling allows for exploitation of ILP aside from TLP and DLP. Required compiler support is the most significant among its drawbacks [42].
TRIPS has features that allow the processing cores and on-chip memory system to be configured and combined in various modes, such as instruction, data, or thread-level parallelism, in order to adapt to concurrency of different sizes. To support small and large-grain concurrency, the TRIPS architecture includes four Grid Processor cores that are out-of-order and have a 16-wide-issue, which can be partitioned if there is easily extractable fine-grained parallelism. This polymorphic approach offers better performance across a wide range of application types compared to an approach where numerous small processors are combined to run workloads with irregular parallelism. The results demonstrate that high performance can be achieved in all three modes, i.e., ILP, TLP, and DLP, showing that the polymorphic coarse-grained approach is a promising option for future microprocessors. Its main innovation lies in using guard bands that separate instruction groups which are then executed atomically. [41, 42]

4.3.3 Reconfigurable GPU architectures

In Bahurupi [43], a polymorphic heterogeneous multi-core architecture is fabricated as a homogeneous multi-core system containing multiple identical, simple cores. The main novelty of Bahurupi lies in its ability to morph itself into a heterogeneous multi-core architecture at runtime under software directives, thus both exploiting explicit Thread-Level-Parallelism and inherent Instruction-Level-Parallelism. it is essentially an implementation of core fusion in GPUs with less distributed hardware and microarchitecture modifications, thus smaller overheads. By coalescing small 2-way Out-of-Order processors into groups of 2-4 it can create large virtual superscalar cores.

In Equalizer [44], it is observed that specific warps of the kernel frequently access a bottleneck resource, thus restricting other threads from accessing it, causing under-utilization of other resources and hindering performance. Equalizer dynamically monitors the resource requirements of a kernel and manages the amount of on-chip concurrency, core frequency and memory frequency, to adapt the hardware to best match the needs of the running kernel. Thus, it can function in two complementary modes. Firstly, by throttling under-utilized resources it can save energy without significant performance degradation. Secondly, it can boost bottleneck resources to reduce contention and provide higher performance without significant energy increase. It does so by adjusting three parameters. Number of concurrent thread blocks (to handle resource contention, such as L1 Data cache), SM frequency and memory frequency to match the requirements of the executing kernels.

Amoeba [45] makes the distinction between recent developments in scale-up and scale-out architectures, accelerating respective workloads exhibiting varying scalability patterns

with architectural resources. Examples include Network-on-Chip, sub-warp coalescing, memory, control divergence, and L1 cache behavior and contention. The observation is made that neither approach optimally suits all domains of applications, leading to significant performance degradation for some applications. AMOEBA aims to dynamically monitor application scalability and adjust the SM configuration to meet the requirements. It is essentially an implementation of core fusion, utilizing an online controller that uses a binary logistic regression to predict application scalability and fusing SMs on a coarse-grain basis. The main criterion for scale-up and scale-out reconfiguration are pipeline stalls due to control hazards. Interestingly, this is found to be an important parameter in our implementation as well, having predictive value over workload scalability regarding exploitation of ILP.

Chapter 5

Implementation Details

5.1 Introduction

A clear picture of the high-level and pipeline stalls on our set workloads is provided in Section 5.2. Motivated by the emerging diversity regarding these stalls, we explore and characterize different clusters of applications according to their respective structural bottlenecks that hinder performance, in Section 5.3. We also determine specific runtime characteristics that are correlated with improvement in LOOG. This characterization will accompany and direct further workload analysis throughout this thesis.
Sections 5.5 and 5.8.8 provide an overview of the power modelling methods utilized.
In Section 5.6 back-end structures directly related to LOOG are right-sized for the Volta V100 architecture, while in Section 5.7, a bottleneck analysis is performed regarding front-end components indirectly related to LOOG and they are optimally configured. A novel Instruction Buffer partitioning reconfiguration controller is presented and evaluated.
In Section 5.8, the reconfigurable GPU architecure with Out-of-Order scalability is presented in its various forms.
Finally, in 5.9, a brief speculation on other architectural axes of reconfiguration takes place, motivated by the analysis in Section 5.3.

5.2 Workload stalls analysis

Motivated by the workload performance analysis that birthed LOOG, we study stalls that cause performance deterioration at various levels of the GPU, for the set of workloads and the architecture we selected.
This analysis of stalls should give us a broad sense about both the diverse features of workloads that can be exploited from an architecture reconfiguration standpoint as well as those that are singificantly correlated with impovement in LOOG. As described in previous sections, the GPU frequently accesses memory and it hides memory stalls by context switching between concurrently executed warps. To get a broad sense of the bottleneck in the execution of each kernel, we use performance counters from GPGPU-sim that track stall events

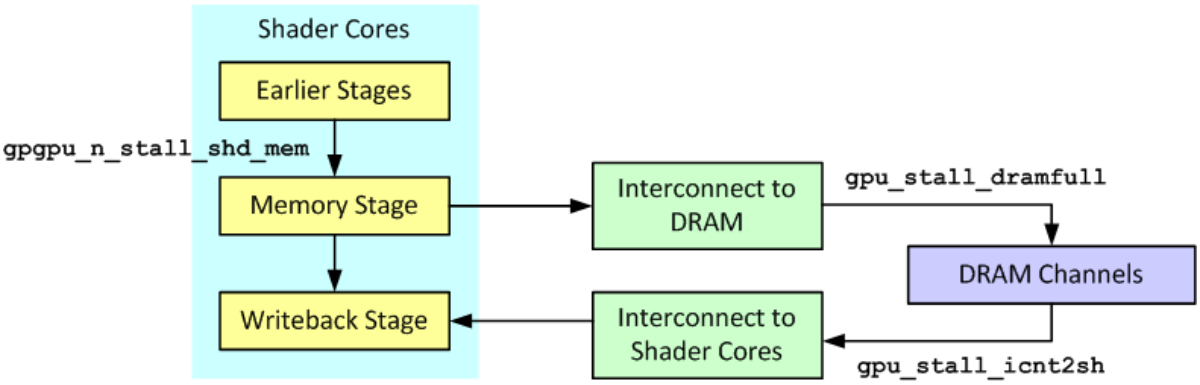


FIGURE 5.1: Memory request flow in GPGPU-sim

at different high-level parts of the GPU. As shown in Figure 5.1, the following memory stall metrics are recorded:

- Shader-memory stalls

  Shared memory bank conflicts

  Non-coalesced memory accesses

  Serialized constant memory accesses

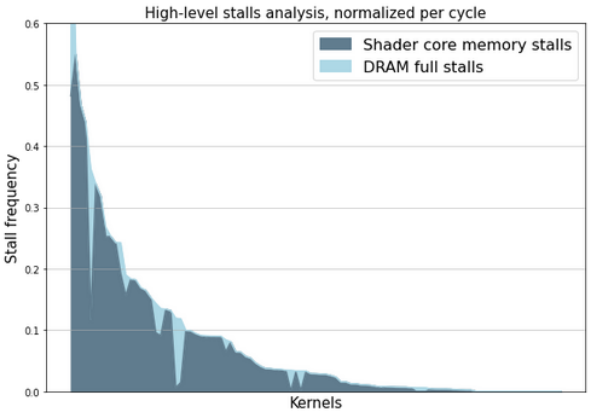
- DRAM-full stalls, due to unserviced DRAM requests

- Interconnect-to-shader stalls

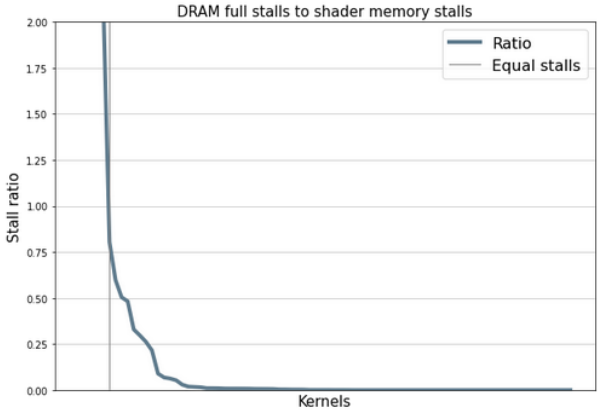
Interconnect to shader stalls occur when the writeback stage is stalled by other types of Functional Units, they represent less than 1% of total stalls and will therefore not be included. As depicted in Figure 5.2a, total stalls on the GPU, normalized over total cycles seem to follow the Pareto distribution, with most kernels having very few stalls overall in this level of analysis and DRAM stalls being scarce comparatively. Moreover, in Figure 5.2b is is seen that only 12% of kernels stall in DRAM more than shader core pipeline. Evidently, warp coalescing and bank conflict stalls amount for a significant number of cycles only in a select few number of kernels simulated. This realization motivates a closer core-level examination of stalls, taking workload diversity into consideration.

As depicted in Figure 5.3a, shader core stalls follow the expected similar distribution with high-level stalls. In Figure 5.3b, it is seen that for the vast majority of GPGPU kernels simulated, the maximum amount of available warps slots is under-occupied, which prevents the GPU from hiding the aforementioned stalls with context switching between active warps. In the core level, issue stalls can be divided into:

- Idle or control hazard stalls, representing no active warps or instruction replay due to taken branches (which is resolved in the Issue stage by comparing the current PC to the top of the SIMT stack)

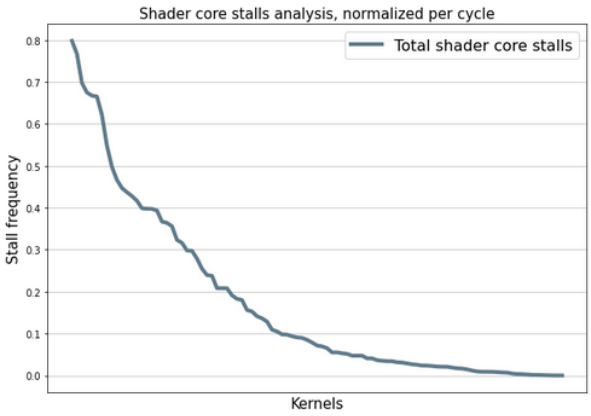


(A) Bottleneck analysis for kernels simulated. Half of the kernels run stall on GPU-level for less than 3% of total cycles.

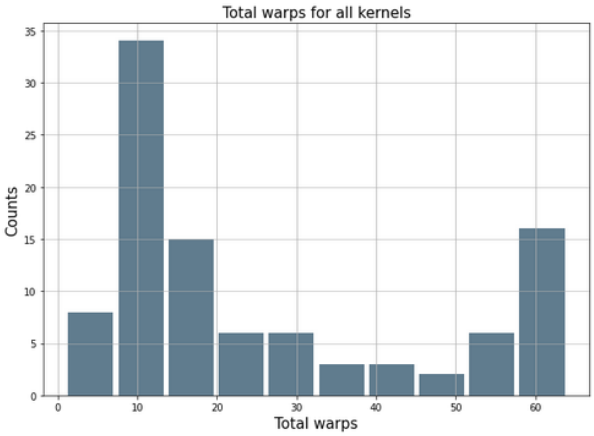


(B) Ratio of DRAM over shader memory stalls. 10% of kernels do not stall on this level. Only 12% stall on the DRAM more than shader memory.

FIGURE 5.2: High-level stalls analysis



(A) Shader core stalls analysis

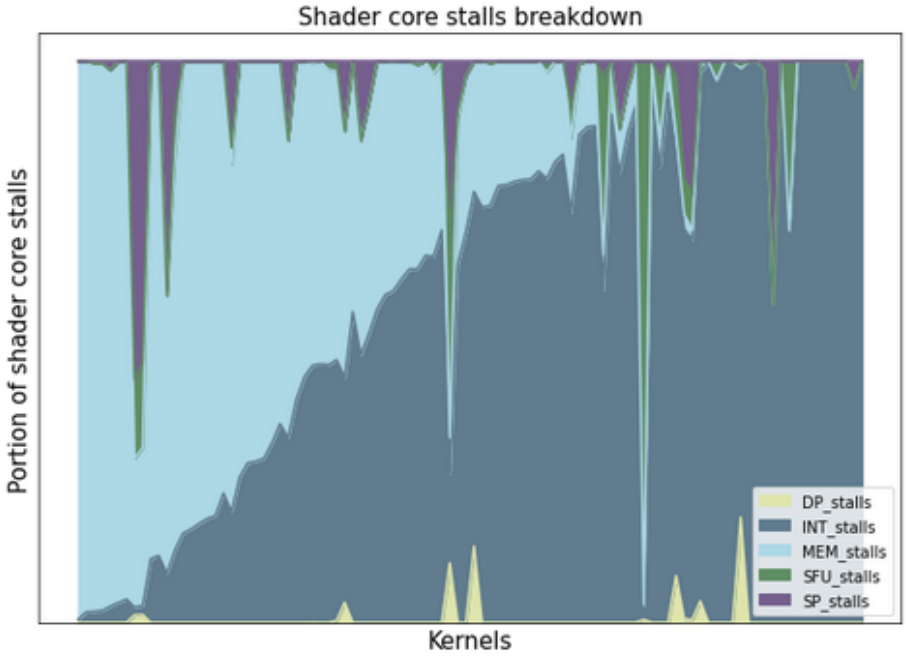


(B) Warp count histogram

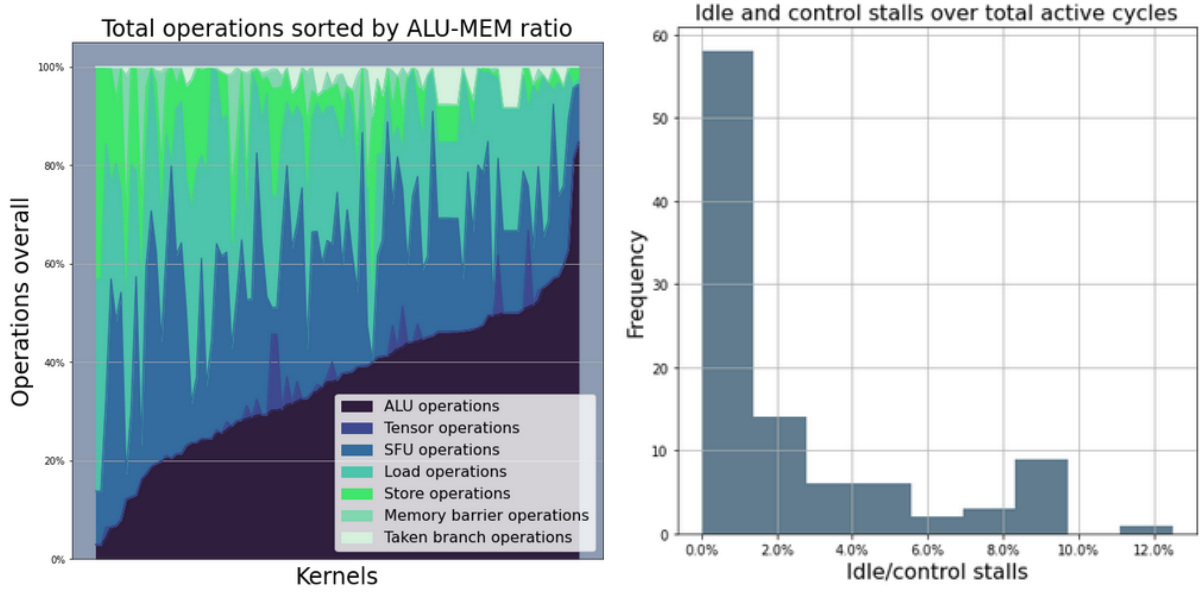
FIGURE 5.3: Most of the GPGPU kernels simulated do not fully occupy their maximum concurrent warp entries, producing shader core stalls

- Scoreboard conflict stalls, due to RAW and WAW hazards
- Stalled pipeline due to stalled backend.

Idle or control stalls during active issue cycles (idle initialization cycles during kernel offloading to the GPU are accounted for by the simulator but not included) represent a median of 0.91% and an average of 2.3% over total active cycles, as seen in Figure 5.5b Scoreboard conflict stalls are mainly caused by dependencies on long latency memory operations, while pipeline stalls only occur when the Execution Units are stalled. Therefore all issue stalls are essentially caused by functional unit stalls, whose distribution is depicted in Figure 5.4a. Evidently, for the GPGPU kernels we used, single and double precision FP stalls and SFU stalls represent a significant amount of overall stalls only for a small subset of specialized kernels.



(A) SM backend stalls breakdown



(A) Total operations breakdown

(B) Idle and control stalls

FIGURE 5.5: Idle and control stalls represent a miniscule amount of pipeline stalls. Issue stalls are mainly dependent on various Execution Unit stalls

It is also evident that INT and MEM stalls, representing almost all of the stalls for most kernels are, as expected, negatively correlated to one another, with their ratio distribution being almost uniform, except for a significant class of kernels (about 25%) that exhibit virtually no memory stalls on the pipeline compared to other kernels. However, as seen in Figure 5.5a, even these kernels have a significant amount of memory operations overall, therefore all their memory stalls happen in DRAM.

	INT_stalls	MEM_stalls	SP_stalls	SFU_stalls	DP_stalls	Total
50th	3.24%	1.54%	0.00%	0.00%	0.00%	7.15%
60th	4.97%	3.16%	0.00%	0.00%	0.00%	12.46%
70th	9.82%	4.72%	0.02%	0.00%	0.00%	20.81%
80th	17.57%	8.39%	0.12%	0.06%	0.00%	35.91%
90th	30.56%	17.48%	0.94%	0.45%	0.04%	45.08%

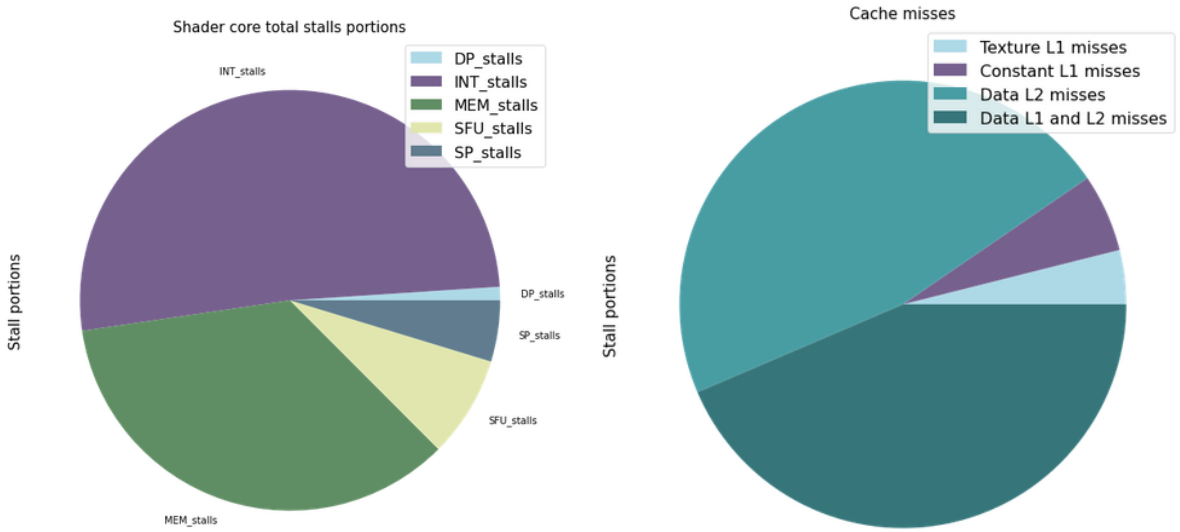
TABLE 5.1: Stall distribution for total stall percentiles. Values do not sum up to total, as percentiles are calculated separately for each type

Table 5.1 provides a more detailed view of stall distribution over total stall percentiles. It can be seen that memory stalls have a steeper distribution, with their relative values increasing faster as percentiles rise.

We conclude that significant memory stalls are present in all kernels, account for nearly all issue stalls due to RAW and WAW hazards (scoreboard collisions) and cannot be effectively addressed by the instruction reordering that takes place in LOOG. In Figure 5.6a, a detailed breakdown of all types of dispatch stalls per Execution Unit (EXU) pipeline stalls is provided. Note that FP and SFU stalls occupy a bigger portion of the total even for the 90th percentile seen in Table 5.1, due to the fact that they are concentrated in a small minority of kernels less than 10% as seen in Figure 5.4a and Figure 5.5a. An analysis of the shader core cache stalls causing memory stalls is seen in Figure 5.6b. Evidently, half of the instructions that miss on the L1 Data cache also miss in the off-chip, per-memory-partition L2. Since the stalls produced by these misses cannot be addressed by instruction reordering and alterations in scheduling, this motivates the idea of potentially increasing cache performance and studying the cache size / Area-Power overhead tradeoff. Misses on other types of caches represent a small portion of the total, as expected, as is evident in Figure 5.7.

5.3 Workload characterization and exploitable ILP analysis

In continuation of the previous section, given the evident and potentially highly exploitable workload diversity we set to categorize GPU applications guided by concepts much like those described in "Whole Picture Analysis" [47]. The purpose of this study is to both determine emerging classes of GPGPU kernels that would dictate respective classes of hardware configuration, as well as correlate said classes to performance improvement in OOO, Long Instruction Window execution schemes such as LOOG [12, 13]. In that direction, we collected runtime statistics from GPGPU-Sim on the workloads mentioned, over multiple configurations, all in the baseline in-order model of the simulator. As in "Whole Picture Analysis" [46, 47], statistics collection for all three levels of the analysis was facilitated by GPGPU-Sim and for our baseline Quadro GV100 GPU model as demonstrated below:



(A) Mean EXU stall distribution over all kernels

(B) Cache miss breakdown

FIGURE 5.6: Mean EXU stall distribution over kernels (kernels are equally weighted) and breakdown of cache misses causing memory stalls

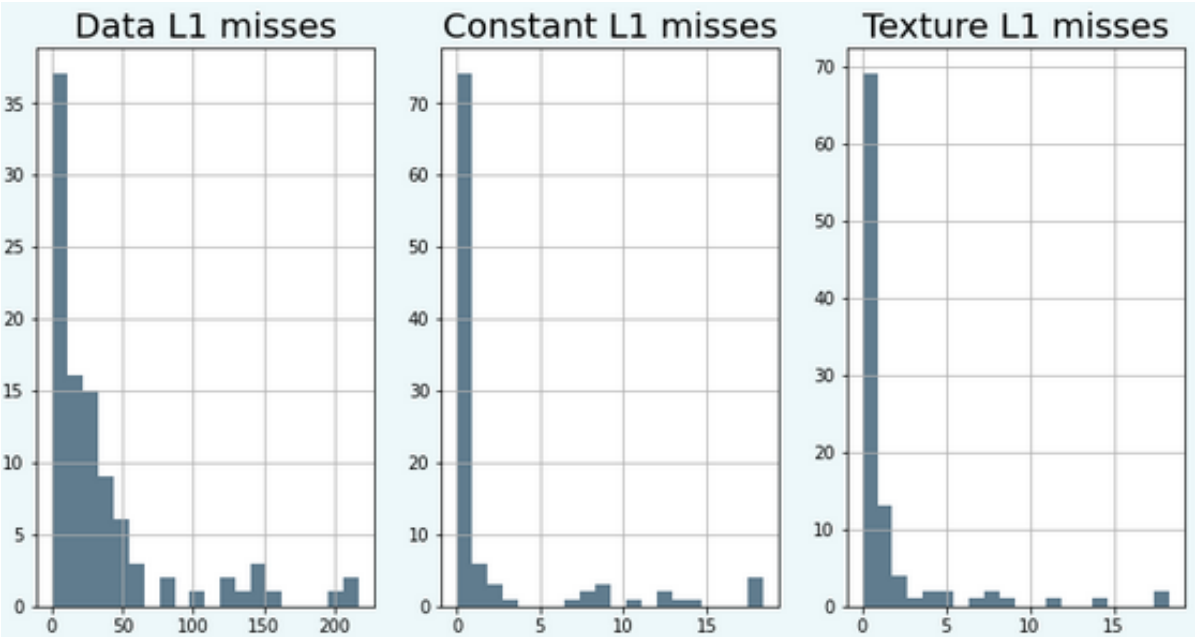


FIGURE 5.7: Miss distribution for each type of cache, normalized over total warp instructions (theoretical maximum of $32 \cdot 100$).

IR-level In the WPC paper [47], static and dynamic instruction stream statistics were collected on a perfect processor model with infinite registers, to track instruction mix, locality metrics and branch behaviors. In our implementation, such statistics were collected with GPGPU-Sim regardless of uarch configuration provided Hence we collect *instruction locality, instruction and data locality and branch metrics.*

ISA-level Microarchitecture-agnostic characteristics are determined by execution on a perfect

processor with a perfect cache model but with no pipeline model. In our implementation, such a model was simulated by configuring caches with minimal latencies, functional units with exaggerated latencies, and tracking data hazards via scoreboard check collisions in the Issue stage. Thus, the characteristics of the previous level along with *perfect cache behavior, parallelism and instruction dependencies* were collected

uArch-level This level contains all the architectural component behaviors manifested during execution on a specific microarchitecture. This stage was implemented by averaging out statistics collected over multiple cache and functional unit configurations, as it was theorized that these were the axes we should reconfigure upon based on workload diversity. Hence, additional statistics we collected regarded *pipeline and cache behavior*. Our analysis was mainly based on pipeline stalls, cache hit ratio, Missed per Thousand Instructions (MPKI) and memory bandwidths.

Values from all three architectural levels were included in our final analysis. They were normalized by the appropriate cumulative values (i.e. Statistics referring to scalar thread activity such as total ALU operations were normalized by total thread instructions, statistics referring to memory or RF accesses were normalized by total warp instructions and statistics referring to cycles such as various types of stalls were normalized by total cycles). Dimensionality reduction was applied on the data to find a smaller set of axes that explains at least 95% of the cumulative variance of the dataset. Features were sorted based on their coefficient in the PCA eigenvector of each principal component multiplied by the eigenvalues themselves. The most significant features producing workload diversity are presented in Figure 5.9.

$$importance(feature) = \sum_{x \in S} \lambda(x) \cdot coeff(x, feature) \tag{5.1}$$

S: *The set of all eigenvectors*
$\lambda(x)$: *The eigenvalue for eigenvector x*
23 of the PCA components explained 95% of the Cumulative Variance Ratio (CVR), as seen in Figure 5.8a. We extracted as many features, since importance scores did not seem to produce any significant local drops, as depicted in Figure 5.8c. This number of selected features explained 74% of the CVR. We opted for a cosine metric as relative and not absolute feature values needed to be interpreted. Kmeans did not yield good results in either metric. For hierarchical with a cosine metric, silhouette scores were locally maximized for a value of 5, as depicted in Figure 5.8b and we selected as many clusters, seen in Figure 5.11 . The clusters produced, as well as saturated LOOG IPC improvements and GPU utilizations are depicted in Figure 5.10.
It is noted that Utilization and Saturated LOOG δIPC were not among the features that the clustering and PCA, was applied upon, rather they were added succeeding it, to aid our

analysis. Saturated LOOG improvement refers to the maximum achievable performance improvement in LOOG configuration, as will be explained in later sections. Early in the examination of GPU utilization and IPC optimization it was clear that the axes of OOO, cache and Execution unit scaling could explain all types of stalls (scoreboard stalls, pipeline stalls and dispatch stalls respectively) and most of the workload diversity. Given this diversity, we studied the potential for reconfiguration of the microarchitecture on these axes, to tailor it to workloads on runtime. Clusters produced, seen in Figure 5.10 were retrospectively interpreted with regards to their main performance bottleneck as follows:

- *DP-bound, high utilization*, significantly stalling on the INT, MEM and DP Execution Units and exhibiting low cache and DRAM accesses (high miss rate is due to the few accesses and is therefore irrelevant). The MEM stalls paired with low memory accesses overall imply separate memory and compute intensive phases. Therefore, it comprises kernels that **mainly perform DP computation**. As a compute-intensive cluster, it stalls frequently due to control hazards, as also seen in Figure 5.5a. Owing to its high warp-count (achieved occupancy in our terminology [84]) it can maintain a high utilization and high average warp readiness. Its improvement in LOOG is therefore high, since it is not dependent upon memory instructions.

- *Cache-bound, low ILP*, having high Data stores and missing on Constant cache, due to parameter memory instructions as is elaborated on in section 5.8.2 (it is also characterized by a high kernel launch count). High CTA (thread blocks) are the direct cause of the increased parameter memory instructions, and because of the low warp count, pipeline stalls due to the subsequent scoreboard collisions cannot be hidden with context switching. It includes kernels that **read from constant memory, perform few computations and write on local and global and local memory** Due to constant memory not being backed by the L2, off-chip memory accesses cause interconnect stalls. The exploitable ILP of this cluster is low for the above reasons.

- *Shared memory-bound*, having high shared memory operations overall (the lowest latency among memory operations) and stalling on the SFU pipeleine. It includes kernels that **perform SFU computation on values read from the shared memory, using few active warps** but exhibit mediocre utilization and low exploitable ILP due to the low warp count.

- *Cache-bound, high ILP*, exhibiting frequent Data cache load accesses, with a low miss rate. This is paired with the relatively high RF read count, signifying the presence of compute phases. Given that dispatch stalls are only present in the MEM pipeline, compute phases are evenly spread among the ALU and FPU pipelines and do not cause congestion. Low scoreboard collisions signify the inclusion of kernels that **read data from the Data cache and perform arithmetic operations on it on separate compute phases**. All of the above produce exploitable ILP.

- *SP-bound* evidently having compute phases (as the high RF reads and ALU stalls imply) despite its frequent DRAM operations (given that the Data cache is accessed frequently with a low miss rate, these operations are due to evictions), without significant dependencies (as seen with only a few scoreboard collisions). This, again, implies its composition from kernels that **load data from global memory, perform SP operations on it and store it again in global memory**. High DRAM traffic is produced by L1 Data cache accesses that hit and produce evictions. High throughput due to compute phases paired with low dependencies equates to high exploitable ILP.

Summing up, in all our preliminary cluster analysis of the kernels provided, a shared memory cluster is formed, due to its distinguishing characteristics of high SFU utilization and low warp count causing relatively low utilization and mediocre exploitable ILP.

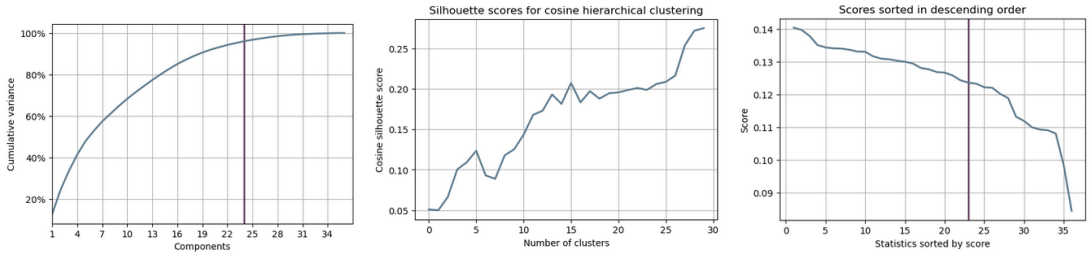
Two pairs of cache bound (memory-intensive) and FPU-bound (compute-intensive) clusters are formed. The distinction between cache bound kernels depends on types of operations and specific cache accesses.

Constant accesses paired with Data store instructions are related to high scoreboard collisions and relatively high utilization, due to a high instruction throughput paired with existing real dependencies. The dependencies are evidently real since the saturated LOOG ΔIPC is not significant.

High Data Load Read instructions are paired with low scoreboard collisions and low utilization but a high warp count and compute phases (as evident by RF reads). These characteristics provide high exploitable ILP.

FPU-bound clusters both have high utilization and low scoreboard collisions, providing exploitable ILP. The DP-bound cluster is less prone to cache utilization and DRAM accesses and has a high warp count that provides the highest instruction throughput among all clusters.

- Scoreboard collisions are frequently paired with the presence of memory instructions.
- RF reads are negatively correlated with memory instructions, providing a criterion for compute phases
- GPU utilization is higher in compute-intensive kernels and is negatively correlated with Data load instructions in memory-intensive kernels.
- For compute-intensive kernels, GPU utilization is higher in DP-bound kernels, and for memory intensive kernels it is higher in shared-memory-bound kernels.
- Exploitable ILP is high in compute-intensive kernels and memory-intensive kernels with local and global memory load instructions (that exhibit compute phases).
- It is, hence, mostly correlated with compute phases per kernel. RF reads are positively and scoreboard collisions negatively correlated to it.



