ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Exploring Core Heterogeneity for GPU Accelerators

Εξερευνώντας την Ετερογένεια Πυρήνων για επιταχυντές GPU

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Αλέξανδρου Π. Μοίρα

Επιβλέπων: Δημήτριος Ι. Σούντρης
Καθηγητής Ε.Μ.Π.

Αθήνα, Απρίλιος 2024

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΜΙΚΡΟΫΠΟΛΟΓΙΣΤΩΝ ΚΑΙ
ΨΗΦΙΑΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Exploring Core Heterogeneity for GPU Accelerators

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Αλέξανδρου Π. Μοίρα

Επιβλέπων: Δημήτριος Ι. Σούντρης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 4η Απριλίου 2024.

.....	.....	.....
Δημήτριος Σούντρης	Παναγιώτης Τσανάκας	Σωτήριος Ξύδης
Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Επίκουρος Καθηγητής Ε.Μ.Π.

Αθήνα, Απρίλιος 2024

.....

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

Acknowledgements

I would like to thank my supervising professor, Professor Dimitrios Soudris for providing me with the opportunity to carry out my diploma thesis at Microlab at the School of Electrical and Computer Engineering of the National Technical University of Athens.

In addition i would like to thank Professor Sotirios Xydis for his invaluable observations and suggestions as well as the knowledge and expertise he shared with me throughout the execution of this project.

I would also like to extend my gratitude and appreciation to Dr. Konstantinos Iliakis for his continuous guidance, and support in overcoming any problems that arose throughout this work, in addition to his advice regarding research in general.

Finally i would like to thank my family and friends for their patience and unwavering support while carrying out this project.

Abstract

Οι GPU, που αρχικά προτάθηκαν αποκλειστικά ως επιταχυντές επεξεργασίας γραφικών, βρίσκουν πλέον εφαρμογή σε ένα συνεχώς αυξανόμενο εύρος τομέων. Η ανάλυση μιας πληθώρας εφαρμογών GPU αποκάλυψε ότι οι πόροι της συχνά δεν αξιοποιούνται στο έπακρο. Οι ομοιογενείς επιταχυντές υλικού δεν μπορούν να ανταπεξέλθουν στη μεγάλη ποικιλία υπολογιστικών απαιτήσεων των εφαρμογών GPU. Ιστορικά, οι πολυπύρηνοι CPU αντιμετώπισαν το αντίστοιχο ζήτημα εισάγοντας ετερογένεια στο εσωτερικό του ολοκληρωμένου σε επίπεδο πυρήνα. Αυτή η διαμόρφωση παρέχει έναν πρόσθετο βαθμό ευελιξίας, επιτρέποντας στις εφαρμογές να εκτελούνται στο υλικό που ταιριάζει καλύτερα στις απαιτήσεις τους. Τέτοιες αρχιτεκτονικές έχουν ήδη υλοποιηθεί και έχουν κερδίσει δημοτικότητα στον τομέα των CPU, όπου παρέχουν καλύτερη ενεργειακή απόδοση χωρίς να θυσιάζουν τις επιδόσεις.

Με κίνητρο τις προαναφερθείσες παρατηρήσεις, η παρούσα εργασία επεκτείνει τον σύγχρονο προσομοιωτή GPU ώστε να υποστηρίζει ετερογενείς πυρήνες εντός του ολοκληρωμένου GPU. Η προτεινόμενη έκδοση παρέχει υποστήριξη για την προσαρμογή της συντριπτικής πλειοψηφίας του pipeline που ήδη παρήχε ο αρχικός προσομοιωτής μεταξύ των SM ξεκλειδώνοντας έναν πολύ ευρύ χώρο σχεδίασης. Τροποποιήσεις εφαρμόζονται επίσης στο αντίστοιχο μοντέλο ισχύος ώστε να προσομοιώνει τις ετερογενείς αρχιτεκτονικές. Η επέκταση του προσομοιωτή συνδυάζεται με μια επέκταση του API η οποία εκθέτει στον προγραμματιστή τον έλεγχο όσον αφορά την προτιμώμενη διαμόρφωση SM για την εκτέλεση κάθε πυρήνα (kernel). Επιπλέον, ο δυνητικός αντίκτυπος των ετερογενών αρχιτεκτονικών GPU καταδεικνύεται μέσω μιας λεπτομερούς μελέτης περίπτωσης που στοχεύει στη συνεγκατάσταση kernels ευαίσθητων σε υπολογιστικούς πόρους και μη ευαίσθητων που ανήκουν στον τομέα των υπολογιστών υψηλών επιδόσεων. Τέλος, η προτεινόμενη αρχιτεκτονική GPU αξιολογείται έναντι εναλλακτικών, ομοιογενών διαμορφώσεων GPU, καταδεικνύοντας μέση επιτάχυνση 24,1% συνοδευόμενη από κέρδος ενέργειας 2,6% σε ένα σύνολο 50 διαφορετικών συνδυασμών εφαρμογών.

Λέξεις Κλειδιά: GPU, Επιτάχυνση, Ετερογένεια, Προσομοίωση, AccelSim, AccelWattch

Abstract

Initially proposed as dedicated graphics processing accelerators, GPUs now find applicability in an ever-growing range of domains, including machine learning, high performance computing, automotive, and bioinformatics. Analyzing a plethora of GPU workloads has revealed that GPU resources are often underutilized. Evidently, homogeneous hardware accelerators cannot cope with the vast variety of computing requirements dictated by GPU application domains. Historically, multi-core CPUs have addressed a similar issue by introducing heterogeneity within the chip in the form of heterogeneous cores. This provides an additional degree of flexibility, allowing applications to execute on the hardware that best fits their demands. Such architectures have already been implemented and gained popularity in the domain of CPUs where they provide better power efficiency without sacrificing performance.

Motivated by the aforementioned observations, this paper extends the state-of-the-art, cycle-accurate GPU simulator to support heterogeneous Streaming Multiprocessors (SMs) within the GPU chip. The proposed version provides support for tweaking the vast majority of the pipeline the original simulator already provided between the SMs unlocking a very wide design space. Modifications are also applied to the corresponding power model to simulate heterogeneous architectures. The simulator extension is coupled with an API extension which exposes programmatic control over the preferable SM type for executing each kernel. Additionally, the potential impact of heterogeneous GPU architectures on specialized domains is demonstrated through a detailed case study targeting the co-location of resource-sensitive and -insensitive applications belonging to the High Performance Computing domain. Finally, the heterogeneous GPU architecture is evaluated against homogeneous GPU baselines, demonstrating a 24.1% average speedup accompanied by a 2.6% energy gain across a set of 50 different application combinations.

Keywords: GPU, Acceleration, Heterogeneity, High Performance Computing, Simulation, Accel-Sim, AccelWattch

Contents

List of Figures

List of Tables

Chapter 0

Εκτεταμένη Ελληνική Περίληψη

0.1 Εισαγωγή

0.1.1 Εισαγωγή στις GPUs ως GPU γενικού σκοπού

Οι Μονάδες Επεξεργασίας Γραφικών (GPUs) είναι παράλληλες συσκευές με πολλαπλά νήματα που αρχικά αποσκοπούσαν στην επιτάχυνση της απόδοσης γραφικών. Αυτός ήταν ο κύριος σκοπός τους για πολλά χρόνια, τον οποίο υπηρέτησαν, και εξακολουθούν να υπηρετούν, χρησιμοποιώντας τις δυνατότητες μαζικής παράλληλης εκτέλεσης για την απόδοση 2D εικόνων προς εμφάνιση στην οθόνη. [81].

Ωστόσο, το εύρος εφαρμογής τους έχει αυξηθεί δραστικά δημιουργώντας έναν ολόκληρο νέο τομέα των GPU γενικού σκοπού (GPGPUs) που επιτρέπει την εκμετάλλευση των πλεονεκτημάτων απόδοσης που επιτυγχάνονται από τον προαναφερθέντα παραλληλισμό από ένα μεγάλο εύρος μη γραφικών εφαρμογών. Οι εν λόγω εφαρμογές αξιοποιούν τις GPU αναθέτοντας χρονοβόρα τμήματα της εκτέλεσής τους σε πυρήνες (kernels) και εκφορτώνοντάς τους στη GPU που χρησιμοποιείται ως επιταχυντής. Αυτοί οι kernels εν συνεχεία θα δημιουργήσουν πολυάριθμα νήματα ομαδοποιημένα σε μπλοκ, τα οποία θα προγραμματιστούν στις υπολογιστικές μηχανές της GPU, τους πολυεπεξεργαστές ροών (SM) σε ορολογία NVIDIA, για να χωριστούν περαιτέρω σε ομάδες νημάτων και να εκτελεστούν ταυτόχρονα από τους υπολογιστικούς αγωγούς (pipelines) που υπάρχουν στο SM. Οι ομάδες νημάτων θα εκκενώσουν τον αγωγό εκτέλεσης όταν συναντήσουν μια καθυστέρηση, όπως μια μεγάλης διάρκειας λειτουργία μνήμης, επιτρέποντας στην εκτέλεση άλλων να καλύψει την καθυστέρηση αυτή με αποτέλεσμα τη σταθερή υψηλή χρήση του υλικού.

0.1.2 Διαφορετικότητα απαιτήσεων υλικού GPU

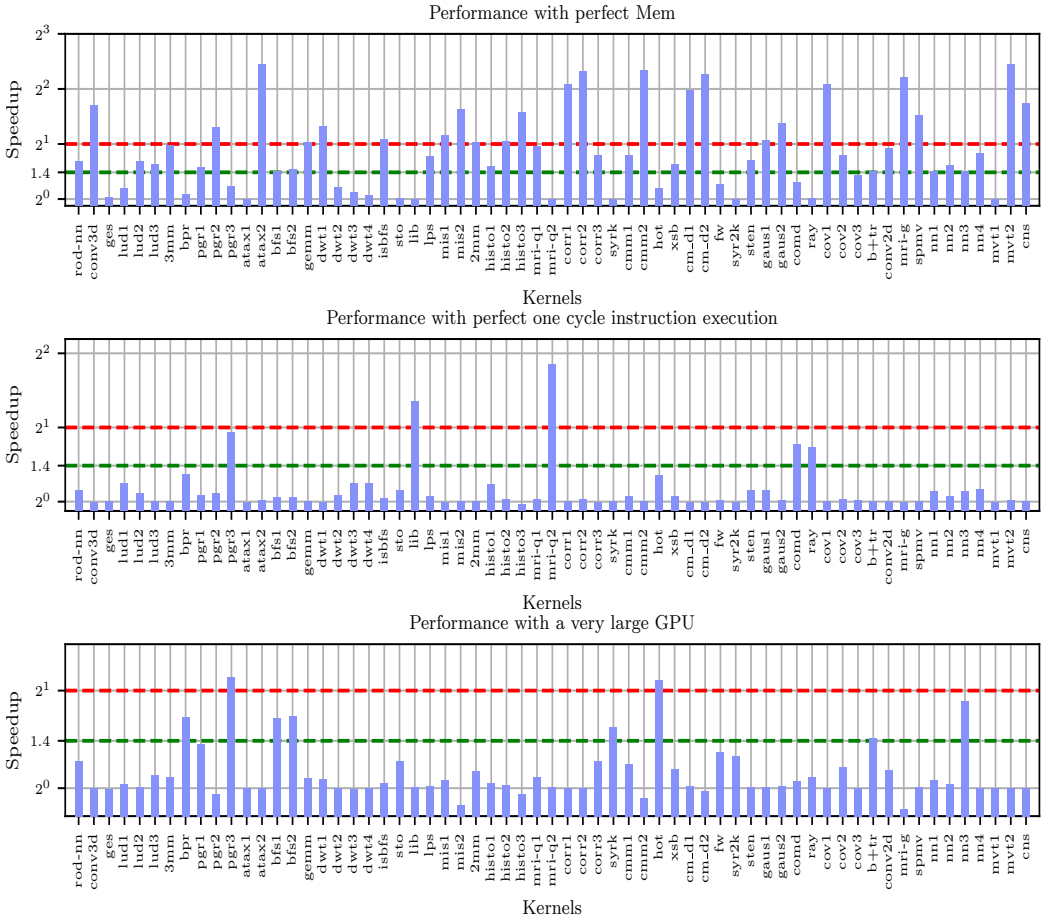


Figure 1: Αύξηση απόδοσης kernels με διαφορετικές διαμορφώσεις. Η διπλάσια επιτάχυνση (κόκκινη γραμμή) θεωρείται πολύ σημαντική, ενώ το κατώτερο όριο βελτίωσης με αυτές τις επιθετικές διαμορφώσεις ορίζεται στο 40% (πράσινη γραμμή) κατά την αξιολόγηση των σημείων συμφόρησης (bottlenecks) κάθε kernel.

Καθώς η χρήση της GPU υιοθετείται όλο και περισσότερο για μια ευρεία ποικιλία εφαρμογών, ο συνεχώς αυξανόμενος όγκος των εργασιών που μεταφορτώνονται σε αυτήν, σε συνδυασμό με την ποικιλομορφία στα χαρακτηριστικά εκτέλεσης των kernels, δεν επιτρέπει σε μια μόνο διαμόρφωση υλικού να εκτελεί αποτελεσματικά κάθε kernel που της παρουσιάζεται. Συχνά ένας kernel αποτυγχάνει να επιτύχει την πλήρη αξιοποίηση της κάρτας λόγω stalls που δεν μπορούν να κρυφτούν με την έκδοση εντολών από διαφορετική ομάδα νημάτων. Τέτοια αδιέξοδα μπορεί να εμφανιστούν σε kernels όπου το σημαν-

Τέλεια Μνήμη	Τέλεια Εκτέλεση	Μεγάλη GPU
όλες οι αστοχίες L1 εξυπηρετούνται σε έναν κύκλο από το υποσύστημα μνήμης, περιοριζόμενες μόνο από τη συμφόρηση στο δίαυλο μνήμης	όλες οι καθυστερήσεις έναρξης και εκτέλεσης εντολών ορίζονται σε έναν κύκλο	5×μονάδες εκτέλεσης, τράπεζες αρχείων καταχωρητών, χρονοπρογραμματιστές στρεβλώσεων (warp schedulers), 2×κοινόχρηστες τράπεζες μνήμης, 10×μονάδες συλλογής τελεστών, 1,5×SMs

Table 1: Διαμορφώσεις που χρησιμοποιούνται για την αναγνώριση των διαφορετικών αναγκών των GPU kernels

τικότερο ποσοστό των ομάδων νημάτων τους τους εκδίδει εντολές μνήμης δημιουργώντας ταυτόχρονα υψηλό ανταγωνισμό για τον αργό πόρο. Όσο περιμένουν τα αποτελέσματα από την αργή μνήμη οι υπολογιστικές μονάδες δεν έχουν διαθέσιμα νήματα να εκτελέσουν. Αντίστοιχα, για βαρύτερους φόρτους εργασίας, οι kernels μπορεί να περιορίζονται από το μέγεθος του pipeline με αποτέλεσμα τα υποψήφια για αποστολή νήματα να αντιμετωπίζουν δομικούς κινδύνους.

Το pipeline θα μπορούσε να παραμείνει αδρανές και στην περίπτωση των kernels που επιβάλλουν χαμηλή πληρότητα GPU, είτε επειδή εκδίδουν χαμηλό αριθμό νημάτων, είτε επειδή δεν μπορούν να γεμίσουν τα SMs στο όριο των νημάτων τους λόγω περιορισμών υλικού, με αποτέλεσμα να προκύπτουν εντελώς άδεια SMs ή SMs των οποίων οι χρονοπρογραμματιστές warp δεν έχουν αρκετά μεγάλη δεξαμενή warps για να επιλέξουν. Όλες οι παραπάνω μορφές υποαπασχόλησης οδηγούν σε ένα σημείο συμφόρησης που τελικά καθορίζει την απόδοση του kernel. Μια ανάλυση 63 kernels GPGPU που περιλαμβάνονται στο πακέτο του Accel-Sim [67] εκτελώντας τους στις τρεις διαμορφώσεις που εξηγούνται στον πίνακα 1 επιβεβαιώνει αυτές τις υποθέσεις. Όπως φαίνεται στο Σχήμα 1 23 από τους 63 kernels διπλασιάζουν την απόδοσή τους με ένα τέλειο σύστημα μνήμης και επιπλέον 21 παρουσιάζουν τουλάχιστον 40% επιτάχυνση σε σύγκριση με την εκτέλεση στη βασική αρχιτεκτονική Volta [97] NVIDIA V100. Αυτό είναι ενδεικτικό συμφόρησης μνήμης για τις εφαρμογές αυτές που περιορίζουν τη χρήση του αριθμητικού αγωγού επιβεβαιώνοντας τις παρατηρήσεις των Adwait Jog et al. [60]. Πέντε άλλοι kernels παρουσιάζουν βελτίωση με τέλειες μονάδες εκτέλεσης. Είναι σημαντικό ότι αυτοί οι kernels δεν συμπίπτουν με τους kernels που βελτιώθηκαν μέσω της χρήσης του τέλειου

υποσυστήματος μνήμης, οπότε φαίνεται να καθυστερούν κυρίως περιμένοντας κινδύνους δεδομένων λόγω των αποτελεσμάτων εκτέλεσης που δεν είναι έτοιμα εγκαίρως και όχι λόγω των αργών αποκρίσεων της μνήμης. Η εκτέλεση των ίδιων kernels σε μία μη ρεαλιστικά μεγαλύερη V100 με 40 περισσότερα SM και κλιμακωμένο pipeline εκτέλεσης εντός αυτών οφελεί οκτώ kernels, υποδεικνύοντας ότι παρουσιάζουν περισσότερο, ανεκμετάλλευτο παραλληλισμό σε επίπεδο νήματος (TLP), εκ των οποίων τρεις δεν βελτιώνονται ούτε με τέλεια μνήμη ούτε με τέλειες εντολές. Κανένας kernel δεν βελτιώνεται και από τα τρία διαφορετικά μοντέλα. Επιπλέον, τουλάχιστον 6 kernels παρουσιάζουν επιβράδυνση όταν εκτελούνται στη μεγαλύτερη V100, γεγονός που δείχνει ότι μερικές φορές ακόμη και περισσότεροι πόροι μπορούν να βλάψουν την απόδοση.

0.1.3 Βήματα προς την εξειδίκευση

Αυτές οι παρατηρήσεις οδήγησαν σε πολλές εξελίξεις σε υλικό όσον αφορά τις τεχνικές εξειδίκευσης πολλές από τις οποίες έχουν ήδη υλοποιηθεί.

Tensor cores προστέθηκαν στις GPUs [96], από την NVIDIA σε μια προσπάθεια να βελτιωθεί η απόδοση των εφαρμογών που βασίζονται σε βαριές υπολογιστικές πράξεις πολλαπλασιασμού πινάκων, όπως η εκπαίδευση και η εξαγωγή συμπερασμάτων (inference) μηχανικής μάθησης. Kernels που δε βασίζονται σε πράξεις πινάκων δε μπορούν ανα αξιοποιήσουν αυτές τις μονάδες. Η εξειδίκευση όσον αφορά το ML έχει ξεφύγει από τα όρια των GPUs με διάφορους επιταχυντές Deep Learning να εμφανίζονται χρησιμοποιώντας χαμηλής ακρίβειας αριθμητικής τανυστές καθώς και μνήμη πολύ υψηλού εύρους ζώνης και να υλοποιούνται είτε σε FPGAs [118] είτε ως ASICS [47]. Τα Tensor Processing Units (TPUs) [62] της Google είναι ένα χαρακτηριστικό παράδειγμα επιταχυντή υλοποιημένου με ASIC. Αν και τέτοιες λύσεις βελτιώνουν την απόδοση του ML, χάνουν τη γενικότητα που προσφέρουν οι συσκευές GPU.

Στη βιβλιογραφία η **Light-Weight Out-Of-Order Execution (LOOG)** [51] βελτιώνει τις επιδόσεις των kernels που αντιμετωπίζουν μακρά stalls εισάγοντας παραλληλισμό σε επίπεδο εντολών για να μετριάσει τον μη παραγωγικό χρόνο εκτέλεσης όταν τα περισσότερα warps είναι αδρανή περιμένοντας μια λειτουργία μεγάλης καθυστέρησης από όπου και αν προέρχεται αυτή. Παρόλο που τα κέρδη απόδοσης είναι σημαντικά όταν στοχεύουν σε τέτοιους kernels, το πλεονέκτημα έναντι των παραδοσιακών GPU είναι πιο περιορισμένο κατά την εκτέλεση υπολογιστικά βαρέων φορτίων.

Η **R-GPU** [13] επιχειρεί επίσης να βελτιώσει τη χρήση για kernels που αντιμετωπίζουν αδιέξοδα λόγω λειτουργιών μνήμης υψηλής καθυστέρησης. Αυτό το επιτυγχάνει με την αναδιαμόρφωση των SMs της σε εξειδικευμένες λειτουργικές μονάδες που διασυνδέονται για την εκτέλεση μιας συγκεκριμένης λειτουργίας. Παρόλα αυτά, όπως σημειώνουν οι συγγραφείς, αυτή η αρχιτεκ-

τονική δεν θα αποφέρει κανένα όφελος για kernels που δεσμεύονται από υπολογισμούς.

0.1.4 Ταυτόχρονη Εκτέλεση

Οι τρέχουσες τάσεις των GPU υποστηρίζουν την ταυτόχρονη εκτέλεση περισσότερων του ενός kernels στην ίδια κάρτα προκειμένου να βελτιωθεί η αξιοποίησή της [130, 57, 16, 111]. Η ταυτόχρονη εκτέλεση kernels υποστηρίζεται επίσημα από την NVIDIA, αρχής γενομένης από την αρχιτεκτονική Fermi [99]. Ωστόσο, ο διαμοιρασμός στην τρέχουσα μορφή του βλέπει τη χρησιμοποίηση σε επίπεδο SM που απαιτεί ο πρώτος από μια ακολουθία kernels να αδυνατεί να γεμίσει το σύνολο των SM, προκειμένου να εκτελεστεί ο επόμενος.

Η προσθήκη εξειδικευμένου υλικού για πολλαπλούς τύπους εφαρμογών θα απαιτούσε να είναι ενεργές περισσότερες από μία από αυτές ταυτόχρονα, ώστε να μην μένουν τμήματα του πυριτίου ανεκμετάλλευτα. Οι τρέχουσες τάσεις των GPU υποστηρίζουν την ταυτόχρονη εκτέλεση περισσότερων του ενός kernels στην ίδια κάρτα προκειμένου να βελτιωθεί η αξιοποίησή της [130, 57, 16, 111, 2]. Η ταυτόχρονη εκτέλεση kernels υποστηρίζεται επίσημα από την NVIDIA, ξεκινώντας από την αρχιτεκτονική Fermi [99] μέσω της χρήσης διαφόρων τεχνολογιών [104, 90, 94]. Τα συστήματα ταυτόχρονης εκτέλεσης θα συζητηθούν λεπτομερέστερα στην ενότητα 3.6. Ωστόσο, ο διαμοιρασμός στην τρέχουσα μορφή του απαιτεί ο πρώτος από μια ακολουθία kernels να μην είναι σε θέση να γεμίσει το σύνολο των SMs προκειμένου να εκδοθεί ο επόμενος στην περίπτωση των streams ή του MPS. Η χωρητικότητα του SM σε αυτό το πλαίσιο δεν μετριέται μόνο σε νήματα/kernel αλλά και σε οποιονδήποτε περιοριστικό παράγοντα πληρότητας όπως το αρχείο καταχωρητών ή η scratch-pad μνήμη, πράγμα που σημαίνει ότι ένας kernel που καταλαμβάνει πλήρως το αρχείο καταχωρητών δεν μπορεί να εκτελεστεί μαζί με έναν άλλο ακόμα και αν ο δεύτερος καταπονεί άλλες δομές. Ο στατικός διαχωρισμός της κάρτας όπως γίνεται στο MIG [94] ή στο Spatial Multitasking [2] επιλύει αυτό το πρόβλημα έχοντας περισσότερες από μία ομάδες πυρήνων στις οποίες μπορούν να εκδώσουν διαφορετικοί kernels, ωστόσο μέσα σε κάθε ομάδα τα προβλήματα χαμηλής χρησιμοποίησης παραμένουν.

0.1.5 Περίληψη της Πρότασης

Παρά το γεγονός ότι οι τεχνικές εξειδίκευσης είναι σαφώς ωφέλιμες για τους GPGPU kernels, αποτυγχάνουν να παρουσιάσουν μια καθολική λύση, δεδομένου ότι διακριτοί kernels, που συχνά προέρχονται από την ίδια εφαρμογή και εκτελούνται στην ίδια GPU χρησιμοποιούν το διαθέσιμο υλικό με ποικίλους τρόπους που δεν μπορούν να στοχευθούν από μία μόνο τεχνική εξειδίκευσης.

Παράλληλα η ταυτόχρονη εκτέλεση στο υπάρχον υλικό θέτει πολλούς περιορισμούς για την αύξηση της χρησιμοποίησης και δεν είναι πάντα αποτελεσματική. Αυτός είναι ο λόγος για τον οποίο στρέφουμε την προσοχή μας προς την ετερογένεια. Τα σημεία συμφόρησης υλικού κάθε kernel καθώς και τα μέρη που μένουν αχρησιμοποίητα μπορούν να εντοπιστούν καθιστώντας δυνατή την εξειδίκευση των SM προς τις ανάγκες τους.

Η εκμετάλλευση αυτού του φαινομένου με την εισαγωγή ετερογένειας σε επίπεδο κάρτας GPU με τη χρήση περισσότερων από μία φυσικών καρτών στις οποίες μπορεί να εκδώσει η εφαρμογή(ες) δεν είναι λογική, δεδομένου ότι το κόστος ενέργειας και χρόνου που θα υποστεί από τη μετακίνηση δεδομένων μεταξύ kernels που εκτελούνται σε διαφορετικές πλατφόρμες αλλά λειτουργούν διαδοχικά, ανάλογα με τα αποτελέσματα των προηγούμενων kernels, θα ήταν σημαντικό. Η χρήση περισσότερων από μία φυσικών καρτών θα απαιτούσε επίσης περισσότερο φυσικό χώρο και πλεονάζουσα χρήση υλικού με τη μορφή πλακετών τυπωμένων κυκλωμάτων (PCB) και άλλων εξαρτημάτων που διαφορετικά θα μπορούσαν να μοιραστούν.

Αυτό προσθέτει αξία στην ιδέα μιας GPU με περισσότερους από έναν τύπους SM, εισάγοντας ετερογένεια σε επίπεδο SM, θεραπεύοντας ένα μεγαλύτερο εύρος αναγκών εφαρμογών GPU εντός του ίδιου ολοκληρωμένου. Αυτή η προσέγγιση είναι παρόμοια με εκείνη των bigLITTLE [8] αρχιτεκτονικών CPU. Μια τέτοια προσέγγιση μπορεί να αξιοποιηθεί βέλτιστα με τον διαμοιρασμό της GPU μεταξύ kernels. Μια στατικά διαμοιρασμένη ετερογενής προσέγγιση θα παρέχει πάντα χώρο για να φιλοξενήσει τουλάχιστον δύο kernels με διαφορετικά χαρακτηριστικά που προορίζονται να εκτελεστούν σε διαφορετικό υλικό. Ως επιπλέον πλεονέκτημα, το σύστημα κοινής μνήμης επιτρέπει την απρόσκοπτη επικοινωνία δεδομένων μεταξύ kernels οποιουδήποτε τύπου, ανεξαρτήτως του υλικού στο οποίο εκτελούνται. Ελλείψει ενός σύγχρονου προσομοιωτή GPU που να υποστηρίζει τη συνύπαρξη δύο διαφορετικών τύπων SMs με διαφορετική εσωτερική μικρο-αρχιτεκτονική στην ίδια GPU, τροποποιούμε την τελευταία έκδοση του προσομοιωτή GPU γενικού σκοπού που περιέχεται στον Accel-Sim, τον GPGPU-Sim [12] v4.2.0 επιτρέποντάς του να μοντελοποιεί έναν αυθαίρετο αριθμό κάθε κατηγορίας SM με πολλές διαθέσιμες διαφορές στις διαμορφώσεις τους, ενώ προσαρμόζουμε επίσης το μοντέλο ισχύος του Accelwattch [63], έτσι ώστε τα αποτελέσματά του να είναι σύμφωνα με το νέο μοντέλο χρονισμού. Επιπλέον, επεκτείνουμε το CUDA API με μια ειδική κλήση που επιτρέπει στον προγραμματιστή να δηλώνει το προτιμώμενο υλικό για κάθε εκκίνηση των kernel του. Ταυτόχρονα προσφέρουμε ένα νέο μοντέλο εφαρμογών που προωθεί την ταυτόχρονη εκτέλεση kernels. Μελετούμε τις επιπτώσεις μιας τέτοιας διαμόρφωσης υλικού σε μια σειρά από kernels HPC. Ως μελέτη περίπτωσης της πρότασής μας για ετερογενείς single-Instruction-Set-Architecture (single-ISA) GPU επιπέδου SM, μελετάμε και αξιολογούμε τα αποτελέσματα μιας τέτοιας

διαμόρφωσης υλικού σε μια σειρά εφαρμογών HPC, δείχνοντας ότι υπάρχει δυνατότητα ταχύτερης εκτέλεσης και διατηρήσιμης ενεργειακής απόδοσης.

