ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

FPGA Design and Analysis of a RISC-V Out-Of-Order GPU

Συγγραφέας:
Μαρία Ζέρβα

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

ΕΡΓΑΣΤΗΡΙΟ ΜΙΚΡΟΥΠΟΛΟΓΙΣΤΩΝ ΚΑΙ ΨΗΦΙΑΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Αθήνα, 17η Μαρτίου 2025

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ΕΡΓΑΣΤΗΡΙΟ ΜΙΚΡΟΥΠΟΛΟΓΙΣΤΩΝ ΚΑΙ ΨΗΦΙΑΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

FPGA Design and Analysis of a RISC-V Out-Of-Order GPU

Συγγραφέας:
Μαρία Ζέρβα

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

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 17η Μαρτίου 2025.

(Υπογραφή)	*(Υπογραφή)*	*(Υπογραφή)*
Σωτήριος Ξύδης Επίκουρος Καθηγητής Ε.Μ.Π.	Δημήτριος Σούντρης Καθηγητής Ε.Μ.Π.	Κιαμάλ Πεκμεστζή Ομότιμος Καθηγητής Ε.Μ.Π.

Αθήνα, 17η Μαρτίου 2025

Declaration of Authorship

Υπογραφή:

Ημερομηνία:

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

Περίληψη

Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Διπλωματική Εργασία

FPGA Design and Analysis of a RISC-V Out-Of-Order GPU

της Μαρίας Ζέρβα

Χάρη στις εξαιρετικές υπολογιστικές επιδόσεις και την αποδοτικότητά τους, οι Μονάδες Επεξεργασίας Γραφικών (GPUs) έχουν εδραιώσει τη θέση τους ως η κορυφαία πλατφόρμα για την επιτάχυνση εφαρμογών γενικού σκοπού. Παρ΄ όλ΄ αυτά, ένα υποσύνολο των τελευταίων τείνει να παρουσιάζει μέτριες επιδόσεις. Το προηγουμένως προταθέν σχήμα εκτέλεσης Light-weight Out-Of-Order GPU (LOOG) αντιμετωπίζει αυτό το ζήτημα ενισχύοντας τον ήδη υπάρχοντα Παραλληλισμό σε Επίπεδο Νήματος με την εκμετάλλευση του εγγενούς Παραλληλισμού σε Επίπεδο Εντολών. Παρόλο που το LOOG έχει μοντελοποιηθεί με τη χρήση εργαλείων προσομοίωσης GPU σε προηγούμενες μελέτες, οι υλοποιήσεις αυτές υπέφεραν, πέρα από την αργή εκτέλεση των εφαρμογών, από περιορισμένη ακρίβεια στον υπολογισμό της καταναλισκόμενης Ενέργειας και στην εκτίμηση του κρίσιμου μονοπατιού της σχεδίασης.

Η παρούσα εργασία προτείνει την ενσωμάτωση του LOOG σε RTL σχεδίαση μιας GPU και συγκεκριμένα στην έκδοση 2.0 της Vortex GPU , μια σχεδίαση ανοικτού κώδικα κατάλληλη για ανάπτυξη σε πλατφόρμες FPGA . Για να διατηρηθεί το κέρδος απόδοσης της LOOG στη βασισμένη σε RISC-V σωλήνωση της Vortex , η επέκταση σχεδιάζεται σχολαστικά ώστε να συμπληρώνει την υπάρχουσα μικροαρχιτεκτονική. Επιπλέον, διεξάγεται μια ολοκληρωμένη διερεύνηση των βελτιστοποιήσεων σχεδίασης για την ενίσχυση των επιδόσεων με ταυτόχρονο περιορισμό της συνολικής επιβάρυνσης σε Επιφάνεια και Ισχύ.

Πριν από την πειραματική αξιολόγηση εκτελείται λεπτομερής χαρακτηρισμός 21 εφαρμογών που προσφέρει η Vortex με βάση τη συμπεριφορά τους ως προς τις καθυστερήσεις που προκύπτουν κατά την εκτέλεση, επιτρέποντας τη σωστή διασταυρούμενη αξιολόγηση της μικροαρχιτεκτονικής σε έναν ευρύ χώρο σχεδίασης που υποστηρίζεται από τη δυνατότητα διαμόρφωσης της Vortex . Τα αποτελέσματα καταδεικνύουν ένα μέσο κέρδος επιτάχυνσης έως και περίπου 23,5%, διατηρώντας παράλληλα χαμηλότερα γινόμενα Επιφάνειας-Καθυστέρησης και Ισχύος-Καθυστέρησης σε σύγκριση με την αρχική αρχιτεκτονική της Vortex για διάφορους συνδυασμούς παραμέτρων.

Λέξεις Κλειδιά: Υψηλής Επίδοσης Υπολογισμός, GPU Μικροαρχιτεκρονική, Εκτέλεση Εκτός Σειράς, RISC-V, Σχεδίαση RTL, Πλατφόρμα FPGA, Εξομοίωση Υλικού

NATIONAL TECHNICAL UNIVERSITY OF ATHENS

Abstract

Division of Computer Science
School of Electrical And Computer Engineering

Master of Engineering

FPGA Design and Analysis of a RISC-V Out-Of-Order GPU

by Maria Zerva

Owing to their exceptional computational performance and cost efficiency, GPUs have solidified their status as the premier platform for accelerating general-purpose workloads. Nonetheless, a subset of these workloads continues to exhibit performance stagnation. The previously proposed Light-weight Out-Of-Order GPU (LOOG) execution scheme addresses this issue by augmenting conventional Thread-Level Parallelism with the exploitation of inherent Instruction-Level Parallelism. Although LOOG has been modeled using GPU simulation tools in previous studies, these implementations have suffered from limited accuracy in power consumption and critical path estimations, in addition to slow execution of applications.

To overcome these limitations, this thesis proposes integrating LOOG into an RTL GPU framework and specifically Vortex GPU version 2.0, an open-source design that is well-suited for deployment on FPGA platforms. To preserve LOOG's performance gain in Vortex's RISC-V–based pipeline, the extension is meticulously designed to complement the existing micro-architecture and the operations it supports. Furthermore, a comprehensive investigation of design optimizations and trade-offs is conducted to enhance performance while constraining the overall Area and Power overhead.

A detailed characterization of 21 Vortex workloads based on their stalling behavior is executed previous to the experimental evaluation, enabling the right-sizing of the micro-architecture across a broad design space that is supported by Vortex's configurability. The results demonstrate an average speedup of up to approximately 23.5%, while maintaining lower Area-Delay and Power-Delay products compared to the in-order Vortex in various configurations.

Keywords: High Performance Computing, GPU Micro-Architecture, Out-Of-Order Execution, RISC-V, RTL Design, FPGA, Hardware Evaluation

Acknowledgements

This thesis marks the culmination of my undergraduate studies at the School of Electrical and Computer Engineering of the National Technical University of Athens. It has been a journey of growth, challenges, and invaluable learning experiences, for which I am deeply grateful.

First and foremost, I would like to express my sincere gratitude to my supervisor, Assistant Professor Sotirios Xydis, for entrusting me with the opportunity to work under his guidance. His insight and support throughout the course of this thesis have been instrumental in shaping both the outcome of this work and my personal development as a researcher.

I am also thankful to Dr. Konstantinos Iliakis for his focused advice and technical insight, which helped steer this project in the right direction. Special thanks go to Ph.D. candidate Panagiotis-Eleftherios Eleftherakis, whose consistent technical guidance and assistance were truly invaluable throughout this process.

I would also like to extend my appreciation to Professor Dimitrios Soudris and the members of MicroLab NTUA, whose presence and camaraderie created a positive and inspiring environment during this time in the laboratory.

Last but certainly not least, I am profoundly grateful to my family and friends for their unwavering support, patience, and encouragement over the years. Their belief in me has been a constant source of strength, and this accomplishment would not have been possible without them.

Contents

List of Figures

List of Tables

Κεφάλαιο 1

Εκτεταμένη Ελληνική Περίληψη

1.1 Εισαγωγή

Ο Gordon E. Moore το 1965 προέβλεψε ότι η πυκνότητα των τρανζίστορ στα ολοκληρωμένα κυκλώματα θα διπλασιαζόταν κάθε δύο χρόνια [1]. Αυτή η τάση, που τροφοδοτήθηκε από την κλιμάκωση του Dennard , παρέμεινε αληθής και διαμόρφωσε την τεχνολογική ανάπτυξη μέχρι πριν από περίπου είκοσι χρόνια [2]. Τότε, όμως, καθώς η πυκνότητα της καταναλισκόμενης Ισχύος αυξανόταν λόγω των περιορισμών της τάσης τροφοδοσίας και της πολυπλοκότητας της κατασκευής σε ατομικό επίπεδο, εμφανίστηκε η εποχή του «Σκοτεινού Πυριτίου» [3]. Η τελευταία ονομάστηκε έτσι καθώς απαιτεί πολλά τρανζίστορ μίας συσκευής να παραμένουν ανενεργά ανά διαστήματα ή να λειτουργούν σε μειωμένες συχνότητες.

Σε τομείς όπως η Τεχνητή Νοημοσύνη και η Ανάλυση Δεδομένων, οι GPUs έγιναν οι επικρατέστεροι επιταχυντές υλικού, συμπληρώνοντας τις Κεντρικές Επεξεργαστικές Μονάδες για την αποτελεσματική κάλυψη των υπολογιστικών απαιτήσεων. Στις μέρες μας, προκειμένου να ξεπερνούν τους αναφερθέντους περιορισμούς, οι αρχιτεκτονικές πρέπει να ενσωματώνουν τόσο την ποικιλομορφία λογισμικού όσο και την ετερογένεια στο υλικό [4].

1.1.1 Γενικού σκοπού GPUs

Οι Κεντρικές Μονάδες Επεξεργασίας (CPUs) έχουν εξελιχθεί ανά τα χρόνια ώστε να αποτελούνται από μερικούς (δεκάδες) πυρήνες που χρησιμοποιούν προηγμένες τεχνικές για γρήγορη εκτέλεση ακόμα και ¨δύσκολων¨ εφαρμογών, ακανόνιστων δηλαδή ως προς τις εντολές τους. Από την άλλη, οι GPUs , αρχικά σχεδιασμένες για την Απόδοση γραφικών, συντελούνται από ένα σύνολο περισσότερων (συνήθως χιλιάδων) απλούστερων πυρήνων, οι οποίοι εκτελούν τις ίδιες πράξεις σε διαφορετικά δεδομένα, με αποτέλεσμα να υπερέχουν έναντι των CPUs σε ¨κανονικές¨ όσον αφορά την επεξεργασία δεδομένων και υπολογιστικά έντονες εφαρμογές [5]. Με την πάροδο του χρόνου, οι GPUs έγιναν πιο ευέλικτες και πλέον είναι ικανές να επιταχύνουν

εφαρμογές γενικού σκοπού, που προέρχονται από χώρους όπως τη μηχανική μάθηση, τις επιστημονικές προσομοιώσεις και την επεξεργασία βίντεο.

1.1.2 Αναδιαμορφώσιμο Υλικό & FPGAs

Τα ετερογενή υπολογιστικά συστήματα περιλαμβάνουν συχνά εξειδικευμένους επιταχυντές υλικού, όπως Ολοκληρωμένα Κυκλώματα Ειδικής Εφαρμογής (ASIC) και Συστοιχίες Προγραμματιζόμενων Πυλών (FPGA). Ενώ τα ASICs προσφέρουν υψηλές επιδόσεις και χαμηλή ισχύ, συνοδεύονται από υψηλό κόστος σχεδιασμού και κατασκευής. Τα FPGAs, από την άλλη πλευρά, παρέχουν ευελιξία και χαμηλό κόστος σχεδίασης, παρά τις προκλήσεις όπως η αυξημένη επιφάνεια πυριτίου και ο πολύπλοκος προγραμματισμός [6]. Λόγω της δυνατότητας αναδιαμόρφωσης, τα FPGAs είναι ιδανικά για εξομοίωση υλικού και ανάλυση επιδόσεων, προσφέροντας τη δυνατότητα για λεπτομερείς δοκιμές νέων αρχιτεκτονικών [7].

1.1.3 Βελτίωση της Απόδοσης ενός πυρήνα με το Σχήμα Εκτέλεσης LOOG

Ενώ οι GPUs επωφελούνται γενικά από τον Παραλληλισμό σε Επίπεδο Νήματος (TLP) , ορισμένες εφαρμογές γενικού σκοπού εμφανίζουν περιορισμένο TLP και αποτυγχάνουν να αξιοποιήσουν αποτελεσματικά τους πόρους της GPU . Το Light-weight Out-Of-Order GPU (LOOG) [8, 9] βελτιώνει τις επιδόσεις εισάγοντας Παραλληλισμό Επιπέδου Εντολών (ILP) στους πυρήνες της GPU . Παραδοσιακά, οι GPU εκτελούν τις εντολές με τη σειρά του προγράμματος, παρακολουθώντας τις εξαρτήσεις δεδομένων των καταχωρητών με τη χρήση Scoreboarding. Το LOOG επαναχρησιμοποιεί τα Collector Units (CUs) της GPU ως Reservation Stations από τον αλγόριθμο του Tomasulo , εισάγοντας τη μετονομασία καταχωρητών και την ξεχωριστή αναδιάταξη εντολών μνήμης για να επιτρέψει την εκτέλεση Εκτός Σειράς, οδηγώντας σε βελτιωμένη ταχύτητα των εφαρμογών και αποδοτικότερη χρήση των πόρων της GPU .

1.1.4 Εξομοίωση GPGPU με Vortex-GPU

Για την αξιολόγηση νέων μικροαρχιτεκτονικών GPUs, απαιτούνται εργαλεία εξομοίωσης υλικού και προσομοίωσης. Πλαίσια όπως το Accel-Sim [10] μοντελοποιούν την Απόδοση της GPU στο λογισμικό, αλλά για την εξομοίωση υλικού, οι επιλογές είναι περιορισμένες. Το Vortex-GPU , ένα πλαίσιο RTL σχεδίασης ανοικτού κώδικα βασισμένο σε RISC-V , υποστηρίζει εξομοίωση GPGPU σε πλατφόρμες FPGA [11, 12]. Το Vortex επεκτείνει την Αρχιτεκτονική Συνόλου Εντολών του RISC-V συμπεριλαμβάνοντας χαρακτηριστικά μιας GPU , επιτρέποντας την έρευνα και τον πειραματισμό με διάφορες παραμέτρους της GPU σε ένα περιβάλλον ανοικτού κώδικα.

1.1.5 Ανασκόπηση της πρότασης

Το προταθέν σχήμα εκτέλεσης LOOG επιδεικνύει υποσχόμενες βελτιώσεις επιδόσεων και εκτιμήσεις για την Επιφάνεια της συσκευής και την καταναλισκόμενη Ισχύ. Ωστόσο, μέχρι στιγμής, το LOOG έχει μοντελοποιηθεί μόνο σε εργαλεία προσομοίωσης (GPGPU-Sim και Accel-Sim), όπου οι εκτιμήσεις βασίζονται γραμμικά κλιμακούμενες τιμές καθώς τροποποιείται η αρχιτεκτονική της GPU - με αποτέλεσμα αποκλίσεις σε σύγκριση με μια πραγματική υλοποίηση υλικού. Επίσης, αυτά τα εργαλεία δεν παρέχουν εκτίμηση της κρίσιμης διαδρομής της αρχιτεκτονικής, η οποία καθορίζει τη συχνότητα ρολογιού της GPU .

Για να καλυφθεί αυτό το κενό, προτείνουμε την υλοποίηση του LOOG σε πραγματικό υλικό με λεπτομερή RTL περιγραφή του μηχανισμού του χρησιμοποιώντας μια γλώσσα περιγραφής υλικού (HDL) όπως η SystemVerilog . Παρόλο που μια τέτοια υλοποίηση για εξομοίωση σε FPGA παρουσιάζει περιορισμούς όσον αφορά το μέγεθος και τη συγκρισιμότητα με εμπορικές GPUs , προσφέρει αρκετά πλεονεκτήματα, όπως οι ταχύτεροι χρόνοι εκτέλεσης, η βελτιωμένη παρατηρησιμότητα σε επίπεδο σήματος. Ακόμα, η δυνατότητα αναδιαμόρφωσης των FPGA διευρύνει τον χώρο σχεδίασης επιτρέποντας τη μελέτη διαφόρων μεγεθών και συμβιβασμών. Αυτή η προσέγγιση θα αποδώσει ακριβέστερες εκτιμήσεις της κατανάλωσης Ενέργειας και της χρήσης των πόρων, που είναι ενδεικτική της επιφάνειας της τελικής συσκευής.

Ως αρχή της σχεδίασης επιλέγουμε την έκδοση 2.0 του πλαισίου Vortex GPU , καθώς είναι ανοικτού κώδικα και βασισμένη σε RISC-V , αλλά και λόγω των επιλογών παραμετροποίησης (ως προς τα νήματα ανά warp, τα warps ανά πυρήνα και τα μεγέθη των διαφόρων στοιχείων), οι οποίες διευρύνουν περαιτέρω το χώρο σχεδίασης. Η προσέγγισή μας περιλαμβάνει την τροποποίηση του σχήματος σωλήνωσης της Vortex ενσωματώνοντας πρώτα ένα στάδιο Συλλογής Ορισμάτων (Operand Collect) εξοπλισμένο με Collector Units -ένα χαρακτηριστικό που απουσιάζει από τον αρχικό σχεδιασμό, σε αντίθεση με τις εμπορικές GPU - και στη συνέχεια την εφαρμογή των μηχανισμών για εκτέλεση Εκτός Σειράς του LOOG , προσαρμοσμένων στα χαρακτηριστικά της αρχιτεκτονικής της Vortex . Στη συνέχεια, διερευνούμε διάφορες βελτιστοποιήσεις και συμβιβασμούς όσον αφορά τις επιδόσεις και τους περιορισμούς που αφορούν ειδικά τα FPGA , όπως η συμφόρηση λόγω της δρομολόγησης και οι επιβαρύνσεις από τη χρήση των πόρων.

Για την πειραματική αξιολόγηση, χαρακτηρίσαμε 21 εφαρμογές, αφού τις ομαδοποιήσαμε σε πέντε κατηγορίες με βάση τις πηγές καθυστέρησής τους όταν εκτελούνται στην βασική αρχιτεκτονική της Vortex . Αυτή η ταξινόμηση μας επιτρέπει να ορίσουμε με ακρίβεια τις εφαρμογές που είναι πιο ευαίσθητες στο LOOG .

Τέλος, εκτελούμε μια ανάλυση της αρχιτεκτονικής LOOG-Vortex για τον προσδιορισμό των βέλτιστων σχεδιαστικών παραμέτρων και διαμορφώσεων εντός του χώρου σχεδίασής μας. Η αξιολόγησή μας θα συγκρίνει τα επιτευχθέντα κέρδη όσον αφορά

την επίδοση και τις επιβαρύνσεις της χρήσης των πόρων και της κατανάλωσης Ενέργειας με εκείνες της αρχικής Vortex . Τα αποτελέσματα υποδεικνύουν μέση βελτίωση των επιδόσεων έως και 23,5%, επιβάρυνση της επιφάνειας που κυμαίνεται από 4,5% έως 27,5% (με περίπου 5% στις καλύτερες διαμορφώσεις) και μείωση του γινομένου Ισχύος-Καθυστέρησης - που αντιστοιχεί στην κατανάλωση Ενέργειας της συσκευής - έως και περίπου 17% σε σχέση με την αρχική σχεδίαση.

1.1.6 Συνεισφορές της Εργασίας

Η παρούσα εργασία παρουσιάζει αρκετές συνεισφορές στον τομέα της αρχιτεκτονικής των GPUs . Οι κυριότερες από αυτές περιλαμβάνουν:

- **Επεκτεταμένη Δυνατότητα Ιχνηλάτησης:** Παροχή λεπτομερών μηχανισμών ιχνηλάτησης με ακρίβεια κύκλου για όλα τα προστιθέμενα σήματα στην εκτέλεση εφαρμογών σε προσομοίωση RTL .
- **Νέοι μετρητές επιδόσεων:** Εισαγωγή μετρητών επιδόσεων για προσομοίωση RTL και εξομοίωση σε FPGA , με στόχο τη συλλογή στατιστικών στοιχείων και μετρήσεων κατά τη διάρκεια εκτέλεσης, όπως:
  - Λειτουργίες ανάγνωσης και εγγραφής αρχείου καταχωρητών
  - Επαναταξινομημένες εντολές & αποστάσεις επαναταξινόμησης
  - Καθυστερήσεις που εξαρτώνται από το σχήμα εκτέλεσης LOOG
  - Χρήση των Collector Units & περίοδος ανάθεσής τους σε εντολές
  - Χρήση του RRS & περίοδος ανάθεσης των πεδίων του σε εντολές
- **Χαρακτηρισμός εφαρμογών της Vortex :** Ομαδοποίηση των εφαρμογών αξιολόγησης της Vortex με βάση τις πηγές καθυστερήσεών τους και ανάλυση της συμπεριφοράς κάθε ομάδας στο LOOG-Vortex .
- **Εύρεση των βέλτιστων παραμέτρων του LOOG για τη Vortex GPU 2.0:** Βελτιστοποίηση του LOOG για την αποτελεσματική ενσωμάτωση και αξιοποίηση των πόρων της έκδοσης 2.0 της Vortex GPU , όσον αφορά τις μετρικές:
  - Απόδοση
  - Χρησιμοποίηση πόρων του FPGA
  - Κατανάλωση Ισχύος του FPGA

1.1.7 Δομή της Εργασίας

Το υπόλοιπο της παρούσας εργασίας διαρθρώνεται ως εξής:

- Το Κεφάλαιο 3 εμβαθύνει στο θεωρητικό υπόβαθρο του παράλληλου υπολογισμού, περιλαμβάνοντας στρατηγικές για την αξιοποίηση του Παραλληλισμού, τις βασικές αρχές της αρχιτεκτονικής των GPUs γενικού σκοπού και το αναδιαμορφώσιμο υλικό. Παρουσιάζει επίσης το πλαίσιο Vortex RTL και το σχήμα LOOG για εκτέλεση Εκτός Σειράς, τα οποία είναι κεντρικά για την παρούσα μελέτη.

- Το Κεφάλαιο 4 διερευνά την υπάρχουσα έρευνα που σχετίζεται με τον Παραλληλισμό σε Επίπεδο Εντολών στις GPU και τα πλαίσια RTL που εξομοιώνουν γενικού σκοπού GPUs , επισημαίνοντας τις προόδους και εντοπίζοντας τα κενά στο σημερινό τοπίο.

- Το Κεφάλαιο 5 περιγράφει λεπτομερώς την ενσωμάτωση του σχήματος LOOG εντός της GPU Vortex , τη δομή των προστιθέμενων στοιχείων και το πλήρες σχήμα εκτέλεσης. Διερευνά επίσης τις απαραίτητες προσαρμογές και βελτιστοποιήσεις της υλοποίησης και, τέλος, χαρακτηρίζει τις εφαρμογές που χρησιμοποιήθηκαν για την πειραματική αξιολόγηση.

- Στο Κεφάλαιο 6 αξιολογείται η Απόδοση της υλοποίησης LOOG-Vortex μέσω δοκιμών με κλιμάκωση των παραμέτρων του για την επίτευξη της επιθυμητής Απόδοσης και αναλύονται τα αποτελέσματα.

- Με το Κεφάλαιο 7 ολοκληρώνεται η εργασία, συνοψίζοντας τις κύριες συνεισφορές της έρευνας. Επίσης, περιγράφονται πιθανές μελλοντικές ερευνητικές κατευθύνσεις.

1.2 Θεωρητικό Υπόβαθρο

Αυτό το υποκεφάλαιο παρέχει τις θεωρητικές βάσεις για την κατανόηση της προτεινόμενης αρχιτεκτονικής LOOG-Vortex . Ξεκινά με θεμελιώδη θέματα όπως η δομή των σύγχρονων αρχιτεκτονικών GPGPUs , προχωρώντας σε πιο σύνθετα θέματα όπως η σχεδίαση RTL για αναδιαμορφώσιμο υλικό όπως τα FPGAs . Επίσης, παρέχεται αναλυτική εξήγηση του σχήματος εκτέλεσης Out-Of-Order LOOG και του πλαισίου Vortex GPU, το οποίο αποτελεί θεμέλιο της προτεινόμενης αρχιτεκτονικής.

1.2.1 Παράλληλος Υπολογισμός και Στρατηγικές Παραλληλισμού

Παράλληλος Υπολογισμός: Ο παράλληλος υπολογισμός αναφέρεται στη διαδικασία διαχωρισμού ενός προβλήματος σε μικρότερα μέρη και επίλυσης αυτών ταυτόχρονα μέσω πολλών φυσικών επεξεργαστών [13].

Τύποι Παραλληλισμού

Παραλληλισμός σε Επίπεδο Δεδομένων (DLP) : Ο DLP εκμεταλλεύεται τον παραλληλισμό σε δεδομένα οργανωμένα σε δομές όπως πίνακες, εκτελώντας την ίδια λειτουργία σε πολλαπλά δεδομένα ταυτόχρονα [14].

Παραλληλισμός σε Επίπεδο Εργασίας (TLP) : Ο TLP διαχωρίζει ένα πρόγραμμα σε ανεξάρτητες εργασίες που εκτελούνται παράλληλα [15].

Παραλληλισμός σε Επίπεδο Εντολής (ILP) : Ο ILP αφορά την εκτέλεση πολλών εντολών ταυτόχρονα μέσα σε έναν κύκλο επεξεργασίας [16].

Σωλήνωση: Η τεχνική της σωλήνωσης αφορά στην εκτέλεση διαφορετικών μερών μιας εντολής σε διαδοχικές μονάδες, επιτρέποντας την ταυτόχρονη εκτέλεση εντολών [17].

Εκτέλεση Εκτός Σειράς: Η εκτέλεση Εκτός Σειράς επιτρέπει την επαναταξινομημένη εκτέλεση εντολών με βάση τη διαθεσιμότητα των πόρων, βελτιώνοντας την Απόδοση [18].

SIMD: Στην τεχνολογία SIMD , η ίδια εντολή εκτελείται από πολλούς επεξεργαστές για διαφορετικά δεδομένα [19].

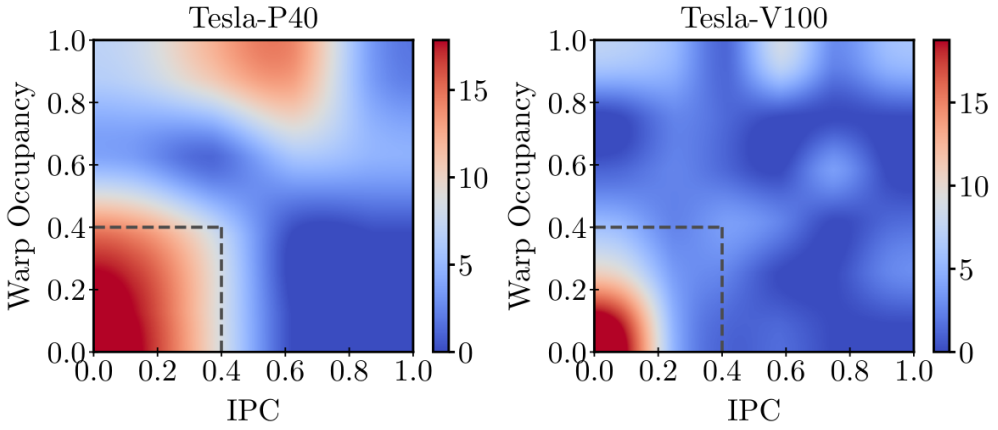
1.2.2 Βασικά Στοιχεία των GPUs

GPGPU: Ως GPGPUs χαρακτηρίζονται οι Επεξεργαστές Γραφικών γενικού σκοπού, οι οποίοι επιτρέπουν την εκτέλεση γενικών υπολογισμών με την ίδια υλική υποδομή που χρησιμοποιείται για την Απόδοση γραφικών [20].

Kernels: Οι kernels είναι τμήματα κώδικα που εκτελούνται σε GPUs , καθοριζόμενα από την CPU [21].

Νήματα: Τα νήματα της GPU εκτελούν ταυτόχρονα την ίδια εντολή σε διαφορετικά δεδομένα [22].

Warps: Τα warps στις GPUs είναι ομάδες νημάτων τα οποία εκτελούν την ίδια εντολή ταυτόχρονα [23].



Σχήμα 1.1: Κατανομή της χρήσης του Warp - IPC για 115 γενικής χρήσης kernels σε δύο φυσικές πλατφόρμες GPU [28].

Streaming Multiprocessors (SMs): Κάθε SM στην GPU εκτελεί παράλληλες εργασίες μέσω πολλαπλών μονάδων επεξεργασίας [24].

Collector Units (CUs): Τα Collector Units είναι δομές της GPU που απασχολούνται από τις εντολές μέχρι τη συλλογή των ορισμάτων τους [21].

1.2.3 Αναδιαμορφώσιμο Υλικό και Τεχνικές Εξομοίωσης

Reconfigurable Hardware: Οι αναδιαμορφώσιμες αρχιτεκτονικές συνδυάζουν έναν επεξεργαστή με ένα αναδιαμορφώσιμο μπλοκ, προσφέροντας την απαραίτητη ευελιξία στα ενσωματωμένα συστήματα [25].

Field Programmable Gate Arrays (FPGAs): Τα FPGAs είναι πλατφόρμες που περιλαμβάνουν προγραμματιζόμενα λογικά μπλοκ και διακόπτες δρομολόγησης για την υλοποίηση ψηφιακών κυκλωμάτων [25].

RTL Σχεδίαση: Η σχεδίαση σε επίπεδο RTL χρησιμοποιεί γλώσσες περιγραφής υλικού όπως η Verilog και η VHDL για την περιγραφή κυκλωμάτων [26].

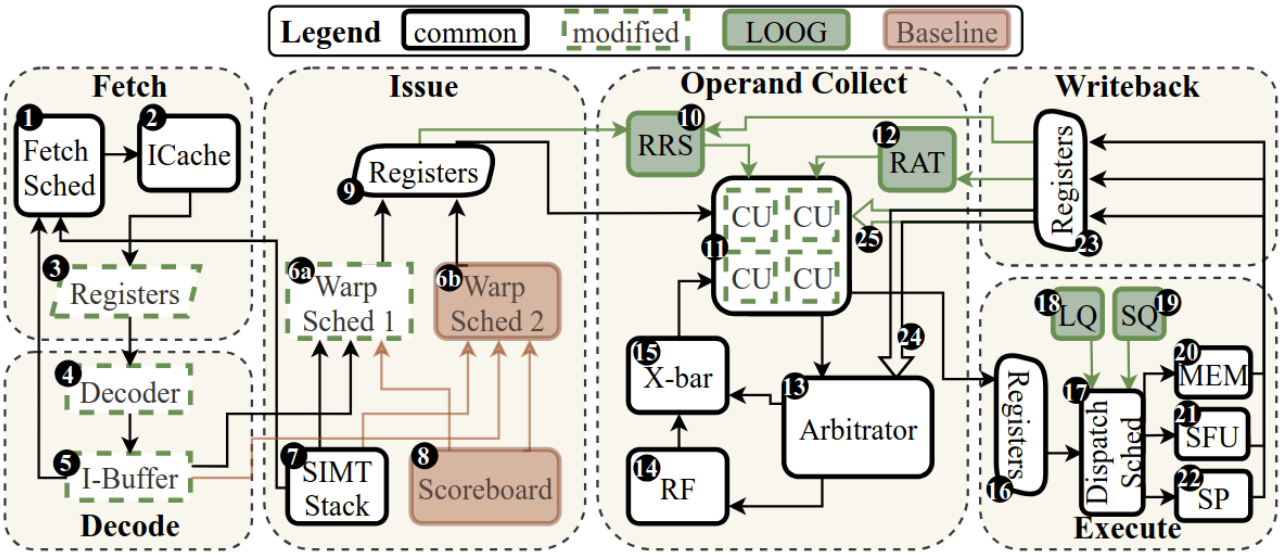
Εξομοίωση στο Υλικό: Η εξομοίωση στο υλικό χρησιμοποιεί πλατφόρμες όπως τα FPGAs για την προσομοίωση και τον έλεγχο συστημάτων [27].

1.2.4 Λεπτομέρειες του LOOG

Η κινητήρια δύναμη πίσω από το σχήμα εκτέλεσης Light-weight Out-Of-Order GPU (LOOG) [8, 9, 28] είναι η παρατήρηση ότι ορισμένα kernels που εκτελούνται σε GPGPUs επιτυγχάνουν χαμηλή χρησιμοποίηση πόρων, μη εκμεταλλευόμενα πλήρως τον Παραλληλισμό σε Επίπεδο Νήματος (TLP) που παρέχει το υλικό. Αυτό φαίνεται στο Σχήμα 1.1, όπου ένα μεγάλο ποσοστό kernels σε δύο πλατφόρμες GPU παρουσιάζει χαμηλή χρήση των warps και χαμηλό IPC. Το LOOG αντιμετωπίζει αυτά τα

προβλήματα εκμεταλλευόμενο τον Παραλληλισμό σε Επίπεδο Εντολών (ILP) μέσω Εκτέλεσης Εκτός Σειράς, χρησιμοποιώντας μια ελαφριά εκδοχή του αλγορίθμου του Tomasulo, προσαρμοσμένη για τη σωλήνωση μιας GPU. Το σχήμα σωλήνωσης του LOOG φαίνεται στο Σχήμα 1.2.

Το LOOG επιτρέπει την επαναταξινόμηση εντολών επαναχρησιμοποιώντας τα Collector Units της GPU ως Σταθμούς Κράτησης των εντολών σε αυτά μέχρι να επιλυθούν οι πιθανές εξαρτήσεις δεδομένων. Αντί του παραδοσιακού Scoreboard για την παρακολούθηση των εξαρτήσεων Ανάγνωσης μετά από Εγγραφή (RAW), το LOOG χρησιμοποιεί έναν Πίνακα Αντιστοίχισης Καταχωρητών (RAT) για να παρακολουθεί τις εξαρτήσεις RAW. Αντίστοιχα, εξαλείφει εκείνες που αφορούν Ανάγνωση μετά από Εγγραφή και Ανάγνωση μετά από Ανάγνωση (WAR) και (WAW) με την τεχνική της μετονομασίας καταχωρητών, αντικαθιστώντας το όνομα του καταχωρητή με τον αντίστοιχο αριθμό ταυτοποίησης του CU. Όταν μια εντολή ανατίθεται σε ένα CU, πρώτα συμβουλεύεται το RAT για κάθε πηγή καταχωρητή. Εάν ο καταχωρητής έχει ήδη μετονομαστεί, η εγγραφή του περιέχει τον αριθμό του CU που παράγει την τιμή του. Αυτή η τιμή αντιγράφεται στο CU της εντολής και το αποτέλεσμα καταγράφεται όταν γίνει η εγγραφή του από το CU με τον εν λόγω αριθμό ταυτοποίησης.



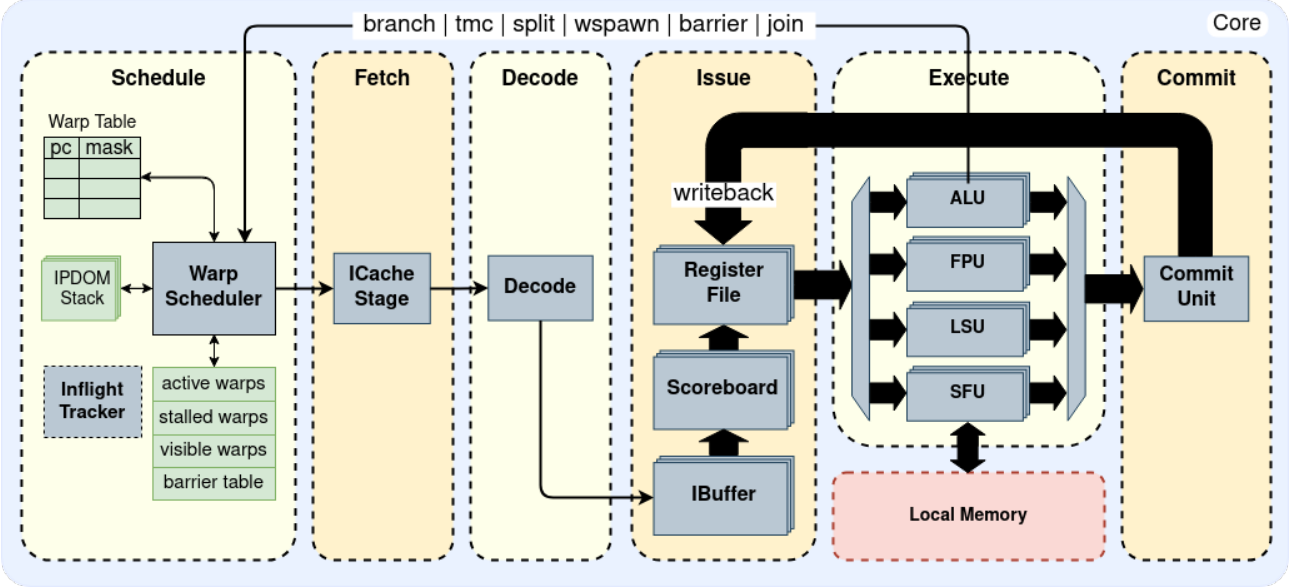
Σχήμα 1.2: Οι τροποποιήσεις του LOOG πάνω στο βασικό σχήμα σωλήνωσης της GPU [28].