(A) Cumulative variance ratio explained by components (B) Silhouette scores per number of clusters (C) Sorted importance scores of extracted features

FIGURE 5.8: Kernels feature extraction and clustering

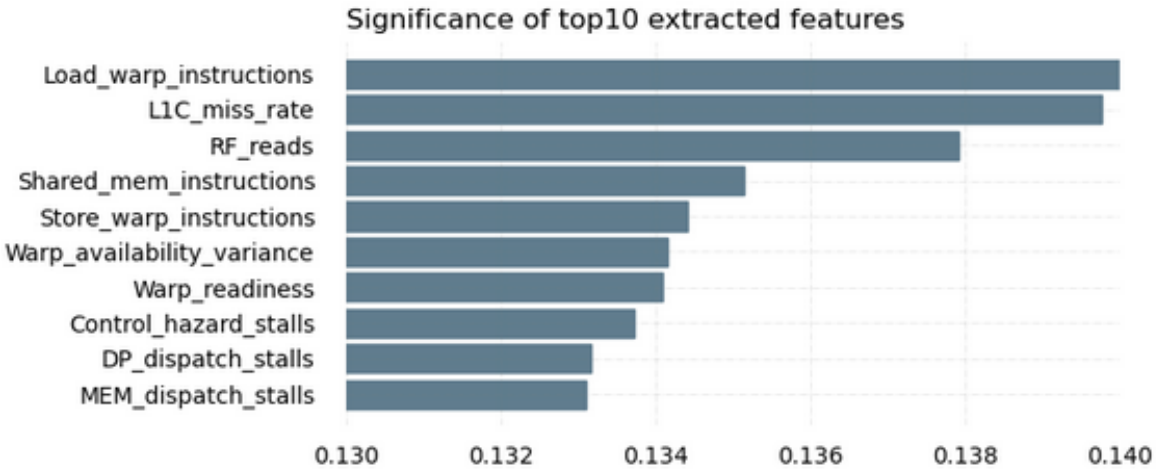


FIGURE 5.9: Relative significance of top extracted features

- From the Cache bound, low utilization and the Shared memory-bound clusters, we can see that exploitable ILP is correlated to higher utilization.

5.3.1 Rodinia - Back propagation

The back propagation benchmark belongs in the pattern recognition algorithm domain and its computation and memory access pattern matches that of the unstructured grid dwarf.[2] It is responsible for training the weights on a layered neural network. It comprises two phases; In the Forward phase, the activation is propagated from the input to the output layer. In the Backward phase, the error between the actual and requested output value is used to update the bias and weights of the layers leading up to the output.[1] Interestingly, the Forward phase kernel of Rodinia 2.0 back propagation belongs to the *Shared memory-bound cluster* , while the weight adjustment kernel belongs to the *DP-bound, high utilization cluster*. Note that even though the latter is actually INT-bound instead of DP-bound, such kernels are grouped together due to similar characteristics overall. Following is a piece of the relevant host code (.cu file):

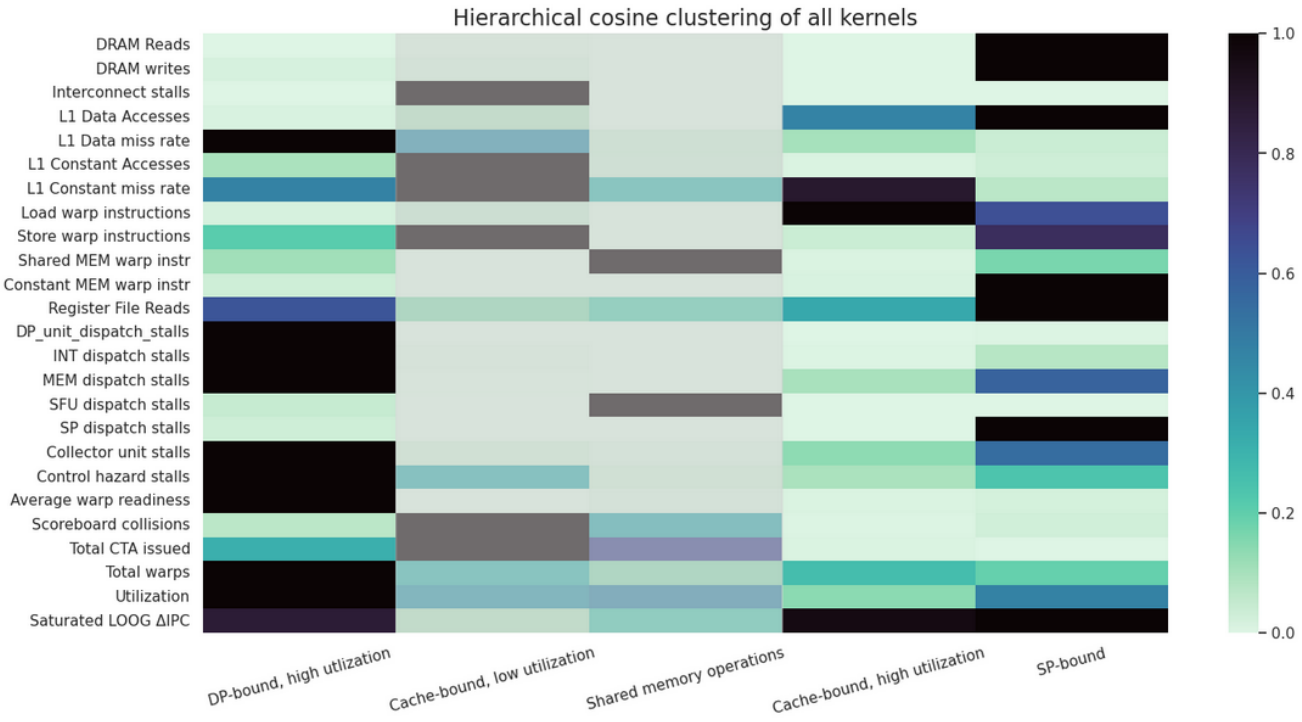


FIGURE 5.10: Relative feature values of clusters produced. LOOG DeltaIPC and Utilization were subsequently added

```
cudaMemcpy(input_cuda, net->input_units, (in + 1) * sizeof(float),
  cudaMemcpyHostToDevice);
cudaMemcpy(input_hidden_cuda, input_weights_one_dim, (in + 1) * (
 hid + 1) * sizeof(float), cudaMemcpyHostToDevice);

bpnn_layerforward_CUDA<<< grid, threads >>>(input_cuda,
                                            output_hidden_cuda,
                      input_hidden_cuda,
                      hidden_partial_sum,
                      in,
                      hid);
cudaThreadSynchronize();

cudaError_t error = cudaGetLastError();
if (error != cudaSuccess) {
  printf("bpnn kernel error: %s\n", cudaGetErrorString(error));
  exit(EXIT_FAILURE);
}
cudaMemcpy(partial_sum, hidden_partial_sum, num_blocks * WIDTH *
 sizeof(float), cudaMemcpyDeviceToHost);
```

LISTING 5.1: Rodinia Backpropagation host code

Figure 5.11: Detailed clustering visualization of the kernels. Note that kernels from the same application may diverge significantly.

Lines 1 and 2 are responsible for host-to-device data transfer of the bias of the input layer and weights between the input and hidden layer respectively. Memory for destination pointers has been allocated on the device using cudaMalloc.
On line 4, the device function responsible for the forward phase is called for execution on the GPU. Its operands are destination, source, allocation size, and memory copy type respectively. It is important to note that data transfers from host to device only take place on contiguous memory spaces using "cudaMalloc()". Therefore the 2d weights matrix must be linearized (variable "input_weights_one_dim", line 2) to be used as a source operand. Creating a 2d matrix on device DRAM using an array of pointers would require multiple calls to cudaMalloc(), and imply a non-contiguous memory space, reducing data locality. On line 11, the function called blocks the host thread execution until all previously issued commands running on all active streams on the device have finished. It returns an error if any of the preceding tasks has failed. It is noted that this function is deprecated and appropriately renamed in CUDA 12.0 as "cudaDeviceSynchronize()", since it serves the purpose of device and not thread-level synchronization. On line 19, the partial sum calculated from the forward propagation function executed on the device, is copied back to host memory [103, 21, 104] .
Following is the forward phase device function from the auxiliary .cu file:

```
1  __global__ void
2  bpnn_layerforward_CUDA(float *input_cuda,
3                         float *output_hidden_cuda,
4                float *input_hidden_cuda,
5                float *hidden_partial_sum,
6                int in,
7                int hid)
8  {
9     int by = blockIdx.y;
10    int tx = threadIdx.x;
11    int ty = threadIdx.y;
12    int index =  ( hid + 1 ) * HEIGHT * by + ( hid + 1 ) * ty + tx +
      1 + ( hid + 1 ) ;
13    int index_in = HEIGHT * by + ty + 1;
14
15    __shared__ float input_node[HEIGHT];
16    __shared__ float weight_matrix[HEIGHT][WIDTH];
17
18    if ( tx == 0 )
19    input_node[ty] = input_cuda[index_in] ;
20    __syncthreads();
21    weight_matrix[ty][tx] = input_hidden_cuda[index];
22    __syncthreads();
```

```
23    weight_matrix[ty][tx] = weight_matrix[ty][tx] * input_node[ty];
24    __syncthreads();
25
26    for ( int i = 1 ; i <= __log2f(HEIGHT) ; i++){
27      int power_two = __powf(2, i);
28      if( ty % power_two == 0 )
29      weight_matrix[ty][tx] = weight_matrix[ty][tx] + weight_matrix[
    ty + power_two/2][tx];
30      __syncthreads();
31    }
32
33    input_hidden_cuda[index] = weight_matrix[ty][tx];
34    __syncthreads();
35
36    if ( tx == 0 ) {
37      hidden_partial_sum[by * hid + ty] = weight_matrix[tx][ty];
38    }
39
40 }
```

Listing 5.2: Rodinia Bacpropagation Layerforward kernel

As the keyword "__global__" signifies, this function is called from the host code as shown above. In lines 9-11 some thread indexing variables are set from kernel launch parameters stored by default in the shared memory. In lines 15,16 some shared memory matrices are defined. Host data that was transferred to device DRAM will be copied there, to be used by all threads in a block. The linearized weights matrix is wrapped as a 2d matrix in the shared memory. At each stage of the computation, before the output is set at line 37, the function __syncthreads() is used, which poses a barrier to all threads within a block and synchronizes them. Comparing the kernels corresponding to the forward and the weight adjustment phases, it is clear why they belong to the respective clusters.
Following is the weight adjustment phase function:

```
1  __global__ void bpnn_adjust_weights_cuda(float * delta,
2                      int hid,
3                      float * ly,
4                      int in,
5                      float * w,
6                      float * oldw)
7  {
8    int by = blockIdx.y;
```

```
9     int tx = threadIdx.x;
10    int ty = threadIdx.y;
11    int index =  ( hid + 1 ) * HEIGHT * by + ( hid + 1 ) * ty + tx +
      1 + ( hid + 1 ) ;
12    int index_y = HEIGHT * by + ty + 1;
13    int index_x = tx + 1;
14
15    w[index] += ((ETA * delta[index_x] * ly[index_y]) + (MOMENTUM *
      oldw[index]));
16    oldw[index] = ((ETA * delta[index_x] * ly[index_y]) + (MOMENTUM *
       oldw[index]));
17    __syncthreads();
18
19    if (ty == 0 && by ==0){
20    w[index_x] += ((ETA * delta[index_x]) + (MOMENTUM * oldw[index_x
      ]));
21    oldw[index_x] = ((ETA * delta[index_x]) + (MOMENTUM * oldw[
      index_x]));
22    }
23 }
```

Listing 5.3: Rodinia Bacpropagation Weights Adjustment kernel

Operations inside the loop in lines 26-31 of the forward phase kernel, which produces most of the kernel's dynamic instruction mix are compiled into shared memory operations in PTX, justifying the inclusion of the kernel to the Shared-memory-bound cluster. .

In the weights adjustment kernel, all kernel parameters dereference to global memory arrays requiring L1 Data cache accesses, therefore operations mainly consist of L1D reads. Since these are one-off reads as seen in lines 15,16,20,21, they cause a high L1 Data miss rate, which characterizes the DP-bound cluster, as seen in Figure 5.10 . Since the conditional in line 19 is executed just for a specific scalar thread, it likely causes frequent control hazards as also seen in Figure 5.10. In the microarchitecture level, warps containing this thread ($thread\_y == 0 \&\& block\_y == 0$) will always assume this branch as not taken [24]. During instruction Issue, the SIMT stack will be updated [24] and two entries, one with the thread active mask bit set and its complementary, will be added to it and executed sequentially [1] . Since the branch will always be assumed as not taken, half of the time the PC seen in the ibuffer entries to be issued will be different than that at the top of the SIMT stack, and control hazards will be produced.

Evidently, in each of the lines 15,16,20,21, an INT operation is produced in PTX for every global load, justifying the inclusion of this kernel in the DP-bound cluster and the INT stalls seen in Figure 5.10.

Since the weights adjustment kernel is compute-intensive and has a high GPU utilization, it

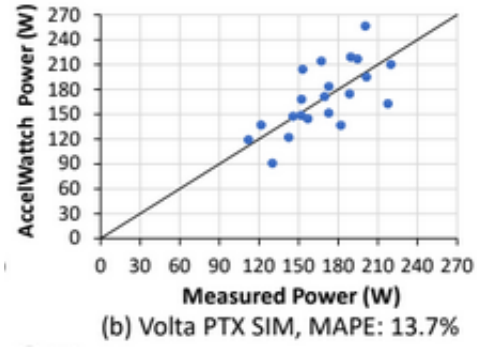


FIGURE 5.12: Accelwattch estimation accuracy [57]

possesses significant exploitable ILP.

5.4 Performance modelling

A detailed, yet slightly outdated (version 3.x) overview of the software model of GPGPU-Sim, the performance model of Accelsim used in our simulations can be found at [24].
The source code for all of our modifications on the performance model (GPGPU-Sim 4.1.0) as well as the power and area model (Accelwattch) of Accelsim, including the baseline configuration files and design space exploration scripts can be found at the Appendix A.1.

5.5 Power modelling

In this section, the method to obtain the configurable LOOG power model is described, with changes implemented in Accelwattch [57], as well as the baseline configuration file used to extrapolate configuration options from.

5.5.1 Changes implemented

Components added due to the transition from the baseline model to LOOG, namely RRS RAT and LSQ were modelled as SRAM arrays. Existing components were modified with extra entries or extra throughput where needed (Decoder, Instruction Issue Window, Collector Unit entries). Extra tags were added to the relevant interconnects corresponding to Collector Unit ID. Hence, scaling-up Collector Units intra-LOOG causes linear Area scaling, while Power scaling is the sum of a linear and a low-coefficient quadratic function.

5.5.2 Leakage and Dynamic power modelling

The manufacturing process used in GV100 is the TSMC 12 nm FFN (FinFET NVIDIA), while Accelwattch modelling of the same architecture is tuned with a 23 nm node technology, roughly corresponding to two generations before as seen in Table 5.2. The leakage power to

Architecture	Min Node
Tesla (2007)	90nm
Tesla (2008)	55nm
Fermi (2010)	40nm
Kepler (2012)	28nm
Maxwell (2014)	28nm
Pascal (2016)	16nm
Volta (2017)	12nm
Turing (2018)	12nm
Ampere (2020)	8nm
Lovelace (2022)	5nm

TABLE 5.2: Nvidia GPU architectures and their minimum node technologies [105, 74, 106]

dynamic power ratio varies significantly with technology node scaling. As it scales down, the dynamic power consumed by CMOS circuits tends to only slightly increase while the leakage power aggressively increases (inversely proportional to gate length) [107]. This is due to the increase in the sub-threshold leakage current and the gate leakage current caused by the thinning of the gate oxide, as well as the junction leakage current. This difference in configuration is expected to skew our results with regards to dynamic to leakage power ratio (approximately by a factor of 2, according to [107]). However, since the sum of these components is used to tune the baseline Accelwattch configuration file we modify, and since the power gating we perform assumes insignificant increases in dynamic power when the structures are not gated, and complete eradication of both power components when they are, this does not affect our analysis.

5.5.3 Accelwattch configuration

The Accelwattch PTX simulator yields satisfactory results when compared against hardware power measurements on a variety of benchmarks suite ran for the NVIDIA Volta architecture, as seen in Figure 5.12. In the aforementioned paper [57], it is further demonstrated that Accelwattch can be reliably utilized for design space exploration, as it can provide accurate power models even for the Pascal and Turing microarchitectures. Accelwattch uses intricate performance counters provided by the performance simulator to model switching power and account for DVFS and aggressive power gating. In the scope of this thesis, maximum power dissipation statistics will be used as provided by Gpuwattch (based on McPAT and Cacti) in

a relative analysis compared to baseline. Power estimations are bound to be even more accurate when it comes to individual components.
The Accelwattch Volta configuration file has been tuned according to power correlations provided by running microbenchmarks targeting individual architectural components. Therefore, power estimations for each component that is dynamically configured come down to McPat calculations that exhibit a linear dependency with the relative configuration size coefficients that are applied. For example, configuring Collector Unit number only affects the size of the crossbar input buffer connecting them to the Register File, apart from the structures themselves. Total crossbar area and power dissipation is proportional to both input and output buffer size. Configuring functional units similarly affects the crossbar size connecting them to the operand collector, as well as the number of lanes withing the SM. Cache sizes similarly exhibit a linear dependency with power dissipation. In conclusion, the fine tuned per-component power dissipation estimations combined with the accurate whole-processor power model provide a decent substrate for our custom configurations to extrapolate upon and compare to baseline [57].

5.6 Right-sizing LOOG on NVIDIA Quadro GV100

Following the implementation of LOOG in version 4.1.0 of GPGPU-sim [23], a reassessment of its right-sizing was deemed necessary, since the original implementation was studied on the GeForce GTX1080Ti model [48, 14], a gaming GPU, implementing Pascal, the previous architecture generation to Volta. Since our new baseline GPU platform is a workstation GPU, LOOG's behavior is expected to differ regarding right-sizing of the Register Renaming Stack [12, 13] , Operand Collector and Instruction Window.

5.6.1 Register Renaming Stack

As elaborated on in chapter 3 , the RRS is a structure designed to alleviate Collector Unit congestion by enabling their deallocation right after instruction dispatch instead of the Write-back stage [14]. As seen in Table 5.3, the Power and Area overhead of RRS scaling is minimal, while the average performance per kernel only significantly improves from 32 to 64 RRS. In Figure 5.13a, the maximum number of RRS entries used over all kernels in the evaluation is presented. The absolute maximum is 146 and the absolute minimum is 41. In Figure 5.13b, the average RRS occupancy over all kernels is presented with RRS configured at 64. Evidently, a minority (30%) of kernels has an average RRS occupancy for which 64 RRS do not suffice. Values close to 64 in Figure 5.13b are produced by high bandwidth throughput, presumably in compute-intensive kernels. Despite such high values, RRS size is not a bottleneck for the respective kernels, since the backend is not stalled and it only prevents Long I-window instruction reordering. The average available I-window length per warp for an RRS configuration of 64 is depicted in Figure 5.13c (average value of 2.93). It is calculated by dividing the

average number of RRS entries used with the number of total warp slots occupied. Finally, in Figure 5.13d, cycles with no CUs allocated (CU stalls) are depicted for an RRS configuration of 64. Only 32% of kernels have their CU stalled more than 25% of the time. Even such a high CU stall rate, as indicated, does not equate to backend stalls and reduced throughput.

RRS size	64	128	256
Power overhead	0.07	0.018%	0.047%
Area overhead	0.002%	0.004%	0.008%
DeltaIPC	7.52%	7.67%	7.67%

TABLE 5.3: Overheads and performance improvement compared to a 32 RRS configuration

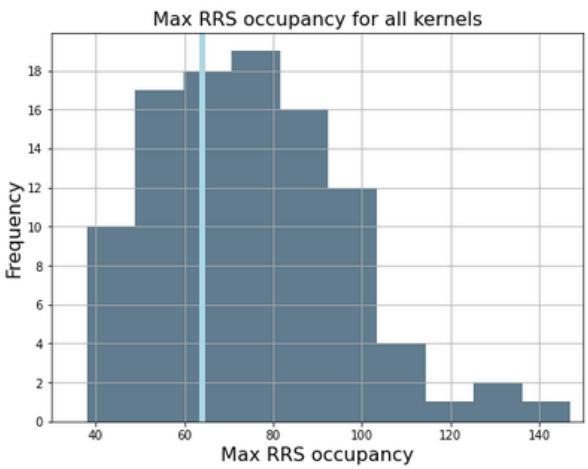
Results are similar to the initial implementation of LOOG [14] and since no significant improvement is provided beyond 64 RRS, this is the configuration used in all subsequent modelling in this thesis.

5.6.2 Instruction Window

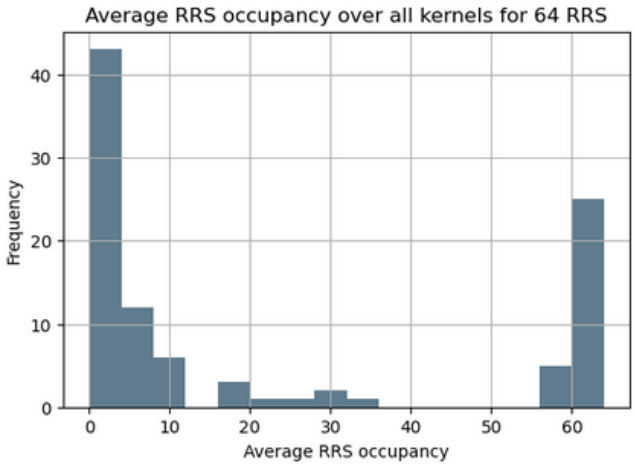
Power and Area overheads for Instruction Window scaling with a baseline of $IWindow = 1$ (IWindow values in the table are normalized to 1 instruction Fetched, Decoded and Issued per cycle per SM processing block) are presented in Table 5.4. It is noted that as explained in the Power modelling section, the overheads presented here contain estimates of the peak dynamic power. It is worth noting that the leakage power overhead that arises from the additional structures is more closely approximated by the Area overheads whose model is precise. Since the total dynamic power for a given number of fetched, decoded and scheduled instructions is approximately constant (the Issue scheduler does not implement any sophisticated policies and considers each scheduled instruction once), the actual total power overhead curve is, hence, expected to more closely follow the Area overhead curve. As elaborated on in section decoder throughput, in the new architecture with per-SM processing blocks [49] (sub-cores[36, 24]) it is never sensible to scale-up the Instruction Window, rather it can be scaled down to $\frac{1\_insn}{processing\_block \cdot cycle}$ without significant performance deterioration. As indicated in section depth scheduling, larger instruction windows can provide more efficient instruction depth scheduling (it is demonstrated that prioritizing the instructions of a given warp with Depth-First Scheduling is preferable to switching warps with Breadth-First Scheduling in LOOG). This scheduling policy, however, can be emulated over many cycles without using a Longer Instruction Window and with the addition of minor components.

5.6.3 Operand Collector

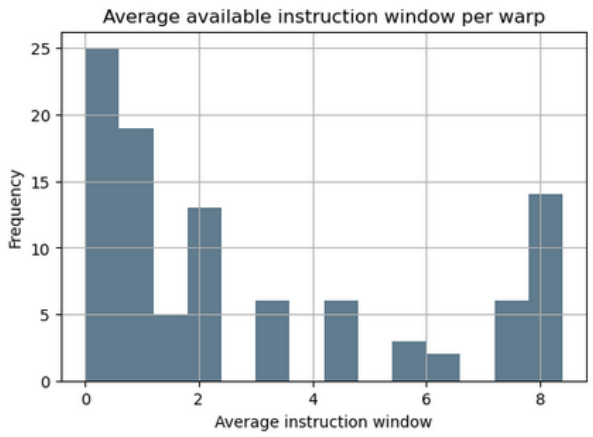
Early in examining the behavior of the LOOG execution scheme, it was evident that the Collector Units, serving as the Reservation Stations of the Tomasulo algorithm [15] are its



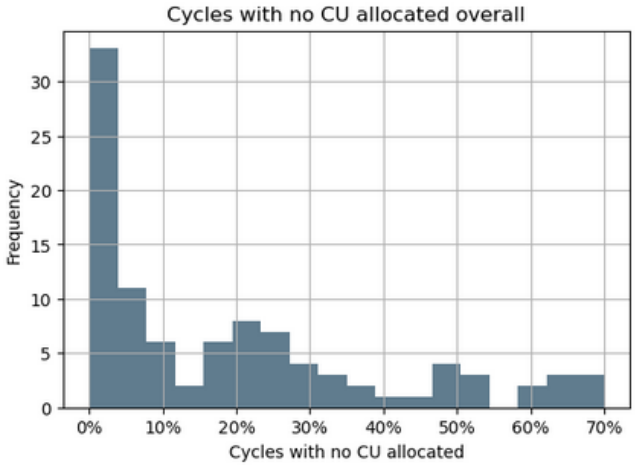
(A) Maximum RRS entries used for all kernels used



(B) Average RRS occupancy for all kernels used



(C) Average instruction window per warp with 64 RRS



(D) Cycles with no CU allocated overall

FIGURE 5.13: Right-sizing of the RRS

IWindow	2	4	6	8	10	12	14	16
Power	4.79%	10.68%	17.58%	24.98%	34.06%	47.01%	60.47%	74.77%
Area	0.22%	0.95%	1.80%	2.71%	3.70%	4.85%	5.99%	7.23%

TABLE 5.4: Scaling Instruction Window (Fetch - Decode - Issue scheduler throughput) overheads

most critical part, both in terms of performance gains as well as Power and Area overheads. The average speedup for all kernels tested as well as Area and Power overheads are presented in Table 5.5. Evidently, the IPC improvement from LOOG saturates beyond 48 CUs for the average kernel.

As no improvement is provided beyond 48 CUs for any workloads, this is the design limit for all our further testing.

We define saturated LOOG improvement as the maximum percentile performance improvement achievable from baseline. It is approximately equal to the improvement at 48 CUs and precisely equal to the improvement at 64 CUs, save for some data anomalies explained in the scheduling section. The histogram of saturated improvement over all kernels run is depicted in Figure 5.14. The improvement curves for CU scaling per saturated improvement kernel percentiles from 50th to 90th are shown in Figure 5.15a. As seen in both of the above figures, the distribution of IPC improvement at each scaling step is wide, with the ΔIPC between percentiles increasing at higher percentiles. We arbitrarily define the "LOOG-sensitive" class as the class of applications whose saturated LOOG improvement is more than 100%.

In Figure 5.15a, the upper two percentile curves, showing marked improvement compared to the rest of the workloads approximately correspond to the LOOG-sensitive kernels.

The LOOG-sensitive class represents 22.0% over all kernels and 9.2% of all kernel launches. We observe, therefore, a clear bias of multiple-launch kernels towards LOOG-insensitivity. This analysis will prove relevant in section 2.2 . Intra-kernel loog-sensitivity is consistent as depicted in Table 5.6 , with the exception of 1 launch. A small minority of kernel launches not belonging to LOOG-sensitive kernels surpass the LOOG-sensitivity threshold. These niche workloads are considered outliers as they belong to only two kernels. We deduce that LOOG sensitivity highly varies inter-kernel and is mostly consistent intra-kernel, nevertheless, with existing exceptions. The observation of said high variance is closely tied to determining the optimal configurations for each class of applications based on different figures of merit (ΔIPC saturation, Power-Delay Product, Energy-Delay Product), as tentatively examined in this subsection, and more closely in section 5.8 . With the realization that optimal configurations vary in a similar fashion, we set to design a reconfiguration mechanism at the granularity seen in Table 5.5 in section 5.8 , in order to optimize the aforementioned figures of merit for each workload at runtime, focusing on performance or energy efficiency.

In Figure 5.15b, Area and Power overheads are co-ploted with the IPC improvement per LOOG sensitivity class. Evidently, the median kernel with regards to LOOG-sensitivity is

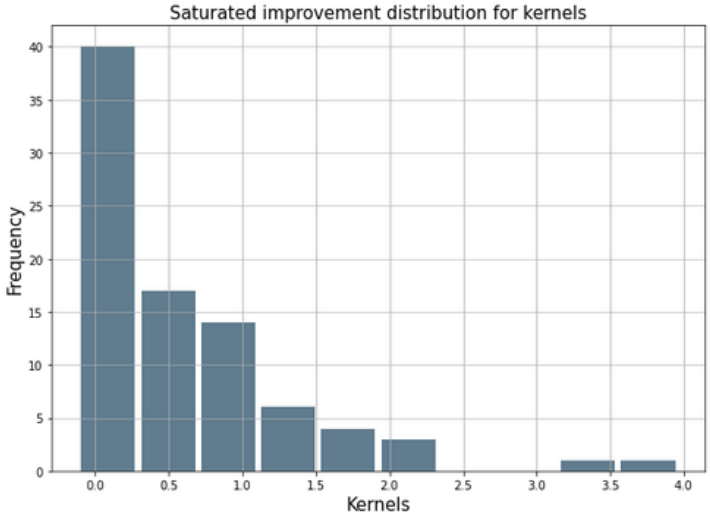


FIGURE 5.14: Histogram of saturated IPC improvement on LOOG

essentially LOOG-insensitive. Mean IPC improvement for LOOG sensitive kernels surpasses 160%.

Collector Units	8_CUs	16_CUs	24_CUs	32_CUs	40_CUs	48_CUs	56_CUs	64_CUs
Area overhead	3.33%	6.80%	10.26%	13.72%	17.19%	20.65%	24.12%	27.58%
Power overhead	6.50%	9.90%	13.64%	17.72%	22.13%	26.87%	31.95%	37.37%
Average DeltaIPC	22.29%	40.76%	48.57%	52%	59.66%	61.14%	61.16%	61.16%

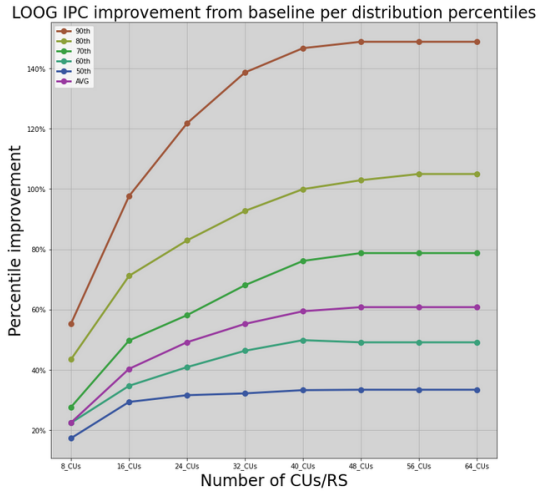
TABLE 5.5: Collector Unit scaling performance improvement and Area, Power overheads

	LOOG-sensitive kernel	non LOOG-sensitive kernel
LOOG-sensitive launch	3,78%	5,45%
non LOOG-sensitive launch	0,15%	90,62%

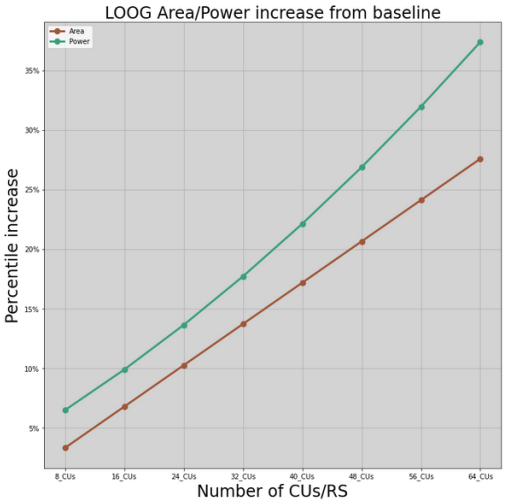
TABLE 5.6: LOOG-sensitivity distribution over kernels and kernel launches

Area and Power scaling with CUs are depicted in Figure 5.15b . As expected, Area is a linear function of CU size due to the increased number of inputs to the crossbar and Power is a small coefficient quadratic, due to increased arbitration logic bitlines and multiplexer size of the crossbar . ADP is depicted in Figure 5.15d, with ADP evidently optimized at 16 CUs for the median and mean LOOG-insensitive kernel. Since it is optimized for such a scaled-down LOOG configuration, further testing in sections will be done with architecture configurations containing a maximum of 16,32 and 48 CUs. For the average and mean LOOG-sensitive kernels ADP is optimized with 32 and 40 CUs respectively.