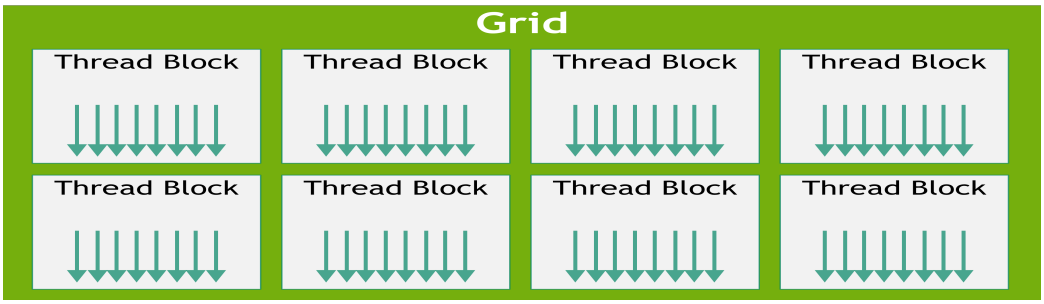


Figure 2: Οργάνωση των νημάτων σε πλέγματα και σε blocks. [86]

0.2 Υπόβαθρο

0.2.1 Μονάδες Επεξεργασίας Γραφικών ως Επεξεργαστές Γραφικών Γενικού Σκοπού

Οι Μονάδες Επεξεργασίας Γραφικών (GPU) είναι παράλληλες συσκευές με τεράστιο αριθμό νημάτων που αρχικά αποσκοπούσαν στην επιτάχυνση της απεικόνισης γραφικών. Ωστόσο, το εύρος εφαρμογής τους έχει αυξηθεί δραστικά, δημιουργώντας έναν ολόκληρο νέο τομέα GPU γενικής χρήσης (GPGPUs), ο οποίος επιτρέπει την εκμετάλλευση των πλεονεκτημάτων απόδοσης που επιτυγχάνονται από τον προαναφερθέντα παραλληλισμό σε μια ευρεία ποικιλία εφαρμογών εκτός γραφικών. Οι εν λόγω εφαρμογές αξιοποιούν τα ειδικά πλεονεκτήματα των GPU έναντι των CPU αναθέτοντας χρονοβόρα τμήματα της εκτέλεσής τους σε πυρήνες (kernels) με καθορισμένο μέγεθος πλέγματος και εκφορτώνοντάς τα στη GPU που χρησιμοποιείται ως επιταχυντής. Αυτοί οι kernels θα προχωρήσουν στη συνέχεια στη δημιουργία πολυάριθμων νημάτων ομαδοποιημένων σε blocks όπως στο Σχήμα 2, τα οποία θα ανατεθούν στις υπολογιστικές μονάδες ισχύος της GPU, τους ταυτόχρονους πολυεπεξεργαστές σε ορολογία NVIDIA (Streaming Multiprocessors - SMs), για να χωριστούν περαιτέρω σε ομάδες threads (warps - NVIDIA), των οποίων θα εκτελεστούν ταυτόχρονα, παράλληλα οι ίδιες ενολές (lockstep execution) στις υπολογιστικές σωληνώσεις (pipeline) που υπάρχουν εντός του SM. Τα warps παύουν να δεσμεύουν το pipeline όταν συναντούν μια καθυστέρηση, όπως μια μακρά λειτουργία μνήμης που πρέπει να εξυπηρετηθεί από το κύριο υποσύστημα μνήμης (DRAM), επιτρέποντας στην εκτέλεση άλλων warps να καλύψει την καθυστέρηση αυτή με αποτέλεσμα τη σταθερή υψηλή χρήση του υλικού. Προκειμένου να εξυπηρετηθεί ο προγραμματισμός Μονάδων Επεξεργασίας Γραφικών ως Επεξεργαστές Γραφικών Γενικού Σκοπού αναπτύχθηκαν προγραμματιστικές διεπαφές όπως η CUDA την οποία θα χρησιμοποιήσει η

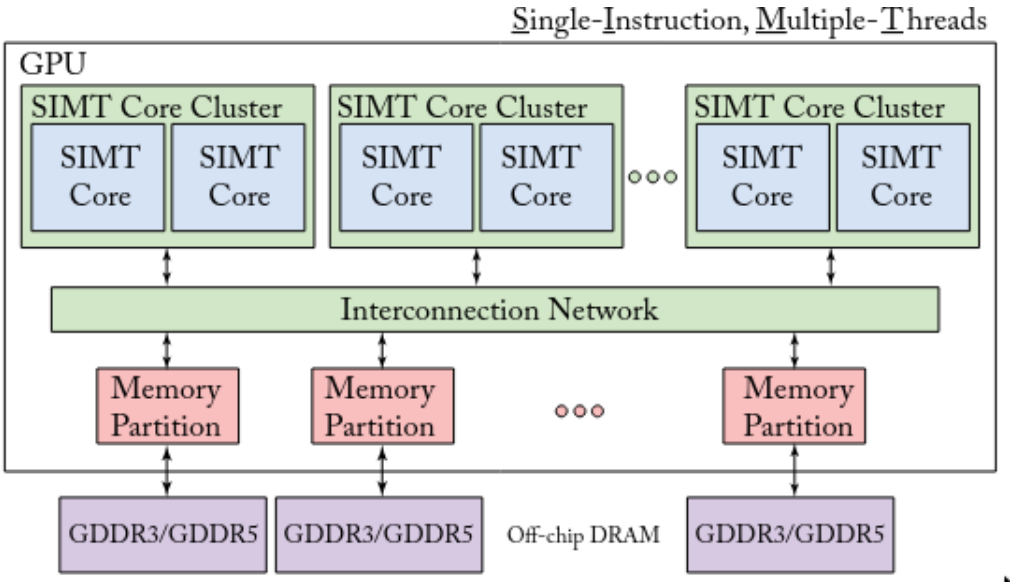


Figure 3: Υψηλού επιπέδου οργάνωση κάρτα γραφικών γενικού σκοπού. [1]

παρούσα εργασία αλλά και άλλες οπως το OpenCL και το HIP. Οι υπολογιστικοί πυρήνες της GPU (SMs), που μπορούν να ομαδοποιηθούν περαιτέρω σε συστάδες, αναλαμβάνουν το έργο της εκτέλεσης ενός προγράμματος μίας εντολής - πολλαπλών νημάτων (SIMT) έχοντας τη δυνατότητα ταυτόχρονης εκτέλεσης της τάξης των χιλιάδων νημάτων. Η επικοινωνία των νημάτων εντός του πυρήνα επιτυγχάνεται μέσω μιας διαμοιραζόμενης γρήγορης μνήμης εντός αυτού, και εκτός αυτού μέσω της κύριας μνήμης, ενώ για την επιβολή του συγχρονισμού χρησιμοποιούνται γρήγορες πράξεις φραγμού (barriers). Αυτοί οι πυρήνες χρησιμοποιούν κρυφές μνήμες εντολών και δεδομένων εντός του chip με στόχο τη μείωση του συνολικού όγκου της εξερχόμενης κίνησης προς τα χαμηλότερα, κοινά μεταξύ των πυρήνων, επίπεδα του συστήματος μνήμης. Όταν είναι αναπόφευκτη η αποστολή ενός αιτήματος σε ένα χαμηλότερο επίπεδο με μεγαλύτερη καθυστέρηση, ο χρονοπρογραμματισμός μεταξύ του τεράστιου αριθμού νημάτων που εκτελούνται στο SM βοηθά στην απόκρυψη του χρόνου απόκρισης του αιτήματος. [1]

Η υψηλή υπολογιστική απόδοση που παρέχουν οι συστάδες πυρήνων πρέπει να συνδυάζεται με ένα σύστημα μνήμης υψηλής απόδοσης, ώστε να μην το περιορίζει. Αυτό επιτυγχάνεται μέσω του παραλληλισμού στο υποσύστημα μνήμης, με τον διαχωρισμό της κύριας μνήμης της GPU σε πολλαπλά κανάλια μνήμης (όπως φαίνεται στο Σχήμα 3), τα οποία συνήθως περιλαμβάνουν επίσης μια κρυφή μνήμη τελευταίου επιπέδου. Το εύρος διευθύνσεων κατανέμεται ομοιόμορφα στα τμήματα της μνήμης.

Όπως αναφέρθηκε προηγουμένως, τα στρώματα μνήμης κάτω από τις

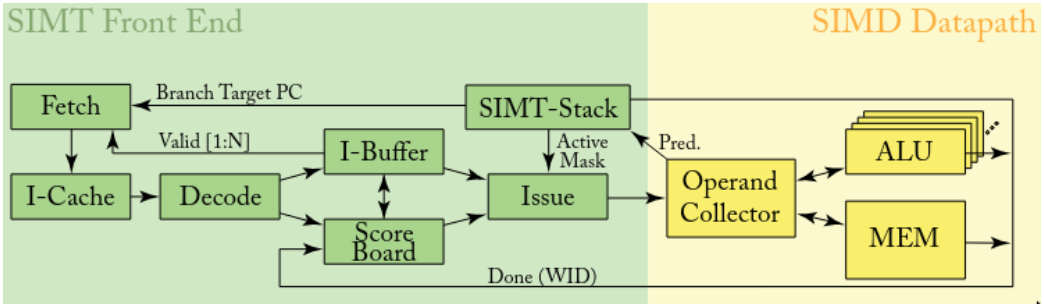


Figure 4: Αφαιρετική όψη της μικρο-αρχιτεκτονικής μιας γενικού σκοπού κάρτας γραφικών [1]

κρυφές μνήμες εντός του SM μοιράζονται και συνεπώς απαιτείται ένας εξελιγμένος μηχανισμός διασύνδεσης. Οι GPU χρησιμοποιούν ένα δίκτυο διασύνδεσης εντός του ολοκληρωμένου (Network-on-Chip - NOC). Προς όφελος της συνολικής απόδοσης χρησιμοποιεί πολιτικές διαμοιρασμού εύρους ζώνης και διαιτησίας [1]

Κάθε SM περιλαμβάνει τον κύριο υπολογιστικό αγωγό, μια αφαιρετική εικόνα του οποίου παρουσιάζεται στο σχήμα 4, ο οποίος μπορεί στη συνέχεια να αναλυθεί περαιτέρω στο μπροστινό άκρο, που περιλαμβάνει την ανάκτηση, αποκωδικοποίηση και έκδοση εντολών με τρόπο SIMT, και στο πίσω άκρο SIMD που περιλαμβάνει προσπελάσεις αρχείων καταχωρητών (Register File) και εκτέλεση εντολών. Περιέχει επίσης τα διάφορα συστήματα μνήμης εντός του SM, δηλαδή τις κρυφές μνήμες εντολών και δεδομένων πρώτου επιπέδου, την κοινή μνήμη scratchpad και το αρχείο καταχωρητών. Ακολουθεί σύντομη περιγραφή αυτών των μηχανισμών.

- **Πρόσθιο τμήμα αγωγού (frontend):**
  Αποτελείται κυρίως από μονάδες ανάκτησης και αποκωδικοποίησης εντολών, συσσωρευτές εντολών και χρονοπρογραμματιστές warps. Τα warps στα οποία αναφερθήκαμε προηγουμένως χρησιμεύουν ως μονάδα χρονοπρογραμματισμού εντός του πυρήνα της GPU. Οι εντολές των warp αντλούνται από τη μνήμη -ή την κρυφή μνήμη εντολών αν υπάρχει ήδη εκεί- στον **συσσωρευτή εντολών**. Ο συσσωρευτής εντολών μπορεί να συγκρατεί τις εντολές που έχουν ανακτηθεί από το warp επιτρέποντάς τους να προγραμματίζουν επόμενες εντολές περιμένοντας την ολοκλήρωση των προηγούμενων, εφόσον δεν υπάρχουν ανεκπλήρωτες εξαρτήσεις. Αυτός ο έλεγχος εξαρτήσεων υλοποιείται μέσω μιας δομής τύπου scoreboard [1, 73]. Αυτή η δομή μπορεί να είναι απλή έως και μόνο μερικά bits ανά warp και να δημιουργεί διανύσματα εξαρτήσεων που τα συνοδεύουν στον συσσωρευτή εντολών. [1]

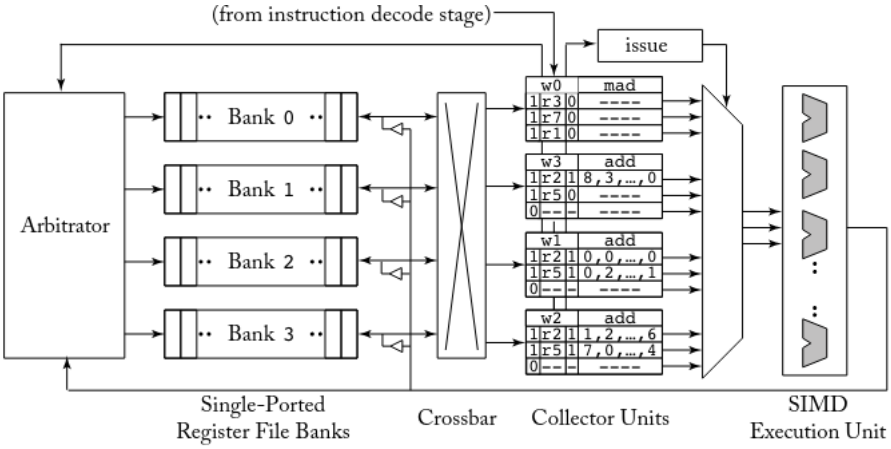


Figure 5: Συλλέκτης τελεστέων με τέσσερις μονάδες συλλέκτη και αρχιτεκτονική αρχείου καταχωρητών. [1]

Οι πυρήνες GPU διαθέτουν ένα άλλο εξειδικευμένο στοιχείο στο front end τους, **τη στοίβα SIMT**. Αυτή προσπαθεί να απαλύνει το πρόβλημα των αποκλινόντων μονοπατιών ελέγχου μεταξύ νημάτων που εκτελούνται σε συστοιχία βημάτων (lockstep). Μέχρι το σημείο που τα νήματα επανέρχονται σε κοινό μονοπάτι, το οποίο εγγράφεται στη δομή, οι καταχωρήσεις της απενεργοποιούν τα νήματα που δεν ακολουθούν την τρέχουσα διαδρομή εκτέλεσης, εμποδίζοντάς τα από το να εκτελούν περιττούς υπολογισμούς. Η NVIDIA έχει επίσης προτείνει έναν μηχανισμό επανασύγκλισης χωρίς στοίβα. [35, 31]
Υπεύθυνος για την επιλογή του warp που θα εκτελεστεί είναι ο **χρονοπρογραμματιστής warp**. Μπορεί να υπάρχουν περισσότεροι του ενός ανά πυρήνα [97, 98] αυξάνοντας το μέγιστο πλάτος έκδοσης ενός πυρήνα. Ο χρονοπρογραμματιστής warp εξετάζει τον συσσωρευτή εντολών και επιλέγει μια εντολή από ένα warp που είναι έτοιμη για έκδοση (στη σειρά της για in-order έκδοση, χωρίς ανεκπλήρωτες εξαρτήσεις). Δεδομένου ότι μπορεί να υπάρχουν περισσότερα από ένα warps με έτοιμη εντολή, οι χρονοπρογραμματιστές warp διαθέτουν ενσωματωμένες πολιτικές για την επίλυση τέτοιων συγκρούσεων. Εάν το υλικό στο back end που απαιτείται για την εκτέλεση της επιλεγμένης εντολής είναι κατειλημμένο η εντολή επαναλαμβάνεται και θα εξεταστεί ξανά για χρονοπρογραμματισμό σε μελλοντικό κύκλο. Οι χρονοπρογραμματιστές warp οφείλουν να διαθέτουν μια μεγάλη δεξαμενή warps προς χρονοπρογραμματισμό σε κάθε κύκλο, προκειμένου να κρύβουν μεγάλες καθυστερήσεις άλλων warps.

- **Οπίσθιο τμήμα αγωγού (backend):**

Για να είναι σε θέση ο χρονοπρογραμματιστής να εναλλάσσει συνεχώς warps, είναι απαραίτητο ένα μεγάλο αρχείο καταχωρητών [97, 98, 107, 91] ικανό να φιλοξενήσει διαφορετικούς καταχωρητές για κάθε warp. Ο **συλλέκτης τελεστέων (Operand Collector)**, εκμεταλλεύεται απρόσκοπτα ένα έξυπνα διαμορφωμένο αρχείο καταχωρητών. Το αρχείο καταχωρητών είναι τμηματοποιημένο (banked), γεγονός δηλαδή χρησιμοποιούνται περισσότερες από μία μνήμες μονής θύρας για την προσομοίωση της σχεδίασης με πολλαπλές θύρες. Η κατανομή των καταχωρητών σε τράπεζες γίνεται με διαδοχικό τρόπο, ώστε να παρέχεται η δυνατότητα σε μία μόνο εντολή να φέρει όλους τους τελεστέους της σε έναν μόνο κύκλο. Για να αξιοποιηθεί αυτός ο παραλληλισμός χρησιμοποιείται ο συλλέκτης τελεστών όπως φαίνεται στο σχήμα 5. Αντικαθιστά τους καταχωρητές σταδιοποίησης εντολών, περιέχοντας πολλαπλές μονάδες συλλέκτη, οι οποίες με τη σειρά τους παρέχουν αρκετό χώρο για να φιλοξενήσουν όλους τους τελεστές μιας εντολής. Με την ύπαρξη πολλαπλών μονάδων συλλέκτη ενισχύεται ο παραλληλισμός σε επίπεδο τράπεζας. Ένας μηχανισμός δαιτησίας αποφασίζει ποιος θα έχει πρόσβαση στις τράπεζες πρώτος σε περίπτωση συγκρούσεων όπου ο συλλέκτης τελεστών χρησιμοποιεί μηχανισμό χρονοπρογραμματισμού για να τις ανεχθεί.
Όταν μια εντολή στρέβλωσης έχει πάρει όλους τους τελεστές της και έχει καθοριστεί η μάσκα εκτέλεσης SIMT, προωθείται στις **μονάδες εκτέλεσης**. Αυτές είναι ετερογενείς, με τις σύγχρονες GPU να περιέχουν μονάδες εκτέλεσης ακεραίων αριθμών, μονάδες εκτέλεσης κινητής υποδιαστολής μονής και διπλής ακρίβειας, ειδικές μονάδες εκτέλεσης για σύνθετες μαθηματικές πράξεις, μονάδες φόρτωσης/αποθήκευσης και εξειδικευμένες μονάδες εκτέλεσης όπως οι πυρήνες τανυστών [97, 98, 107, 91]. Κάθε λειτουργική μονάδα μπορεί να έχει πλάτος όσο το μέγεθος του warp της GPU ή όσο το μισό. Μπορούν να χρονίζονται σε υψηλότερες συχνότητες και να διοχετεύονται σε σωληνώσεις για να βελτιώσουν την απόδοση ανά περιοχή λειτουργικής μονάδας. [1]

Στις κάρτες γραφικών της NVIDIA ξεκινώντας από την Pascal, τα SM διχοτομούνται σε μικρότερους υπολογιστικούς πυρήνες (sub-cores) οι οποίοι περιέχουν αποκλειστικούς συσσωρευτές εντολών, χρονοπρογραμματιστές εντολών, αρχείο καταχωρητών, μονάδες συλλέκτη τελεστέων και μονάδες εκτέλεσης (Σχήμα 6) [95, 97, 107, 98, 91].

Figure 6: Πολυεπεξεργαστής Ροής (SM) της NVIDIA Volta V100 [97]

0.2.2 Εργαλεία Προσομοίωσης

Ο GPGPU-sim [12] είναι ένας σύγχρονος προσομοιωτής με ακρίβεια κύκλου που μοντελοποιεί μεγάλης κλίμακας, εξαιρετικά παράλληλες σύγχρονες GPU γενικού σκοπού. Υποστηρίζει την εκτέλεση κώδικα Parallel Thread Execution (PTX) [103], μια χαμηλού επιπέδου εικονική μηχανή και αρχιτεκτονική συνόλου εντολών για τον προγραμματισμό της GPU. Μοντελοποιεί το μεγαλύτερο τμήμα του αγωγού της GPU που συζητήθηκε προηγουμένως παρέχοντας ακριβή αποτελέσματα. Η εξερεύνηση του χώρου σχεδίασης είναι επίσης διαθέσιμη μέσω αρχείων διαμόρφωσης που καθορίζουν τις παραμέτρους του προσομοιωμένου υλικού. Λειτουργεί παρεμβαίνοντας ως βιβλιοθήκη χρόνου εκτέλεσης CUDA και ανακατευθύνοντας τις κλήσεις προς τη διεπαφή προγραμματισμού εφαρμογών (API) χρόνου εκτέλεσης CUDA στις δικές του συναρτήσεις προσομοίωσης. Ο GPGPU-Sim είναι προσομοιωτής χρονισμού και είναι ξεχωριστός, αλλά λειτουργεί σε συνδυασμό με τον λειτουργικό προσομοιωτή CUDA-sim που υπολογίζει τα αποτελέσματα του προγράμματος. Ο GPGPU-sim διασυνδέει τα SMs με το υποσύστημα μνήμης που περιλαμβάνει την L2 και τη DRAM μέσω δικτύου διασύνδεσης υλοποιημένο στον booksim [56]. Υποστηρίζονται τέσσερις τομείς ρολογιών (υπολογιστικοί πυρήνες,

δίκτυο διασύνδεσης, L2, DRAM) Οι υπολογιστικοί πυρήνες εκτελούν blocks νημάτων που τους έχουν ανατεθεί κυκλικά από τον χρονοπρογραμματιστή blocks. Παρακάτω παρουσιάζονται εν συντομεία τα στάδια διοχέτευσης των υπολογιστικών πυρήνων όπως μοντελοποιούνται στον GPGPU-sim [123].

- Το στάδιο **ανάκτησης** που προηγείται της αποκωδικοποίησης απλώς παίρνει το PC ενός warp, είτε από την κρυφή μνήμη εντολών είτε από τη μνήμη, και το τοποθετεί σε έναν συσσωρευτή ανάκτησης, εφόσον έχει χώρο και το ίδιο το warp δεν έχει εκκρεμείς εντολές στον απομονωτή εντολών. Η προαναφερθείσα κρυφή μνήμη εντολών μοντελοποιείται παρόμοια με οποιαδήποτε άλλη κρυφή μνήμη στον προσομοιωτή και είναι ιδιαίτερα διαμορφώσιμη.

- Στο στάδιο της **αποκωδικοποίησης**, ο συσσωρευτής εντολών τροφοδοτείται με εντολές από το στάδιο αποκωδικοποίησης, ζητώντας την εντολή, με τον μετρητή προγράμματος (PC), από τον λειτουργικό προσομοιωτή και τις τοποθετεί στον απομονωτή εντολών, εφόσον δεν είναι γεμάτος από έγκυρες εντολές που δεν έχουν ακόμη εκτελεστεί.

- Η **έκδοση** αναλαμβάνεται από τους χρονοπρογραμματιστές warps. Ο αριθμός των warp schedulers εντός του πυρήνα είναι παραμετροποιήσιμος και κατά μήκος τους γίνεται η υποδιαίρεση κάθε επιμέρους SM σε υποπυρήνες κατά την ορολογία του GPGPU-Sim. Ο χρονοπρογραμματιστής warp επιλέγει πρώτα ένα διαθέσιμο warp με βάση την πολιτική χρονοπρογραμματισμού του. Κατά τη διάρκεια αυτού του βήματος η εντολή που επιλέγεται ελέγχεται για κινδύνους ελέγχου, δεδομένων, δομικούς κινδύνους, φράγματα τοποθετημένα από τον προγραμματιστή κλπ. Μόλις μια εντολή εκδοθεί επιτυχώς, ενεργοποιείται η λειτουργική της εκτέλεση και το αποτέλεσμα γίνεται γνωστό διευκολύνοντας την ενημέρωση της στοίβας SIMT. Επίσης, στο scoreboard γίνονται νέες εγγραφές κράτησης καταχωρητών για τους καταχωρητές τους οποίους θα επηρεάσει η εντολή.

- Στο GPGPU-Sim οι λειτουργίες του **συλλέκτη τελεστέων** συνίστανται στην κατανομή εντολών σε ελεύθερες μονάδες συλλέκτη, στην επεξεργασία αιτήσεων ανάγνωσης καταχωρητών που δεν παρουσιάζουν συγκρούσεις banks, στην προώθηση των αποτελεσμάτων της ανάγνωσης στις μονάδες εκτέλεσης και στον προγραμματισμό των εγγραφών στο αρχείο καταχωρητών. Στον προσομοιωτή οι μονάδες συλλογής είναι ιδιαίτερα διαμορφώσιμες, αποκαλύπτοντας όχι μόνο τον αριθμό των μονάδων συλλογής που περιέχουν μαζί με τον αριθμό των θυρών τους, αλλά και το είδος τους. Συγκεκριμένα, οι μονάδες συλλογής μπορούν είτε να είναι εξειδικευμένες ανά τύπο εντολής είτε να είναι γενικευμένες.

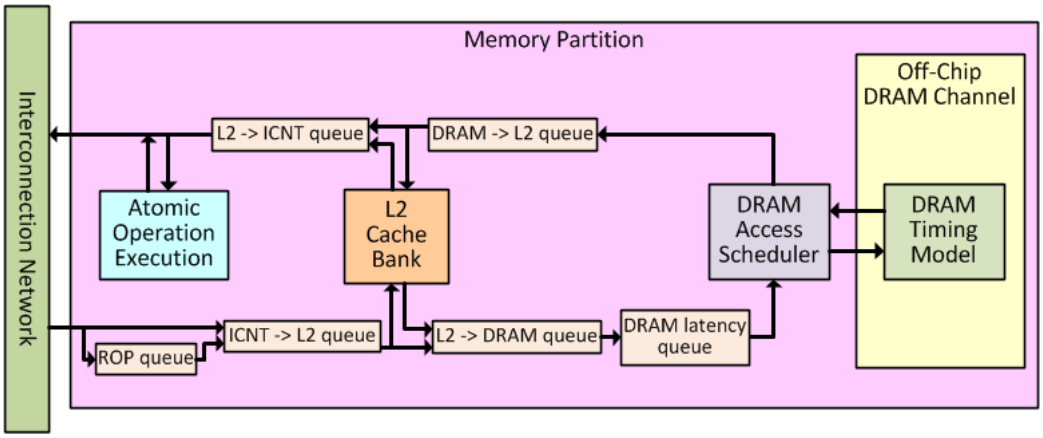


Figure 7: Αρχιτεκτονική διαμερίσματος μνήμης κάρτας γραφικών γενικού σκοπού [123]

- Ο GPGPU-Sim μοντελοποιεί τις λειτουργικές μονάδες ακεραίων, κινητής υποδιαστολής μονής/διπλής ακρίβειας, ειδικές λειτουργικές μονάδες, πυρήνες τανυστών και μια ενιαία μονάδα φόρτωσης/αποθήκευσης, καθώς και μια ειδική λειτουργική μονάδα καταχώρησης για πιθανές νέες λειτουργίες. Η ποσότητα καθεμιάς από τις λειτουργικές μονάδες είναι παραμετροποιήσιμη. Οι λειτουργικές μονάδες λαμβάνουν εντολές και τελεστές από τη μονάδα συλλογής και όταν ολοκληρώσουν την εκτέλεσή τους γράφουν τα αποτελέσματα σε ένα σύνολο καταχωρητών αγωγού εγγραφής, μοντελοποιώντας στο ενδιάμεσο την παραμετροποιήσιμη καθυστέρηση των εντολών.

Εκτός των SM, το δίκτυο διασύνδεσης μοντελοποιείται ως δύο ξεχωριστά δίκτυα, το ένα εξυπηρετεί την κυκλοφορία από τους πυρήνες προς τα διαμερίσματα μνήμης και το άλλο την κυκλοφορία που κινείται προς την αντίθετη κατεύθυνση. Τα πακέτα διασπώνται σε μικρότερα τμήματα (flits) όταν μετακινούνται από τους συσσωρευτές έγχυσης προς το δίκτυο και συσκευάζονται ξανά όταν ανασύρονται από τους οριακούς συσσωρευτές, προς τη μνήμη, όπου τοποθετούνται μετά την έξοδο από τους συσσωρευτές εξόδου.

Η μνήμη διασπάται σε διαμερίσματα μία αφαιρετική εικόνα των οποίων παρουσιάζεται στο Σχήμα 7. Οι αιτήσεις που λαμβάνονται από το δίκτυο διασύνδεσης προωθούνται στο κατάλληλο διαμέρισμα μέσω μιας ουράς FIFO. Σε αυτό το στάδιο προσομοιώνεται και η καθυστέρηση της L2, της οποίας οι τράπεζες λαμβάνουν αιτήματα μέσω μίας ουράς και απαντούν μέσω μίας άλλης αντίθετης κατεύθυνσης . Οι αστοχίες στην L2 προωθούνται σε μια άλλη ουρά που παρεμβάλλεται μεταξύ της τράπεζας L2 και της DRAM. Αφού προσομοιωθεί και η καθυστέρηση της DRAM οι απαντήσεις τοποθετούνται σε ουρά για

να εγγραφούν πίσω στην L2 και να προωθηθούν στο δίκτυο διασύνδεσης. Η DRAM διαθέτει χρονοπρογραμματιστή υπεύθυνο για την μεταβίβαση αιτήσεων από την ουρά L2 στην ουρά DRAM και την έκδοσή τους στις συστοιχίες DRAM.

Ο **AccelWattch** [63] είναι ένα μοντέλο ισχύος σε επίπεδο κύκλου GPU που έχει επικυρωθεί σε μια GPU NVIDIA Volta και παρουσιάζει υψηλή ακρίβεια. Πολλές από τις παραμέτρους του είναι ρυθμιζόμενες όπως και του GPGPU-Sim [12] από το οποίο μπορεί να αποκτήσει στατιστικά στοιχεία επιδόσεων κατά τη διάρκεια εκτέλεσης προσομοιώνοντας την κατανάλωση ενέργειας του εξεταζόμενου σεναρίου. Αναφέρει την κατανάλωση δυναμικής, στατικής και σταθερής ισχύος για ολόκληρη την κάρτα, ενώ λαμβάνει υπόψη το power-gating, την απόκλιση ροής ελέγχου και τη δυναμική κλιμάκωση συχνότητας τάσης (DVFS) [9] εφόσον υφίσταται. Είναι σε θέση να το κάνει αυτό χρησιμοποιώντας εξειδικευμένα micro-benchmarks με γνωστό αντίκτυπο στο υλικό, όπως η ενεργοποίηση μόνο ορισμένων δομών μιας φυσικής GPU, για να λάβει μετρήσεις ισχύος για τις δομές, τις οποίες χρησιμοποιεί για να δημιουργήσει τις κατάλληλες σταθερές και τα κατάλλα βάρη τα οποία σε συνδυασμό με τα στατιστικά προσομοιωμένου χρόνου εκτέλεσης θα οδηγήσουν στην τελική κατανάλωση ισχύος. Ο προσομοιωτής ισχύος παραδίδεται προ-ρυθμισμένος έναντι της Volta V100 και προσφέρει παραμετροποιήσιμες ρυθμίσεις που επιτρέπουν την εξερεύνηση του χώρου σχεδίασης.

Ο **Accel-Sim** [67] είναι ένα framework προσομοίωσης που παρέχει ένα front-end το οποίο διευκολύνει την εκτέλεση της εγγενούς μηχανικής αρχιτεκτονικής συνόλου εντολών (ISA) της NVIDIA GPU με βάση ίχνη πραγματικής εκτέλεσης, διατηρώντας παράλληλα την υποστήριξη της προσομοίωσης της εικονικής PTX ISA. Ο Accel-Sim χρησιμοποιεί το NVBit [128] για την εξαγωγή εντολών ISA μηχανής από όλα τα δυαδικά προγράμματα CUDA, ακόμη και αυτά που έχουν κατασκευαστεί με τη χρήση κλειστού κώδικα εξαιρετικά βελτιστοποιημένων βιβλιοθηκών και στη συνέχεια τις μετατρέπει σε μια εσωτερική αναπαράσταση που δεν εξαρτάται από το ISA. Περιέχει την τελευταία έκδοση του GPGPU-Sim που αναπτύχθηκε μαζί του. Έχει τη δυνατότητα αυτόματης ρύθμισης και επικύρωσης ενός νέου μοντέλου επιδόσεων για μια πρόσφατα κατασκευασμένη GPU χρησιμοποιώντας συμπεριλαμβανόμενα microbenchmarks μέσω των οποίων μπορούν να εντοπιστούν αρχιτεκτονικές αλλαγές.

Η big.LITTLE [10] αρχιτεκτονική της ARM είναι το πιο γνωστό παράδειγμα **Ετερογενούς Συστήματος Κοινής Αρχιτεκτονικής Συνόλου Εντολών (ISA) στον κόσμο της Κεντρικής Μονάδας Επεξερ-**

γασίας (CPU). Η αρχική του έκδοση περιλαμβάνει έναν επεξεργαστή πολλαπλών αναθέσεων, εκτός σειράς, και έναν απλό επεξεργαστή 8 σταδίων εντός σειράς. Απευθύνεται σε κινητές συσκευές, αναγνωρίζοντας την εγγενή ανάγκη τους για ενεργειακή απόδοση. Οι κινητές συσκευές εκτελούν μια μεγάλη ποικιλία εφαρμογών που κυμαίνονται από εντατικά παιχνίδια έως υπηρεσίες συστήματος android χαμηλής έντασης. Η βελτιστοποίηση για τις πιο απαιτητικές εφαρμογές και η χρήση ισχυρών ενεργοβόρων επεξεργαστών είναι επιζήμια για την ενεργειακή απόδοση της συσκευής κατά την εκτέλεση συχνά αδρανών μόνιμων υπηρεσιών παρασκηνίου ενώ η βελτιστοποίηση για τις τελευταίες θα υποβάθμιζε τις επιδόσεις των απαιτητικότερων. Και οι δύο επεξεργαστές μοιράζονται παρά τις μικροαρχιτεκτονικές διαφορές τους το ίδιο ISA με τους προγραμματιστές να έχουν τη δυνατότητα να προγραμματίζουν απρόσκοπτα για έναν τέτοιο επεξεργαστή όπως θα έκαναν για οποιονδήποτε άλλο. Οι διεργασίες που εκτελούνται σε έναν τέτοιο επεξεργαστή μπορούν να μετακίνουνται από τους πυρήνες ενός τύπου στου άλλου βάσει συγκεκριμένων πολιτικών εξαρτόμενων από τα χαρακτηριστικά εκτέλεσής των διεργασιών.

0.3 Υλοποίηση

0.3.1 Εισαγωγή

Σε αυτό το κεφάλαιο θα παρουσιάσουμε την εργασία μας για την επέκταση του GPGPU-Sim [12], που περιέχεται στον Accel-Sim, ώστε να υποστηρίζει την προσομοίωση ετερογενών, κοινού ISA GPUs με δύο είδη SM τα οποία μοιράζονται ένα κοινό σύστημα μνήμης. Επίσης θα παρουσιάσουμε τις δυνατότητες και τους περιορισμούς της νέας έκδοσης, ενώ θα τροποποιήσουμε επίσης τον AccelWattch [63] ώστε να είναι συμβατό με το σχεδιασμό μας. Παρουσιάζουμε την υλοποίησή μας μιας επέκτασης του CUDA API που επιτρέπει στον προγραμματιστή να καθορίσει τον τύπο του SM στον οποίο θα πρέπει να εκτελούνται οι πυρήνες (kernels) του, κατά τη μεταγλώττιση ή την εκτέλεση. Τέλος, παρουσιάζουμε το μοντέλο λογισμικού των benchmarks που έχουν σχεδιαστεί για να εκτελούνται σε αυτή την έκδοση του προσομοιωτή και μια τυποποιημένη και εύκολα επεκτάσιμη σουίτα από benchmarks.