Επιπλέον, το LOOG εισάγει έναν ειδικό μηχανισμό για την επαναταξινόμηση των εντολών πρόσβασης στη μνήμη, που μπορεί να περιλαμβάνουν και εξαρτήσεις διευθύνσεων. Αυτές δεσμεύουν θέσεις είτε σε μια ουρά Φόρτωσης από τη Μνήμη (LQ) είτε σε μια ουρά Αποθήκευσης στη Μνήμη (SQ) για την παρακολούθηση αυτών των εξαρτήσεων. Πριν την επαναταξινόμηση οποιασδήποτε τέτοιας εντολής, συγκρίνεται η διεύθυνσή της με τις διευθύνσεις προηγούμενων εντολών αποθήκευσης. Όταν

διαπιστωθεί ισότητα, η εκτέλεση της εντολής αναστέλλεται μέχρι να επιλυθεί το πρόβλημα.

Για να βελτιστοποιήσει περαιτέρω τη χρήση των πόρων, το LOOG χρησιμοποιεί μια Στοίβα Μετονομασίας Καταχωρητών (RRS), η οποία διατηρεί μια λίστα μοναδικών αριθμών που χρησιμοποιούνται στον RAT , αντί των πραγματικών αριθμών ταυτοποίησης των CUs. Κατά την ανάθεση ενός CU στην εντολή, της αποδίδεται επίσης ένας μοναδικός αριθμός από το RRS . Ο τελευταίος καταγράφεται στο RAT και οι εξαρτώμενες εντολές το χρησιμοποιούν για να λάβουν τα αποτελέσματα. Αυτή η προσέγγιση επιτρέπει στην εντολή να ελευθερώσει το CU πριν την εκτέλεση της πράξης, διατηρώντας μόνο τον αριθμό του RRS μέχρι την ολοκλήρωση της εγγραφής του αποτελέσματος.

1.2.5 Σωλήνωση της Vortex GPU



Σχήμα 1.3: Επισκόπηση της μικροαρχιτεκτονικής της σωλήνωσης της Vortex GPU [29].

Η Vortex GPU [11] [12] [30] χρησιμοποιεί μια αρχιτεκτονική σωλήνωσης 6 σταδίων RISC-V που συνδυάζεται με εξειδικευμένα στοιχεία για επεξεργασία SIMT . Το σχήμα σωλήνωσης περιλαμβάνει τα κάτωθι στάδια:

- **Schedule:** Διαχείριση των ενεργών και ανεσταλμένων warps.
- **Fetch:** Ανάκτηση εντολών από τη μνήμη και διαχείριση αιτημάτων της ιεραρχίας μνημών cache.
- **Decode:** Μετάφραση των εντολών και προσαρμογή στις περιπτώσεις αλλαγής της ροής εκτέλεσης.

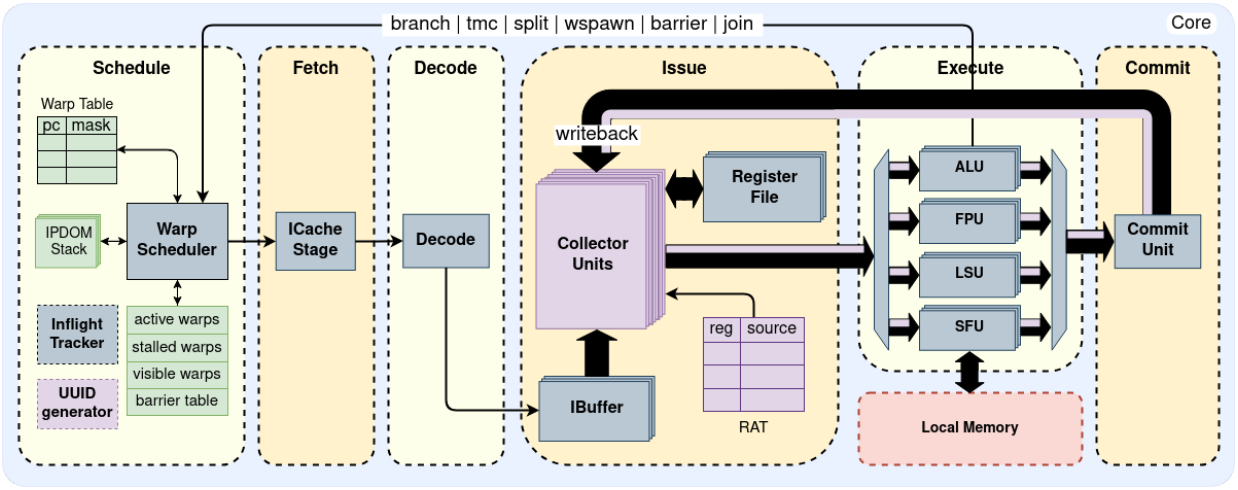
- **Issue:** Παρακολούθηση εξαρτήσεεων δεδομένων και πρόσβαση σε δεδομένα του αρχείου καταχωρητών.
- **Execute:** Εκτέλεση των εντολών σε ειδικές μονάδες επεξεργασίας (ALU, FPU, LSU, SFU).
- **Commit:** Εγγραφή των αποτελεσμάτων στους καταχωρητές και ενημέρωση του Scoreboard .

Η αρχιτεκτονική, επιπλέον, υποστηρίζει κλιμάκωση μέσω ιεραρχιών L1 και L2 caches , επιτρέποντας τον αποδοτικό διαμοιρασμό δεδομένων μεταξύ πυρήνων.

1.3 Λεπτομέρειες της Υλοποίησης

1.3.1 Δομές που προστέθηκαν

Θα συνεχίσουμε περιγράφοντας τις βασικές δομές που προστέθηκαν στην μικροαρχιτεκτονική της Vortex , οι οποίες φαίνονται με μωβ χρώμα στο Σχήμα 1.4.



Σχήμα 1.4: Η επικαιροποιημένη μικροαρχιτεκτονική της Vortex GPU μετά την υλοποίηση του σχήματος εκτέλεσης LOOG.

1.3.2 Collector Units

Η βασική δομή που επιτρέπει την αναδιάταξη των εντολών στο LOOG είναι τα Collector Units (CUs), τα οποία δεν υπήρχαν στη Vortex, οπότε υλοποιήθηκαν με τιν αριθμό τους να είναι παραμετροποιήσιμος. Όπως αναφέρθηκε προηγουμένως, τα CUs κρατούνται από εντολές, επομένως είναι απαραίτητο να έχουν τη χωρητικότητα για όλες τις πληροφορίες της εντολής (διεύθυνση, από ποιο warp προέρχεται, ενεργά νήματα, καταχωρητές ορισμάτων κλπ.). Επιπλέον, στα CUs υπάρχουν πεδία σημάτων

που υποδεικνύουν την εγκυρότητα δεδομένων ή την κατάσταση του CU, ενώ τα μεγαλύτερα πεδία τους είναι αυτά που αφορούν τα δεδομένω των καταχωρητών-πηγών και του καταχωρητή του αποτελέσματος. Όλα αυτά τα πεδία φαίνονται αναλυτικά στο Σχήμα 1.5.

Collector Unit	
1	allocation status
1	dispatch status
1	rd data valid status
3	rs data valid status
3	rs to be read from RF
3 x (CU ID width)	rs source ID
(INSTR DATA width)	instr. UUID, warp ID, PC, tmask, regs etc.
T x (DATA width)	rd data
3 x T x (DATA width)	rs1 data, rs2 data, rs3 data

Σχήμα 1.5: Τα πεδία ενός Collector Unit) μαζί με τα αντίστοιχα μεγέθη τους σε δυαδικά ψηφία.

1.3.3 Πίνα ας Αντιστοίχισης Καταχωρητών

Στο LOOG δε χρησιμοποιείται Scoreboard για τον χειρισμό των εξαρτήσεων δεδομένων, οπότε η αρμοδιότητα αυτή ανατίθεται σε έναν Πίνακα Αντιστοίχησης Καταχωρητών (RAT). Αυτός έχει τόσες θέσεις όσοι οι καταχωρητές όλων των warps (64 καταχωρητές ανά warp) και κάθε θέση αποτελείται από δύο πεδία. Το πρώτο υποδεικνύει αν ο καταχωρητής πρόκειται να διαβαστεί από το Αρχείο Καταχωρητών και το δεύτερο είναι ένας αρισμός ταυτοποίησης CU, στην περίπτωση που πρόκειται να διαβαστεί από την επιστροφή του αποτελέσματος ενός άλλου CU. Το RATφαίνεται στο Σχήμα 1.6.

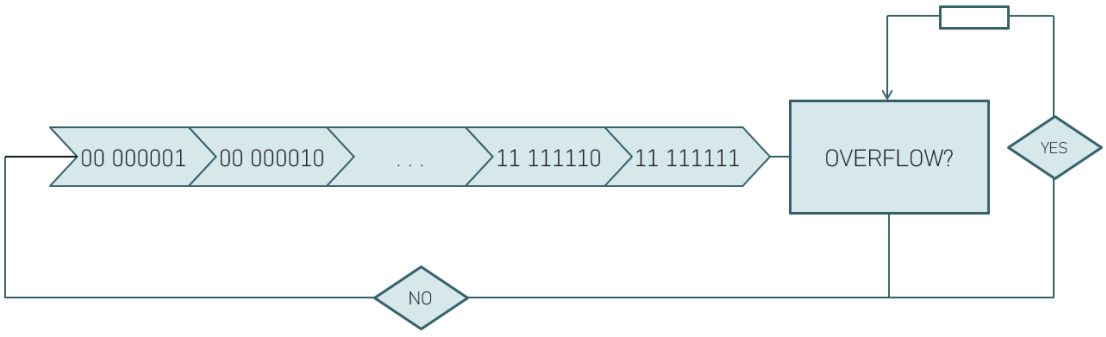
1.3.4 Γεννήτρια αριθμού UUID

Καθώς στο πλαίσιο αυτής της εργασίας δεν υλοποιείται μηχανισμός επαναταξινόμησης των εντολών πρόσβασης στη μνήμη, είναι απαραίτητη η πληροφορία της σωστής

RAT				
	warp0	warp1	...	warp(N-1)
reg0 (int)	CU ID			
	from RF			
...				
reg31 (int)				
reg0 (fp)				
...				
reg31 (fp)				

Σχήμα 1.6: Ο Πίνακας Αντιστοίχησης Καταχωρητών με τα αντίστοιχα πεδία του.

σειράς των εντολών. Για αυτόν τον σκοπό, θα ήταν χρήσιμος ένας μοναδικός, αύξων αριθμός ταυτοποίησης των εντολών ώστε να γνωρίζουμε πότε μια εντολή προηγείται μίας άλλης. Ωστόσο, κάτι τέτοιο θα απαιτούσε το μέγεθος αυτού του πεδίου να είναι τεράστιο, κάτι που στη σχεδίαση υλικού πρέπει να αποφευχθεί. Έτσι, χρησιμοποιείται μια κυκλική γεννήτρια αριθμών ταυτοποίησης UUIDανά warp, αποτελούμενων από 8 δυαδικά ψηφία χωρισμένα σε 2 κσι 6, που ξεκινά αναθέτοντας την τιμή 00000001. Κάθε επόμενη εντολή έχει μεγαλύτερο UUID, εκτός από την περίπτωση που η πρώτη εντολή έχει τα πιο σημαντικά της ψηφία ίσα με ΄11΄ και η δεύτερη αντίστοιχα ΄00΄, όπου η δεύτερη θεωρείται μεγαλύτερη. Με αυτόν τον τρόπο, επιτυγχάνεται ένα ¨περιθώριο ασφαλείας¨ 65 εντολών που μπορούν να εκτελεστούν μέχρι να δημιουργηθεί πρόβλημα. Αν, ωστόσο, γίνει κάτι τέτοιο, αυτό εντοπίζεται αμέσως κατά την άφιξη της νέας εντολής με αριθμό 00000001 στο στάδιο της Συλλογής Ορισμάτων και η εντολή περιμένει μέχρι να επιλυθεί το ζήτημα.

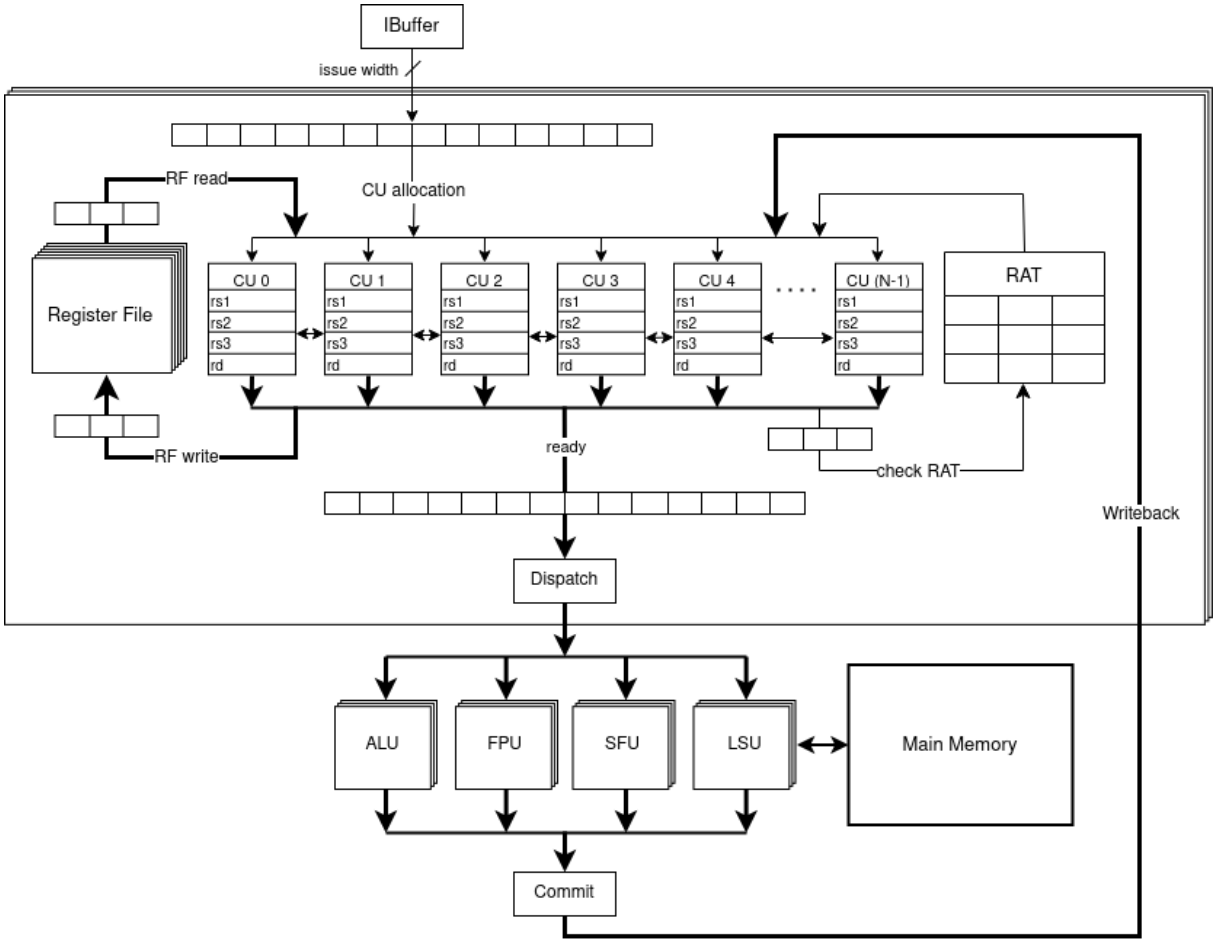


Σχήμα 1.7: Ελαφρύς μηχανισμός για την ανάθεση Μοναδικών Αριθμών Ταυτοποίησης UUID στις εντολές.

1.3.5 Ροή μιας εντολής στο LOOG-Vortex

Σε αυτή την ενότητα θα περιγραφεί η ροή μιας εντολής στο LOOG-Vortex από τη στιγμή που αυτή φτάνει στο στάδιο της Συλλογής Ορισμάτων μέχρι και το τέλος της εκτέλεσής της, στις δομές που φαίνονται στο Σχήμα 1.8. Αρχικά γίνεται η ανάθεση ενός από τα ελεύθερα CUs στην εντολή και αυτή αντιγράφει τις πληροφορίες της στα αντίστοιχα πεδία του, που περιγράφηκαν προηγουμένως. Στον επόμενο κύκλο ρολογιού, συμβουλεύεται το RAT για κάθε όρισμά της. Συγκεκριμένα, για τους καταχωρητές-πηγές, αντιγράφει από το αντίστοιχο πεδίο την πηγή από την οποία πρέπει να διαβαστεί ο καθένας από αυτούς, ενώ, αν πρόκειται να επιστρέψει το αποτέλεσμά της σε κάποιον καταχωρητή-προορισμό, γράφει σε εκείνο το πεδίο τον αριθμό ταυτοποίησης του CU . Η εντολή είναι πλέον έτοιμη να συλλέξει τα ορίσματά της. Αν χρειάζεται να διαβάσει κάποιο όρισμα από το Αρχείο Καταχωρητών, το CU της μπαίνει στην ουρά των CUs ¨για ανάγνωση¨ και, όταν επιλεχθεί, διαβάζει όσους καταχωρητές-πηγές χρειάζεται από το Αρχείο Καταχωρητών. Η ανάγνωση αυτή διαρκεί τόσους κύκλους, όσα είναι τα ορίσματα της εντολής που πρόκειται να διαβαστούν από το Αρχείο Καταχωρητών, καθώς μόνο ένα όρισμα μπορεί να διαβαστεί ανά κύκλο ρολογιού. Όταν όλα τα δεδομένα των καταχωρητών-πηγών της έχουν σημανθεί ως έγκυρα, η εντολή είναι έτοιμη να περάσει στο επόμενο στάδιο, δηλαδή αυτό της Εκτέλεσης στην αντίστοιχη μονάδα. Αν σε αυτό το σημείο διαπιστωθεί ότι η εντολή δεν επιστρέφει κάποιο αποτέλεσμα, τότε οι πόροι του CU απελευθερώνονται και αυτό σημαίνεται ως ελεύθερο ώστε να ανατεθεί σε επόμενη εντολή.

Αν, ωστόσο, η εντολή πρόκειται να επιστρέψει κάποια τιμή σε έναν καταχωρητή-προορισμό, το CU κρατείται μέχρι το τέλος της Εκτέλεσής της. Μετά την Εκτέλεση, η εντολή μπορεί να επιστρέψει μία ή περισσότερες εγγραφές αποτελεσμάτων στο αντοίστοιχο πεδίο του CU(πολλαπλές εγγραφές προκύπτουν στην περίπτωση που διαφορετικά νήματα του ίσιου warp επιστρέφουν τα αποτελέσματά τους σε διαφορετικές χρονικές στιγμές). Η τελευταία εγγραφή επισημαίνεται από μια ειδική σημαία τέλους η οποία πυροδοτεί άλλη μια επαφή με το RAT. Αν στο πεδίο του καταχωρητή-προορισμού εντοπιστεί ο αριθμός ταυτοποίησης του CU από το οποίο προέρχεται, η εντολή υποχρεούται να γράψει το αποτέλεσμά της στο Αρχείο Καταχωρητών στον επόμενο κύκλο ρολογιού, αλλιώς δεν είναι απαραίτητη αυτή η εγγραφή. Σε κάθε περίπτωση, στον ακόλουθο κύκλο γίνεται η μετάδοση του αποτελέσματος ώστε να συλλεχθεί από τα CUs που αναμένουν την άφιξή του. Στον επόμενο και τελευταίο κύκλο λειτουργίας της εντολής, απελευθερώνονται όλα τα πεδία του CU και αυτό σημαίνεται ως ελεύθερο, όπως αναφέρθηκε προηγουμένως.



Σχήμα 1.8: Οι δομές των σταδίων της Συλλογής Ορισμάτων, της Εκτέλεσης και της Ολοκλήρωσης της εντολής στο LOOG-Vortex.

1.3.6 Στοίβα Μετονομασίας Καταχωρητών

Σε αντίθεση με τις συμβατικές αρχιτεκτονικές ΓΠΥ, όπου οι εντολές παραμένουν στα CUs μόνο για λίγους κύκλους ρολογιού — έως ότου διαβαστούν τα δεδομένα των ορισμάτων από το Αρχείο Καταχωρητών — το σχήμα εκτέλεσης LOOG απαιτεί την παραμονή των εντολών στα CUs μέχρι και την εγγραφή του αποτελέσματος. Αυτό έχει ως αποτέλεσμα σημαντική αύξηση της μέσης περιόδου κατοχύρωσης ενός CU, γεγονός που δημιουργεί κινδύνους δομών για τις νέες εντολές που εισέρχονται στο στάδιο της Συλλογής Ορισμάτων, προκαλώντας καθυστερήσεις. Η προφανής λύση για την αποφυγή τέτοιων καθυστερήσεων θα ήταν η αύξηση του αριθμού των διαθέσιμων CUs . Ωστόσο, κάτι τέτοιο συνεπάγεται σημαντικό κόστος σε επίπεδο επιφάνειας, οδηγώντας σε σχεδιαστικό συμβιβασμό, ο οποίος αναλύεται εκτενέστερα σε επόμενο σημείο της εργασίας.

Η εναλλακτική που προτείνεται είναι η εισαγωγή μιας νέας δομής: της Στοίβας Μετονομασίας Καταχωρητών (RRS). Το RRS λειτουργεί ως ενδιάμεσος αποθηκευτικός χώρος για εντολές που έχουν περάσει προς την Εκτέλεση, αλλά δεν έχουν ακόμη

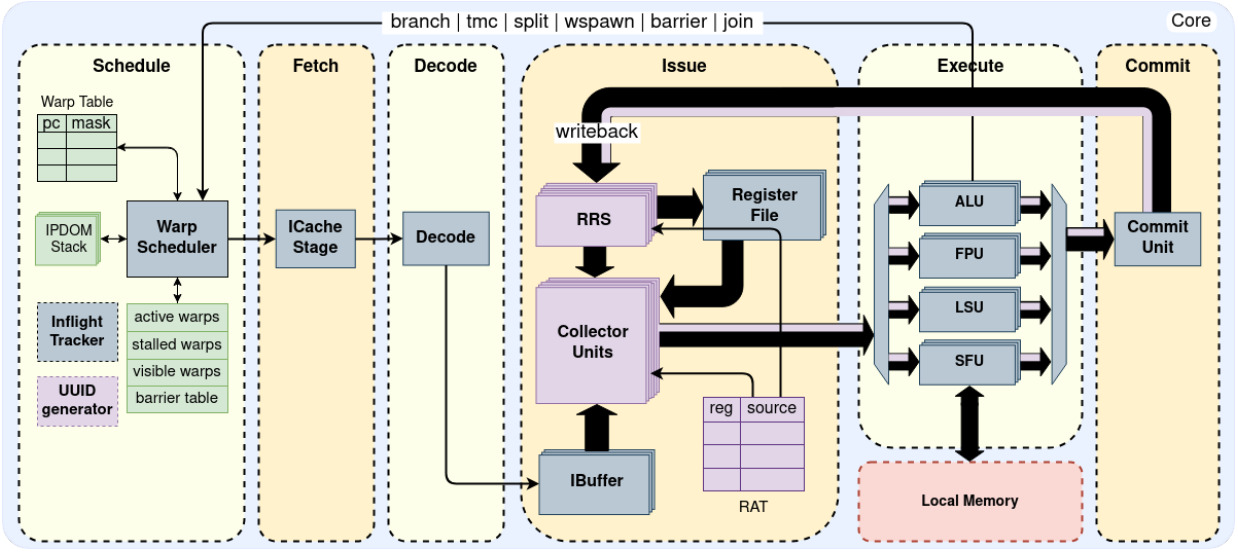
επιστρέψει το αποτέλεσμά τους. Κάθε θέση του RRS περιλαμβάνει μια ένδειξη απασχόλησης από εντολή, ένα πεδίο για τα δεδομένα του καταχωρητή-προορισμού, ενδείξεις των ενεργών νημάτων και της εγκυρότητας του αποτελέσματος, το οποίο ενεργοποιείται όταν εντοπιστεί το ειδικό σήμα τέλους των εγγραφών. Τα δεδομένα και τα πεδία εγκυρότητας μεταφέρονται από τα CUs του αρχικού σχεδιασμού στο RRS, χωρίς να προστίθεται σημαντικό κόστος σε επίπεδο υλικού.

Collector Unit	
1	allocation status
1	dispatch status
3	rs data valid status
3	rs to be read from RF
T	tmask
3 x (RRS ID width)	rs source ID
(INSTR DATA width) - T	instr. UUID, warp ID, PC, regs etc.
3 x T x (DATA width)	rs1 data, rs2 data, rs3 data

RRS entry	
1	allocation status
1	rd data valid status
T	tmask
T x (DATA width)	rd data

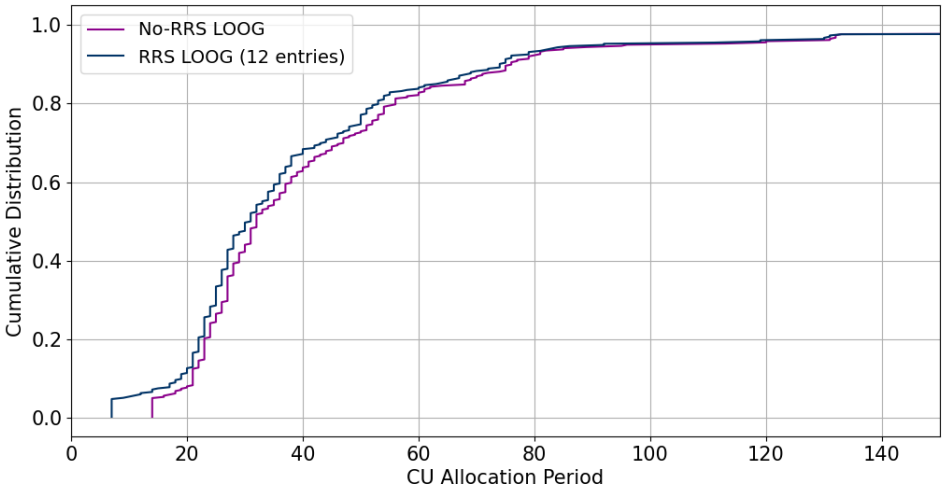
Σχήμα 1.9: Πεδία Collector Unit και RRS με τα αντίστοιχα μεγέθη τους σε δυαδικά ψηφία.

Μια θέση του RRS ανατίθεται σε κάθε εντολή (που δεν απαιτεί εγγραφή αποτελέσματος) ταυτόχρονα με την κατοχύρωση CU, κατά την είσοδο της στο στάδιο Συλλογής Ορισμάτων. Ο αριθμός της θέσης του RRS αποθηκεύεται στο αντίστοιχο πεδίο του CUκαι η σημαία απασχόλησης ενεργοποιείται. Κατά τη διέλευση της εντολής από το RAT, για εκείνες που έχουν προορισμό, ο αριθμός του RRS αποθηκεύεται στο πεδίο του καταχωρητή-προορισμού. Η εντολή προχωρά στο στάδιο της Εκτέλεσης με τον αριθμό του RRS αντί για εκείνον του CU. Αυτό επιτρέπει την απελευθέρωση του CU στον επόμενο κύκλο ρολογιού, καθώς τα αποτελέσματα θα αποθηκευτούν και θα μεταδοθούν όπως και προηγουμένως, αλλά από το RRS .

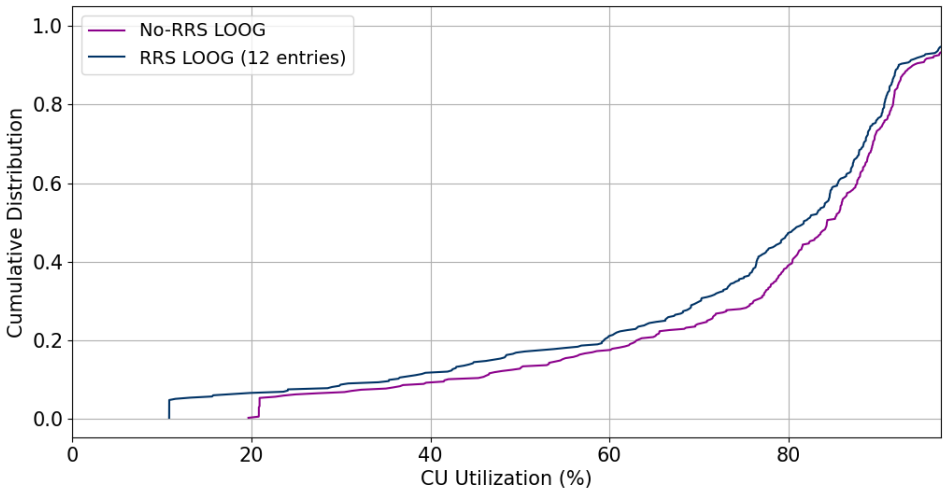


Σχήμα 1.10: Η τροποποιημένη αρχιτεκτονική LOOG-Vortex με προσθήκη του RRS.

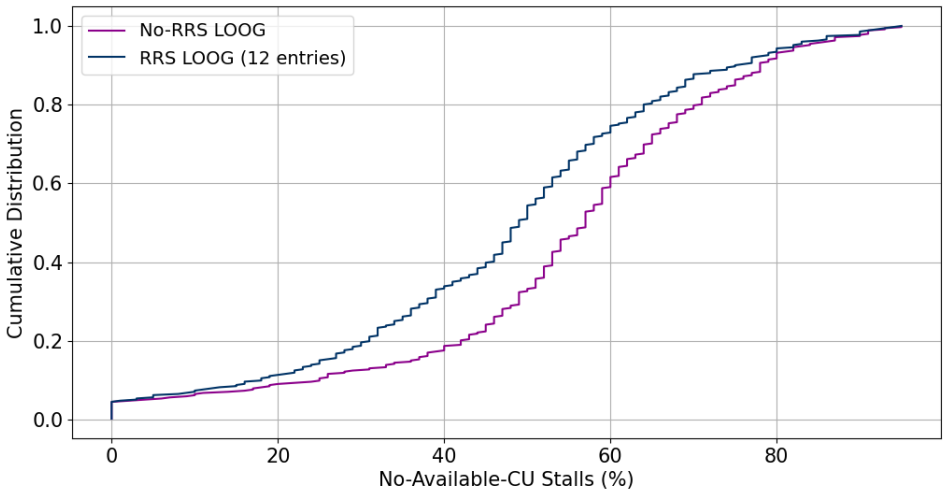
Η προσθήκη του RRS βελτιώνει σημαντικά τη διαθεσιμότητα των CUs, όπως αποδεικνύεται από την Συνάρτηση Κατανομής της περιόδου κατοχύρωσης των CUs στο Σχήμα 1.11. Παρατηρείται μείωση της συμφόρησης, επιτρέποντας σε περισσότερες εντολές να εισέλθουν εγκαίρως στο στάδιο της Συλλογής Ορισμάτων. Ακόμη, στο Σχήμα 1.13, παρατηρείται σαφής μείωση των καθυστερήσεων λόγω μη διαθέσιμων CUs. Τέλος, το Σχήμα 1.12 δείχνει πως η αξιοποίηση των CUs είναι πιο αποδοτική με την προσθήκη της RRS.



Σχήμα 1.11: Συνάρτηση Κατανομής της κατοχύρωσης των CUs για το LOOG-Vortex με και χωρίς RRS.



Σχήμα 1.12: Συνάρτηση Κατανομής της χρήσης των CUs για το LOOG-Vortex με και χωρίς RRS.



Σχήμα 1.13: Συνάρτηση Κατανομής των καθυστερήσεων λόγω μη διαθέσιμων CUs για το LOOG-Vortex με και χωρίς RRS.

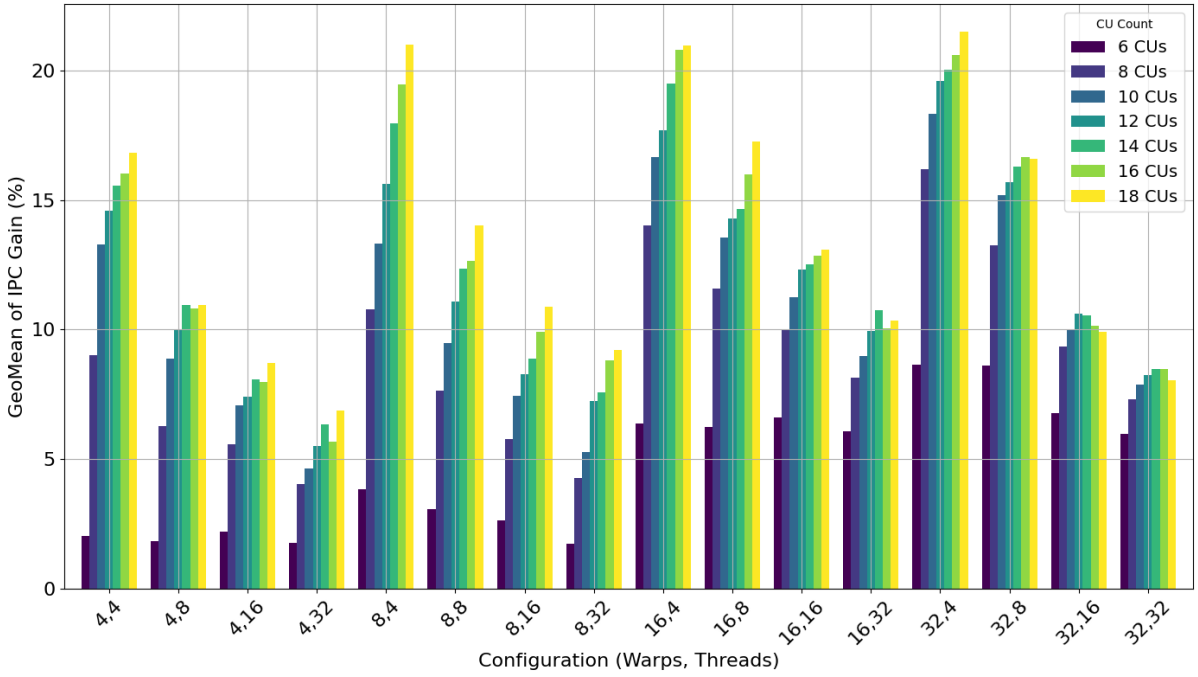
1.4 Αξιολόγηση

Σε αυτή την ενότητα, αξιολογείται η Απόδοση του LOOG-Vortex σε σύγκριση με την αρχική αρχιτεκτονική της Vortex GPGPU 2.0.

Αρχικά, αξιολογείται η Απόδοση του ΛΟΟΓ χωρίς την Στοίβα Μετονομασίας Καταχωρητών (RRS)όσον αφορά τις Εντολές ανά Κύκλο (IPC). Υπολογίζεται η γεωμετρική μέση τιμή των κερδών IPC (εκφρασμένων ως ποσοστά %) του LOOG-Vortex σε σχέση με το αρχικό και παρουσιάζονται αυτά τα κέρδη για κάθε διαμόρφωση SM

(warps, νήματα), όπως φαίνεται στο Σχήμα 1.14. Κάθε χρώμα αντιστοιχεί σε έναν συγκεκριμένο αριθμό διαθέσιμων CUs.

Αξιοσημείωτο είναι ότι κάθε διαμόρφωση δείχνει θετικά κέρδη, ξεπερνώντας διαρκώς τη βασική αρχιτεκτονική. Συνήθως, η αύξηση του αριθμού των CUs σχετίζεται με υψηλότερα κέρδη, καθώς η άφιξη νέων εντολών από το προηγούμενο στάδιο είναι λιγότερο πιθανό να μπλοκαριστεί από την έλλειψη κενών CUs. Επιπλέον, είναι ιδιαίτερα ενδιαφέρον ότι το LOOG-Vortex παρουσιάζει σημαντικό πλεονέκτημα στην Απόδοση όταν λειτουργεί με μικρότερο αριθμό νημάτων ανά warp. Αυτό οφείλεται στην αύξηση της δυνατότητας επαναταξινόμησης των εντολών όσο αυξάνονται οι εντολές ανά warp.