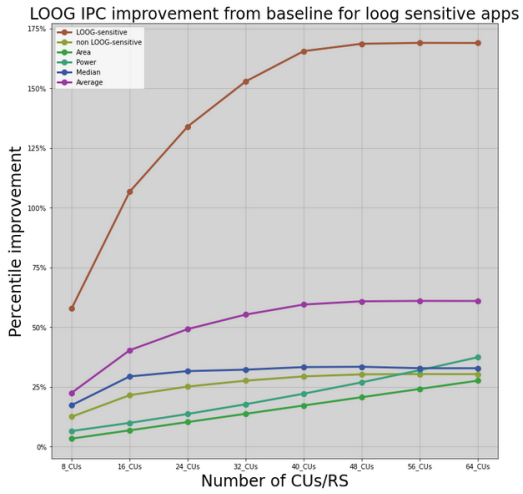
Power-Delay Product for improvement percentiles is depicted in Figure 5.15e. For each percentile, PDP is optimized in the CU size range [16,40] non monotonously. Energy-Delay Product, depicted in Figure 5.15f taking Delay into grater account, is optimized at 48 CUs only



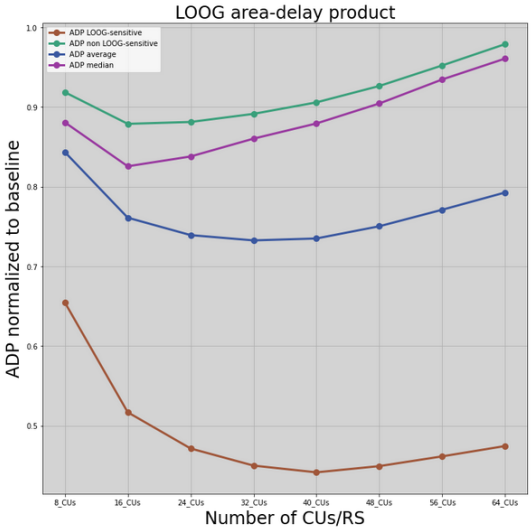
(A) LOOG IPC improvement compared to baseline per saturated improvement percentiles



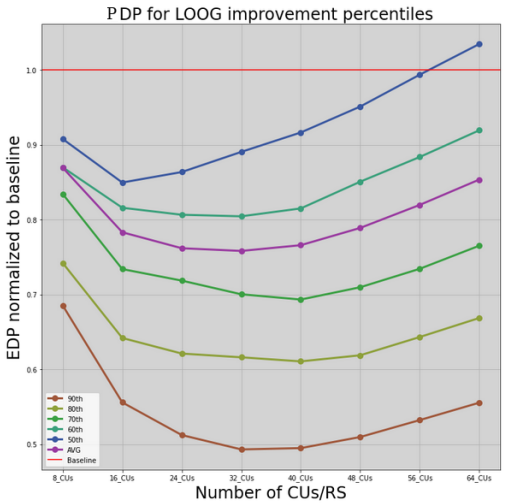
(B) LOOG Area and Power overheads scaling compared to baseline



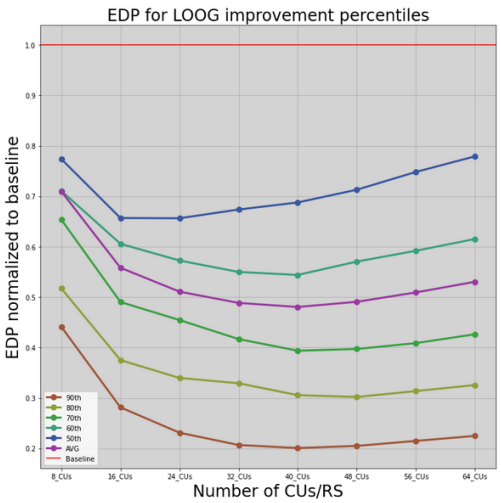
(C) LOOG IPC improvement plotted with Area and Power overheads



(D) LOOG Area-Delay Product scaling



(E) LOOG Power-Delay Product scaling



(F) LOOG Energy-Delay product scaling

FIGURE 5.15: LOOG scaling behavior across LOOG improvement percentiles

Benchmark	Suite	Kernel	Sat- ΔIPC
MST	Lonestar	Find Minimum	405%
LPS	Ispass-2009	3D Laplace calculation	396%
LIB	Ispass-2009	Path calculation	319%
Gramschmidt	Polybench	Gramschmidt	245%
Correlation	Polybench	Mean calculation	232%
MST	Lonestar	Verify minimum element	226%
MST	Lonestar	Find minimum element	194%
MST	Lonestar	Find minimum element 2	185%
QTC	Shoc	Compute degrees	180%
testAmr	Dragon	Refinement kernel	157%
Hotspot	Rodinia-3.1	Calculate temperature	153%
Backprop	Rodinia-3.1	Weights adjustment	145%
NN	Ispass-2009	Execute second layer	140%
Reduction	Shoc	Reduce	127%
Histo	Parboil	Input image	125%
Gramschmidt	Polybench	Initialization	119%
S3D	Shoc	Find Minmum	116%
Histo	Parboil	Intermediate kernel	115%
Histo	Parboil	Histogram calculation	112%
CFD	Rodinia-3.1	Compute	110%
Stencil2D	Parboil	Stencil kernel	106%
FFT	Shoc	FFT kernel	102%

Table 5.7: LOOG sensitive kernels and their percentile IPC saturated improvements

for the 80th percentile, the right-sizing value set for LOOG in our current implementation. Energy metrics are mostly optimized at 40 CUs for higher percentiles and all metrics are optimized at 16 CUs for the median kernel. The above results are summarized in Table 5.8. A more detailed view of the per-kernel and per-launch optimal CU configuration distribution is provided in section 5.8 .

Percentile	ADP	sat	PDP	EDP
Median	16	16	16	16
60	24	32	32	40
70	32	40	40	40
80	40	48	40	48
90	48	40	32	40

Table 5.8: Optimal CU configuration across saturated IPC improvement kernel percentiles for metrics ADP, IPC saturation, PDP, EDP

5.7 Accommodating LOOG on the front-end of NVIDIA Quadro GV100

The minor studies this section entails, aim to determine the new microarchitecture bottlenecks that might arise with the implementation of LOOG in the new architecture, in components not directly related to LOOG. Since LOOG directly imposes only backend modifications and the frontend virtually remains intact (save for the Scoreboard collision check that is replaced with register renaming and seamless Issue scheduling to the Operand Collect stage), we examine the optimal behavior of the Decoder, Issue scheduler and Instruction Buffer frontend components. Resizing and optimally configuring these components with regards to LOOG is essential. RRS are configured to 64 and CUs to 32 for all simulations used in this section.

5.7.1 Fetch-Decode stage Bandwidth study

Introducing Out-of-Order execution on the GPU increases backend throughput, therefore a right-sizing of the frontend stage is crucial. We seek to determine whether the traditionally CPU-specific distinction of front-end bound and back-end bound workloads is extensible to the GPU and likewise categorise applications to properly adjust the Decoder BandWidth. As previously mentioned, SMs in Tesla V100 [93] microarchitecture, comprise 4 processing blocks with that only share the Instruction cache and the memory subsystem [36]. In the actual implementation [93], instruction fetching is optimized with a per-partition L0 instruction cache. This is approximately modeled in GPGPU-sim by a right-sized SM-wide Instruction cache with 4 times the port throughput.

Instructions fetched in each of the processing blocks are ultimately stored in a statically per-warp partitioned Instruction Buffer. In the baseline model, each cycle, 2 instructions (per SM partition) are fetched from the Icache to the pipeline registers leading to the currently fetch-scheduled warp's Instruction buffer entries. Fetch-scheduling is done in a Round-Robin fashion on the active warps that are ready to fetch. Instructions are decoded with an equal throughput, and are placed in the appropriate Ibuffer entries. Each scheduler-isolated SM partition has one Issue scheduler by default, scheduling two instructions per cycle to the Collector Units in a Breadth-First fashion on Least-Recently-Used warp ordering. Therefore the total baseline Fetch-Decode Bandwidth (maximum throughput), normalized $\frac{1\_insn}{processing\_block \cdot cycle}$ is *8*. Determining the instruction throughput of the backend (based on the number and the activity Execution Units and Load Store Unit including Caches) is much more complicated and depends on instruction mix and latencies and throughput of Execution Units for each instruction type (latency of execution becomes significant in high IPC workloads that cause EXU congestion). In our implementation, runtime statistics are collected on a per-kernel-launch granularity to measure actual runtime backend throughput. The extension of the definition of Instructions Per Cycle typically refers to scalar thread instructions per cycle, not

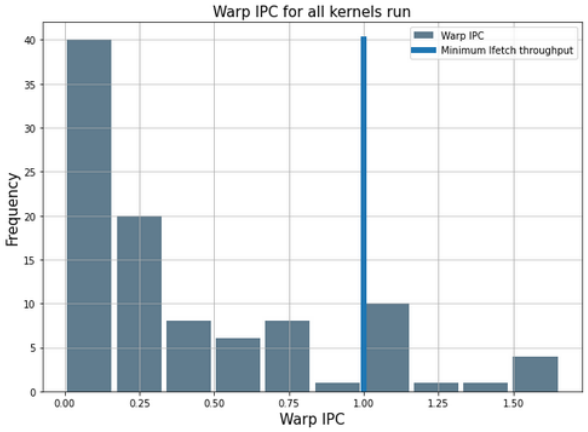


FIGURE 5.16: Warp IPC (equivalent to backend throughput) for 83% of workloads is below the minimum frontend throughput (GV100 frontend throughput is 8)

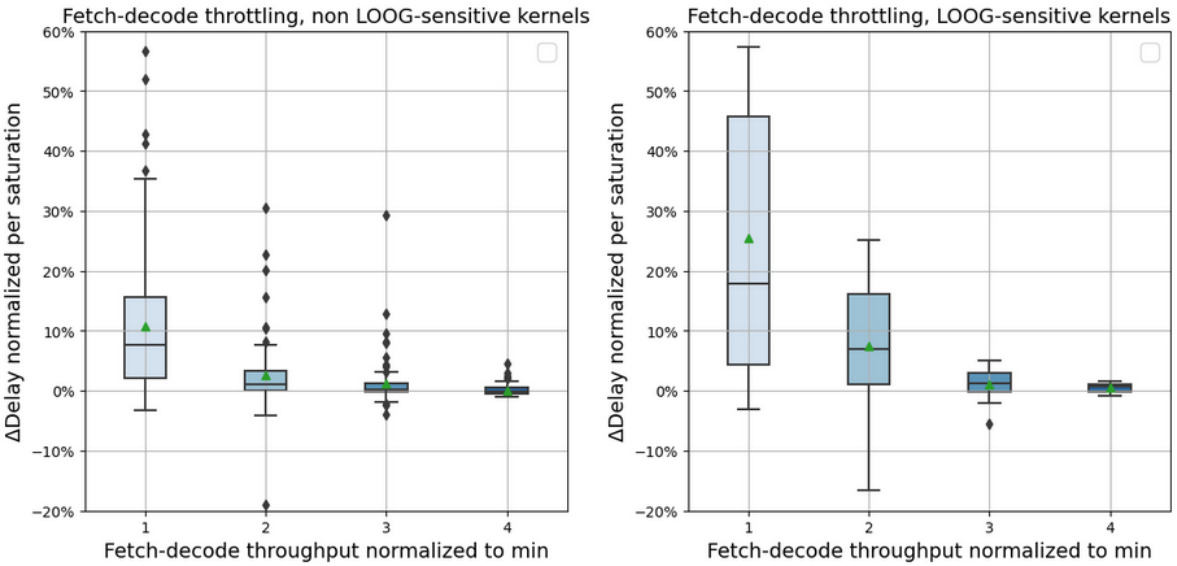
accounting for warp instructions, which is the actual unit that is executed in the backend:

$$IPC = \frac{scalar\_thread\_instructions}{total\_cycles} \tag{5.2}$$

When warp divergence is met, the backend equivalent of one warp instruction in the frontend is multiple instructions, potentially 32 if all threads within the warp diverge. The NVIDIA Volta GV100 is the first architecture to support independent thread scheduling [93] as mentioned in section, however independent warp scheduling and sub-warp coalescing are not implemented in GPGPU-sim, and it is assumed that their effect is only significant for a niche subset of workloads. Given the Pascal architecture single Program Counter and per-warp active mask divergence model (were inter-warp threads cannot be coalesced), we aptly redefine the IPC metric in terms of frontend instruction throughput as:

$$\begin{aligned} W\_IPC = \frac{warp\_insn}{total\_cycles} &= \frac{scalar\_thread\_insn}{total\_cycles} \cdot \frac{warp\_instructions}{scalar\_thread\_insn} \\ &= \frac{IPC}{warp\_size \cdot warp\_occupancy} \end{aligned} \tag{5.3}$$

Thus we can compare Warp IPC, which is a backend metric to its frontend equivalent. As depicted in Figure 5.16, when run with a saturated frontend, the backend throughput SM-wide and temporal average is below the minimum frontend throughput (equivalent to 1 instruction fetched per SM-wide L1 Icache per cycle) for 83% of kernels. Collecting further live runtime statistics on backend throughput would not be sensible since with such a low average, spikes in backend execution throughput would be amortized by the large number of dispatch-ready instructions in the RRS in LOOG. Hence, the concept of frontend-throughput bound workloads is not applicable to the GPU.



(A) (B)

FIGURE 5.17: Fetch-Decode throughput throttling Delay overheads across LOOG-sensitivity classes

In Figure 5.17a and Figure 5.17b, the Delay overhead for each of the scaled down Decoder bandwidths and for each LOOG-sensitivity class is displayed.

In Table 5.9, the IPC, normalized to baseline frontend throughput (8 per SM per cycle or 2 per processing block per cycle) is presented for each of the throttled Fetch-Decode bandwidth values. These values are normalized as explained above.

Evidently, LOOG-sensitive kernels suffer a greater overhead in smaller configurations due to the increased backend instruction throughput. Despite the average backend throughput being less than 2 for all kernels, as seen in Figure 5.16, throttling the Fetch-Decode throughput to this value poses a significant bottleneck for most kernels due to the aforementioned spikes in backend throughput, that are evidently not totally amortized by the instructions residing in the CUs. As seen in Table 5.9, this bottleneck causes a 2.63% and a 8.61% delay overhead for LOOG-insensitive and LOOG-sensitive kernels respectively. From the last column we conclude that Fetch-Decode throughput in LOOG and for GPGPU kernels can be throttled to half of the baseline with insignificant delay overheads (less than 1% even for LOOG-sensitive kernels). The power and area savings due to throttling in these stages and the concomitant Issue stage are 4.57% and 0.22% respectively. As seen in Table 5.4. Note that an IWindow of 2 in Table 5.4 is equivalent to a Fetch-Decode bandwidth of 2 per processing block, therefore 8 in our current analysis. Hence, while scaling down the relevant structures in the design would not be sensible, clock or power gating them would provide significant energy efficiency for GPGPU kernels even in LOOG. Due to the insignificant delay overheads in the downscaled Decoder bandwidth configuration, further simulations were performed with the baseline Decoder bandwidth of 2 per SM partition.

Decoder BW	1	2	3	4
LOOG-sensitive IPC	89,22%	97,43%	98,78%	99,88%
LOOG-insensitive IPC	74,43%	92,54%	98,84%	99,48%

TABLE 5.9: Average kernel IPC normalized to baseline frontend throughput (8) for each of the Fetch-Decode bandwidths normalized to $\frac{1\_insn}{SM \cdot cycle}$. Issue scheduler throughput is 2 for all of the above configurations

5.7.2 Issue scheduling depth study

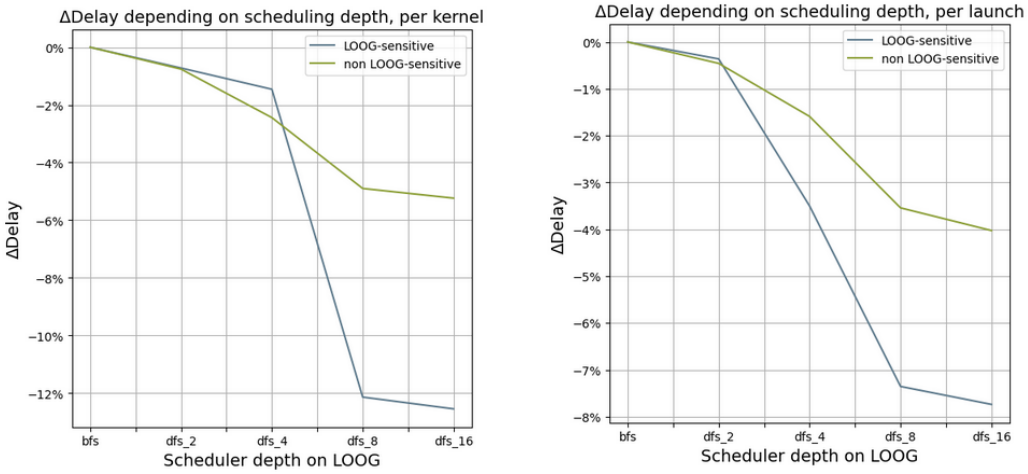
As depicted in Figures 5.17a and 5.17b , IPC tends to saturate asymptotically even when approaching a decoder throughput of 4 per SM, half that of the actual fetch-decode throughput of the Volta partitioned SM. However, it is worth noting that unexpected negative Delay overhead fluctuations before saturation are recorded. For a normalized Fetch-Decode bandwidth of 2, a quartile of the distribution belongs in the range [0,-4%] and [0,-17%] for LOOG-insensitive and LOOG-sensitive kernels respectively, thereby showing greater performance in a smaller Fetch-Decode bandwidth configuration. These random fluctuations were correctly attributed to suboptimal instruction scheduling. In the original LOOG GTX1080Ti implementation [14] , a Round-Robin instruction issue scheduler was used, presuming that the instruction reordering taking place in the next stage would inherently not require a more sophisticated scheduling policy. While LOOG remains insensitive to such sophisticated policies in the current implementation, a per-warp IWindow depth scheduling scheme was not investigated. In the baseline model, 2 instructions are considered for scheduling by the Issue scheduler every cycle per processing block, potentially from the same warp. We kept this implementation, in contrast to the original LOOG implementation [14], thereby producing the aforementioned fluctuations as explained below:
When configuring the fetch-decode throughput to 1, the instruction buffer of each warp fills with one instruction that is immediately scheduled for Issue as no scoreboard check takes place in LOOG [12, 13]. Therefore, instructions are seamlessly scheduled in a per-warp Breadth-First fashion. For a throughput of 2, the Depth-First scheduling that takes place in our implementation, similar to the in-order baseline, causes seamless scheduling of 2 instructions per cycle from the same warp, immediately emptying the respective warp's instruction buffer and making it eligible for fetch. Note that for any throughput greater than 1, the number of instructions scheduled is restricted by the Icache line size when approaching its end. For a bigger throughput, 2 instructions are still Issued per cycle, however, not emptying the IBuffer, thereby not making the current warp eligible again and causing fetch in other warps as well,implementing a "fairer" scheduling policy that, evidently, produces worse IPC. We conclude that prioritizing the execution of instructions from the same warp, when possible, improves performance in LOOG by enhancing the exploitable ILP when fewer warps concurrently occupy the CUs.

To gain more insight in this behavior, we configure the Fetch, Decode and issue scheduler

bandwidths to an unrealistic 16 per cycle, to Issue as many instructions as possible from individual warps before switching, producing prohibitive Area and Power overheads as seen in Table 5.4. Nevertheless, this is just to ease simulation and a low Area and Power overhead implementation of such a mechanism is provided at the end of this subsection.

We test 5 different Issue scheduling depth policies: Breadth-First Scheduling, where one instruction is issue scheduled per warp before switching warps, similar to the original LOOG implementation [14], and $[2, 4, 8, 16] - deep$ Depth-First Scheduling, where for N-deep DFS, the maximum number of instructions possible is scheduled from the same warp up to N, before switching. The last configuration fetches a whole Icache line each time. Results are collected for both kernels and individual launches as well as LOOG-sensitivity classes.

As depicted in Figures 5.18a and 5.18b , a scheduler with the option to utilize Longer Instruction Windows from individual warps, provides significantly better results with LOOG, especially for LOOG-sensitive applications. This improvement is saturated beyond 8 instructions. Specifically in Figure 5.18a we see a bias of single-launch kernels toward LOOG-sensitivity that is extensively revisited in Section 5.8.2 .



(A) (B)

FIGURE 5.18: Delay improvement across Issue scheduling depths

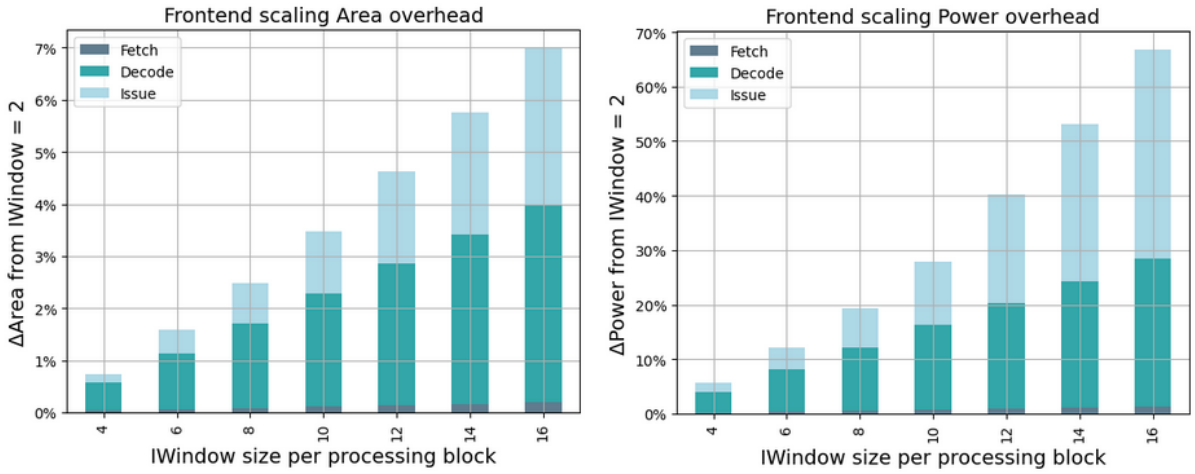
IWindow	4	6	8	10	12	14	16
Fetch	0.03%	0.05%	0.08%	0.11%	0.14%	0.16%	0.19%
Decode	0.54%	1.09%	1.63%	2.17%	2.72%	3.26%	3.80%
Issue	0.16%	0.44%	0.77%	1.19%	1.77%	2.33%	3.00%
Total	0.73%	1.58%	2.48%	3.47%	4.62%	5.76%	6.99%

TABLE 5.10: Area overhead contribution of each stage in frontend scaling with a baseline IWindow of 2

As mentioned before, all the above simulations were performed with an unrealsitic Area and Power hungry IWindow to ease the implementation. This provides the required Depth-First Scheduling but causes instructions to enter the pipeline in a faster rate as well and

IWindow	4	6	8	10	12	14	16
Fetch	0.20%	0.39%	0.59%	0.79%	0.98%	1.18%	1.38%
Decode	3.86%	7.72%	11.58%	15.44%	19.30%	23.17%	27.03%
Issue	1.56%	4.09%	7.09%	11.70%	20.00%	28.79%	38.38%
Total	5.62%	12.21%	19.27%	27.93%	40.29%	53.13%	66.78%

TABLE 5.11: Power overhead contribution of each stage in frontend scaling with a baseline IWindow of 2



(A) (B)

FIGURE 5.19: Frontend scaling Area and Power overheads (baseline IWindow of 2)

occupy the CUs sooner, thereby falsely increasing the instruction reordering potential early on. We speculate this exaggerated ILP exploitation due to higher frontend throughput, however, is minimal for all workloads, since the average backend throughput is less than a mere 2 instructions SM-wide (including the 4 processing blocks) per cycle even for the highest IPC kernels as depicted in Figure 5.16. Therefore, when configuring the IWindow to $\frac{8\_insn}{processing\_block \cdot cycle}$ (equivalent to 32 in Figure 5.16), the collector units are expected to fill in approximately 1 cycle instead of approximately 4 with the baseline Fetch-Decode throughput configuration of $\frac{2\_insn}{processing\_block \cdot cycle}$ in section "Fetch-Decode stage Bandwidth study", gaining an insignificant head start. When full, behavior of the baseline and exaggerated IWindow is identical, save for the Issue scheduling depth, which can be emulated as described below.

Issue scheduling at the correct depth

Since frontend throughput is not significant for the instruction scheduling scheme described in this section, an Issue scheduler needs to be implemented that provides scheduling in the same order among warps with a lower instruction window. For an IWindow of 16 (where the performance increase from increased scheduling depth is evidently approximately saturated in Figure 5.18a), the addition of a 3-bit Fetch-Decode counter is required to provide this

behavior. Since instruction Fetching and Decoding happens at a rate of 2 per cycle per processing block, the same warp needs to be prioritized every cycle, incrementing the respective counter, up to a total of 8 times or an Icache miss, when priority is given to the next warp. In the Issue stage, switching of the prioritized warp happens when the current warp's entries are emptied, following the aforementioned events. Thus, the same scheduling behavior is provided with a smaller Iwindow of 2, as in the baseline.

Due to the inherent error in the above speculations, this study is kept separate and for the results presented in this thesis, LOOG is configured with an 2-deep Depth-First Issue Scheduler, thereby Fetching, Decoding and Issuing $\frac{2\_insn}{processing\_block \cdot cycle}$ as in the baseline model.

Algorithm 3
Warp instruction depth scheduling

```
order_warps_RR()
total_issued = 0
while total_issued < max_issue_per_cycle do
  get_next_warp()
  issued = 0
  while issued < issue_depth && total_issued < max_issue_per_cycle do
    if !ibuffer_empty() && !warp_waiting() && pipeline_avail() then
      issue_instruction()
      issued += 1
      total_issued += 1
    else
      break
    end if
  end while
end while
```

5.7.3 Instruction Buffer reconfiguration

LOOG improvement on deeper Issue scheduling Instruction Windows raised the issue of inter-warp homogeneity regarding exploitable ILP. Since LOOG-sensitive kernels benefit more from deeper Issue scheduling windows, by having less warps concurrently utilize the Operand Collector and benefit from more aggressive reordering, it naturally follows that in the presence of inter-warp ILP heterogeneity more concurrent Operand Collector access should be given to high-ILP warps.

Using GPGPU-Sim, we collected per-warp and per-kernel-launch "reconfiguration metrics", elaborated on below, representing exploitable warp ILP for the whole kernel's runtime. Given the wide distribution of the aforementioned metrics across warps for most kernels, we make the per-warp Ibuffer partitioning configurable instead of static (as in the baseline model [24]) and ensure that it adapts to the respective warp's exploitable ILP at runtime. An Ibuffer reconfiguration controller was implemented in GPGPU-sim providing the needed

functionality. It uses a reconfiguration policy and a per-warp recorded reconfiguration metric, dynamically modifying the Ibuffer to suit each warp's exploitable Instruction window depth.

The following analysis on the reconfiguration metrics is performed on the basis of kernel launch statistics instead of cumulative kernel statistics, normalized to the per-launch maximum and only regarding active warp slots. For instance, given a warp with 8 active warp slots, only those are used in further calculations and all the relative metrics are normalized to the highest value among them, as depicted in Figure 5.20 . As described in section, upon instruction Issue and CU allocation, the Register Alias Table (indexed by register ID and warp ID, containing a Collector Unit ID) is read once for each source operand. It is also updated so that the destination register points to the allocated CU [14]. Total RAT entries essentially represent the average destination registers per instruction for each warp. Since these registers are source operands for subsequent dependent instructions, total RAT entries provide a vague measure of per-warp average instruction dependencies, therefore, a reverse measure of the exploitable ILP of a warp. The per-warp standard deviation of the total RAT entries vector across launches is depicted in Figure 5.22b.

Used RAT entries reconfiguration metric

Since a destination register may be used multiple times as a source operand and given that dependent instructions may present far down the instruction stream, essentially making the dependency irrelevant for the Instruction Windows we examine, used RAT entries provide a more accurate reverse measure of exploitable ILP. Each time a CU ID is read from the RAT for a source operand, a per-warp used RAT entries counter is incremented. Thus, both of the above concerns are addressed. Used RAT entries are roughly the equivalent of Scoreboard collisions in the baseline in-order model. It could be argued that used RAT entries should be used to prioritize warps, since RAT entry usage equates to less high-latency Register File reads and traffic. However, in the presence of used RAT entries, more instructions concurrently occupy the CUs, preventing other independent instructions from entering and potentially being immediately scheduled for Dispatch. As seen in Figure 5.13b, a significant fraction of kernels completely occupy the CUs for their whole execution. Therefore utilizing used RAT entries to de-prioritize warps, provides a smooth instruction scheduling to the backend for warps with independent instructions. Warps with high dependencies can occupy the CUs when the former have exhausted their IWindow or encountered Icache stalls. RAT entry usage widely varies across launches, as seen in Figure 5.22a, where kernel launches are sorted by their maximum percentile RAT entry usage across all their active warps. In Figure 5.23a, the standard deviation of normalized RAT entry usage is provided. Figure 5.21a and Figure 5.21b depict the per-kernel-launch and per-warp total RAT entries and RAT entry usage respectively, sorted by the standard deviations provided above, showing significant variance.

Warp readiness reconfiguration metric

In the Fetch stage, a warp is considered ready (eligible for instruction Fetch) if it has not terminated, has no pending Icache misses and all its IBuffer entries are empty. Hence, warp readiness varies among warps in the presence of control hazards due to loops, causing more Icache hits (the only significant source of Icache hits, save for subsequent instructions on the same cache line, since there is no prefetching) and due to Issue scheduling preference of the respective warps, causing frequent emptying of their IBuffer entries. We collect the per-launch and per-warp statistics regarding warp readiness. Across launches, the standard deviation of (normalized to per-launch max) warp readiness is depicted in Figure 5.23. Evidently, the distribution for this metric is wider than both total RAT entries and used RAT entries, with more kernel launches exhibiting high variance among their active warps. This potentially explains the superior results provided below by this metric, by prioritizing less warps more heavily. Another factor that explains said results is the more prominent "snowball effect" by reconfiguring with this metric. When a warp is prioritized for Issue, its readiness metric is increased, providing even higher priority to it and so on. This happens to a greater extent than with the above metrics.

Split reconfiguration policy

When using the Split policy shown in Equation 5.4, available Ibuffer entries are split fairly among warps. The first line in Equation 5.4 corresponds to reverse reconfiguration metrics such as used RAT entries, while the second line corresponds to direct reconfiguration metrics, such as warp readiness.

$$
\begin{aligned}
warp\_score &= \frac{1}{1 + reverse\_reconf\_metric} \\
warp\_score &= reconf\_metric \\
total\_score &= \sum_{warp \in Ready\_Warps} warp\_score(warp) \\
warp\_entry\_pool &= \sum_{warp \in Ready\_Warps} current\_entries(warp) \\
entries(warp) &= floor(warp\_entry\_pool \cdot \frac{warp\_score}{total\_score}) \\
remaining\_entries &= warp\_entry\_pool - \sum_{warp \in Ready\_Warps} entries(warp) \\
remaining\_score(warp) &= \frac{warp\_score}{total\_score} - floor(\frac{warp\_score}{total\_score})
\end{aligned}
\tag{5.4}
$$

Remaining entries are distributed among warps based on remaining_score.

Winner-take-all reconfiguration policy

Winner-take-all policy greedily redistributes available entries among warps with the highest scores. Warp scores, total scores and the warp entry pool is calculated as in Equation 5.4.Redistribution of entries takes place as described in Algorithm 4.

Algorithm 4
Winner-take-all IBuffer reconfiguration policy

```
sort_warps_by_score()
while warp_entry_pool do
    warp = get_next_warp()
    entries[warp] = min(max_entries_per_warp, warp_entry_pool)
    warp_entry_pool -= entries[warp]
end while
```

IBuffer reconfiguration results

The results provided by combining the two reconfiguration policies with the two metrics, across kernels are depicted in Figure 5.24a, Figures 5.24b, 5.24c and 5.24d. Split policy using a warp readiness metric provides significant results, with an average 1.8% speedup over all kernels and 4.4% for LOOG-sensitive kernels, up to 10.2%.