0.3.2 Ενεργοποιώντας την Ετερογένεια

Στις προσπάθειές μας να ενισχύσουμε τον Accel-Sim [67] με τη δυνατότητα μοντελοποίησης ετερογενών μονάδων επεξεργασίας μέσα σε μια GPU, έχουν γίνει διάφορες τροποποιήσεις όπως φαίνεται στο Σχήμα 8. Πρώτα απ' όλα, επεκτείναμε τις δυνατότητές του για να χειρίζεται πολλαπλά αρχεία ρυθμίσεων. Αντί για ένα μόνο αρχείο διαμόρφωσης, ο Accel-Sim αναλύει τώρα τρία διαφορετικά αρχεία (όπως φαίνεται στο Σχήμα ①). Το ένα αρχείο περιγράφει τις κοινές δομές GPU και τις επιλογές προσομοίωσης, ενώ τα άλλα δύο είναι αφιερωμένα σε κάθε έναν από τους δύο τύπους πυρήνων, παρέχοντας μια λεπτομερή περιγραφή των εσωτερικών μικρο-αρχιτεκτονικών διαμορφώσεων τους.

Επιπλέον, έχει τροποποιηθεί η αρχικοποίηση των δομών GPU από τον προσομοιωτή ②. Δεδομένου ότι ο προσομοιωτής είναι κατασκευασμένος με αντικειμενοστραφή τρόπο και κάθε συστάδα SM και τα SM που περιέχει είναι ξεχωριστά αντικείμενα, με κατάλληλες προσαρμογές, οι ετερογενείς πυρήνες μπορούν να εξακολουθούν να εμφανίζονται ως συστάδες SIMT της ίδιας κλάσης παρά τις εσωτερικές τους διαφορές. Αυτή η αλλαγή εξασφαλίζει ότι οι διαφορές μεταξύ των ετερογενών SM παραμένουν διαφανείς στο υπόλοιπο σύστημα. Είναι σημαντικό ότι δεν χρειάζεται να αλλάξουν οι εσωτερικές διαδικασίες κάθε SM, εκτός από ορισμένες πτυχές της διευθυνσιοδότησης που εξαρτώνται από τον συνολικό αριθμό των SM, καθώς η ετερογένεια ενσωματώνεται απρόσκοπτα στο σύστημα.

Τα ετερογενή SM πρέπει να μπορούν να λειτουργούν σε διαφορετικές ταχύτητες ρολογιού, επομένως η προτεινόμενη έκδοση χρησιμοποιεί δύο ξε-

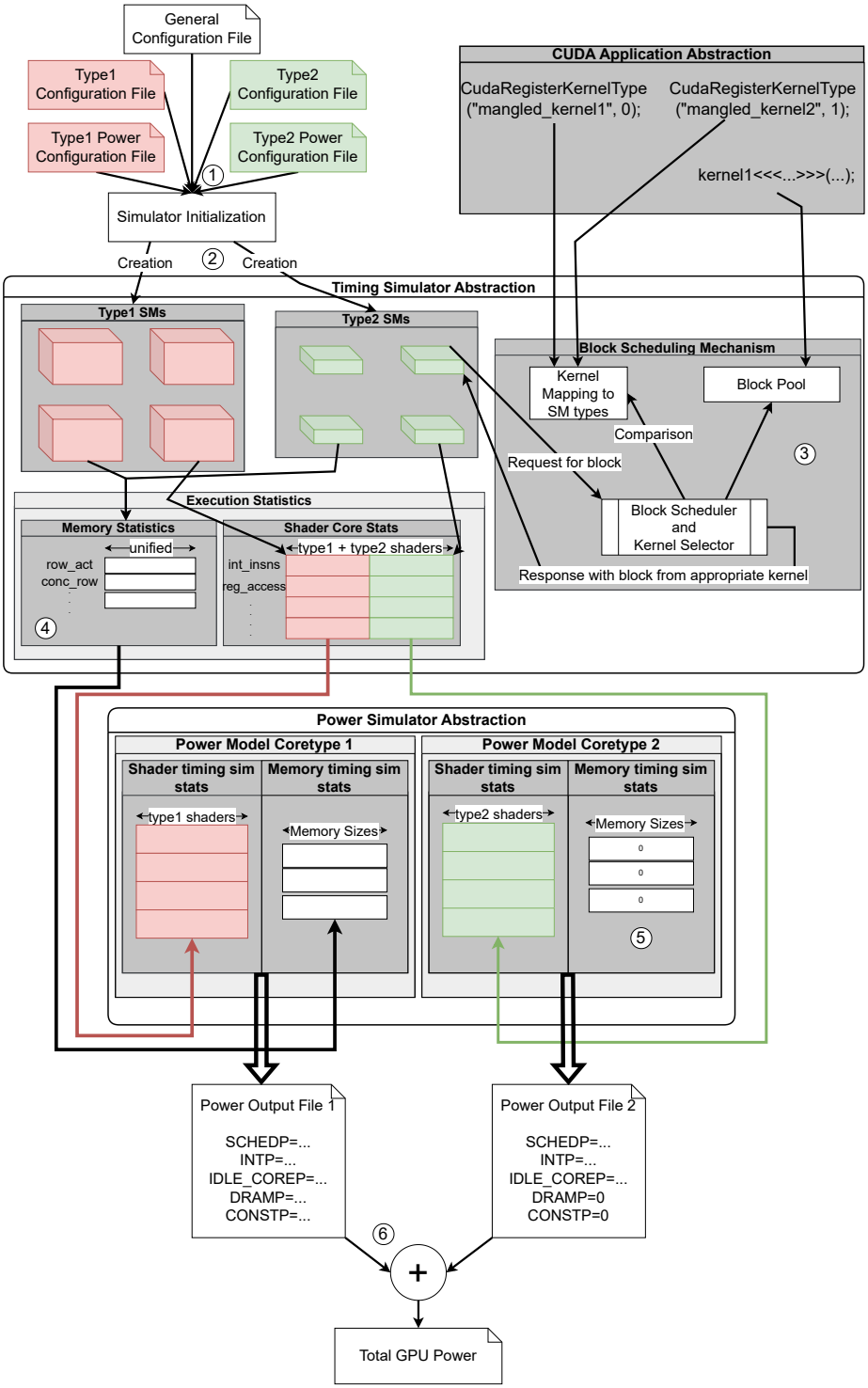


Figure 8: Μετατροπές στον simulator προς υποστήριξη ετερογενών αριτεκτονικών

χωριστά, ανεξάρτητα ρολόγια πυρήνων για να υποστηρίξει αυτήν τη λειτουργία. Ωστόσο, στον Accel-Sim το ρολόι των cores είναι το παγκόσμιο ρολόι του προσομοιωτή, το οποίο χρησιμοποιείται για τον υπολογισμό διαφόρων καθυστερήσεων και την παρακολούθηση του συνολικού χρόνου εκτέλεσης, επομένως με δύο διαφορετικά ρολόγια υπάρχει έλλειψη ενός παγκόσμιου ρολογιού που θα μπορούσε να οδηγήσει σε χρονική απόκλιση μεταξύ των καθυστερήσεων λειτουργιών που εκδίδονται από τα διαφορετικού τύπου SMs. Οι καθυστερήσεις δομών εκτός του SM όπως η DRAM θα έπρεπε να είναι σταθερές ως προς το ρολόι των cores στο πεδίο του χρόνου. Για την αντιμετώπιση αυτού του ζητήματος εισάγουμε ένα τρίτο τεχνητό ρολόι προσομοίωσης του οποίου η συχνότητα είναι το ελάχιστο κοινό πολλαπλάσιο των συχνοτήτων των δύο ρολογιών των cores. Με αυτό ως παγκόσμιο ρολόι οι καθυστερήσεις που θα έπρεπε να είναι ανεξάρτητες από τη συχνότητα του πυρήνα, αλλά μοντελοποιούνται με βάση το παγκόσμιο ρολόι, μπορούν να κλιμακωθούν αναλόγως και να παραμείνουν σταθερές ως προς τον χρόνο για όλα τα SM, ενώ οι λειτουργίες εντός του SM συμβαίνουν στην καθορισμένη συχνότητα κάθε τύπου πυρήνα.

Η προσέγγισή μας απαιτεί επίσης τροποποίηση του μηχανισμού χρονοπρογραμματισμού block ③. Στον Accel-Sim, κάθε SM που δεν ασχολείται επί του παρόντος με την εκτέλεση κάποιου block νημάτων ζητά ένα προς εκτέλεση. Σε αυτό το σημείο αναλαμβάνει δράση ο επιλογέας kernel, ο οποίος επιλέγει ένα block από τα διαθέσιμα. Εισάγουμε ένα χαρακτηριστικό κατά το οποίο αυτός ο επιλογέας εξετάζει την πηγή του αιτήματος, προσδιορίζοντας συγκεκριμένα τον τύπο του SM που κάνει το αίτημα. Στη συνέχεια, συγκρίνει τα υποψήφια block προς επιλογή με έναν κατάλογο όλων των kernels που έχουν καταχωρηθεί από τον προγραμματιστή εφαρμογών CUDA, όπως φαίνεται στην υπο-ενότητα 0.3.5 και στο ③ για να εκτελεστούν στον συγκεκριμένο τύπο SM. Εάν ένα συμβατό και διαθέσιμο block ευθυγραμμίζεται με τον τύπο SM, ορίζεται για εκτέλεση. Περισσότερες λεπτομέρειες σχετικά με την κατηγοριοποίηση των kernels θα συζητηθούν εν συνεχεία στην υπο-ενότητα 0.3.5.

Προκειμένου να παρέχεται στους μελλοντικούς ερευνητές η δυνατότητα να παρατηρούν με σαφήνεια τη συμπεριφορά καθενός από τους σχεδιασμένους τύπους SM, η αναφορά στατιστικών στοιχείων πρέπει να αλλάξει από την τρέχουσα, συσσωρευμένη σε όλα τα SM, μορφή της. Για τον σκοπό αυτό, τα συσσωρευμένα στατιστικά στοιχεία για ολόκληρη την GPU θα πρέπει μερικές φορές να συσσωρεύονται και για τους πυρήνες κάθε τύπου ξεχωριστά. Αυτή η έκδοση του προσομοιωτή προσφέρει μια μεγάλη ποικιλία στατιστικών ανά τύπο πυρήνα για να βοηθήσει τον χρήστη να κατανοήσει τις λεπτομέρειες της ταυτόχρονης εκτέλεσης των kernels του. Η αναφορά καθυστερήσεων για προσπελάσεις σε δομές εκτός του SM, όπως η DRAM, έχει επίσης εμπλουτιστεί με ατομική αναφορά για κάθε τύπο πυρήνα επιτρέποντας την καλύτερη ανάλυση

συμφόρησης στις κοινές δομές καθώς και την ανάλυση παρεμβολών μεταξύ kernels (περισσότερα σχετικά στην υπο-ενότητα 0.5.2). Αυτό καθίσταται εφικτό με την επισήμανση των αιτήσεων μνήμης που εξέρχονται από τους πυρήνες με πληροφορίες σχετικά με τον τύπο του SM που τις εξέδωσε. Επιπλέον, έχουν υλοποιηθεί ορισμένα νέα στατιστικά στοιχεία, τα οποία δεν υποστηρίζονταν προηγουμένως στο Accel-Sim, όπως η ανάλυση των scoreboard καθυστερήσεων ανάλογα με τον τύπο της εντολής που τις προκάλεσε.

Η αναφορά πρέπει να υποστηρίζεται από τροποποιήσεις στον τρόπο με τον οποίο κρατώνται τα εν λόγω στατιστικά στοιχεία εντός του Accel-Sim. Ορισμένα από αυτά μεταβιβάζονται επίσης στο AccelWattch [63] ως μετρητές επιδόσεων, οπότε η κατάλληλη καταμέτρησή τους είναι ζωτικής σημασίας. Από προεπιλογή στο Accel-Sim τα περισσότερα στατιστικά στοιχεία πυρήνων παρακολουθούνται ανά πυρήνα μέσα σε ένα αντικείμενο στατιστικών. Η προσέγγισή μας ④ περιλαμβάνει επίσης τη χρήση ενός ενιαίου αντικειμένου στατιστικών που περιλαμβάνει όλους τους τύπους πυρήνων, διατηρώντας μια ολοκληρωμένη καταγραφή των στατιστικών προσομοίωσης ανά πυρήνα. Για τα στατιστικά στοιχεία που δεν αφορούν ειδικά κάθε πυρήνα αλλά παρακολουθούνται σε όλα τα SM, έχουμε προσθέσει μια επιπλέον διάσταση στους πίνακες στατιστικών στοιχείων που αντιπροσωπεύουν τον τύπο του πυρήνα. Όταν έρθει η ώρα να αναφερθούν τα στατιστικά στοιχεία, η συσσώρευση γίνεται ανά τύπο πυρήνα όπου κρίνεται σκόπιμο. Το ξεχωριστό αντικείμενο στατιστικών που χρησιμοποιείται για την παρακολούθηση των λειτουργιών μνήμης και των απομονωμένων στατιστικών μνήμης αλλάζει επίσης κατάλληλα το μέγεθος όταν χρειάζεται, με ορισμένα στατιστικά στοιχεία μνήμης να διαφοροποιούνται βάσει του τύπου πυρήνα που οδήγησε σε αυτά, ενώ άλλα, όπως τα στατιστικά στοιχεία χαμηλού επιπέδου DRAM που χρησιμοποιούνται για εκτιμήσεις ισχύος, παρακολουθούνται για ολόκληρο το σύστημα.

0.3.3 Μοντέλο Ισχύος

Για να μοντελοποιήσουμε την κατανάλωση ενέργειας, έχουμε εισαγάγει δύο ξεχωριστά μοντέλα ενέργειας ⑤ τα οποία χρησιμοποιούν το προ-υλοποιημένο AccelWattch [63]. Το ένα μοντέλο είναι αφιερωμένο στην εκτίμηση της κατανάλωσης δυναμικής και στατικής ισχύος του πρώτου τύπου SMs και του διαμοιραζόμενου τμήματος μνήμης καθώς και της σταθερής ισχύος όλης της κάρτας. Το δεύτερο μοντέλο ισχύος επικεντρώνεται στη μοντελοποίηση της κατανάλωσης δυναμικής και στατικής ισχύος μόνο του δεύτερου τύπου SMs. Κατά συνέπεια, απαιτείται η ανάλυση δύο διαφορετικών αρχείων διαμόρφωσης ισχύος ① για την υποστήριξη αυτής της ολοκληρωμένης προσέγγισης μοντελοποίησης ισχύος. Για τον υπολογισμό της συνολικής δυναμικής κατανάλωσης ισχύος της GPU, εξαιρουμένων των διαμερισμάτων μνήμης, αξιοποιούμε την αρχή της

υπέρθεσης [46], η οποία δηλώνει ότι για ένα γραμμικό σύστημα η απόκριση που προκαλείται από δύο ή περισσότερα ερεθίσματα είναι το άθροισμα των αποκρίσεων που θα προκαλούσε κάθε ερέθισμα ξεχωριστά. Στην περίπτωσή μας το γραμμικό σύστημα είναι το άθροισμα όλων των συνεισφορών ισχύος των SMs, τα δύο διαφορετικά ερεθίσματα είναι οι δύο kernels που εκτελούνται ταυτόχρονα και η απόκριση είναι η συνολική δυναμική κατανάλωση ισχύος των SMs τύπου-1 συν εκείνη των SMs τύπου-2. Σε αυτήν προσθέτουμε τη δυναμική κατανάλωση ισχύος του υποσυστήματος μνήμης, η οποία υπολογίζεται για ολόκληρη την κάρτα στο πρώτο μοντέλο ισχύος, λαμβάνοντας υπόψη όλες τις προσπελάσεις, ανεξάρτητα από τον τύπο του SM από τον οποίο προήλθε. Μέσω αυτής της διαδικασίας υπολογίζεται η δυναμική κατανάλωση ισχύος για ολόκληρη την κάρτα ⑥ όπως φαίνεται στην 0.1.

$$P_{dyn,Total} = P_{dyn,SMsType1} + P_{dyn,SMsType2} + P_{dyn,ICNTfromType1SMs} + P_{dyn,ICNTfromType2SMs} + P_{dyn,DRAM} \quad (0.1)$$

Αυτό καθίσταται εφικτό επειδή κατά τη μοντελοποίηση της δυναμικής κατανάλωσης ισχύος το AccelWattch χρησιμοποιεί τα στατιστικά στοιχεία προσομοίωσης χρονισμού ανά πυρήνα που αναφέρθηκαν προηγουμένως, τα οποία μπορούν να διαχωριστούν με βάση τον τύπο του πυρήνα και να διαβιβαστούν στο κατάλληλο μοντέλο ισχύος μαζί με ορισμένα άλλα θεμελιώδη στατιστικά στοιχεία, όπως οι εκτελεσθείσες εντολές ανά τύπο SM. Τα στατιστικά στοιχεία που αφορούν το υποσύστημα μνήμης που βρίσκεται κάτω από το δίκτυο διασύνδεσης διαβιβάζονται συνολικά μόνο στο πρώτο μοντέλο, το οποίο, όπως αναφέρθηκε στην προηγούμενη παράγραφο, είναι υπεύθυνο για τον υπολογισμό ολόκληρης της δυναμικής κατανάλωσης ισχύος για το υποσύστημα μνήμης εκτός SM.

Κατά την αντιμετώπιση της στατικής ισχύος, οι υπολογισμοί του AccelWattch βασίζονται σε μετρήσεις που λαμβάνονται από προσομοιωμένο υλικό για διάφορες δομές και ενεργοποίηση λωρίδων. Δεν μοντελοποιείται αναλυτικά (ανά στοιχείο) αλλά μόνο ως ενιαία αριθμητική τιμή με βάση τους τύπους των λωρίδων που ενεργοποιούνται, γεγονός που καθιστά δύσκολο για το Accelwatch να αναφέρει με ακρίβεια τη στατική ισχύ κατά την εξερεύνηση του χώρου σχεδίασης, καθώς αναμένεται να αυξομειωθεί κατά τη δοκιμή διαφορετικών διαμορφώσεων, κάτι που δεν μοντελοποιείται. Ωστόσο, μοντελοποιούμε την κατανάλωση ισχύος με τα πρότυπα του AccelWattch για το ετερογενές μας σύστημα. Η συνολική στατική ισχύς υπολογίστηκε αρχικά προσθέτοντας τη μετρούμενη βασική στατική ισχύ της διαμόρφωσης - η οποία αντιστοιχεί στην πρώτη ενεργοποίηση της λωρίδας - συν την επιπλέον στατική ισχύ που προστίθεται για κάθε επιπλέον ενεργοποίηση της λωρίδας πολλαπλασιασμένη με τον

μέσο ενεργών αριθμό νημάτων εντός του warp κατά τη διάρκεια της εκτέλεσης του kernel. Δεδομένου ότι οι τιμές ενεργοποίησης που χρησιμοποιήθηκαν στις διαμορφώσεις μετρήθηκαν όταν όλα τα SM της κάρτας ήταν ενεργά, το AccelWattch εφαρμόζει έναν συντελεστή κλιμάκωσης με βάση τον αριθμό των ενεργών SM κατά τη διάρκεια της προσομοίωσης. Τώρα αλλάζουμε τον συντελεστή κλιμάκωσης που εφαρμόζεται σε αυτή τη μέτρηση ώστε να είναι

$$\frac{ActiveSMsPerCoreType}{\#AllSMs}$$

. Με αυτόν τον τρόπο υπολογίζουμε το τμήμα της στατικής ισχύος που συνεισφέρουν τα SM κάθε τύπου πυρήνα σε κάθε ένα από τα δύο μοντέλα μας, προσθέτοντάς τα και πάλι για να προκύψει η συνολική στατική ισχύς.

Από την άλλη πλευρά, η σταθερή ισχύς είναι ανεξάρτητη από τη διαμόρφωση του συστήματος SM και μνήμης [67] και έχει ήδη προσδιοριστεί ως σταθερός αριθμός για κάθε GPU που δοκιμάστηκε στο AccelWattch. Αυτή η τιμή αναφέρεται μαζί με τους υπόλοιπους υπολογισμούς ισχύος και ενσωματώνεται στο πρώτο μοντέλο ισχύος συνεπώς αναφέρεται μόνο μία φορά.

Είναι σημαντικό να επισημανθεί ότι τα δύο μοντέλα ισχύος λειτουργούν ανεξάρτητα και υπολογίζουν συλλογικά την κατανάλωση για όλα τα στοιχεία της GPU. Χρησιμοποιώντας την αρχή της υπέρθεσης, όπως αναφέρθηκε προηγουμένως, μπορούμε να προσθέσουμε τα αποτελέσματα και των δύο μοντέλων για ένα σταθερό χρονικό παράθυρο. Αυτή η προσέγγιση αποδίδει ουσιαστικά τη συνολική κατανάλωση ισχύος για ολόκληρη την κάρτα γραφικών. Ο προσομοιωτής αναφέρει ξεχωριστά τα αποτελέσματα των δύο μοντέλων παρέχοντας στον τελικό χρήστη καθαρότερη εικόνα για το ποιο μέρος της διαμόρφωσής του είναι υπεύθυνο για ποιο μέρος της συνολικής κατανάλωσης ισχύος.

0.3.4 Υποστηριζόμενη Ετερογένεια

Οι προαναφερθείσες επεκτάσεις και τροποποιήσεις παρέχουν στους χρήστες τη δυνατότητα να εισάγουν ετερογένεια μεταξύ των δύο τύπων πυρήνων σε σχεδόν κάθε αρχιτεκτονικό χαρακτηριστικό εντός του SM για το οποίο το Accel-Sim ήδη υποστηρίζει τροποποιήσεις. Συγκεκριμένα, οι χρήστες έχουν την ευελιξία να διαφοροποιούν τους δύο τύπους πυρήνων στις ακόλουθες διαστάσεις:

- **Clock gating των γραμμών και του αρχείου καταχωρητών**
- **Μέγιστος αριθμός νημάτων/warps και block ανά SM**
- **Μέγιστο πλάτος ανάκτησης(fetch) εντολών**

- **Μοντέλο υποπυρήνα:** Ο Accel-Sim υποστηρίζει αυτή τη λειτουργία για τις αρχιτεκτονικές Volta/Pascal, όπου οι χρονοπρογραμματιστές απομονώνονται σε υποδιαιρέσεις του SM (υπο-ενότητα 0.2.1)

- **Πλήρης διαμόρφωση κρυφής μνήμης εντός SM (εκτός από το μέγεθος γραμμής κρυφής μνήμης)**

- **Διαμόρφωση διαμοιραζόμενης μνήμης**

- **Αρχείο καταχωρητών**

- **Συλλέκτες Τελεστεών:** Αριθμός τους και οι θύρες τους, καθώς και ο τύπος τους. Ο Accel-Sim υποστηρίζει τόσο γενικευμένους όσο και εξειδικευμένους (ανά τύπο μονάδας εκτέλεσης) συλλέκτες τελεστέων

- **Χρονοπρογραμματιστές warp:** Αριθμός ανά πυρήνα και η πολιτική χρονοπρογραμματισμού τους, δυνατότητα διπλής έκδοσης όταν οι δύο εντολές χρησιμοποιούν διαφορετικές μονάδες λειτουργίας κ.λ.π.

- **Αριθμός καταχωρητών αγωγού για όλα τα στάδια του αγωγού εκτέλεσης**

- **Αριθμός μονάδων εκτέλεσης και οι τύποι τους ανά SM**

Πέρα από το εσωτερικό των SM, η ετερογένεια μπορεί να ενισχυθεί με την ανάθεση ρολογιών διαφορετικών συχνοτήτων στους δύο τύπους πυρήνων και με τη μεταβολή του αριθμού των SM κάθε τύπου. Ορισμένα χαρακτηριστικά περιορίζονται να είναι ίδια μεταξύ των διαφορετικών τύπων πυρήνων. Συγκεκριμένα:

- **Warp size**: Δεδομένου ότι χρησιμοποιείται κατά τη διάρκεια της ανάλυσης του κώδικα PTX ολόκληρου του εκτελέσιμου δυαδικού αρχείου πριν αρχικοποιηθεί οποιαδήποτε από τις διάφορες δομές.

- **Μέγεθος γραμμής κρυφής μνήμης για την κρυφή μνήμη υφής (texture cache)**: Χρησιμοποιείται στη λειτουργική προσομοίωση των προσπελάσεων 2D texture caches. Θα πρέπει να σημειωθεί ότι η αλλαγή του μεγέθους γραμμής ακόμη και στην προεπιλεγμένη έκδοση του Accel-Sim είναι δύσκολη όταν χρησιμοποιούνται τομεακές (sectored) κρυφές μνήμες, δεδομένου ότι ο αριθμός των τομέων που αποτελούν μια γραμμή και το μέγεθος του τομέα ορίζονται κατά τη μεταγλώττιση μέσα στον προσομοιωτή και καθορίζουν ρητά το μέγεθος της γραμμής της κρυφής μνήμης.

- **Στρατηγική επανασύγκλισης**: Καθώς χρησιμοποιείται κατά την ανάλυση του PTX για τον εντοπισμό σημείων επανασύγκλισης. Οι μελλοντικές εκδόσεις θα πρέπει να επιτρέπουν σε αυτή τη στρατηγική να διαφέρει μεταξύ των δύο τύπων πυρήνων

- **Καθυστερήσεις Εντολών**: διότι κωδικοποιούνται μέσα στις εντολές κατά την ανάλυση του PTX.

Οι περιορισμοί είναι ελάχιστοι σε σύγκριση με τους βαθμούς ελευθερίας που προσφέρονται, καθιστώντας την προτεινόμενη έκδοση του προσομοιωτή ένα ισχυρό εργαλείο για την εξερεύνηση του χώρου σχεδιασμού.

0.3.5 Δίνοντας τον Έλεγχο στον Προγραμματιστή

Ενεργοποιήσαμε με επιτυχία την ετερογένεια στην προτεινόμενη έκδοση του προσομοιωτή, ωστόσο δεν έχει καθοριστεί ακόμη πώς οι τελικοί χρήστες μπορούν να την εκμεταλλευτούν αναθέτοντας kernels σε στοχευμένο υλικό. Εισάγουμε ένα νέο χαρακτηριστικό στον προσομοιωτή μας, επιτρέποντας τον ρητό ορισμό του τύπου ενός kernel μέσω μιας κλήσης API που ονομάζεται **cudaRegisterKernelType**. Αυτή η κλήση δέχεται δύο ορίσματα, όπως φαίνεται στην υπογραφή της συνάρτησης που εμφανίζεται στο Listing 1: το όνομα συμβόλου του kernel και έναν ακέραιο αριθμό (0 ή 1) που υποδηλώνει τον τύπο των SMs όπου θα πρέπει να εκτελεστεί. Στο πλαίσιο του προσομοιωτή, το όνομα συμβόλου του kernel γράφεται σε ένα set που αντιστοιχεί σε έναν από τους δύο τύπους SMs, που υποδηλώνεται από τη δεύτερη ακέραια παράμετρο. Στη συνέχεια, ο χρονοπρογραμματιστής block θα συγκρίνει, για κάθε block που εξετάζει να εκδώσει σε ένα SM, το όνομα του kernel στον οποίο ανήκει με το set του. Εάν βρεθεί ταύτιση και το block είναι κατά τα άλλα επιλέξιμο να βρίσκεται στο συγκεκριμένο SM, πραγματοποιείται έκδοση block. Αυτή η καινοτομία παρέχει στους προγραμματιστές την ευελιξία να κατηγοριοποιούν τους kernels τους απευθείας στον πηγαίο κώδικα, αποφεύγοντας την ανάγκη για οποιαδήποτε τροποποίηση του προσομοιωτή ή των αρχείων διαμόρφωσης με βάση το εκτελούμενο benchmark.

```
1 __host__ __cudart_builtin__ cudaError_t CUDARTAPI
    cudaRegisterKernelType(char *kernelName, int type);
```

Listing 1: Ορισμός συνάρτησης cudaRegisterKernelType. Πρόκειται για CUDA host συνάρτηση

Το χαρακτηριστικό αυτό έχει ενσωματωθεί στην προτεινόμενη έκδοση του προσομοιωτή. Ωστόσο, ο μεταγλωττιστής NVIDIA CUDA δεν υλοποιεί εγγενώς αυτή τη λειτουργία. Ως εκ τούτου, για να τη χρησιμοποιήσουν, οι προγραμματιστές πρέπει να λαμβάνουν ασήμαντα, ωστόσο πρόσθετα, μέτρα κατά

τη μεταγλώττιση εφαρμογών που προορίζονται να εκτελούνται στην ετερογενή πλατφόρμα, ώστε να διασφαλίζεται η επιτυχής μεταγλώττιση και σύνδεση. Αυτή η έκδοση του προσομοιωτή περιλαμβάνει ένα σενάριο εγκατάστασης (σε συνδυασμό με το αντίστοιχο της απεγκατάστασης) το οποίο θα εγκαταστήσει ένα αρχείο επικεφαλίδων που περιέχει αυτόν τον ορισμό μεταξύ των προεπιλεγμένων αρχείων επικεφαλίδων CUDA που αποστέλλονται με την εγκατάσταση του CUDA. Με αυτόν τον τρόπο το αρχείο κεφαλίδας μπορεί να συμπεριληφθεί όπως θα συμπεριελάμβανε οποιοδήποτε άλλο αρχείο βιβλιοθήκης CUDA. Με τη χρήση ορισμένων ειδικών σημαιών του linker σφάλματα σύνδεσης λόγω έλλειψης αυτής της κλήσης στην αυθεντική βιβλιοθήκη CUDA μετατρέπονται σε προειδοποιήσεις και παρά το γεγονός ότι το σύμβολο δεν βρίσκεται κατά τη στατική σύνδεση, δεσμεύεται μια εγγραφή γι' αυτό στον πίνακα δυναμικών συμβόλων του δυαδικού, εξασφαλίζοντας τη σωστή δυναμική σύνδεση.

Αυτή η υλοποίηση έχει σχεδιαστεί ειδικά για να ανταποκρίνεται στις ανάγκες και να είναι συμβατή με το περιβάλλον προσομοίωσης. Στο πραγματικό υλικό δεν υπάρχει η ανάγκη συσχέτισης του τύπου ενός kernel με το όνομά του και το επιθυμητό υλικό εκτέλεσης μπορεί να καθοριστεί ως μέρος της κλήσης του kernel, απαιτώντας ελάχιστη επέκταση υλικού στη GPU - ουσιαστικά ένα bit ανά block που δηλώνει τον τύπο του kernel στον οποίο ανήκει και μια πύλη XOR για την αντιστοίχιση του τύπου του kernel με τον τύπο SM, ο οποίος μπορεί επίσης να υποδηλώνεται από ένα bit, όταν εξετάζεται η έκδοσή του. Ωστόσο, δεδομένου ότι δεν έχουμε πρόσβαση στα εσωτερικά του nvcc που μετατρέπει τη σύνταξη <<<...>>> σε πραγματικές κλήσεις χρόνου εκτέλεσης CUDA, οι οποίες στη συνέχεια μεταβιβάζονται στον προσομοιωτή, υλοποιούμε μια παράκαμψη χρησιμοποιώντας το όνομα συμβόλου του kernel ως αναγνωριστικό του.

Το πραγματικά υλοποιημένο υλικό θα μπορούσε επίσης να εξαλείψει την ανάγκη για τον προγραμματιστή να αναθέσει στατικά έναν kernel σε έναν τύπο SMs, χρησιμοποιώντας δυναμικά μοντέλα ωστόσο, αυτή η στατική ανάθεση είναι καταλληλότερη στο πλαίσιο της παροχής στον χρήστη πλήρους ελέγχου των δοκιμασμένων μοντέλων.

0.3.6 Μοντελοποίηση εφαρμογών με ταυτόχρονες εκτελέσεις kernels

Ειδικά στο πλαίσιο της παρούσας έρευνας, το προσομοιωμένο σενάριο απαιτεί την ταυτόχρονη εκτέλεση δύο ή περισσότερων kernels διαφορετικών τύπων σε μια GPU για να φιλοξενηθούν kernels διαφορετικών τύπων και στις δύο κατηγορίες SM (περαιτέρω αναλύεται στην ενότητα 0.4). Για να το επιτύχουμε αυτό, χρησιμοποιούμε τη στρατηγική της ταυτόχρονης εκτέλεσης δύο ξεχωριστών

εφαρμογών CUDA. Δεδομένου ότι ο Accel-Sim δεν υποστηρίζει επί του παρόντος την υπηρεσία πολλαπλών διεργασιών (MPS), προκειμένου να καταστεί δυνατή η ταυτόχρονη εκτέλεση kernels εντός του προσομοιωτή, αξιοποιούμε το Streams API, το οποίο έχει υλοποιηθεί εκ των προτέρων στον Accel-Sim. Με την εκκίνηση των δύο ανόμοιων kernels σε ξεχωριστά streams, επιτρέπουμε την ταυτόχρονη εξέτασή τους για χρονοπρογραμματισμό.