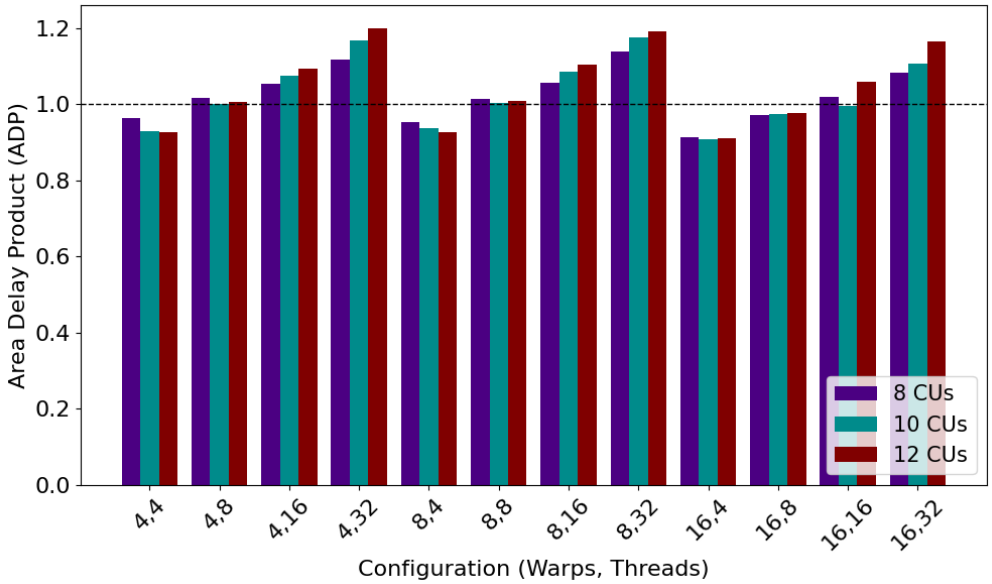


Σχήμα 1.14: Γεωμετρική μέση τιμή κερδών IPC (21 εφαρμογές) ανά διαμόρφωση για διαφορετικούς αριθμούς CUs στη μικροαρχιτεκτονική LOOG-Vortex χωρίς RRS.

Αξιολογούμε επίσης το LOOG-Vortex σε όρους Επιφάνειας και Κατανάλωσης Ενέργειας, υλοποιώντας το για μια πλατφόρμα FPGA (συγκεκριμένα την AMD Alveo U50 Data Center Acceleration Card). Με χρήση του εργαλείου Vivado 2021.1 για τη σύνθεση και την υλοποίηση, τα αποτελέσματα για την επιφάνεια υπολογίζονται από τους χρησιμοποιούμενους πόρους ως εξής:

$$\text{Επιφάνεια}(\%) = \frac{CLBRegisters(\%) + CLBLUTs(\%) + BRAM(\%) + DSPs(\%)}{4}$$

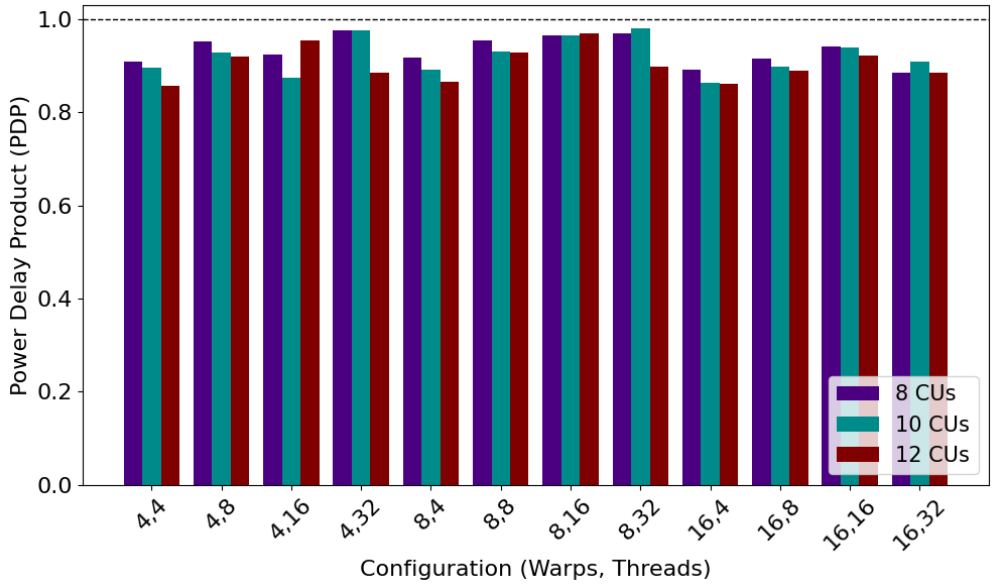
Η ενσωμάτωση των αποτελεσμάτων της Απόδοσης του LOOG-Vortex με τις μετρήσεις επιφάνειας για κάθε διαμόρφωση επιτυγχάνεται μέσω της μετρικής του γινομένου Επιφάνειας-Καθυστέρησης που απεικονίζεται στο Σχήμα 1.15. Με αυτή την προσέγγιση, η αρχική Vortex επιτυγχάνει γινόμενο Επιφάνειας-Καθυστέρησης (ADP) ίσο με 1 για όλες τις διαμορφώσεις, και τιμές κάτω από 1 υποδεικνύουν πιο βελτιστοποιημένο σχέδιο. Χρησιμοποιούμε αυτόν τον δείκτη για να αξιολογήσουμε το LOOG-Vortex για διαμορφώσεις με 8, 10 και 12 CUs.



Σχήμα 1.15: Γινόμενο Επιφάνειας-Καθυστέρησης του LOOG-Vortex χωρίς RRS ανά διαμόρφωση για διάφορους αριθμούς CUs (κανονικοποιημένο στην αρχική αρχιτεκτονική).

Σύμφωνα με αυτά τα ευρήματα, οι διαμορφώσεις του Vortex που ευνοούνται περισσότερο από το LOOG είναι αυτές με τα μικρότερα μεγέθη warp. Λιγότερα νήματα ανά warp αποφέρουν τα υψηλότερα κέρδη Απόδοσης σε σχέση με την αρχική διαμόρφωση, ενώ διατηρούν και τη χαμηλότερη χρήση πόρων. Συνεπώς, αυτές οι διαμορφώσεις θεωρούνται βέλτιστες για το σχέδιο μας.

Επιπλέον, χρησιμοποιώντας τις εκτιμήσεις του εργαλείου υλοποίησης, μπορούμε να αξιολογήσουμε την Καταναλισκόμενη Ενέργεια του LOOG-Vortex για διαφορετικές διαμορφώσεις και αριθμούς CUs. Όπως φαίνεται από το γινόμενο Ισχύος-Καθυστέρησης (PDP) του Σχήματος 1.16, όλες οι διαμορφώσεις είναι βελτιωμένες σε σχέση με την αρχική αρχιτεκτονική.

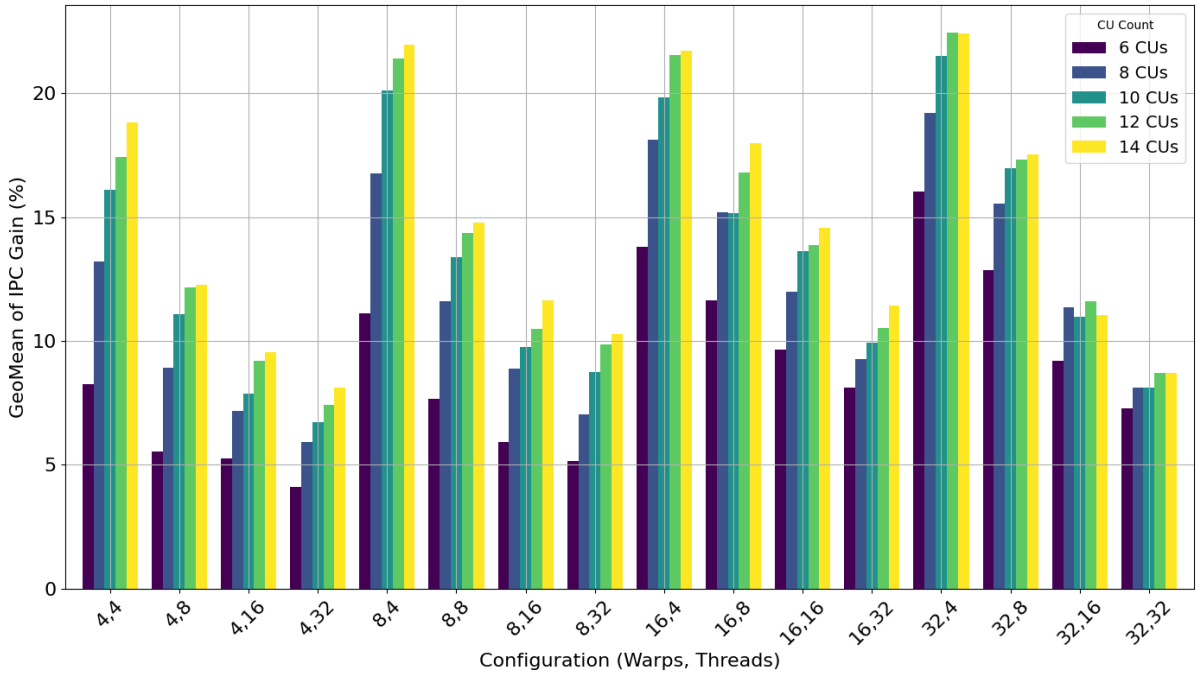


Σχήμα 1.16: Γινόμενο Ισχύος-Καθυστέρησης του LOOG-Vortex χωρίς RRS ανά διαμόρφωση για διάφορους αριθμούς CUs (κανονικοποιημένο στην αρχική αρχιτεκτονική).

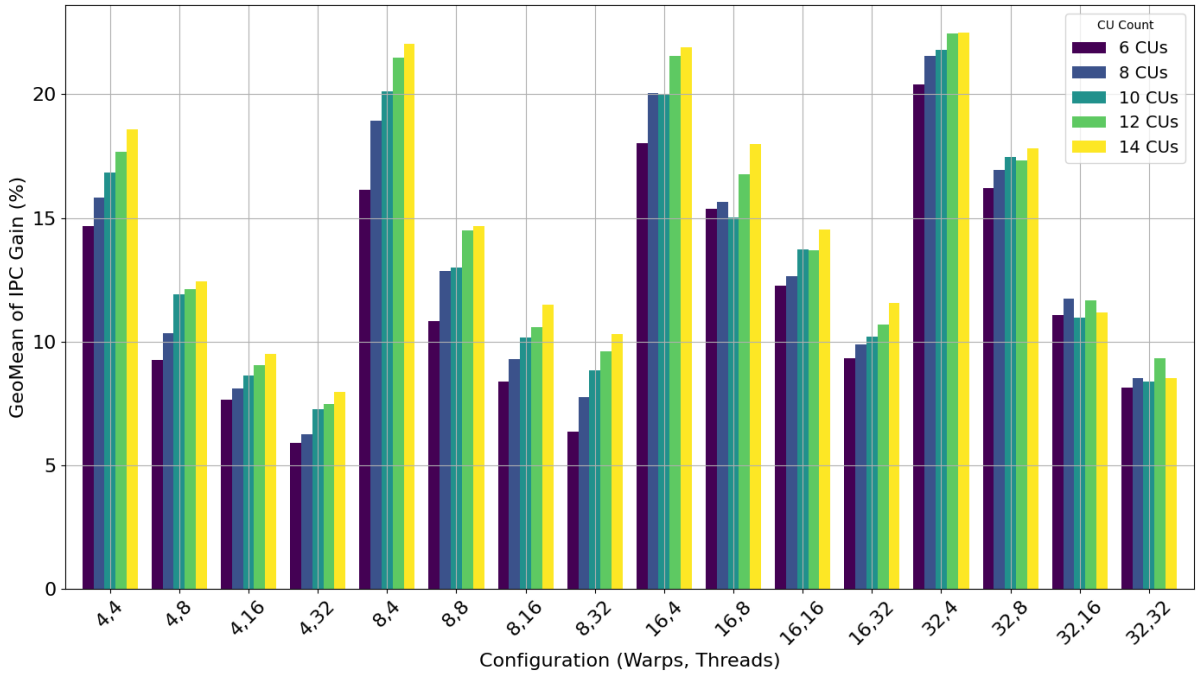
Το επόμενο βήμα είναι η αξιολόγηση του LOOG-Vortex με την προσθήκη του RRS όσον αφορά τις ίδιες μετρικές που χρησιμοποιήθηκαν προηγουμένως. Χρησιμοποιείται το ίδιο σύνολο εφαρμογών και εξετάζονται διαφορετικά μεγέθη του RRS, τα οποία εξαρτώνται από τον αριθμό των CUs. Συγκεκριμένα, χρησιμοποιούνται θέσεις RRS ίσες με τον αριθμό των CUs πολλαπλασιασμένο με έναν παράγοντα 1.5, 2 και 2.5, και τα αποτελέσματα για τις Εντολές ανά Κύκλο (IPC) παρουσιάζονται στον Πίνακα 1.1. Εύκολα παρατηρεί κανείς πως όταν προστίθενται επιπλέον θέσεις στο RRS, τα κέρδη Απόδοσης σταματούν να αυξάνονται για τους περισσότερους αριθμούς CUs στον παράγοντα 1.5 και για μικρούς αριθμούς CUs στον παράγοντα 2.

Πίνακας 1.1: Γεωμετρική μέση τιμή IPC για διαφορετικούς παράγοντες θέσεων του RRS του LOOG-Vortex.

CUs	No CUs	#RRS = #CUs × 1.5	#RRS = #CUs × 2	#RRS = #CUs × 2.5
6	4.5406	8.2399	11.1017	11.8398
8	8.6834	11.0765	12.0894	12.0894
10	10.3601	12.3409	12.6604	12.6620
12	11.4342	13.3033	13.3528	13.3528
14	12.2760	13.7937	13.7747	13.7747



Σχήμα 1.17: Γεωμετρική μέση τιμή IPC (21 εφαρμογές) ανά διαμόρφωση για διάφορους αριθμούς CUs στην αρχιτεκτονική LOOG-Vortex με RRS (παράγοντας 1.5).



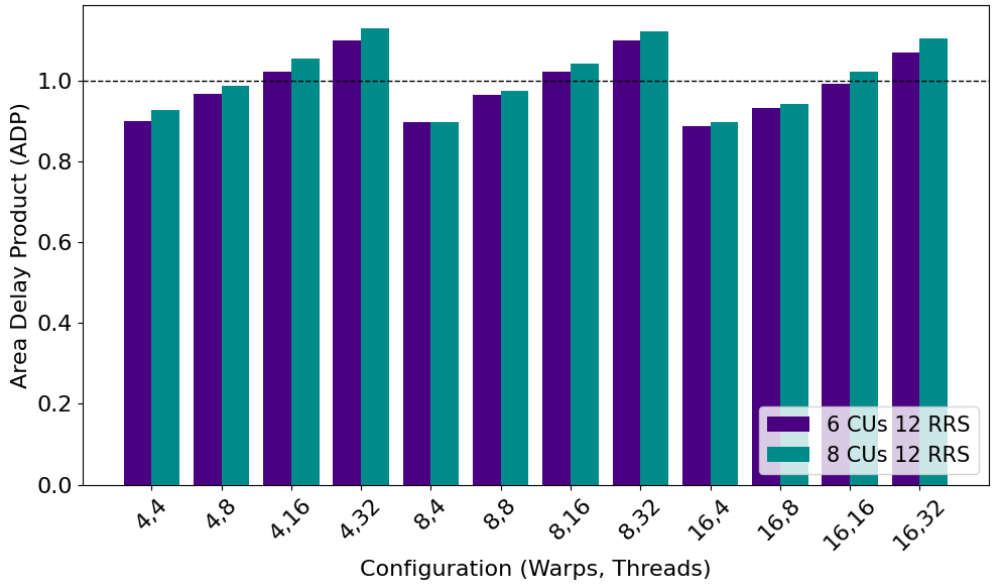
Σχήμα 1.18: Γεωμετρική μέση τιμή IPC (21 εφαρμογές) ανά διαμόρφωση για διάφορους αριθμούς CUs στην αρχιτεκτονική LOOG-Vortex με RRS (παράγοντας 2).

Στα Σχήματα 1.17 και 1.18 φαίνονται αναλυτικά τα κέρδη IPC σε σχέση με την αρχική αρχιτεκτονική της Vortex, για όλες τις διαμορφώσεις warps, νημάτων και

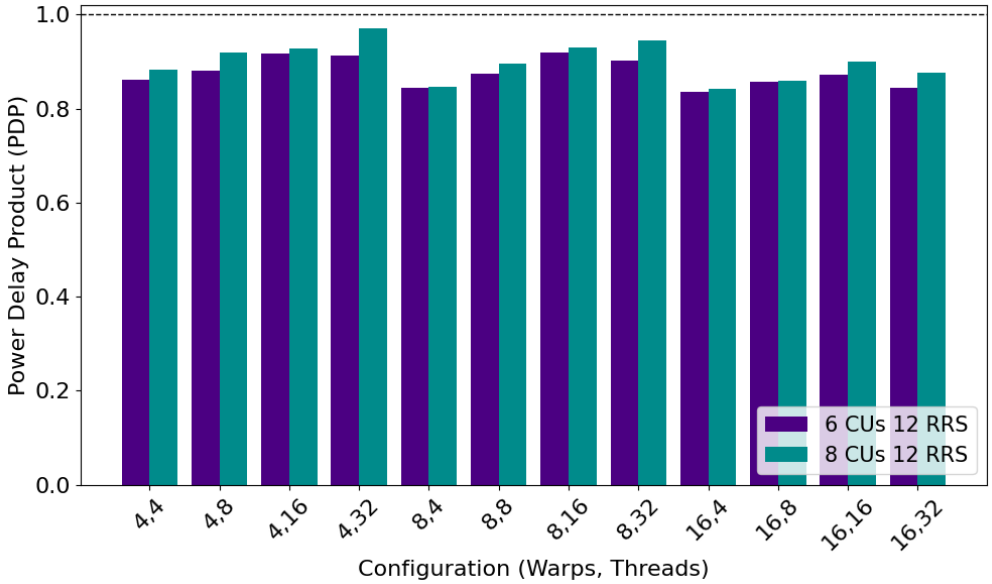
διαφορετικών αριθμών CUs για παράγοντες RRS 1.5 και 2, αντίστοιχα. Όπως αναμενόταν, ιδιαίτερα για μικρότερους αριθμούς CUs, η βελτίωση Απόδοσης είναι μεγαλύτερη απ΄ ό,τι στην αρχιτεκτονική χωρίς το RRS του Σχήματος 1.14.

Επιπλέον, αυτά τα αποτελέσματα υποδεικνύουν ότι, σε σύγκριση με την προσθήκη περισσότερων CUs, η επέκταση του RRS προσφέρει καλύτερη Απόδοση. Ένα ενδιαφέρον στοιχείο προκύπτει όταν συγκρίνουμε διαφορετικούς παράγοντες RRS: για παράδειγμα, μια διαμόρφωση με 8 CUs και παράγοντα RRS 1.5 (δηλαδή 12 θέσεις στο RRS) υπερτερεί έναντι της διαμόρφωσης με 6 CUs και παράγοντα 2 (επίσης 12 θέσεις RRS), παρά το γεγονός ότι η τελευταία έχει λιγότερες δομές. Αυτή η ασυνέπεια μπορεί να αποδοθεί σε διαφορές στην αναδιάταξη των εκτελέσεων, οι οποίες επηρεάζονται από τα διάφορα επίπεδα συμφόρησης και καθυστερήσεων στις λειτουργίες των CUs.

Σχετικά με τη χρήση πόρων στο FPGA και την Κατανάλωση Ενέργειας, υπολογίζουμε και πάλι τους δείκτες γινομένων Επιφάνειας-Καθυστέρησης και Ισχύος-Καθυστέρησης, αυτή τη φορά για την αρχιτεκτονική LOOG-Vortex με RRS, όπως φαίνεται στα Σχήματα 1.19 και 1.20. Τα αποτελέσματα του ADP δείχνουν ότι περισσότερες από τις μισές των διαμορφώσεων που δοκιμάστηκαν υπερτερούν έναντι της αρχικής αρχιτεκτονικής, με τις καλύτερες να αντιστοιχούν σε χαμηλότερο αριθμό νημάτων ανά warp, όπως αναμένονταν από την έως τώρα ανάλυση. Όσον αφορά το PDP, όλες οι διαμορφώσεις παρουσιάζουν υποσχόμενα αποτελέσματα, με μείωση της Καταναλισκόμενης Ενέργειας που φτάνει περίπου το 17%.



Σχήμα 1.19: Γινόμενο Επιφάνειας-Καθυστέρησης του LOOG-Vortex χωρίς RRS ανά διαμόρφωση για 6 και 8 CUs (κανονικοποιημένο στην αρχική αρχιτεκτονική).



Σχήμα 1.20: Γινόμενο Ισχύος-Καθυστέρησης του LOOG-Vortex χωρίς RRS ανά διαμόρφωση για 6 και 8 CUs (κανονικοποιημένο στην αρχική αρχιτεκτονική).

1.5 Συμπεράσματα και Μελλοντικές Επεκτάσεις

Η εφαρμογή του σχήματος εκτέλεσης LOOG (Light-weight Out-Of-Order GPU) στην αρχιτεκτονική της Vortex GPU , τόσο με όσο και χωρίς τη Στοίβα Μετονομασίας Καταχωρητών (RRS) , απέδειξε σημαντικές βελτιώσεις στην Απόδοση με χαμηλό κόστος Επιφάνειας και Κατανάλωσης Ενέργειας σε σύγκριση με την αρχική αρχιτεκτονική in-order του Vortex. Το σχήμα LOOG υλοποιήθηκε με χρήση της SystemVerilog για την έκδοση 2.0 της Vortex GPU, που είναι βασισμένη σε αρχιτεκτονική RISC-V.

Η πειραματική αξιολόγηση των διαφόρων διαμορφώσεων του αρχικού, του LOOG και του RRS-LOOG Vortex έδειξε σημαντικές βελτιώσεις στην Απόδοση (έως και 23,5%), με το κόστος Επιφάνειας να κυμαίνεται μεταξύ 4,5-27,5% (περίπου 5% στις ταχύτερες διαμορφώσεις) και μείωση Κατανάλωσης Ενέργειας μέχρι 17% για την RRS-LOOG προσέγγιση.

Αν και η τρέχουσα εργασία προσφέρει πολύτιμες πληροφορίες για την βελτιστοποίηση της εκτέλεσης LOOG στη Vortex GPU, υπάρχουν αρκετοί τομείς για περαιτέρω εξερεύνηση. Ένας τέτοιος τομέας είναι η ενσωμάτωση μεγαλύτερων συνόλων εφαρμογών OpenCL, όπως το Rodinia, προκειμένου να εξεταστεί πώς το LOOG-Vortex χειρίζεται πιο περίπλοκες εφαρμογές. Επιπλέον, η αξιολόγηση της σχεδίασης LOOG-Vortex σε μεγαλύτερες πλατφόρμες FPGA θα επιτρέψει την εκτίμηση της απόδοσης του συστήματος καθώς κλιμακώνεται σε πιο σύνθετες διαμορφώσεις.

Επιπλέον, υπάρχουν ευκαιρίες για εξερεύνηση πιο εξελιγμένων πολιτικών προγραμματισμού για την ανάγνωση του Αρχείου Καταχωρητών από τα CUs και τον

χρονοπρογραμματισμό της εκτέλεσης των εντολών των warps. Ένας άλλος τομέας βελτίωσης είναι η υλοποίηση ενός Αρχείου Καταχωρητών πολλαπλών εισόδων, για να μειωθούν οι καθυστερήσεις στο στάδιο της Συλλογής Ορισμάτων και να βελτιωθεί η συνολική απόδοση.

Τέλος, η υλοποίηση και σύγκριση διαφορετικών σχημάτων εκτέλεσης Εκτός Σειράς για GPUs που προτείνονται από την ερευνητική κοινότητα θα προσφέρει πολύτιμες γνώσεις για την αναβάθμιση του Παραλληλισμού σε Επίπεδο Εντολών στα μελλοντικά σχέδια των GPUs, προσφέροντας ιδέες για τη βελτίωση της ταχύτητας και της ενεργειακής απόδοσης.

Chapter 2

Introduction

2.1 Hardware Accelerators Today

2.1.1 The end of the Scaling Laws

Gordon E. Moore in 1965 observed a very significant trend: the density of components on integrated circuits, more specifically transistors, doubled approximately every two years [1]. Not only did this observation hold true, but it propelled technological innovation for decades until about twenty years ago. During this period, the semiconductor industry benefited from Dennard's scaling, another underlying scaling law, which contended that chip power density would remain constant as transistors were scaled down in size [2]. This enabled the creation of more power-friendly, faster, and power-saving transistors, which justified the astronomical expenses of bringing up new process nodes.

But supply voltage limitations and increased fabrication complexities at the atomic level soon pushed power density on chips upwards, ushering in the age of "Dark Silicon" [3]. This necessitates a large proportion of transistors to be turned off, operated at reduced frequencies, or reconfigured into more power-friendly clusters at runtime.

To overcome these challenges, particularly in continuously computing-capacity-dependent fields like Artificial Intelligence and Data Analytics, software optimization and the creation of highly advanced microarchitectures are crucial. These architectures must effectively integrate software diversity with hardware heterogeneity [4]. In this context, GPUs have become essential hardware accelerators, complementing multi-core CPUs to meet the intensive computational demands of modern data analytics workloads.

2.1.2 General-Purpose GPUs

As Central Processing Units (CPUs) have evolved through the years, they have become multicore processors capable of performing several tasks in parallel, as they can utilize sophisticated computational, scheduling and memory techniques to minimize latency.

Conversely, Graphics Processor Units (GPUs) are manycore processors, devoting thousands of threads on executing simple computations in parallel for massive amounts of data. Thus, GPUs serve as co-processing units to CPUs, optimally suited for tasks that demonstrate high regularity and arithmetic intensity [5].

While GPUs were initially designed for graphics rendering applications and gained popularity and development interest for enhancing gaming visuals, their capabilities have expanded significantly over time. Starting from the introduction of vertex shaders into NVIDIA Tesla GPU architecture [31], they soon started to efficiently handle complex linear algebra tasks [32]. Today, GPUs can be used to accelerate a wide range of data-parallel applications across various domains. These include Machine Learning, scientific simulations (such as weather modeling, physics and computational chemistry simulations), medical imaging and video editing. This way, GPUs evolved from being specialized devices for graphics rendering to becoming versatile, general-purpose machines capable of handling a diverse array of computing tasks.

2.1.3 Reconfigurable Hardware & FPGAs

In Heterogeneous computing platforms, alongside CPUs and the GPUs, additional accelerating hardware components are often present to enhance energy efficiency and improve performance when handling specific tasks. The current design of such hardware accelerators is based primarily on Application-specific Integrated Circuit (ASIC) and Field Programmable Gate Array (FPGA).

ASICs are custom-designed to serve a particular purpose, with high-performance, low power consumption and compact size but come with lengthy design cycles and high manufacturing costs. In contrast, FPGAs represent a class of reconfigurable devices that support both the flexibility of software and the performance of hardware. They attain extremely low design cost compared to ASIC accelerators, although they tend to increase the silicon area and involve complex programming that can extend R&D cycles [6].

Despite these challenges and due to their reconfigurability, FPGAs are favored over simulation-aided analysis for emulating novel computer architectures. Their inherently programmable fabric and operation at hardware speed allow for accurate and flexible emulation across various metrics such as cycle-accurate performance, area utilization, and power estimation. Fast and affordable hardware emulation and verification systems like these are essential to address the limitations of current methods [7].

2.2 Improving single-core Performance with Light-weight Out-Of-Order execution

GPUs leverage Thread-Level Parallelism (TLP) by scheduling and executing multiple threads (usually 32 or 64 in commercial GPUs) grouped into units called warps or wavefronts. Pairing that with the fast context switching between different warps' execution hides stalls related to memory access and utilizes more efficiently the Execution Units of the GPU, leading to highly-parallelized operation of tasks. This is not always the case for General-Purpose GPUs, where some functions offloaded to the GPU for computation (also called kernels) exploit limited TLP and are not capable of utilizing the GPU's units sufficiently. With Light-weight Out-Of-Order GPU (LOOG) execution, as proposed by Iliakis et al. [8, 9], kernels can also benefit from Instruction-Level Parallelism by executing warps' instructions Out-Of-Order (OOO).

Out-Of-Order execution is not new to CPUs, as it dates back to the 1960s with the development of the IBM System/360 Model 91 in 1964, which introduced Dynamic Instruction Scheduling to reorder instructions that stalled due to data dependencies [33] In 1967 Tomasulo's Algorithm was published, mitigating the delays caused by operand unavailability by leveraging Register Renaming and Reservation Stations [34]. Modern processors have continued to refine and expand upon such techniques, with OOO execution remaining a fundamental part of ILP exploitation in Computer Architecture [35].

The proposed LOOG execution scheme, on the other hand, targets GPU architectures, which traditionally issue instructions following the program order and track operand dependencies using Scoreboarding, thus avoiding RAW and WAW data hazards. In LOOG, the GPU's Collector Units (modules occupied by instructions until their source operands are fetched from the Register File) are repurposed to also be used as Tomasulo's Reservation Stations. Also, a Register Alias Table (RAT) is added to resolve name dependencies and a sophisticated Load-Store reordering scheme is leveraged for handling memory operations. By adding those components to the GPU pipeline, warp instructions can execute Out-Of-Order, benefitting from both ILP and TLP and leading to low-cost Performance gains.

2.3 GPGPU Emulation with Vortex-GPU

To explore and develop new microarchitectures and execution mechanisms in GPUs, certain simulation and emulation tools are needed to validate their potential before committing to expensive and time-consuming hardware fabrication. These offer a deeper understanding of how modifications to the microarchitecture, such as changes in the memory hierarchy or execution units, could impact the GPU's performance and efficiency.

For this purpose, a few simulators have been proposed, like Accel-Sim [10], an open-source GPU simulation framework for modelling NVIDIA GPU architectures, built around the performance model of GPGPU-Sim 4.1.0, a cycle-level GPU performance simulator [36], as well as frameworks suitable for GPU power modelling, like the Accel-Wattch tool [37]. When it comes to hardware emulation, the availability of such tools, which are also open-source, is more limited. However, the Vortex GPU framework stands out as it supports GPGPU emulation on FPGA platforms [11, 12, 30].

Vortex is an open-source, RISC-V based General-Purpose GPU (GPGPU) that supports programming with OpenCL and CUDA. It is designed to facilitate research in GPU architectures by providing a full-stack implementation that includes a minimal instruction set architecture (ISA) extension tailored for GPU operations. Vortex is designed using Verilog and SystemVerilog Hardware Description Languages, making it capable of running on various platforms such as FPGAs and simulators, and is aimed at enabling the execution of OpenCL / CUDA applications in a customizable and scalable environment. This project is particularly notable for extending the RISC-V ISA to support GPU functionalities, making it a valuable tool for developers and researchers interested in exploring GPU computing within the open-source RISC-V ecosystem.

2.4 Proposal Overview

The proposed LOOG execution scheme demonstrates promising performance improvements alongside attractive area and power overhead estimates. However, thus far, LOOG has only been modeled in simulation tools (GPGPU-Sim and Accel-Sim), where the estimations rely on simplistic, linearly scaling values as the GPU architecture is modified—resulting in discrepancies compared to an actual hardware implementation. Also, these tools do not provide an estimation of the architecture's Critical Path, which is crucial for determining the clock frequency of the GPU.

To address this gap, we propose implementing LOOG on real hardware by fully detailing its mechanism at the Register Transfer Level (RTL) using a Hardware Description Language (HDL) such as SystemVerilog. Although an RTL implementation of a GPU for FPGA emulation presents limitations in terms of size and direct comparability to commercial GPUs, it offers several advantages. These include faster execution

times, enhanced signal-level observability, and the reconfigurability of FPGAs, which broadens the design space by allowing exploration of various component sizes and design trade-offs. Crucially, this approach will yield more accurate assessments of power consumption and resource utilization, providing a reliable indicator of the final chip's area usage.

We have selected the Vortex GPU framework version 2.0 as our baseline architecture due to its open-source nature, RISC-V foundation, and parameterization options (threads per warp, warps per core, and component sizes), which further expand our design space. Our approach involves modifying the in-order Vortex pipeline by first incorporating an Operand Collect stage with dedicated Collector Units —a feature absent in the original design and in contrast to commercial GPUs— followed by the implementation of LOOG's Out-of-Order mechanisms, tailored to the unique characteristics of the Vortex architecture.

Subsequently, we will explore various optimizations and trade-offs regarding performance and FPGA-specific limitations, such as routing congestion and resource utilization overheads. For our experimental evaluation, we have characterized a workload comprising 21 applications, which we have grouped into five categories based on their stall sources when executed on the in-order Vortex architecture. This classification enables us to accurately define LOOG-sensitive workloads.

Finally, we will perform a LOOG-Vortex right-sizing analysis to identify the optimal design parameters and configurations within our extensive design space. Our evaluation will compare the achieved performance gains, area utilization, and power consumption overheads with those of the original in-order Vortex execution scheme. Preliminary findings indicate average performance improvements of up to 23.5%, area overheads ranging from 4.5% to 27.5% (with around 5% in our best configurations), and a reduction in the Power-Delay Product—equating to the device's energy consumption—of up to 17% compared to the in-order design.

2.5 Contributions

This thesis presents several significant contributions to the field of GPU architecture enhancements within the Vortex GPU framework version 2.0, specifically integrating the LOOG scheme. The main contributions include:

- **Hardware Implementation of Basic LOOG Scheme in Vortex GPU framework:**
  - Accommodation of the LOOG scheme in RISC-V based 6-stage GPU pipeline.
  - Provision of black-box reconfiguration capabilities for LOOG parameters within the Vortex framework.

  - Introduction of new microarchitectural features, including a lightweight per-warp UUID generation mechanism.
  - Implementation of LOOG's Register Renaming Stack (RRS).
  - Implementation of different schedulers for RF read operations.

- **Enhanced Traceability:** Provision of detailed, cycle-accurate traces for all added signals in RTL-simulated application execution.
- **New Performance Counters:** Introduction of performance counters for RTL simulation and FPGA emulation, aimed at collecting runtime statistics and metrics such as:
  - Register File read and write operations
  - Reordered instructions & reordering distances
  - LOOG-dependent stalls
  - Collector Units utilization & allocation period
  - Register Renaming Stack utilization & allocation period
- **Vortex workload characterization:** Clustering of Vortex benchmarking applications based on their stall sources and analysis of each cluster's behavior in LOOG-Vortex.
- **LOOG Right-Sizing for Vortex GPU Version 2.0:** This includes optimization and calibration of LOOG to efficiently integrate with and utilize the resources of the Vortex GPU version 2.0, in terms of:
  - Performance
  - FPGA Resource Utilization
  - FPGA Power Consumption

2.6 Thesis structure

The remainder of this thesis is structured as follows:

- Chapter 3 delves into the theoretical background of parallel computing, encompassing strategies for harnessing parallelism, fundamentals of General Purpose GPU architecture, and reconfigurable hardware. It also introduces the Vortex RTL framework and the LOOG scheme for out-of-order execution, which are central to this study.

- Chapter 4 explores existing research related to instruction-level parallelism in GPUs and reviews various RTL frameworks that facilitate GPGPU emulation, highlighting the advancements and identifying gaps in the current landscape.
- Chapter 5 details the integration of the LOOG scheme within the Vortex GPU, the structure of the added components and the complete execution scheme. It also explores the necessary adjustments and optimizations of the implementation and, lastly, characterizes the workload used in the experimental evaluation.
- Chapter 6 evaluates the performance of the LOOG-Vortex implementation through rigorous testing, scaling the LOOG-Vortex configuration to achieve the desired performance and analyzing the results across different scenarios.
- Chapter 7 wraps up the thesis by summarizing the main discoveries and contributions of the research. It also outlines potential future research directions and experiments to further this field of study.

Chapter 3

Background

3.1 Introduction

This chapter aims to provide a comprehensive understanding of the theoretical foundations necessary to grasp the intricacies of our proposed LOOG-Vortex architecture. Beginning with fundamental concepts, such as the structure of contemporary GPGPU architectures, the discussion will progress to more complex topics including the design of Register Transfer Level (RTL) for reconfigurable devices such as FPGAs. Furthermore, an in-depth explanation of the LOOG Out-Of-Order logic will be provided, underpinning the innovative aspects of our architecture. Additionally, this chapter will detail the operations of our GPGPU RTL framework, Vortex, which serves as our baseline. This thorough exploration prepares the reader to fully engage with the subsequent material presented in this thesis.

3.2 Parallel Computing and Parallelism Strategies

Parallel Computing: Parallel computing refers to the process of breaking down a problem of size n into k smaller parts (with k more or equal to 2), and solving these parts simultaneously using p physical processors. It involves dividing the workload to execute multiple tasks concurrently [13].

Types of Parallelism

Data Level Parallelism (DLP): DLP exploits the parallelism inherent in data structures, such as arrays or matrices, by performing the same operation on multiple data elements at the same time [14].

Task Level Parallelism (TLP): TLP involves breaking down a program into separate tasks that can be executed concurrently, where each task operates on a different portion of the problem. These tasks do not necessarily operate on the same data but are independent of each other. Each task is a self-contained unit of work that can be executed in parallel with other tasks, provided that dependencies between them are minimal or nonexistent [15].