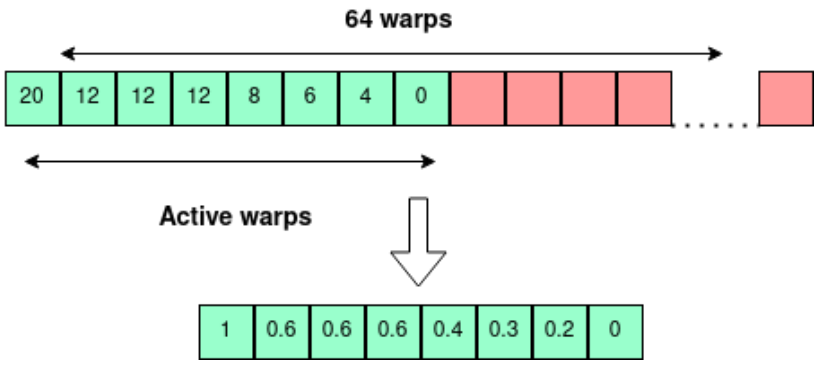
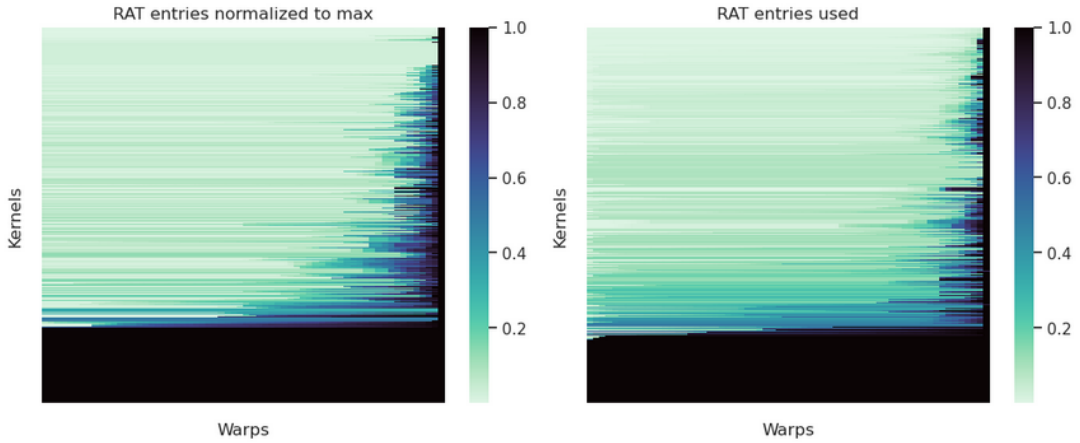


Figure 5.20: Data visualization provided in Figure 5.21

5.8 Out-Of-Order reconfiguration

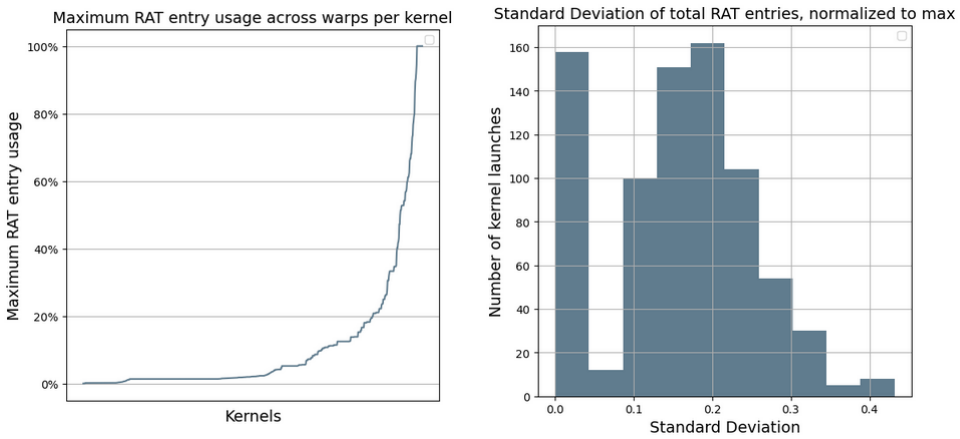
5.8.1 Observations leading to Collector Unit reconfiguration

The analysis presented in Section 5.3 regarding exploitable ILP, provided us with insight into kernel categories characterized by diverse features (compute-intensive kernels, cache-bound kernels with separate Data load and compute phases) whose ILP can be leveraged to yield significant speedup with LOOG. Furthermore, the examination of the significant performance



(A) RAT entries total (B) RAT entries used

FIGURE 5.21: RAT entries and used RAT entries normalized to the per-kernel maximum across warps, as seen in Figure 5.20

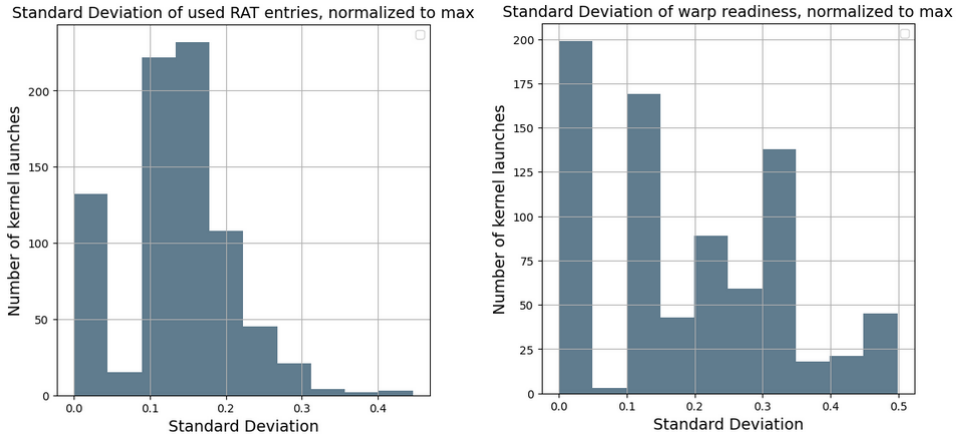


(A) Maximum RAT entry usage among warps per kernel, with kernels sorted (B) Per-launch std of per-warp total RAT entries normalized to launch maximum

FIGURE 5.22: Measures of total and used RAT entries deviation

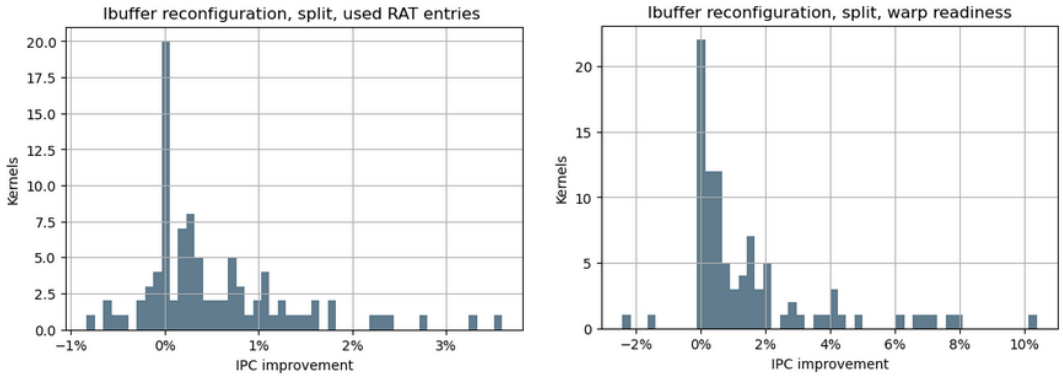
scaling as well as Area and Power overhead scaling in Section 5.6.3 lead to the realization that optimizing performance or figures of merit accounting for Power dissipation (PDP, EDP) requires highly varying Collector Unit configurations for different kernels. Specifically, scaling the CUs produces enough of a performance increase for some workloads (LOOG-sensitive) that justifies the power and area overhead of the largest configurations, while for others, no LOOG configuration provides a big enough speedup to justify the overheads.

As depicted in Figure 5.15c in section Section 5.6.3 , IPC is saturated after a certain point (slightly varying across workloads) with increasing CU numbers, while performance and power continue to increase linearly with the CU - RF crossbar outputs. The performance improvement distribution is anomalous, with the subset of LOOG-sensitive kernels improving greatly on LOOG. LOOG improvements from baseline for different percentiles of saturated improvement are more accurately presented in Table 5.12

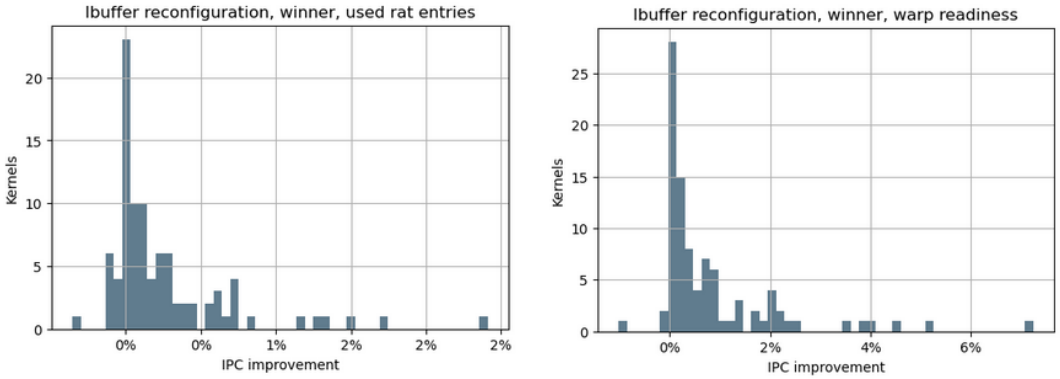


(A) Used RAT entries (B) Warp readiness

FIGURE 5.23: Per-warp STD of reconfiguration metrics normalized to maximum, across kernel launches



(A) Split policy and Used RAT entries metric (B) Split policy and Warp Readiness metric



(C) Winner-Take-All policy and Used RAT entries metric (D) Winner-Take-All policy and Warp Readiness metric

FIGURE 5.24: IPC improvement across Ibuffer reconfiguration metrics and policies

It naturally follows that the number of Collector Unit is a major potential axis of reconfiguration, solely corresponding to the size of LOOG, since the bandwidth of frontend components has been addressed in Section 5.7. As demonstrated in Section 5.7.1 , performance

	8_CUs	16_CUs	24_CUs	32_CUs	40_CUs	48_CUs	56_CUs	64_CUs
90th	55.39%	97.63%	121.77%	138.65%	146.71%	148.84%	148.85%	148.89%
80th	43.50%	71.17%	82.92%	89.10%	99.94%	104.97%	105.10%	105.40%
70th	27.65%	49.72%	58.13%	68.07%	76.13%	78.74%	79.66%	79.50%
60th	22.42%	34.69%	40.89%	46.32%	49.84%	49.12%	49.31%	49.41%
50th	17.32%	29.35%	31.58%	32.18%	33.26%	33.39%	32.79%	32.77%
AVG	22.50%	40.32%	49.15%	55.24%	59.44%	60.79%	60.96%	60.93%

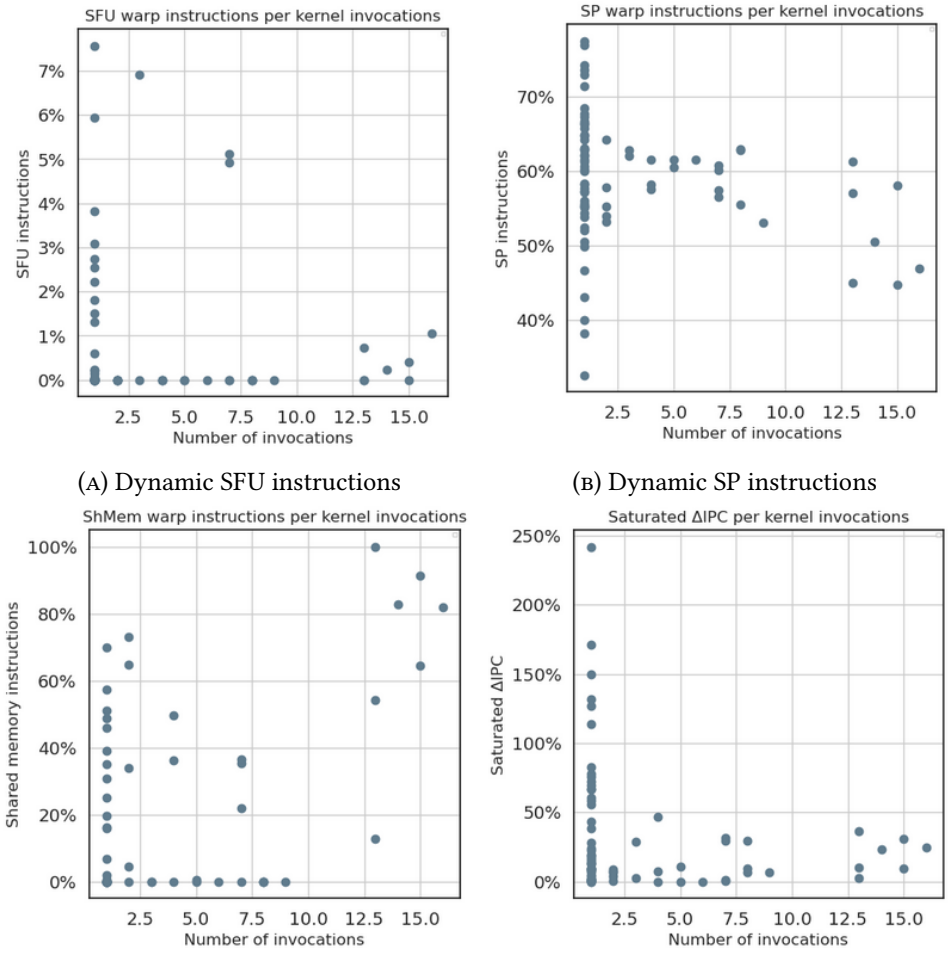
TABLE 5.12: IPC increase across CU configurations for saturated LOOG improvement percentiles

gain by scaling the decoder throughput is saturated and as examined in Section 5.7.3 , there is significant potential for performance improvement when reconfiguring Ibuffer entries. However, the most significant performance improvement in LOOG comes from increasing the number of Collector Units, as presented in Section 5.6.3 . Given that in LOOG, CUs act as the equivalent of Reservation Stations in the Tomasulo algorithm, an increasing number of instructions concurrently occupying CUs equates to greater reordering potential as well as Register File reads avoided, as said instructions have their source operands filled in one cycle by listening to a common data bus for instructions writing on the respective registers. In this section, it is crucial to determine the aforementioned spectrum of optimal CU configuration for each workload based on figures of merit, that will potentially allow us to exclude parts of the design space that are optimal for no launches at all. Given these configurations, the maximum benefit in terms of Energy efficiency optimization will be provided with minimal performance deterioration from the biggest scale-up LOOG configurations.

5.8.2 Behavior of individual kernel launches with LOOG

Reconfiguring the architecture by power gating poses significant energy and delay (wake-up time) overheads that can be traded off. In order to introduce minimal energy overheads, we need to amortize them over long delays, potentially in the order of microseconds. With a boost clock of 1530 MHz in Quadro GV100 [49] , this equates to several thousands of cycles in the SM clock domain. In [108], the CUDA API call latencies such as $cudaMemcpy()$ and $cudaLaunch()$ are determined and it is proven that they all surpass 1 microsecond, hence, offloading a kernel to the GPU lasts at least several microseconds. Therefore, as individual kernel launches provide the finest granularity we can opt for in terms of reconfiguration without introducing performance degradation during execution, it is imperative to examine kernel behavior regarding launches (calls/invocations). "Kernel launches" will be used interchangeably with "kernel invocations" in the rest of this thesis.

Regarding the number of launches for the kernels under examination, in Table 3.1 42% of the kernels had one launch in total. The distribution of number of launches for the rest of the kernels is depicted in Figure 5.26. Note that said percentages may slightly vary as workloads



(A) Dynamic SFU instructions

(B) Dynamic SP instructions

(C) Dynamic shared memory instructions

(D) Saturated DeltaIPC - invocations

FIGURE 5.25: Various types of dynamic instructions overall, associated with the number of invocations of their kernel. High launch kernels are memory-intensive

run across different configurations introduced different simulation overheads, prohibiting statistics collection among some of them and restricting our analysis to those that provided results for all the configurations.

It is worth noting, that as previously observed, kernels that are invoked multiple times tend to be significantly less LOOG-sensitive. This correlation was thought to be explained in theory by the assumption that more frequently invoked kernels tend to have less instructions and execute for less cycles, therefore failing to fill up the caches and consequently stalling more and not improving by OOO execution, as explained in Section 5.3 . This assumption proved to be false and it was determined that multi-launch kernels tend to have more memory instructions overall, therefore longer latency operations on average, making them less LOOG-sensitive. As depicted in Figure 5.25b, 5.25c and 5.25a, there is a clear correlation between SFU instructions and Number of invocations. The vast majority of kernels with a significant number of SFU instructions overall are invoked just once. An even stronger negative correlation exists between SP instructions and number of invocations. Shared memory

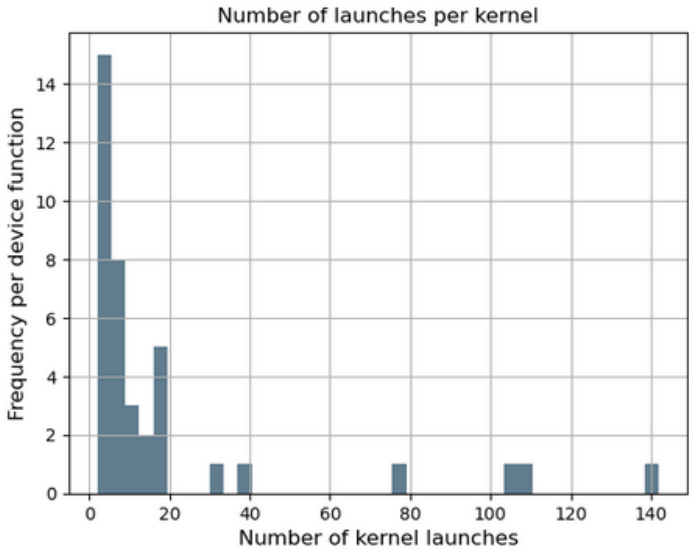
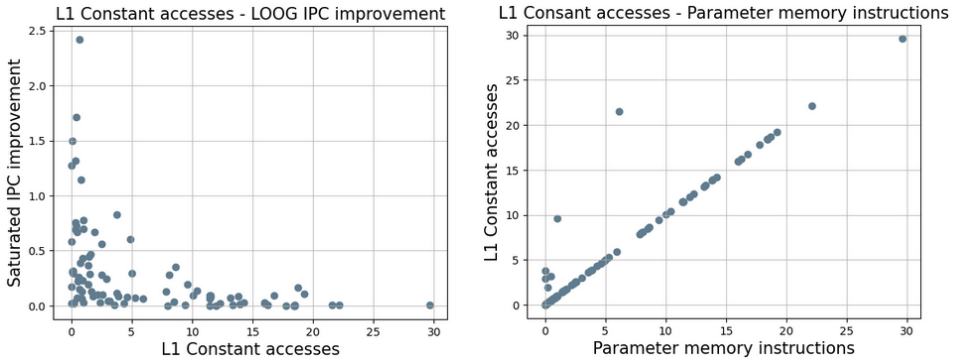


FIGURE 5.26: Number of invocations (launches) across all kernels examined

instructions naturally increase with more kernel launches, given the mutually exclusive relationship between most types of memory operations and other pipelines. Evidently, multi-launch kernels tend to be memory intensive contrary to single-launch kernels whose vast majority tends to be compute intensive and specifically highly utilize the SFU pipeline. Another related explanation is the strong correlation exhibited between constant memory parameter memory warp instructions and constant memory operations. As seen in Figure 5.27b, constant memory accesses of our workloads are mostly correlated to parameter memory instructions. Indeed, this is due to low total runtimes. Saturated IPC improvement in LOOG is strongly negatively correlated with (Figure 5.27a) with constant memory instructions as explained above, given that reordering parameter memory operations does not provide any performance gains. Low launch runtime, therefore, is a significant consideration when profiling kernels according to LOOG improvement, as the cost of having high parameter memory operations overall is not amortized over the whole execution below certain runtimes. This is manifested further in section, where it is proven that total launch runtime is a highly significant feature for predicting saturated LOOG IPC improvement from hardware performance counters.

This skewed distribution produces different results for kernels and launches in all levels of the analysis. Results will mainly be presented across cumulative kernel statistics or individual kernel launch statistics and across LOOG-sensitivity classes. Plots regarding percentiles of a given metric for the set of workloads are always per-kernel, as percentiles would be heavily skewed towards multi-launch kernels.



(A) Saturated IPC improvement in LOOG is inversely proportional to L1C accesses

(B) The majority of L1 constant accesses service parameter memory instructions

FIGURE 5.27: Saturated LOOG IPC improvement correlation with L1C and parameter memory accesses

5.8.3 Figures of merit used in our analysis

Defining the optimal reconfiguration is relative to the required figure of merit, representing some sort of efficiency domain we need to optimize. Since performance improvement with LOOG is asymptotically saturated for CU values less than 48 and totally saturated for greater values, we do not define the optimal configuration based on performance as the maximum available CUs, instead we use the saturated IPC improvement metric. Figures of merit taking power into account as well are used along with it:

- *Saturated 98% IPC improvement*
  Considered to be optimized when the current configuration provides performance within of an arbitrary 2% of the saturated improvement performance (maximum achievable IPC improvement on the most scale-up LOOG configuration).

- *Saturated 95% IPC improvement*
  The provided IPC should be within 5% of the maximum achievable IPC improvement.

- *Power-Delay Product*
  Equivalent to the total energy dissipation of the application.

- *Energy-Delay Product*
  Equivalent to $Power - Delay^2$ Product, taking delay into greater account.

These are the metrics our reconfiguration model will be evaluated upon as well, relative to their optimal values. We will often refer to them as "reconfiguration metrics".

5.8.4 Classifying the types of reconfiguration examined

In this subsection, we classify the types of reconfiguration our implementation provides by three different characteristics. Generally, the optimal configurations are inferred and applied

to the microarchitecture by a reconfiguration controller which receives input according to its level of implementation and on a varying temporal granularity.

Based on the level of reconfiguration controller implementation

Software level Referring to the implementation of a software reconfiguration controller that can even utilize launch-level execution information retrospectively to provide new suitable configurations.

Hardware level Regarding the implementation of a hardware controller that can only utilize runtime metrics from hardware counters of the current kernel launch.

Based on the input to the controller

Optimal reconfiguration When optimally reconfiguring the microarchitecture, it is assumed that the most suitable configurations for each application and metric are known in advance. This is provided by storing workload scalability characteristics based on previous executions on a kernel-launch granularity and on all the available configurations. Hence, only a software implementation of the controller can provide optimal reconfiguration.

Runtime reconfiguration When reconfiguring during runtime, the only source of information regarding application scalability are the values of hardware performance counters, that can be utilized at both levels of implementation.

Based on the temporal granularity of reconfiguration

Static / Whole-kernel reconfiguration In whole-kernel reconfiguration, it is hypothesized that cumulative measures regarding scalability have been previously collected over all of the kernel's launches and used to profile the kernel.

Semi-dynamic / per-launch reconfiguration In semi-dynamic reconfiguration, the kernel is profiled on a per-launch granularity and the microarchitecture is likewise reconfigured. In the scope of this thesis, only optimal reconfiguration is examined with a per-launch granularity.

First-launch reconfiguration In first-launch reconfiguration, the kernel is profiled based on scalability measures collected on its first launch and the resulting optimal configuration is applied to all its subsequent launches.

Regarding first-launch reconfiguration, it is noted that all kernel profiling with our model presented in section 5.8.7 is done with hardware counters from the inorder configuration. Profiling based on other initial configurations would require multiple models, unless the inorder model with its optimal features and fitted coefficients can sufficiently predict optimal

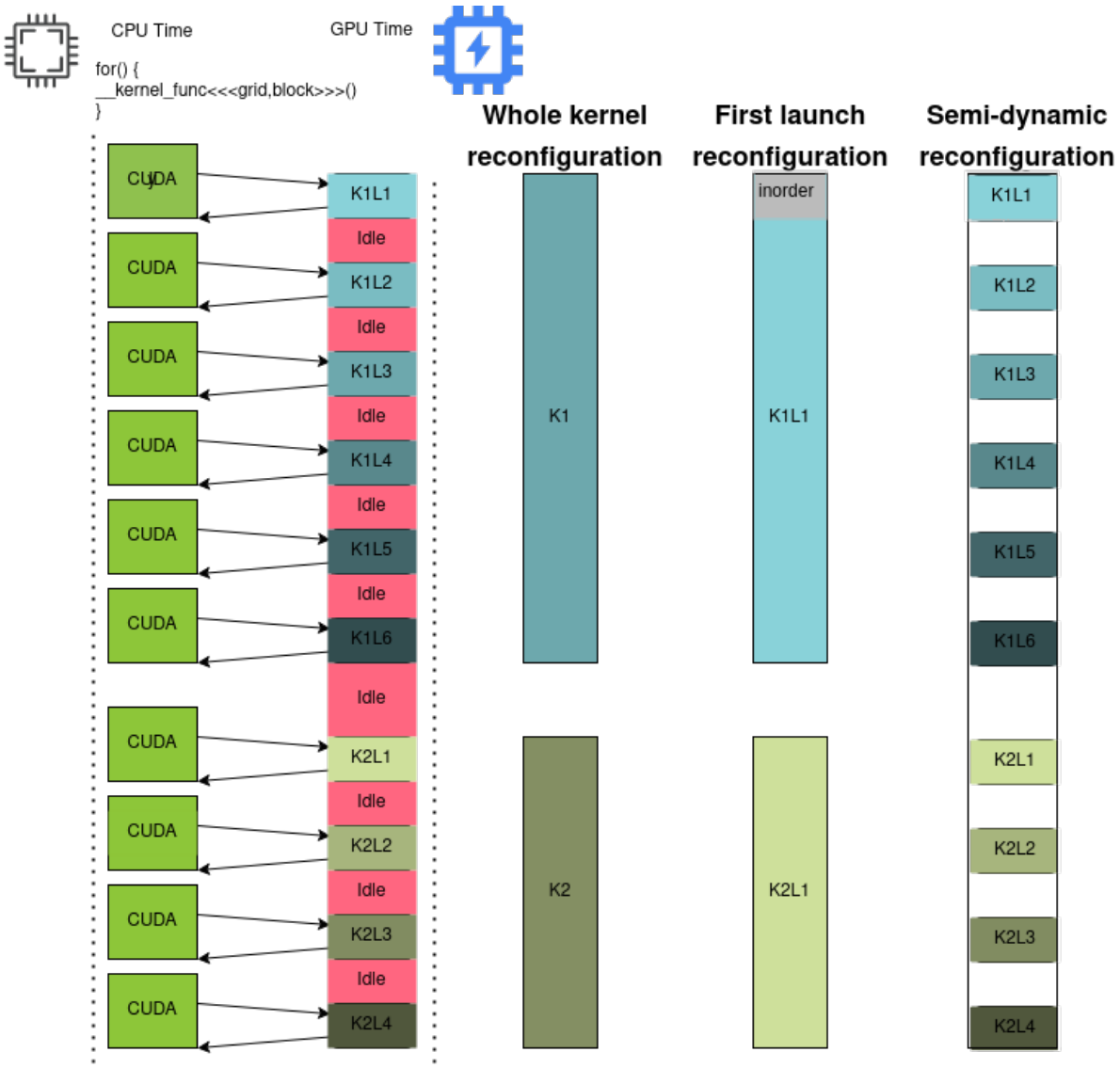


FIGURE 5.28: Types of reconfiguration examined, varying in temporal granularity

reconfigurations from counters produced by scale-up initial configurations, which remains to be examined. In Figure 5.28, all reconfiguration schemes based on temporal granularity are presented.

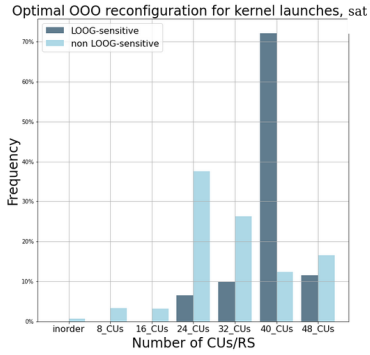
5.8.5 Optimal configurations

Optimal static reconfiguration (whole-kernel reconfiguration) assumes perfect knowledge of the optimal configuration for all of the kernel's launches cumulatively (given the reconfiguration metric at hand) and applies it for its whole execution. *Optimal launch reconfiguration* (equivalent to "Optimal semi-dynamic reconfiguration") refers to reconfiguring for all of the kernel's launches based on the optimal CU configuration in a per-launch granularity for the

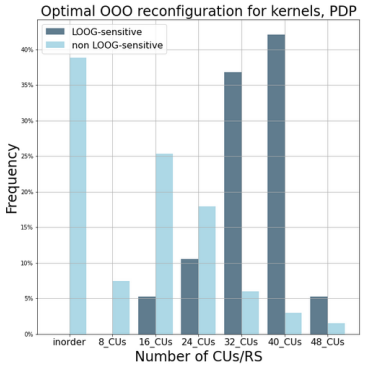
given metric. As already mentioned, the observation that the IPC increase is generally saturated beyond 48 collector units sets the limit for OOO right-sizing, as processor area and power continue to increase beyond that point. No kernels and scarce kernel invocations were found to behave optimally beyond 48 collector units, therefore such designs are excluded from our analysis. To produce optimal reconfiguration results it is crucial to determine the per-kernel and per-launch configuration that optimizes the metric at hand, based on the metrics defined above. Determining the results of optimal reconfiguration sets the limit for any actual reconfiguration methods we shall implement. In plots **??** and **??** , optimal configurations are depicted for both LOOG-sensitive and LOOG-insensitive classes of kernels, across all metrics. As expected, in Figure 5.29e it is seen that a higher percentage of the kernels relative to launches saturate at the highest of CU configurations. The aforementioned skewing of optimal configurations for whole kernels towards bigger CU configurations is not observed to as big a degree. It is apparent that for higher values of saturated OOO IPC, saturation often tends to happen in smaller configurations. It is newly seen that most of the launches saturate at 24 CUs, while optimization in inorder and small LOOG configurations is almost monopolized by kernels. Therefore the average kernel tends to have a wider saturation distribution, while the average launch saturates in medium configurations. As observed in Figure 5.29a, virtually none of the LOOG-insensitive launches saturate in the baseline configuration and none of the LOOG-sensitive launches saturate below 24 CUs. 36% of the LOOG-insensitive launches saturate at 24 CUs and the median saturation value is 32 CUs. 72% of the LOOG-sensitive launches saturate at 40 CUs, which is the median optimization value. In Figure 5.29c it is seen that for LOOG-sensitive whole kernels, the median saturation value is 40CUs, while for LOOG-insensitive kernels it is 32 CUs, the same as launches with a smoother distribution.