Η εκκίνηση του προσομοιωτή γίνεται μέσω της εκκίνησης μιας εφαρμογής CUDA, η οποία οδηγεί στη φόρτωση της libcudart (της βιβλιοθήκης χρόνου εκτέλεσης CUDA με την οποία είναι συνδεδεμένο το δυαδικό πρόγραμμα αναφοράς και η οποία παρακάμπτεται από τον προσομοιωτή) και στη δημιουργία ενός GPU context. Κατά συνέπεια, κάθε εκκίνηση μιας εφαρμογής CUDA αρχικοποιεί μια μεμονωμένη περίπτωση του προσομοιωτή. Για να διασφαλιστεί η απρόσκοπτη εκτέλεση, είναι επιτακτική ανάγκη όλοι οι kernels ενός ενιαίου benchmark να εκκινούνται από μια ενιαία διεργασία, η οποία χρησιμοποιεί εγγενώς το ίδιο GPU context και, συνεπώς, την ίδια εκτέλεση προσομοιωτή. Επιπλέον, είναι απαραίτητο οι δύο εφαρμογές να λειτουργούν ανεξάρτητα, όσον αφορά την εκτέλεση της CPU, ώστε να αποφεύγεται η αδράνεια της GPU που προκύπτει από αδικαιολόγητη σειριοποίηση που προκαλείται από τον συνδυασμό των δύο εφαρμογών.

Για αυτούς τους λόγους, χρησιμοποιείται μια ενιαία διεργασία-οδηγός, υπεύθυνη για την ανάλυση των παραμέτρων των δύο εφαρμογών, τη δημιουργία δύο διαφορετικών CUDA Streams και την εκκίνηση δύο νημάτων, το καθένα αφιερωμένο σε μια από τις εφαρμογές. Αυτά τα νήματα είναι εξοπλισμένα με τα ορίσματα της αντίστοιχης εφαρμογής, ένα από τα δύο streams και τυχόν πρόσθετες βασικές επιλογές.

Επιπλέον, τα νήματα εφαρμογών συστήνεται να ξεκινούν την εκτέλεση των πρώτων kernels τους με συγχρονισμένο τρόπο, μετριάζοντας έτσι πιθανές συνθήκες ανταγωνισμού και μη ντετερμινιστικής συμπεριφοράς στα αποτελέσματα της προσομοίωσης λόγω υψηλών χρόνων CPU που αποδίδονται στην ανάγνωση εισόδου κ.λπ. και να διατηρούν χαμηλούς χρόνους CPU μεταξύ πρόσθετων εκτελέσεων kernels. Αυτός ο συγχρονισμός μπορεί να επιτευχθεί εύκολα μέσω της εφαρμογής κατάλληλων φραγμών. Στη λίστα 2 παρουσιάζεται ένα πολύ βασικό παράδειγμα που βασίζεται στις αρχές που συζητήθηκαν παραπάνω. Σε αυτό το παράδειγμα, δεδομένου ότι δεν υπάρχει σημαντικός κώδικας CPU και η μόνη σειριοποίηση μεταξύ των δύο εκκινήσεων των kernels είναι η ελάχιστη επιβάρυνση που απαιτείται για την ασύγχρονη εκφόρτωση του kernel στην προσομοιωμένη GPU, χρησιμοποιείται ένα μόνο νήμα.

```
cudaStream_t stream1, stream2;
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);
```

```
4
5 cudaRegisterKernelType("mangled_kernel1", 0);
6 cudaRegisterKernelType("mangled_kernel2", 1);
7
8 kernel1<<<grid1,block1,smem1,stream1 >>>(...);
9 kernel2<<<grid2,block2,smem2,stream2>>>(...);
10
11 cudaStreamSynchronize(stream1);
12 cudaStreamSynchronize(stream2);
```

Listing 2: Αφαιρετική όψη benchmark που αποτελείται από δύο kernels που προορίζονται για ταυτόχρονη εκτέλεση στην προτεινόμενη έκδοση του προσομοιωτή. Οι kernels εκκινούνται σε ξεχωριστές ροές μετά την καταχώρηση του καθορισμένου τύπου πυρήνα εκτέλεσης, χωρίς να υπάρχει σειριοποίηση.

Αναγνωρίζοντας τις προαναφερθείσες οδηγίες δημιουργίας συγκριτικών δοκιμών (benchmarks), αναπτύξαμε μια σουίτα benchmarks κατάλληλη για εκτέλεση με αυτή την έκδοση του προσομοιωτή για να βοηθήσουμε την έρευνα σε αυτόν τον τομέα. Περιέχει δέκα εφαρμογές που σχετίζονται με HPC (το σενάριο-στόχος αναλύεται περαιτέρω στην ενότητα 0.4) και συνδυασμούς μεταξύ αυτών που περιέχουν kernels με διαφορετικά χαρακτηριστικά. Προσφέρει μια τυποποίηση μέσω της οποίας αυθαίρετα benchmarks (σχεδιασμένα ως benchmarks για τον αρχικό Accel-Sim ή ως κανονικές εφαρμογές CUDA) μπορούν να προστεθούν σε αυτό και με μικρές αλλαγές να εκτελούνται ταυτόχρονα μαζί με άλλους kernels. Αυτό διευκολύνεται μέσω μίας τυποποιημένης διεργασίας οδηγού που χρησιμοποιεί το Pthreads [84] το οποίο όπως εξηγήθηκε παραπάνω περιέχει τον κώδικα για τη δημιουργία των δύο ανόμοιων CUDA streams, τη δημιουργία και αρχικοποίηση δυνητικά χρήσιμων locks και barriers και την ανάγνωση των παραμέτρων των δύο εφαρμογών καθώς και την εκκίνησή τους. Μια υλοποιημένη δομή είναι υπεύθυνη να μεταβιβάζει σε κάθε benchmark ό,τι από τα παραπάνω χρειάζεται για την εκτέλεσή του. Επιλέγουμε επίσης να δηλώσουμε τους καθορισμένους πυρήνες εκτέλεσης (μέσω του cudaRegisterKernelType) στο νήμα του οδηγού για λόγους που θα γίνουν σαφείς σύντομα. Τα δύο νήματα προσομοίωσης περιέχουν πραγματικό πηγαίο κώδικα benchmark ή και τυπικής εφαρμογής CUDA με ελάχιστες τροποποιήσεις. Για να προσθέσει ο χρήστης ένα νέο benchmark στη σουίτα απαιτείται να προσαρμόσει τον πηγαίο κώδικα των benchmarks με τους ακόλουθους τρόπους:

- αντικατάσταση της κύριας συνάρτησης του benchmark με μια υπορουτίνα νήματος αφού το benchmark θα εκκινείται από τη διεργασία οδηγό
- ανάγνωση της δομής που παρέχεται από τον οδηγό για να λάβει τα ορίσματά του, τα οποία διαφορετικά θα διαβάζονταν από την τυπική είσοδο,

το stream και τυχόν πρόσθετα απαιτούμενα στοιχεία

- αντικατάσταση των blocking CUDA calls και των καθολικών συγχρονισμών με τους αντίστοιχους ασύγχρονους και ανά stream αντίστοιχα
- προσθήκη του CUDA stream ως τελεστή στις κλήσεις εκκίνησης του kernel
- προσθήκη φραγμών, εάν είναι απαραίτητο, πριν από την πρώτη εκκίνηση του kernel ενός benchmark

Αφού προστεθεί επιτυχώς ένα benchmark μπορεί να συνδυαστεί με όσα από τα άλλα ήδη υπάρχοντα benchmarks ο χρήστης κρίνει απαραίτητο δημιουργώντας μόνο νέα προγράμματα-οδηγούς χωρίς να αλλάξει ο πηγαίος κώδικας του κάθε benchmark. Αυτό είναι εφικτό επειδή ο κώδικας κάθε νήματος είναι ανεξάρτητος από το συζευγμένο benchmark. Η διεργασία οδηγός είναι ένα σταθερό κομμάτι κώδικα όπου οι μόνες αλλαγές που απαιτούνται σε αυτό προκειμένου να δημιουργηθεί ένας νέος συνδυασμός συγκριτικών δεικτών αναφοράς είναι οι εξής:

- παροχή των σωστών ονομάτων ρουτινών νήματος για να εκκινήσουν τα επιθυμητά benchmarks
- ορισμός του κατάλληλου υλικού για την εκτέλεση των kernels. Εκτελώντας αυτό το βήμα στον οδηγό οι kernels νημάτων δύνανται να εκδοθούν σε διαφορετικούς τύπους SM αναλόγως του συνδυασμού στον οποίο συμμετέχουν ή άλλων συνθηκών χωρίς να επηρεαστεί ο πηγαίος κώδικας του ίδιου του benchmark.
- ορισμός των παραμέτρων που πρέπει να λάβει κάθε εφαρμογή

Ως εκ τούτου έχουμε επιλύσει το ζήτημα που προκύπτει από την έλλειψη διαθέσιμων benchmarks για τη μελέτη μιας τέτοιας αρχιτεκτονικής, ενώ παράλληλα προκαλούμε ελάχιστη επιβάρυνση στον τελικό χρήστη για την προσαρμογή οποιουδήποτε benchmark ενδιαφέροντος για να εκτελεστεί σε αυτόν τον προσομοιωτή.

Στο σχήμα 9 παρουσιάζεται ένα παράδειγμα εκτέλεσης μιας εφαρμογής που αποτελείται από ένα benchmark που περιέχει αποκλειστικά kernels τύπου-1 και ένα άλλο που περιέχει πυρήνες τύπου-2 σχεδιασμένο με τα πρότυπα που περιγράφονται παραπάνω.

Σε περίπτωση που ο χρήστης ενδιαφέρεται για την ταυτόχρονη εκτέλεση συγκεκριμένων kernels από benchmarks που περιέχουν πολλούς, στο πρότυπο υπάρχουν απλοί μηχανισμοί συγχρονισμού που εξαρτώνται από την είσοδο της

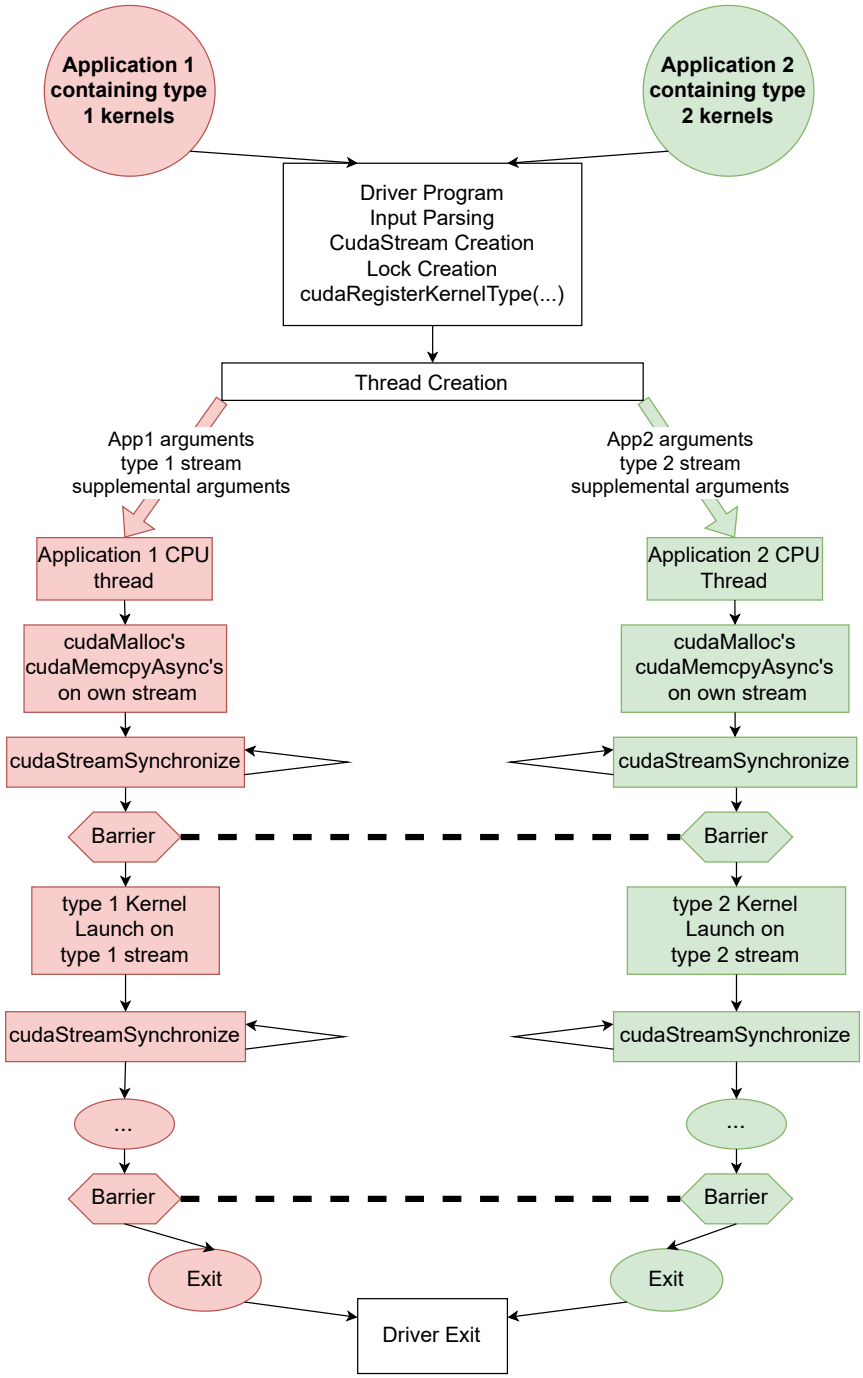


Figure 9: Παράδειγμα ταυτόχρονης εφαρμογής ετερογενών kernels που προορίζεται να δοκιμαστεί στον προσομοιωτή Single-ISA Heterogeneous GPU και σχεδιάστηκε ως μέρος της σουίτας benchmarks που τον συνοδεύει

διεργασίας οδηγού και εξυπηρετούν την εκτέλεση συγκεκριμένων τμημάτων από κάθε benchmark. Αυτό σημαίνει ότι είναι δυνατή η δοκιμή ταυτόχρονης εκτέλεσης kernels που προέρχονται από την ίδια εφαρμογή, συνδυάζοντας μια εφαρμογή με τον εαυτό της.

0.4 Μελέτη περίπτωσης

Το σενάριο εφαρμογής, το οποίο μελετάμε, για την προσαρμογή της ετερογενούς single-ISA GPU είναι η εκτέλεση εφαρμογών επιστημονικών υπολογισμών σε συμπλέγματα υπολογιστών υψηλής απόδοσης (HPC). Συνήθως σε ένα ερευνητικό κέντρο με ανάγκες επιστημονικών υπολογισμών ανατίθεται μια μεγάλη συστάδα HPC (HPC Cluster) η οποία μπορεί να περιέχει πολλούς κεντρικούς υπολογιστές και κόμβους [127]. Αυτοί οι κόμβοι είναι συχνά εξοπλισμένοι με GPUs [15] τις οποίες χρησιμοποιούν για να εκμεταλλευτούν τον παραλληλισμό που παρουσιάζουν τέτοιες εφαρμογές [79]. Παρόλο που η ανάπτυξη των GPU έχει σημειώσει τεράστια πρόοδο στον τομέα της μηχανικής μάθησης, το HPC έχει μείνει πιο πίσω, καθιστώντας το ελκυστικό τομέα-στόχο.

Ο στόχος αυτής της εργασίας είναι να επιτραπεί η παράλληλη εκτέλεση kernels με διαφορετικά χαρακτηριστικά εκτέλεσης στην ίδια GPU που διαθέτει ετερογενείς πυρήνες επεξεργασίας. Οι φόρτοι εργασίας CUDA HPC τείνουν να χρησιμοποιούν πολλαπλούς πυρήνες CUDA που εκτελούνται διαδοχικά, ο καθένας από τους οποίους καταναλώνει τα δεδομένα του προηγούμενου για τον υπολογισμό του τελικού αποτελέσματος. Δεδομένου ότι οι κόμβοι HPC τείνουν να εκτελούν έναν μόνο τύπο εφαρμογής κάθε φορά (π.χ. μια συγκεκριμένη εφαρμογή ρευστοδυναμικής) και οι πυρήνες της εκτελούνται με αυστηρά σειριακό τρόπο εντός των διεργασιών λόγω των εξαρτήσεων δεδομένων που επιβάλλει ο αλγόριθμος, θα μπορούσε κανείς εύλογα να καταλήξει στο συμπέρασμα ότι δεν υπάρχει χώρος για παραλληλισμό σε επίπεδο πυρήνα σε αυτό το σενάριο. Ωστόσο, τόσο οι προμηθευτές όσο και η βιβλιογραφία προτείνουν τον διαμοιρασμό των GPU σε επίπεδο διεργασιών/νημάτων [75, 130, 57, 16, 104, 94], με τις εφαρμογές HPC να χρησιμοποιούν τον εν λόγω παραλληλισμό σε επίπεδο CPU. Ένα πραγματικό παράδειγμα αυτής της μορφής διαμοιρασμού GPU είναι αυτό μεταξύ διεργασιών MPI [23], από τις οποίες συνήθως αποτελούνται οι εργασίες HPC, μια πρακτική που διευκολύνεται σημαντικά από το MPS [90]. Στόχος του διαμοιρασμού είναι η αύξηση της χρησιμοποίησης του υλικού και η μείωση της συνολικής κατανάλωσης ενέργειας. Η βιβλιογραφία προτείνει επίσης τον διαμοιρασμό σε επίπεδο κόμβου HPC [111] κάνοντας τον παραλληλισμό σε επίπεδο πυρήνα ακόμη πιο εμφανή. Επιπλέον, οι χρήστες συστάδων συνηθίζουν να προγραμματίζουν πολλαπλές εργασίες ακόμη και αν περιέχουν μόνο μία εφαρμογή οι οποίες θα προγραμματίζονται στους ίδιους κόμβους, όπως είναι σύνηθες σε συστάδες HPC. Έτσι, αυτές οι εργασίες θα μπορούσαν να εκτελεστούν ταυτόχρονα και δεδομένου ότι είναι ανεξάρτητες. Σε ένα σταθερό χρονικό σημείο θα βρίσκονται πιθανότατα σε διαφορετικά σημεία εκτέλεσης και θα έχουν διαφορετικούς πυρήνες προς εκτέλεση οι οποίοι με τη σειρά τους μπορούν να προγραμματιστούν παράλληλα

στην GPU. Έτσι, ο παραλληλισμός μεταξύ πυρήνων διαφορετικών εργασιών είναι επίσης εκμεταλλεύσιμος.

Οι προτάσεις αυτές είναι επίσης εφαρμόσιμες σε άλλα πεδία, όπως τα νευρωνικά δίκτυα (NN) ή το Cloud Computing [112]. Ωστόσο, η παρούσα εργασία επικεντρώνεται στο σενάριο επιστημονικού υπολογισμού HPC. Ο στόχος αυτής της ενότητας είναι η δημιουργία μιας ετερογενούς GPU με τα SM κάθε κατηγορίας προσαρμοσμένα για kernels συγκεκριμένων χαρακτηριστικών ώστε να μελετηθεί ως απόδειξη της ιδέας (proof of concept). Για να το πετύχουμε αυτό θα συλλέξουμε ένα σύνολο kernels αντιπροσωπευτικούς επιστημονικών υπολογισμών, θα τους κατατάξουμε σε δύο ομάδες και θα πραγματοποιήσουμε μια αδρή αλλά λεπτομερή ανάλυση σε σημαντικούς άξονες μικρο-αρχιτεκτονικής αναδιαμόρφωσης για να καταλήξουμε στο τελικό μοντέλο της ετερογενούς GPU μας.

0.4.1 Διαμορφώνοντας το σύνολο των test

Σύμφωνα με το σενάριό μας, έχουμε επιλέξει μια πληθώρα επιστημονικών υπολογιστικών εφαρμογών. Διαθέτουμε εφαρμογές από διάφορους τομείς, όπως φαίνεται στον πίνακα 2. Πιο συγκεκριμένα, περιλαμβάνουμε:

τα πέντε κύρια συστατικά του framework του Beam Longitudinal Dynamics (BLonD) [52], τον οποίο χρησιμοποιεί το CERN και αποτελείται από αυτόνομα benchmarks

το LULESH [50] [64] που επιλύει ένα απλό πρόβλημα έκρηξης Sedov [116]

το hotspot, ένα εργαλείο θερμικής προσομοίωσης, μέρος της σουίτας rodinia-3.1 [18]

το GASAL2 [3] μια βιβλιοθήκη CUDA που περιέχει αλγορίθμους ευθυγράμμισης ακολουθιών [5] και το test case του που περιέχεται στη σουίτα Genomics-GPU [77]

δύο βασικές εφαρμογές πράξεων γραμμικής άλγεβρας από τη σουίτα αναφοράς polybench polybench: τις υλοποιήσης ενός αλγορίθμου της μεθόδου bi-conjugate gradient (BiCG) [108] και μια αντίστοιχη της διαδικασίας Gram-Schmidt [74]. Οι λειτουργίες γραμμικής άλγεβρας συχνά συμμετέχουν σε ποικίλες επιστημονικές εφαρμογές και οφείλουν να εκπροσωπόνται στο test set μας.

0.4.2 Μεθοδολογία Πειραματισμού

Προσπαθούμε να ομαδοποιήσουμε τους kernels των παραπάνω εφαρμογών βάσει των χαρακτηριστικών εκτέλεσής τους και των στενώσεων (bottlenecks)

Benchmark Name	Benchmark Suite	Domain	#kernels
convolution	BLonD	Particle Physics	1
histogram	BLonD	Particle Physics	1
kick	BLonD	Particle Physics	1
interpolation-kick	BLonD	Particle Physics	2
drift	BLonD	Particle Physics	1
LULESH	(Standalone)	Hydrodynamics	26
hotspot	rodinia	Thermal Physics	1
GASAL2.0	(Standalone)	Bionformatics	5
gramschmidt	polybench-gpu	Linear algebra	3
bicg	polybench-gpu	Linear algebra	2

Table 2: Benchmarks επιστημονικών εφαρμογών που χρησιμοποιήθηκαν για δοκιμή και αξιολόγηση παραμετροποιήσεων

που αντιμετωπίζουν προκειμένου να προσαρμόσουμε δύο τύπων SM βάσει των αναγκών των δύο ομάδων που θα δημιουργήσουμε. Προτείνουμε την ακόλουθη κατηγοριοποίηση:

- **Compute Intensive Kernels:** Αυτή η κατηγορία περικλείει kernels που καταπονούν έντονα το υπολογιστικό backend του αγωγού SMs, οι οποίοι θα μπορούσαν να εκμεταλλευτούν ακόμη περισσότερο τον παραλληλισμό σε επίπεδο νήματος εντός του SM, αλλά η απόδοσή τους περιορίζεται από την έλλειψη πόρων σχετικών με τον υπολογισμό.

- **Low-Utiliation Kernels:** Οι kernels που τοποθετούνται σε αυτή την ομάδα θα υπο-αξιοποιήσουν τον υπολογιστικό αγωγό κατά την εκτέλεσή τους, σπαταλώντας ουσιαστικά πόρους σε όλη τη GPU, χωρίς να αφήνουν περιθώρια για σημαντική βελτίωση των επιδόσεων λόγω της μη ευαισθησίας τους σε μικρο-αρχιτεκτονικές προσαρμογές. Συνήθεις λόγοι πίσω από αυτή την αναισθησία μπορεί να είναι ότι ο πρωταρχικός λόγος περιορισμού της απόδοσης είτε βρίσκεται εκτός SM (π.χ. DRAM) είτε ότι η ίδια η διαρρύθμιση του kernel δεν επιτρέπει υψηλή αξιοποίηση του υλικού (π.χ. μικρό μέγεθος πλέγματος kernel)

Έχοντας ορίσει τις δύο κατηγορίες-στόχους των kernels πρέπει τώρα να αλύσουμε τις εφαρμογές επιστημονικών υπολοιγισμών HPC ούτως ώστε να δημιουργήσουμε ένα set από benchmarks και να κατηγοριοποιήσουμε τους kernels του στα δύο προαναφερθέντα groups.

Επειδή τα compile-time χαρακτηριστικά των kernels παρέχουν περιορισμένη πληροφορία σχετικά με τη χρησιμοποίηση του hardware προσεγγίζουμε

την κατηγοριοποίηση των kernels ως εξής. Δοκιμάζουμε εκτενώς τους διαθέσιμους kernels σε στοχευμένες τροποποιημένες εκδόσεις της V100 GPU η οποία χρησιμεύει ως η βάση για τις παρατηρήσεις μας (baseline). Στον πίνακα 5.2 παρουσιάζονται τα κύρια χαρακτηριστικά υλικού της. Αυτές είναι οι τιμές που χρησιμοποιούνται στον προσομοιωτή και δεν συσχετίζονται απαραίτητα 1:1 με την πραγματική GPU ωστόσο είναι διαμορφωμένες έτσι ώστε να δίνουν το κοντινότερο αποτέλεσμα στο πραγματικό hardware. Οι δοκιμές εκτελούνται στο περιβάλλον προσομοίωσης PTX που προσφέρεται από τον Accel-Sim [67] και τα αναφερόμενα στατιστικά στοιχεία προσομοίωσης χρονισμού συλλέγονται μαζί με μετρήσεις ισχύος από τον AccelWattch [63]. Στα ραβδογράμματα που ακολουθούν κάθε ζεύγος ράβδων για έναν συγκεκριμένο kernel αντιπροσωπεύει μια δοκιμασμένη διαμόρφωση. Οι μπάρες μπλε απόχρωσης αντιπροσωπεύουν την επιτάχυνση, ενώ οι μπάρες πορτοκαλί απόχρωσης αντιπροσωπεύουν τη, σε σχέση με τη βασική γραμμή, μέση κατανάλωση ισχύος καθ' όλη τη διάρκεια εκτέλεσης του kernel. Τόσο η βασική όσο και η δοκιμασμένη διαμόρφωση θα διαθέτουν 40 SMs (τα μισά από τα αρχικά 80), εκτός αν αναφέρεται διαφορετικά, καθώς η ομαδοποίηση θα αποσκοπεί στην προσαρμογή των αντίστοιχων SMs κάθε κατηγορίας για αυτή τη νέα αρχιτεκτονική, η οποία θα διαθέτει ένα κλάσμα των αρχικών SMs για κάθε τύπο υπολογιστικού πυρήνα. Επιπλέον, η προσομοιωμένη, σταθερή καθυστέρηση εκκίνησης kernel θα μηδενιστεί σε όλες τις διαμορφώσεις. Αυτές οι δύο αλλαγές είναι οι μόνες που ενσωματώνονται στο baseline μοντέλο μας, διαφοροποιώντας το από τη δοκιμασμένη διαμόρφωση V100. Αφού προσδιοριστούν οι ομάδες στις οποίες εμπίπτει κάθε kernel, μπορούν να επιχειρηθούν πιο συγκεκριμένες στοχευμένες αλλαγές στο υλικό και να δοκιμαστούν με τον ίδιο τρόπο, οδηγώντας στην τελική διαμόρφωση των SM.

Σε αυτό το σημείο είναι σημαντικό να σημειωθούν οι περιορισμοί του μοντέλου ισχύος που χρησιμοποιείται στον AccelWattch [63] [63]. Συγκεκριμένα, ο AccelWattch [63] λειτουργεί λαμβάνοντας μετρητές επιδόσεων από το μοντέλο χρονισμού του Accel-Sim [67], όσον αφορά τις προσπελάσεις σε συγκεκριμένα τμήματα του υλικού, και στη συνέχεια χρησιμοποιεί προ-υπολογισμένους συντελεστές κλιμάκωσης για την εκτίμηση της κατανάλωσης ισχύος. Η στατική και η σταθερή ισχύς υπολογίζονται επίσης με βάση μετρήσεις από την πραγματική GPU που μοντελοποιείται, όπως εξηγείται στην υπό-ενότητα 0.2.2. Παρόλο που αυτό οδηγεί σε ακριβή αποτελέσματα προσομοίωσης ισχύος κατά τη δοκιμή εφαρμογών έναντι επικυρωμένων διαμορφώσεων για υπάρχουσες GPU, παρέχει φτωχή εξερεύνηση του χώρου σχεδίασης, καθώς οι παράγοντες κλιμάκωσης και οι προ-υπολογισμένες τιμές για τη στατική και τη σταθερή ισχύ δεν μπορούν να προσαρμοστούν με βεβαιότητα κατά την τροποποίηση της μικρο-αρχιτεκτονικής διαμόρφωσης της GPU. Παρ' όλα αυτά, ο AccelWattch [63] και ο Accel-Sim [67] είναι τα πλέον σύγχρονα εργαλεία

Cores	40
Core Clock	1447MHz
RF Size	256KB/Core
RF Banks/Core	16
Max Threads/Core	2048
Warp Size	32
INT,DP,SP,SFU,Tensor Units/Core	4
Warp Scedulers/Core	4
Issue Width	1
Collector Units/Core	8 8x8ports
Unified L1/Shared mem/Core	32KB/96KB
Shared mem banks/Shared	32
L2 cache	6MB (96KB per partition)

Table 3: Διαμόρφωση της baseline V100 που χρησιμοποιούμε στις δοκιμές μας.

προσομοίωσης GPU ανοικτού κώδικα και παρόλο που οι μετρήσεις ισχύος μπορεί να μην αντικατοπτρίζουν πραγματικά το προσομοιωμένο υλικό όταν εφαρμόζονται αλλαγές, μπορούν να παρέχουν ένα προσεγγιστικό άνω όριο, καθώς μέσω της χρήσης μετρητών επιδόσεων, η αυξημένη χρήση του δυνητικά διευρυμένου υλικού θα μεταφράζεται σε υψηλότερη κατανάλωση ισχύος.