Instruction Level Parallelism (ILP): ILP is the technique of executing multiple instructions in parallel within a single processor cycle. It exploits the fact that many instructions in a program can be executed independently of each other, and modern processors use techniques such as pipelining, out-of-order execution, and superscalar architecture to achieve this parallelism [16].

Pipelining: Pipelining is a form of parallelism where a computational process (e.g., an instruction) is segmented into subprocesses, each executed by dedicated, autonomous units. These subprocesses operate concurrently, much like an industrial assembly line, enabling efficient execution by overlapping successive instructions [17].

Out-of-Order Execution: Out-of-order execution is a processor design technique that allows instructions to be executed as resources become available, rather than strictly following their original program order. This approach enhances CPU performance by minimizing idle times and optimizing resource utilization [18].

Data Dependencies: An instruction j is data dependent on instruction i under the following conditions:

- Instruction i generates a result that instruction j utilizes.
- There is a chain of dependencies where instruction j depends on instruction k, and instruction k in turn depends on instruction i. This chain can extend across the entire program [19].

Data Hazards: A hazard arises when dependent instructions overlap during execution, potentially altering the intended operand access order. These hazards are categorized into three primary types:

- **Read-After-Write (RAW)**: This hazard occurs when an instruction i writes to a register followed by another instruction j that reads from the same register. It reflects a true data dependence, representing a direct flow of data between instructions.
- **Write-After-Write (WAW)**: This happens when both instruction i and instruction j write to the same register. The correct execution of the program requires that the last write, by instruction j, determines the final value in the register. This type of hazard is known as output dependence.
- **Write-After-Read (WAR)**: This arises when an instruction i reads from a register, and a subsequent instruction j writes to the same register. To ensure data integrity, the read operation by i must occur before the write operation by j. This hazard corresponds to an antidependence or name dependence [19].

SIMD (Single Instruction Stream, Multiple Data Streams): The same instruction is executed by multiple processors using different data streams. SIMD computers exploit data-level parallelism by applying the same operations to multiple items of data in parallel [19].

3.3 Fundamentals of GPUs

GPGPU (General-Purpose GPU): The key characteristic of GPGPUs is their ability to handle general-purpose computations using the same hardware designed for graphics rendering. This is achieved by leveraging programming models such as CUDA or OpenCL, which allow developers to write programs that run on the GPU, taking full advantage of its massive parallel processing power [20].

Kernels: Kernels represent the segments of code designated to run on a GPU, initiated by the CPU through a driver. The CPU specifies the kernel, the number of threads, and the input data for the computation. This information is conveyed to the GPU via the driver, which signals the GPU to start executing the computations [21].

Threads: GPU threads refer to execution units that handle individual tasks. In GPUs, each threads fetches the same instruction concurrently to execute on different data, being more precisely defined as Single-Instruction Multiple Threads (SIMT), unlike SIMD vector processors, which fetch one instruction for a whole vector of data. [22]

Warps: Warps are defined as groups of threads within a GPU that execute the same instruction simultaneously. This grouping aligns with the GPU's Single Instruction, Multiple Threads (SIMT) execution model, where each warp typically consists of 32 threads [23]

Streaming Multiprocessors (SMs): In GPU architecture, a Streaming Multiprocessor (SM) is a fundamental unit responsible for executing parallel operations. Each SM contains multiple CUDA cores (processing units) and other components such as special function units and load/store units. These cores within an SM execute threads in parallel, enabling efficient processing of large-scale computations. The number of SMs in a GPU varies depending on the specific model and architecture, contributing to the overall processing power of the GPU. [24].

Collector Units: Collector Units are structures inside the GPU pipeline, specifically in the register read stage. Each instruction is assigned a Collector Unit, enabling concurrent reading of source operands for different instructions and enhancing throughput. Each collector unit is equipped with sufficient buffering space to store all the source operands needed to execute its associated instruction [21].

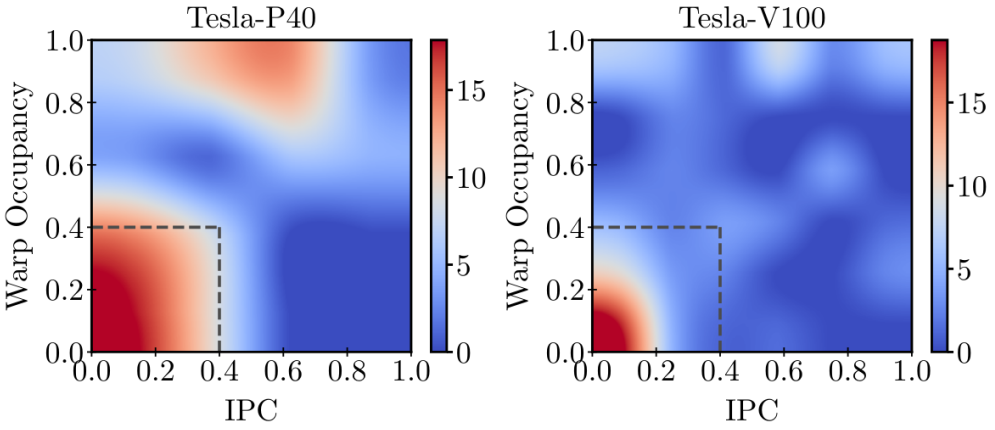


FIGURE 3.1: Warp occupancy - IPC distribution for 115 general- purpose kernels in two physical GPU platforms [28].

3.4 Reconfigurable Hardware and Emulation Techniques

Reconfigurable Hardware: Reconfigurable architectures combine a processor with a reconfigurable block, providing the adaptability needed by hardware for embedded applications. This setup offers both the performance of standard processors and the flexibility of Application Specific Integrated Circuits (ASICs) [25].

Field Programmable Gate Arrays (FPGAs): FPGAs are reconfigurable hardware platforms containing programmable logic blocks and routing switches. The logic blocks consist of transistors, gates, multiplexers, look-up tables (LUTs) and registers and they are responsible for implementing logic functions. The routing switches are used to connect the input and output pins of the logic blocks [25].

RTL Design: Designing an Integrated Circuit (IC) at the Register Transfer Level (RTL) is an more abstract than a Netlist description and less abstract that a Behavioral description of the circuit. It is implemented using an HDL like Verilog or VHDL and described using registers (flip-flops, latches), logic operators between them and wires for interconnection [26].

Hardware Emulation: In hardware emulation, a reconfigurable hardware platform (like an FPGA) is used to simulate the functionality of a system for debugging and verification purposes. Using Hardware Description Languages (HDLs) like Verilog to implement the system's logic, it allows for flexible design testing before the actual hardware is developed, but comes at the cost of routing limitations visibility, as well as long mapping times [27].

3.5 LOOG Details

The motivation behind the Light-weight Out-Of-Order GPU (LOOG) execution scheme [8, 9, 28] lies in the observation that some kernels executed on GPGPUs achieve low utilization, not being able to fully leverage the TLP provided by the hardware. Figure 3.1 shows that a large amount of kernels running on two GPU platforms exhibit low warp occupancy—resulting in restricted thread-level parallelism—and low IPC, which ultimately hampers execution efficiency. LOOG tackles these issues by harnessing Instruction Level Parallelism through Out-Of-Order execution, employing a light-weight version of Tomasulo's Algorithm, adapted for GPU pipelines. The LOOG pipeline is illustrated in Figure 3.2.

LOOG enables instruction reordering by repurposing the GPU's Collector Units (CUs) to double as Reservation Stations. In this configuration, instructions are held until their dependencies are met. Instead of relying on a traditional Scoreboard to track Read-After-Write (RAW) hazards, LOOG replaces it with a Register Alias Table (RAT). This unit not only monitors RAW dependencies, but also removes Write-After-Read (WAR) and Write-After-Write (WAW) hazards by register renaming—substituting the register name with the corresponding CU ID. When an instruction is assigned a CU, it first accesses the RAT for each of its source operands. Indexed by operand and warp IDs, each RAT entry specifies the source of the register's value, which can either be another CU's ID or the RF. If a register has already been renamed, its entry contains the ID of the CU responsible for producing its value. This ID is then copied into the allocated CU, and the result broadcast bus is monitored to capture the value once the matching CU ID appears. Simultaneously, the RAT is updated so that the destination register for the instruction now points to the newly allocated CU ID.

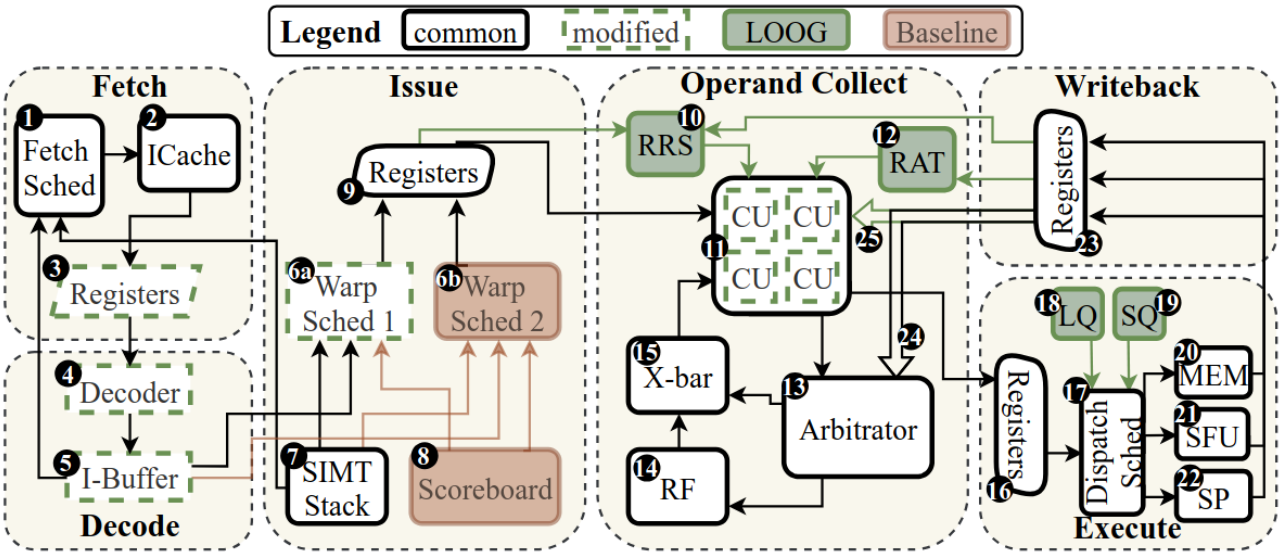


FIGURE 3.2: The LOOG modifications on top of the baseline GPU architecture pipeline [28].

Additionally, LOOG introduces a tailored mechanism to handle the reordering of Load-Store instructions, which may involve address dependencies. Memory instructions reserve entries in either a Load Queue (LQ) or a Store Queue (SQ) to track these dependencies. Before reordering any memory operation, the address operand is checked against those of all earlier issued store instructions. If a matching address is found, the operation is held back from reordering. Moreover, any store that has not yet resolved its target address will block subsequent memory operations, preventing potential address conflicts. Finally, to adhere to the GPU memory consistency model, no memory instructions are reordered ahead of memory barrier operations.

To further optimize resource utilization, LOOG employs a Register Renaming Stack (RRS). This structure maintains a list of unique IDs that are used in the RAT instead of the actual CU IDs. When an instruction is dispatched, it is allocated a free CU and draws a unique ID from the RRS. This ID is written into the RAT in place of the CU ID, and any dependent instructions use it to capture results from the result broadcast bus. This approach permits the instruction to free up its CU immediately after dispatch, retaining only the RRS ID until the writeback stage is completed.

3.6 Vortex GPU Pipeline

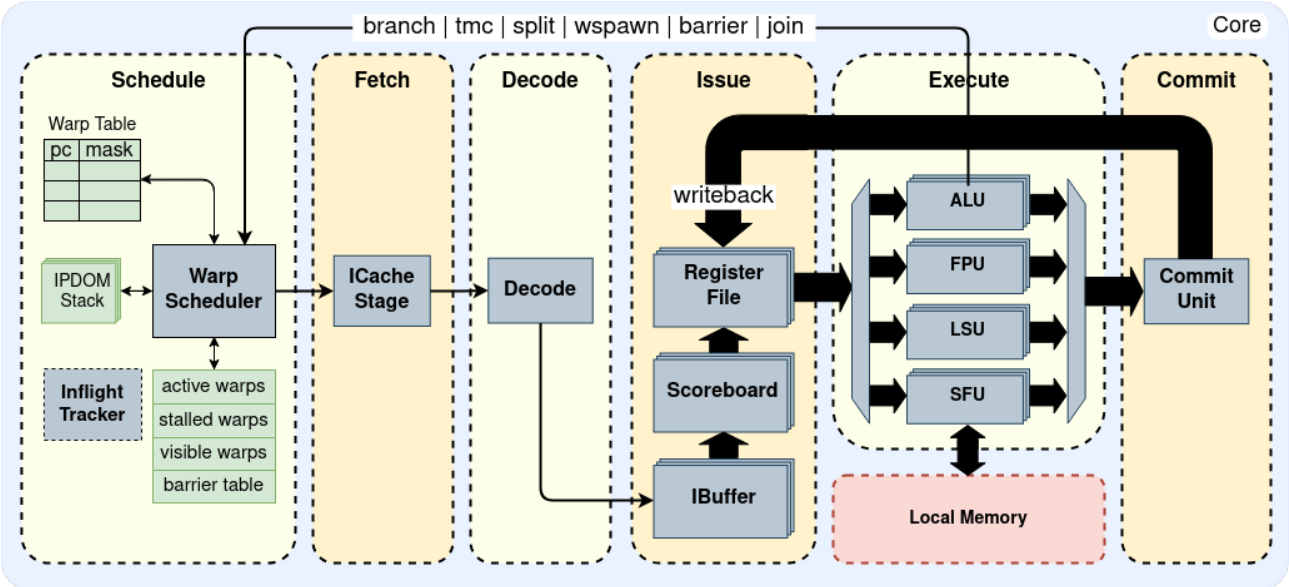


FIGURE 3.3: Pipeline microarchitecture overview of Vortex GPU [29].

The Vortex GPU pipeline [11] [12] [30] is built on a robust 6-stage design that integrates traditional RISC-V elements with specialized hardware for SIMT processing. It starts with the Schedule stage, where a warp scheduler determines the next program counter while tracking active and stalled warps. An IPDOM stack is employed to save split/join states for divergent threads, and an inflight tracker monitors all active instructions.

This careful scheduling ensures that multiple warps are effectively managed and that the pipeline maintains a steady stream of work.

Following scheduling, the Fetch stage retrieves instructions from memory while handling instruction cache requests and responses, keeping the pipeline adequately fed. In the Decode stage, these fetched instructions are translated into operations, with control instructions prompting notifications to the warp scheduler. This approach facilitates rapid adaptation to control flow changes and thread divergence, allowing for efficient parallel execution.

In the Issue stage, decoded instructions are stored in per-warp queues within an instruction buffer. A Scoreboard then tracks register usage and checks operand availability, while an Operands Collector fetches the required data from the register file. These steps prepare instructions for the Execute stage, where specialized units—such as the ALU for arithmetic and branch operations, the FPU for floating-point computations, the LSU for load/store operations, and the SFU for warp control and CSR tasks—carry out the core processing tasks.

Finally, the Commit stage writes the results back to the register file and updates the Scoreboard, marking the completion of instruction execution. Complementing the pipeline, Vortex also features a clustering architecture: cores are grouped into sockets that share an L1 cache, and these sockets are further clustered to share an L2 cache. This hierarchical design promotes scalability and efficient data sharing, making the Vortex GPU pipeline a versatile platform for both high-performance computing and graphics processing tasks.

Chapter 4

Related Work

4.1 MIAOW: RTL Implementation of a GPGPU

The MIAOW (Many-core Integrated Accelerator Of Wisconsin) is an open-source Register Transfer Level (RTL) implementation of a general-purpose GPU (GPGPU) [38] [39]. Designed for low-level hardware explorations, MIAOW enables detailed analysis of GPGPU microarchitecture and allows researchers to evaluate GPU designs from a physical design perspective. The architecture closely follows the AMD Southern Islands GPGPU Instruction Set Architecture (ISA), integrating with a comprehensive pipeline to execute OpenCL-based applications.

MIAOW's primary contributions include its realistic and flexible design, which supports experiments in GPU hardware research by providing insights into the area, power, and performance trade-offs involved in GPU design. It is not merely a simulator but an actual RTL model capable of running unmodified applications. The system is designed to emulate a full GPU, providing useful benchmarks for those researching GPGPU systems. It is particularly valuable for exploring architectural innovations and validating designs at the RTL level, a stage where simulator-based approaches may fall short in terms of hardware accuracy. Additionally, MIAOW facilitates research on novel GPU features, such as memory management and thread scheduling, contributing to advancing the field of GPGPU hardware design.

4.2 GhOST: OOO Scheduling for GPUs

The GhOST (GPU Out-Of-Order Scheduling Technique) paper [40] introduces a highly efficient and low-overhead Out-Of-Order (OOO) execution technique designed to reduce instruction stalls in GPUs. GhOST operates by leveraging the decode stage, which already stores a pool of decoded instructions, to enable Out-Of-Order instruction issue without the need for expensive hardware components. This innovative approach allows independent instructions from different warps to be scheduled and executed out of order, minimizing idle times caused by data hazards, while maintaining a simple and efficient hardware design.

One of the key advantages of GhOST is its minimal hardware overhead. By avoiding complex components such as register renaming, load-store queues, and speculative execution—techniques commonly used in prior OOO designs like LOOG—GhOST reduces the need for expensive hardware modifications, making it highly suitable for practical GPU implementations. GhOST's design relies on a straightforward modification to the instruction scheduling process, without altering the warp scheduler itself. As a result, it provides significant improvements in performance with a minimal increase in area. GhOST is particularly effective in scenarios with low warp occupancy, where traditional thread-level parallelism (TLP) optimizations are less impactful, thus enabling more efficient execution in a wider range of workloads.

4.3 SIMIL: Simple Issue Logic for GPUs

SIMIL (SIMple Issue Logic for GPUs) [41] introduces a novel architectural enhancement aimed at optimizing the instruction issue stage in General-Purpose GPUs (GPGPUs). It replaces the traditional scoreboard-based mechanism with a Dependence Matrix to track dependencies between instructions, enabling more efficient handling of data hazards. This is particularly beneficial for workloads where operands are reused frequently, such as in machine learning applications. By eliminating the need for complex components like scoreboards, SIMIL simplifies the hardware and enhances performance, energy efficiency, and area utilization.

Moreover, SIMIL extends the traditional in-order execution with a restricted Out-Of-Order execution (OOO) capability. This OOO mechanism allows instructions to be issued as soon as their operands are ready, rather than strictly in program order, leading to faster execution. By avoiding speculative execution and complex register renaming, the OOO extension provides substantial performance improvements with minimal hardware overhead. SIMIL delivers up to a 2.39x speedup for specific machine learning applications, reduces energy consumption by 12.81%, and incurs only a 1.5% area overhead, making it a highly efficient solution for handling Instruction-Level Parallelism in GPUs.

4.4 TURBULENCE: OOO GPU Execution whth Distance-based ISA

TURBULENCE [42] introduces an innovative approach to Out-Of-Order (OOO) execution on GPUs with minimal hardware overhead. This architecture aims to exploit Instruction-Level Parallelism in GPU workloads, which are often highly data-parallel but also contain hidden Instruction-Level Parallelism that could be unlocked through OOO execution. TURBULENCE consists of a novel instruction set architecture (ISA)

that replaces traditional register-based operand references with a distance-based operand referencing system. This system eliminates false dependencies and allows for more flexible scheduling without the need for complex and power-hungry components such as register renaming, reorder buffers, or load-store queues, which are typically required for OOO execution.

The architecture includes a hybrid ISA that dynamically switches between distance-based operands and traditional register numbers when the reference distance is too large, optimizing both performance and instruction count. The microarchitecture implementing TURBULENCE uses a scheduler similar to that of Out-Of-Order CPUs but with low complexity, allowing for efficient instruction reordering within threads. This approach reduces latency and enhances throughput without introducing substantial hardware costs. The evaluation of TURBULENCE shows significant performance improvements—up to 17.6% faster execution—while maintaining energy efficiency, with a 6.1% reduction in energy consumption compared to a baseline GPU configuration. This work demonstrates that low-cost Out-Of-Order execution is a viable technique for enhancing GPU performance, especially for workloads that can benefit from Instruction-Level Parallelism.

Chapter 5

Implementation Details

5.1 Introduction

The purpose of this chapter is to give the reader a thorough understanding of the implementation and the architecture specific details of our proposed LOOG-Vortex Pipeline. The chapter is organized as follows: in Section 5.2 we present the new components added to the Vortex pipeline to support LOOG and in 5.3 we describe the workflow of an Instruction in the updated Pipeline. In Section 5.4, we consider some microarchitectural trade-offs that lead to critical design choices. Section **??**, finally, presents the exploration of different design configurations and the right-sizing of LOOG-Vortex parameters.

5.2 LOOG-Vortex additional Components

This section lists the components that have been added to the baseline Vortex design in order to effectively accommodate the LOOG instruction flow. An abstraction of the updated pipeline schematic is shown in Figure 5.1.

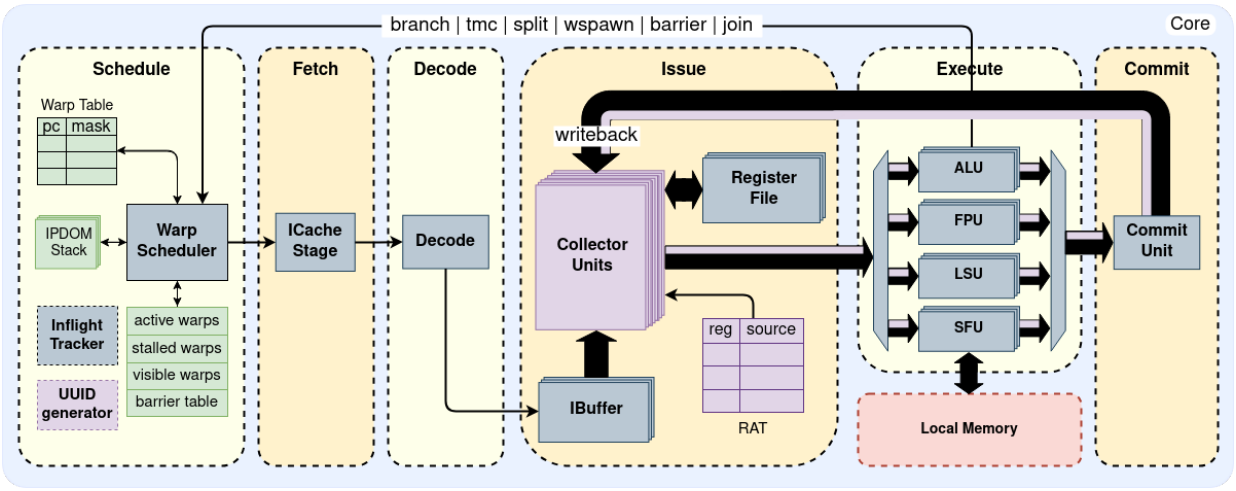


Figure 5.1: The LOOG-Vortex Modified Pipeline.

5.2.1 Collector Units

As described in Chapter 3, traditional GPU architectures incorporate Collector Units (CUs) responsible for holding the data of instructions until their source registers are retrieved from the Register File. In LOOG-Vortex GPGPU, we have implemented a configurable number of Collector Units, in order to explore the possible exploitation of instruction reordering and their reordering depths, a core component of architectures that support dynamic instruction scheduling (further analyzed in Chapter 6).

Each of the CUs is able to fit the data of an instruction exiting the IBuffer stage, containing the **warp's ID**, **PC address**, **operation type**, **source registers** and **destination register names**, **thread mask** etc., along with several fields. These include an **allocation status bit** to indicate whether the CU is "busy" or "empty", thereby determining its availability for new allocation. Another bit signifies whether or not the instruction has been **dispatched** for Execution. Further bits indicate the **"readiness" of the source registers** and the **source** from which they will be acquired. This latter field is critical in our LOOG design, where operands may be fetched either directly from the Register File or from another Collector Unit currently in the Writeback stage, in cases where there is a Read-After-Write (RAW) dependency. All the fields described above, along with their respective size in bits, are illustrated in Figure 5.2.

Collector Unit	
1	allocation status
1	dispatch status
1	rd data valid status
3	rs data valid status
3	rs to be read from RF
3 x (CU ID width)	rs source ID
(INSTR DATA width)	instr. UUID, warp ID, PC, tmask, regs etc.
T x (DATA width)	rd data
3 x T x (DATA width)	rs1 data, rs2 data, rs3 data

FIGURE 5.2: Collector Unit entry fields with their respective size in bits (first column).

Incontrovertibly, the role of Collector Units lies in obtaining the instruction's operands, necessitating dedicated fields for the **data of source registers**. On top of those, we also incorporated a field for the **data of the destination register**. This feature is crucial in the LOOG-Vortex configuration, where an instruction within a warp can execute multiple Writebacks across distinct and not necessarily sequential clock cycles, particularly in scenarios involving intra-warp thread divergence. To facilitate this, Vortex equips each instruction, alongside the Writeback data, with a **start-of-packet (sop)** and an **end-of-packet (eop) bit**, enabling each instruction to initiate up to the number of threads per warp Writeback operations. In the scenario of Out-Of-Order execution, managing these poses challenges when writing the packet back to the Register File or during broadcast. Our approach addresses these by retaining the data in a designated Collector Unit field until the end of the packet is reached, as indicated by an additional rd ready bit.

5.2.2 Register Alias Table

Rather than using a Scoreboard to monitor Read-After-Write (RAW) dependencies, LOOG employs a Register Alias Table (RAT). This mechanism not only tracks RAW dependencies but also eliminates Write-After-Read (WAR) and Write-After-Write (WAW) dependencies through register renaming, replacing the register name with the corresponding CU ID. The RAT functions as an array structure within the Vortex architecture, configured with individual entries for each register associated with a warp – that includes 32 integer and 32 floating-point registers in the Vortex default setup. Each entry within the RAT is composed of two key components: a bit that indicates whether the register's data needs to be sourced from the Register File and a CU ID field, as shown in Figure 5.3. This holds the identification number of the Collector Unit that houses the instruction slated to broadcast the result of the specified register. As instructions enter the Operand Collect stage, they have to first consult the RAT. This process determines the precise location of the operands' data, ensuring correct data retrieval before execution proceeds.

The discussed Register Alias Table functionality, supported also from the fields within the Collector Units, allows for Write-After-Write (WAW) dependent instructions to prevent unnecessary multiple writings to the Register File, particularly when two or more instructions from the same warp target the same register. In such scenarios, only the result of the latter operation is written back to the Register File, thereby optimizing resource usage and avoiding some structural hazards. Additionally, this configuration helps avoid certain reads from the Register File when dealing with Read-After-Write (RAW) dependencies, as they can be resolved by broadcasting data from one Collector Unit and made immediately available to other Collector Units, facilitating quicker data access and streamlining the execution process within the pipeline.

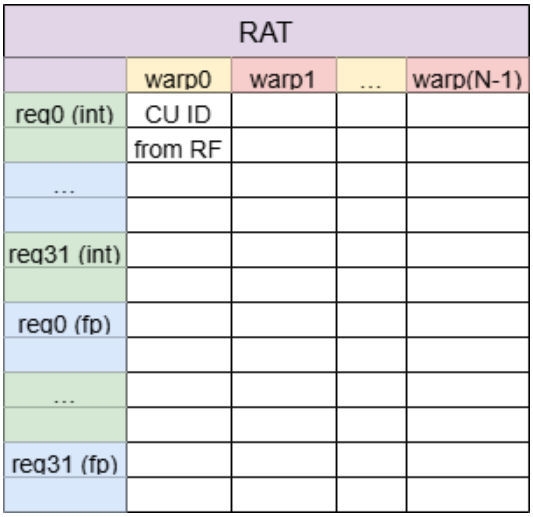


FIGURE 5.3: The Register Alias Table with the respective fields.

5.2.3 UUID Generation Unit

When executing a kernel Out-Of-Order, it is essential to ascertain the sequential precedence of instructions to effectively manage issues arising when deviating from 'program order'. Furthermore, we do not implement a reordering mechanism for memory operations here, so the information of an instruction preceding another is necessary. To facilitate this, we have implemented a UUID Unit responsible for generating an ascending Universal Unique Identification (UUID) number for each instruction that is scheduled to traverse through the pipeline. Also, UUID generation is conducted on a per-warp basis, since inter-warp dependencies are managed at a higher level using barriers, which are processed in the hardware during the Schedule stage. Once generated, the UUID is embedded within each instruction's data, accompanying it through the pipeline until arriving at the Commit stage, where it is finally discarded.

Although the availability of an infinite —or substantially large— pool of unique numbers for UUIDs would be simple and trouble-free, it would also lead to unnecessary area overhead in the hardware. To address this, we aim to minimize the number of bits allocated for this field and allow for the reuse of IDs after a certain number of instructions have been processed. As mentioned, we generate the UUID from a different pool for each warp and maintain further seperation of individual instructions by combining this with the warp ID. Furthermore, we employ a modified definition of the "less than ($<$)" operator, whereby a UUID with its most significant bits (MSBs) set to '00' is considered greater than (or in our context, subsequent to) a UUID with MSBs set to '11'. This adjustment allows us, if the UUID bit width is N, to obtain a margin of 2^{N-2} instructions that can be safely processed before potential serialization errors arise. Should this margin be exceeded, an overflow is detected, prompting a pipeline stall until it is cleared of all instructions from the affected warp.

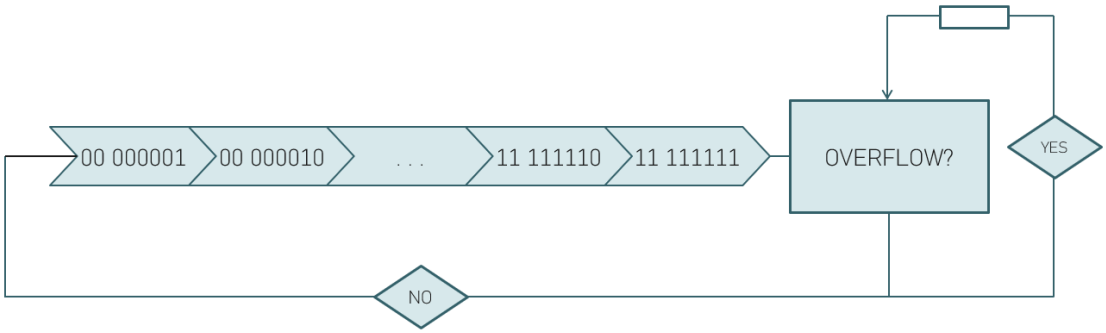


FIGURE 5.4: Light-weight UUID Overflow Mechanism.

5.3 LOOG-Vortex Instruction Flow

In this section, we describe assiduously the data flow of an instruction passing through our proposed LOOG-Vortex pipeline architecture. We will use figure 5.5 as the basis of our description.

5.3.1 CU Allocation & consulting the RAT

When an instruction transitions from the I-Buffer to the Operand Collect Stage, the initial step involves allocating an empty Collector Unit (CU). A CU is designated as ”unoccupied” when its allocation status bit is set to ’0’. Upon identification, such a CU is assigned to the instruction, which subsequently stores all pertinent data from the preceding stage into the CU.

In the next clock cycle, the newly allocated CU consults the Register Alias Table (RAT) to examine the entries corresponding to its source registers. It retrieves and copies the necessary source information into its respective fields. This process enables the CU to determine the appropriate source of each register’s data, discerning whether it should be read directly from the Register File or retrieved from another CU’s designated broadcasting. Concurrently, if the instruction requires a Writeback to a destination register, the CU writes its own ID within the register’s field in the RAT. This way, subsequent instructions that have a Read-After-Write (RAW) dependency upon the CU’s instruction will be alerted to stall until the result is broadcasted and the correct data can be aqcuired.

5.3.2 Reading from the Register File

Upon obtaining the necessary details from the Register Alias Table (RAT), the Collector Unit (CU) enters the subsequent cycle tasked with gathering the correct data for each of its source registers. Should any operands require direct retrieval from the Register File (RF), a ”reading” status is assigned to the CU, indicating active data collection.

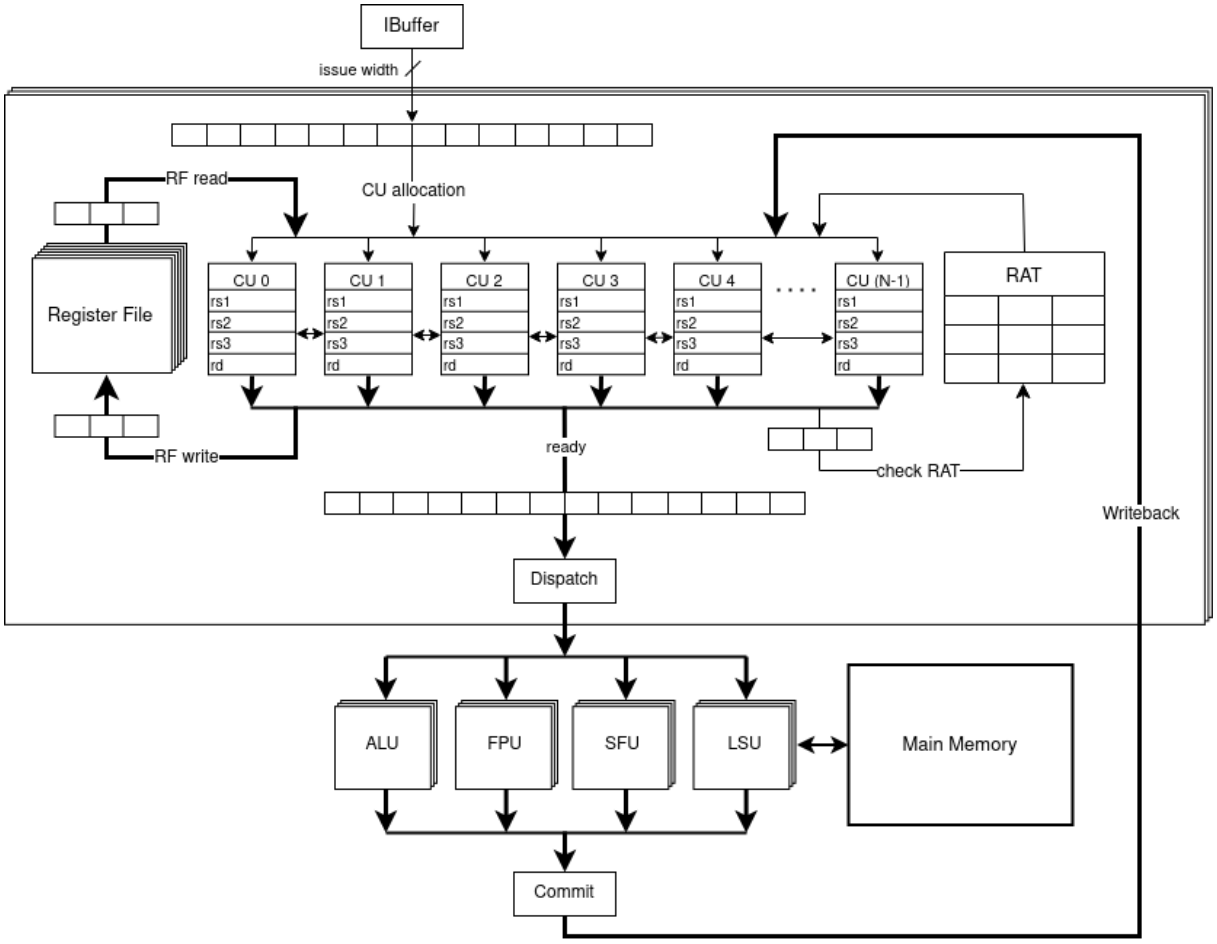


FIGURE 5.5: LOOG-Vortex Issue, Execute & Commit Stages.

The Vortex architecture limits the RF to single-port reads, necessitating a stall in the CU's operations if another instruction is simultaneously accessing the RF. The CU will proceed to fetch its operands sequentially from the RF, allocating one cycle per operand retrieval (this involves collecting the data for all active threads within the warp). Consequently, for an instruction that requires three source registers' data from the RF, the complete reading process will span three cycles. As data for each register is secured and stored within the CU, that register is then flagged as "valid", signifying the successful acquisition of data. Following the completion of all data collection from the RF, the CU transitions to a non-reading state, allowing the scheduling of another CU for operand collection, ensuring efficient management of access to the RF and coordination among multiple CUs.

5.3.3 Dispatching for Execution & CU Deallocation

Once a Collector Unit (CU) has acquired the data for all its source operands and each is marked as "valid," it is flagged as "ready." From this pool of ready CUs, one is selected

each cycle to advance to the Execution stage. Upon Dispatch, the instruction's fundamental data, combined with the operands' data, are packaged and forwarded to the next stage and a flag is set to indicate the CU as "dispatched."