As was expected from the previous analysis, in Figure 5.29f it is seen that whole kernels monopolize the highest scale-up CU configurations due to LOOG-sensitivity, but also the inorder configuration, where 30.3% of kernels are optimized. Once again it is seen that launches tend to occupy the center of the distribution, and be distributed smoothly, Contrarily, whole kernels tend to have a smooth distribution only beyond the inorder threshold. Indeed, when observing Figure 5.15b, it is clear that Δ Power from 8 CUs to 16 CUs is more than double the transition from inorder to minimum LOOG. This creates a significant threshold when configuring to minimum LOOG that is easily surpassed for bigger CU configurations. Since LOOG-sensitive kernels are only defined by their speedup in the biggest scale-up LOOG configuration, those that saturate slower with regards to CU scaling tend to optimize PDP in smaller LOOG configurations, as seen in Figure 5.29b. The narrower distribution pointed out for the saturation metric for launches compared to whole kernels, is seen for the PDP metric as well in Figure 5.29d

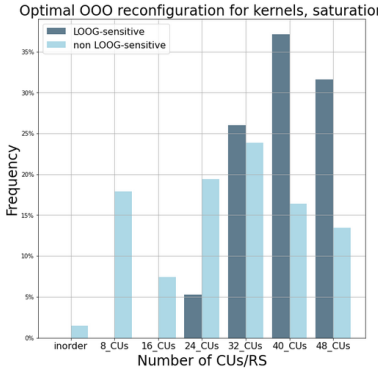
The observations made earlier regarding the "improvement threshold" from inorder to minimum LOOG do not apply to EDP, as factoring in delay to a greater degree eradicates



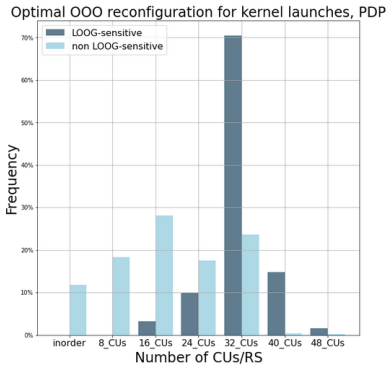
(A) sat98,launches, LOOG-sensitivity



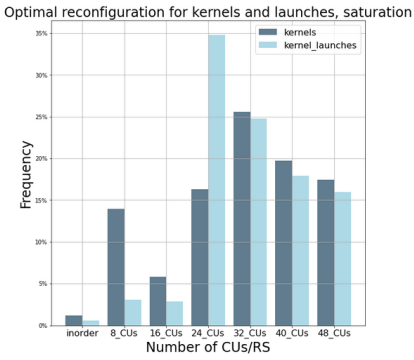
(B) PDP, kernels, LOOG-sensitivity



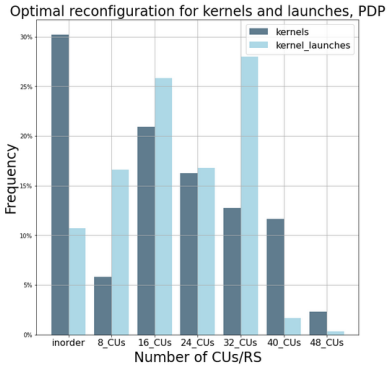
(C) sat98, kernels, LOOG-sensitivity



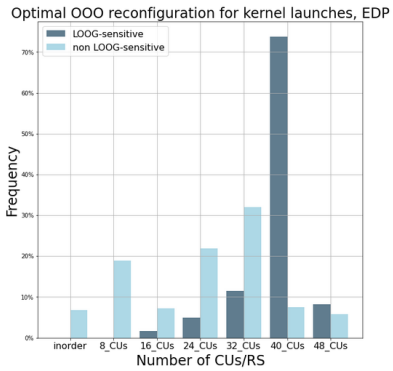
(D) PDP, launches, LOOG-sensitivity



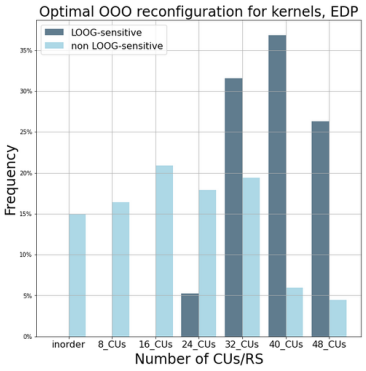
(E) sat98, kernels and launches



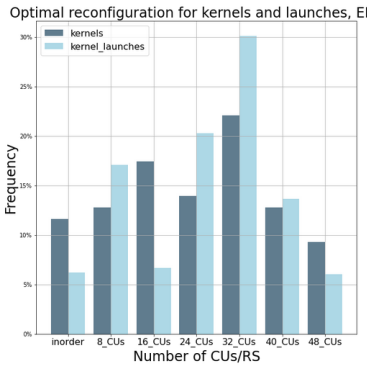
(F) PDP, kernels and launches



(G) EDP, launches, LOOG-sens



(H) EDP, kernels, LOOG-sens



(I) EDP, kernels and launches

FIGURE 5.29: Optimal CU configurations across kernels or launches, reconfiguration metrics and LOOG-sensitivity classes

it. Whole kernels and launches are evenly distributed across configurations, with the median optimal value for both being 24 CUs, as seen in Figure 5.29i, while LOOG-sensitive kernel distribution is zero up to 16 CUs and LOOG-insensitive whole kernel distribution is uniform in low configurations, as depicted in Figure 5.29h. In Figure 5.29g, it is apparent that LOOG-sensitive launches are heavily biased towards 40 CUs, while the median value for LOOG-insensitive launches is 24 CUs.

The above observations are summarized in Table 5.13:

Values	98% saturation	PDP	EDP
LOOG-sensitive			
min	24	16	24
max	48	48	48
median	40	32	40
LOOG-insensitive			
min	inorder	inorder	inorder
max	48	48	48
median	32	16	16

TABLE 5.13: Distribution parameters for optimal configurations based on the provided metrics across kernels

Presenting the respective kernel launch distribution table would be superfluous as the launch distribution limits are identical to whole kernels and medians are skewed. Having performed optimal static reconfiguration on all 100 kernels tested, based on all the optimization metrics defined, we analyze the average delay improvement and energy dissipation change both for single-launch and multi-launch kernels. Plots 5.31a and 5.31 clearly depict the observation that multi-launch kernels tend to not be LOOG-sensitive. Static reconfiguration average Delay and Energy similarly differs for single- and multi-launch kernel classes across all metrics , manifesting this inclination. It is further observed that all metrics yield similar results, except 98% saturation which is the most strictest performance metric, yielding the top performance improvement at -25% Delay for multiple-launch kernels and -31% for single launch kernels and the lowest Energy improvement at -19.4% for single launch kernels and -13.5% for multiple-launch kernels. Evidently, the less LOOG-sensitive, on average, multi-launch kernels are better suited in smaller CU configurations but the single launch kernels improve enough in higher configurations, to offset the power overhead to negative with more significant performance gains. The highest energy improvement is, as expected, provided by the PDP metric at -22.3% for single launch and -16.3% for multiple-launch kernels.

Single launch kernel optimization of PDP,EDP,95% saturation

Examining the behavior of single-launch kernels more closely, we collect statistics on delay deterioration for the optimization of all the other metrics, when comparing to the most

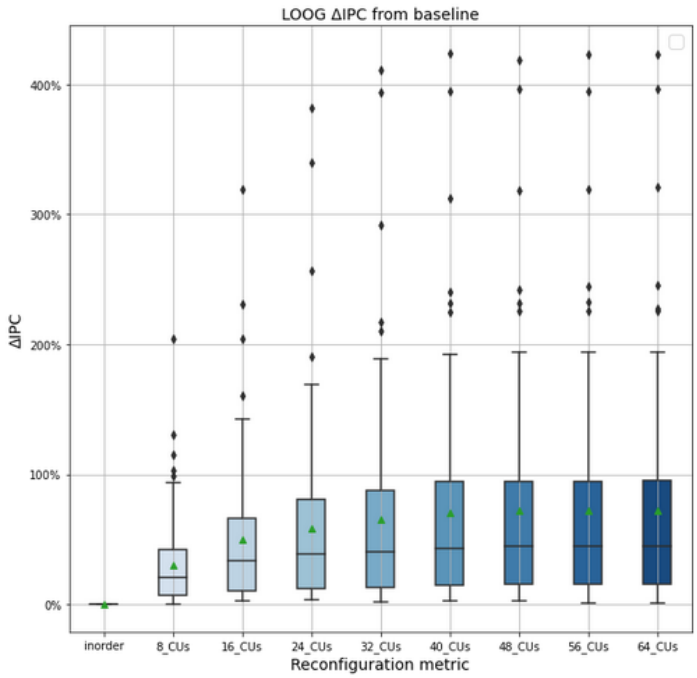
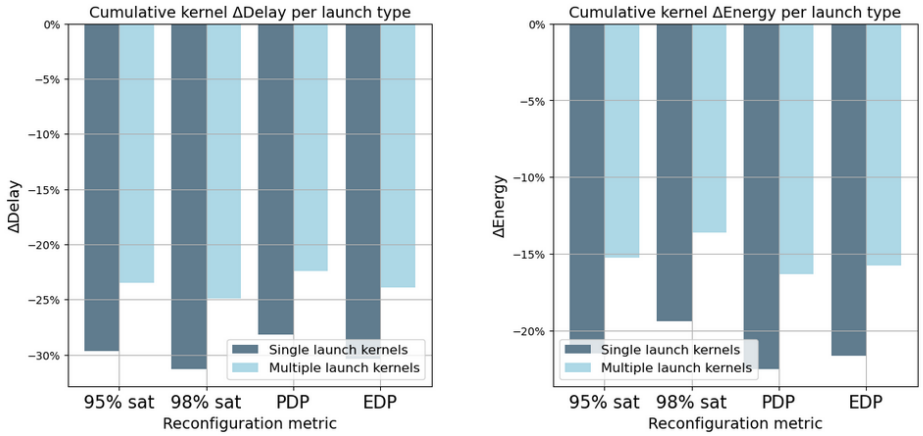


FIGURE 5.30: LOOG DeltaIPC from the baseline (inorder) microarchitecture

strict performance metric, 98% IPC saturation. Since single launch kernels are biased towards LOOG-sensitivity, we expect optimization of power metrics to frequently coincide with IPC saturation. As seen in Figure 5.32a, optimizing for the other metrics, causes an expected slight deterioration in delay compared to 98%sat. A 3.4% deterioration is observed for both 95% saturation and PDP metrics, while a 1.4% deterioration is observed for EDP. It is worth noting that in both the distributions factoring in power there are some outliers not seen in the saturation distribution, due to LOOG-insensitive kernels that do not gain enough of a speed-up, and occupy smaller configurations to optimize power. As for the power delta, in Figure 5.32 we observe that an average - 1.1%, -1.4% and -0.9% energy delta is produced by the 95% IPC saturation, PDP and EDP metrics respectively. The above positive delay deltas and negative delay deltas are produced by the smaller LOOG configurations kernels are forced to occupy when optimizing metrics other than top IPC saturation. Optimal configurations for outliers of these distributions are in Table 5.14. As expected, these kernels do not gain a sufficient OOO performance increase in order to minimize their PDP in any LOOG configuration. Optimally configuring for them, comes with a mediocre delay overhead of less than 18% and their energy overhead is avoided.

Single launch kernels optimal reconfiguration

Optimally reconfiguring for single launch kernels based on optimization of all of the metrics defined, we receive results for mean delay improvement and mean energy efficiency across all kernels and LOOG-sensitivity classes in Table 5.15.

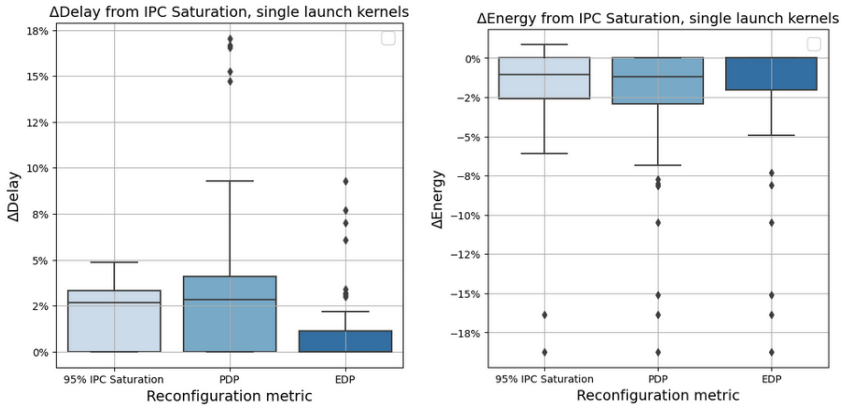


(A) Delay improvement (B) Energy improvement

FIGURE 5.31: Whole kernel (static) reconfiguration improvement per kernel type regarding launches

	98% Saturation	PDP	EDP	95% Saturation
testJoin_4	48_CUs	inorder	48_CUs	32_CUs
testJoin_3	48_CUs	inorder	inorder	40_CUs
lonestar-bfs-wlc_1	48_CUs	inorder	16_CUs	24_CUs
lonestar-bfs-atomic_1	32_CUs	inorder	32_CUs	24_CUs
lonestar-bfs-wlc_2	40_CUs	inorder	inorder	inorder
testJoin_2	48_CUs	inorder	inorder	inorder
lonestar-sssp_1	40_CUs	inorder	24_CUs	24_CUs
polybench-gramschmidt_2	40_CUs	inorder	inorder	40_CUs
lonestar-bfs-wlw_2	48_CUs	inorder	16_CUs	32_CUs

TABLE 5.14: Outliers of the distributions in Figures 5.32a and 5.32 that optimize energy efficiency in the inorder configuration



(A) Delay overhead from 98% IPC saturation (B) Energy improvement from 98% IPC saturation

FIGURE 5.32: Energy improvement and Delay deterioration when optimizing metrics other than 98% IPC saturation

	95% sat	98% sat	PDP	EDP
Delay				
Total	-29.4%	-30.1%	-28.2%	-30%
LOOG-sensitive	-59.8%	-60%	-60%	-60%
Energy				
Total	-20%	-19%	-21%	-20%
LOOG-sensitive	-53%	-52.9%	-52.8%	-52.8%

TABLE 5.15: Single launch optimal reconfiguration results relative to baseline across metrics and LOOG-sensitivity classes

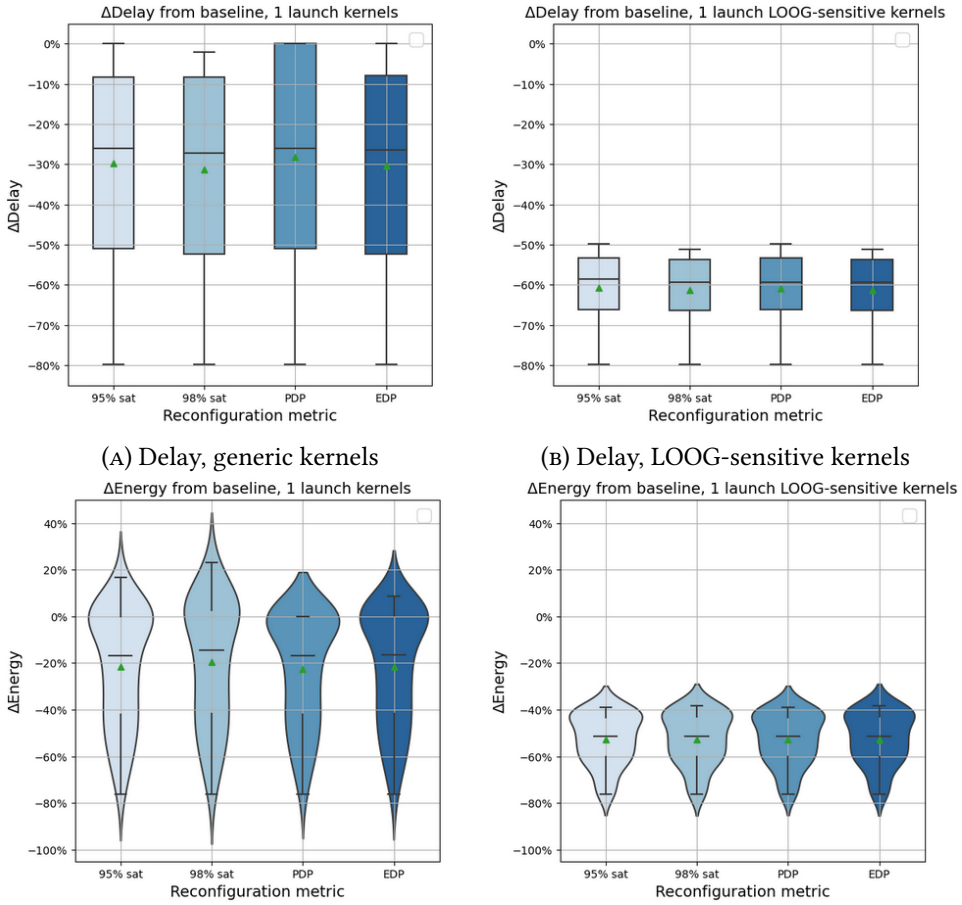
It is noted that as seen in the Energy distributions in Figure 5.33c, apart from the expected PDP optimization, a set of kernels does not gain a significant enough speedup in order to offset the power overhead of bigger LOOG configurations. These kernels represent 25% for the 95% saturation and EDP metrics, and 27% for the 98% saturation metric. As seen in Figure 5.33d, all LOOG-sensitive kernels reduce Energy dissipation by at least 38% when optimally configured.

Multiple launch kernels

In this subsection, we determine whether optimally profiling a multi-launch kernel based on cumulative characteristics on its whole execution is sufficient for optimally reconfiguring for all of its launches individually. Stated differently, we investigate the inter-kernel-launch scalability behavior consistency with LOOG and compare the optimal whole kernel reconfiguration to the optimal semi-dynamic reconfiguration seen in Figure 5.28.

In order to evaluate static reconfiguration schemes, it is imperative to examine the potential gain from optimal semi-dynamic reconfiguration. Delay and Energy statistics were collected on this basis and compared to the static reconfiguration optimal results.

The median value for kernel launches of multi-launch kernels is 7. The relevant statistics were analyzed for two groups of multi-launch kernels, above and below that threshold, to determine compare the improvement deltas on a both a coarser- and finer-grain reconfiguration scheme. As seen in Figures 5.34a and 5.34b, the distribution averages of deltas for delay and energy when configuring on a per-launch basis and using the appropriate metric (IPC saturation for delay and PDP for energy) do not differ significantly from static reconfiguration. A -0.7% and a - 1.1% delay is provided for multi-launch kernels with less than 7 launches and multi-launch kernels with more than 7 launches, on average, respectively. A - 0.8% and a -0.9% energy overhead is avoided. Overheads of optimal static reconfiguration compared to optimal semi-dynamic reconfiguration for the LOOG-sensitivity classes do not significantly differ and are not plotted. A reason for this is the tendency towards LOOG-insensitivity for multi-launch kernels, and another is asymptotic delay behavior on scale-up configurations, as well as declining power ratios, making a one-off optimal configuration error less significant in scale-up LOOG. Apart from a few outliers with an exaggerated amount of kernel



(A) Delay, generic kernels

(B) Delay, LOOG-sensitive kernels

(C) Energy, generic kernels

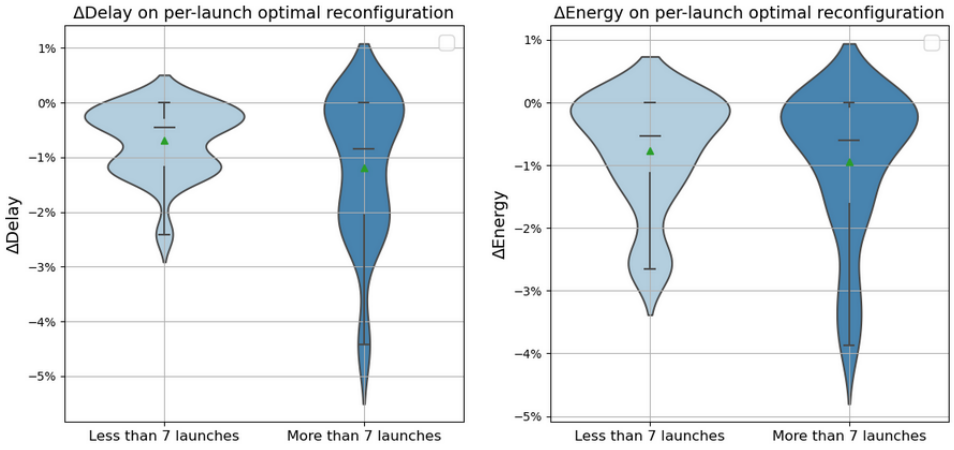
(D) Energy, LOOG-sensitive kernels

FIGURE 5.33: Single launch kernels optimal reconfiguration Delay and Energy improvement for generic and LOOG-sensitive kernels across reconfiguration metrics

launches seen in Table 5.16 , whose inputs and parameters possibly change over time, results are alike for the two groups based on number of launches. Evidently, even a small number of invocations is enough for the error due to imperfect reconfiguration to be completely amortized for most kernels. As expected, the distribution medians of energy and delay difference are slightly greater for kernels with more than 7 launches, owing to the fact that kernels with very few launches do not benefit as much from fine grain reconfiguration. It is also evident that simplifying the reconfiguration scheme by using cumulative kernel metrics comes with a mostly negligible overhead of 1% that only becomes significant in fringe cases.

Benchmark suite	Name	Description	Characteristics	Kernels	Launches	deltaDelay	deltaEnergy
Rodinia-3.1	NW	Sequence alignment	Compute intensive	1	104	-2.4%	-4.3%
Rodinia-3.1	CFD	Fluid Dynamics	Compute intensive	3	15	-2%	-3.5%
Polybench	3DConvolution	3D filtering	Compute & BW	1	32	-1.8%	-3%
SHOC	Spmv	Sparse Vector Mul	Sparse lin Algebra	1	129	-2.1%	-4.1%
SHOC	Scan	Parallel Scan	Memory BW	3	45	-1.8%	-3.1%

TABLE 5.16: Multi-launch kernels with inconsistent OOO scalability behavior among launches



(A) Delay (B) Energy

FIGURE 5.34: Per-launch (semi-dynamic) optimal reconfiguration Delay and Energy improvement compared to optimal static reconfiguration. This finer granularity is not worthwhile

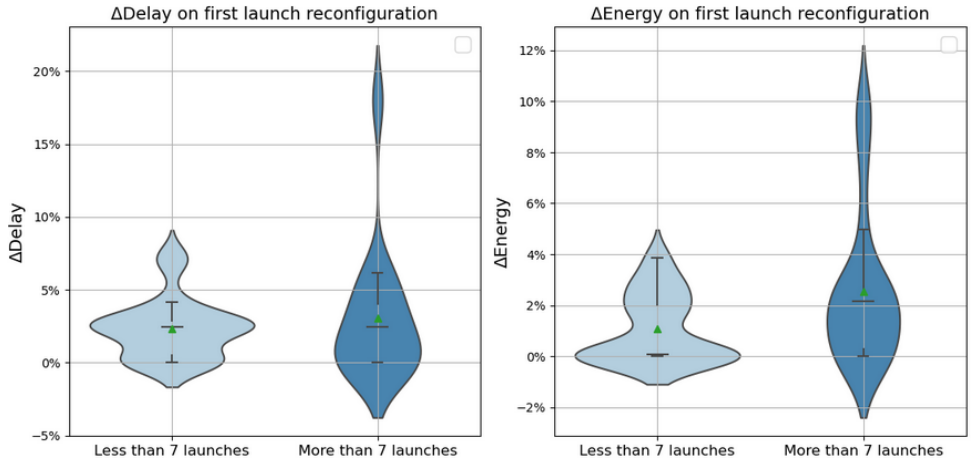
Considering these results, it is sensible to examine whether optimally profiling the first launch provides a decently suited configuration for the rest of the kernel's execution. As depicted in Figure 5.35a, a 2.6% and a 3.9% average delay overhead is introduced for multi-launch kernels with less than 7 launches and more than 7 launches respectively, when applying first-launch optimal reconfiguration compared to optimal static reconfiguration. The respective energy overheads are 1.2% and 2.6%, as seen in Figure 5.35b. This motivates the implementation of a hardware reconfiguration controller that profiles the first launch of a kernel regarding OOO scalability and applies the configuration best suited to it, to the rest of its execution. This implementation is presented in Section 5.8.6.

5.8.6 Predicting optimal configurations at runtime

As explained in Section 5.8.2 and seen in Figure 5.28 , kernel launches represent the finest granularity for microarchitecture reconfiguration that does not impose significant delay overheads, as reconfiguration can happen in Idle periods of the GPU between launches. As outlined in Section 5.6.3, optimal configurations for each application based on our defined metrics (IPC saturation, PDP, EDP) essentially depend on two application characteristics:

- Application OOO scalability, essentially measured by its saturated IPC improvement on LOOG.
- Rate of performance saturation, referring to the scale-up LOOG configuration beyond which virtually no performance improvement is obtained. Such varying rates can be seen in Figures 5.29e, 5.29c and 5.29a.

Considering that optimal static (whole-kernel) reconfiguration and optimal first-launch reconfiguration closely approximate optimal semi-dynamic reconfiguration (as seen in Figures



(A) Delay overhead (B) Energy overhead

FIGURE 5.35: First-launch optimal reconfiguration Delay and Energy overheads compared to optimal static reconfiguration

5.34 and 5.35), we set out to implement a predictor that uses hardware performance counters on runtime to output measures for both the above parameters and, thus, perform first-launch reconfiguration. Each kernel's first launch is initially run on the in-order configuration. The aforementioned performance counters are collected throughout its execution. The hardware reconfiguration controller is essentially a multivariate regressor fitted on these counters, that predicts application behavior in scale-up configurations as explained in the rest of this subsection. It would be sensible to treat this problem as a classification problem, since the nature of target variable is categorical (7 possible configurations). However by treating it as a regression problem, we exclude the reconfiguration metric that should be taken into account in categorization, requiring a separate model for each metric. We only factor it in after having predicted scalability behavior. Moreover, the regressor is less error-prone due to the asymptotic nature of the target variable. This method exclusively concerns multiple-launch kernels, but can be extended to single-launch kernels in the form of an online dynamic reconfiguration controller. A meticulous reconfiguration overhead study would be required as well as determining optimal sampling periods, switching control algorithms and break-even periods.

Predicting application OOO scalability

Multiple regression models were tested to predict saturated LOOG improvement (essentially a measure of OOO scalability) from the metrics collected on all levels of workload characterization. The cross validation RMSE scores for all the models fitted are plotted in Figure 5.36. Due to the non-linear correlations in the data, and its categorical aspect, Decision Tree and Random Forest models provide the best fitting. Decision tree is selected due to ease of implementation.

A linear regression model was initially tested, with its most important features depicted in Figure 5.37. Features with negative coefficients are colored red. As extensively discussed

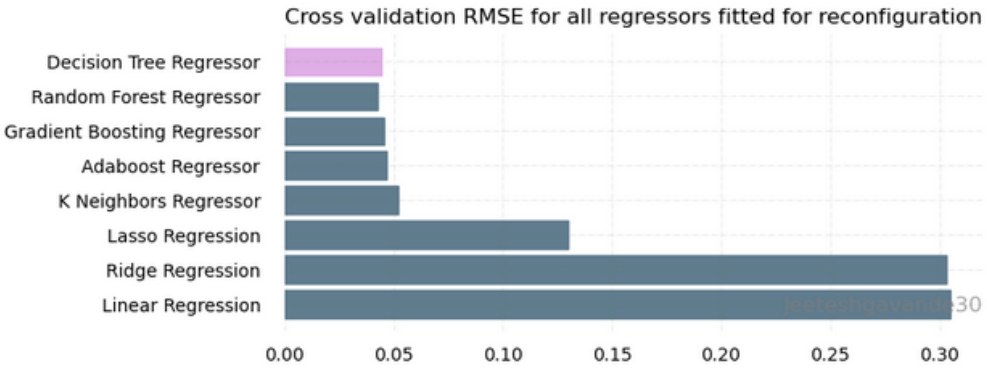


FIGURE 5.36: Cross validation RMSE scores for all regressors fitted on the data

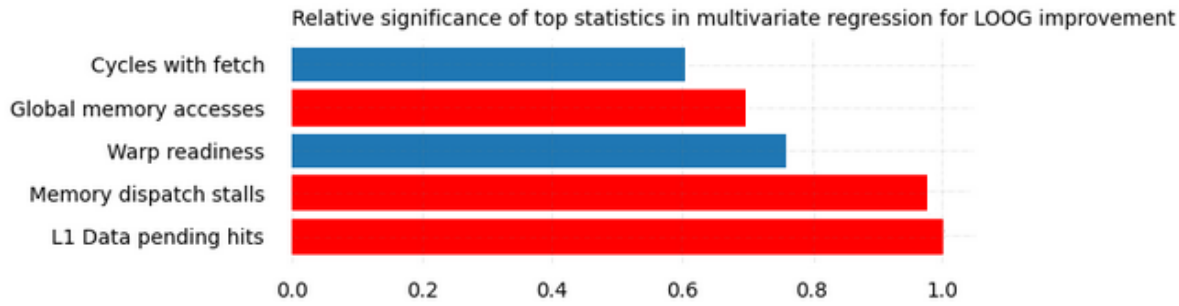


FIGURE 5.37: Most important features for multivariate linear regression fitted on the data

in Section 5.3, runtime statistics associated with memory accesses (Global accesses, MEM dispatch stalls, L1 Data pending hits -requests for in-flight misses-) are negatively correlated with speedup on LOOG configurations, while those signifying high instruction throughput are positively correlated. Nevertheless, the Linear Regressor provided the worst results, as seen in Figure 5.36. The correlation of the target variable to the Regressor's most important feature, seen in Figure 5.38 is hardly linear.

The train set comprised all 42 single-launch kernels as well as 28 multi-launch kernels and the test set contained the remaining 40 multi-launch kernels. Fitting was performed on cumulative whole-kernel statistics (averaged out over all launches for multi-launch kernels). To avoid overfitting and due to the prohibitive size of the training data for cross-validation, the model selected provided an RMSE within 5% of the median of 100 fittings (final RMSE of 0.12 and a MAPE of 27%).

Predicting performance improvement for intermediate LOOG configurations

In theory, we could fit a regression model for each intermediate configuration in the range [8_CUs, 48_CUs] to predict the respective speedups. However, this is both impractical and superfluous as is ascertained below. Motivated by the observation that normalized performance improvement scaling with LOOG is similar among kernels, as seen in Figure 5.39 we try to predict intermediate configuration performance improvement given the saturated IPC

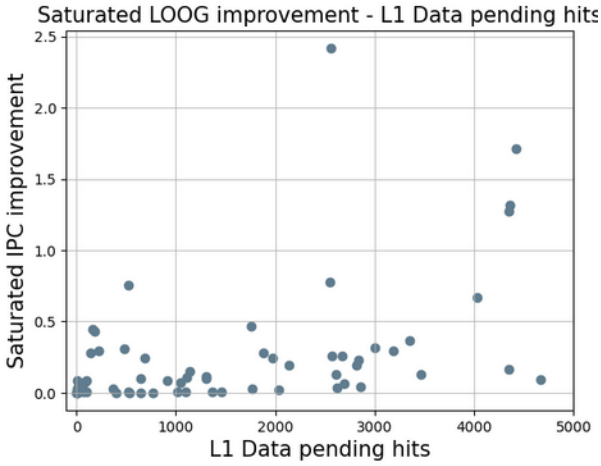


FIGURE 5.38: Correlation of L1 Data pending hits to saturated IPC improvement

improvement, and the mean normalized IPC improvement curve (dots depicted in Figure 5.39). Initially, predicting saturated IPC improvement (essentially IPC improvement on a 64 CU configuration) was thought to be enough in order to determine application scalability in intermediate configurations. We use known values of the saturated improvement instead of those predicted by the regressor, in order to assess the optimal attainable results. IPC improvement at the 64 CU configuration is set to the known saturation value. Intermediate values are calculated by multiplying the mean normalized IPC improvement at each configuration with the saturation value. The results are depicted in Figure 5.40b. Evidently, the errors are beyond acceptable margins for 8 CUs.

Predicting the rate of performance saturation

Considering that application scalability (and its corresponding saturated IPC improvement measure) is not sufficient in order to predict intermediate configuration speedup for the smaller LOOG configurations, we implement a second predictor, with IPC improvement at the minimum LOOG configuration of 8 CUs as its target variable. Given the saturated IPC improvement prediction, this second predictor essentially estimates the rate of performance saturation for LOOG scaling. When the predicted speedup values at 8 CUs are low, saturation is slower and vice versa. Speedups at intermediate configurations are calculated as described before, using the mean speedup curve in the range of [8_CUs, 48_CUs]. Results are depicted in Figure 5.40a.

5.8.7 Hardware reconfiguration controller design

The decision trees for the minimum LOOG IPC improvement prediction and saturated IPC improvement prediction are depicted in Figures 5.41a and 5.41.