Επιλέγουμε να μελετήσουμε kernels και όχι ολόκληρες τις εφαρμογές, δεδομένου ότι αυτοί θα αποτελέσουν την μονάδα χρονοπρογραμματισμού για κάθε τύπο SM στο στοχευμένο σενάριο. Για όλα τα πειράματα εκτελούμε τα benchmarks μέχρι τέλους στο baseline μοντέλο, με εξαίρεση το GASAL που περιορίζεται σε 100 εκατομμύρια εντολές νήματος λόγω του πολύ μεγάλου χρόνου προσομοίωσής του. Κρατάμε τουλάχιστον έναν kernel από κάθε μία από τις παραπάνω εφαρμογές ενώ παράλληλα παραλείπουμε αυτούς που καταλαμβάνουν αμελητέα τμήματα του συνολικού χρόνου εκτέλεσης της κάθε εφαρμογής όπως αυτός προκύπτει για εκτέλεση στο baseline μοντέλο. Για το lulesh που περιέχει 26 kernels διατηρούμε μόνο τους τρεις σημαντικότερους βάσει χρόνου εκτέλεσης. Μετά από αυτήν τη διαδικασία το δοκιμαστικό σύνολό μας αποτελείται από τους 18 παρακάτω kernels:

Έχοντας καθορίσει πλήρως το σύνολο δοκιμών, το πρώτο βήμα που κάνουμε είναι να επιθεωρήσουμε τα μίγματα εντολών των πυρήνων μας μέσω εκτέλεσης στον προσομοιωτή (Σχήμα 10), παρέχοντας μας μια πρώτη εικόνα για το πώς συμπεριφέρεται ο κάθε kernel και πώς μπορεί να στοχευθεί για βελτιστοποίηση. Τρεις από τους δεκαοκτώ kernels αποτελούνται τουλάχιστον 55% από εντολές μνήμης και άλλοι πέντε από πάνω από 25%. Όσο υψηλότερο

Kernel Name	Abbreviation	Data size	Block Size	Grid Size
CalcHourglassControlForElems	LUL1	46 edge nodes	(256,1,1)	(2848,1,1)
CalcFBHourglassForceForElems	LUL2	46 edge nodes	(256,1,1)	(2848,1,1)
CalcKinematicsForElems	LUL3	46 edge nodes	(256,1,1)	(356,1,1)
convolution	BLco	(nsignal, nkernel)=(1000, 1000)	(64,1,1)	(512,1,1)
histogram	BLhi	nparticles=1000000	(1024,1,1)	(512,1,1)
drift	BLdr	nparticles=1000000	(64,1,1)	(512,1,1)
kick	BLki	nparticles=1000000	(64,1,1)	(512,1,1)
linear_interp_kick	BLik	Bionformatics	(1024,1,1)	(512,1,1)
calculate_temp	HOT	grid=(512,512)	(16,16,1)	(43,43,1)
bicg_kernel1	BIC1	grid=(1024,1024)	(256,1,1)	(4,1,1)
bicg_kernel2	BIC2	grid=(1024,1024)	(256,1,1)	(4,1,1)
gramschmidt_kernel1	GRA1	grid=(512,512)	(256,1,1)	(1,1,1)
gramschmidt_kernel3	GRA3	grid=(512,512)	(256,1,1)	(2,1,1)
gasal_pack	GApa	default query and target [3]	(128,1,1)	(40,1,1)
gasal_ksw	GAksw	default query and target [3]	(128,1,1)	(40,1,1)
gasal_local	GAlo	default query and target [3]	(128,1,1)	(40,1,1)
gasal_semi_global	GAsg	default query and target [3]	(128,1,1)	(40,1,1)
gasal_global	GAgl	default query and target [3]	(128,1,1)	(40,1,1)

Table 4: Δοκιμαστικό σύνολο 18 kernels από πολλαπλές εφαρμογές και πεδία επιστημονικών υπολογισμών

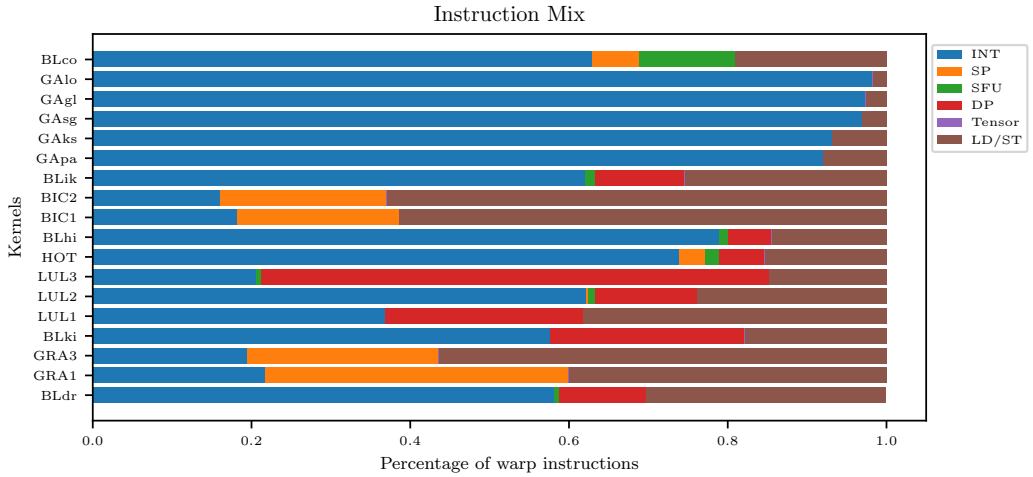


Figure 10: Μίγμα εντολών 18 μελετημένων HPC kernels

είναι το τμήμα ενός kernel που καταλαμβάνουν οι εντολές μνήμης, τόσο πιθανότερο είναι ο kernel αυτός να δεσμεύεται από περιορισμούς όπως το εύρος ζώνης ή η καθυστέρηση της μνήμης. Για τους kernels που διαθέτουν κάτω από 55% εντολές μνήμης είναι προφανές ότι κυριαρχούν οι εντολές ακεραίων. Αυτό είναι κατανοητό αν σκεφτεί κανείς ότι όλες οι εντολές που σχετίζονται με τον έλεγχο εκτελούνται στον αγωγό ακεραίων. Ο συνδυασμός αυτών μαζί με τις πράξεις ακέραιου αριθμού στην πραγματική διαδρομή δεδομένων του προγράμματος αποτελεί την πλειοψηφία του προγράμματος σε όλες σχεδόν τις περιπτώσεις που μελετήθηκαν. Ακολουθούν οι πράξεις διπλής ακρίβειας. Λαμβάνοντας υπόψη το γεγονός ότι οι επιλεγμένοι kernels ανήκουν στον τομέα HPC, αναμένεται ότι οι kernels θα εκφορτώσουν μεγάλο μέρος της εκτέλεσής τους στον αγωγό DP, δεδομένου ότι οι επιστημονικοί υπολογισμοί περιλαμ-

βάνουν συνήθως αριθμητική κινητής υποδιαστολής υψηλής ακρίβειας [11]. Οι εντολές SP είναι σημαντικές μόνο σε μία περίπτωση μη κορεσμένης μνήμης και οι εντολές SFU καταλαμβάνουν αμελητέο ποσοστό των συνολικών εντολών, ενώ οι τανυστές απουσιάζουν εντελώς οπότε θα αγνοηθούν στο υπόλοιπο της παρούσας διατριβής.

Για να ελέγξουμε για συμφορήσεις μνήμης χρησιμοποιούμε την τέλεια διαμόρφωση μνήμης, την ίδια μέθοδο που χρησιμοποιήσαμε στην υπο-ενότητα 0.1.2. Αυτή είναι μια εντελώς μη ρεαλιστική προσέγγιση από πλευράς υλοποίησης, καθώς εξαλείφει σχεδόν εξ ολοκλήρου όλους τους περιορισμούς μνήμης, επομένως θέτουμε τον πήχη ψηλά όσον αφορά την αύξηση της απόδοσης που απαιτείται για να δικαιολογήσουμε την ταξινόμηση ενός kernel ως περιορισμένης μνήμης. Οι kernels που βελτιώνονται με αυτή τη διαμόρφωση κατά τουλάχιστον 40% θα ταξινομηθούν ως μέτρια περιορισμένοι στη μνήμη και εκείνοι που διπλασιάζουν την απόδοσή τους θα θεωρηθούν ισχυρά περιορισμένοι στη μνήμη. Όπως παρατηρούμε στο Σχήμα 11 έξι kernels τουλάχιστον διπλασιάζουν την απόδοσή τους με αυτή τη διαμόρφωση και επομένως θα πρέπει να τοποθετηθούν στη δεύτερη κατηγορία καθώς δεν αναμένεται να βελτιωθούν με περισσότερους υπολογιστικούς πόρους. Αυτή η παρατήρηση είναι εν μέρει σύμφωνη με την προηγούμενη παρατήρησή μας σχετικά με το μίγμα εντολών με εξαιρέσεις τον BIC2 που παρουσιάζει υψηλό ποσοστό εντολών μνήμης αλλά καμία βελτίωση με τέλεια μνήμη και τον GRA1 και τον LUL3 που παρουσιάζουν την αντίθετη συμπεριφορά. Αυτές οι εξαιρέσεις υπογραμμίζουν την ανάγκη για σκιαγράφηση των επιδόσεων σε χρόνο εκτέλεσης. Kernels με λιγότερο αισθητή βελτίωση σίγουρα χρίζουν περαιτέρω ανάλυσης. Όσοι δεν εμφανίζουν καμία βελτίωση δεν θα θεωρηθούν δεσμευμένοι ως προς τη μνήμη. Υπάρχει πολύ μικρό περιθώριο βελτίωσης για kernels περιορισμένους από τη μνήμη, μέσω μικρο-αρχιτεκτονικών αλλαγών εντός του SM, ενώ οι καθυστερήσεις που αντιμετωπίζουν οδηγούν σε υπο-εκμετάλλευση του υλικού. Ως εκ τούτου, αυτοί οι kernels σε πρώτη φάση κατηγοριοποιούνται ως Low-Utiliation.

Το μοντέλο υπο-πυρήνων περιορίζει τις αλλαγές που μπορούμε να εισάγουμε στα SM οπότε για πολλές παραμετροποιήσεις όπως η παρακάτω χρειάζεται να το απενεργοποιήσουμε. Οι επιπτώσεις αυτής της αλλαγής θα συζητηθούν περαιτέρω στην υπο-ενότητα 0.4.3. Στη συνέχεια, αναζητούμε kernels που απαιτούν μεγάλη υπολογιστική ισχύ. Δεδομένου ότι έχουμε ήδη διαπιστώσει ότι σχεδόν σε όλους τους kernels ο κυρίαρχος τύπος εντολής είναι η ακέραια πράξη, αυξάνουμε τις ακέραιες λειτουργικές μονάδες (INT FUs) κατά τέσσερα οκτώ και δεκαέξι σε σχέση με τα αρχικά τέσσερα. Αναλόγως κλιμακώνουμε τους καταχωρητές αγωγού ώστε να αντιστοιχούν σε αυτές τις λειτουργικές μονάδες καθώς και τις μονάδες συλλογής τελεστέων και τα banks αρχείου καταχωρητών. Το Σχήμα 11 αποκαλύπτει ότι τέσσερις από τους kernels μας παρουσιάζουν αξιοσημείωτη αύξηση της απόδοσης όταν εκτελούνται με αυτή τη

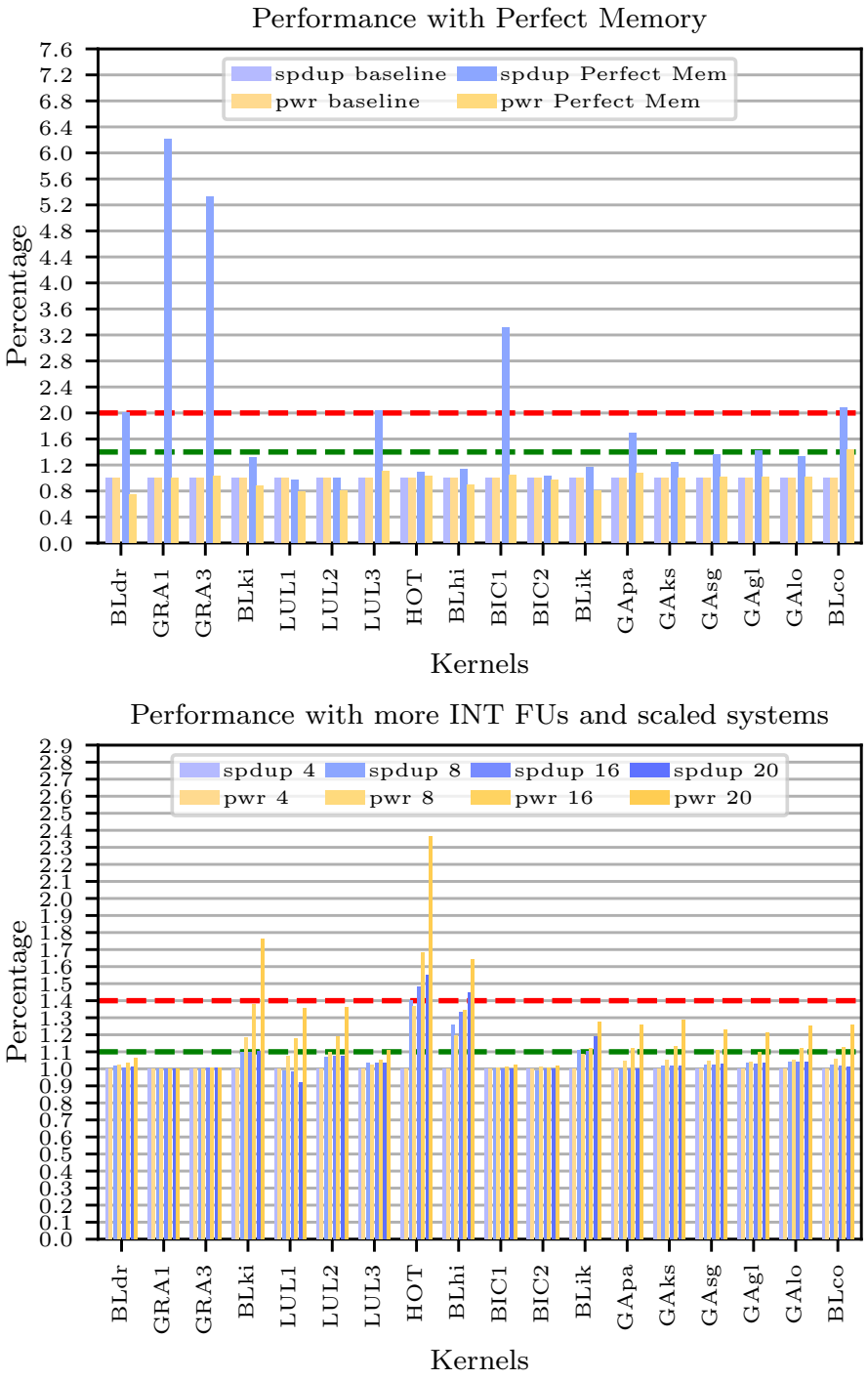


Figure 11: Συμπεριφορά kernels με τέλεια μνήμη και μεγαλύτερο υπολογιστικό pipeline εστιασμένο σε υπολογισμούς ακεραίων

διαμόρφωση. Δεδομένου ότι αυτή η τροποποίηση δεν είναι τόσο επιθετική όσο το τέλειο υποσύστημα μνήμης, θεωρούμε σημαντική την επιτάχυνση από 40% και πάνω, ενώ θέτουμε το κατώτατο όριο στο 10%. Αυτό το όριο ορίζει τους kernels HOT και BLhi ως ισχυρά Compute-Intensive ενώ οι kernels BLki και BLik τοποθετούνται στην ίδια κατηγορία αλλά ασθενώς.

Απαιτείται περεταίρω εξέταση για τους kernels για τους οποίους υπάρχουν βάσιμες αμφιβολίες για την κατηγορία που εντάχθηκαν και για αυτούς που ακόμη δεν έχουν ενταχθεί σε κάποια. Για τους kernels του GASAL και τον BIC2 που είναι ανένταχτοι παρατηρούμε στην παραμετροποίηση εκτέλεσής τους ότι χρησιμοποιούνται πολύ μικρά grids ανίκανα σε συνδυασμό με το μέγεθος block να προσφέρουν αρκετά warps για ικανό παραλληλισμό και κρύψιμο καθυστερήσεων. Οπότε αυτοί ορίζονται ως Low-Utiliation.

Ο LUL3 παρά το επαρκές μέγεθος grid-block του, καταλήγει σε 12.5% πληρότητα στην κάρτα οδηγώντας σε παρόμοια προβλήματα με την προηγούμενη περίπτωση. Ωστόσο τώρα η χαμηλή πληρότητα οφείλεται στο μέγεθος του αρχείου καταχωρητών, το οποίο είναι ανίκανο να στεγάσει τους καταχωρητές ανά thread που απαιτεί ο αλγόριθμος. Κάποιοι από τους Compute-Intensive kernels επίσης εμφανίζουν τέτοια συμπεριφορά όπως ο HOT αλλά σε μικρότερο βαθμό. Εφόσον μεγαλύτερο αρχείο καταχωρητών θα ωφελούσε τη συμπεριφορά τέτοιων kernels και αφού είναι άρρηκτα συνδεδεμένο με το υπολογιστικό pipeline, ο LUL3 τοποθετείται στους Compute-Intensive.

Οι LUL1 και LUL2 έχουν και αυτοί θέμα πληρότητας λόγω του αρχείου καταχωρητών ωστόσο όπως φανερώνει το σχήμα 18 ακόμη και όταν τους δίνονται αρκετοί ώστε να επιτύχουν τη μέγιστη δεν υπάρχει βελτίωση στην επίδοσή τους. Περαιτέρω ανάλυση φανερώνει πώς αμφότεροι και ειδικά ο LUL2 έχουν περιορισμένα διαθέσιμα ενεργά warps καθώς παρά την υψηλή πληρότητα που επιτυγχάνουν, κατά την εκτέλεσή τους πολλά warps καταλήγουν αδρανή πάνω σε φράγματα. Αυτό οδηγεί σε αντίστοιχα προβλήματα με εκείνους που έχουν λίγα διαθέσιμα warps για άλλους λόγους και εμφανίζονται αναίθητοι σε αύξηση των υπολογιστικών πόρων των SM που τους φιλοξενούν. Επιπλέον ανάλυση των λόγων για τους οποίους οι χρονοπρογραμματιστές οδηγούνται να μην στείλουν κάποιο warp για εκτέλεση (stall) υποδεικνύει πως αυτοί οι kernels περιορίζονται από δυσεπίλυτες εξαρτήσεις δεδομένων. Επομένως και αυτοί καταλήγουν στην κατηγορία των Low-Utiliation.

Πλέον όλοι οι kernels μας έχουν κατηγοριοποιηθεί όπως φαίνεται στον πίνακα 5.

Ο τελικός χρήστης της πλατφόρμας δε θα χρειαστεί να επωμιστεί αυτές τις δυσκολίες κατηγοριοποίησης καθώς διαθέτοντας την πλατφόρμα αρκεί να δοκιμάσει την επίδοση των εφαρμογών του στα δύο είδη SM και να κρίνει.

Compute-Intensive	Low-Utiliation		
HOT	BLdr	BIC2	GAlo
BLhi	BLco	GRA1	GAsg
BLik	LUL1	GRA3	GAgl
BLki	LUL2	GApa	
LUL3	BIC1	GAksw	

Table 5: Τελική κατηγοριοποίηση των kernels

0.4.3 Μελέτη διαμορφώσεων SM

Διαθέτοντας πλέον τις δύο κατηγορίες αναζητούμε άξονες βελτίωσης.

Το πιο συνηθισμένο χαρακτηριστικό που εντοπίζουμε στους Compute-Intensive kernels είναι η έλλειψη ενός ή περισσότερων πόρων υλικού να παρεμποδίζει την εκτέλεση. Αυτά τα σημεία συμφόρησης μπορούν να αντιμετωπιστούν με την προμήθεια των υπολογιστικών πυρήνων με το απαιτούμενο υλικό. Στόχος είναι η επίτευξη καλύτερης απόδοσης, διατηρώντας παράλληλα την καταναλισκόμενη ισχύ σε λογικά επίπεδα, ώστε να αυξηθεί η αποδοτικότητα της GPU. Οι πυρήνες που θα στοχεύουν αυτήν την κατηγορία kernels θα ονομαστούν High-Performance (HP). Για τους Low-Utilization kernels ακολουθούμε την αντίστροφη προσέγγιση. Γνωρίζοντας ότι η συμφόρηση που αντιμετωπίζουν δεν επιλύεται εύκολα με αλλαγές εντός του SM το οποίο δε χρησιμοποιείται στο μέγιστο των δυνατοτήτων του, εξετάζουμε τη σμίκρυνση του pipeline τους με την υπόθεση ότι η απόδοσή τους δεν θα επηρεαστεί σημαντικά. Ωστόσο, τα κατά τα άλλα αχρησιμοποίητα εξαρτήματα που αφαιρούνται δεν θα καταναλώνουν πλέον ενέργεια και θα απελευθερώσουν χώρο επιτρέποντάς μας να επεκτείνουμε τους πυρήνες που εστιάζουν στους Compute-Intensive kernels. Αυτοί θα είναι οι Low-Power (LP) πρήνες της αρχιτεκτονικής μας. Συνολικά, ο στόχος είναι η ανάπτυξη μιας αποδοτικής πλατφόρμας που θα επιτρέπει στους Compute-Intensive kernels να εκμεταλλεύονται τον έμφυτο παραλληλισμό τους σε επίπεδο warp και να εκτελούνται ταχύτερα, ενώ οι kernels που δεν αξιοποιούν επαρκώς το υλικό θα αποδίδουν σχεδόν το ίδιο καλά με το βασικό μοντέλο, ωστόσο πιο οικονομικά, οδηγώντας τόσο σε ταχύτερη συνολική εκτέλεση όσο και σε μειωμένη κατανάλωση ενέργειας για ολόκληρη την GPU.

Καθώς υπάρχει ανάγκη να μη χρησιμοποιηθεί για πολλές από τις παρακάτω δοκιμές το μοντέλο υπό-πυρήνα θα μελετηθεί πρώτο. Η απενεργοποίηση αυτής της ρύθμισης του πυρήνα σημαίνει ότι όλοι οι χρονοπρογραμματιστές warp θα μοιράζονται πλέον δομές όπως ο συλλέκτης τελεστέων και οι σωληνώσεις εκτέλεσης στο σύνολό τους. Αυτό μπορεί να οδηγήσει σε καλύτερη αξιοποίηση των πόρων αλλά απαιτεί περισσότερες διασυνδέσεις με αποτέλεσμα την αύξηση της έκτασης και της ισχύος. Σε όλες τις περιπτώσεις, εκτός από την περίπτωση

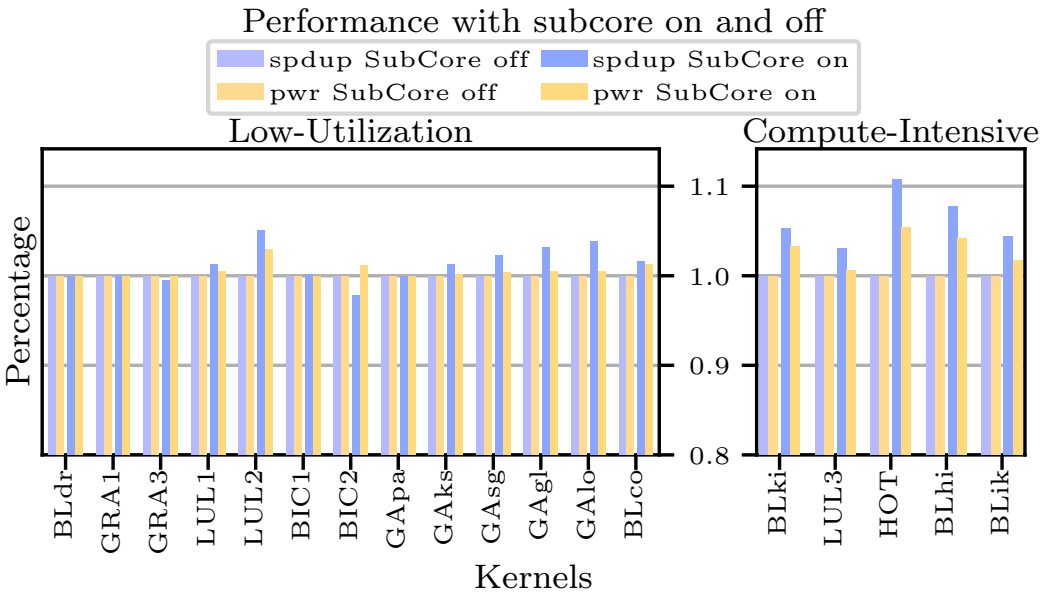


Figure 12: Συμπεριφορά των kernels με και χωρίς το μοντέλο υπο-πυρήνα

δύο kernels η αύξηση της ισχύος που προκαλείται από την υψηλότερη χρησιμοποίηση των μονάδων του SM λόγω αυτής της αλλαγής είναι μικρότερη από την αύξηση της απόδοσης, καθιστώντας αυτή την αναγκαία αλλαγή υποφερτή (Σχήμα 12).

Έπειτα θα εξετάσουμε πώς θα αντιμετωπίζονται οι λειτουργικές μονάδες κάθε πυρήνα. Όσον αφορά τα High-Performance SMs το πρώτο και σημαντικότερο τμήμα του αγωγού που πρέπει να κλιμακώσουμε είναι ο ακέραιος αγωγός. Παρατηρήσαμε στο σχήμα 11 σημαντική βελτίωση με τον διπλασιασμό τους και περιθώρια βελτίωσης με κάποια πρόσθετη επέκταση έως και τριπλάσια της αρχικής χωρίς να υπερβάλλουμε σε όρους ισχύος σε σχέση με το baseline μοντέλο. Καθώς είναι ο πιο δημοφιλής τύπος εντολών θα χρησιμοποιήσουμε 12 μονάδες εκτέλεσης ακεραίων στους kernels με έμφαση στον υπολογισμό. Για τις μονάδες διπλής ακρίβειας τα οφέλη από την προσθήκη περισσότερων με ταυτόχρονη κλιμάκωση σχετικών τμημάτων του αγωγού όπως στην περίπτωση των INT FUs ήταν μικρότερα (Σχήμα 13) και θα τις αυξήσουμε πιο συντηρητικά κατά 50%. Δεδομένου ότι δεν παρατηρούνται δομικοί κίνδυνοι στον αγωγό SFU ή SP και το μίγμα εντολών υποδηλώνει ότι οι εντολές αυτές είναι σπάνιες δεν θα αυξήσουμε τις αντίστοιχες λειτουργικές μονάδες. Επίσης διπλασιάζουμε τα στάδια του αγωγού εντολών μνήμης και τετραπλασιάζουμε το κοινό στάδιο writeback στο οποίο οδηγούν όλοι οι λωρίδες εκτέλεσης προκειμένου να συμβαδίσουμε με το υψηλότερο throughput που θα επιτευχθεί.

Όπως παρατηρήσαμε η περικοπή των μονάδων SFU και SP ακόμη και σε

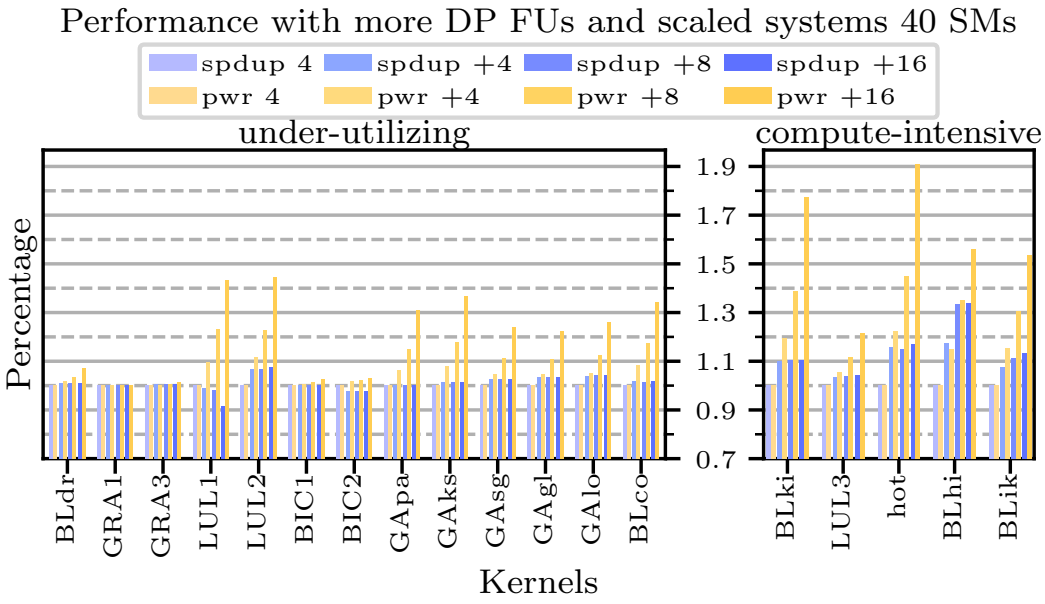


Figure 13: Συμπεριφορά των kernels με μεγαλύτερο υπολογιστικό αγωγό εστιασμένο σε πράξεις διπλής ακρίβειας

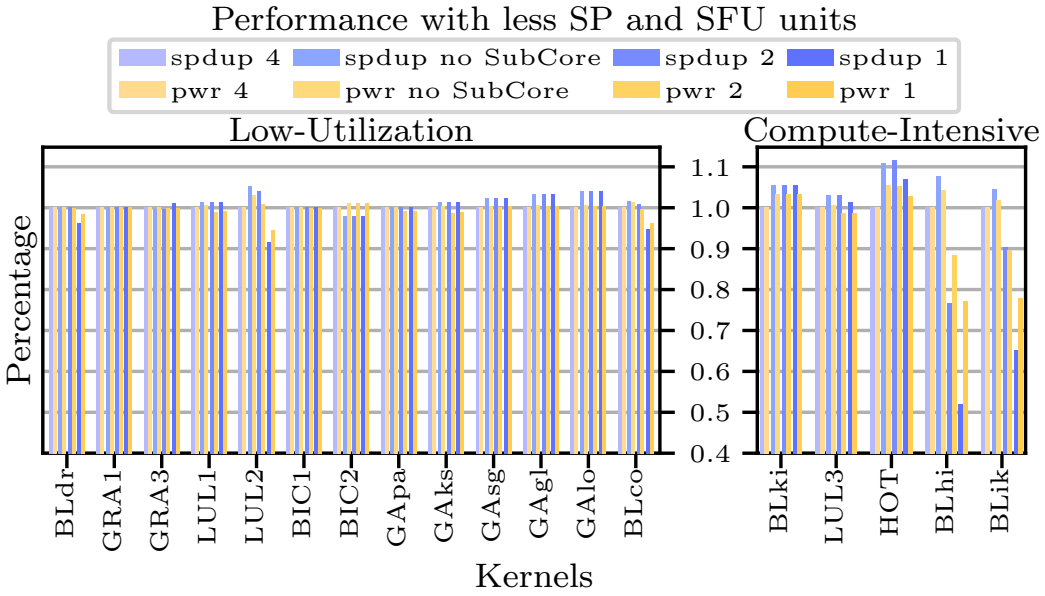


Figure 14: Συμπεριφορά των kernels με λιγότερες SP και SFU υπολογιστικές μονάδες

μία μόνο αυξάνει ελάχιστα το χρόνο εκτέλεσης στη μέση περίπτωση των Low-Utilization kernels (Σχήμα 14), ενώ μειώνει το αρχικό υλικό της κάρτας κατά

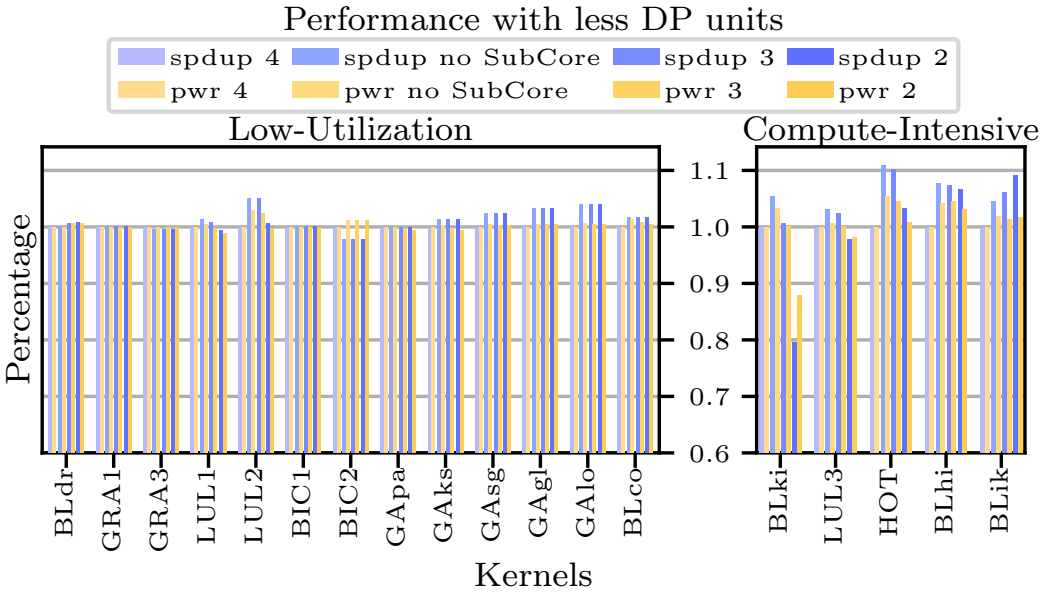


Figure 15: Συμπεριφορά των kernels με λιγότερες DP υπολογιστικές μονάδες

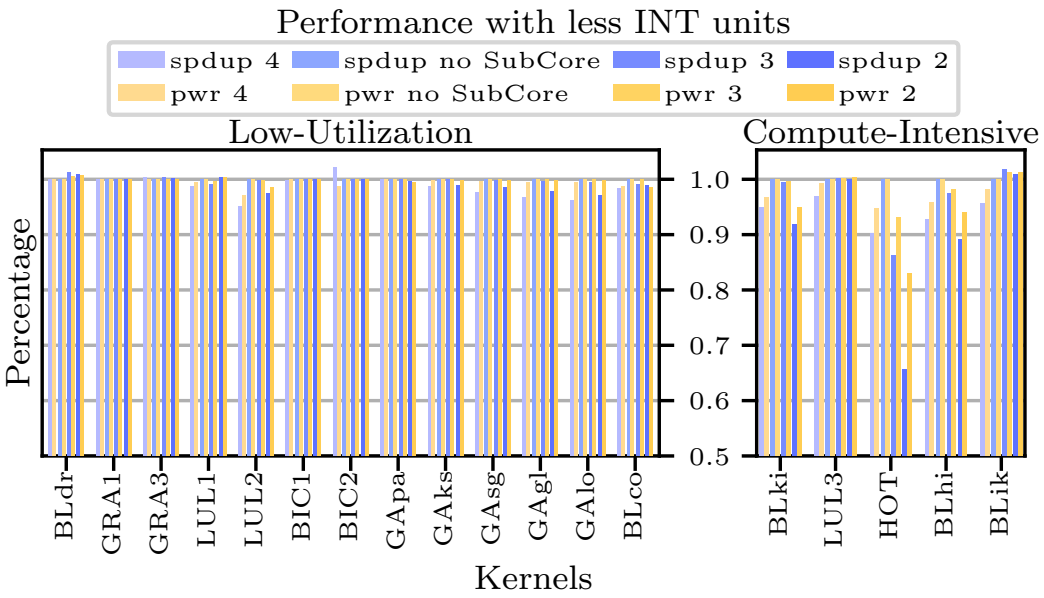


Figure 16: Συμπεριφορά των kernels με λιγότερες INT υπολογιστικές μονάδες

ένα σημαντικό ποσοστό. Γι' αυτό και τα Low-Power SMs θα περιέχουν μόνο μία SFU και μία SP μονάδα. Η μείωση των λειτουργικών μονάδων DP από τις αρχικές τέσσερις σε δύο είχε ελάχιστη επίπτωση στην απόδοση στο των Low-Utiliation kernels (Σχήμα 15. Επειδή πρόκειται για πιο δημοφιλή τύπο

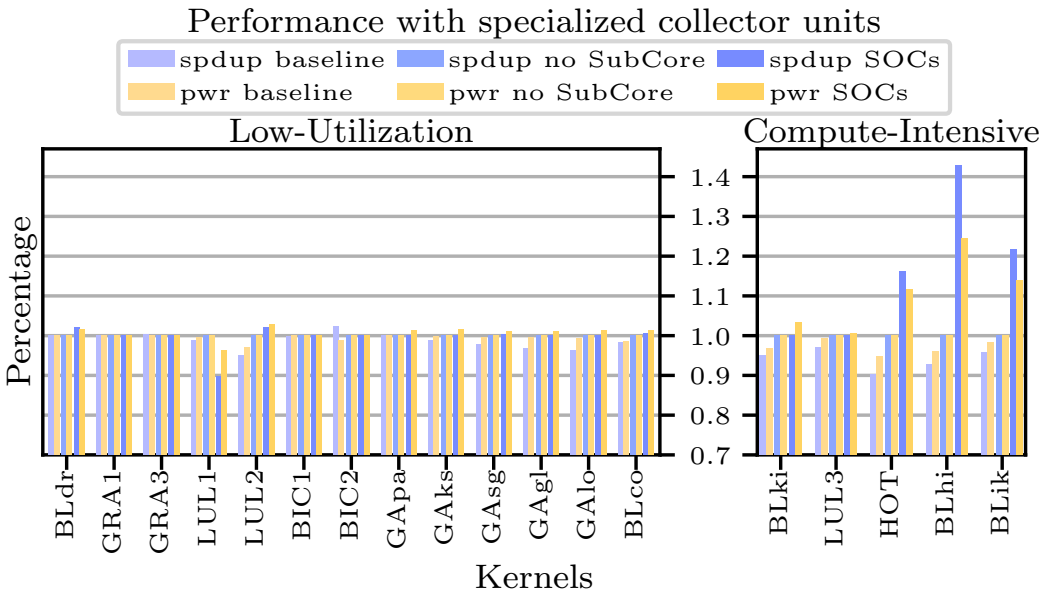


Figure 17: Συμπεριφορά των kernels με οχτώ γενικού σκοπού, οχτώ θυρών μονάδες συλλογής τελεστέων και με προσαρμοσμένες αναλογίες εξειδικευμένων μονάδων

εντολών μειώνουμε τις υπολογιστικές μονάδες σε δύο και όχι μία. Για τον ίδιο λόγο μειώνουμε τις μονάδες Integer μόνο κατά μία, παρά το γεγονός ότι η μελέτη μας δείχνει ότι η μείωσή τους κατά δύο δεν επηρεάζει ιδιαίτερα την απόδοση 16. Σε όλες τις μελέτες με λιγότερες υπολογιστικές μονάδες λαμβάνεται υπόψη η επιρροή της απενεργοποίησης του μοντέλου υπό-πυρήνα όσον αφορά τη σύγκριση των επιδόσεων.