Simultaneously, the instruction undergoes an evaluation to determine if it necessitates a Writeback (for instance, a Store memory operation does not require writing back a result to a register). If the instruction has no Writeback obligations, the CU is scheduled for Deallocation in the subsequent cycle. This process involves resetting the allocation and dispatch status flags to '0' and clearing any fields that indicate the "readiness" of operands and the CU overall. Conversely, if the instruction does require a Writeback, the CU retains its current state until this task is completed, after which it is promptly Deallocated.

5.3.4 Writing Back the Result

When an instruction completes its Execution and Commit stages , it may need to write back its result to its designated destination register. Upon Committing, the result is routed back to the Operand Collect stage where the Collector Units (CUs) and the Register File (RF) are located. In the Vortex architecture, which supports multiple Writebacks per instruction, results for threads within the same warp can arrive asynchronously. Writeback is only deemed complete upon receipt of a special end-of-packet (eop) signal; until then, the incoming data are temporarily stored in a dedicated field within the relevant CU.

Once the eop signal is detected, indicating that the destination register's data within the CU is finalized, the corresponding entry in the Register Alias Table (RAT) is reevaluated. If the CU discovers its own ID in this field, it is responsible for transferring the result back to the RF and updating the RAT; otherwise, this task falls to another CU. The write operation to the RF, if required, occurs in the following clock cycle, simultaneously with the update of data in all CU source register fields that are dependent on this instruction's broadcasted result. Subsequently, the CU has fulfilled all its responsibilities and is prepared for Deallocation, as previously described.

5.3.5 CU ID carried through Execution

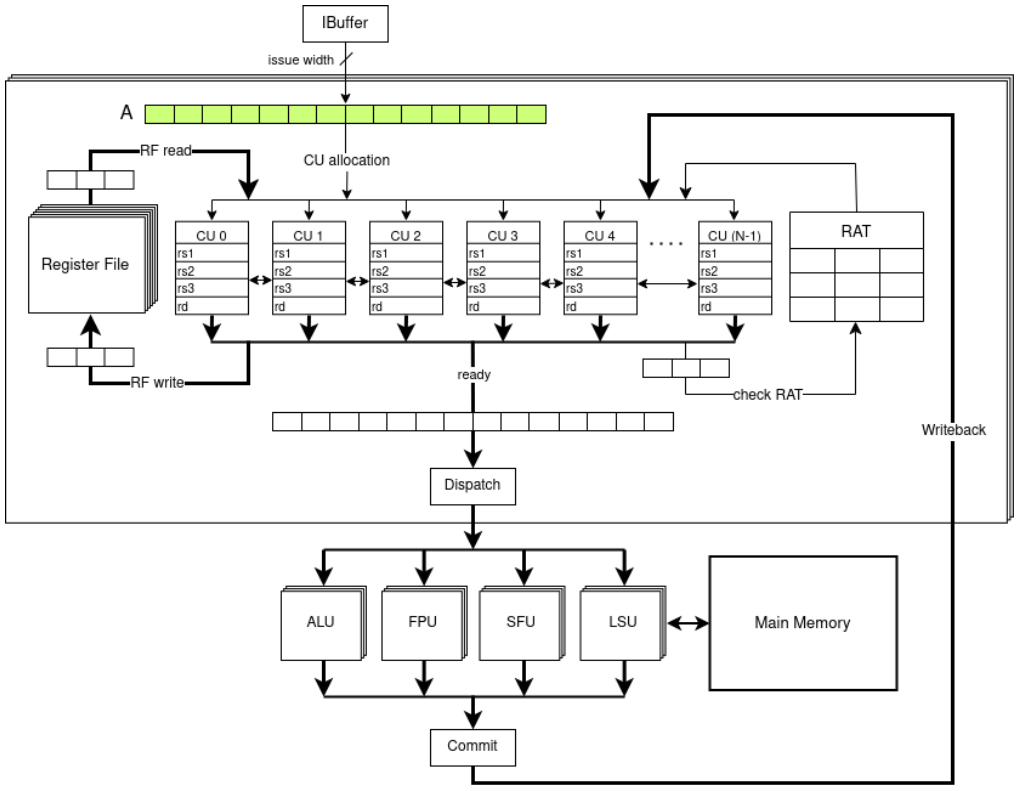
Instructions that are stalled in the Operand Collect stage due to RAW dependencies must ascertain from which Collector Unit to retrieve data when the valid result is broadcasted. For this purpose, each stalled CU retains the ID of the CU on which it is contingent. Hence, this CU ID must be carried from the Dispatch through the Execution and Commit stages for all instructions necessitating a Writeback. Accordingly, the Execution Units are equipped with a designated CU ID field, along with the interfaces that connect the inputs and outputs of each stage.

5.3.6 Execution Example

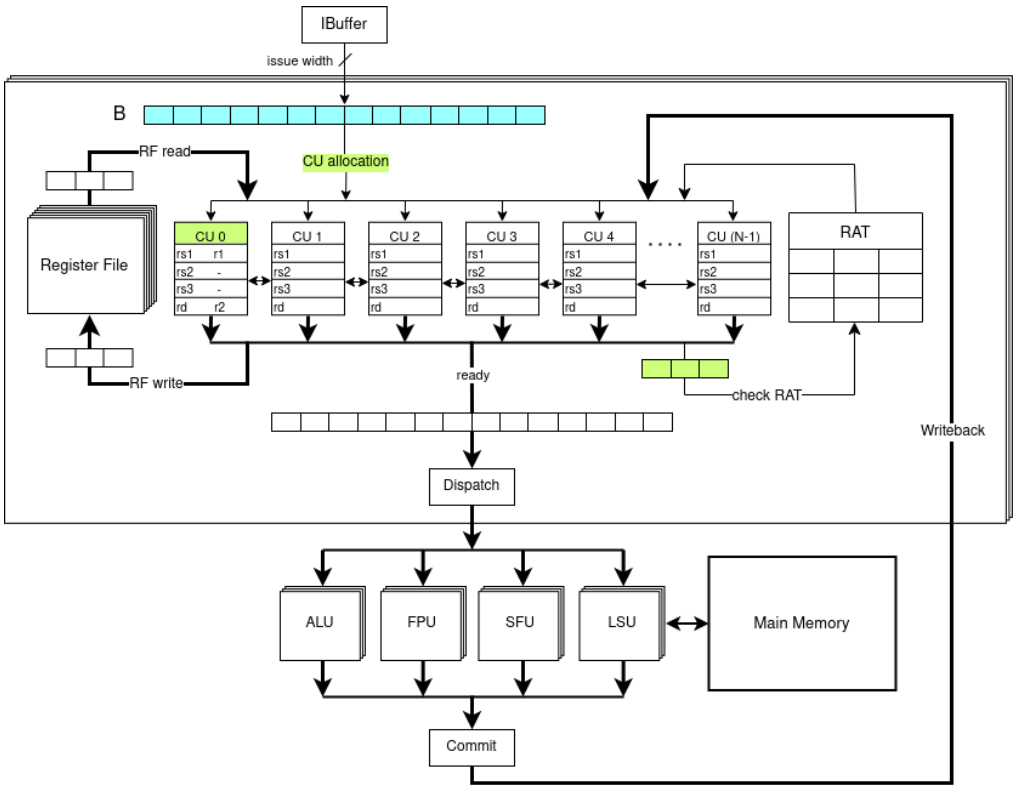
We will now present an example demonstrating the execution of three instructions within the LOOG-Vortex pipeline, focusing on their progression through the Issue, Execute, and Commit stages, particularly emphasizing the Operand Collect phase. This practical illustration ties into our earlier discussions and helps clarify the pipeline's operational details. In this example, instruction B haw a RAW dependency on A, because it has r2 as a source register. Instruction C is completely independent, since it comes from another warp, and it doesn't require a Writeback operation.
Consider the instructions:

A) `w0: addi r2, r1, 1`
B) `w0: add r2, r2, r3`
C) `w1: sw r1, 0(r2)`

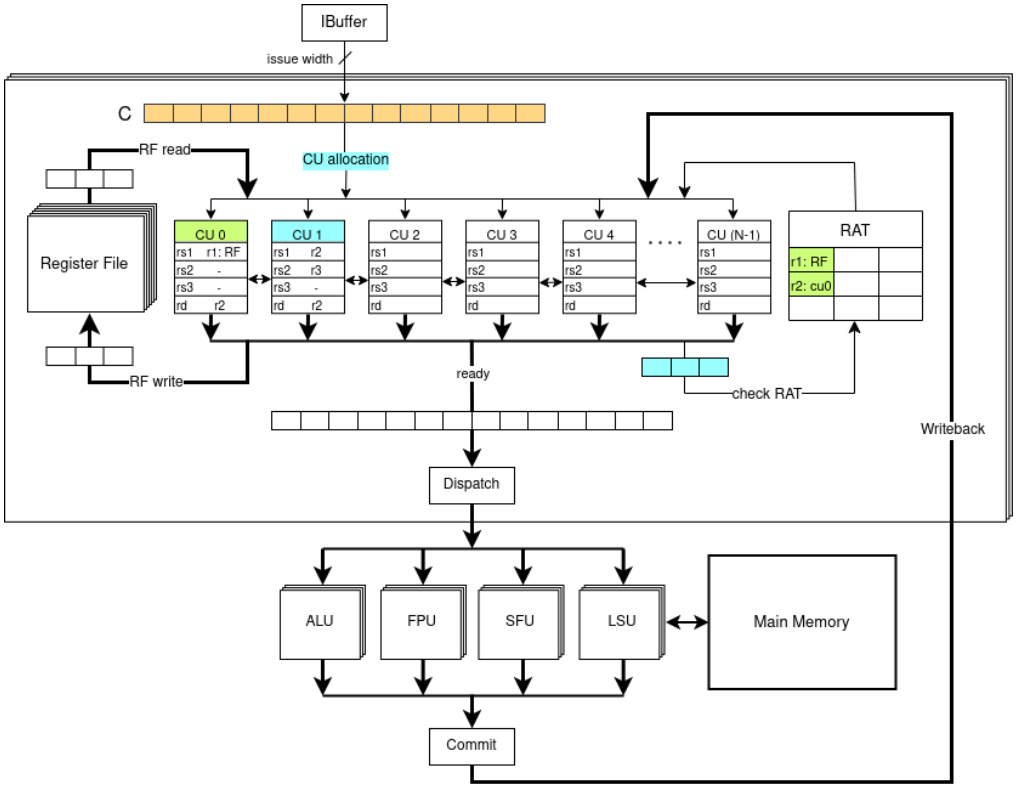
The following diagrams will illustrate step-by-step how these instructions are processed, highlighting the key tasks and interactions at each stage of the pipeline. Note that the cycle counts for the ALU and LSU Execution are not accurate (set at 2 and 3 cycles respectively) for simplicity and brevity in this example.



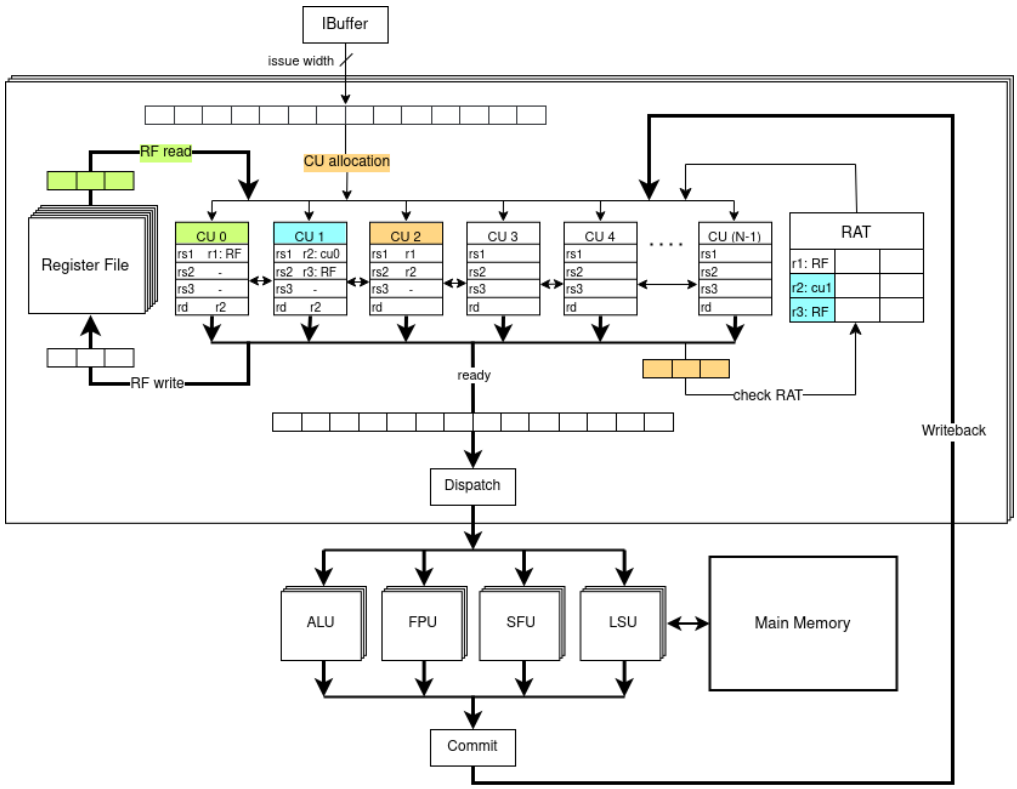
(A) **Clock Cycle 1:** Instruction (A) (green) is fetched from the IBuffer.



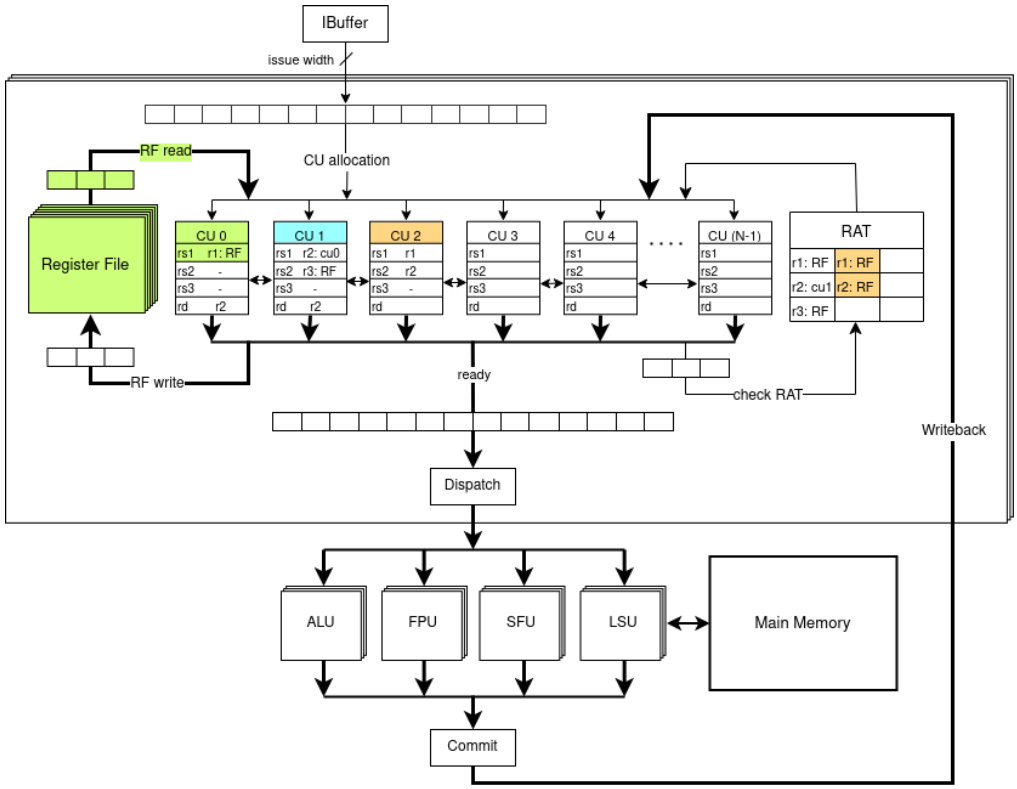
(B) **Clock Cycle 2:** Instruction (A) allocates CU 0 storing all its data to its corresponding fields and is scheduled to consult the RAT. Instruction (B) (blue) is fetched from the IBuffer.



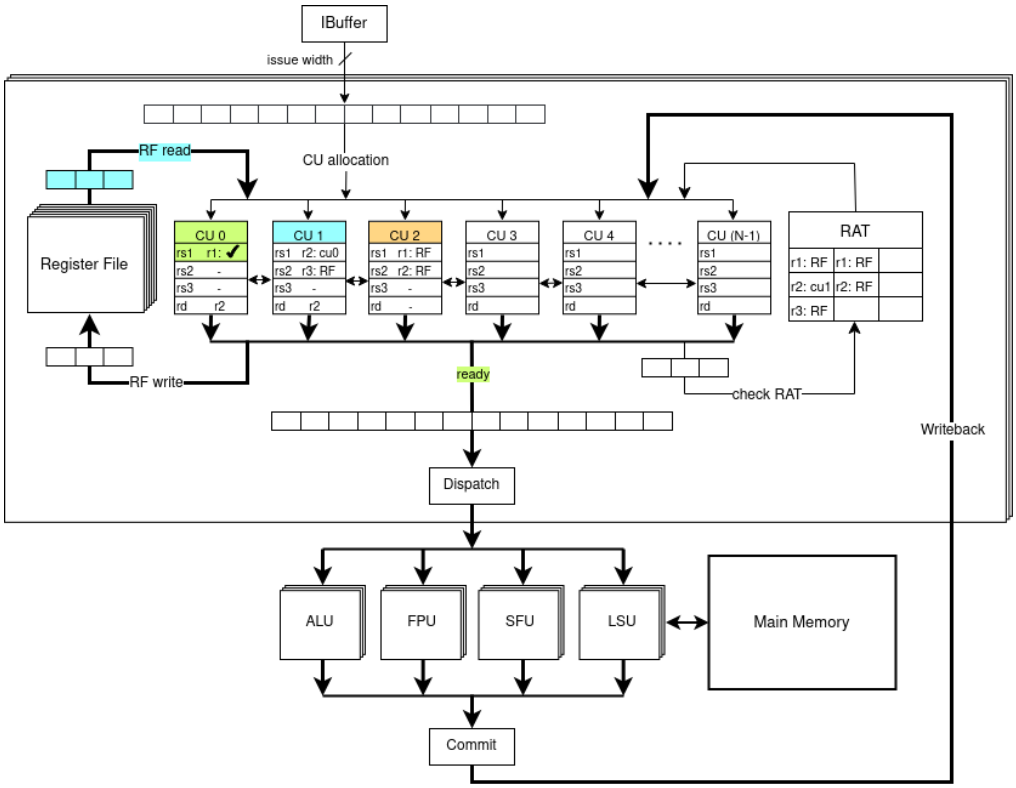
(C) **Clock Cycle 3:** Instruction (A) copies its source register's RAT entry to the corresponding CU field and writes its ID (CU 0) to its destination register's RAT entry. (B) allocates CU 1 and is scheduled to next read the RAT and (C) (orange) is fetched from the IBuffer.



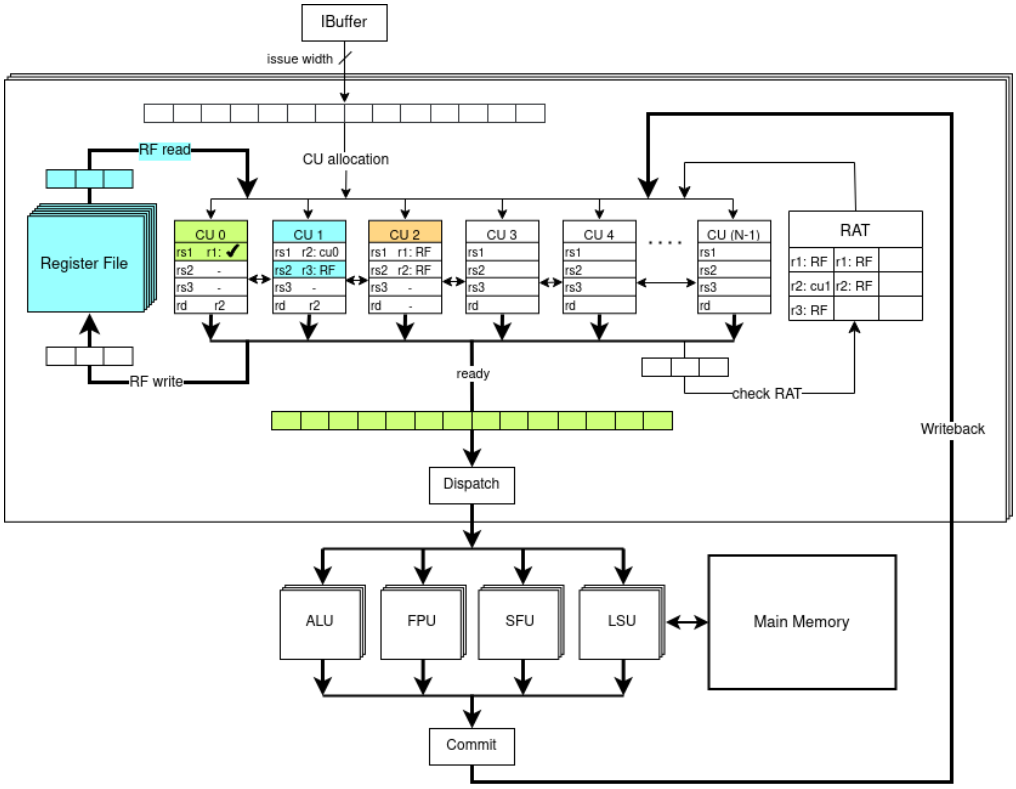
(D) **Clock Cycle 4:** CU 0 is set to ”reading” mode to fetch r1 from the Register File, proceeding as no other CUs are currently accessing the RF. Simultaneously, Instruction (B) updates its CU fields with RAT source data and adjusts the RAT entry for its destination register, r2.(C) allocates CU 2 and is scheduled to next consult the RAT.



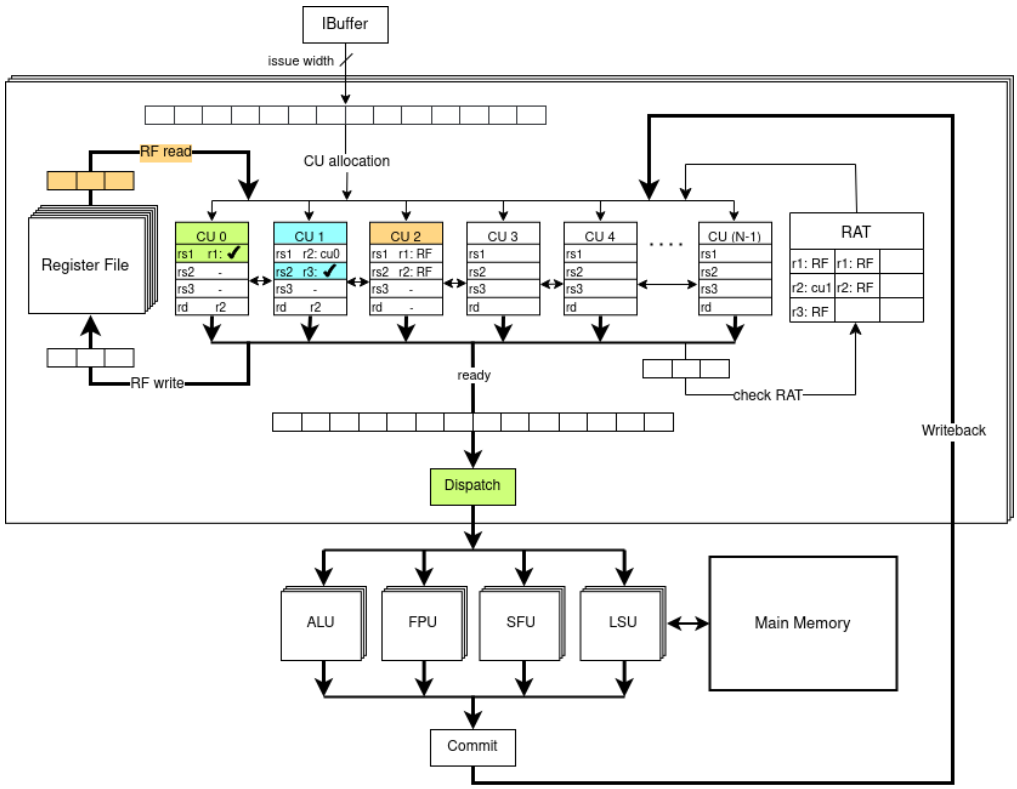
(E) **Clock Cycle 5:** CU 0 collects the data for r1 from the Register File. CU 1 is marked as ”reading”, as it has to read r3 from the RF. Instruction (C) consults the RAT and copies the relevant entries to its CU fields, making no modifications to the RAT since it requires no Writeback.



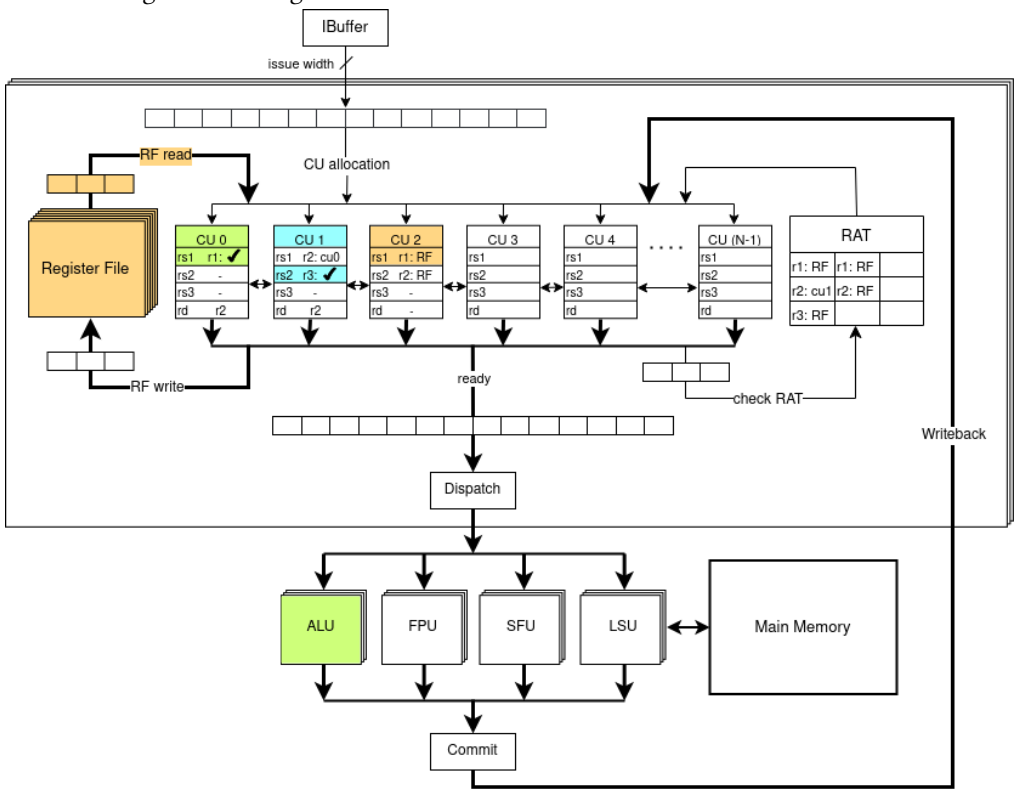
(F) **Clock Cycle 6:** CU 0 has finished accessing the RF, setting r1 as "valid" and marking itself as "ready". CU 1 is scheduled to read from the RF, while CU 2 awaits its turn to access it.



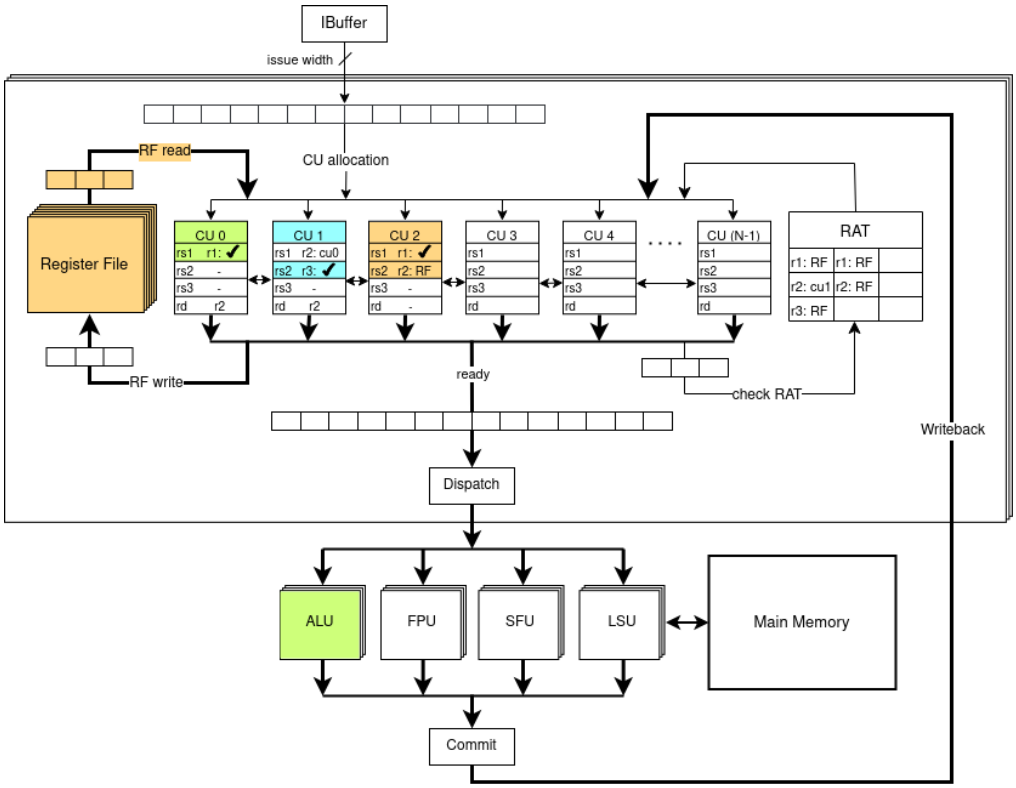
(G) **Clock Cycle 7:** Instruction (A) is scheduled to Dispatch. CU 1 reads r3 data from the Register File and CU 2 remains stalled.



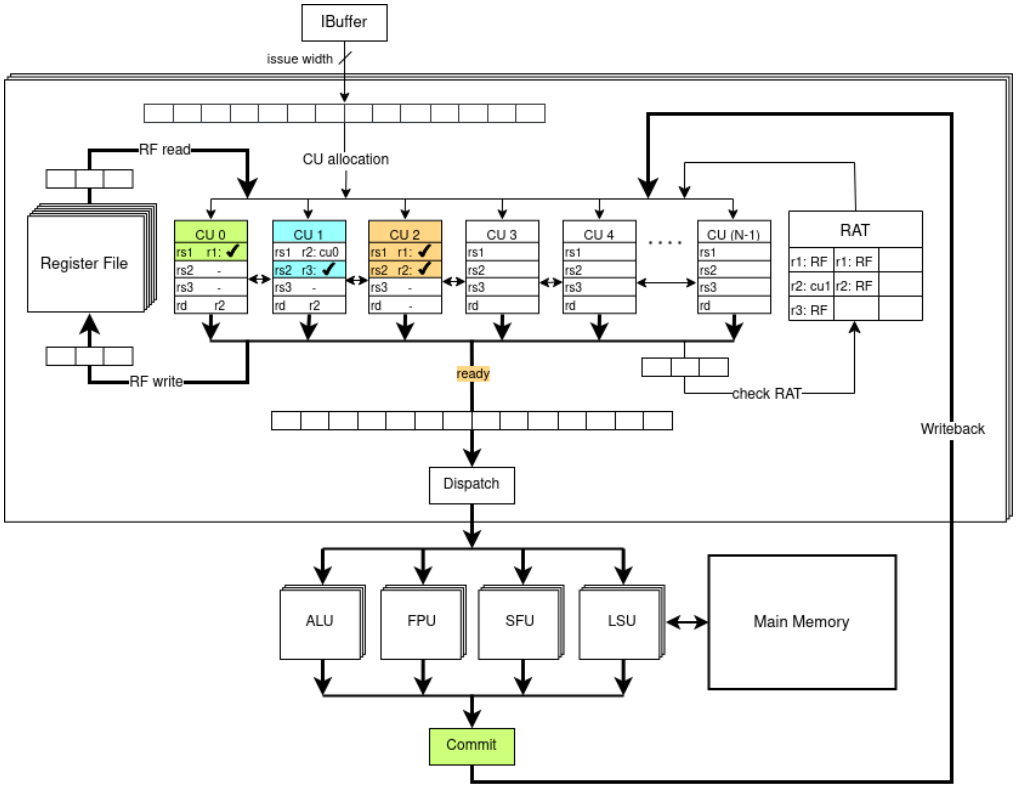
(H) **Clock Cycle 8:** Instruction (A) moves to the Dispatch stage to be assigned a Function Unit. CU 1 updates rs2 as "valid" but cannot be marked as "ready" yet, awaiting the result for r2 from CU 0. CU 2 is queued for reading from the Register File.



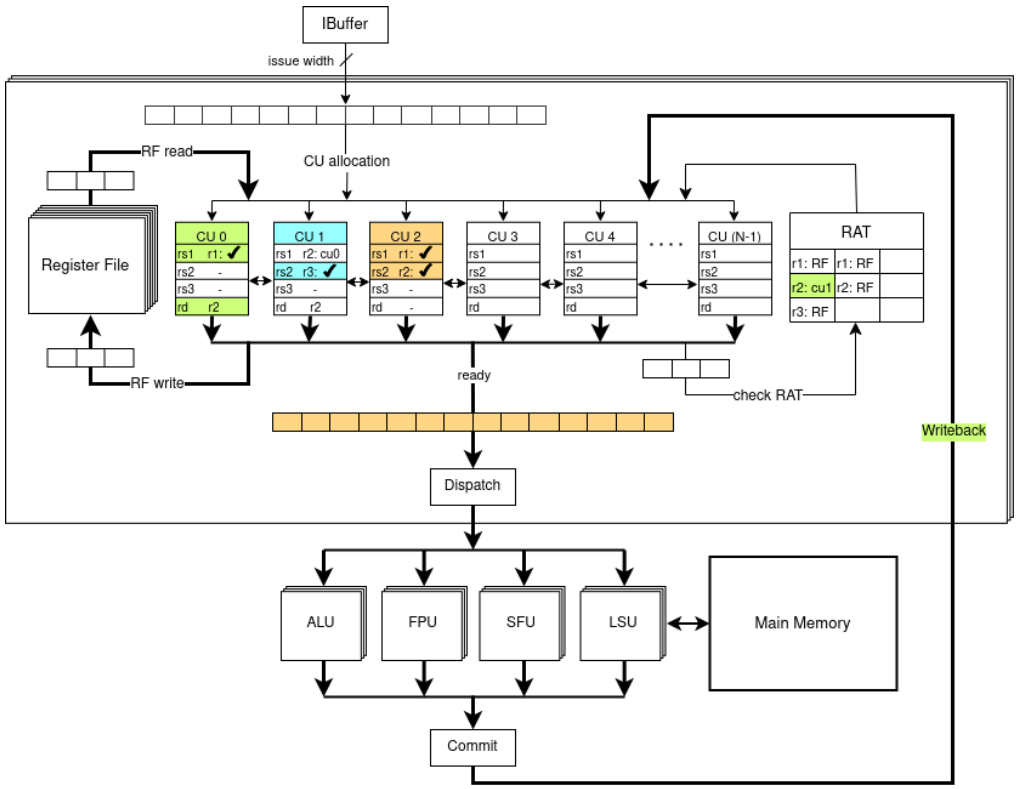
(I) **Clock Cycle 9:** Instruction (A) occupies the Arithmetic Logic Unit (ALU) for Execution, CU 1 remains stalled and CU 2 acquires r1 data from the Register File.