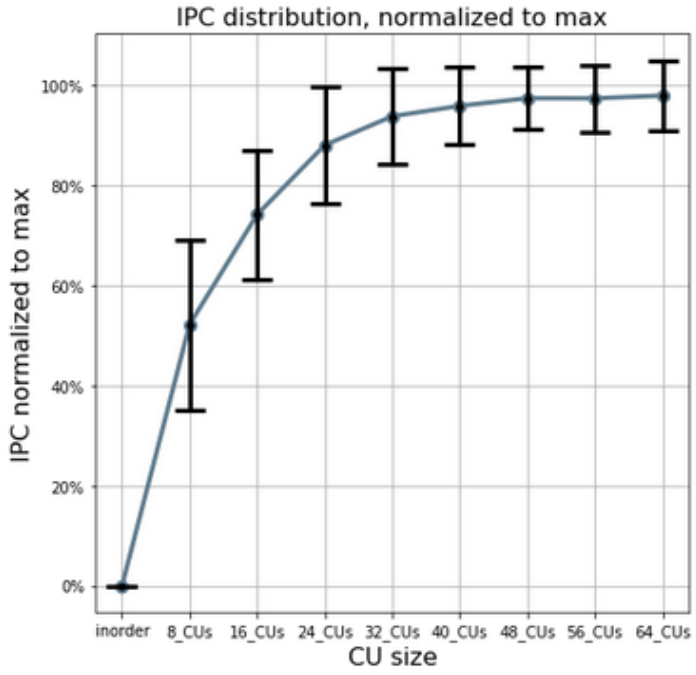


FIGURE 5.39: Normalized IPC improvement curve and STD across kernels on LOOG scaling-up

Performance counters utilized by the decision trees

Regarding the minimum LOOG IPC decision tree predictor in Figure 5.41a, the following hardware performance counters resulted by fitting the model:

Reply network active cycles Leading to significantly diverging predicted IPC values, it refers to the active cycles of the virtual Reply network implemented physically by the interconnect seen in Figure 3.5. High values equate to memory traffic and therefore, lesser speedups.

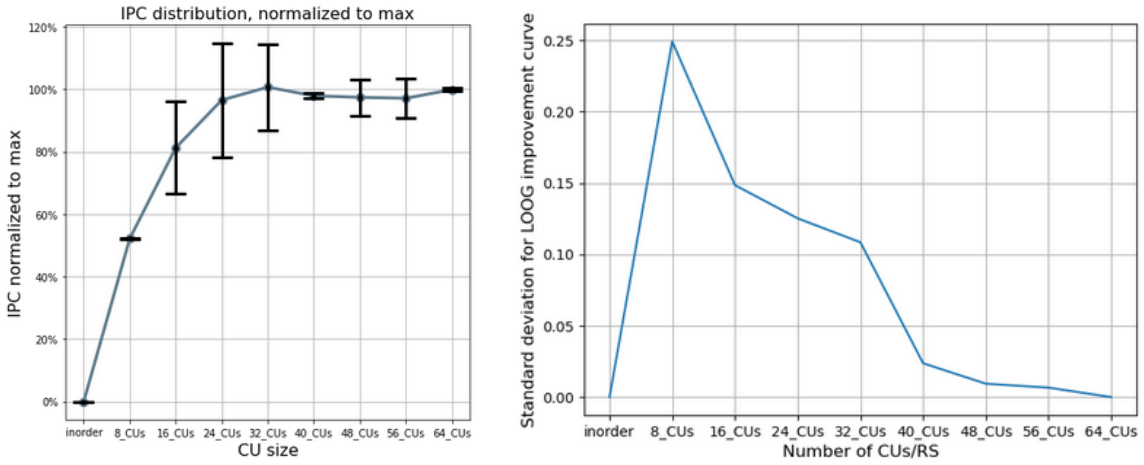
Warp occupancy Referring to the average number of active threads per warp in the scope of this thesis. High values equate to low warp divergence and less control hazards, improving speedup on LOOG.

RET commissions Dynamic instructions for returning from subroutines. Similarly equate to control hazard stalls, preventing LOOG from providing significant acceleration.

Control hazard stalls Cause acceleration deterioration on LOOG as explained.

Total warp instructions Are directly correlated to the kernel's runtime. As elaborated on in Section 5.8.2, low kernel runtimes equate to worse speedups on LOOG due to the dynamic instruction mix being occupied by parameter memory instructions.

In the saturated LOOG IPC improvement decision tree, the metrics below are utilized in addition to the above:



(A) STD of the IPC values predicted from the controller across CU configurations

(B) Correlation of L1 Data pending hits to saturated IPC improvement

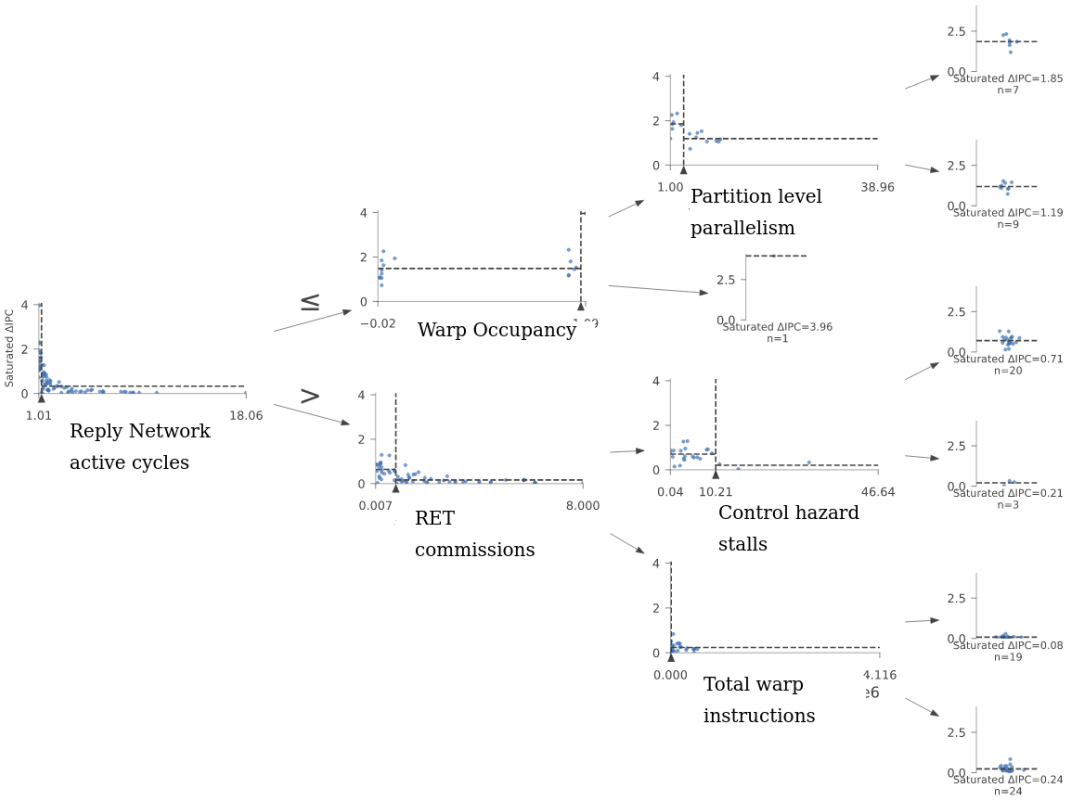
Total cycles Cause deterioration of LOOG acceleration as explained above.

GPU occupancy Essentially refers to the average dynamic warp occupancy while also accounting for pipeline stalls. Identically affects the speedup.

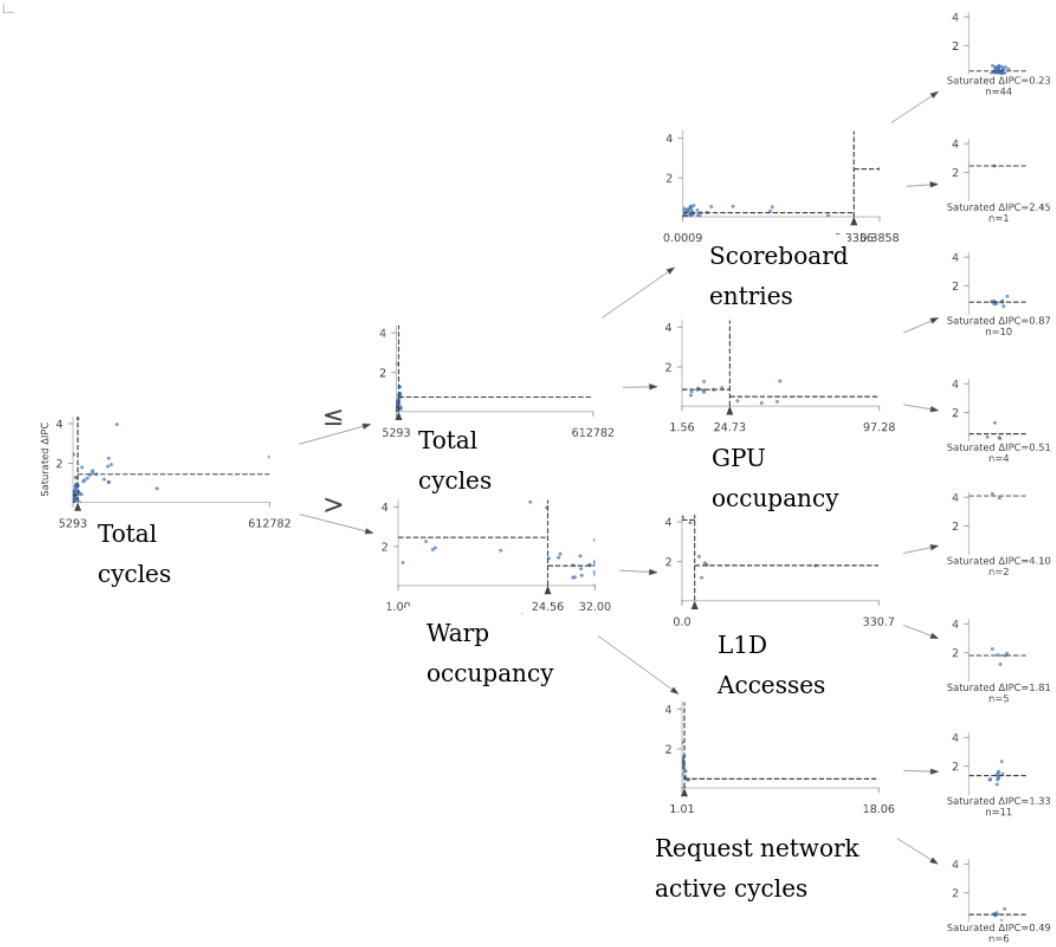
L1D Accesses Is used to predict the highest of values. Equates to high memory accesses, therefore a memory intensive kernel, which is negatively correlated with LOOG-sensitivity.

Scoreboard entries used Signify dependencies between instructions from the same stream (at a depth of 1 in the baseline implementation), which are, by definition, negatively correlated to ILP, and therefore speedup on LOOG.

The design of the hardware reconfiguration controller is depicted in Figure 5.42. As seen in the above figure, the first kernel launch is executed in the inorder configuration. The hardware performance counters input to the decision trees are collected. Threshold values used by the decision trees remain constant and are hardcoded into them. For each Metric - Hardcoded threshold value pair a comparator is needed in each of the two decision trees. Each decision tree leaf corresponds to the Boole product of at most three comparisons. The respective bitlines are fed to a ROM module containing the hardcoded output values, replacing the decoder in the typical implementation. The outputs of the decision trees are fed to a multiplier predicting intermediate configuration values based on the hardcoded mean IPC improvement curve. The result is multiplied with the estimated hardcoded max power vector according to the reconfiguration input to the model. If IPC saturation is the selected optimization metric, the predicted IPC vector is differentiated and compared with the 2% or 5% threshold at each stage. The CU value optimizing the metric is determined (minimum for PDP, EDP, first negative value for IPC saturation) and for subsequent launches, the microarchitecture is reconfigured according to it. Output values for each decision tree of the reconfiguration controller are presented in Table ??.



(A) Minimum LOOG IPC improvement (predicting the rate of performance saturation)



(B) Saturated IPC improvement decision tree (predicting OOO scalability)

FIGURE 5.41: Fitted decision trees predicting LOOG performance on the most scale-down (8 CUs) and the most scale-up (48 CUs) configuration

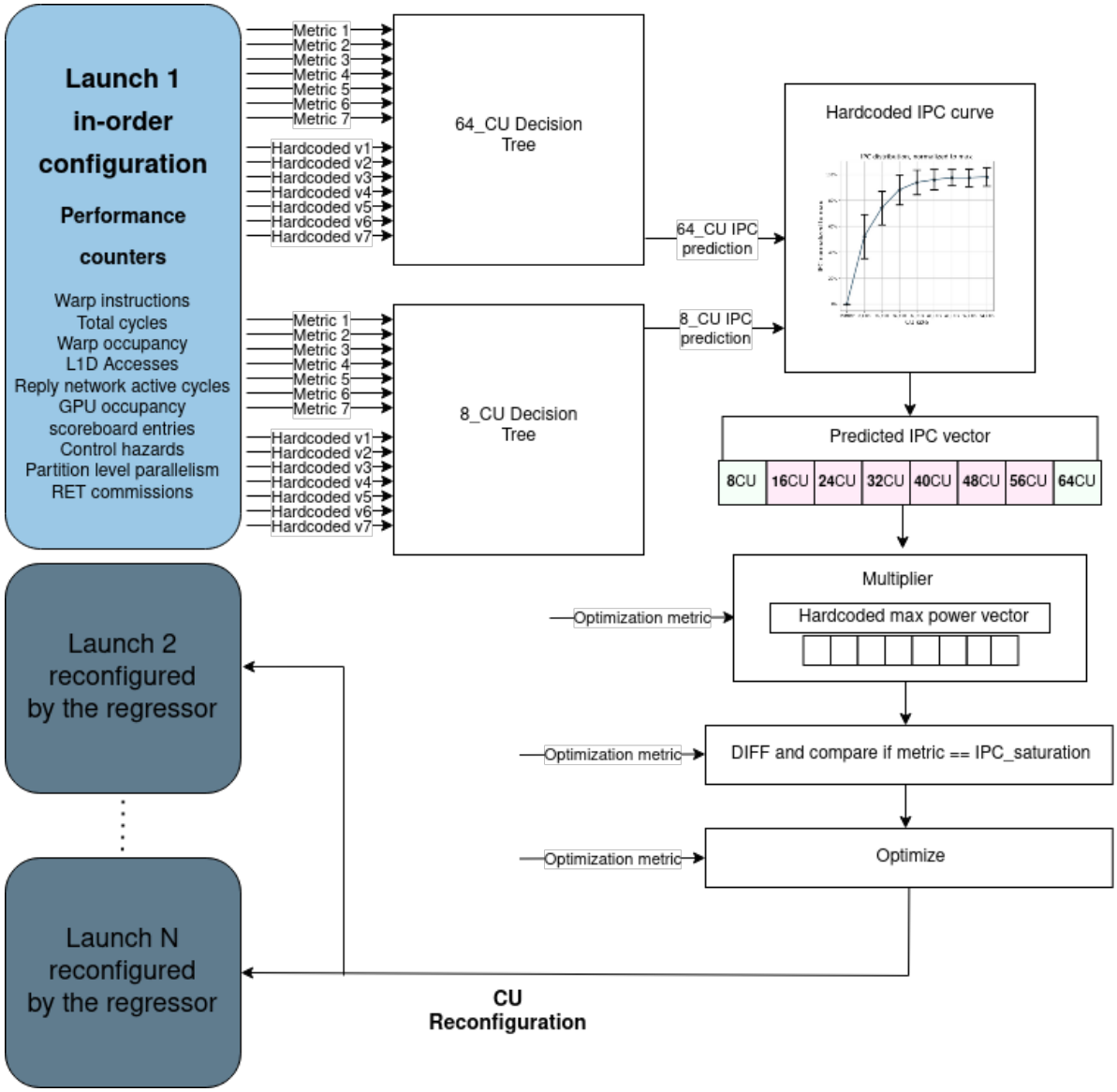


FIGURE 5.42: Design of the hardware reconfiguration controller

Predictor	RMSE	MAPE	Leaf values								Test set mean	Test set 90th
8_CU regressor	0.08	21%	0.08	0.21	0.24	0.71	1.19	1.85	3.96	-	0.18	0.57
64_CU regressor	0.12	27%	0.23	0.49	0.51	0.87	1.33	1.81	2.45	4.1	0.61	1.52

TABLE 5.17: Errors and predicted IPC output values of the decision tree regressors

5.8.8 Power gating reconfiguration overhead estimations

As briefly discussed in Section 5.8.2, inter-launch reconfiguration provides us with significant time windows in order to amortize the Power Gating sleep transistor wake-up time overhead, thus reducing the associated energy dissipation. Since this window approaches several microseconds, which a high-end value for fine-grain power gating, we may assume that no interference with workload execution will take place even in the sem-dynamic per-launch reconfigurable architecture and the energy overhead will be negligible. A 3.09% and 9.28% static and maximum dynamic power is respectively produced by Accelwattch using the 23nm node technology configuration. As seen in Table 5.2, the actual node technology is 12mnm, which according to [107], would double the subthreshold leakage and quadruple subthreshold power, being the most significant component by an absolute factor of 81% over total leakage power in our simulation. Therefore, in the Volta architecture, subthreshold leakage power is expected to be about equal to maximum dynamic power. In Flicker [11], 3 pipeline stages over 4 lanes for a total of 12 fine-grain structures per core are dynamically power gated, compared to 6 structures per core in our reconfigurable microarchitecture, that can share the same virtual ground for respective structures across SMs due to workload homogeneity. Area overhead of the sleep transistors needed is overestimated at 2-6%, while a 90% reduction of static power id provided for an approximate 2% increase in dynamic power (due to supply voltage levels increase to power the sleep transistors and the related decoupling capacitance). We, therefore, infer that power gating the LOOG-related structures can be approximated by not accounting for them.

5.9 Speculating on other axes of reconfiguration

Motivated by the observations made in Section 5.3 regarding application diversity across different axes, we briefly speculate on the potential of a scalable reconfigurable architecture, performing static reconfiguration with fine-grain power gating on the Execution Units and Caches of the GPU. As described in Section 4.3, other power-aware scalable cores employ fine-grain per-component DVFS to maximize performance and energy efficiency [10] as well as unit-level clock or power gating to tailor the application to runtime demands regarding OOO scalability [**morph-core**] or pipeline width (OOO and concurrency scalability)[11]. With respect to the GPU, Equalizer [44] also leverages DVFS at a core/memory level granularity to suit the architecture to dynamic requirements, while Bahurupi [43] and Amoeba

[45] implement a core-fusion architecture to target bottlenecks regarding OOO scalability and various other architectural bottlenecks (such as cache contention) respectively. To the best of our knowledge, a reconfigurable architecture employing fine-grain power gating of Execution Units as seen in Flicker [11] and caches has not been implemented on the GPU. These structures were specifically selected upon discovering that they are a source of significant workload diversity, as seen in Section 5.3 and pose impactful respective pipeline bottlenecks, as seen in Section 5.2. In order to briefly speculate on the potential of such an architecture, we collect runtime statistics with GPGPU-Sim across different Execution Unit (EXU), cache and Collector Unit (CU) configurations, as well as area and maximum power dissipation with Accelwattch.

5.9.1 Fine-grain Caches and Execution Units scaling

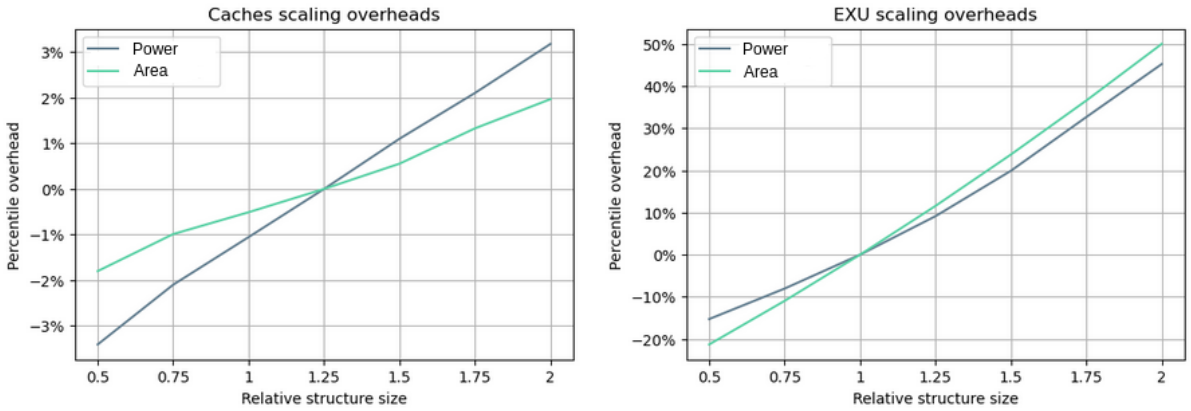
Similarly to our method in Section 5.6.3 , we first determine the Power and Area overheads when scaling the respective structures as seen in Table 5.18. These values represent the GPGPU-Sim configuration values corresponding to our modifications. When configuring Accelwattch to obtain Power and Area overheads we linearly extrapolate upon them, as explained in Section 5.5. Note that the absolute values of the components used to configure GPGPU-Sim and Accelwattch do not necessarily correspond to the actual Volta architecture values, rather, they are fine-tuned to simulate its performance across various workloads.

Caches Scaling							
Relative size	Unified L1D	L1D banks	L1 DCache	Shared mem	ICache	Texture cache	Constant cache
0.5	64	2	16	48	64	64	32
1	128	4	32	96	128	128	64
2	256	8	64	192	256	256	128
EXU Scaling							
Relative size	Pipeline width	EX/WB width	SP units	DP units	INT units	SFU units	Tensor core units
0.5	2	4	2	2	2	2	2
1	4	8	4	4	4	4	4
2	8	16	8	8	8	8	8

TABLE 5.18: Caches and Execution Units scaling configurations

In Figures 5.43a and 5.43b the Caches and Execution Units scaling overheads are displayed. Evidently, Execution Unit overheads are much more significant and all scaling is approximately linear.

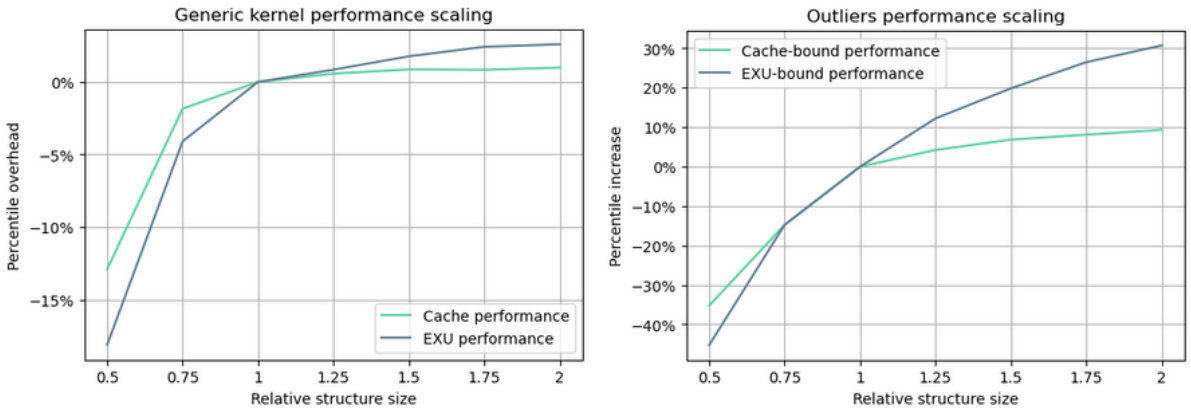
In Figure 5.44a, the average kernel performance increase with component scaling is displayed. Evidently, generic kernel speedup upon scaling the respective structures is minimal. In Figures 5.45a and 5.45b the histograms of kernel speedup on the most scale-up configurations tested (2x) are displayed. It is apparent that contrary to the negligible speedup obtained for most kernels, a select few exhibit marked performance improvement. We define the class of kernels exhibiting a speedup of at least 4.5% with cache scaling as Cache-bound (12 resulting kernels) and the class of kernels exhibiting a speedup of at least 20% with EXU scaling as EXU-bound (10 resulting kernels). Kernels belonging to these classes are listed in Table



(A) Caches scaling (B) EXU scaling

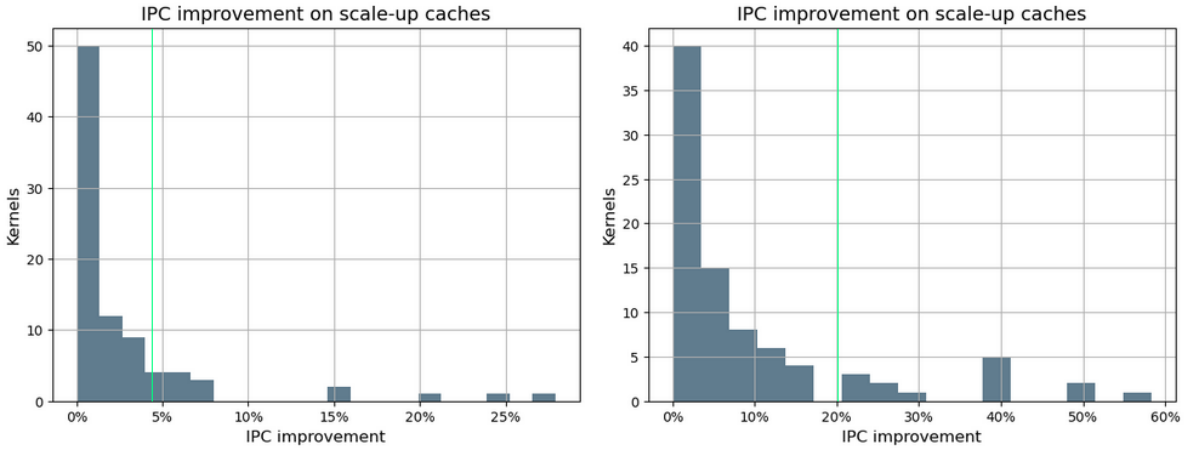
FIGURE 5.43: Power and Area overheads of scaling Caches and Execution Units

5.19 and the overlap among them and the LOOG-sensitive class is depicted in Table 5.20. Evidently, half of the EXU-bound kernels are LOOG-sensitive compared to a quarter of the Cache-bound kernels. This is expected since the EXU-bound kernels are by definition compute intensive, while the LOOG-sensitive cache-bound kernels belong in the Cache-bound, high ILP cluster seen in Figure 5.11. Kernels that belong to both the EXU-bound and Cache-bound classes, are all produced by the lonestar-mst benchmark. They also belong to the Cache-bound, high ILP cluster, which is in line with our analysis in Section ??. Since this cluster contains applications that comprise separate high hit-rate cache access phases and compute phases, it is expectedly accelerated by scaling both structures. The speedup of the highly accelerated outliers seen in Figures 5.45a and 5.45b is plotted in Figure 5.44b. Evidently, performance increase for these kernels is not saturated even when doubling the size of relevant components, contrary to generic kernels in Figure 5.44a which seem to saturate in the baseline configuration. This motivates the analysis taking place in the next subsection.



(A) Generic kernels (B) High speedup outliers

FIGURE 5.44: Performance improvement for generic and highly improving kernels with Cache, EXU scaling



(A) Scale-up Cache configuration

(B) Scale-up EXU configuration

FIGURE 5.45: Performance improvement on the most scale-up Cache and EXU configuration

Kernel	DeltaIPC
Cache-Bound	
lonestar-mst_4	20,83%
lonestar-mst_3	18,30%
lonestar-mst_5	15,87%
lonestar-mst_6	11,52%
ispass-2009-AES_1	11,01%
shoc-Spmv_1	5,75%
cfd-rodinia-3.1_2	5,69%
testJoin_1	5,38%
polybench-correlation_1	4,93%
shoc-spmv-modified_1	4,91%
cfd-rodinia-3.1_3	4,82%
shoc-spmv-modified_2	4,70%
EXU-Bound	
testAmr_3	43,56%
hotspot-rodinia-3.1_1	37,38%
shoc-S3D_1	36,74%
shoc-S3D_2	30,20%
lonestar-mst_4	30,16%
lonestar-mst_5	29,59%
shoc-BFS_1	28,72%
lonestar-mst_3	28,19%
shoc-S3D_3	21,21%
lonestar-mst_6	20,30%

TABLE 5.19: Cache and EXU bound kernels (application_kernel-uid), along with speedup on the respective scale-up configurations

Class intersections	EXU-Bound	Cache-bound	LOOG-sensitive
EXU-Bound	10	4	5
Cache-bound	4	12	3
LOOG-sensitive	5	3	22

TABLE 5.20: Intersection of component-bound and LOOG-sensitive kernels

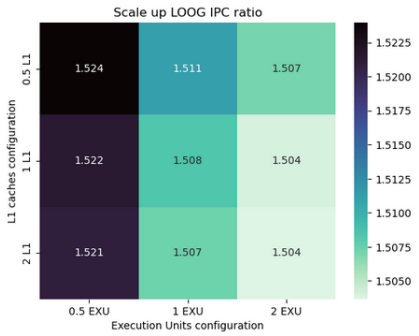


FIGURE 5.46: IPC ratio of scale-up to scale-down LOOG across Cache, EXU configurations

5.9.2 Component scaling design space

To further investigate the reconfiguration potential among the previously defined axes of application scalability, we co-scale the values in Table 5.18 with Collector Units set to [16,32,48], producing a design space with 27 points. The respective simulations on GPGPU-Sim and Accelwattch produce the results seen in Figure 5.47. As previously established and also seen in Figures 5.47 and 5.47, Area and Power overheads when scaling Execution Units are an order of magnitude greater than when scaling the Caches. The same is true for the speedup of the average kernel as seen in Figure 5.47. To examine the EXU and Cache scaling performance sensitivity of the applications relative to LOOG scaling, we divide the delays seen in the scale-down 16 CU block in Figure 5.47 with the respective scale-up 48 CU delays, effectively calculating the OOO scalability of the applications across the Cache and EXU size design space. Results are depicted in Figure 5.46. Evidently, a negligible but existent speedup saturation is observed in scale-up Cache and EXU configurations. That is to say, acceleration from LOOG scaling-up is lesser in scale-up configurations in the other axes. This is explained by the fact that a wider pipeline can already provide significant speedup in scale-down LOOG configurations without relying on leveraging ILP. When scaling up both EXUs and CUs, the combined OOO and TLP scalability of the application is exhausted to an extent.

For the average kernel, the speedup obtained is so minimal compared to the Power overheads that no EXU or Cache scale-up configuration optimizes even the EDP figure of merit, as indicatively displayed in Figure 5.47. Likewise, the performance deterioration offsets the power reduction in the scale-down configurations, rendering them disadvantageous. These conclusions arise from the fact that within each CU scaling block in Figure 5.47, the baseline configuration has the optimal value.