Όπως φαίνεται στο Σχήμα 17, η χρήση εξειδικευμένων, ανά τύπο εντολής, μονάδων συλλογής τελεστέων και η προσαρμογή των μεγεθών τους για κάθε τύπο εντολής με βάση τις απαιτήσεις στον αντίστοιχο αγωγό (16 4x4 INT, 8 1x1 memory, 4 1x1 SP, 4 1x1 SFU, 6 2x2 DP) μπορεί να βελτιώσει δραματικά την απόδοση των Compute-Intensive kernels και για αυτό θα την εφαρμόσουμε στα High-Performance SM. Θα διατηρήσουμε τις γενικευμένες μονάδες συλλογής τελεστεών στους μικρότερους Low-Power πυρήνες μας αλλά θα μειώσουμε κατά 25% τόσο τις ίδιες όσο και τις θύρες εισόδου και εξόδου λόγω του μειωμένου παραλληλισμού αυτών των SMs.

Όσον αφορά το αρχείο καταχωρητών στο οποίο έχουν πρόσβαση αυτοί οι χρονοπρογραμματιστές στρεβλώσεων, πρέπει να ληφθούν υπόψιν δύο παράμετροι. Το μέγεθός και ο αριθμός των banks του. Παρατηρώντας το Σχήμα 18 διαπιστώνουμε ότι ένα αυξημένο μέγεθος επιτρέπει σε kernels με έντονη υπολογιστική δραστηριότητα που έχουν ανάγκη για καταχωρητές

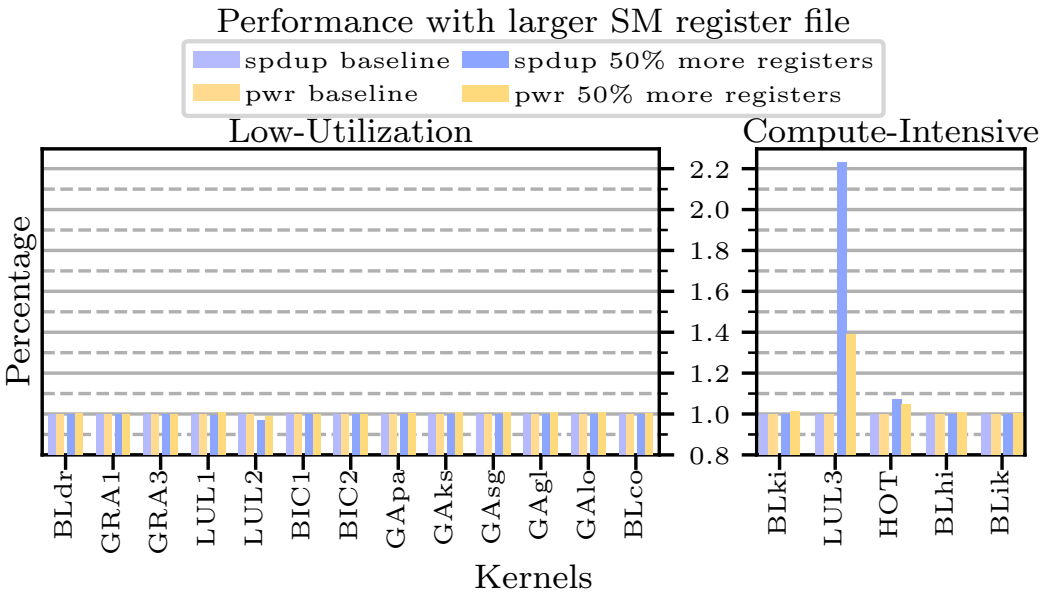


Figure 18: Συμπεριφορά των kernels με μεγαλύτερο αρχείο καταχωρητών

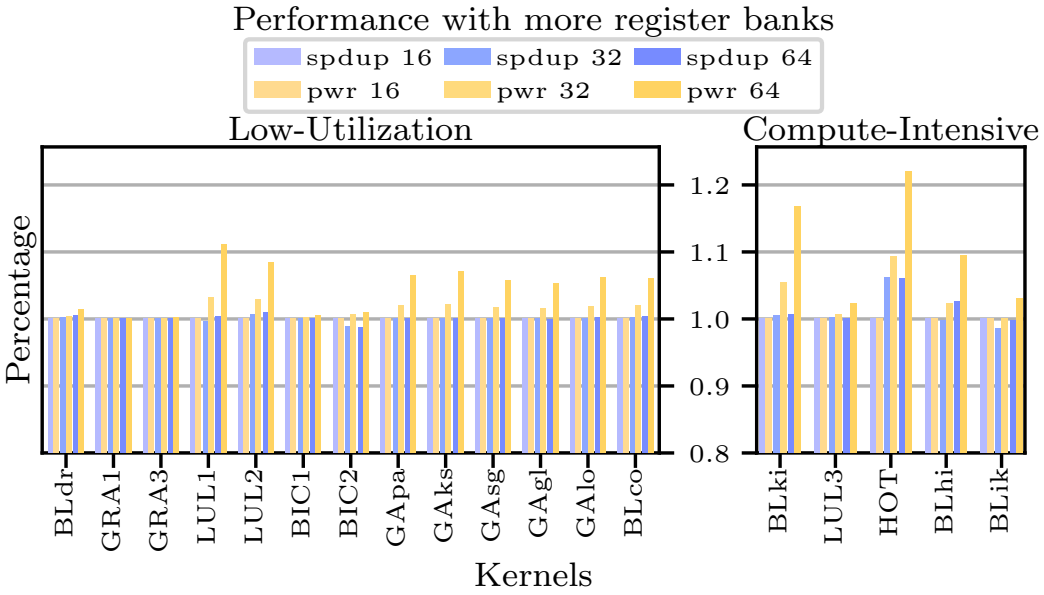


Figure 19: Συμπεριφορά των kernels με περισσότερα banks αρχείου καταχωρητών

να επιτύχουν υψηλότερη πληρότητα που συχνά οδηγεί σε σημαντικά υψηλότερες επιδόσεις. Για το λόγο αυτό θα αυξήσουμε το μέγεθος του αρχείου καταχωρητών κατά 50% στα 384KB για τους ισχυρότερους πυρήνες. Για τους υπόλοιπους θα το διατηρήσουμε σταθερό, ώστε να μην επιδεινώσουμε τα

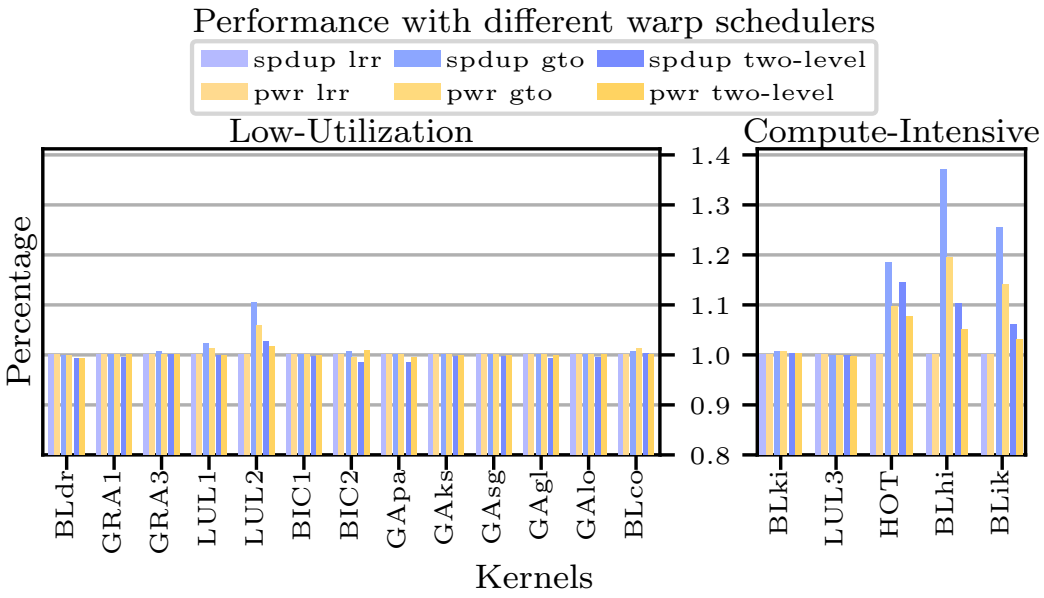


Figure 20: Συμπεριφορά των kernels με Loose Round Robin (LRR) χρονοπρογραμματιστή warp, Greedy then Oldest (GTO) και two-level

ήδη παρατηρούμενα προβλήματα πληρότητας που αντιμετωπίζουν αρκετοί Low-Utiliation kernels. Τα banks αρχείων καταχωρητών πρέπει να τροποποιηθούν με προσοχή, καθώς παρατηρήσαμε ότι παίζουν βασικό ρόλο στην κατανάλωση ενέργειας (Σχήμα 19. Παρόλο που από μόνα τους δεν παρείχαν μεγάλο όφελος για τους Compute-Intensive kernels, τώρα με τα High-Performance SM να παρέχουν περισσότερο παραλληλισμό και τον αυξημένο αριθμό αιτημάτων τελεστών που υποκινείται από την διαμόρφωση του εξειδικευμένου συλλέκτη τελεστών απαιτείται και από το αρχείο καταχωρητών υψηλότερο επίπεδο παράλληλης εξυπηρέτησης αιτημάτων. Αρχείο καταχωρητών με 32 banks σε σύγκριση με το αρχικό των 16 θα εξυπηρετήσει αυτόν τον σκοπό. Αντιστρόφως για τα Low-Power SM , δεδομένου ότι ο μέγιστος διαθέσιμος παραλληλισμός τους μειώνεται από πλευράς εκτέλεσης, θα πρέπει να μειωθεί και στο αρχείο καταχωρητών οπότε μειώνουμε τα banks σε δώδεκα.

Για τους χρονοπρογραμματιστές warp που είναι υπεύθυνοι για την απόφαση του ποιο warp θα εκτελεστεί έχουμε δύο βαθμούς ελευθερίας κατά την προσαρμογή τους για κάθε τύπο kernel. Ο πρώτος είναι ο τύπος τους. Εξετάζουμε τρία είδη: τον Loose Round Robin (LRR) ο οποίος εξετάζει κυκλικά όλα τα warps, τον Greedy Then Oldest (GTO) που δίνει προτεραιότητα στο τελευταίο warp που εκτέλεσε και έναν two-level που διαθέτει δύο επίπεδα των οχτώ warps με το ένα να περιέχει αυτά που αντιμετωπίζουν μεγάλες καθυστερήσεις και το άλλο τα ενεργά τα όποια εξετάζονται για προγραμματισμό με LRR πολιτική. Ο

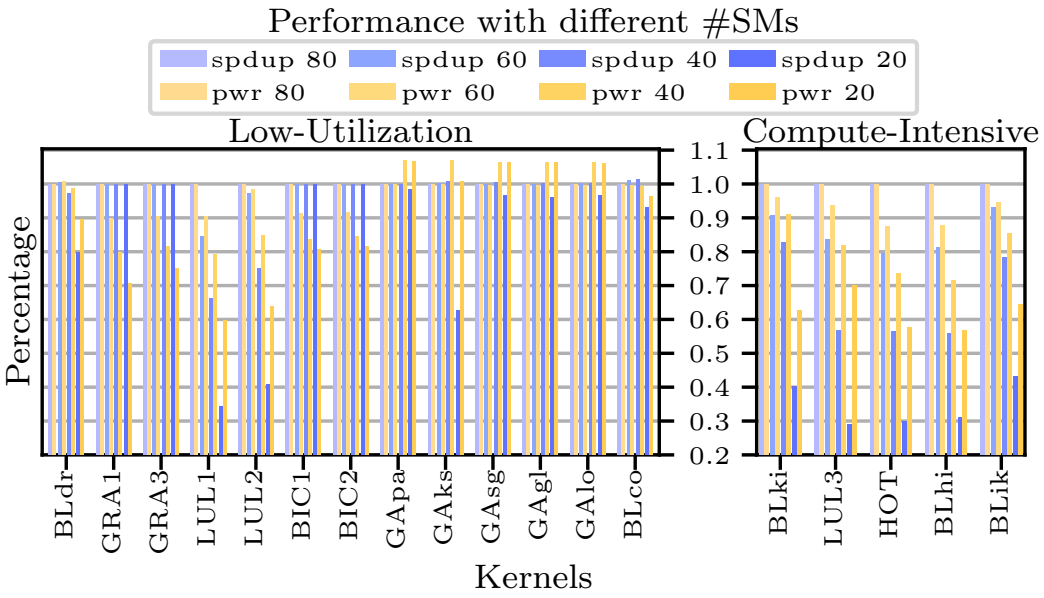


Figure 21: Συμπεριφορά των kernels με διαφορετικό αριθμό ομοιογενών SMs

χρονοπρογραμματιστής GTO υπερτερεί σαφώς έναντι των άλλων δύο κατά την έκδοση warps από Compute-Intensive kernels όπως φαίνεται στο Σχήμα 20 καθιστώντας σαφές ότι θα πρέπει να επιλέξουμε τον χρονοπρογραμματιστή GTO για τους High-Performance πυρήνες, ενώ για Low-Utilization εφαρμογές η επιλογή φαίνεται να είναι αδιάφορη από πλευράς επιδόσεων. Υπό αυτές τις συνθήκες θα διατηρήσουμε τον χρονοπρογραμματιστή LRR για τους Low-Power πυρήνες προκειμένου να προωθήσουμε την ομοιόμορφη πρόοδο μεταξύ των warps, φροντίζοντας να μην επιδεινώσουμε το ζήτημα της χαμηλής διαθεσιμότητας warp καθώς προχωρά η εκτέλεση του kernel. Η δεύτερη επιλογή που πρέπει να κάνουμε όσον αφορά τη διαμόρφωση των χρονοπρογραμματιστών warp είναι ο αριθμός τους. Το αυξημένο επίπεδο παραλληλισμού που επιχειρούμε να εκμεταλλευτούμε στους ισχυρότερους High-Performance πυρήνες μας απαιτεί αυξημένο αριθμό χρονοπρογραμματιστών warps τους οποίους αυξάνουμε κατά 50% για να υπάρχει περισσότερη διαθέσιμη εργασία για τους διευρυμένους αγωγούς. Για τους Low-Power πυρήνες τους μειώνουμε κατά 50% αναγνωρίζοντας τις μειωμένες ανάγκες παραλληλισμού επιπέδου warp των αντίστοιχων kernels και τις μειωμένες δυνατότητες υποστήριξής του από τους στενότερους αγωγούς.

Τέλος όσον αφορά τον αριθμό των SM κάθε τύπου το Σχήμα 21 καταδεικνύει υπο-γραμμική μείωση της απόδοσης σε σχέση με τον αριθμό των SMs ακόμα και για τους Compute-Intensive kernels οι οποίοι υποφέρουν περισσότερο από τους Low-Utiliation. Αυτό είναι κρίσιμο καθώς δίνει νόημα στον διαμοιρασμό

	High-Performance	V100	Low-Power
SMs	40	80	40
Core Clock	1447MHz	1447MHz	1447MHz
RF Size(KB)	384	256	256
RF Banks	32	16	12
Max Threads/Core	2048	2048	2048
INT Units/Core	12	4	3
DP Units/Core	6	4	2
SP Units/Core	4	4	1
SFU Units/Core	4	4	1
Warp Scedulers/Core	6	4	2
Warp Scedulers Type	GTO	LRR	LRR
Collector Units	INT 16 4x4 DP 6 2x2 SP 4 1x1 SFU 4 1x1 MEM 6 1x1	General Purpose 8 8x8	General Purpose 6 6x6
Unified L1/Shared mem	32KB/96KB	32KB/96KB	32KB/96KB

Table 6: Τελική διαμόρφωση των δύο τύπων SM για την πρωτότυπη proof-of-concept ετερογενή κάρτα γραφικών μας

των SMs μεταξύ kernels αφού τα περισσότερα δε χρησιμοποιούνται πάντα και πιο αποδοτικά. Ωστόσο η ποσόστωση των δύο τύπων τους θα είχε νόημα να μελετηθεί αφού πρώτα οριστεί η μικρο-αρχιτεκτονική διαμόρφωση του καθενός και για αυτό το αρχικό μοντέλο θα είναι δίκαια μοιρασμένο με 40 και 40.

Συνοψίζοντας, τα κύρια χαρακτηριστικά των δύο τύπων SM παρατίθενται στον πίνακα 6 μαζί με αυτά της V100 για σύγκριση. Τα συστήματα εκτός του SM, όπως η NOC, η κρυφή μνήμη L2 και η DRAM, παραμένουν πανομοιότυπα με τη διαμόρφωση της V100.

0.5 Αξιολόγηση

0.5.1 Εισαγωγή

Στην ενότητα 5.3 δημιουργήσαμε την πρώτη αναθεώρηση μιας ετερογενούς single-ISA GPU, την οποία θα δοκιμάσουμε σε αυτό το κεφάλαιο προσομοιώνοντας την ταυτόχρονη εκτέλεση kernels από τις δύο ομάδες που προσδιορίστηκαν στο προηγούμενο κεφάλαιο στους προσομοιωτές των ενοτήτων 4.2 και 4.3, προκειμένου να αξιολογήσουμε την απόδοσή της σε σχέση με πολλά baselines. Αυτή η ταυτόχρονη εκτέλεση θα διευκολυνθεί από την αναπτυχθείσα σουίτα benchmarks που περιγράφεται στην ενότητα 4.6 και περιέχει όλους τους kernels που μελετήθηκαν προηγουμένως.

0.5.2 Στήσιμο Πειράματος

Οι δοκιμαστικές μονάδες αφορούν την ταυτόχρονη εκτέλεση δύο εφαρμογών CUDA με συγκρίσιμο μεταξύ τους χρόνο εκτέλεσης GPU, καθεμία από τις οποίες περιέχει kernels μιας από τις δύο καθορισμένες κλάσεις kernels.

Προς αυτό επιλέγουμε να δοκιμάσουμε συνδυασμούς σε επίπεδο εφαρμογής και όχι σε επίπεδο kernel, καθώς αυτή η μέθοδος επιτρέπει μια πιο κοντινή προσέγγιση του σεναρίου στο οποίο στοχεύουμε, όπου ροές δυνητικά πολλαπλών kernels με αυστηρά καθορισμένη από την εφαρμογή σειρά καταλήγουν στην ίδια GPU και η εκτέλεσή τους γίνεται σειριακά ανά τύπο kernel. Με αυτόν τον τρόπο και προσεγγίζουμε το πραγματικό σενάριο και αποφεύγουμε ενα πολύπλευρα παραμετροποιήσιμο πρόβλημα δύσκολο να προσομοιωθεί. Αρκετές από τις εφαρμογές μας περιλαμβάνουν μόνο έναν kernel ωστόσο και αυτές παραμένουν πιστές στη βάση του σεναρίου μας που είναι η ταυτόχρονη εκτέλεση και παρέχουν πιο εύκολα ερμηνεύσιμα αποτελέμσατα για τις αλληλεπιδράσεις μεταξύ συγκεκριμένων kernels. Στις εφαρμογές που περιλαμβάνουν πάνω από έναν, όσοι καταλαμβάνουν σημαντικό τμήμα του χρόνου εκτέλεσής τους είναι του ίδιου τύπου διευκολύνοντας μας στο να έχουμε συνεχώς διαθέσιμους kernels και για τα δύο είδη SMs έως ότου μία εφαρμογή ολοκληρωθεί. Εξαίρεση αποτελεί το lulesh για το οποίο σπάμε τους τρεις kernels του που εξετάζουμε σε τρία διαφορετικά benchmarks. Είναι επίσης σημαντικό οι δύο εφαρμογές που θα εκτελούνται ταυτοχρόνως να έχουν παρόμοιο χρόνο εκτέλεσης GPU προκειμένου να έχουμε πλήρη χρήση της GPU όπως στο πραγματικό σενάριο για το μεγαλύτερο μέρος της εκτέλεσης. Από τις συνολικά 5 Compute-Intensive εφαρμογές και 10 Low-Utilization εφαρμογές που διαθέτουμε προκύπτουν 50 συνδυασμοί μοντελοποιημένοι στη σουίτα benchmarks που περιεγράφη στην υπο-ενότητα 0.3.6 και παρουσιάζονται μαζί με τα μεγέθη των προβλημάτων που τους αποτελούν στον πίνακα 7.

Low-Utilization App	Compute-Intensive App	Low-Utilization Size	Compute-Intensive Size
Polybench BICG (BIC1,BIC2)	BLonD Histogram (BLhi)	grid=(1024,1024)	nparticles=40000000
Polybench BICG (BIC1,BIC2)	Rodinia Hotspot (HOT)	grid=(1024,1024)	grid=(2048,2048)
Polybench BICG (BIC1,BIC2)	BLonD Interpolation-Kick (BLik_pre,BLik)	grid=(1024,1024)	nparticles=10000000
Polybench BICG (BIC1,BIC2)	BLonD Kick (BLki)	grid=(1024,1024)	nparticles=6000000
Polybench BICG (BIC1,BIC2)	lulesh (LUL3)	grid=(1024,1024)	46 edge nodes
BLonD Convolution (BLco)	BLonD Histogram (BLhi)	(nsignal,nkernel)=(1000,1000)	nparticles=1000000
BLonD Convolution (BLco)	Rodinia Hotspot (HOT)	(nsignal,nkernel)=(1000,1000)	grid=(512,512)
BLonD Convolution (BLco)	BLonD Interpolation-Kick (BLik_pre,BLik)	(nsignal,nkernel)=(1000,1000)	nparticles=1000000
BLonD Convolution (BLco)	BLonD Kick (BLki)	(nsignal,nkernel)=(1000,1000)	nparticles=1000000
BLonD Convolution (BLco)	lulesh (LUL3)	(nsignal,nkernel)=(1000,1000)	26 edge nodes
BLonD Drift (BLdr)	BLonD Histogram (BLhi)	nparticles=1000000	nparticles=1000000
BLonD Drift (BLdr)	Rodinia Hotspot (HOT)	nparticles=1000000	grid=(512,512)
BLonD Drift (BLdr)	BLonD Interpolation-Kick (BLik_pre,BLik)	nparticles=1000000	nparticles=1000000
BLonD Drift (BLdr)	BLonD Kick (BLki)	nparticles=1000000	nparticles=1000000
BLonD Drift (BLdr)	lulesh (LUL3)	nparticles=1000000	26 edge nodes
Polybench Gramschmidt (GRA1,GRA2,GRA3)	BLonD Histogram (BLhi)	grid=(512,512)	nparticles=40000000
Polybench Gramschmidt (GRA1,GRA2,GRA3)	Rodinia Hotspot (HOT)	grid=(512,512)	grid=(2048,2048)
Polybench Gramschmidt (GRA1,GRA2,GRA3)	BLonD Interpolation-Kick (BLik_pre, BLik)	grid=(512,512)	nparticles=10000000
Polybench Gramschmidt (GRA1,GRA2,GRA3)	BLonD Kick (BLki)	grid=(512,512)	nparticles=7000000
Polybench Gramschmidt (GRA1,GRA2,GRA3)	lulesh (LUL3)	grid=(512,512)	46 edge nodes
GASAL ksw (GApa,GAks)	BLonD Histogram (BLhi)	default query and target [3]	nparticles=5000000
GASAL ksw (GApa,GAks)	Rodinia Hotspot (HOT)	default query and target [3]	grid=(1024,1024)
GASAL ksw (GApa,GAks)	BLonD Interpolation-Kick (BLik_pre,BLik)	default query and target [3]	nparticles=2000000
GASAL ksw (GApa,GAks)	BLonD Kick (BLki)	default query and target [3]	nparticles=1000000
GASAL ksw (GApa,GAks)	lulesh (LUL3)	default query and target [3]	46 edge nodes
GASAL local (GApa,GAlo)	BLonD Histogram (BLhi)	default query and target [3]	nparticles=5000000
GASAL local (GApa,GAlo)	Rodinia Hotspot (HOT)	default query and target [3]	grid=(1024,1024)
GASAL local (GApa,GAlo)	BLonD Interpolation-Kick (BLik_pre,BLik)	default query and target [3]	nparticles=2000000
GASAL local (GApa,GAlo)	BLonD Kick (BLki)	default query and target [3]	nparticles=1000000
GASAL local (GApa,GAlo)	lulesh (LUL3)	default query and target [3]	46 edge nodes
GASAL global (GApa,GAgl)	BLonD Histogram (BLhi)	default query and target [3]	nparticles=5000000
GASAL global (GApa,GAgl)	Rodinia Hotspot (HOT)	default query and target [3]	grid=(1024,1024)
GASAL global (GApa,GAgl)	BLonD Interpolation-Kick (BLik_pre,BLik)	default query and target [3]	nparticles=2000000
GASAL global (GApa,GAgl)	BLonD Kick (BLki)	default query and target [3]	nparticles=1000000
GASAL global (GApa,GAgl)	lulesh (LUL3)	default query and target [3]	46 edge nodes
GASAL semi global (GApa,GAsg)	BLonD Histogram (BLhi)	default query and target [3]	nparticles=5000000
GASAL semi global (GApa,GAsg)	Rodinia Hotspot (HOT)	default query and target [3]	grid=(1024,1024)
GASAL semi global (GApa,GAsg)	BLonD Interpolation-Kick (BLik_pre,BLik)	default query and target [3]	nparticles=2000000
GASAL semi global (GApa,GAsg)	BLonD Kick (BLki)	default query and target [3]	nparticles=1000000
GASAL semi global (GApa,GAsg)	lulesh (LUL3)	default query and target [3]	46 edge nodes
lulesh (LUL1)	BLonD Histogram (BLhi)	23 edge nodes	nparticles=1000000
lulesh (LUL1)	Rodinia Hotspot (HOT)	23 edge nodes	grid=(512,512)
lulesh (LUL1)	BLonD Interpolation-Kick (BLik_pre,BLik)	23 edge nodes	nparticles=1000000
lulesh (LUL1)	BLonD Kick (BLki)	23 edge nodes	nparticles=1000000
lulesh (LUL1)	lulesh (LUL3)	23 edge nodes	23 edge nodes
lulesh (LUL2)	BLonD Histogram (BLhi)	23 edge nodes	nparticles=1000000
lulesh (LUL2)	Rodinia Hotspot (HOT)	23 edge nodes	grid=(512,512)
lulesh (LUL2)	BLonD Interpolation-Kick (BLik_pre,BLik)	23 edge nodes	nparticles=1000000
lulesh (LUL2)	BLonD Kick (BLki)	23 edge nodes	nparticles=1000000
lulesh (LUL2)	lulesh (LUL3)	23 edge nodes	23 edge nodes

Table 7: Σύνολο αξιολόγησης της ετερογενούς αρχιτεκτονικής single-ISA αποτελούμενο από 50 συνδυασμούς Low-Utilization και Compute-Intensive benchmarks μαζί με τις κύριες παραμέτρους που καθορίζουν το μέγεθός τους.