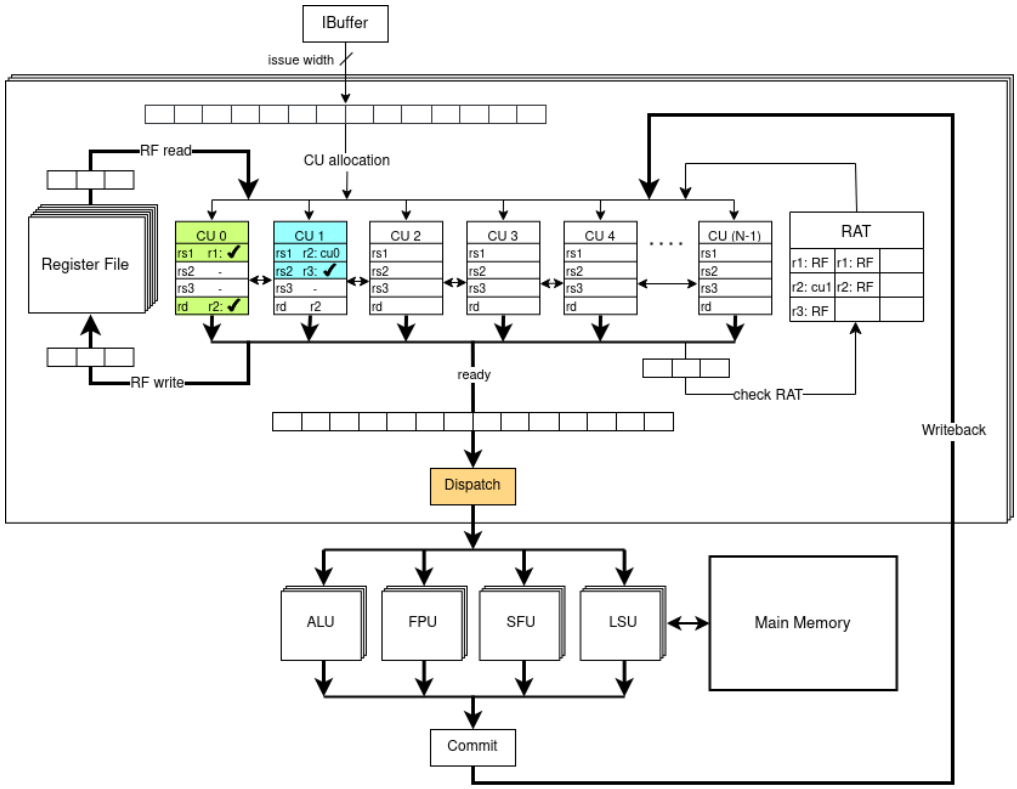
(J) **Clock Cycle 10:** (A) is still Executing, CU 1 is stalled and CU 2 flags rs1 as ”valid” and reads the data for r2 from the RF.



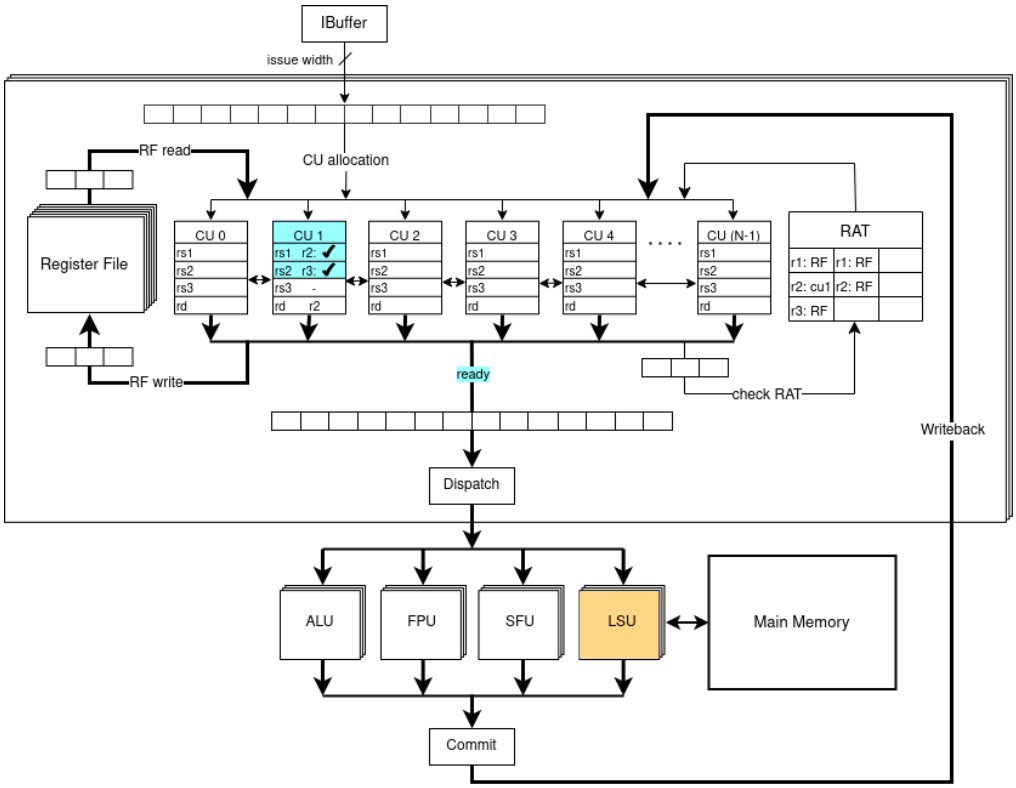
(K) **Clock Cycle 11:** Instruction (A) has completed its Execution and Commits, while CU 1 continues to wait. CU 2 updates rs2 status as ”valid” and marks itself as ”ready.”



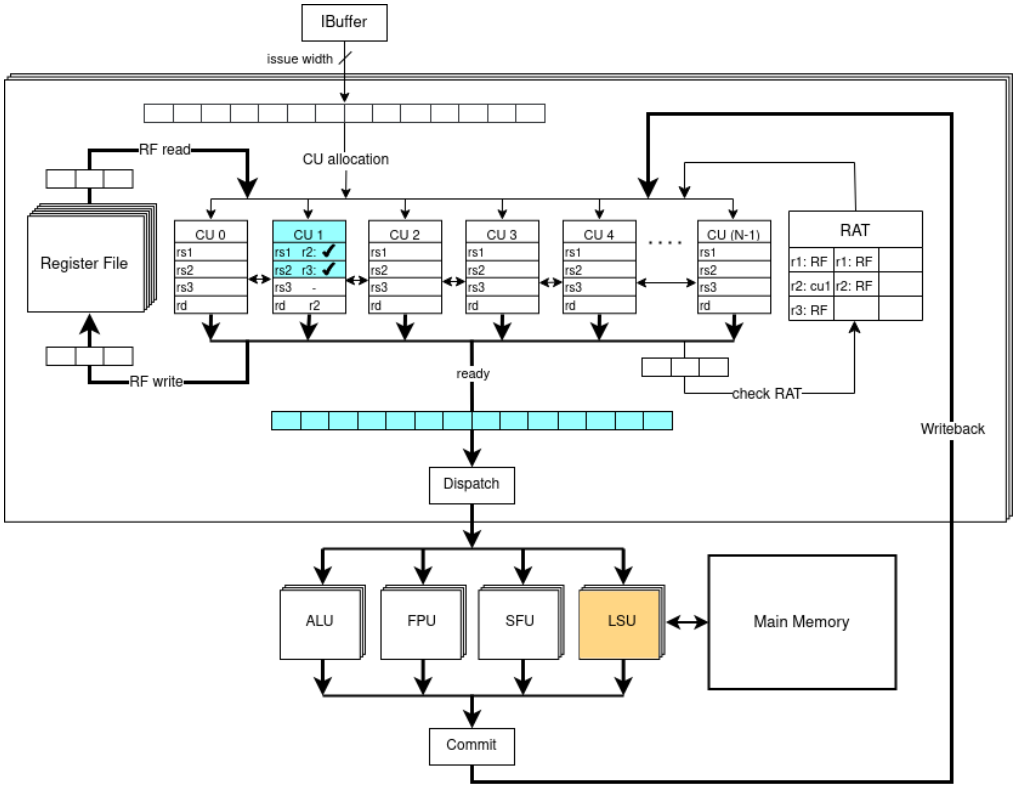
(L) **Clock Cycle 12:** Instruction (A) proceeds to Writeback, updating its CU's destination register data field with the result. Upon checking the corresponding RAT entry and finding another CU's ID, it determines there is no need to write the result to the RF. Meanwhile, CU 1 remains stalled, and CU 2 is scheduled for Dispatch.



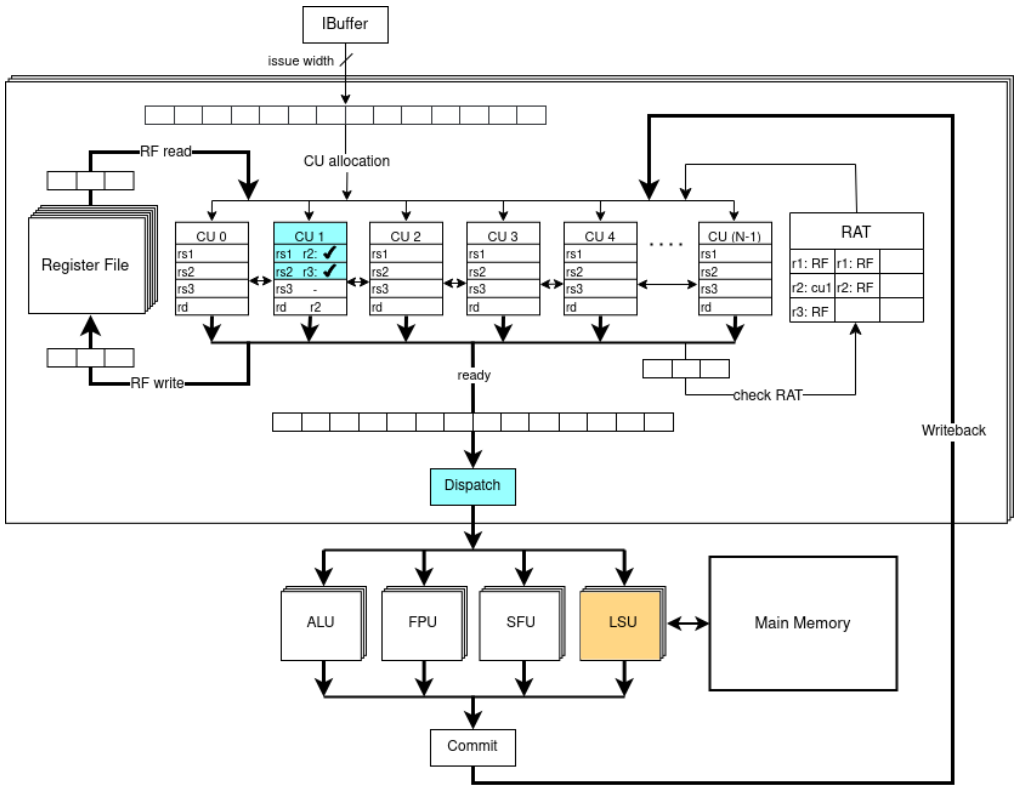
(M) **Clock Cycle 13:** CU 0 marks its destination register's (r2) data as "valid" and CU 1 obtains that data. Instruction (C) is in the Dispatch stage to be assigned a Function Unit.



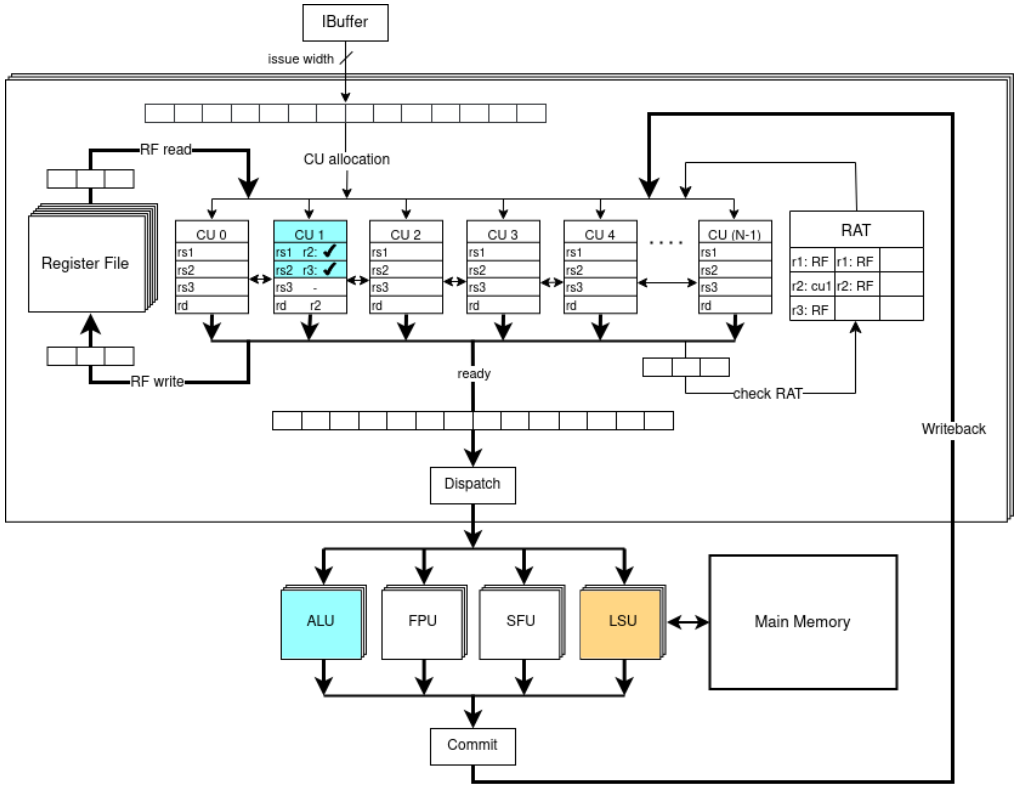
(N) **Clock Cycle 14:** Having finished with all obligations, CU 0 is Deallocated. CU 1 updates rs1 as "valid" and marks itself as "ready," while Instruction (C) begins its Execution in the Load Store Unit (LSU).



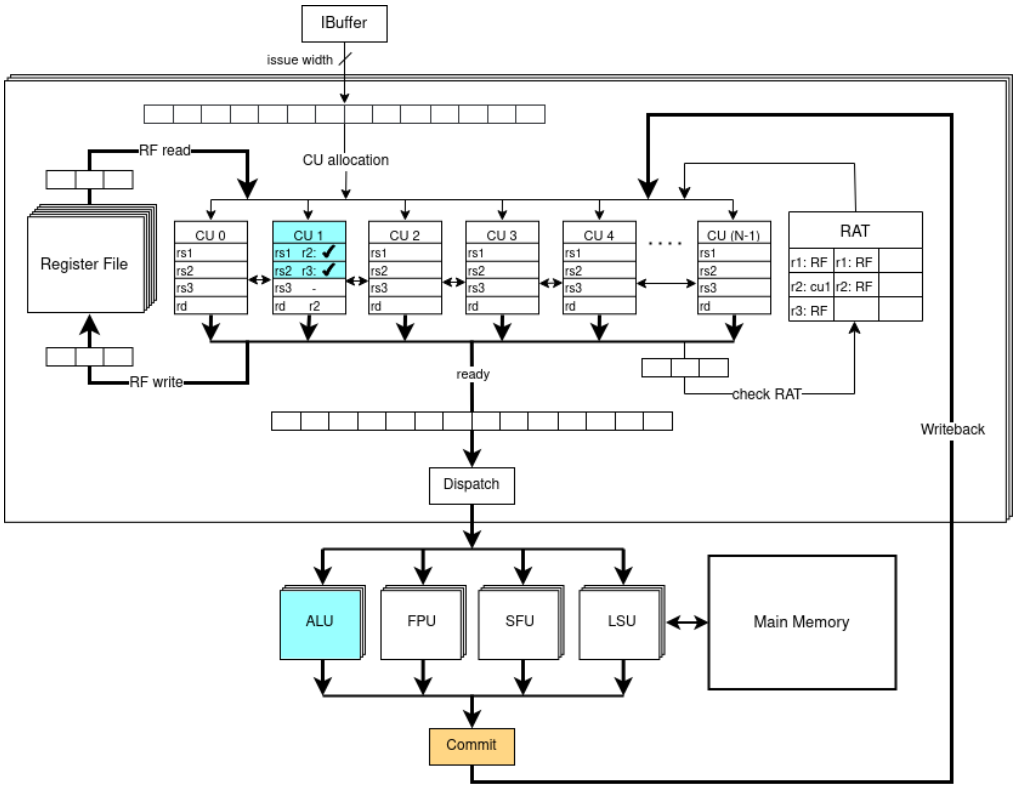
(O) **Clock Cycle 15:** Instruction (B) proceeds to the next stage and instruction (C) continues Executing in the LSU.



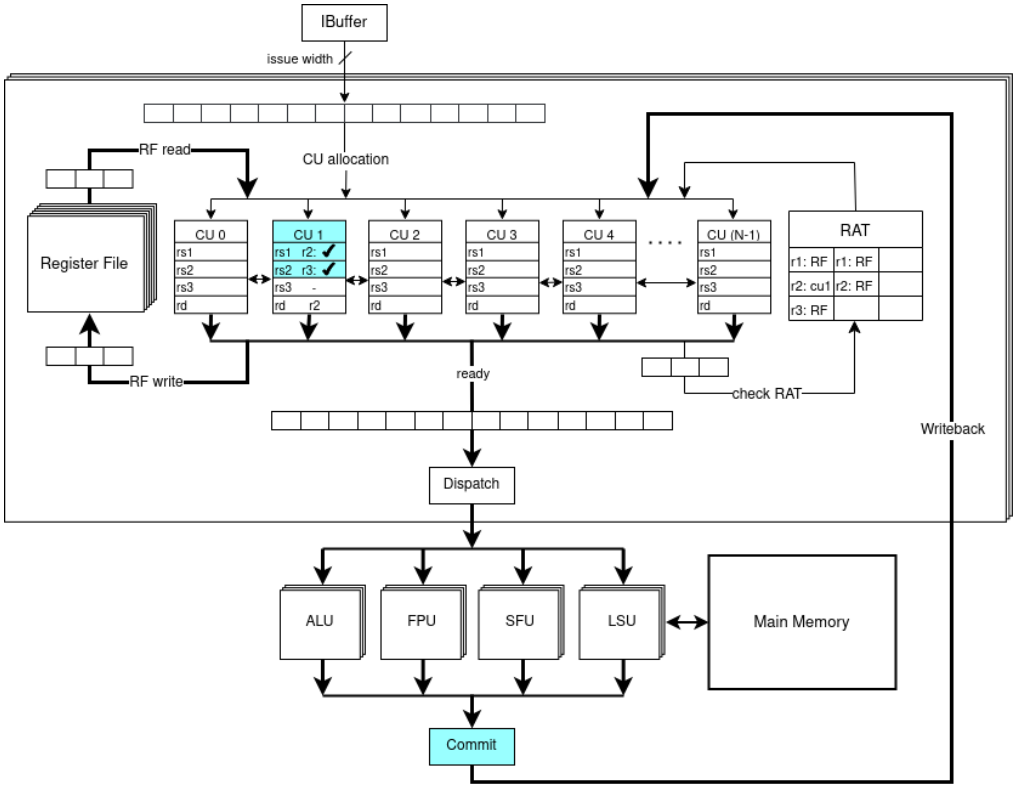
(P) **Clock Cycle 16:** Instruction (B) gets assigned a Function Unit in the DIspatch stage and (C) still Executes.



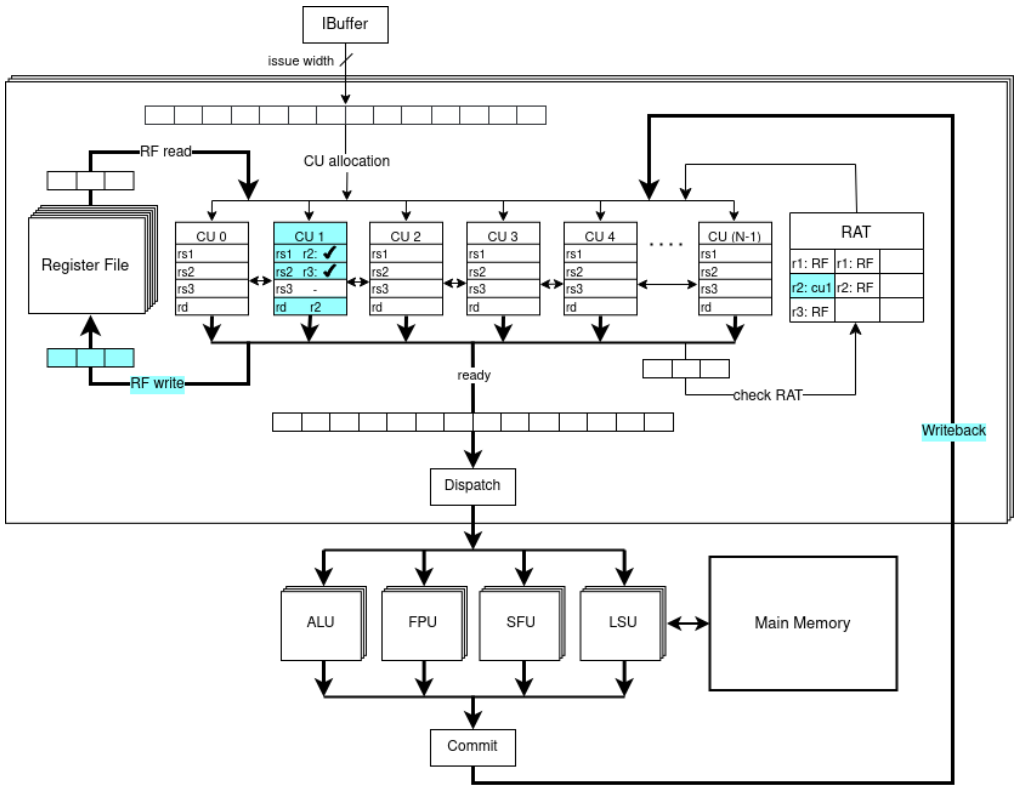
(Q) **Clock Cycle 17:** (B) starts its Execution in the ALU and (C) continues its Execution in the LSU.



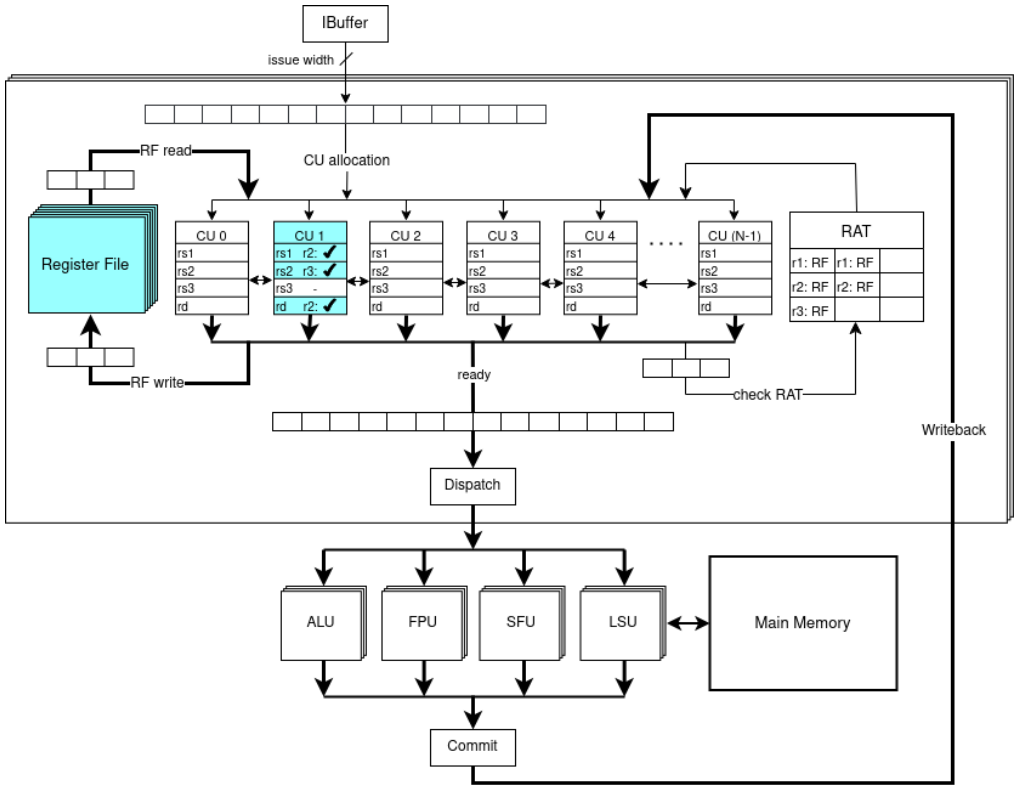
(R) **Clock Cycle 18:** Instruction (B) still Executes in the ALU. Instruction (C), having finished Executing, is currently in the Commit stage.



(S) **Clock Cycle 19:** Instruction (B) is in the Commit stage, while (C) has been discarded as it no longer has pending tasks.



(T) **Clock Cycle 20:** Instruction (B) performs a Writeback, transferring its result to the specified CU field. It also identifies its ID in the r2 entry of the RAT, preparing for a subsequent write to the Register File.



(U) **Clock Cycle 21:** CU 1 marks its destination register's data as "valid" and writes it to the Register File. It is set for deallocation in the following cycle.

FIGURE 5.6: Per-cycle Execution of 3-Instructions Example in the LOOG-Vortex Pipeline

5.4 Microarchitecture Tradeoffs & Optimizations

5.4.1 Optimizing Routing between Collector Units

For the Collector Units to acquire broadcasted data from another CU for any of their 3 source registers, there arises a routing issue due to the excessive number of connections required. Specifically, for N Collector Units and T Threads per Warp, the number of wires needed for data transfer is given by the formula: $3 \times N \times (N-1) \times T \times \text{data\_size}$. To minimize this problem, we introduce an in-betweeen register to hold the valid result from the broadcasting CU and, in the subsequent cycle, pass this data to any CUs that depend on it. Note that the area overhead is minimal as only one register is needed, because only one Writeback can occur in each cycle.

This modification reduces the number of wires for N Collector Units and T Threads per Warp to $4 \times N \times T \times \text{data\_size}$, significantly easing the routing congestion, especially for large N. Furthermore, this adjustment does not impact the latency of the broadcast, as it can be "hidden" within the two pipelined cycles already necessitated by the Writeback with `eop`, the rise of the `rd valid` flag, and the possible write to the Register File. In practice, this approach not only slightly decreases the overall area of the design on the FPGA, as mapping can utilize the resources more efficiently, but it also significantly reduces the length of the critical path, thus optimizing the GPU's clock frequency that can be targeted.

The routing schemes discussed earlier are depicted in the figures provided below. Figure 5.7 illustrates the original "all to all" routing scheme for 4 CUs, while figure 5.8 displays the routing configuration after the implementation of the proposed modifications.

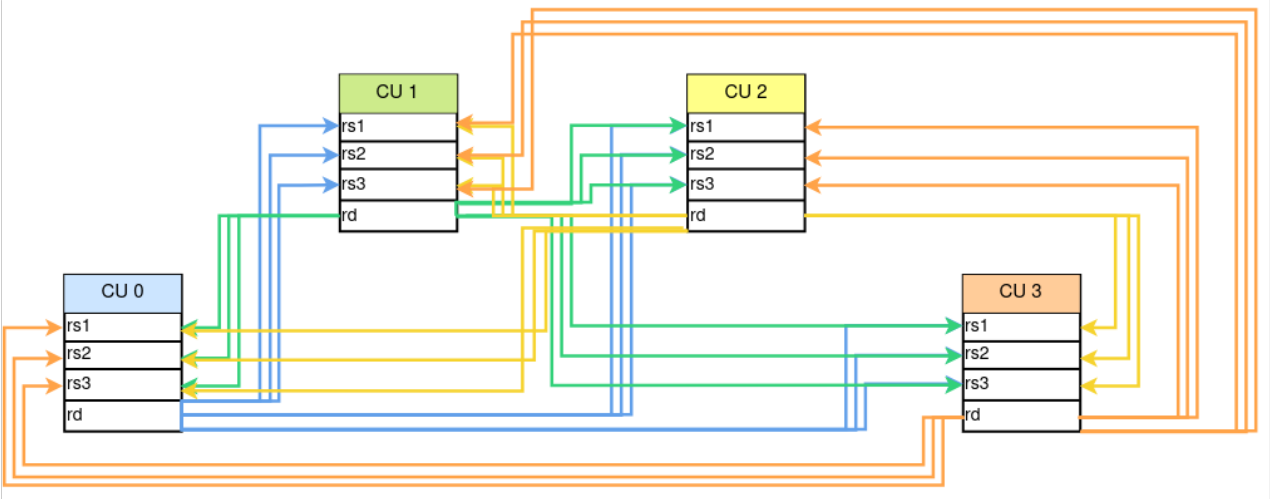


FIGURE 5.7: Collector Units register routing scheme before optimization.

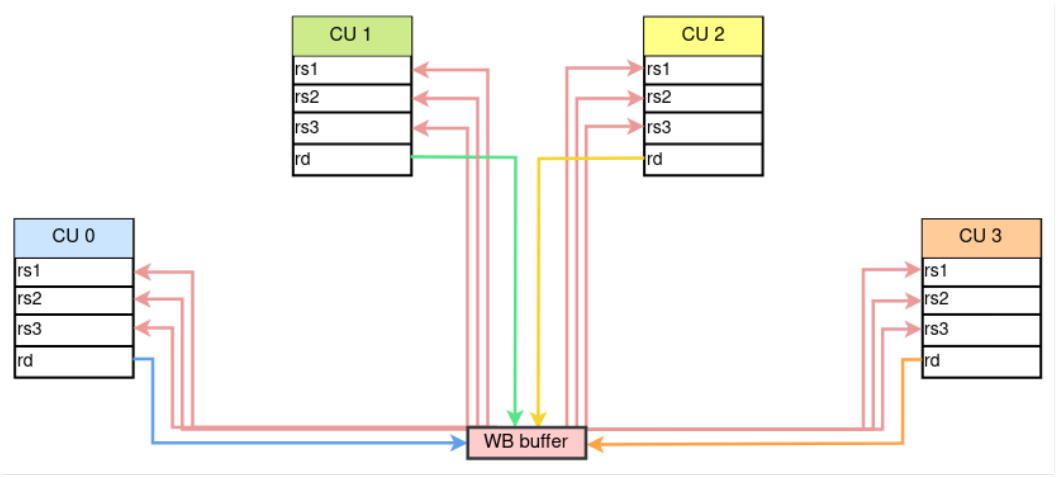


FIGURE 5.8: Collector Units register routing scheme before optimization.

Additionally, when configuring an SM with a large number of Collector Units (e.g., more than 12), routing congestion on the FPGA can be further reduced by employing a second Writeback buffer. This secondary buffer registers the same result as the primary one but is connected to a different subset of Collector Units. Although with this approach the total number of wires are slightly increased,the routing load is effectively distributed across multiple components, thereby alleviating congestion. In this configuration, the first buffer broadcasts results to one half of the Collector Units (identified by distinct IDs), while the second buffer serves the remaining units. An illustrative example is shown in figure 5.9.

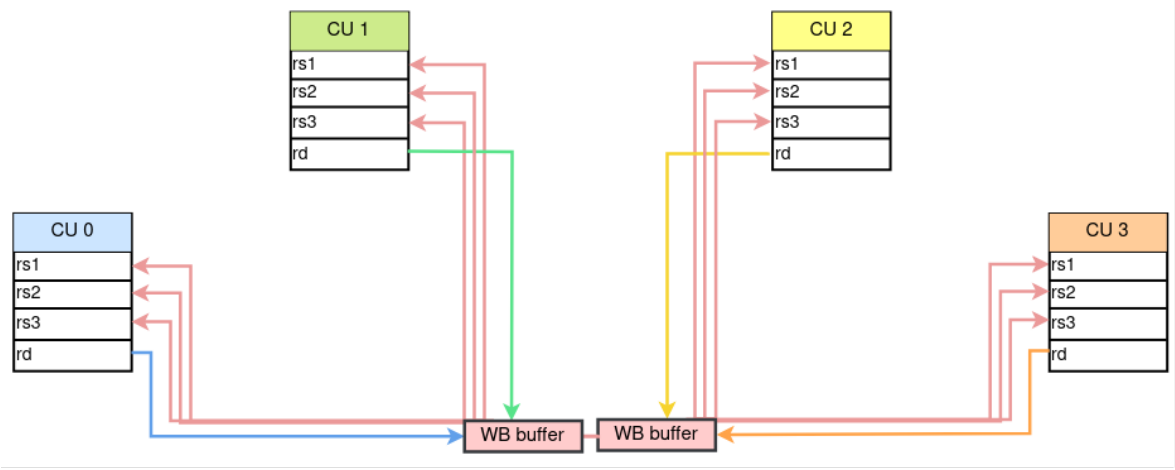


FIGURE 5.9: Collector Units register routing scheme after optimization with 2 Writeback buffers.

This optimization significantly reduces routing congestion, thereby enabling a more efficient utilization of design resources. Figure 5.10 illustrates this improvement by

comparing the area of the baseline in-order Vortex pipeline with that of LOOG-Vortex incorporating 8 Collector Units, both before and after the optimization. The light-colored part corresponds to the LUTs used as memory and the darker part to the LUTs used for implementing the cirquit's logic functions. The comparison is provided across various configurations of the SM in terms of warps and threads.

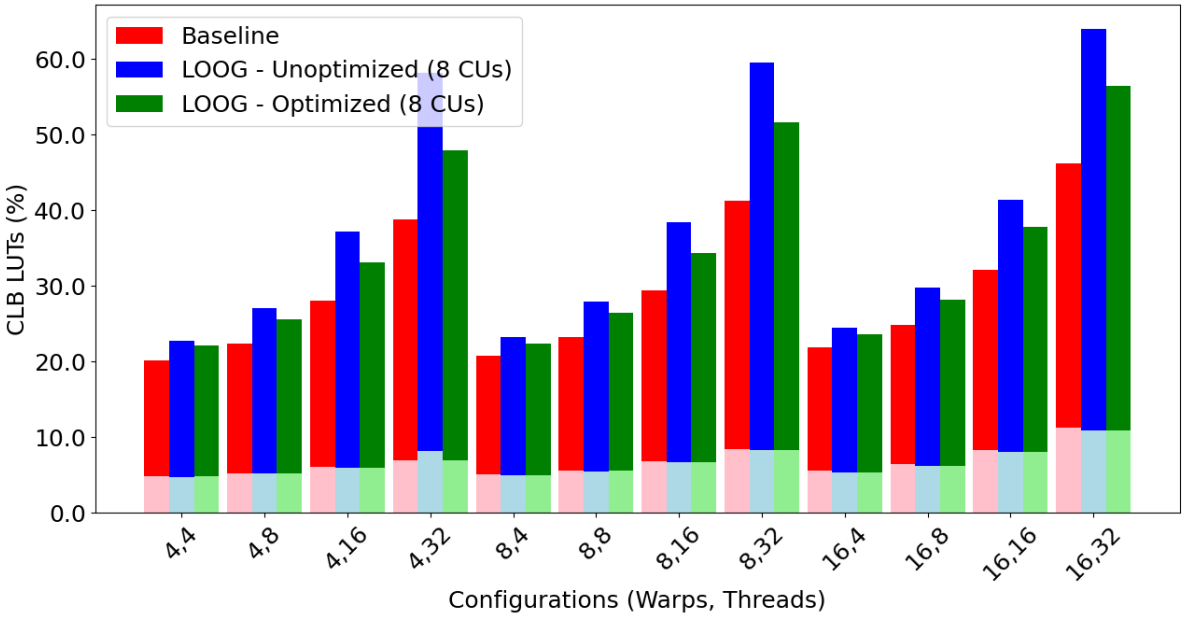


FIGURE 5.10: Comparison of CLB LUTs utilization of in-order Vortex (baseline), LOOG-Vortex before the routing optimization (8 CUs) and LOOG-Vortex after the routing optimization (8 CUs) across different warps-threads configurations of the SM.

5.4.2 Exploring Scheduling Strategies for RF Reads

In our LOOG-Vortex design, a common reason of instruction stalls during the Operand Collect stage is the single-ported Register File. With only one Collector Unit permitted to access the RF at any given time—and given that each access requires as many cycles as the number of source operands to be read—a queue of Collector Units often develops. This raises the question of how best to schedule these RF access requests.

We investigated several scheduling schemes: an ID-based arbiter, which first grants access to the Collector Unit with the smallest ID among those with valid requests; a Round-Robin arbiter, which cyclically prioritizes valid Collector Units; and finally a Round-Robin arbiter paired with Round-Robin scheduling selecting which Collector Unit to allocate from a pool of empty units. As illustrated in Figure 5.11, these alternative scheduling methods did not have a significant impact—either positive or

negative—on LOOG-Vortex's IPC. Consequently, we have chosen to retain our original approach, which employs an ID-based prioritization for RF accesses.