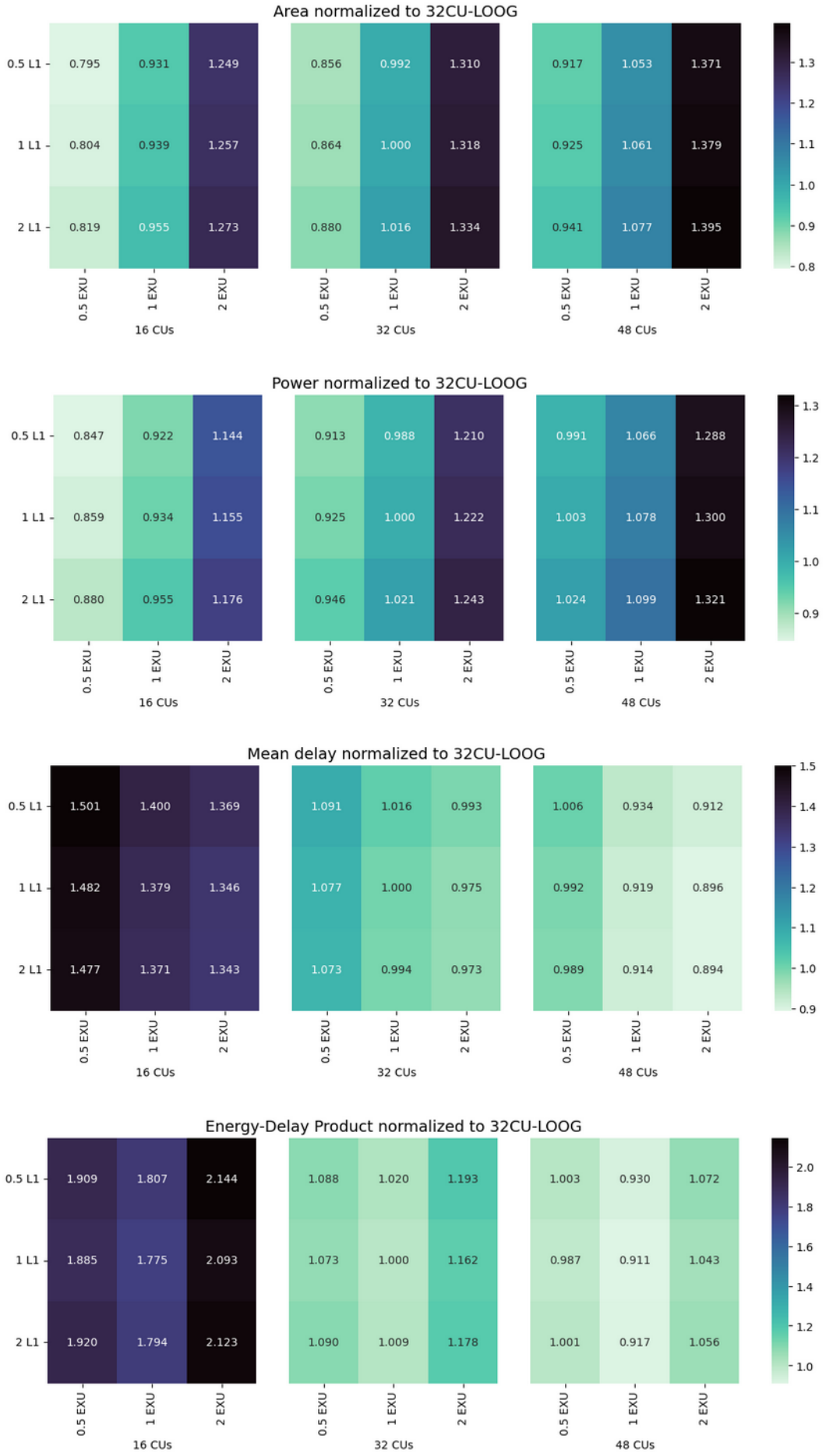


FIGURE 5.47: Figures of merit across the design space (Delay refers to the average kernel)

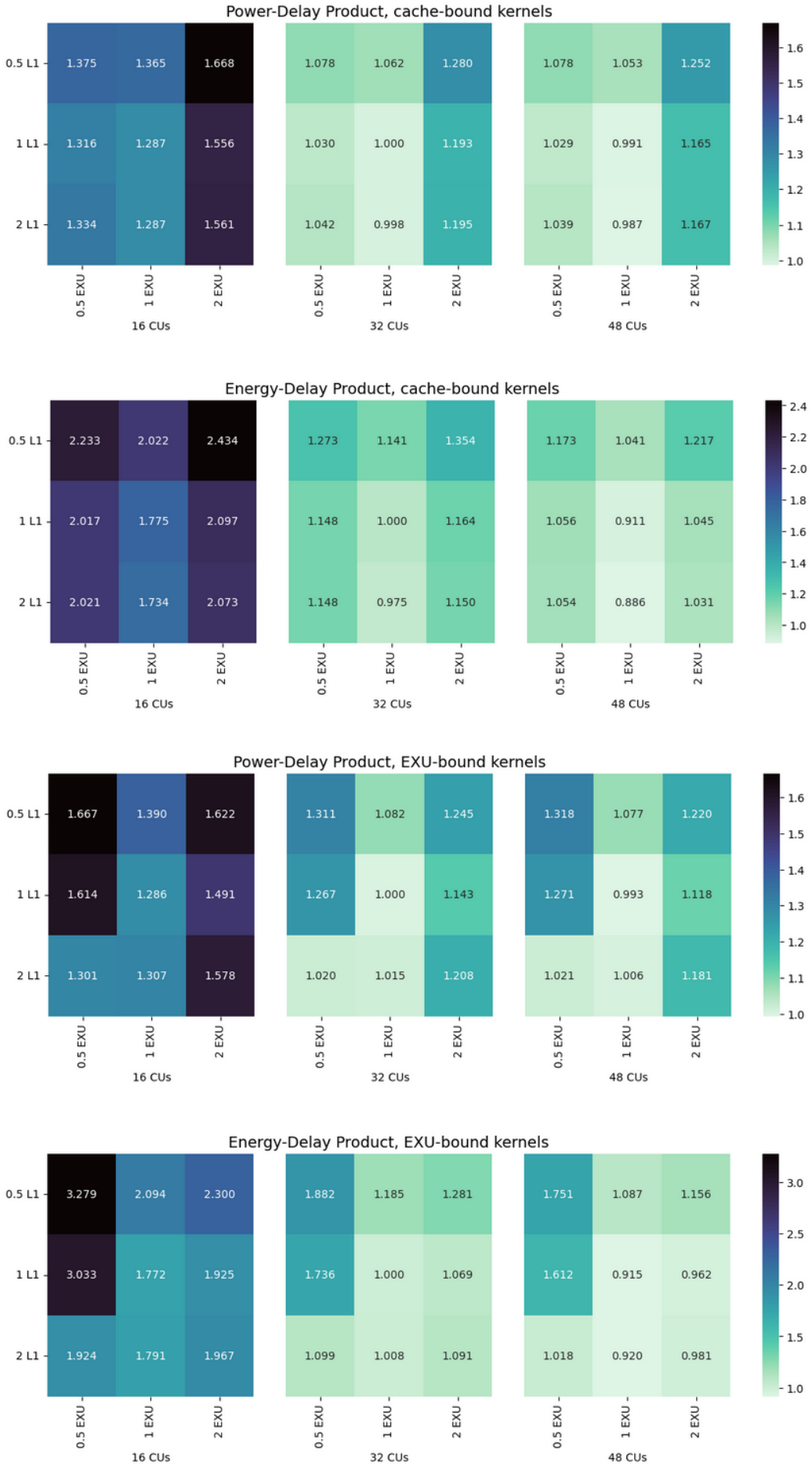


FIGURE 5.48: Figures of merit across the design space for Cache-bound, EXU-bound kernels (Delay refers to the average kernel)

Cache-bound kernels provide substantial speedups and marginally optimize the figures of merit in some EXU and Cache scaling configurations. In Figure 5.48, it is seen that the energy efficiency of the average Cache-bound kernel is insignficantly optimized in the Cache up-scale configurations for the 32 CU and 48 CU LOOG configurations. The respective EDP improvements amount to 2.50% and 2.45%. No improvement is seen for the EXU-bound kernels. It is important to note that while performance for the average Cache-bound and EXU-bound kernel saturates beyond the maximum scale-up configuration we examined (as seen in Figure 5.44b), the optimal figures of merit for most kernels likely lie in intermediate configurations. Due to practical limitations, a more detailed design space exploration surpasses the scope of this thesis, but we do not expect worthwhile reconfiguration potential regarding these axes.

Chapter 6

Evaluation of the OOO reconfigurable microarchitecture

6.1 Right-sizing the reconfigurable Operand Collector

As seen in Section 5.6.3, the most scale-up LOOG configurations utilizing many Collector Units come with significant Area and maximum Power dissipation overheads. Furthermore, the Area overheads of the sleep transistors servicing the coarse-grain power gating necessary for reconfiguration must be taken into account (although to the extend they are used in this thesis they are considered insignificant and precisely calculating them surpasses its scope). We determine the values of various efficiency figures of merit, normalized to baseline, when optimally reconfiguring across three different reconfigurable microarchitectures. Said microarchitectures have 16, 32 and 48 CUs in total. In Figure 6.2a the ADP metric, normalized to baseline, is provided across the examined microarchitectures and using PDP and EDP reconfiguration metrics. It is minimized on the most down-scale microarchitecture of 16 CUs with 32 CUs closely following. In Figures 6.2c, 6.2 and 6.2b, it is evident that all other metrics are optimized in the most scale-up, 48 CU configuration. However, the most significant drop happens from 16 to 32 CUs, with minimal improvement from 32 to 48.

Average values of optimal reconfiguration results for a given architecture per optimization metric are summarized in Table 6.1a , normalized to baseline (i.e. A uArch with 16 CUs, produces an average delay of 0.749 normalized to baseline when optimally reconfigured with a PDP metric). As also seen in the table, all metrics except ADP are minimized for maximum CUs. For all the figures of merit, the 32 CU configuration provides values approximating the optimal. As seen in Table 6.1b, for all the figures of merit used (Delay, Energy, ADP, PDP) the 32 CU microachitecture provides overheads within 2% of the optimal.

Evidently, for the average kernel, the 32 CU microarchitecture provides the most sensible trade-off between optimization of ADP and the other composite metrics. In Tables 6.1c and 6.1d, the respective optimal reconfiguration results and overheads are presented for the average LOOG-sensitive kernel. In this context, the 48-CU architecture provides the most efficient tradeoff between figures of merit, with zero overhead from the optimal for metrics that account for power, and less than 1.15% for ADP, while the 32-CU microarchitecture poses

signficant Delay and EDP overheads. Note that for LOOG-sensitive kernels, figures of merit are identical between reconfiguration metrics (PDP,EDP) in the 16-CU and 32-CU microarchitectures, since for both PDP and EDP, the most scale-up configuration available is optimal. Considering the above, the 48-CU microarchitecture was used in all further evaluation, to assess the microarchitecture reconfiguration potential for LOOG-sensitive kernels.

CU	16	32	48
ADP			
PDP	0,800	0,815	0,857
EDP	0,790	0,796	0,833
Delay			
PDP	0,749	0,716	0,710
EDP	0,739	0,699	0,690
Energy			
PDP	0,800	0,780	0,776
EDP	0,803	0,786	0,784
EDP			
PDP	0,599	0,558	0,551
EDP	0,593	0,549	0,541

(A) Figures of merit across microarchitectures for generic kernels.

CU	16	32	48
ADP			
PDP	0	1.87%	7.12%
EDP	0	0.76%	5.44%
Delay			
PDP	5.49%	0.85%	0
EDP	7.1%	1.3%	0
Energy			
PDP	3.09%	0.52%	0
EDP	2.42%	0.26%	0
EDP			
PDP	8.71%	1.27%	0
EDP	9,61%	1,48%	0

(B) Overheads compared to the minimum for each figure, generic kernels. 32 is the optimal tradeoff

CU	16	32	48
ADP			
PDP	0,582	0,533	0,537
EDP	0,582	0,525	0,531
Delay			
PDP	0,545	0,462	0,445
EDP	0,545	0,462	0,440
Energy			
PDP	0,599	0,536	0,528
EDP	0,599	0,536	0,530
EDP			
PDP	0,346	0,270	0,258
EDP	0,346	0,270	0,256

(C) Figures of merit across microarchitectures for LOOG-sensitive kernels

CU	16	32	48
ADP			
PDP	9,13%	0	0,70%
EDP	10,78%	0	1,15%
Delay			
PDP	22,45%	3,79%	0
EDP	23,73%	4,88%	0
Energy			
PDP	13,45%	1,58%	0
EDP	13,03%	1,20%	0
EDP			
PDP	34,21%	4,66%	0
EDP	35,10%	5,36%	0

(D) Overheads compared to the minimum for each figure, LOOG-sensitive kernels. 48 is the optimal tradeoff

FIGURE 6.1: Comparing differently sized reconfigurable microarchitectures when optimally reconfiguring with the PDP and EDP reconfiguration metrics

6.2 Software reconfiguration

In this section, the reconfiguration potential a software controller operating with software directives described in Section 5.8.4 is evaluated. As previously mentioned, optimal static

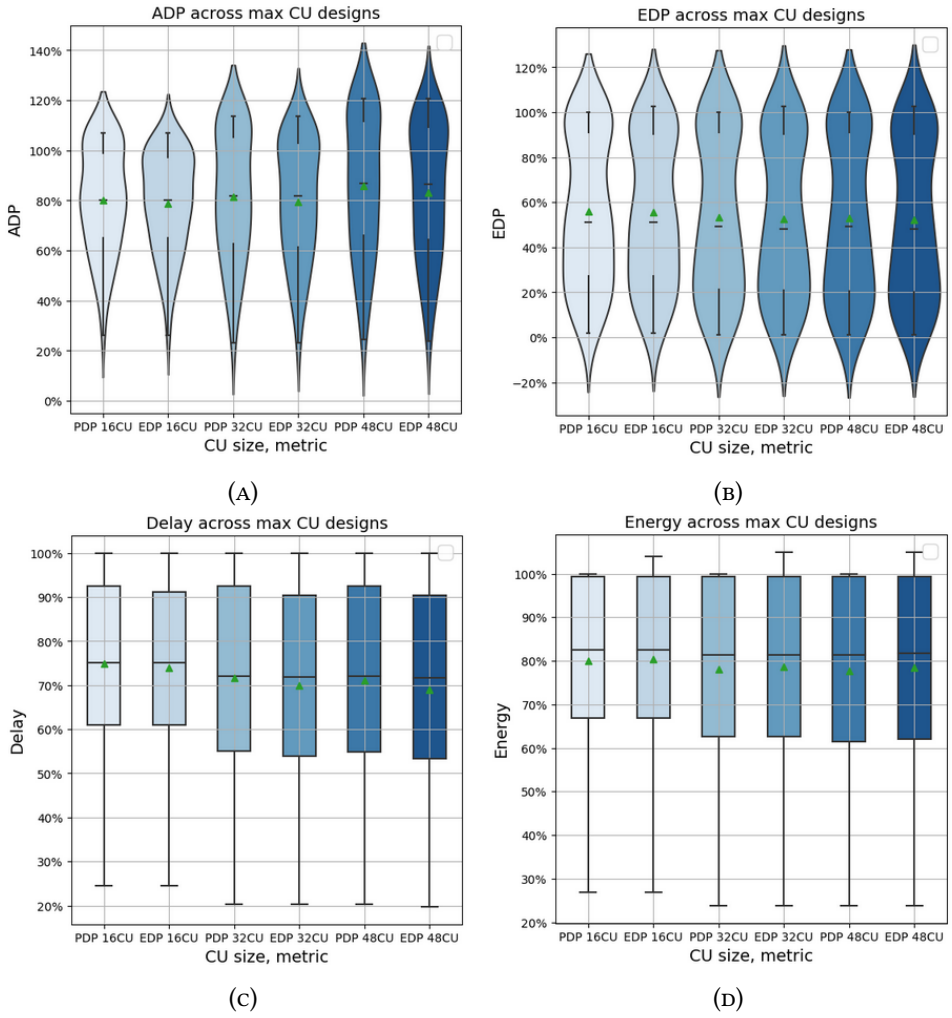


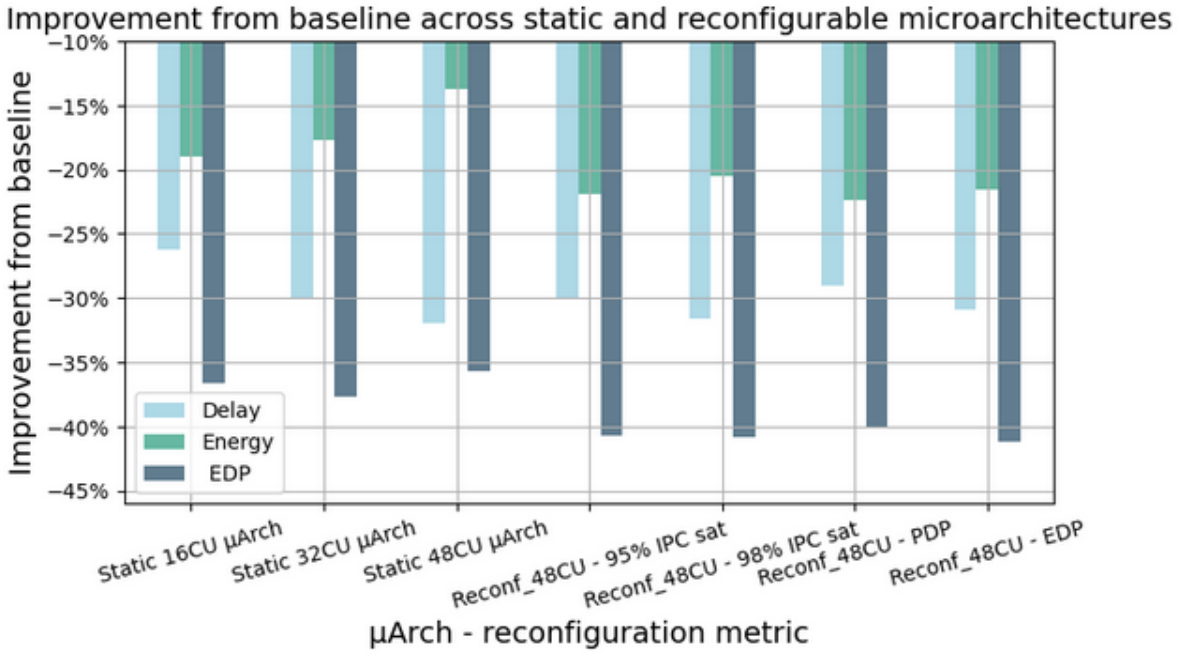
FIGURE 6.2: Figures of merit distributions across differently sized microarchitectures and reconfiguration metrics, for optimal static reconfiguration

reconfiguration (whole-kernel reconfiguration) is performed, assuming perfect knowledge of the configuration that optimizes the cumulative value of the given reconfiguration metric over all launches. Both single launch and multi-launch kernel are taken into account. Static reconfiguration, referring to applying a specific configuration for the whole kernel's execution should not be confused with static microarchitectures, referring to set-in-stone designs, without the potential to reconfigure.

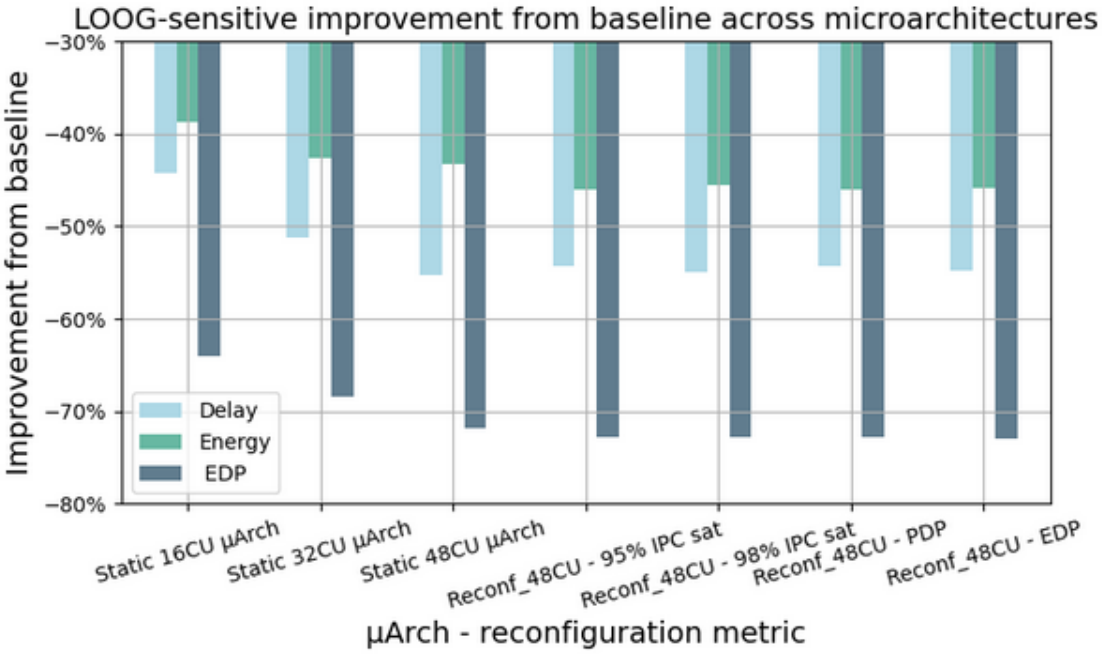
6.2.1 Static reconfiguration

Generic kernels

As seen in Figure 6.3a, regarding the static (set-in-stone) microarchitectures, the energy-efficiency-delay trade-off is apparent for 16 through 32 CUs, with EDP minimizing at 32 CUs. Performance, energy efficiency and EDP are optimized at 48, 16 and 32 CUs respectively. Interestingly, even for generic kernels, reconfiguration by any metric is more energy efficient



(A) Improvement for generic kernels



(B) Improvement for LOOG-sensitive kernels

FIGURE 6.3: Improvement from baseline uArch across set-in-stone uArchs and the reconfigurable 48-CU uArch optimizing different reconfiguration metrics, for generic and LOOG-sensitive kernels

than any static configuration. When comparing to scale-up static uArchs, this is explained by the ability of the reconfigurable uArch to scale-down on demand. When comparing to scale-down static uArchs, it is explained by the speedup gained in scale-up configurations not accesible statically. The same is true for EDP. When reconfiguring for performance (98% IPC saturation), the reconfigurable uArch achieves a -31.56% delay improvement, minimally worse than the -31.98% achieved by the optimal static uArch (48 CUs), while energy overhead differs by 6.6%. When reconfiguring for energy-efficiency (PDP), the reconfigurable uArch achieves a -22.34% change from baseline, compared to -18.93% by the optimal static 16 CU uArch. When reconfiguring for minimizing of EDP, a -41.25% change is provided, compared to the -37.68% optimal by the 32 CU static uArch. In conclusion, the reconfigurable microarchitecture provides significant improvement across all figures of merit when comparing to the optimal static uArch in context.

LOOG-sensitive kernels

Regarding LOOG-sensitive kernels, seen in Figure 6.3b the same qualitative conclusions are true for reconfiguration, nevertheless with lesser relative differences, due to this class of kernels more frequently optimizing the figures of merit by occupying scale-up configurations. Regarding the static uArchs, the tradeoff seen in Figure 6.3a is not present, with all figures of merit being optimized at 48 CUs. This is, likewise explained by the high OOO scalability of the applications. Reconfiguring for performance provides a -55.00% delay improvement, approximating the -55.30% static optimal by the 48 CU uArch, with a 2.23% energy overhead difference. Reconfiguring for energy-efficiency improves dissipation by -46.11%, compared to the -43.29% optimal. EDP is reduced by 72.98% compared to the 71.82% optimal.

6.2.2 Semi-Dynamic reconfiguration

In this subsection, the effect of a semi-dynamic reconfiguration is examined exclusively for multi-launch kernels, across two classes of kernels, above and below the median launch number. As presented in Section 5.8.5, static reconfiguration does not pose significant overheads compared to the finer-grain semi-dynamic reconfiguration seen in Figure 5.28. More specifically, as evident in Table 6.1, minimal average and mean overhead values are seen for both classes of kernels (according to number of invocations), with the distribution being fairly wide. Therefore, fringe cases of low intra-kernel scalability consistency producing significant overheads are seen in Tables 6.3 and 6.2. Especially for the kernels seen in Table 6.3, mismatched configurations represent the majority of launches in some cases. It is worth noting that the various optimal configurations tend to be interleaved, therefore increasing the reconfiguration energy overheads considered insignificant in the scope of this thesis. It is also worth noting that kernels with launches above the median value of 7 tend to not be the most inconsistent regarding scalability, as was initially expected. In conclusion, the static

(whole-kernel) reconfiguration overhead is generally insignificant compared to a per-launch granularity reconfiguration, which motivates the hardware-level implementation of the reconfiguration controller.

Launches	Less than 7 launches		7 or more launches	
	Delay	Energy	Delay	Energy
Min	0	0	0	0
Median	-0.45%	-0.52%	-0.82%	-0.58%
Mean	-0.52%	-0.80%	-1.10%	-0.86%
Max	-2.41%	-2.65%	-4.42%	-4.89%

Table 6.1: Distribution of Delay and Energy improvement from static to semi-dynamic reconfiguration

Kernel	DEnergy	Description	static configuration	inorder	8_CUs	16_CUs	24_CUs	32_CUs	40_CUs	48_CUs
testSssp	-4,89%	Reg. expand kernel	8_CUs	3	3	0	0	0	0	0
ispass-2009-BFS	-3,26%	BFS	16_CUs	1	0	8	0	0	0	0
shoc-Reduction	-2,85%	Reduce operation	32_CUs	0	0	0	0	3	1	0
shoc-Sort	-1,77%	Scan	16_CUs	2	0	2	0	0	0	0
test-Amr	-1,49%	Reg refine kernel	24_Cus	0	0	1	2	1	0	0

Table 6.2: Kernels with the highest energy overhead in static reconfiguration compared to semi-dynamic reconfiguration

Kernel	DDelay	Description	static configuration	inorder	8_CUs	16_CUs	24_CUs	32_CUs	40_CUs	48_CUs
dwt2d-rodinia-3.1	-4,42%	DWT2D kernel	32_CUs	0	0	0	0	3	4	0
lonestar-sssp-wln	-4,27%	RelaxGraphWorklist	16_CUs	0	0	4	2	1	1	0
cfd-rodinia	-3,87%	Initialization	16_CUs	0	2	0	1	0	0	0
shoc-Spmv	-3,69%	spmv_scalar	48_CUs	0	0	0	0	13	4	82
test-Amr	-2,55%	Reg refine kernel	32_CUs	0	0	0	3	1	0	0

Table 6.3: Kernels with the highest delay overhead in static reconfiguration compared to semi-dynamic reconfiguration

6.2.3 Static reconfiguration across clusters of applications

In Figure 6.4, the optimal static reconfiguration scheme is evaluated against the static 48 CU uArch across the clusters of applications defined in Section 5.3 . It can be seen that the performance of the DP-bound, high utilization cluster with applications that frequently stall on the ALU and DP pipelines closely approximates the average kernel execution under the reconfigurable architecture. When reconfiguring with the PDP metric, a -31.0% delay improvement and a -24.5% energy improvement is provided. Frequent control hazards restrict the application OOO scalability as discussed in Section 5.3. As expected, the Cache-bound, low ILP and shared-memory bound clusters suffer the least improvement on the reconfigurable architecture, with energy-efficiency and EDP being even lower than that of the static 48 CU architecture. The SP-bound and Cache-bound, high ILP clusters perform significantly better than average, regarding the reconfigurable architecture. The lowest performance overall is

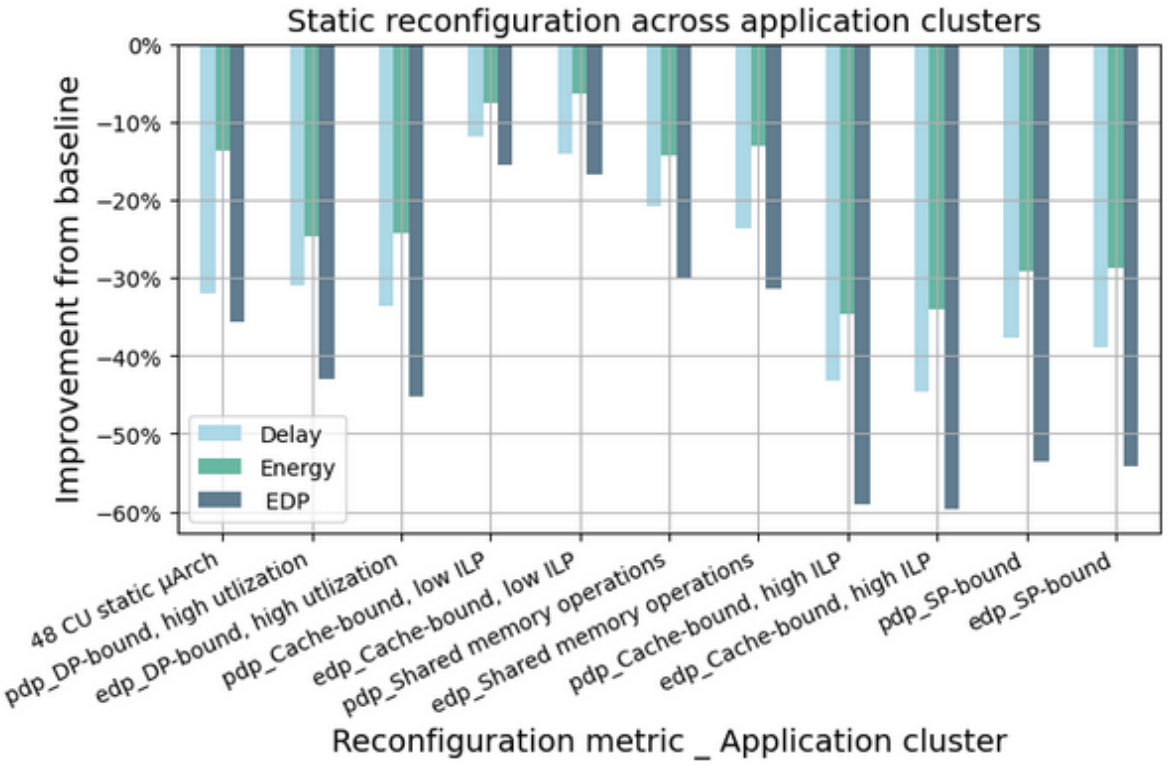


FIGURE 6.4: Evaluating the reconfigurable 48 CU architecture across all clusters of applications defined in Section 5.3, against the static 48 CU architecture for the average application

exhibited for the Cache-bound, low ILP cluster (due to high thread block count and parameter memory instructions, as discussed in Section 5.3) and interestingly, the Cache-bound high-ILP cluster performs better on the reconfigurable architecture across all figures of merit and clusters, even better than the purely compute-intensive kernels. This is due to low miss rate on cache accesses and computation on separate phases that does not cause backend congestion, as discussed in Chapter 5.

6.3 Hardware reconfiguration controller

In this section, the hardware reconfiguration controller is evaluated against optimal software-implemented static reconfiguration as described in Section 5.8.4, as well as a static (set-in-stone) microarchitecture with 48 CUs.

6.3.1 First-launch reconfiguration

In Figure 6.5, the aforementioned comparison is presented. The hardware controller is less efficient than the software static reconfiguration controller across all figures of merit and

reconfiguration metrics. Compared to the static 48 CU uArch,it provides a -30.6% delay improvement when maximizing EDP, as opposed to -32% (-31$ for optimal static reconfiguration). The respective differences in energy efficiency are -19.5%,-13.7% (-22.4% for optimal static reconfiguration) and -39.8%, -35.7% regarding EDP (-41.3%). In conclusion, the hardware reconfiguration controller attains 67% of the energy efficiency improvement and 73% of the EDP improvement of the optimal static reconfiguration compared to baseline. Maximizing for performance brings it within 1% of the optimal. Results for the average kernel are summarized in Table 6.4.

Static 48CU uArch		
Delay	0,680	
Energy	0,863	
EDP	0,643	
Optimal static reconf		
Metric	PDP	EDP
Delay	0,710	0,690
Energy	0,776	0,784
EDP	0,599	0,587
Regressor		
Metric	PDP	EDP
Delay	0,726	0,694
Energy	0,805	0,817
EDP	0,633	0,602

TABLE 6.4: Reconfiguration controller Delay and Energy normalized to baseline, for PDP and EDP reconfiguration metrics

The RMSE for the predicted normalized IPC improvement vector over the test set is shown in Figure 6.6.