Αυτές οι εφαρμογές εκτελούνται στον προσομοιωτή ετερογενών single-ISA GPU για τη διαμόρφωση τύπου πυρήνα του πίνακα 6. Τα εκτελούμε επίσης σε αυτή την έκδοση του προσομοιωτή για μια διαμόρφωση της V100 χωρισμένη σε δύο τμήματα των 40 SM όπως και η ετερογενής GPU μας. Ο σκοπός αυτής της δοκιμής είναι να απομονώσει την επίδραση της κατάτμησης και μόνο της GPU. Το baseline μας είναι η αρχική έκδοση της V100 προσομοιωμένη στον αρχικό Accel-Sim [67] που διαθέτει 80 ομογενή SMs και θα εκτελέσει τις δύο εφαρμογές διαδοχικά. Με τον ίδιο τρόπο θα δοκιμάσουμε με τον αρχικό προσομοιωτή δύο άλλες διαμορφώσεις και οι δύο με 80 SM, αλλά για την πρώτη

θα έχουν όλες τη διαμόρφωση του τύπου πυρήνα High-Performance ενώ για την άλλη αυτή του τύπου Low-Power. Τα στοιχεία εκτός του SM παραμένουν αμετάβλητα σε σύγκριση με την V100 για όλες τις διαμορφώσεις. Για όλα τα μοντέλα ο χρόνος εκτέλεσης μετριέται σε κύκλους ρολογιού πυρήνων με το ρολόι να είναι ίδιο σε όλες τις διαμορφώσεις. Η ενέργεια υπολογίζεται ως το γινόμενο της μέσης ισχύος που καταναλώθηκε σε ένα διάστημα ως την περάτωση ενός kernel επί τους αντίστοιχους κύκλους

Τέλος, η επίδοση ταυτόχρονης εκτέλεσης kernel στην GPU αναμένεται να επηρεαστεί από παρεμβολές στα κοινόχρηστα υποσυστήματα όπως το NOC, η L2 και η DRAM. Τα αιτήματα μνήμης από δύο διαφορετικούς πυρήνες θα δρομολογούνται μέσω του δικτύου διασύνδεσης στα διάφορα διαμερίσματα μνήμης προκαλώντας αυξημένες καθυστερήσεις απόκρισης λόγω συγκρούσεων μεταξύ αιτημάτων των kernels σε όλες αυτές τις δομές και τη σειριοποίηση των απαντήσεων λόγω περιορισμένου εύρους ζώνης [58, 25, 26]. Προκειμένου να εκτιμήσουμε την απόδοση που χάνεται λόγω αυτού του φαινομένου, θα συμπεριλάβουμε μια ακόμη ανάλυση στην αξιολόγηση της απόδοσής μας. Αυτή θα αφορά την εκτέλεση και των δύο εφαρμογών στον αρχικό προσομοιωτή, μία κάθε φορά, σε μια διαμόρφωση που διαθέτει 40 SMs του τύπου στον οποίο προοριζόταν η κάθε μία να τρέξει. Με αυτόν τον τρόπο κάθε ένας από τους kernels έχει στη διάθεσή του ακριβώς τους ίδιους πόρους που είχε στην ετερογενή πλατφόρμα, ωστόσο με ολόκληρο το σύστημα μνήμης εκτός SM στη διάθεσή του. Στη συνέχεια, επικαλύπτοντας τα αποτελέσματα του χρονισμού των εκτελέσεων των δύο kernels μπορούμε να κάνουμε μια εκτίμηση για το πώς θα απέδιδαν στην ετερογενή πλατφόρμα αν δεν υπήρχε καμία παρεμβολή με τον συνολικό χρόνο εκτέλεσης σε αυτή την περίπτωση να αντιπροσωπεύεται από τον μέγιστο χρόνο εκτέλεσης των δύο εφαρμογών. Παραλείπουμε την εκτίμηση ενέργειας για αυτό το μοντέλο καθώς δεν μπορεί να μετρηθεί αξιόπιστα για το κομμάτι της κοινής εκτέλεσης.

0.5.3 Επισκόπηση Αποτελεσμάτων

Όλα τα αποτελέσματα, τόσο για την ισχύ όσο και για το χρόνο εκτέλεσης αντίστοιχα, είναι κανονικοποιημένα σε σχέση με εκείνα της διαδοχικής εκτέλεσης στην αρχική V100 που αντιπροσωπεύεται από την πρώτη μπάρα. Η δεύτερη μπάρα αναφέρεται στη σειριακή εκτέλεση στην GPU με 80 High-Performance SMs και η τρίτη σε εκείνη με 80 Low-Power SMs. Η τέταρτη (no_interf) αντιστοιχεί στην ιδεατή χωρίς παρεμβολές διαμόρφωση, ενώ η πέμπτη αναπαριστά την διχοτομημένη V100. Τέλος, η τελευταία μπάρα παρουσιάζει τα αποτελέσματα που επιτεύχθηκαν από την ετερογενή GPU. Σε όλες τις διαμορφώσεις, το τμήμα των αποτελεσμάτων που αποδίδεται σε εφαρμογές που διαθέτουν Low-Utilization kernels είναι χρωματισμένο με μπλε χρώμα, ενώ το τμήμα που αν-

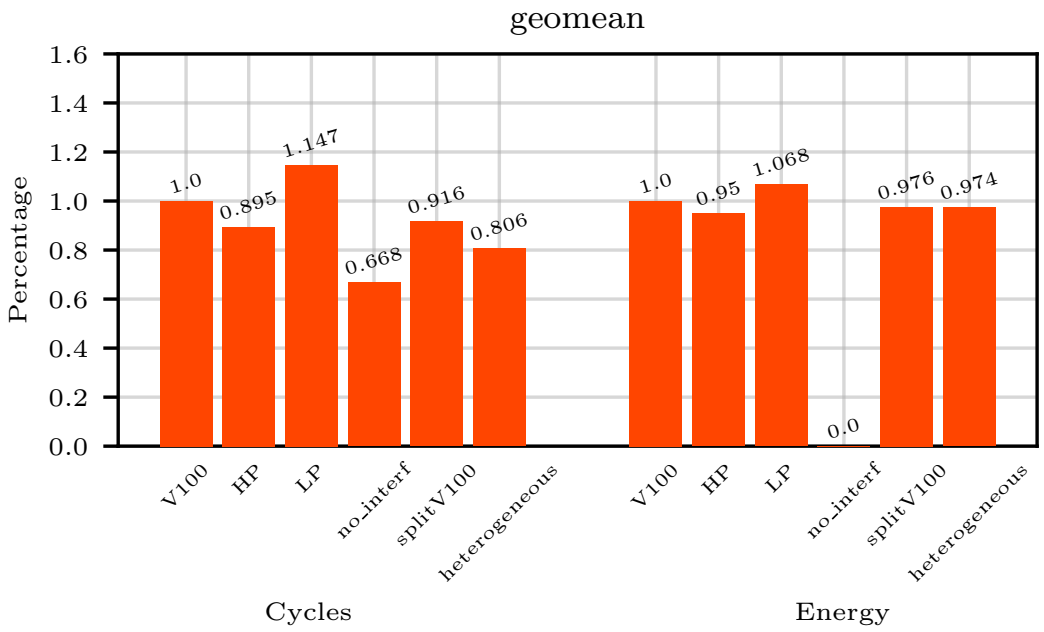


Figure 22: Αξιολόγηση της ταυτόχρονης εκτέλεσης Compute-Intensice και Low-Utilization kernels σε μοντελοποιημένη ετερογενή single-ISA GPU για 50 διαφορετικούς συνδυασμούς εφαρμογών από τις δύο κατηγορίες

τιστοιχεί στην εκτέλεση Compute-Intensive kernels είναι χρωματισμένο πορτοκαλί. Το μωβ τμήμα υποδηλώνει την ταυτόχρονη εκτέλεση kernels και από τις δύο εφαρμογές.

Μια σύνοψη των αποτελεσμάτων και για τα 50 benchmarks εμφανίζεται στο Σχήμα 22. Η ανάλυση επικεντρώνεται στον γεωμετρικό μέσο όρο των αποτελεσμάτων, καθώς ο αριθμητικός μέσος όρος θεωρείται αναξιόπιστος όταν συνοψίζονται τα αποτελέσματα των κανονικοποιημένων τιμών απόδοσης σε όλες τις μηχανές [37]. Όπως βλέπουμε κατά μέσο όρο το μοντέλο μας μειώνει το συνολικό χρόνο εκτέλεσης κατά 19,4% σε σύγκριση με την παραδοσιακή διαδοχική εκτέλεση των δύο εφαρμογών στο βασικό μοντέλο V100. Η απλή κατάτμηση της κάρτας χωρίς να χρησιμοποιείται καμία ετερογένεια έχει επίσης καλύτερη απόδοση από το βασικό μοντέλο, αλλά το ετερογενές μοντέλο εξακολουθεί να είναι ταχύτερο από αυτό, απαιτώντας 12% λιγότερους κύκλους για την ολοκλήρωση της εκτέλεσης. Το τίμημα που επιβάλλει η παρεμβολή στο ετερογενές μοντέλο γίνεται εμφανές από την επιτάχυνση κατά 49,7% κατά μέσο όρο που επιτυγχάνει η προσομοιωμένη προσέγγιση χωρίς παρεμβολή σε σχέση με την V100 σε σύγκριση με την 24,1% που επιτυγχάνει το ρεαλιστικό ετερογενές μοντέλο μας. Προφανώς το μοντέλο με τους 80 High-Performance

πυρήνες αποδίδει καλύτερα από την αρχική V100, αν και λιγότερο καλά από το ετερογενές μοντέλο, λόγω της έλλειψης του κέρδους που αποδίδεται στην ταυτόχρονη εκτέλεση και της ανικανότητας των Low-Utilization kernels να εκμεταλλευτούν δυνατότερους πυρήνες, ενώ αντίστροφα το μοντέλο με τους 80 μικρότερους Low-Power πυρήνες αποδίδει χειρότερα.

Δυστυχώς όσον αφορά την κατανάλωση ενέργειας, οι πολύ μικρές διακυμάνσεις που παρατηρούνται μεταξύ των διαφορετικών μοντέλων που δοκιμάστηκαν, καθιστούν αρκετά εμφανές το γεγονός ότι περιοριζόμαστε από το μοντέλο ισχύος του AccelWattch [63], όπως σημειώθηκε στην υπο-ενότητα 0.4.2. Δεδομένου ότι με αυτό το μοντέλο ισχύος οι δομές που αρχικά δεν χρησιμοποιήθηκαν και περικόπηκαν για τους Low-Utilization kernels δεν θα επηρεάσουν τα αποτελέσματα όπως θα αναμενόταν, λόγω του ότι οι μετρητές απόδοσης παραμένουν περίπου ίδιοι μεταξύ των διαμορφώσεων, δεν μπορούμε να απεικονίσουμε το κέρδος που μας προσφέρει η εκτέλεση σε αυτούς τους πυρήνες από άποψη ισχύος. Ωστόσο, το μοντέλο ισχύος παρέχει μια ένδειξη της ενεργειακής απόδοσης, δεδομένου ότι η επιπλέον αξιοποίηση του - και εξαιτίας του - πρόσθετου υλικού στους ισχυρότερους πυρήνες, η οποία λαμβάνεται υπόψη μέσω των μετρητών απόδοσης, δεν προκαλεί ραγδαία αύξηση στην κατανάλωση ενέργειας του ετερογενούς μοντέλου, αντιθέτως, η σχεδίασή μας έχει οριακά καλύτερες επιδόσεις όσον αφορά τη συνολική κατανάλωση ενέργειας σε σύγκριση με την αρχική V100 κατά ένα περιθώριο 2,6%. Αυτό μπορεί να αποδοθεί σε μεγάλο βαθμό στους ταχύτερους χρόνους εκτέλεσης που μειώνουν το χρόνο που μένει ενεργή η GPU.

0.6 Συμπεράσματα και μελλοντικές κατευθύνσεις

0.6.1 Σύνοψη

Προτείνουμε την έλευση της single-ISA ετερογένειας σε επίπεδο πυρήνα στον κόσμο των GPU, ο ερχομός της οποίας αποτελεί μια λογική μελλοντική κατεύθυνση λαμβάνοντας υπόψη τη γενική πορεία προς την εξειδίκευση και την ετερογένεια που ακολουθείται στην πληροφορική. Επεκτείνουμε τους σύγχρονους και ευρέως αποδεκτούς προσομοιωτές προσομοίωσης κύκλων Accel-Sim [67] και AccelWattch [63] ώστε να υποστηρίζουν ετερογενείς διαμορφώσεις που διαθέτουν SMs διαφορετικής σύνθεσης και να εκτελούν ταυτόχρονα kernels σε αυτούς. Επεκτείνουμε το CUDA API με μια νέα κλήση που δίνει στους τελικούς χρήστες τη δυνατότητα να δηλώνουν τον τύπο του SM στον οποίο θέλουν να εκτελείται ο κάθε kernel τους μέσα στον πηγαίο κώδικα των benchmarks. Για να διευκολύνουμε τη μελέτη της ταυτόχρονης εκτέλεσης kernels στις εν λόγω αρχιτεκτονικές αναπτύσσουμε μια επεκτάσιμη σουίτα benchmarks προ-εξοπλισμένη με πολλαπλά benchmarks στα οποία μπορούν να προστεθούν περισσότερα μέσω μιας τυποποιημένης διαδικασίας.

Για να αναδείξουμε τη σημασία της πρότασής μας, διεξάγουμε μια μελέτη περίπτωσης που αφορά τη βελτίωση της απόδοσης επιστημονικών εφαρμογών που εκτελούνται σε συστάδες HPC με GPU. Οι πολλαπλές παρτίδες εκτέλεσης εφαρμογών μαζί με τις τεχνικές διαμοιρασμού και εικονικοποίησης των GPU που χρησιμοποιούνται σε τέτοιες συστάδες παρέχουν το έδαφος για παραλληλισμό σε επίπεδο kernel. Επιλέγουμε ένα αντιπροσωπευτικό σύνολο kernels που εμπλέκονται σε επιστημονικούς υπολογισμούς και μελετάμε τα χαρακτηριστικά τους κατά το χρόνο μεταγλώττισης καθώς και τα χαρακτηριστικά της εκτέλεσής τους στο περιβάλλον PTX του Accel-Sim [67], δείχνοντας ότι αποκαλύπτουν διαφορετικά μοτίβα χρήσης υλικού και σημεία συμφόρησης, με βάση τα οποία τοποθετούνται σε δύο διαφορετικές κατηγορίες. Στη συνέχεια επιχειρούμε να προσαρμόσουμε τους πυρήνες της GPU ξεκινώντας από τη Volta V100 για να δημιουργήσουμε δύο νέες διαμορφώσεις που η καθεμία είναι καταλληλότερη για την εκτέλεση μιας από τις δύο κατηγορίες kernels, δοκιμάζοντας τον τρόπο με τον οποίο οι kernels που μελετήθηκαν ανταποκρίνονται σε συγκεκριμένες αλλαγές της μικρο-αρχιτεκτονικής.

Στη συνέχεια, προχωρούμε στην αξιολόγηση αυτής της νέας αρχιτεκτονικής με την ταυτόχρονη εκτέλεση benchmarks που περιέχουν kernels από τις δύο ομάδες που προσδιορίστηκαν προηγουμένως, παρουσιάζοντας επιτάχυνση 24,1% σε σχέση με τη διαδοχική εκτέλεση στη βασική V100, χωρίς συμβιβασμούς στην ενέργεια αλλά με κέρδος 2,6%.

0.6.2 Μελλοντικές εργασίες

- Περαιτέρω επέκταση και δοκιμή του Accel-Sim [67] για την υποστήριξη προσομοίωσης με βάση τα ίχνη (traces) στην ετερογενή πλατφόρμα, παρέχοντας ακριβέστερα αποτελέσματα προσομοίωσης και έκθεση σε περισσότερα εγγενή χαρακτηριστικά εκτέλεσης, τα οποία δεν προσομοιώνονται όταν χρησιμοποιείται το virtual PTX ISA.

- Ενεργοποίηση της υποστήριξης των προς το παρόν μη υποστηριζόμενων διαφοροποιήσεων μεταξύ των τύπων πυρήνων, όπως η στρατηγική επανασύγκλισης και το πλάτος γραμμής texture cache.

- Ανάπτυξη ενός μοντέλου ισχύος ικανού να προσαρμόζεται πλήρως στην επαναδιαμόρφωση του υλικού και να μην βασίζεται αποκλειστικά σε μαθηματικά προσαρμοσμένους συντελεστές που προκύπτουν από μετρήσεις πραγματικού, υλοποιημένου υλικού.

- Δυνατότητα υποστήριξης δραστικών αρχιτεκτονικών αλλαγών μεταξύ των δύο τύπων πυρήνων. Για παράδειγμα, παρατηρήσαμε kernels που παρουσίαζαν περιορισμένη διαθεσιμότητα ενεργών warps, των οποίων τις επιδόσεις δεν μπορέσαμε να βελτιώσουμε μέσω των διαθέσιμων μικρο-αρχιτεκτονικών αλλαγών και θεωρήθηκαν Low-Utilization. Ωστόσο, η βιβλιογραφία υποδεικνύει ότι υπάρχουν μικρο-αρχιτεκτονικές ικανές να επιταχύνουν τέτοιους kernels όπως η LOOG [51] η οποία εισάγει ελαφριά εκτέλεση εκτός σειράς στην GPU επιτρέποντας στην παραλληλία επιπέδου εντολών να κρύψει τις καθυστερήσεις που δεν μπορεί να κρύψει η εκτέλεση διαφορετικών warps. Αν και ένα τέτοιο σχήμα θα μπορούσε να ωφελήσει οποιονδήποτε kernel έχει μια συγκεκριμένη ομάδα-στόχο, όπως είθισται για μια τεράστια ποικιλία εργασιών στις GPUs λαμβάνοντας υπόψη την τάση για εξειδίκευση. Η ετερογένεια σε επίπεδο SM επιτρέπει την εφαρμογή τέτοιων αρχιτεκτονικών μοντέλων αποκλειστικά στους επωφελούμενους kernels, επιτρέποντας στις υπόλοιπες εφαρμογές να εκτελούνται σε SM προσαρμοσμένα στις ιδιαίτερες ανάγκες τους. Με την εφαρμογή εξειδικευμένων αρχιτεκτονικών που έχουν μελετηθεί στη βιβλιογραφία και επιτρέποντας τον συνδυασμό τους ανοίγουμε έναν πολύ ευρύτερο χώρο σχεδιασμού που επιτρέπει ακόμη καλύτερα αποτελέσματα.

- Βαθύτερη μελέτη χαρακτηριστικών εκτός του SM όπως ο αριθμός των SM ή οι συχνότητες ρολογιού στις οποίες λειτουργούν.

- Εφαρμογή μηχανισμών για την αύξηση της απόδοσης με την άμβλυνση των παρεμβολών στο κοινό σύστημα μνήμης. Η βιβλιογραφία έχει ήδη διερευνήσει την επίδραση που έχει η ταυτόχρονη εκτέλεση kernels στην

GPU όσον αφορά την αναποτελεσματικότητα της μνήμης λόγω των αιτημάτων από περισσότερους του ενός kernels και έχουν προταθεί πιθανές λύσεις όπως εναλλακτικοί χρονοπρογραμματιστές DRAM [58], εξισορρόπηση των αιτήσεων μνήμης από τους πυρήνες ή περιορισμός τους [26] και ανάθεση διευθύνσεων με διαφορετικό τρόπο για την απομόνωση των διαφορετικών πυρήνων εντός του διαμερίσματος μνήμης όπως γίνεται στο MIG [94].

- Εφαρμογή μηχανισμών ασφαλείας για τη διασφάλιση της απομόνωσης των kernels στο μοιραζόμενο σύστημα μνήμης, όταν κρίνεται απαραίτητο, και την παροχή ανοχής σφαλμάτων μεταξύ εφαρμογών που εκδίδουν ταυτόχρονα εργασίες στην GPU.

Chapter 1

Introduction

1.1 Introduction to GPUs as General Purpose GPUs

Graphics Processing Units (GPUs) are massively thread parallel devices originally aimed towards graphics rendering acceleration as their name suggests. This was their main purpose for many years which they served, and still do, by employing their massively parallel execution potential to render 2D images to be displayed to the screen. They did and still do so in a series of steps starting from vertices to assemble the geometry of the image before mapping the textures to it through fixed or programmable pipelines [81].

However their range of applicability has drastically increased creating an entire new field of General Purpose GPUs (GPGPUs) allowing their throughput benefits achieved by the aforementioned parallelism to be exploited by a wide variety of non-graphics applications like Neural Network training and inference, Alignment algorithms, Physics calculations, Data Analytics, Cryptography and many more. Those applications leverage the specific advantages of GPUs over CPUs by assigning time consuming parts of their execution to kernels and offloading them to the GPU which is used as an accelerator. Those kernels will then move on to spawn numerous threads grouped into blocks which will be scheduled to the GPUs computational powerhouses, the Streaming Multiprocessors (SMs - NVIDIA terminology is used for short but the concept is vendor independent) to be further divided in groups of threads and executed in lockstep by the computational pipelines present within the SM. Thread groups will vacate the execution pipeline when encountering a stall, like a long memory operation that has to be serviced by the DRAM, allowing the execution of others to mask that delay resulting in a constant high utilization of the hardware.

1.2 GPU hardware demands diversity

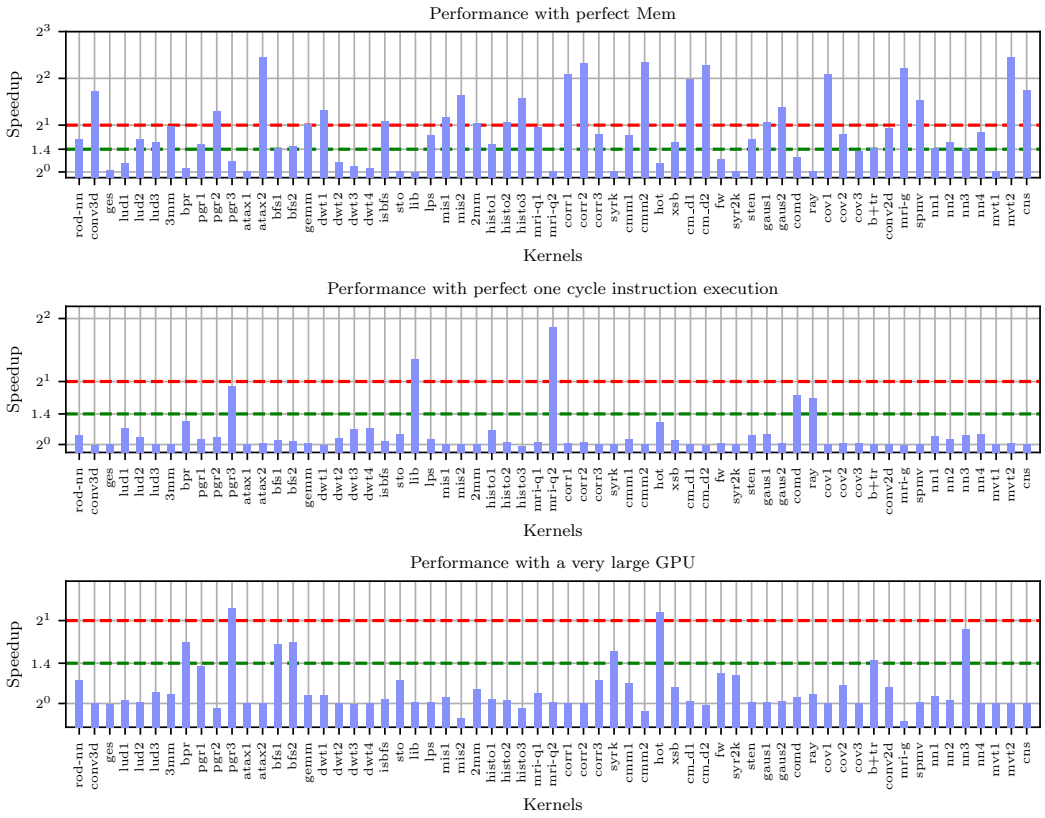


Figure 1.1: Different general purpose GPU kernels performance improvement with different configurations. Double speedup (red line) is considered major while the lower improvement threshold with those aggressive configurations is set at 40% (green line) when judging the bottlenecks of each kernel.

As GPU usage becomes more and more adopted for a wide variety of applications, the ever-growing amount of workloads offloaded to it, coupled with the diversity in execution characteristics of the kernels does not allow for a single hardware configuration of a GPU to efficiently execute any kernel presented to it. Often a kernel will fail to achieve full utilization of the card because of stalls that cannot be hidden by issuing instructions from a different group of threads. Such stalls can occur in kernels where the most significant percentage of their threads issue long memory instructions thus creating a high contention for the high latency resource along with its buses. All the while the computational units are starved of threads to execute awaiting

Perfect Memory	Perfect Execution	Large GPU
all L1 misses serviced in one cycle by the memory subsystem, only limited by the amount of traffic on the memory bus	all instruction initiation & execution latencies set to one cycle	5×execution units, register file banks, schedulers, 2×shared memory banks, 10×operand collector units, 1.5×SMs

Table 1.1: Configurations used to acknowledge the dissimilar needs of GPU kernels

results from the slow main memory. Respectively, for heavier workloads, kernels may be limited by the size of the execution pipeline causing threads candidate for dispatching to face structural hazards limiting the throughput of the GPU in this manner.

The pipeline could remain idle for a plethora of other reasons as well. A significant example is kernels that force low GPU occupancy either because they explicitly issue a low number of threads, or because they can not fill the SMs to their thread limit due to hardware restrictions, like register or scratchpad memory usage, resulting in completely empty SMs or SMs whose schedulers do not have a large enough pool of thread groups to select from. All the above forms of under-utilization result in a single bottleneck point which ultimately determines the performance of the kernel. An analysis of 63 GPGPU kernels included in Accel-Sim's [67] package, by executing them on the three configurations explained in Table 1.1 confirms those assumptions. As shown in Figure 1.1 23 out of the 63 kernels double their performance with a perfect memory system and an additional 21 show at least 40% speedup compared to their baseline execution on the Volta [97] architecture NVIDIA V100. This is indicative of a memory bottleneck for those applications limiting the utilization of the arithmetic pipeline, a statement confirmed by the observations of Adwait Jog et al. [60]. Five other kernels show improvement with perfect execution modules. It is important to note that those kernels do not overlap with the kernels that improved via the use of perfect memory subsystem so those kernels seem to stall mainly awaiting for data hazards due to execution results not being ready on time and not memory responses being slow. Running the same kernels on a V100 with 40 more SMs and an unrealistically largely scaled execution pipeline within the SMs benefits eight kernels, indicating that they present more, unexploited thread level parallelism (TLP), of which three are not improved with either perfect memory or perfect instructions. No kernels are improved by all three different

models. Furthermore five kernels show a slowdown of more than 2.5% and two of them more than 10% when ran on the larger V100 showcasing that sometimes even more resources can hurt performance by adding pressure on existing bottlenecks.

The heterogeneity between kernels, in the forms discussed above and more, is a phenomenon widely observed and documented in literature [131, 60, 51, 40, 2, 20].

1.3 Steps towards specialization

These observations have led to a lot of advancements regarding specialization techniques many of which are already implemented on hardware.

GPUs have introduced specialized execution units regarding matrix operations to improve the performance of kernels limited by the device's computational throughput when executing such calculations [96]. Nevertheless a plethora of kernels consisting mostly of non-matrix related operations will usually fail to utilize said units, experiencing no gains by the introduction of the added silicon. Literature has also shifted its approach towards specialization, even inflicting drastic changes on the execution model in order to improve the performance and efficiency of specific kernel sub-classes [51, 13]. Still although such models are great fits for specific kernels they offer little to no edge over existing GPUs when executing kernels they were not specifically designed to target. More details regarding specialization techniques are provided in Section 3.3.

1.4 Concurrent Execution

Current GPU trends support executing more than one kernels concurrently on the same card in order to improve its utilization by allowing another kernel to take advantage of resources not being used by the currently executing kernel [130, 57, 16, 111, 2]. Concurrent kernel execution is officially supported by NVIDIA, starting with the Fermi [99] architecture through the use of various implemented technologies [104, 90, 94]. Concurrent execution schemes will be discussed in more detail in Section 3.6. However sharing in its current form requires the first of a sequence of kernels to not be able to fill the entirety of the SMs in order for the next one to issue in the case of streams or MPS. The capacity of the SM in this context is not measured only in threads/core but any occupancy limiting factor like the register file or scratchpad memory, meaning that a kernel fully occupying the register file

cannot be co-executed with another even if the second stresses other structures. Static splitting of the card like MIG [94] or Spatial Multitasking [2] resolves this issue by having more than one group of cores to which different kernels can issue however within each group the problems of low utilization remain.

1.5 Proposal Overview

Despite the fact that specialization techniques are clearly beneficial to GPGPU kernels, they fail to present a universal solution since discrete kernels, often originating from a single application and running on the same GPU, utilize the available hardware in a diversity of ways unable to be targeted by a single specialization technique. At the same time concurrent execution over existing hardware features severe limitations when used to increase utilization and is not always effective. This is why we shift our focus towards core-level heterogeneity in order to achieve the best of both worlds. The hardware bottlenecks of each kernel as well as the parts left unused can be identified making it possible for SMs to become specialized towards their needs by removing resources that are not utilized while adding the lacking ones. Concurrent kernel execution on such GPUs can lead to optimal utilization of each category of SMs by assigning it the appropriate kernels.

Exploiting this phenomenon by introducing heterogeneity on a GPU-card level using more than one physical card to which the application(s) can issue is not sensible since the energy and time cost to be suffered from data movements between kernels with data dependencies executing on different platforms would be immense. An example of such a dependency are kernels of the same application requiring different hardware but operating sequentially, depending on the results of previous kernels. Even with the NVLink [101] interconnection offered by NVIDIA between the GPUs the bandwidth available between them is 3x and 2.6x [102] lower than that offered by each card's DRAM for Volta [97] and Ampere [107] and their supported NVLink versions respectively. This indicates that the new bottlenecks introduced by alternating execution platforms would be severe since this system is slower than the already limiting DRAM. Using more than one physical card would also require increased space at the deployment rack, and redundant hardware use in the form of Printed Circuit Boards (PCBs) and other components who could otherwise be efficiently shared.

This adds value to the idea of a GPU with more than one type of SM, introducing heterogeneity on a SM level, curing a larger range of GPU applica-

tion needs on the same die. This approach is similar to that of bigLITTLE [8] CPU architectures who categorize applications based on their computational intensity and run them either on smaller energy efficient cores if they cant utilize the faster big cores or vice versa, while the memory system of the card is shared between the heterogeneous cores. A more analytical view of bigLITTLE is presented in section 3.2. Such an approach can be optimally exploited by sharing of the GPU among kernels, of one or more applications, who feature different characteristics. Unlike traditional GPU sharing where the execution of more than one kernel depends on the first kernel not fully occupying the processor, a statically split heterogeneous approach would always provide room to accommodate at least two kernels with different characteristics meant to execute on different hardware. As an extra advantage the shared memory system allows for seamless data communication between kernels of any type regardless of the hardware they execute on. For lack of a state-of-the-art GPU simulator that supports two different types of SMs with different internal micro-architecture to coexist on the same GPU and since heterogeneity is starting to become the go-to solution on more and more general purpose computational devices across domains [10, 114], we modify the latest version of Accel-Sim's [67] underlying General Purpose GPU simulator, GPGPU-Sim [12] v4.2.0 enabling it to model an arbitrary amount of each class of SMs with numerous available differences between their internal configurations while also adapting AccelWattch's [63] power model, so that its results are in agreement with the new timing model. Additionally we expand the CUDA API allowing the programmer to denote the preferred hardware for each of their kernel launches. At the same time we propose a novel application model for CUDA applications written to be benchmarked on this version of Accel-Sim [67] which allows more than one kernel to be issued and ran simultaneously on different types of SMs. As a case study of our proposal of SM level single-Instruction-Set-Architecture (single-ISA) heterogeneous GPUs, we study and evaluate the effects of such a hardware configuration on a range of HPC applications showing that there is potential for faster execution and sustained energy efficiency.

1.6 Contributions

This thesis contributes the following:

- Extension of GPGPU-Sim to support simulation of up to two distinct, heterogeneous types of SMs on the same chip with potentially different:
  - front-end structures like schedulers and instruction buffers

  - back-end structures like operand collectors and execution units
  - on-chip memory modules configurations like caches or shared memory
  - clock domains
- Modification and expansion of AccelWattch [63] to produce power and energy estimates for such configurations
- Enhancements of reported statistics to facilitate heterogeneous architecture analysis
- CUDA Runtime API expansion, handing control of kernel-hardware mapping to the programmer
- An HPC application suite designed to be ran with this new version of the simulator, expandable to include more kernels
- HPC workload characterization based on sensitivity to the underlying hardware
- Proposal of a novel proof of concept single-ISA Heterogeneous GP-GPU based on the characterization results

1.7 Thesis Structure

This thesis consists of five chapters:

- Chapter 2 serves as a deep look in the background supporting this work such as parallel computing, the GPU architecture, the CUDA programming model along with the simulator tools used.
- Chapter 3 is an overview of prior work regarding single-ISA heterogeneous architectures of CPUs and GPU specialization techniques
- Chapter 4 discusses in depth the implementation details of our extended simulator and API as well as the benchmark suite software model
- Chapter 5 proposes the HPC Cluster instance as a target for heterogeneous single-ISA GPUs and goes in detail exploring HPC kernels needs and characteristics
- Chapter 6 suggests a heterogeneous configuration based on the findings of Chapter 5 and evaluates the performance results against multiple baselines

- Chapter 7 offers a summary of our findings and provides pointers for future work

Chapter 2

Background

2.1 Introduction

In this chapter we will briefly cover the fundamental concepts which serve as the pillar of this work. We will shortly go through parallelism models as well as the basics of the General Purpose Graphics Processing Unit (GPGPU) architecture and its programming model. Additionally we will look into the latest publicly available research GPGPU performance and power simulators included within the Accel-Sim [67] and AccelWattch [63] frameworks respectively. Finally we will discuss heterogeneity concepts in computing.