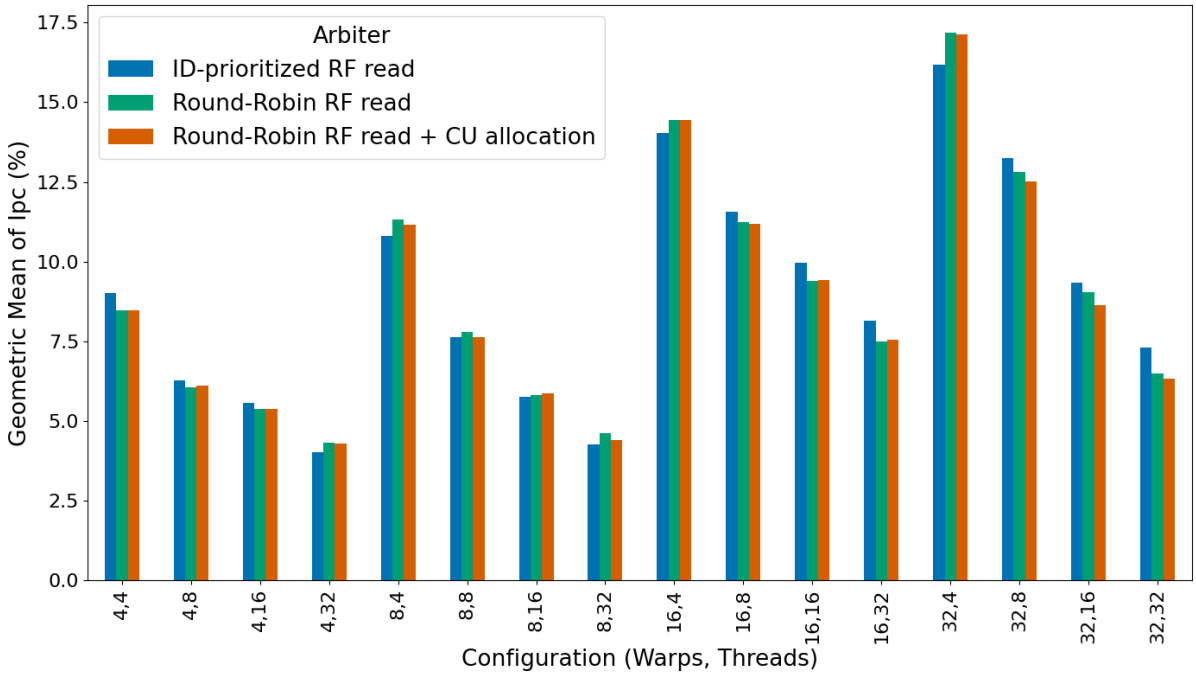


FIGURE 5.11: Comparison of LOOG-Vortex IPC across different SM configurations (warps, threads) for different scheduling schemes for RF accesses in the Operand Collect stage. The figure contrasts an ID-based arbiter, a Round-Robin arbiter, and a Round-Robin approach with additional Round-Robin CU allocation scheme.

5.4.3 Introducing the Register Renaming Stack (RRS)

Unlike conventional GPU architectures, where instructions remain in the Collector Units only for a few clock cycles - until their source operands' data are read from the Register File, the LOOG execution scheme necessitates that instructions occupy CUs up to the Writeback stage. Consequently, the average CU allocation period is significantly extended in LOOG. This can become a bottleneck to LOOG's performance gains, as the structural hazard posed by the lack of available CUs for new instructions arriving at the Operand Collect stage can lead to stalls. Minimizing such stalls within the current microarchitecture would mean increasing the number of CUs, which come at a substantial area cost. This requirement introduces a significant tradeoff, which will be elaborated upon later in this thesis.

The solution to this, as proposed by Iliakis et al. [9], involves introducing a new structure to the pipeline, called the Register Renaming Stack, designed to accommodate instructions from the Dispatch stage until they reach the Writeback stage. The entries within the RRS comprise several components: an allocation status bit, a destination register data field, a thread mask field (to ensure only the correct part of the result data

is written back), and a `rd_valid` bit, which indicates that the `eop` signal has been detected and the result is valid. While this configuration might seem complex, it does not entail a significant area overhead; the data and validation fields are transferred from the CUs of the previous design to the RRS. This adjustment effectively shifts the burden from the Collector Units, relieving them of the responsibility to store the result data and its validation status until the completion of all Writeback operations for the housed instruction, as well as the responsibility to broadcast the data to other CUs and potentially write it to the RF.

Collector Unit	
1	allocation status
1	dispatch status
3	rs data valid status
3	rs to be read from RF
T	tmask
3 x (RRS ID width)	rs source ID
(INSTR DATA width) - T	instr. UUID, warp ID, PC, regs etc.
3 x T x (DATA width)	rs1 data, rs2 data, rs3 data

RRS entry	
1	allocation status
1	rd data valid status
T	tmask
T x (DATA width)	rd data

FIGURE 5.12: Collector Unit and Register Renaming Stack entry fields with their respective size in bits.

In implementing the RRS, two methodologies were evaluated. The initial approach involved allocating an RRS entry during the dispatch phase by the Collector Unit. In this scenario, the Register Alias Table would need to accommodate either a CU ID or an RRS ID for each register source. Subsequently, the CU ID would be updated to the corresponding RRS ID at dispatch, which introduced significant area and routing overheads. Consequently, the decision was made to adopt the second approach.

Under the selected method, each instruction is assigned an RRS entry concurrently with a CU allocation as it enters the Operand Collect stage. The RRS ID is then stored within the Collector Unit, and the entry's allocation status bit is set. Additionally, the thread mask (tmask), which indicates the active threads within the warp, is recorded both in the RRS and in the CU, because its information is only needed for this stage during the Writeback. It's important to note that if an instruction does not require a result to be written back to a destination register, it does not utilize an RRS entry.

For those instructions that necessitate a Writeback, during their visit to the RAT the designated RRS ID is recorded under the destination register's source field. The process

proceeds normally until the CU's source registers' data is valid and it is prepared for the next stage, at which point the instruction transitions carrying its RRS ID instead of the CU ID. The subsequent clock cycle then permits the deallocation of the Collector Unit.

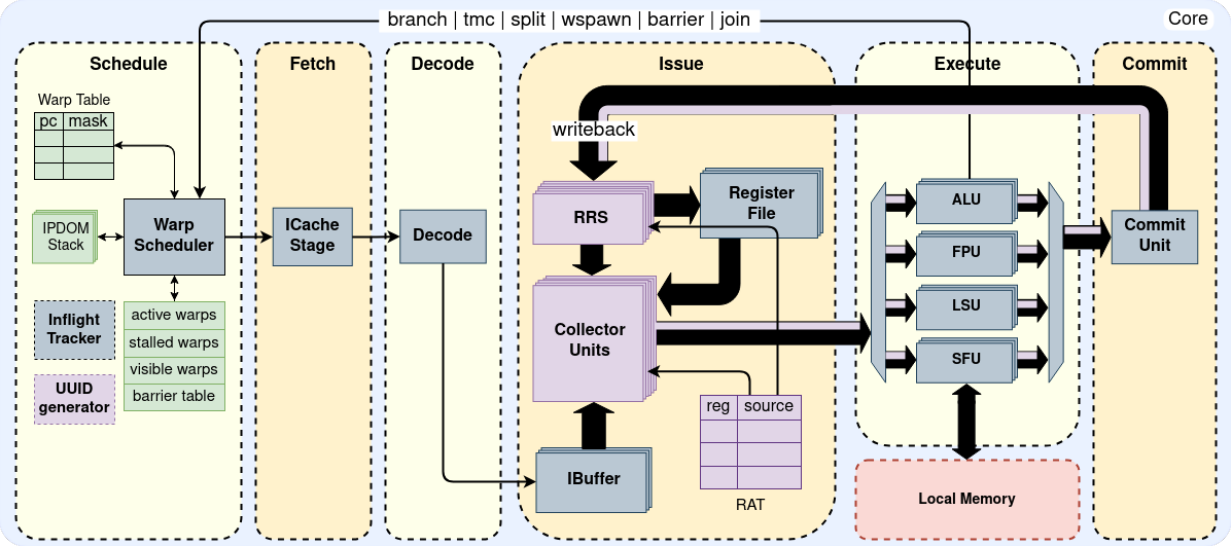


FIGURE 5.13: The LOOG-Vortex Modified Pipeline after the addition of RRS.

Once Execution concludes and Writeback operations commence, the result data is written into the rd data field of the relevant RRS. Upon detection of an end-of-operation (eop) signal, the rd valid bit of the RRS is set and the RRS verifies with the RAT whether the result should be written back to the Register File (RF). In the next cycle, the data is broadcasted and written back and, immediately after this operation, the RRS entry is cleared by resetting the rd valid and allocation status bits, rendering the entry "empty" and available for reallocation by a new instruction.

Indeed, the modification introduced by the Register Renaming Stack effectively addresses the issue of long Collector Unit allocation times, as shown in Figure 5.14. This figure presents the Cumulative Distribution Function (CDF) of the CU allocation period for both the no-RRS LOOG-Vortex (with 8 CUs) and the RRS LOOG-Vortex (with 8 CUs and 12 RRS entries). The results demonstrate that the RRS configuration significantly reduces contention for Collector Units, allowing instructions to enter the Operand Collect stage more quickly. Additionally, Figure 5.16 provides further insight into the stalls caused by unavailable CUs. This CDF highlights that the no-RRS LOOG-Vortex configuration experiences considerably more stalls when CUs are unavailable compared to the RRS LOOG-Vortex. The reduction in these stalls reflects the improved efficiency of CU allocation with the introduction of RRS. Finally, Figure 5.15 presents the CDF of the average CU utilization for both configurations, clearly showing that the RRS LOOG-Vortex achieves more efficient CU utilization.

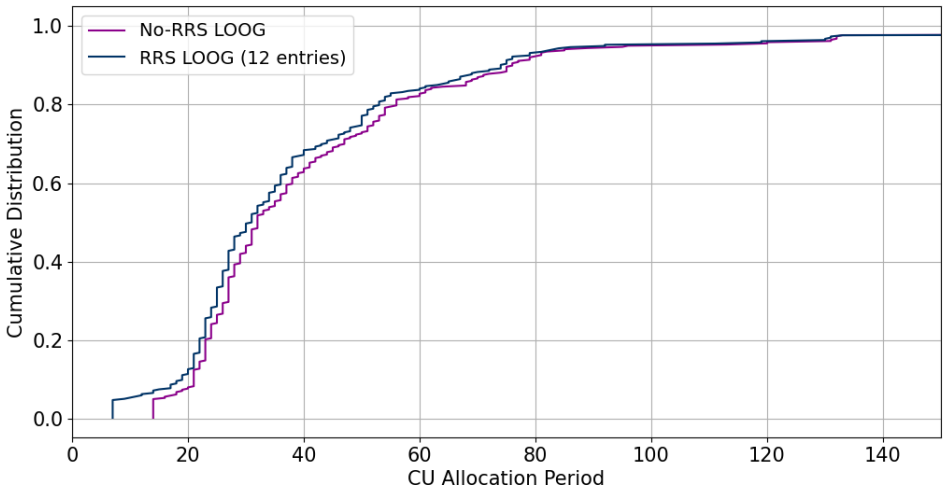


FIGURE 5.14: Cumulative Distribution Function of Collector Unit Allocation Period of no-RRS LOOG-Vortex (8 CUs) and of RRS LOOG-Vortex (8 CUs, 12 RRS entries).

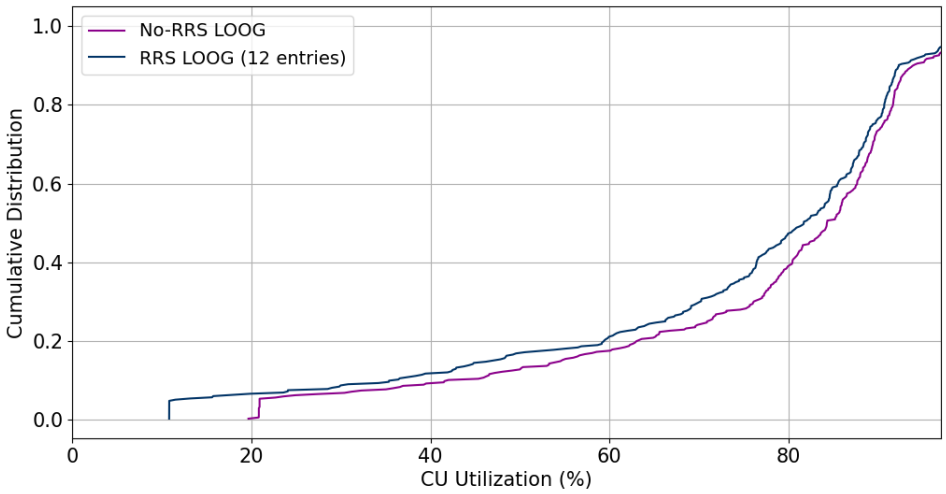


FIGURE 5.15: Cumulative Distribution Function of Collector Unit Utilization of no-RRS LOOG-Vortex (8 CUs) and of RRS LOOG-Vortex (8 CUs, 12 RRS entries).

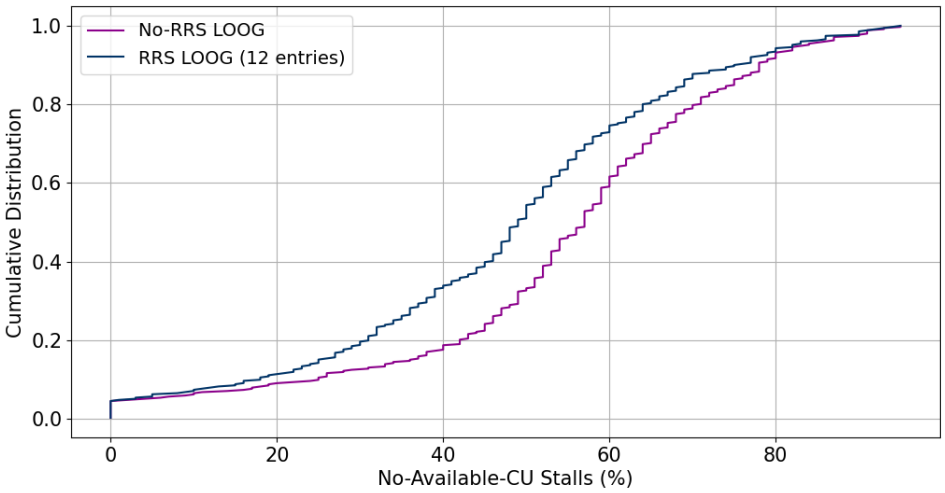


FIGURE 5.16: Cumulative Distribution Function of No-Available-CU stalls of no-RRS LOOG-Vortex (8 CUs) and of RRS LOOG-Vortex (8 CUs, 12 RRS entries).

5.5 Workload Characterization

To evaluate the LOOG-Vortex architecture, we will benchmark our design both before and after the architectural modifications using the 21 applications provided with Vortex 2.0. To gain insight into the workload diversity and its behavior under the LOOG scheme, we characterized the applications based on their stall sources when executed on the in-order Vortex pipeline.

An instruction may stall between the Decode and Issue stages if the Issue stage — specifically the Scoreboard — is occupied by a data-dependent instruction, a stall referred to as an IBuffer stall. Additionally, structural hazards can cause stalls when an instruction must dispatch to an Execution Unit that is currently busy. These are categorized into ALU, FPU, or LSU stalls, corresponding to stalls due to occupied Arithmetic Logic Units, Floating-Point Units, or Load-Store Units, respectively. Finally, stalls can occur in the Control and Status Register stage when synchronization of warp execution information, such as active thread status, is required; however, these account for a very small portion of the total execution stalls.

To classify applications according to their stall source behavior, we employed Hierarchical Clustering using the cosine metric. This approach emphasizes the proportional relationships between stall types rather than their absolute magnitudes. The clustering yielded five distinct groups, as shown in Figure 5.17.

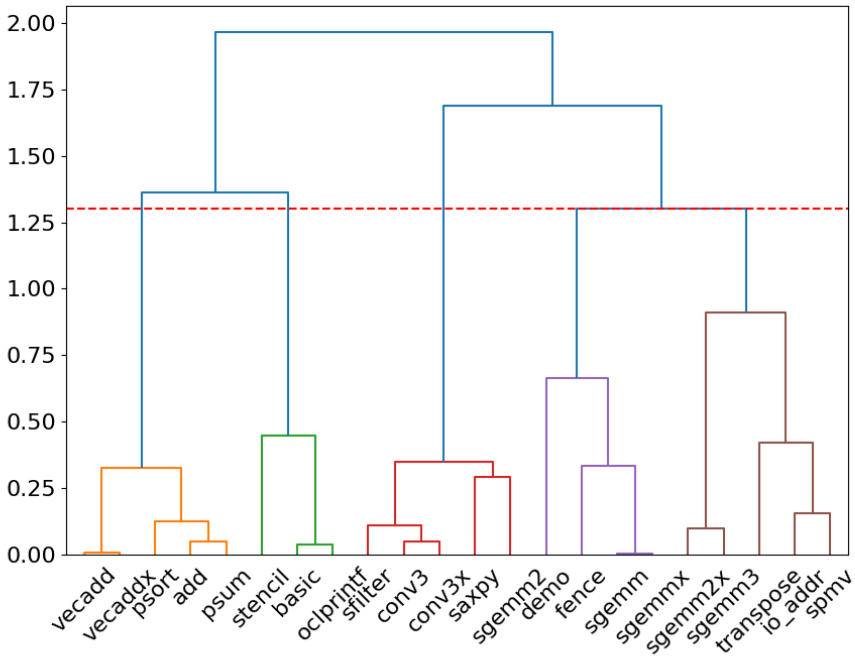
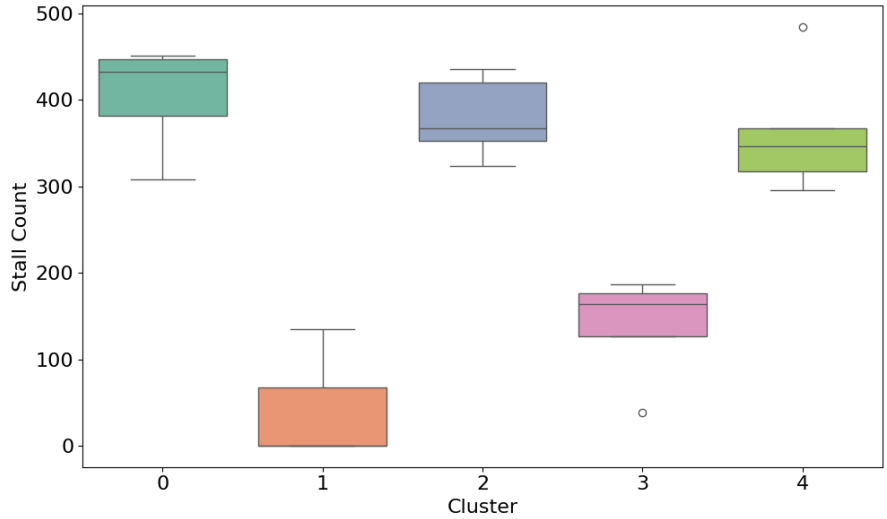


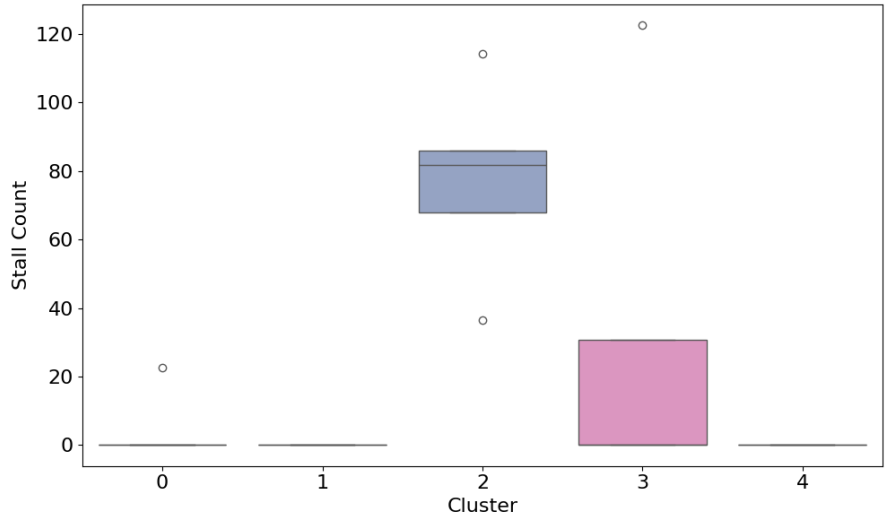
FIGURE 5.17: Dentrogram of Hierarchical Clustering of 21 applications based on their stall types using the cosine metric.

To further elucidate each cluster's characteristics, we generated boxplots for the various stall sources, presented in Figure 5.18.

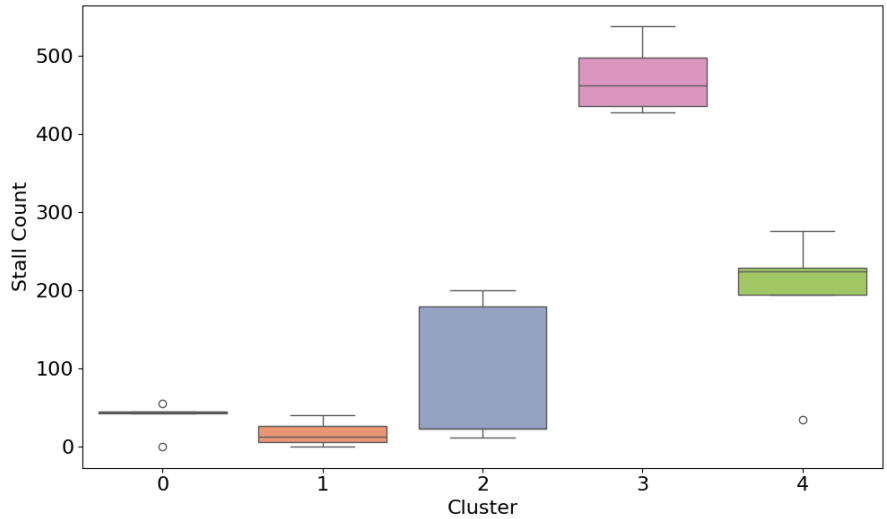
In Figure 5.18b, only clusters 2 and 3 exhibit stalls related to floating-point operations. When combined with the insights from subfigures 5.18a and 5.18c, we can characterize cluster 2 as being both floating-point and ALU compute-instensive, whereas cluster 3 is primarily floating-point and memory-intensive. Additionally, subfigure 5.18c indicates that cluster 3 is memory-intensive in the absence of significant floating-point activity. Clusters 0 and 1, on the other hand, show neither notable memory-intensive behavior nor floating-point stalls; instead, they have the highest CSR stall counts (5.18e). The difference between these two clusters lies in their ALU operations—cluster 0 suffers more performance degradation from ALU-induced stalls, while the IBuffer analysis in 5.18d suggests that cluster 1 is relatively free from data dependency issues.



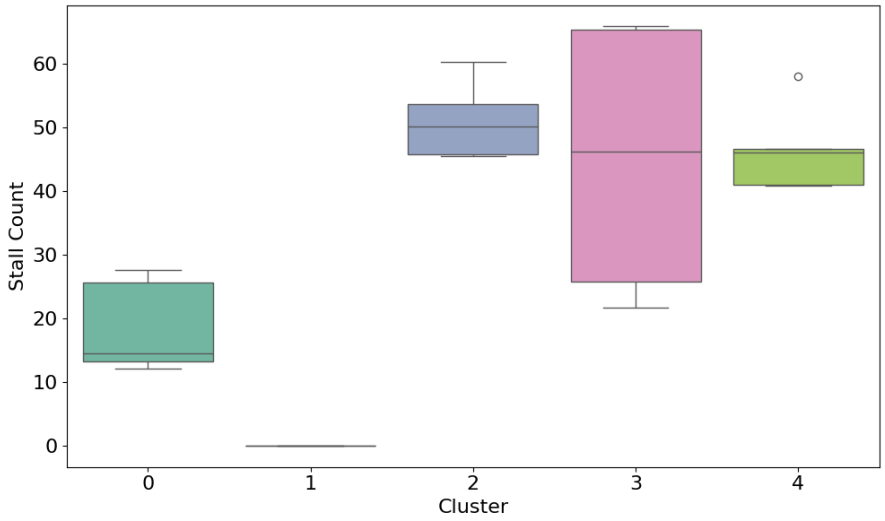
(A) Stalls caused by ALU Execution.



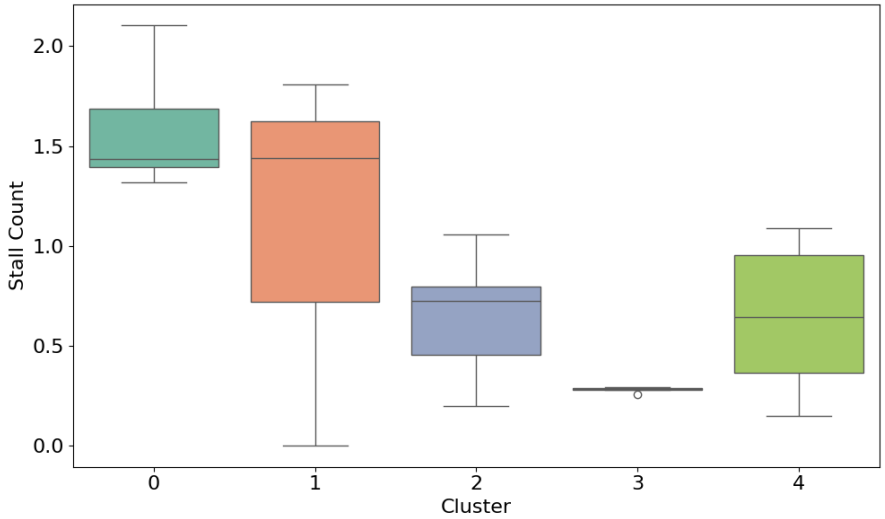
(B) Stalls caused by FPU Execution.



(C) Stalls caused by Memory Accesses.



(D) Stalls caused by IBuffer delays.



(E) Stalls caused by CSR unit.

FIGURE 5.18: Boxplot analysis of pipeline stalls in the cosine complete benchmark.

Chapter 6

Experimental Evaluation

6.1 Introduction

In this section, we assess the runtime performance of LOOG-Vortex against our baseline—the in-order Vortex GPGPU 2.0. We analyze various SM configurations by adjusting the number of warps per SM and the threads per warp, evaluating performance using 21 micro-benchmarks that represent common GPU applications integrated into Vortex. Additionally, we optimize LOOG-Vortex by fine-tuning its LOOG-specific parameters, such as the number of Collector Units and RRS entries.

6.2 LOOG-Vortex without the RRS

Initially, we evaluate the performance of LOOG without the Register Renaming Stack in terms of Instructions Per Count (IPC). We calculate the geometric mean of the IPC gains (expressed as a percentage %) of LOOG-Vortex over the baseline Vortex performance and depict these gains for each SM configuration (warps, threads), as seen in Figure 6.1. Each color in the figure corresponds to a specific number of Collector Units (CUs) available in the Operand Collect stage.

Notably, every configuration demonstrates a positive IPC gain, consistently outperforming the baseline. In general, increasing the number of CUs correlates with higher IPC gains, as the arrival of new instructions from the preceding stage is less likely to be stalled by a lack of empty CUs. Additionally, it is particularly interesting that LOOG-Vortex exhibits a marked performance advantage over the baseline when operating with a smaller number of threads per warp. This can be attributed to the increased volume of warp instructions that results from having fewer threads per warp; under data dependency conditions, in-order Vortex incurs stall penalties more frequently, whereas LOOG-Vortex is often able to bypass these delays.

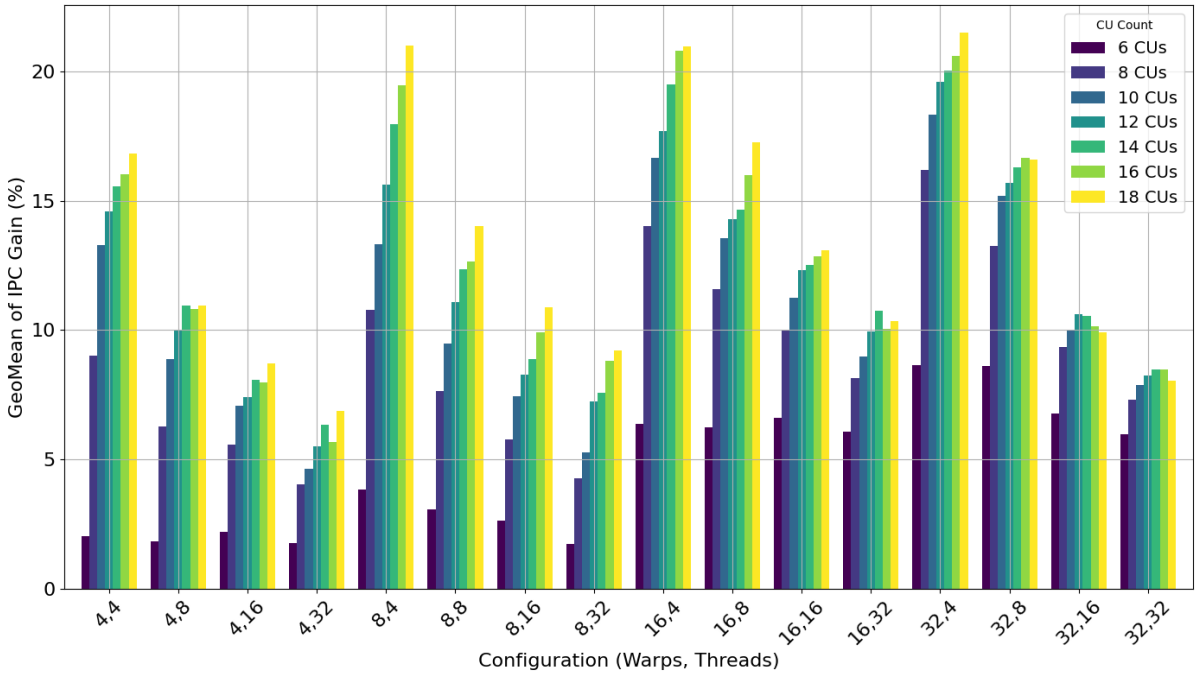


FIGURE 6.1: Geometric Mean of IPC (21 applications) per SM configuration (warps, threads) for different amounts of Collector Units in the no-RRS LOOG-Vortex microarchitecture.

Additionally, it is noteworthy to observe the IPC gains across the application clusters, as defined in Section 5.5, which are depicted in Figure 6.2. First, Cluster 3—which comprises memory-bound floating-point benchmarks—shows the smallest gains with LOOG-Vortex, as expected since no reordering is applied to Load-Store instructions. Although Cluster 4 is also characterized by a moderate amount of stalls in memory operations, its performance improves with an increasing number of CUs because non-memory, fixed-point instructions can exploit the additional CUs to bypass previous high-latency operations. In Cluster 1, the in-order Vortex experiences minimal stalls due to dependencies or memory operations, resulting in moderate, CU-independent gains with LOOG-Vortex. Conversely, Cluster 0, while having relatively few backend stalls in the in-order configuration, is affected by a higher number of dependencies, leading to generally higher gains that do not scale linearly with the number of CUs. Finally, Cluster 2, which is both compute- and memory-intensive, leverages additional CUs more effectively to mask Load-Store induced stalls.

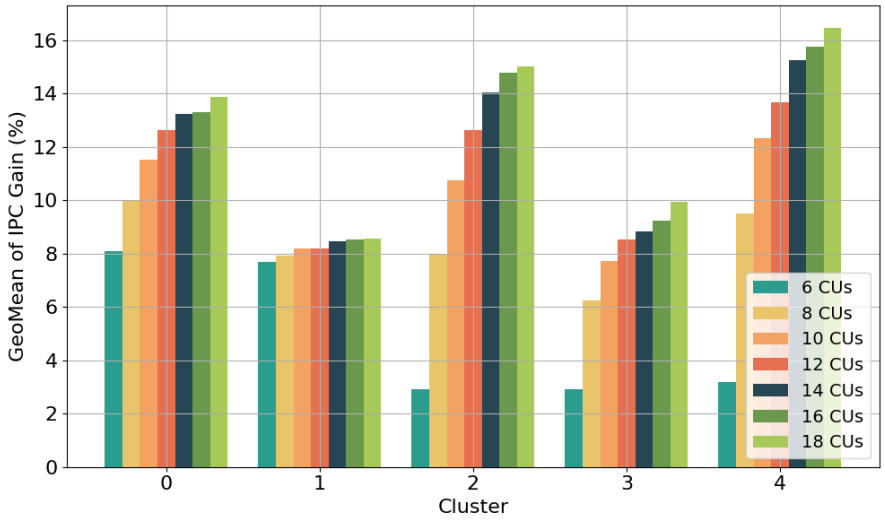


FIGURE 6.2: Geometric Mean of IPC (16 SM configurations) per Cluster of applications for different amounts of Collector Units in the no-RRS LOOG-Vortex microarchitecture.

We also evaluate LOOG-Vortex in terms of area and power by implementing the LOOG-Vortex design for an FPGA device (we use the AMD Alveo U50 Data Center Accelerator Card). We use Vivado 2021.1 for the synthesis and implementation of our design for the specific device. Figure 6.3 presents a per-configuration comparison of CLB LUT utilization—both for memory and logic—across varying numbers of CUs. As anticipated, the warp size (threads) is the primary factor influencing LUT area, as both the data stored in the CUs and the interconnections among them scale with the number of threads.

Next, we integrate our LOOG-Vortex performance results with the area utilization metrics for each SM configuration by constructing the Area-Delay Product (ADP), as illustrated in Figure 6.4. For the performance part, we calculate the geometric mean of the Cycles Per Instruction (CPI) values across the 21 microbenchmarks — normalizing these values to the CPI of the in-order Vortex execution. For the area metric, we average the utilization percentages of CLB Registers, CLB LUTs (both for memory and for logic), BRAM, and DSPs, again normalizing to the corresponding baseline figures.

$$\text{Total Area}(\%) = \frac{CLBRegisters(\%) + CLBLUTs(\%) + BRAM(\%) + DSPs(\%)}{4}$$

With this approach, the baseline architecture consistently achieves an ADP of 1 across all configurations, and values below 1 indicate a more optimized design. We use this to evaluate the LOOG-Vortex microarchitecture for configurations featuring 8, 10, and 12 CUs, respectively.

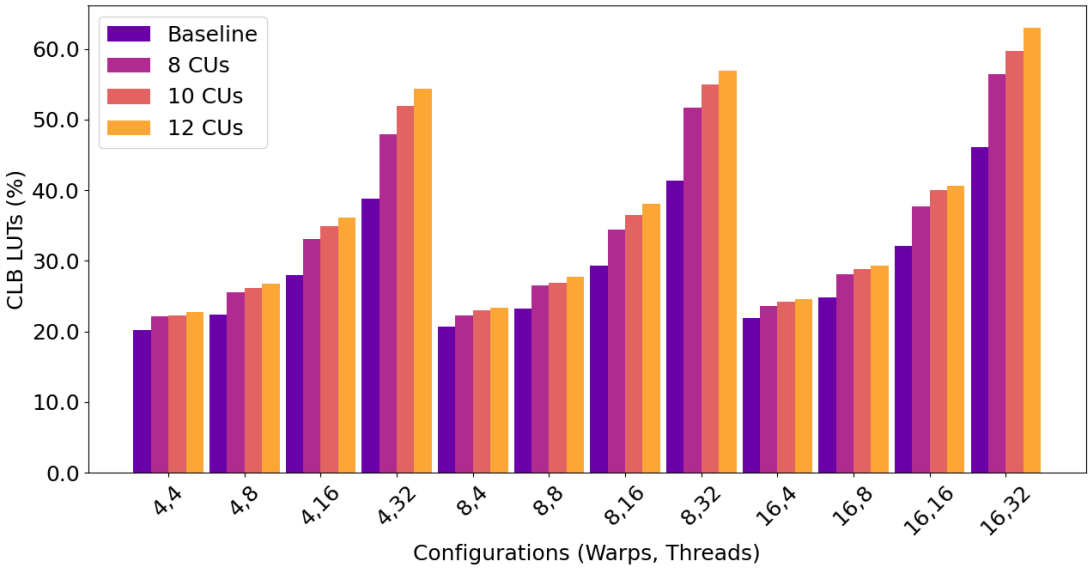


FIGURE 6.3: CLB LUTs utilization of the no-RRS LOOG-Vortex microarchitecture per SM Configuration for different amounts of CUs.

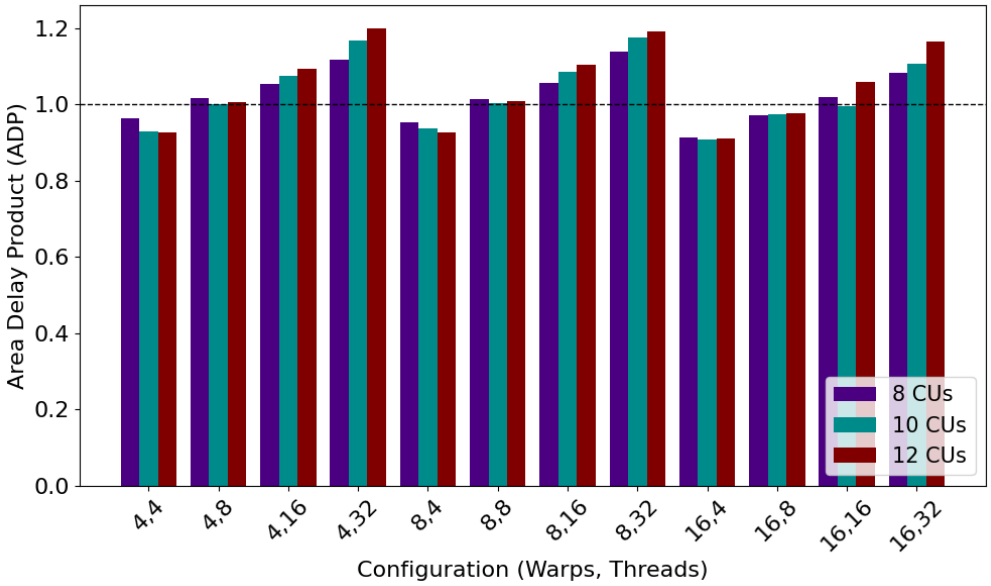


FIGURE 6.4: Area-Delay Product of no-RRS LOOG-Vortex per SM Configuration (Warps, Threads) for different amounts of CUs (normalized to baseline).