6.3.2 Static reconfiguration

In this subsection, the controller is evaluated on static (whole kernel) instead of first-launch reconfiguration. Hardware performance counters are collected over all invocations of the kernel, during one of its executions and used to infer an optimal configuration that is then applied to subsequent executions of the kernel over all its launches. The purpose of this evaluation is to expand the limited multi-launch kernels dataset to include all kernels, and determine controller behavior regarding LOOG-sensitive kernels. In Figures 6.7a and 6.7b the delay provided by regressor reconfiguration normalized to baseline is presented for the average and the average LOOG-sensitive kernel respectively. In Figures 6.7c and 6.7d the respective normalized energy distributions are provided. For both delay and energy, the reconfiguration controller is more accurate for LOOG-sensitive kernels, due to the aforementioned asymptotic behavior of IPC improvement and declining power ratios between

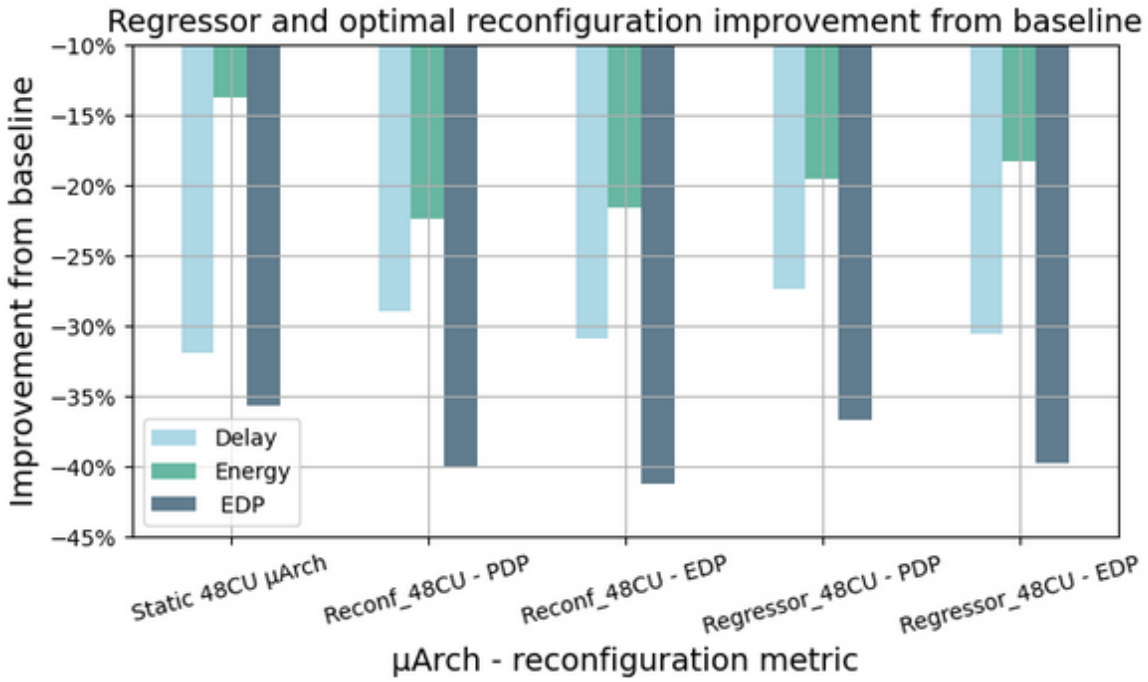


FIGURE 6.5: Evaluating the hardware reconfiguration controller (regressor) against optimal static reconfiguration and a static 48 CU uArch

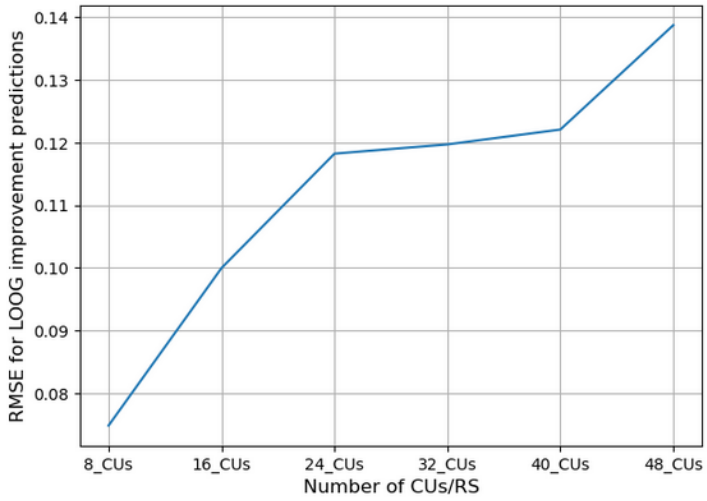
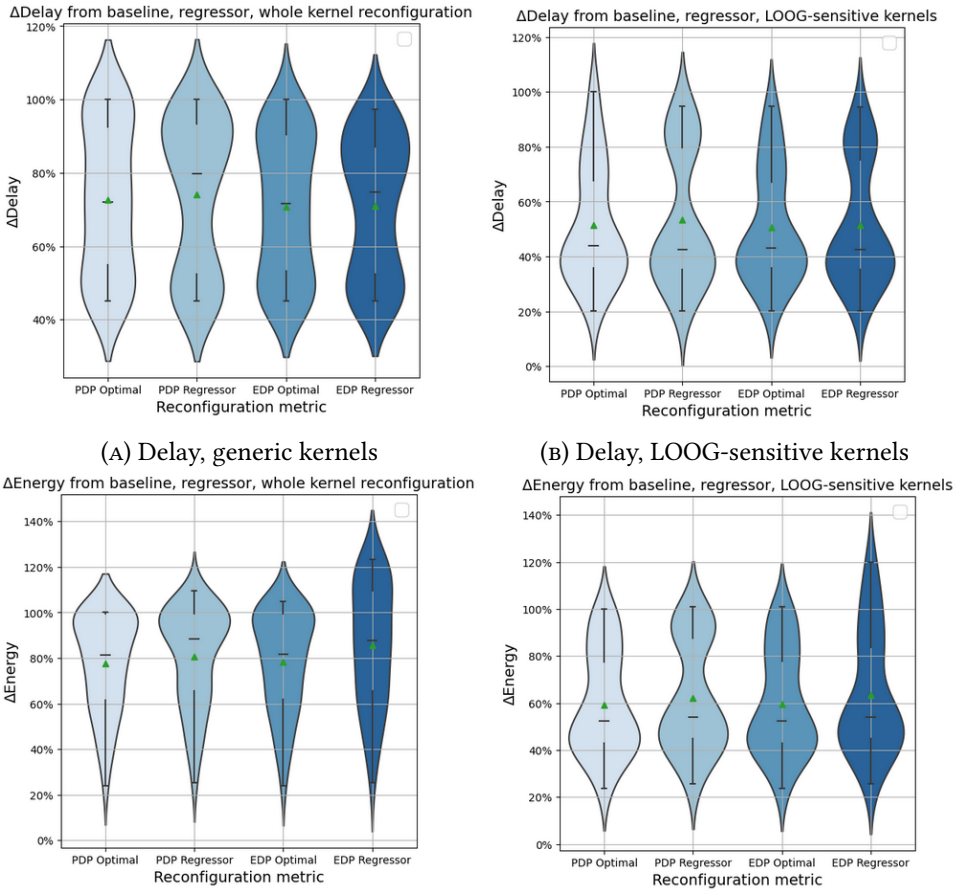


FIGURE 6.6: RMSE for reconfiguration controller predictions and intermediate configuration IPC calculation



(A) Delay, generic kernels

(B) Delay, LOOG-sensitive kernels

(C) Energy, generic kernels

(D) Energy, LOOG-sensitive kernels

FIGURE 6.7: Delay and Energy improvement for optimal static (whole-kernel) reconfiguration with the hardware controller

consecutive configurations on the most scale-up LOOG configurations. In Figure 6.7a, it is evident that while average values between the regressor and optimal reconfiguration are similar, medians vary due to mispredicted outliers. In Figure 6.7b, a tighter distribution is seen for the regressor,failing to predict the baseline inorder configuration. In Figures 6.7c and 6.7d, significant mispredictions are seen when maximizing EDP. This is due to the inherent error in IPC prediction that is exacerbated in the square of the delay, frequently leading to the most scale-down configurations. The same is seen to an extent in Figure 6.7c regarding PDP, due to inorder mispredictions. The fact that mispredictions tend to be scale-down rather than scale-up is clearly depicted in Figure 6.7a, with the bloated delay distribution near the baseline.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

With the implementation of the LOOG execution scheme on Accel-Sim, we gain the ability to increase simulation accuracy on many fronts. We also gain access to HPC-relevant microbenchmarks-tuned datacenter GPU configurations, including the Quadro GV100 used in our analysis.
Having collected microarchitecture runtime statistics across 7 benchmark suites and 100 kernels, we use them to characterize applications into groups regarding architecture bottlenecks, as well as correlate specific features they possess with speedup and high GPU utilization on scale-up LOOG configurations. A select class of kernels that gain significant speedup on such configurations are deemed "LOOG-sensitive". This analysis provides the backbone for further workload characterization throughout this thesis.
Components of the pipeline front-end are right-sized and optimally configured in conjuction with LOOG, leading to the conclusions that the Decoder bandwidth can be throttled (-4.57% Power, -0.22% Area) and that Depth-First instruction Issue scheduling provides optimal performance (speedup of 1.14 for LOOG-senstitive kernels) . The latter, along with the realization that exploitable ILP widely varies inter-warp, leads to the implementation of an Instruction Buffer partitioning reconfiguration controller that provides an average 4.4% performance increase for LOOG-sensitive kernels, up to 10.2%.
LOOG-relevant structures (Collector Units, Register Renaming Stack and Instruction Buffer) are studied regarding scalability, leading to the conclusion that Collector Units are the main component driving speedup but are also Power and Area hungry.
The potential to benefit from the significant speedup gained in scale-up LOOG configurations regarding LOOG-sensitive kernels while preserving energy efficiency when necessary, is provided by a scalable runtime-reconfigurable microarchitecture that power gates groups of Collector Units. The reconfigurable architecture is initially evaluated on the basis of an oracle software controller performing semi-dynamic reconfiguration on a per-launch granularity. Realizing that profiling the first kernel launch is sufficient for reconfiguration on subsequent invocations, we implement a hardware reconfiguration controller using runtime performance counters, based on decision tree regressors.

A static scale-up LOOG configuration provides a speedup of 1.48 for generic kernels and a 13.7% reduction in energy dissipation, compared to the baseline architecture. Reconfiguration under software directives and using the hardware controller can provide the same speedup when needed and have the potential to improve energy efficiency from baseline by 22.4% and 19.5% respectively

7.2 Future work

- Minor Accel-Sim source code refactoring where necessary (Appendix A.1) as well as appropriate setups to exploit the trace-based (mISA) simulation functionality that Accel-Sim provides is our immediate priority to increase performance simulation accuracy.
- Cycle-accurate power modelling with LOOG-relevant runtime performance counters input from the performance model (as Accelwattch operates for the baseline model) is necessary to ascertain quantitative credibility of our results.
- A more accurate assessment of Power Gating sleep transistor Area, Energy and wake-up time overheads is needed to confirm our assumptions. Due to the reconfiguration schemes utilized, Energy and wake-up Delay can be traded off and have a great margin for error but Area is crucial.
- With a high degree of certainty, the hardware reconfiguration controller regressors can be fit on intermediately-scaled LOOG configurations, to avoid the first launch in-order configuration overhead. This has not been performed due to practical time constraints.
- A more meticulous study of inter-warp heterogeneity regarding exploitable ILP can be performed and new Instruction Buffer reconfiguration policies or metrics can be tested. Since Ibuffer reconfiguration was not the main focus of this thesis, we speculate that relevant results are suboptimal.

Appendix A

Source Code

A.1 Source code repository

The source code for the implementation elaborated on in Subsection 2.5 can be found at:
https://github.com/pelef/accel-sim-framework-LOOG/tree/loog
https://github.com/pelef/accel-sim-framework-LOOG/tree/inorder

A.2 Original License

As the code was developed based on Accel-Sim, it is distributed with the relevant license.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Bibliography

[1] Tor M Aamodt, Wilson Wai Lun Fung, and Timothy G Rogers. *General-purpose graphics processor architectures*. Synthesis lectures on computer architecture 44. Morgan Claypool Publishers, 2018.

[2] TechTarget Contributor. *What is GPGPU (general Purpose Graphics Processing Unit)?: Definition from TechTarget*. 2015. URL: https://www.techtarget.com/whatis/definition/GPGPU-general-purpose-graphics-processing-unit.

[3] *Technology partner*. 2022. URL: https://www.nextdimensioninc.com/wp-content/uploads/2018/07/AI-Considerations-For-Scaling-GPU-Ready-Data-Centers-Whitepaper.pdf.

[4] Christophe Bobda. *Introduction to Reconfigurable Computing: Architectures, Algorithms, and Applications*. 1st. Springer Publishing Company, Incorporated, 2007. ISBN: 1402060882.

[5] Artur Podobas, Kentaro Sano, and Satoshi Matsuoka. "A Survey on Coarse-Grained Reconfigurable Architectures From a Performance Perspective". In: *IEEE Access* 8 (2020), pp. 146719–146743. DOI: 10.1109/access.2020.3012084. URL: https://doi.org/10.1109/access.2020.3012084.

[6] Frank Vahid and Tony D Givargis. *Embedded System Design*. en. Nashville, TN: John Wiley & Sons, Oct. 2001.

[7] URL: https://hps.ece.utexas.edu/pub/TR-HPS-2007-001.pdf.

[8] Khubaib et al. "MorphCore: An Energy-Efficient Microarchitecture for High Performance ILP and High Throughput TLP". In: *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, Dec. 2012. DOI: 10.1109/micro.2012.36. URL: https://doi.org/10.1109/micro.2012.36.

[9] Hyoun Kyu Cho and Scott Mahlke. "Embracing heterogeneity with dynamic core boosting". In: *Proceedings of the 11th ACM Conference on Computing Frontiers*. ACM, May 2014. DOI: 10.1145/2597917.2597932. URL: https://doi.org/10.1145/2597917.2597932.

[10] Mohammad Khavari Tavana et al. "ElasticCore". In: *Proceedings of the 52nd Annual Design Automation Conference*. ACM, June 2015. DOI: 10.1145/2744769.2744833. URL: https://doi.org/10.1145/2744769.2744833.

[11] Paula Petrica et al. "Flicker". In: *Proceedings of the 40th Annual International Symposium on Computer Architecture*. ACM, June 2013. DOI: 10.1145/2485922.2485924. URL: https://doi.org/10.1145/2485922.2485924.

[12] Konstantinos Iliakis, Sotirios Xydis, and Dimitrios Soudris. "LOOG: Improving GPU Efficiency With Light-Weight Out-Of-Order Execution". In: *IEEE Computer Architecture Letters* 18.2 (July 2019), pp. 166–169. DOI: 10.1109/lca.2019.2951161. URL: https://doi.org/10.1109/lca.2019.2951161.

[13] Konstantinos Iliakis, Sotirios Xydis, and Dimitrios Soudris. "Repurposing GPU Microarchitectures with Light-Weight Out-Of-Order Execution". In: *IEEE Transactions on Parallel and Distributed Systems* 33.2 (Feb. 2022), pp. 388–402. DOI: 10.1109/tpds.2021.3093231. URL: https://doi.org/10.1109/tpds.2021.3093231.

[14] Konstantinos Iliakis. "Large-Scale Software Optimization And Micro-Architectural Specialization for Accelerated High-Performance Computing". PhD thesis. National Technical University of Athens, 2022. URL: https://dspace.lib.ntua.gr/xmlui/handle/123456789/55823.

[15] R. M. Tomasulo. "An Efficient Algorithm for Exploiting Multiple Arithmetic Units". In: *IBM Journal of Research and Development* 11.1 (Jan. 1967), pp. 25–33. DOI: 10.1147/rd.111.0025. URL: https://doi.org/10.1147/rd.111.0025.

[16] *Cuda Zone - Library of resources*. 2022. URL: https://developer.nvidia.com/cuda-zone.

[17] Lena Oden. "Lessons learned from comparing C-CUDA and Python-Numba for GPU-Computing". In: *2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. IEEE, Mar. 2020. DOI: 10.1109/pdp50117.2020.00041. URL: https://doi.org/10.1109/pdp50117.2020.00041.

[18] Peter Zunitch. *Cuda vs. opencl vs. OpenGL*. 2022. URL: https://www.videomaker.com/article/c15/19313-cuda-vs-opencl-vs-opengl/.

[19] *CUDA toolkit documentation 12.1*. 2023. URL: https://docs.nvidia.com/cuda/index.html.

[20] URL: https://web.stanford.edu/class/ee380/Abstracts/080227-Nickolls-CUDAScalableParallelProgramming.pdf.

[21] *1. introduction*. 2023. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[22] *Cuda*. URL: https://nyu-cds.github.io/python-gpu/02-cuda/.

[23] Ali Bakhoda et al. "Analyzing CUDA workloads using a detailed GPU simulator". In: *2009 IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE, Apr. 2009. DOI: 10.1109/ispass.2009.4919648. URL: https://doi.org/10.1109/ispass.2009.4919648.

[24] *Main page*. URL: http://gpgpu-sim.org/manual/index.php/Main_Page.

[25] Keith Barr. *ASIC design in the silicon sandbox: A complete guide to building mixed-signal integrated circuits*. New York, NY: McGraw-Hill Professional, Dec. 2006.

[26] Anthony M. Cabrera and Roger D. Chamberlain. "Designing Domain Specific Computing Systems". In: *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, May 2020. DOI: 10.1109/fccm48280.2020.00052. URL: https://doi.org/10.1109/fccm48280.2020.00052.

[27] URL: https://passlab.github.io/CSCE513/notes/lecture26_DSA_DomainSpecificArchitectures.pdf.

[28] Manuel Padial PérezLead the Technical Content Maintenance Team of doEEEt platform. at Alter TechnologyManuel Padial has a Degree in Industrial Electronic. Engineering, a Master (M.Eng.)/Advanced Degree in Electrical, and Electronic Engineering.Since 200. *ASIC or FPGA, how to choose between them*. 2023. URL: https://www.doeeet.com/content/eee-components/actives/choosing-between-asic-or-fpga/.

[29] *What is an FPGA? field programmable gate array*. URL: https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html.

[30] Dimitrios Soudris and Stamatis Vassiliadis, eds. *Fine- and Coarse-Grain Reconfigurable Computing*. en. 2007th ed. New York, NY: Springer, Oct. 2007.

[31] T.J. Todman et al. "Reconfigurable computing: architectures and design methods". In: *IEE Proceedings - Computers and Digital Techniques* 152.2 (2005), p. 193. DOI: 10.1049/ip-cdt:20045086. URL: https://doi.org/10.1049/ip-cdt:20045086.

[32] Moghaddam Shahraki Mansureh, Jae-Min Cho, and Kiyoung Choi. "Reconfigurable Architectures". In: *Handbook of Hardware/Software Codesign*. Springer Netherlands, 2016, pp. 1–42. DOI: 10.1007/978-94-017-7358-4_12-1. URL: https://doi.org/10.1007/978-94-017-7358-4_12-1.

[33] Jason G. Tong, Ian D. L. Anderson, and Mohammed A. S. Khalid. "Soft-Core Processors for Embedded Systems". In: *2006 International Conference on Microelectronics*. IEEE, Dec. 2006. DOI: 10.1109/icm.2006.373294. URL: https://doi.org/10.1109/icm.2006.373294.

[34] URL: https://web.archive.org/web/20091026171102/http://1-core.com/library/digital/soft-cpu-cores/.

[35] Hugo Andrade and Ivica Crnkovic. "A Review on Software Architectures for Heterogeneous Platforms". In: *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, Dec. 2018. DOI: 10.1109/apsec.2018.00035. URL: https://doi.org/10.1109/apsec.2018.00035.

[36] Mahmoud Khairy et al. "Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling". In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, May 2020. DOI: 10.1109/isca45697.2020.00047. URL: https://doi.org/10.1109/isca45697.2020.00047.

[37] Engin Ipek et al. "Core fusion". In: *Proceedings of the 34th annual international symposium on Computer architecture*. ACM, June 2007. DOI: 10.1145/1250662.1250686. URL: https://doi.org/10.1145/1250662.1250686.

[38] Changkyu Kim et al. "Composable Lightweight Processors". In: *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*. IEEE, 2007. DOI: 10.1109/micro.2007.41. URL: https://doi.org/10.1109/micro.2007.41.

[39] Shubham Kamdar and Neha Kamdar. "big. LITTLE Architecture: Heterogeneous Multicore Processing". In: *International Journal of Computer Applications* 119.1 (June 2015), pp. 35–38. DOI: 10.5120/21034-3106. URL: https://doi.org/10.5120/21034-3106.

[40] Chris Fallin, Chris Wilkerson, and Onur Mutlu. "The heterogeneous block architecture". In: *2014 IEEE 32nd International Conference on Computer Design (ICCD)*. IEEE, Oct. 2014. DOI: 10.1109/iccd.2014.6974710. URL: https://doi.org/10.1109/iccd.2014.6974710.

[41] D. Burger et al. "Scaling to the end of silicon with EDGE architectures". In: *Computer* 37.7 (July 2004), pp. 44–55. DOI: 10.1109/mc.2004.65. URL: https://doi.org/10.1109/mc.2004.65.

[42] Karthikeyan Sankaralingam et al. "Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture". In: *Proceedings of the 30th annual international symposium on Computer architecture - ISCA '03*. ACM Press, 2003. DOI: 10.1145/859618.859667. URL: https://doi.org/10.1145/859618.859667.

[43] Mihai Pricopi and Tulika Mitra. "Bahurupi". In: *ACM Transactions on Architecture and Code Optimization* 8.4 (Jan. 2012), pp. 1–21. DOI: 10.1145/2086696.2086701. URL: https://doi.org/10.1145/2086696.2086701.

[44] Ankit Sethia and Scott Mahlke. "Equalizer: Dynamic Tuning of GPU Resources for Efficient Execution". In: *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, Dec. 2014. DOI: 10.1109/micro.2014.16. URL: https://doi.org/10.1109/micro.2014.16.

[45] Xianwei Cheng et al. *AMOEBA: A Coarse Grained Reconfigurable Architecture for Dynamic GPU Scaling*. 2019. DOI: 10.48550/ARXIV.1911.03364. URL: https://arxiv.org/abs/1911.03364.

[46] URL: https://www.benchcouncil.org/WPC/.

[47] Lei Wang et al. "WPC: Whole-Picture Workload Characterization Across Intermediate Representation, ISA, and Microarchitecture". In: *IEEE Computer Architecture Letters* 20.2 (July 2021), pp. 86–89. DOI: 10.1109/lca.2021.3087828. URL: https://doi.org/10.1109/lca.2021.3087828.

[48] URL: https://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_1080_Whitepaper_FINAL.pdf.

[49] Nvidia Corporation. *Volta Architecture Whitepaper*. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf. Accessed: March 25, 2023. 2017.

[50] Hadi Esmaeilzadeh et al. "Dark silicon and the end of multicore scaling." In: *ISCA*. Ed. by Ravi Iyer, Qing Yang, and Antonio González. ACM, 2011, pp. 365–376. ISBN: 978-1-4503-0472-6. URL: http://dblp.uni-trier.de/db/conf/isca/isca2011.html#EsmaeilzadehBASB11.

[51] Adrian McMenamin. *The end of Dennard scaling*. Blog post. 2013. URL: https://cartesianproduct.wordpress.com/2013/04/15/the-end-of-dennard-scaling/.

[52] R. Saleh et al. "System-on-Chip: Reuse and Integration". In: *Proceedings of the IEEE* 94.6 (June 2006), pp. 1050–1069. DOI: 10.1109/jproc.2006.873611. URL: https://doi.org/10.1109/jproc.2006.873611.

[53] *What is HPC? introduction to high-performance computing*. URL: https://www.ibm.com/topics/hpc.

[54] Rahul Awati. *What is high-performance computing (HPC)?* 2021. URL: https://www.techtarget.com/searchdatacenter/definition/high-performance-computing-HPC.

[55] *High-performance computing*. 2023. URL: https://en.wikipedia.org/wiki/High-performance_computing.

[56] *General-purpose computing on graphics processing units*. 2023. URL: https://en.wikipedia.org/wiki/General-purpose_computing_on_graphics_processing_units.

[57] Vijay Kandiah et al. "AccelWattch: A Power Modeling Framework for Modern GPUs". In: *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, Oct. 2021. DOI: 10.1145/3466752.3480063. URL: https://doi.org/10.1145/3466752.3480063.

[58] *What is Parallel Computing - javatpoint*. URL: https://www.javatpoint.com/what-is-parallel-computing.

[59] Lillian Cassel et al. "Concurrency and parallelism in the computing ontology". In: *Proceedings of the 14th annual ACM SIGCSE conference on Innovation and technology in computer science education*. ACM, July 2009. DOI: 10.1145/1562877.1563044. URL: https://doi.org/10.1145/1562877.1563044.

[60] V.P. Kumar and A. Gupta. "Analyzing Scalability of Parallel Algorithms and Architectures". In: *Journal of Parallel and Distributed Computing* 22.3 (Sept. 1994), pp. 379–391. DOI: 10.1006/jpdc.1994.1099. URL: https://doi.org/10.1006/jpdc.1994.1099.

[61] James Dinan et al. "Scalable work stealing". In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM, Nov. 2009. DOI: 10.1145/1654059.1654113. URL: https://doi.org/10.1145/1654059.1654113.

[62] T.E. Anderson. "The performance of spin lock alternatives for shared-money multiprocessors". In: *IEEE Transactions on Parallel and Distributed Systems* 1.1 (1990), pp. 6–16. DOI: 10.1109/71.80120. URL: https://doi.org/10.1109/71.80120.

[63] URL: https://www.mcs.anl.gov/~itf/dbpp/text/node17.html.

[64] *Single instruction single data*. URL: https://www.sciencedirect.com/topics/computer-science/single-instruction-single-data.

[65] M.J. Flynn. "Very high-speed computing systems". In: *Proceedings of the IEEE* 54.12 (1966), pp. 1901–1909. DOI: 10.1109/proc.1966.5273. URL: https://doi.org/10.1109/proc.1966.5273.

[66] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. 5th ed. Amsterdam: Morgan Kaufmann, 2012. ISBN: 978-0-12-383872-8.

[67] URL: https://www.ece.ucdavis.edu/~jowens/171/lectures/dlp3.pdf.

[68] *Types of parallelism in processing execution*. URL: https://www.tutorialspoint.com/types-of-parallelism-in-processing-execution.

[69] URL: https://www.inf.ed.ac.uk/teaching/courses/pa/Notes/lecture02-types.pdf.

[70] NVIDIA. *Fermi, the NVIDIA CUDA Architecture*. 2021. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#fermi-the-nvidia-cuda-architecture.

[71] NVIDIA Corporation. *Programming Guide for CUDA C++*. 2019. URL: https://docs.nvidia.com/cuda/.

[72] Matt Pharr and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann, 2016.

[73] *CUDA Toolkit Documentation*. NVIDIA Corporation. 2021. URL: https://docs.nvidia.com/cuda/.

[74] URL: https://images.nvidia.com/aem-dam/en-zz/Solutions/geforce/ampere/pdf/NVIDIA-ampere-GA102-GPU-Architecture-Whitepaper-V1.pdf.

[75] *Kernel Profiling Guide*. URL: https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html.

[76] Mike O'Connor et al. "Fine-grained DRAM". In: *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, Oct. 2017. DOI: 10.1145/3123939.3124545. URL: https://doi.org/10.1145/3123939.3124545.

[77] URL: https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.

[78] Dr Ranjani Parthasarathi. *Computer architecture*. 2018. URL: https://www.cs.umd.edu/~meesh/411/CA-online/chapter/exploiting-data-level-parallelism/index.html.

[79] *Cuda Refresher: The Cuda Programming Model*. 2023. URL: https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/.

[80] Mark Gebhart et al. "Energy-efficient mechanisms for managing thread context in throughput processors". In: *Proceedings of the 38th annual international symposium on Computer architecture*. ACM, June 2011. DOI: 10.1145/2000064.2000093. URL: https://doi.org/10.1145/2000064.2000093.

[81] M. Abdel-Majeed and M. Annavaram. "Warped register file: A power efficient register file for GPGPUs". In: *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, Feb. 2013. DOI: 10.1109/hpca.2013.6522337. URL: https://doi.org/10.1109/hpca.2013.6522337.

[82] *US20110161616A1 - On Demand Register allocation and deallocation for a multithreaded processor*. URL: https://patents.google.com/patent/US20110161616A1/en.

[83] John Kloosterman et al. "Regless". In: *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, Oct. 2017. DOI: 10.1145/3123939.3123974. URL: https://doi.org/10.1145/3123939.3123974.

[84] URL: https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/achievedoccupancy.htm.

[85] Stephen M. Trimberger. "Three Ages of FPGAs: A Retrospective on the First Thirty Years of FPGA Technology". In: *Proceedings of the IEEE* 103.3 (Mar. 2015), pp. 318–331. DOI: 10.1109/jproc.2015.2392104. URL: https://doi.org/10.1109/jproc.2015.2392104.

[86] Lakshmi N. Chakrapani et al. "Trimaran: An Infrastructure for Research in Instruction-Level Parallelism". In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2005, pp. 32–41. DOI: 10.1007/11532378_4. URL: https://doi.org/10.1007/11532378_4.

[87] URL: https://www.etp4hpc.eu/pujades/files/ETP4HPC_WP_Heterogeneous-HPC_20220216.pdf.

[88] Gregory Gailliard et al. "Transaction Level Modelling of SCA Compliant Software Defined Radio Waveforms and Platforms PIM/PSM". In: *2007 Design, Automation &amp Test in Europe Conference &amp Exhibition*. IEEE, Apr. 2007. DOI: 10.1109/date.2007.364418. URL: https://doi.org/10.1109/date.2007.364418.

[89] T.Y. Morad et al. "Performance, Power Efficiency and Scalability of Asymmetric Cluster Chip Multiprocessors". In: *IEEE Computer Architecture Letters* 5.1 (Jan. 2006), pp. 4–4. DOI: 10.1109/l-ca.2006.6. URL: https://doi.org/10.1109/l-ca.2006.6.

[90] R. Kumar et al. "Processor Power Reduction Via Single-ISA Heterogeneous Multi-Core Architectures". In: *IEEE Computer Architecture Letters* 2.1 (Jan. 2003), pp. 2–2. DOI: 10.1109/l-ca.2003.6. URL: https://doi.org/10.1109/l-ca.2003.6.

[91] S. Balakrishnan et al. "The Impact of Performance Asymmetry in Emerging Multicore Architectures". In: *32nd International Symposium on Computer Architecture (ISCA'05)*. IEEE. DOI: 10.1109/isca.2005.51. URL: https://doi.org/10.1109/isca.2005.51.

[92] *Asymmetric multiprocessing on heterogeneous multiprocessor systems*. 2022. URL: https://www.variscite.com/blog/asymmetric-multiprocessing-on-heterogeneous-multiprocessor-systems/.

[93] URL: https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/productspage/quadro/quadro-desktop/quadro-volta-gv100-data-sheet-us-nvidia-704619-r3-web.pdf.

[94] *NVIDIA Quadro: Professional Graphics Cards*. https://www.nvidia.com/en-us/design-visualization/quadro/. Accessed: March 25, 2023.

[95] *What's the difference between GeForce and Quadro Graphics Cards?* URL: https://www.engineering.com/story/whats-the-difference-between-geforce-and-quadro-graphics-cards.

[96] M. Graham Lopez et al. "Examining recent many-core architectures and programming models using SHOC". In: *Proceedings of the 6th International Workshop on Performance Modeling, Benchmarking, and Simulation of High Performance Computing Systems*. ACM, Nov. 2015. DOI: 10.1145/2832087.2832090. URL: https://doi.org/10.1145/2832087.2832090.

[97] Milind Kulkarni et al. "Lonestar: A suite of parallel irregular programs". In: *2009 IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE, Apr. 2009. DOI: 10.1109/ispass.2009.4919639. URL: https://doi.org/10.1109/ispass.2009.4919639.

[98] Shuai Che et al. "Rodinia: A benchmark suite for heterogeneous computing". In: *2009 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, Oct. 2009. DOI: 10.1109/iiswc.2009.5306797. URL: https://doi.org/10.1109/iiswc.2009.5306797.

[99] John Stratton et al. "Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing". In: (Mar. 2023).

[100] *Intel® Quartus® prime software features partial reconfiguration: ...* URL: https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/partial-reconfiguration.html.

[101] URL: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/prtutorial_1.pdf.

[102] Gabriel Fornari and Valdivino Alexandre de Santiago Júnior. "Dynamically Reconfigurable Systems: A Systematic Literature Review". In: *Journal of Intelligent &amp Robotic Systems* 95.3-4 (Aug. 2018), pp. 829–849. DOI: 10.1007/s10846-018-0921-6. URL: https://doi.org/10.1007/s10846-018-0921-6.

[103] Jayshree Ghorpade et al. *GPGPU processing in Cuda Architecture*. 2012. URL: https://arxiv.org/abs/1202.4347.

[104] *Cuda FAQ*. 2021. URL: https://developer.nvidia.com/cuda-faq.

[105] *Nvidia Technologies amp; Architectures*. URL: https://www.nvidia.com/en-us/technologies/.

[106] URL: https://images.nvidia.com/aem-dam/Solutions/geforce/ada/nvidia-ada-gpu-architecture.pdf?ref=jan-eric-schafrich.

[107] Zia Abbas and Mauro Olivieri. "Impact of technology scaling on leakage power in nano-scale bulk CMOS digital standard cells". In: *Microelectronics Journal* 45.2 (Feb. 2014), pp. 179–195. DOI: 10.1016/j.mejo.2013.10.013. URL: https://doi.org/10.1016/j.mejo.2013.10.013.

[108] D. Lustig and M. Martonosi. "Reducing GPU offload latency via fine-grained CPU-GPU synchronization". In: *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, Feb. 2013. DOI: 10.1109/hpca.2013.6522332. URL: https://doi.org/10.1109/hpca.2013.6522332.