Consistent with our earlier findings on IPC and LUT utilization, the Vortex SM configurations that are most favored by LOOG are the ones with smaller warp sizes. Fewer threads per warp yield the highest performance gains compared to the in-order configuration, while also maintaining lower FPGA resource utilization. Consequently,

these configurations are deemed optimal for our design.

Additionally, using the FPGA implementation tool's estimations, we can evaluate the Power consumption of LOOG-Vortex across different SM configurations and for different numbers of Collector Units. As demonstrated by the Power-Delay Product in Figure 6.5, all configurations outperform the in-order baseline. The minimal power consumption overheads are effectively offset by significant performance gains.

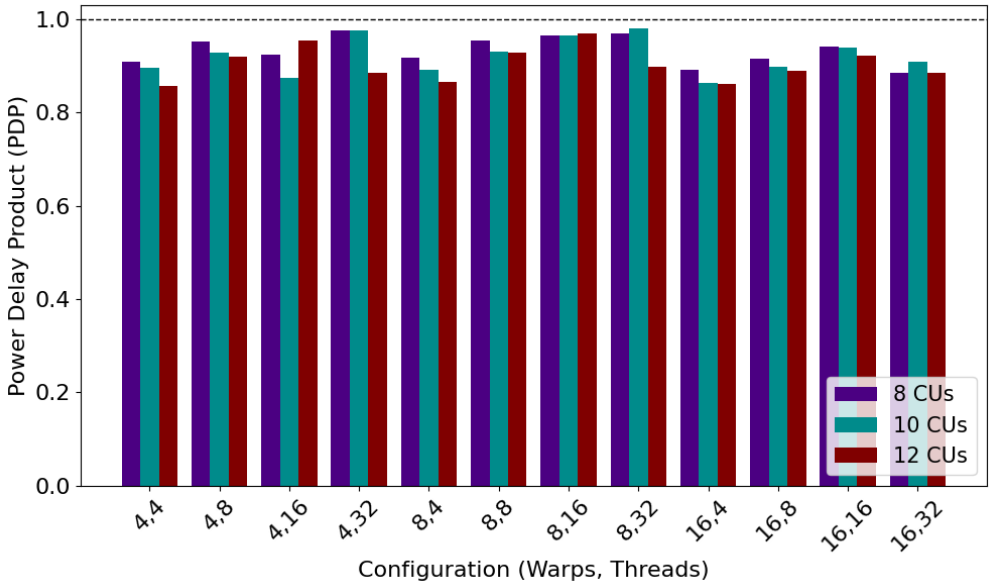


FIGURE 6.5: Power-Delay Product of no-RRS LOOG-Vortex per SM Configuration (Warps, Threads) for different amounts of CUs (normalized to baseline).

6.3 LOOG-Vortex with the RRS

The next step is to evaluate LOOG-Vortex with the Register Renaming Stack in terms of the same metrics as before for Performance, Area and Power. We use the same workload and explore different sizes of the RRS which is dependent on the number oF Collector Units. More specifically, we use RRS entries that are equal to the number of CUs multiplied by a factor 1.5, 2 and 2.5 and the results for the Intructions Per Count are shown in Table 6.1 per amount of CUs and in Table 6.2 per configuration (number of warps, threads per warp). We can see that when adding extra RRS entries to the design, the performance gain saturates for most amounts of CUs in factor 1.5 and for small amounts of CUs in factor 2.

To evaluate LOOG-Vortex equipped with the Register Renaming Stack (RRS), we assessed the design based on the same Performance, Area, and Power metrics as the previous architecture. Using the same workload as before, we varied the RRS size according to the number of Collector Units (CUs), setting the number of RRS entries to

the product of the number of CUs and a factor of 1.5, 2, or 2.5. Table 6.1 presents the geometric mean values of the Instructions Per Count results as a function of the number of CUs, while Table 6.2 details the outcomes across different configurations (varying the number of warps and threads per warp). The findings reveal that the performance benefits of increasing RRS entries saturate: at a factor of 1.5 for most CU counts and at a factor of 2 for smaller CU counts.

Table 6.1: Geomean IPC for Different RRS LOOG Configurations

CUs	No CUs	#RRS = #CUs × 1.5	#RRS = #CUs × 2	#RRS = #CUs × 2.5
6	4.5406	8.2399	11.1017	11.8398
8	8.6834	11.0765	12.0894	12.0894
10	10.3601	12.3409	12.6604	12.6620
12	11.4342	13.3033	13.3528	13.3528
14	12.2760	13.7937	13.7747	13.7747

Table 6.2: Geomean IPC for Configurations at Different $\#RRS$ Multipliers (Including No RRS)

Warps	Threads	No RRS	#RRS = #CUs × 1.5	#RRS = #CUs × 2	#RRS = #CUs × 2.5
4	4	11.0565	14.2037	16.6557	16.6812
4	8	8.0886	9.6174	11.1462	11.7023
4	16	6.6155	7.6426	8.5589	8.8934
4	32	5.1969	6.2939	6.9217	7.2443
8	4	13.4249	17.7534	19.6198	19.8010
8	8	9.5557	12.0388	13.0930	13.4439
8	16	7.4347	9.1152	9.9228	10.1210
8	32	6.1361	7.9722	8.4442	8.6913
16	4	16.1167	18.7691	20.2537	20.6203
16	8	13.2758	15.1972	16.1180	16.2648
16	16	11.4371	12.6132	13.3414	13.5300
16	32	9.3337	9.7893	10.3011	10.6420
32	4	18.0502	20.1719	21.7298	22.1808
32	8	14.8502	15.9538	17.1416	17.4342
32	16	10.1262	10.8004	11.3135	11.4361
32	32	8.1468	8.1162	8.5670	8.8225

Figure 6.6 and Figure 6.7 illustrate the detailed IPC gains, relative to the baseline in-order Vortex architecture, across all configurations of warps, threads, and varying

numbers of CUs for RRS factors of 1.5 and 2, respectively. As anticipated, particularly for smaller CU counts, the performance improvement is more pronounced than in the design without the RRS, as depicted in Figure 6.1.

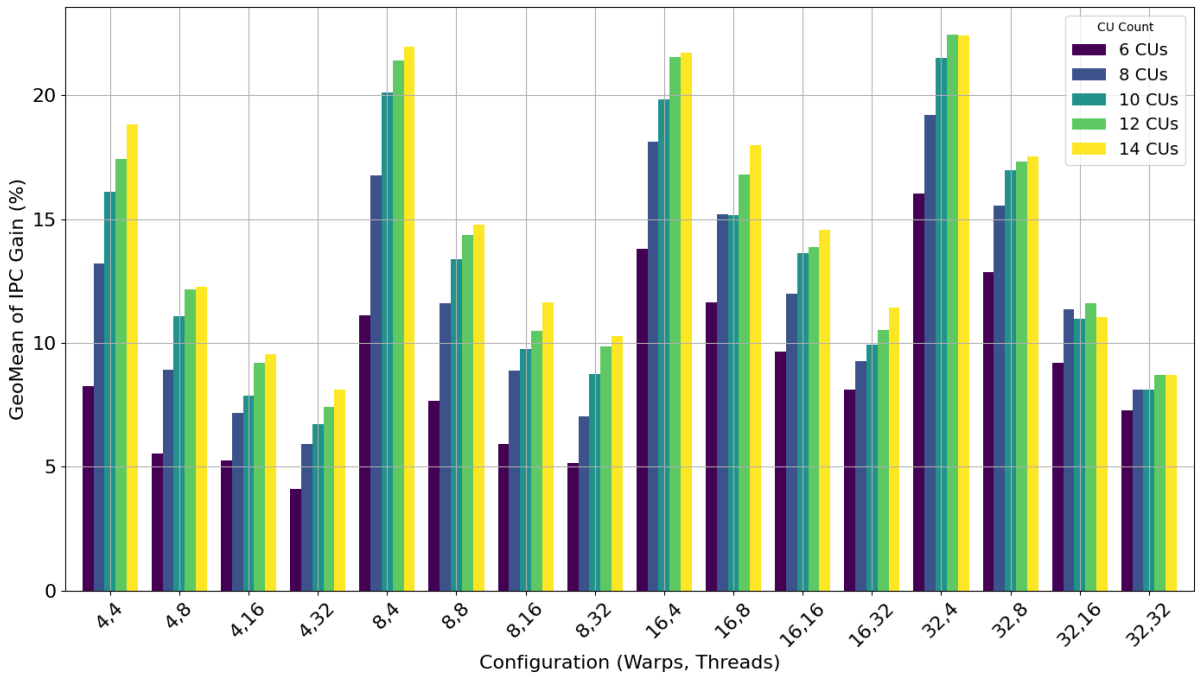


FIGURE 6.6: Geometric Mean of IPC (21 applications) per SM configuration (warps, threads) for different amounts of Collector Units in the RRS (factor 1.5) LOOG-Vortex microarchitecture.

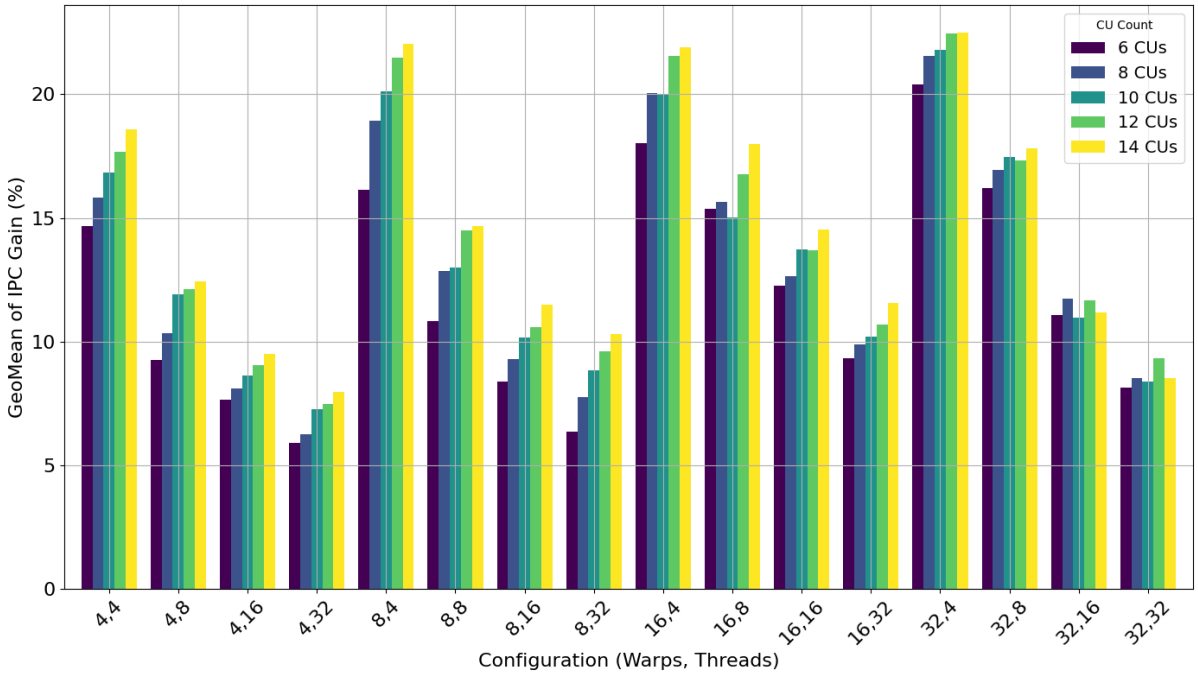


FIGURE 6.7: Geometric Mean of IPC (21 applications) per SM configuration (warps, threads) for different amounts of Collector Units in the RRS (factor 2) LOOG-Vortex microarchitecture.

Moreover, these results indicate that, compared to adding more CUs, expanding the RRS delivers better performance. An intriguing observation arises when comparing different RRS factors: for instance, an 8-CU configuration with an RRS factor of 1.5 (yielding 12 RRS entries) is outperformed by a 6-CU configuration with an RRS factor of 2 (also 12 RRS entries), despite the latter having fewer components. This discrepancy may be attributed to differences in execution reordering, which are influenced by varying levels of congestion and stalls in the CU operations.

We also computed the geometric mean of the IPC results for all configurations within each application cluster, as defined in Section 5.5. Figure 6.8, which presents results for an RRS factor of 1.5, shows that the application clusters exhibit a similar sensitivity to LOOG reordering as observed in the no-RRS design, but benefit from additional performance gains when the RRS is included.

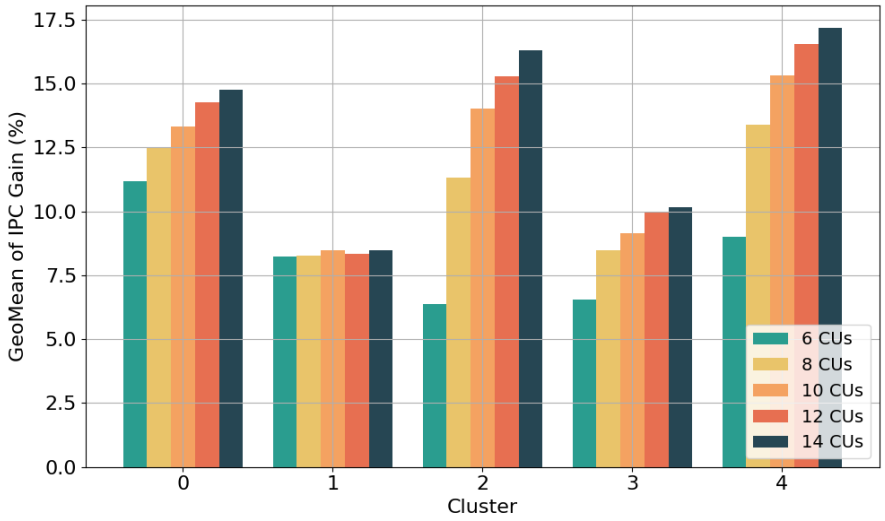


FIGURE 6.8: Geometric Mean of IPC (16 SM configurations) per Cluster of applications for different amounts of Collector Units in the RRS (factor 1.5) LOOG-Vortex microarchitecture.

Regarding Resource Utilization on the FPGA and Power Consumption, we again calculate the Area-Delay Product and Power-Delay Product metrics, now for the RRS LOOG-Vortex architecture, as presented in Figure 6.9 and Figure 6.10, respectively. The ADP results indicate that more than half of the configurations tested outperform the baseline, with the best performing configurations corresponding to those with lower Threads per Warp, as expected from our previous analysis. As for the PDP, all configurations deliver promising outcomes, demonstrating up to approximately 17% reduction in energy consumption.

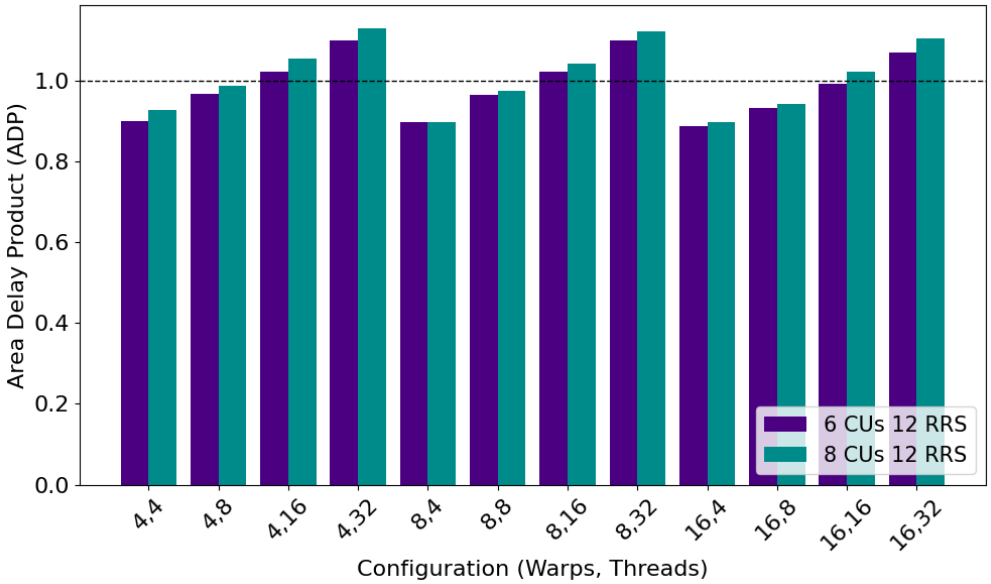


FIGURE 6.9: Area-Delay Product of LOOG-Vortex (with 12 RRS entries) per SM Configuration (Warps, Threads) for 6 and 8 CUs (normalized to baseline).

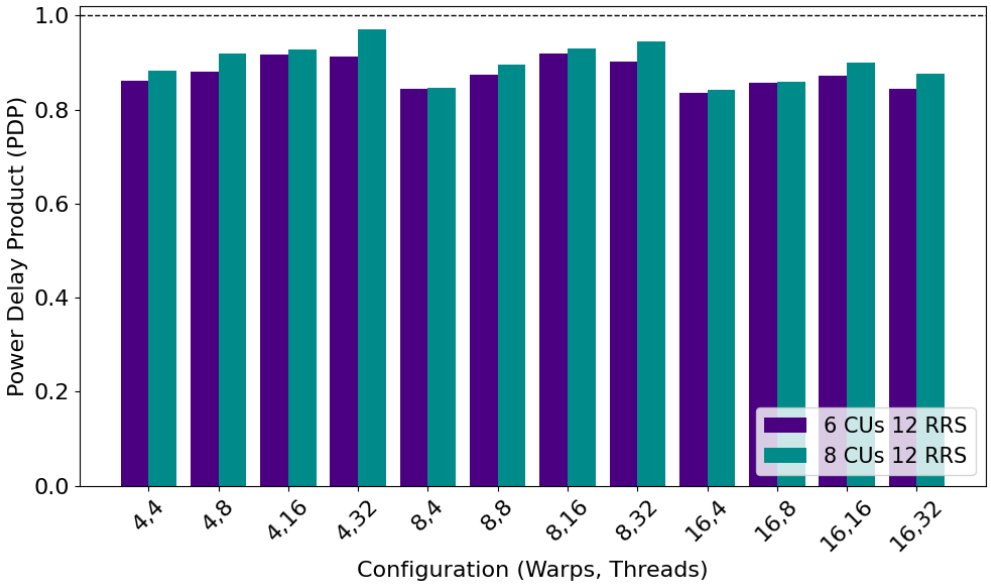


FIGURE 6.10: Power-Delay Product of LOOG-Vortex (with 12 RRS entries) per SM Configuration (Warps, Threads) for 6 and 8 CUs (normalized to baseline).

6.4 Instruction Level Parallelism Analysis

Finally, we examined ILP in our microbenchmarks by quantifying the reorder ”distances” during out-of-order executions. This metric represents the number of instructions between an executing instruction and a bypassed preceding instruction within

the same warp. The geometric mean of these distances of all applications and all configurations tested determined for the no-RRS LOOG-Vortex architecture is presented in Figure 6.11 and for the LOOG-Vortex architecture with an RRS factor of 1.5 in Figure 6.12. The findings clearly indicate that increasing the number of Collector Units (and RRS entries for the RRS configuration) enables a greater number of instructions to bypass their predecessors, thereby enhancing ILP and boosting overall GPU performance.

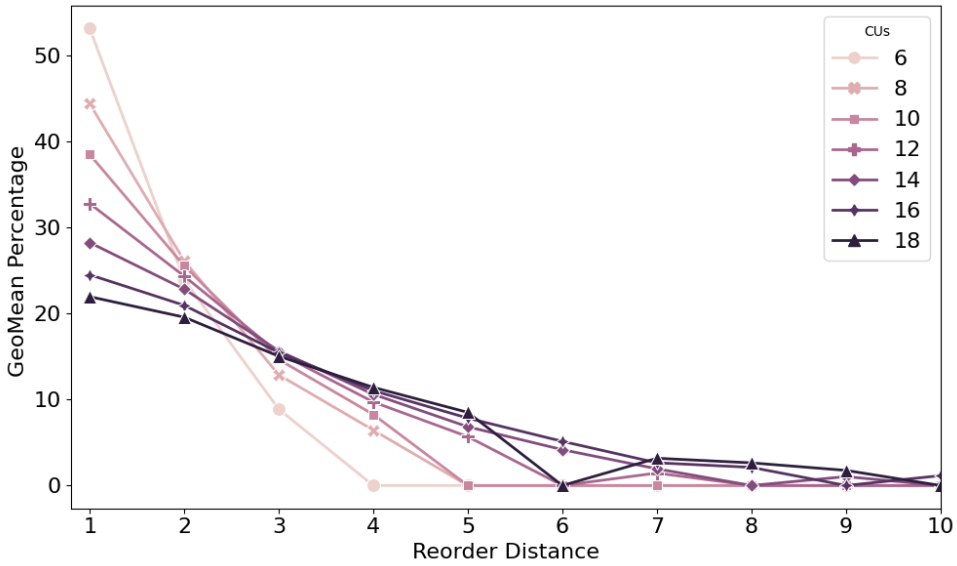


FIGURE 6.11: Reorder distances and their percentages normalized to the total instructions that were executed OOO for no-RRS LOOG-Vortex with different amounts of CUs.

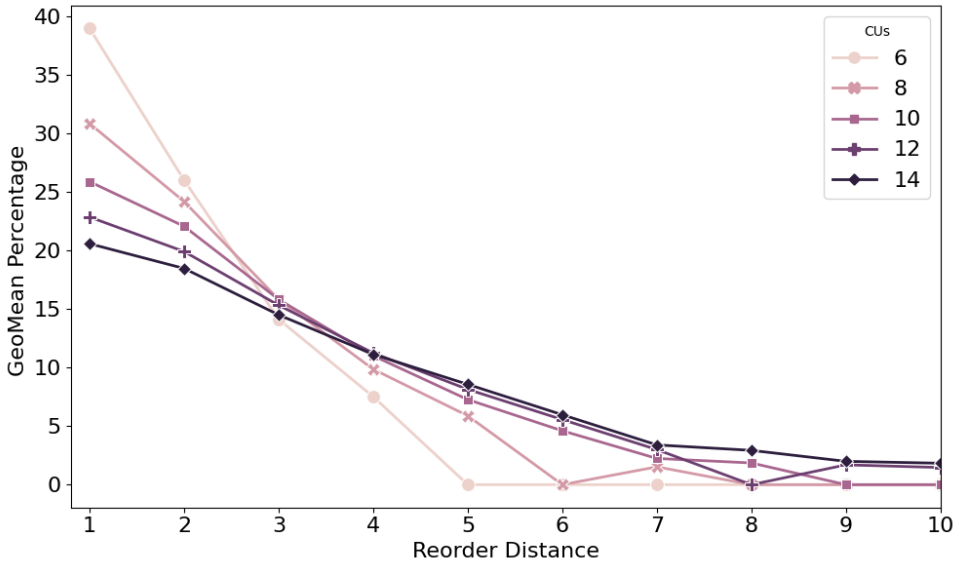


FIGURE 6.12: Reorder distances and their percentages normalized to the total instructions that were executed OOO for LOOG-Vortex with RRS (factor 1.5) with different amounts of CUs.

Chapter 7

Conclusions & Future Work

7.1 Conclusions

The implementation of the LOOG (Light-weight Out-Of-Order GPU) execution scheme in the Vortex GPU architecture, both with and without the Register Renaming Stack (RRS), has demonstrated substantial improvements in Performance with a low Area overhead and Power consumption when compared to the baseline in-order Vortex design. The LOOG scheme was implemented in SystemVerilog for Vortex GPU version 2.0, in accordance with the RISC-V based pipeline and special micro-architecture characteristics of the Vortex GPU framework.

In this study, we explored various aspects of the Vortex architecture to assess the effectiveness of LOOG, including the design trade-offs concerning Performance, Area, and routing. These include the insertion of registers to reduce connections between the Collector Units, which are the key components of the LOOG reordering scheme, the explotarion of different scheduling policies for RF read operations and the insertion of the Register Renaming Stack (Section 5.4). These optimizations were crucial in achieving a balanced design that not only enhances throughput but also minimizes resource consumption on the FPGA platform.

A key part of this work involved characterizing the Vortex workload in order to define LOOG-sensitive applications (Section 5.5). Specifically, we evaluated 21 of Vortex's micro-benchmarks based on their stall sources upon execution in the baseline configuration. This step was critical in identifying which applications are more sensitive to the LOOG-Vortex scheme, particularly the ones with low Memory accesses and more Compute-intensive workloads.

We also focused on the right-sizing of the LOOG-Vortex architecture, exploring a large design space that included various parameters such as Threads per Warp, Warps per Core, and the number of Collector Units and RRS entries. Through extensive experimentation and evaluation, we identified the optimal configurations that resulted in the best Performance, Area, and power efficiency trade-offs, which were the ones with low Threads per Warp and moderate CUs and RRS entries.

The experimental evaluation of different configurations—in-order, LOOG, and RRS-LOOG Vortex—showed significant Performance improvements (Chapter 6). Average Performance gains of up to 23.5% were achieved, with the Area overhead ranging between 4.5-27.5%, depending on the configuration, and approximately 5% for the best configurations. Specifically, configurations such as 8 CUs with 16 Warps and 4 Threads per Warp demonstrated an Area overhead of 5% for the no-RRS configuration and 4.5% for the 12 RRS configuration. Furthermore, the Power-Delay Product (PDP) showed up to approximately 17% reduction, indicating the Energy efficiency of the RRS-LOOG approach in comparison to the baseline in-order execution.

7.2 Future Work

While the current work provides valuable insights into the optimization of LOOG execution in the Vortex GPU, several fields remain for future exploration and enhancement. One such field is accommodating larger OpenCL benchmark suites, such as Rodinia, to better understand how LOOG-Vortex handles a broader range of real-world applications. Expanding to these larger suites would help further refine the architecture and ensure that it can scale effectively across different types of workloads and benchmark environments.

Another important avenue for future work is to evaluate the LOOG-Vortex design on larger FPGA platforms for varying numbers of GPU cores and sockets. This would enable us to assess how the architecture performs as the system scales, particularly when dealing with more complex configurations and increased resource demands.

Additionally, there is significant potential to explore more sophisticated scheduling policies for Register File (RF) reads and warp instruction scheduling. Another key design improvement would be to make the Register File multi-banked, which could reduce stalls in the Operand Collect stage. By enabling concurrent access to different banks of the RF, we can further reduce Collector Units contention and improve overall throughput.

Moreover, it would be valuable to investigate more advanced, yet lightweight, Load / Store instructions reordering mechanisms. These could improve memory access patterns, ensuring that address data dependencies are managed more efficiently, leading to faster execution without introducing significant hardware overhead, thus reducing the latency of Memory instructions that have not yet been reordered.

Finally, implementing and comparing different Out-Of-Order GPU execution schemes proposed by the research community (like the ones mentioned in Chapter 4) would provide valuable insights into alternative approaches for improving Instruction-Level Parallelism in GPUs. This would broaden our understanding of the trade-offs between

various OOO schemes, helping to inform the design of future GPU architectures, particularly in terms of hardware complexity, performance gains, and energy efficiency.

Appendix A

Source Code

The source code for the implementation elaborated on in Chapter 5 can be found at:
https://github.com/mariazerva/diploma/tree/loogvortex

Bibliography

[1] R.R. Schaller. "Moore's law: past, present and future". In: *IEEE Spectrum* 34.6 (1997), pp. 52–59. DOI: 10.1109/6.591665.

[2] R.H. Dennard et al. "Design of ion-implanted MOSFET's with very small physical dimensions". In: *IEEE Journal of Solid-State Circuits* 9.5 (1974), pp. 256–268. DOI: 10.1109/JSSC.1974.1050511.

[3] "Post-Dennard Scaling and the final Years of Moore ' s Law Consequences for the Evolution of Multicore-Architectures". In: 2014. URL: https://api.semanticscholar.org/CorpusID:139088320.

[4] Viet Bui et al. "Heterogeneous Computing and The Real-World Applications". In: Dec. 2021, pp. 0747–0751. DOI: 10.1109/UEMCON53757.2021.9666740.

[5] Marko Misic, Đorđe Đurđević, and Milo Tomasevic. "Evolution and trends in GPU computing". In: Jan. 2012, pp. 289–294. ISBN: 978-1-4673-2577-6.

[6] Ian Kuon and Jonathan Rose. "Measuring the Gap Between FPGAs and ASICs". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 26.2 (2007), pp. 203–215. DOI: 10.1109/TCAD.2006.884574.

[7] Juergen Ributzka et al. "DEEP: an iterative fpga-based many-core emulation system for chip verification and architecture research". In: *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA '11. Monterey, CA, USA: Association for Computing Machinery, 2011, 115–118. ISBN: 9781450305549. DOI: 10.1145/1950413.1950438. URL: https://doi.org/10.1145/1950413.1950438.

[8] 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 (2019), pp. 166–169. DOI: 10.1109/LCA.2019.2951161.

[9] 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 (2022), pp. 388–402. DOI: 10.1109/TPDS.2021.3093231.

[10] 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)*. 2020, pp. 473–486. DOI: 10.1109/ISCA45697.2020.00047.

[11] Blaise Tine et al. *Vortex: Extending the RISC-V ISA for GPGPU and 3D-GraphicsResearch*. 2021. arXiv: 2110.10857 [cs.AR]. URL: https://arxiv.org/abs/2110.10857.

[12] Fares Elsabbagh et al. "Vortex: OpenCL Compatible RISC-V GPGPU". In: *CoRR* abs/2002.12151 (2020). arXiv: 2002.12151. URL: https://arxiv.org/abs/2002.12151.

[13] Cristobal Navarro, Nancy Hitschfeld, and Luis Mateu. "A Survey on Parallel Computing and its Applications in Data-Parallel Problems Using GPU Architectures". In: *Communications in Computational Physics* 15 (Sept. 2013), pp. 285–329. DOI: 10.4208/cicp.110113.010813a.

[14] Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Education Group, 2003. ISBN: 0071232656.

[15] Barry Wilkinson and Michael Allen. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers (2nd Edition)*. USA: Prentice-Hall, Inc., 2004. ISBN: 0131405632.

[16] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*. 5th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013. ISBN: 0124077269.

[17] C. V. Ramamoorthy and H. F. Li. "Pipeline Architecture". In: *ACM Comput. Surv.* 9.1 (Mar. 1977), 61–102. ISSN: 0360-0300. DOI: 10.1145/356683.356687. URL: https://doi.org/10.1145/356683.356687.

[18] Sergey Berezin et al. "Verification of Out-Of-Order Processor Designs Using Model Checking and a Light-Weight Completion Function". In: *Form. Methods Syst. Des.* 20.2 (Mar. 2002), 159–186. ISSN: 0925-9856. DOI: 10.1023/A:1014170513439. URL: https://doi.org/10.1023/A:1014170513439.

[19] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. 5th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. ISBN: 012383872X.

[20] David Luebke et al. "GPGPU: General purpose computation on graphics hardware". In: Aug. 2004. DOI: 10.1145/1103900.1103933.

[21] Tor M. Aamodt, Wilson Wai Lun Fung, and Timothy G. Rogers. "Introduction". In: *General-Purpose Graphics Processor Architectures*. Cham: Springer International Publishing, 2018, pp. 1–7. ISBN: 978-3-031-01759-9. DOI: 10.1007/978-3-031-01759-91. URL: https://doi.org/10.1007/978-3-031-01759-91.

[22] Hyeran Jeon. "GPU Architecture". In: *Handbook of Computer Architecture*. Ed. by Anupam Chattopadhyay. Singapore: Springer Nature Singapore, 2022, pp. 1–29. ISBN: 978-981-15-6401-7. DOI: 10.1007/978-981-15-6401-766-1. URL: https://doi.org/10.1007/978-981-15-6401-766-1.

[23] Veynu Narasiman et al. "Improving GPU Performance via Large Warps and Two-Level Warp Scheduling". In: Dec. 2011.

[24] Teng Li, Vikram K. Narayana, and Tarek El-Ghazawi. "Exploring Graphics Processing Unit (GPU) Resource Sharing Efficiency for High Performance Computing". In: *Computers* 2.4 (2013), pp. 176–214. ISSN: 2073-431X. DOI: 10.3390/computers2040176. URL: https://www.mdpi.com/2073-431X/2/4/176.

[25] Jonathan Rose, Abbas El Gamal, and Alberto Vincentelli. "Architecture of field-programmable gate arrays". In: *Proceedings of the IEEE* 81 (Aug. 1993), pp. 1013 –1029. DOI: 10.1109/5.231340.

[26] Sapan Garg Sanjay Churiwala. *Principles of VLSI RTL Design*. Springer New York, NY, 2011. DOI: https://doi.org/10.1007/978-1-4419-9296-3.

[27] Gregory D. Peterson. "Performance Tradeoffs for Emulation, Hardware Acceleration, and Simulation". In: *System-on-Chip Methodologies & Design Languages*. Ed. by Peter J. Ashenden, Jean P. Mermet, and Ralf Seepold. Boston, MA: Springer US, 2001, pp. 255–267. ISBN: 978-1-4757-3281-8. DOI: 10.1007/978-1-4757-3281-822. URL: https://doi.org/10.1007/978-1-4757-3281-822.

[28] 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.

[29] Krishna Yalamarthy Hyesoon Kim Blaise Tine Fares Elsabbagh. *Vortex GPU 2.0*. GitHub repository "vortexgpgpu / vortex "; accessed March 13, 2025. URL: https://github.com/vortexgpgpu/vortex/tree/master.

[30] Fares Elsabbagh et al. *Vortex: OpenCL Compatible RISC-V GPGPU*. 2020. arXiv: 2002.12151 [cs.DC]. URL: https://arxiv.org/abs/2002.12151.

[31] Erik Lindholm et al. "NVIDIA Tesla: A Unified Graphics and Computing Architecture". In: *IEEE Micro* 28.2 (2008), pp. 39–55. DOI: 10.1109/MM.2008.31.

[32] J. Kruger and R. Westermann. "Acceleration techniques for GPU-based volume rendering". In: *IEEE Visualization, 2003. VIS 2003.* 2003, pp. 287–292. DOI: 10.1109/VISUAL.2003.1250384.

[33] D. W. Anderson, F. J. Sparacio, and R. M. Tomasulo. "The IBM System/360 Model 91: Machine Philosophy and Instruction-Handling". In: *IBM Journal of Research and Development* 11.1 (1967), pp. 8–24. DOI: 10.1147/rd.111.0008.

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

[35] J.P. Shen and M.H. Lipasti. *Modern Processor Design: Fundamentals of Superscalar Processors*. McGraw-Hill, 2003. ISBN: 9780071230070. URL: https://books.google.gr/books?id=VIWLAAAACAAJ.

[36] Ali Bakhoda et al. "Analyzing CUDA workloads using a detailed GPU simulator". In: *2009 IEEE International Symposium on Performance Analysis of Systems and Software*. 2009, pp. 163–174. DOI: 10.1109/ISPASS.2009.4919648.

[37] Vijay Kandiah et al. "AccelWattch: A Power Modeling Framework for Modern GPUs". In: Oct. 2021, pp. 738–753. DOI: 10.1145/3466752.3480063.

[38] Raghuraman Balasubramanian et al. "MIAOW - An open source RTL implementation of a GPGPU". In: *2015 IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS XVIII)*. 2015, pp. 1–3. DOI: 10.1109/CoolChips.2015.7158663.

[39] Raghuraman Balasubramanian et al. "Enabling GPGPU Low-Level Hardware Explorations with MIAOW: An Open-Source RTL Implementation of a GPGPU". In: *ACM Trans. Archit. Code Optim.* 12.2 (June 2015). ISSN: 1544-3566. DOI: 10.1145/2764908. URL: https://doi.org/10.1145/2764908.

[40] Ishita Chaturvedi et al. "GhOST: a GPU Out-of-Order Scheduling Technique for Stall Reduction". In: *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 2024, pp. 1–16. DOI: 10.1109/ISCA59077.2024.00011.

[41] Rodrigo Huerta et al. "SIMIL: SIMple Issue Logic for GPUs". In: *Microprocessors and Microsystems* 111 (2024), p. 105105. ISSN: 0141-9331. DOI: https://doi.org/10.1016/j.micpro.2024.105105. URL: https://www.sciencedirect.com/science/article/pii/S0141933124001005.

[42] Reoma Matsuo et al. “TURBULENCE: Complexity-effective Out-of-order Execution on GPU with Distance-based ISA”. In: *2023 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2023, pp. 1–2. DOI: 10.23919/DATE56975.2023.10137216.