2.2 Parallel Computing

Parallel Computing refers to the fragmentation of a large workload in smaller ones which are then executed simultaneously by more than one computational units while sharing computational and control data via at least one communication system whether this involves a shared memory module or a messaging system (with the exception of embarrassingly parallel applications who do not require any communication between computational units [129]). The term parallelism can be divided based on the level it occurs into Bit-Level, Instruction-Level, Superword-Level, Task-Level or Data-Level parallelism with the last two being the most prevalent forms of parallelism exploited by GPGPUs [49]. In the following subsection we will briefly break down the aforementioned types of parallelism.

2.2.1 Types of Parallelism

Bit-Level Parallelism refers to an increase of the processors word size in order to reduce the number of instructions necessary to perform operations on variables of size greater than the processor's word size. Via this process 4-bit processors were replaced by 8-bit counterparts then 16-bit ones until the trend for increased Bit-Level parallelism came to a halt at 32-bits to be replaced decades later by 64-bits. It is self-evidently exploited by all modern processors. [132]

Instruction-Level Parallelism (ILP) is the exploitation of lack of dependencies between instructions of a single sequential program enabling them to be executed concurrently. In order to take advantage of such parallelism processors embed multiple execution units coupled with the ability to fetch, decode and issue multiple instructions in a single step (superscalar cores [119]). High instruction throughput is facilitated by Out-of-Order execution mechanisms, which allow for the execution of instructions appearing later in the instruction stream provided they do not have unmet dependencies and speculative execution, and speculative execution mechanisms preventing the halt of instruction execution when branches are met by predicting their outcome. By exploiting ILP execution time is reduced while hardware utilization is increased since more execution units are occupied and memory latencies are effectively hidden by issuing independent instructions for execution [29].

Superword-Level Parallelism (SLP) is achieved by packing source operands and results in memory and executing isomorphic statements (statements that contain the same operations, in the same order) on even a single computational core simultaneously. This method of parallelism is particularly helpful in multimedia applications [72].

Task-Level Parallelism involves the breakdown of a large job into smaller independent tasks and their assignment to computational cores. This is a less restricting form of parallelism than the ones mentioned above, since each task can comprise arbitrary instructions which can operate on different data, and can be exploited by the programmer directly instead of relying on compilation techniques and underlying hardware. A popular form of Task-Level Parallelism is Thread-Level Parallelism where tasks are assigned to computational threads who execute independently, synchronizing when needed [22].

When discussing the processing of multiple data simultaneously we are referring to **Data-Level Parallelism**. This is an especially broad definition and not as insightful. The term is most commonly associated with the simultaneous execution of the same code on multiple data across multiple

processors (as opposed to Superword-Level Parallelism) [78].

Graphics Processing Units mainly exploit the two latter forms of parallelism. Groups of GPU threads execute code in a lockstep SIMD manner on data indexed by thread-id thus exploiting underlying Data-Level Parallelism. Also GPU threads can implicitly be assigned tasks by forcing them to diverge, executing different parts of code operating on distinct data thus exploiting task level parallelism. The underlying GPU hardware facilitating those mechanisms is described in subsection 2.3.3.

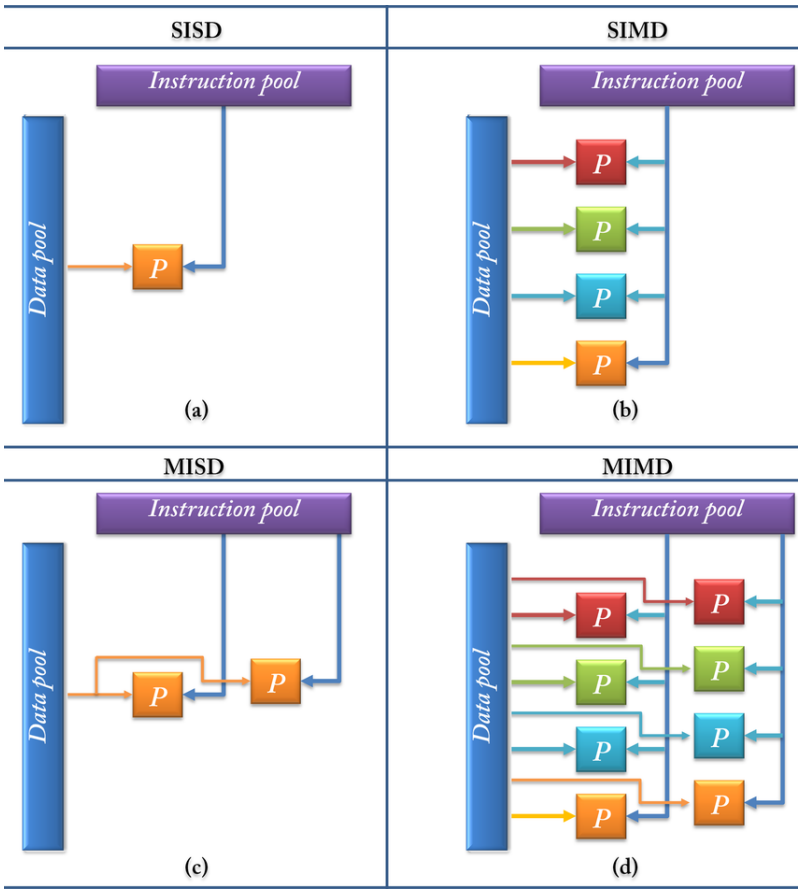


Figure 2.1: Flynn's Taxonomy of computer architectures [83]

2.2.2 Computer Architecture Classification

Now that the potential parallelism that can be tapped into as well as a handful techniques of doing so have been discussed, the architectures necessary to exploit it must be analyzed. One of the most popular classification systems

of computer architectures relevant to Parallel Computing is **Flynn's taxonomy** [39, 38]. Flynn's categorization is based on the number of instruction streams and data streams present in the architecture, with the four types of computer architectures existing according to it being presented below:

- Single Instruction Stream-Single Data Stream (SISD): refers to the uniprocessor - a sequential computer fetching a single instruction from memory to direct a single processing element into operating on a specific data stream. This computer exploits no parallelism.

- Single Instruction Stream-Multiple Data Stream (SIMD): includes processors consisting of n execution units who read from a single instruction stream but multiple data streams. These execution units subsequently execute the single instruction read in lockstep. Such an architecture can take advantage of Data-Level Parallelism. In his 1972 revision of the taxonomy [38] Flynn included array processors as a subcategory of SIMD processors. Those are the predecessor and closest relative to Single Instruction Multiple Threads (SIMT) processors. SIMT contrary to SIMD executes all instructions in all threads in lockstep and allows for control divergence enabling threads to be executed independently. It is the main scheme which GPUs are built around.

- Multiple Instruction Stream-Multiple Data Stream (MISD): architecture is rarely used for very specific cases, for instance flight control systems where calculation redundancy from multiple processing elements on the same data aids in fault prevention. This architecture, like SISD, does not utilize parallelism.

- Multiple Instruction Stream-Multiple Data Stream (MIMD): is the most flexible of all types. It includes architectures consisting of multiple autonomous processors reading from separate instruction and data streams. Those processors can communicate via shared or distributed memory as is the case in many super computers. As the most flexible architectural scheme it supports both Data-Level Parallelism and Task-Level Parallelism.

Although not mentioned above, obviously all models can support at least Bit-Level parallelism since data is not constrained to be single bit. Clearly Flynn's Taxonomy which was suggested in papers written in 1966 [39] and 1972 [38] has been outdated. For example although on the one hand modern superscalar Out-of-Order cores indeed do read from a single instruction stream and operate on a single data stream, on the other hand they feature

more than one "processing elements" in order to take advantage of existing ILP within the program's instructions, who they can fetch out of order. The presence of multiple "processing elements" along with the larger than one instruction decode per cycle surely exclude them from the SISD class and the lack of more than one instruction or data streams excludes them from the rest of the classes as well therefore marking them as an outlier in this taxonomy. Nevertheless nowadays architectures featuring single data and instruction streams can take advantage of ILP. Also each processing unit in MIMD architectures can itself be superscalar core thus including another level of parallelism in each stream. It becomes apparent that a single instruction stream does not rule out instruction parallelism in modern computer architectures. Those architectures among others are the reason why although Flynn's taxonomy is still very much relevant many attempts at new revisions been made [33, 61].

2.3 General Purpose Graphics Processing Unit (GPGPU)

2.3.1 Introduction

The potential GPUs exhibited at applications other than 3D processing had not been tapped in until the middle 2000s. Up until then, they were viewed as merely function-specific accelerators, vastly improving 3D graphics computation times by parallelly processing independent vertices and fragments ignoring other computing aspects. The first academic attempts at offloading general purpose computations were strenuous and involved mapping the general purpose computations to graphics APIs, without direct access to the computational units. Having such an application executing correctly on a GPU was considered an accomplishment in those early stages and for many years to follow could not be done by the average non-specialized programmer. Moving forward as their massive arithmetic and memory bandwidth combined with the ability of application to exploit them through massive parallelism was looked into more and more, new programming APIs emerged and matured to facilitate productive GPGPU programming, starting from assembly tools all the way to higher level languages resembling CPU traditional languages like C which will be examined below. [109]

2.3.2 GPGPU Programming Model - APIs

Although historically many GPGPU programming frameworks and APIs have been developed by various vendors and up to date there is not a single golden-standard globally accepted API the most widely used one by far is NVIDIA Compute Unified Device Architecture (CUDA). We will briefly discuss alternatives but our emphasis will be placed on the CUDA API.

OpenCL
OpenCL is a framework, originally developed by Apple and currently maintained by the non-profit Khronos Group, for authoring programs meant to be executed on heterogeneous platforms including, but not limited to, CPU-GPU platforms. Computing systems executing OpenCL compatible applications consist of compute devices which process functions, named kernels, created to exploit task and data level parallelism. All well known GPU vendors have implemented OpenCL compatible APIs for their products [45].

NVIDIA CUDA
CUDA [86] is a general purpose parallel computing platform and programming model introduced by NVIDIA in 2006 enabling NVIDIA GPUs to utilize their parallel compute engine to solve complex computational problems. It is implemented as an extension of various programming languages with the most prevalent ones being C/C++. It is designed to enable automatic scalability adapting to the wide model availability of NVIDIA GPUs and their different characteristics i.e. the number of processing cores as shown in Figure 2.2 (more on processing cores in subsection 2.3.3).

CUDA provides programmers with the opportunity to write specialized functions, named kernels just like their OpenCL counterparts, which will be then offloaded by the CPU to the GPU via a PCI-Express interface to be executed in parallel N times by N different CUDA threads with N being defined by the programmer implicitly via a specialized kernel call containing an execution configuration.

As shown in Figure 2.3 CUDA threads are organized within blocks which are then further organized into grids. Block size is constricted to 1024 in current GPUs due to hardware limitations since all threads comprising a block are executed on the same computing unit which has limited memory resources. This limit does not directly constrict the total number of threads able to be launched by a kernel since CUDA exposes a level of hierarchy above the thread block - the grid which consists of thread blocks. Within the block threads are differentiated based on a one two or three dimensional vector which represents a unique thread id. In the same manner thread blocks are

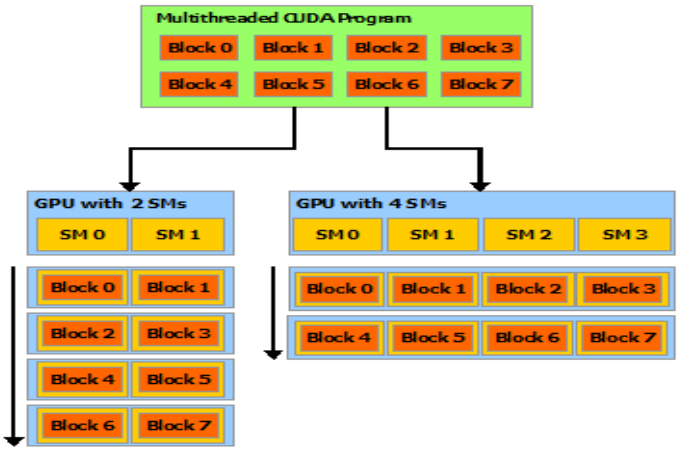


Figure 2.2: CUDA GPU model scalability demonstrated by the ability to evenly distribute blocks among compute cores on cards with a different number of them. [86]

uniquely identified within the grid by another one two or three dimensional index called the block index. Both the size of each block and the size of the grid are defined during the kernel launch. Combining a block id with a thread id yields a unique identity for a thread within the grid.

CUDA threads belonging to the same block are grouped into warps of 32 threads which are executed in lockstep in a SIMT manner. Synchronization among the threads of a block can be explicitly specified by the programmer using specialized barriers while one must always assume independent execution between blocks in order to allow them to be freely scheduled to the available computing cores.

When it comes to memory, threads have access to more than one memory resources. Each thread has its own memory in the form of registers and local memory. Threads within the same block may share another fast-response in-chip scratchpad memory called shared memory, while threads from all blocks have access to a shared global memory residing outside the chip. Two more read-only specialized memory spaces are available, the constant and texture memory spaces. Aside from launching kernels the CUDA API exposes multiple other utilities like copying of data between host (CPU) and device (GPU), reading GPU specifications and synchronizing operations on the card. [86]

HIP

HIP [7] is a solution for GPGPU programming implemented as a C++ runtime API. It is included in the ROCm software stack and resembles the CUDA Programming model. Applications compiled with HIP are portable

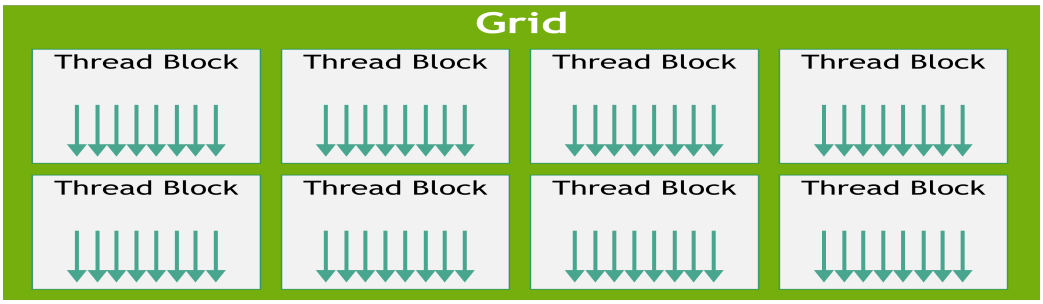


Figure 2.3: Organization of threads into grids, into blocks. The blue downward arrows represent threads. [86]

and can be executed on either AMD or NVIDIA GPUs.

2.3.3 GPGPU Architecture

This subsection takes a look at the architecture of a modern Graphics Processing Unit with emphasis on the hardware utilized by General Purpose computations.

High Level View of the GPU

GPUs are composed of many computational cores named Streaming Multiprocessors (SMs) by NVIDIA and Compute Units by AMD which can be further grouped into clusters. These cores undertake the task of executing a SIMT program as described by the kernel launch that preceded the execution. They are able to execute concurrently up to the order of a thousand threads. Communication of threads within the core is achieved via a scratchpad memory (shared memory in the case of NVIDIA GPUs) and outside of it through the main memory while fast barrier operations are used to impose synchronization. Those cores - much like CPU cores in that way - employ in-chip instruction and data caches aiming to reduce the total amount of outgoing traffic to the lower, shared between cores, memory system levels. When it is inevitable to prevent sending a request to a longer latency memory level, scheduling between the the vast amount of threads executed in the SM assists in hiding the requests response time. [1]

The high computational throughput provided by the core clusters has to be matched with a high throughput memory system in order not to bottleneck it. This is achieved through parallelism in the memory subsystem, by splitting the main memory of the GPU in multiple memory channels (as shown in Figure 2.4) which commonly also include a last level cache. Then address interleaving [34, 1] is used to evenly distribute memory addresses throughout those partitions. Reverse-engineering attempts have shown more

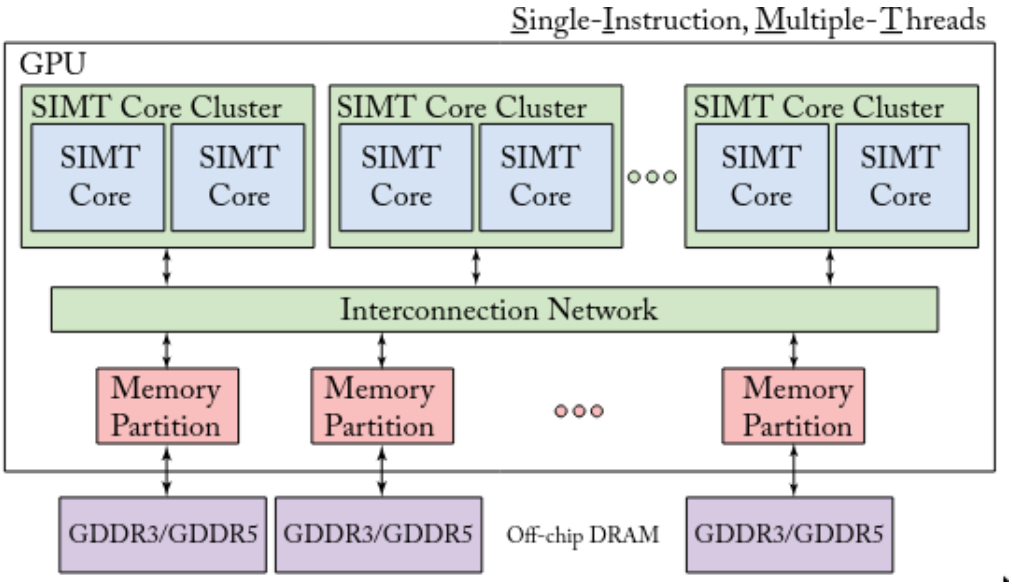


Figure 2.4: High level GPGPU organization. [1]

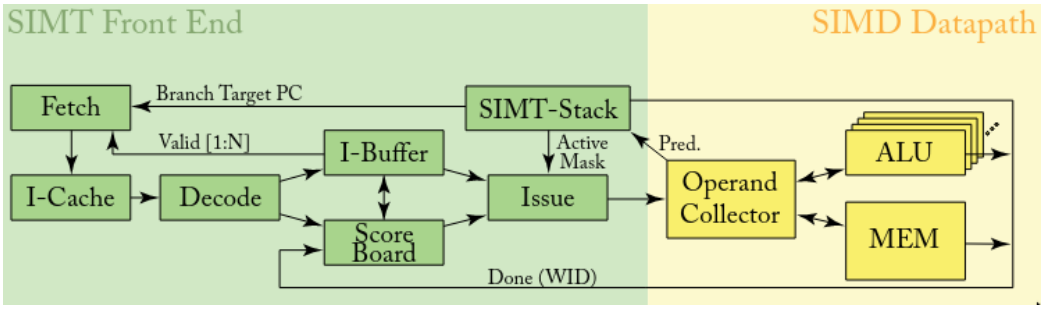


Figure 2.5: Abstraction of the micro-architecture of a generic GPGPU core [1]

complex hash functions being used to divide addresses among partitions [54].

As mentioned previously the layers of memory below the in-SM caches are shared among the cores (or core clusters) and thus a more sophisticated interconnection mechanism between them and the cores is required. GPUs employ a Network-on-Chip (NOC), likely a crossbar [4, 43, 124], which is used to provide the aforementioned connectivity. It does so transparently albeit not blindly by using bandwidth sharing and arbitration policies to benefit the overall performance. [1]

Intra-SM architecture and scheduling

Each SM comprises the main computational pipeline, an abstraction of

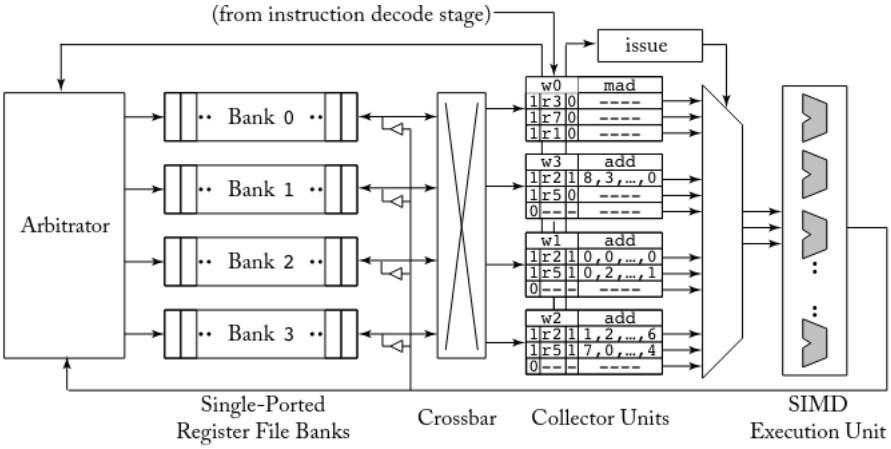


Figure 2.6: Operand collector with four collector units and register file architecture. The collector units are shown buffering instructions and already fetched operands while having blank and invalid slots for the operands that have not yet been fetched by the register file. [1]

which is shown in Figure 2.5, which can then be further broken down into the front-end, involving instruction fetch, decoding and issuing in SIMT fashion and an SIMD back-end involving register file accesses and instruction execution. It also contains the various intra-SM memory systems namely first level instruction and data caches, shared scratchpad memory and the register file. Following is a brief description of those mechanisms.

- **Pipeline Front-End:**
  The pipeline front-end consists mainly of instruction fetch and decode units, instruction buffers and warp schedulers. As mentioned earlier in subsection 2.3.2 threads within a block are grouped in batches. For NVIDIA these groups contain 32 threads and are named "warps" while for AMD cards they can contain up to 64 threads in recent architectures and are called "wavefronts". The term warps will be used from now on. Warps serve as the unit of scheduling within the GPU core. Warp instructions are fetched from memory - or the instruction cache if already present there - in the **instruction buffer**. The instruction buffer can hold the warp's fetched instructions allowing warps to schedule subsequent instructions while waiting for the completion of previous ones provided there are no unmet dependencies. This dependency check is implemented by comparing the data dependencies of the next instruction of a warp already present in the I-buffer against a scoreboard-like structure [1, 73] to eliminate Read-After-Write and

Write-After-Write hazards (Write-After-Read hazards are eliminated by default since most GPUs use in-order instruction issuing). This "scoreboard" can either simply be a single bit per register indicating that a write from a previous instruction to that register is pending or a smaller structure containing few entries per warp which is read when an there is a new entry in the I-buffer and responds with a short bit vector to be stored alongside the instruction indicating its dependencies. When instructions complete and write back to the register file and the appropriate dependency-indicating bits are cleared [1]
GPU cores posses another specialized component in their front-end called the **SIMT-stack**. Since the programmer is presented with the idea that all threads can execute independently while the hardware operates in lockstep this mechanism allows for the serialization of execution of different control paths within the warp. Although this does hurt performance, by imposing unnecessary serialization and inefficient use of the SIMD hardware, and is to be avoided as a programming technique, the SIMT stack attempts to soothe the issue. To do so its entries mask off threads that do not follow the current execution path, blocking them altogether from performing needless computations and not having them be performed only to be discarded later in a similar logic. A re-convergence point is found and placed in each SIMT stack entry indicating when instructions of this warp can continue to execute in lockstep. Usually this is the earliest point after the branch although it is still heavily researched. NVIDIA has also proposed a stack-less re-convergence mechanism which avoids problems like SIMT-deadlocks and seems to be employed in Volta series GPUs [35, 31]
Responsible for the selection of the warp to be executed next is the **warp scheduler**. More than one warp schedulers may exist per core as we see in many modern chips [97, 98] increasing the maximum issue width of a single core. The warp scheduler examines the instruction buffer and selects an instruction from a warp that is ready to issue (in turn for in-order issuing, no unmet dependencies). Since there may be more than one warps with an instruction ready to issue to the back-end warp schedulers feature built-in scheduling policies to resolve such conflicts. If the hardware in the back-end required for the execution of the selected instruction is occupied (i.e. a structural hazard occurred which can occur in various stages of the pipeline) the instruction is replayed and will be considered for scheduling again in a future cycle. Warp schedulers will have a vast pool of warps to schedule from each cycle, enabling them to switch out of the executional pipeline warps facing long stalls hiding this latency with the execution of another warp.

- **Pipeline Back-End:**
  For the warp scheduler to be able to constantly cycle between warps, a large register file (256KB/SM in Kepler, Maxwell, Pascal, Volta, Turing, Ampere and Ada Lovelace [97, 98, 107, 91]) capable of accommodating distinct registers for each warp is necessary, since a context switch where the contents of a warp's registers have to be stored in memory to make room for another warp in the register file is unreasonably expensive for a GPU considering the rate at which it would occur and the high memory latencies. This register file is implemented as an SRAM array whose size is proportional to the number of its ports. In order to achieve a sensible sized register file without compromising on the parallelism of its accesses and thus its throughput, GPU designers have come up with a new structure, the **operand collector**, which seamlessly takes advantage of a cleverly banked register file without the need for programmer interference by exposing the banks to the ISA. Since the operand collector and register file work hand-in-hand they will be both be briefly discussed here despite the fact that the register file typically is not a part of the computational pipeline. As mentioned previously the register file is banked, meaning that instead of having multiple ports on a large register block more than one single-ported memories are used simulating the multi-ported design. Registers required for computations are allocated to banks in an interleaved manner in order to provide the chance to a single warps instruction to fetch all its operands in a single cycle. In order to take advantage of this parallelism the operand collector is employed as shown in Figure 2.6. It replaces instruction staging registers, while containing multiple collector units which in turn contain enough space to house all the operands of an instruction. By having multiple collector units, multiple instructions can be scheduled concurrently for register file reading, resulting in a wider bank coverage and thus higher bank level parallelism. Register file fetches are forwarded to the register file where an arbitrator decides who will access the banks first in case of conflicts. In case of such conflicts the operand collector utilizes scheduling mechanism to tolerate them. Special care is taken to ensure that Write-after-Read hazards are handled appropriately since the operand collector may issue instructions for execution out of program order, when an earlier warp instruction faces multiple conflicts.
  When a warp instruction has fetched all its operands and the SIMT execution mask has been determined it is forwarded to the **execution units**. Those are heterogeneous, with modern GPUs containing integer function units, single and double precision floating point function

units, special function units for mathematical operations like sine or square root, load/store units and specialized function units such as tensor cores [97, 98, 107, 91]. Each function unit can be as wide as the GPU's warp size, meaning it has 32 lanes for modern NVIDIA GPUs, or as wide as a half-warp. They can be clocked at higher frequencies and be pipelined to improve performance per functional unit area. [1]

- **Scratchpad/shared memory:**
  Also known as local memory by OpenCL standards shared memory is an in-core low latency memory space shared between all threads of a block. Its latency is low enough for it to have been called Global Register File as it is comparable to that of the register file with its data not being exclusive to a single warp. It is exposed to the programmer who explicitly defines what will be stored there. Since it is implemented as a banked SRAM of limited size programmers need to be wary of how much data they store there - as it could limit the amount of warps able to fit within the SM - and the way data stored there is accessed as it could result in bank conflicts increasing response latency.

- **L1 data cache:**
  Much like CPU Level 1 caches, GPU SM L1 data cache stores a subset of the global memory address space and is shared within the SM. It caches frequently used and mainly read only data as GPUs do not normally implement cache coherence protocols which would make writes to the L2 challenging [1]. Typically the same chip is used both for data L1 cache and shared memory with the size each will occupy being configurable [109]. However L1 caches in GPUs are typically small and shared by many threads due to the high amount of threads residing within the SM resulting in high levels of thrashing hurting locality [121, 36]. L1 caches can still help coalesce accesses from within a warp by propagating coalesced request to lower levels even though individual thread accesses might be irregular provided the data needed by the threads fall within a single cache block. [1]

- **L1 texture cache:**
  The texture cache is a specialized read-only cache which stores images, called textures, who are meant to be mapped on triangles of the desired 3D scene essentially giving texture, as the name implies, to the scene via a process called texture mapping. Texture mapping involves the lookup of the corresponding texel (TEXture ELements) for each pixel on the screen. Commonly bilinear and other more sophisticated filtering techniques such as anisotropic filtering are used. There exists

substantial spatial locality in texture mapping since the computation of texture for one pixel usually involves its neighboring pixels and the texture maps level is chosen to match the pixel level of detail, hence movements in the pixel space mean analogous movement in the texture space. That is why texture caches are optimized around 2D spatial locality and aim to provide low latency - high throughput responses to texture units. It has been observed in GPUs for the L1 data cache and the texture cache to be unified. [48, 32, 1]

- **Constant cache** The constant cache is once again a read-only cache. NVIDIA GPUs feature another logical, limited at 64KB memory space called the constant memory space. Typically data used by all threads is stored there by host code. Although this is part of the main global device memory it is cached in the SM at 8KB caches for Compute Capability greater or equal to 6.1 NVIDIA GPUs. This serves especially well when multiple threads require the same value as this cache is optimized for the purpose of broadcasting values to multiple threads. [86, 126]

In NVIDIA GPUs starting with Pascal, SMs are partitioned in two and four subpartitions for Pascal [95] and Volta [97], Ampere [107], Turing [98], Ada Lovelace [91] respectively. Each partition contains its own instruction buffer, warp scheduler, register file partition, operand collector units and functional units as shown in figure 2.7.

Shared Memory Architecture As mentioned in the start of the subsection, global GPU memory accessible from all threads is mainly stored outside the cores in partitions. Each partition commonly contains last level caches (L2 exists in almost all commercially available GPUs [1] with newer AMD GPUs also implementing a third level of cache -L3- named, under their brand, Infinity Cache [6]), a Raster Operation Pipeline (ROP) responsible for graphics as well as atomic operations (Although in more recent GPU architectures such as Ampere it is decoupled from structures like the L2 and DRAM and placed closer to the SMs [92]), one or more memory access schedulers (also referred to as memory controllers) connecting the L2 portion to its DRAM counterpart, , and the DRAM itself. [1, 54]. In the following paragraph these components will be broken down:

- **Last Level Caches:** The main focus will be placed on the L2 cache as it is a more established technology with more available information around it. L2 cache is optimized to improve throughput per unit area. It is typically a sectored cache [115] with each cache line, allocated

Figure 2.7: Volta GV100 Streaming Multiprocessor (SM) [97]

for use either by store instructions or load instruction, containing four 32-byte sectors so that individual DRAM fetches match GDDR5's and GDDR6's supported 32 byte atom size [120, 55]. Since GPUs are built around coalesced accesses L2 caches are also optimized around them, not reading data from memory before overwriting entire lines, in contrast with CPU cache architectures, when writes are attempted. If the access is uncoalesced a less optimized pathway has to be followed, either bypassing the L2 or using valid bits. L2 caches are commonly sub-partitioned into slices with disjoint tag and data arrays. [1]

- **ROP:** In General Purpose usage of the GPU the ROP's role is to use its included function units to execute atomic operations used directly by the executed algorithm or in synchronization mechanisms between different thread blocks. The ROP includes a cache useful for pipelining atomic operations to the same memory location.

- **Memory Access Schedulers:** Also known as DRAM schedulers can be as simple as a single scheduler in each partition connecting the L2 and DRAM up to partitions containing multiple such schedulers. Each one contains separate logic for reads/writes from and to DRAM while it employs scheduling policies regarding the order in which accesses are sent to the DRAM. One of the most common orderings is that of grouping together requests that will access the same DRAM row but there is ongoing research regarding new general purpose and application specific DRAM schedulers [17, 71, 59]. [1]

- **DRAM:** DRAM is the last level of storage within the GPU. It is much larger than the memories described earlier and is used to house the vast datasets necessary for many computational algorithms. It uses capacitors to store individual data bits which are read in rows and stored in a memory structure called row buffer. Since there are physical delays when switching rows, related to the low-level mechanisms involved in DRAM accesses, consecutive read/write operations on the same row face smaller latencies. Mitigating the performance hit row-misses impose, is achieved by breaking down DRAM further into banks with separate row buffers and wisely configuring the DRAM schedulers as mentioned above.

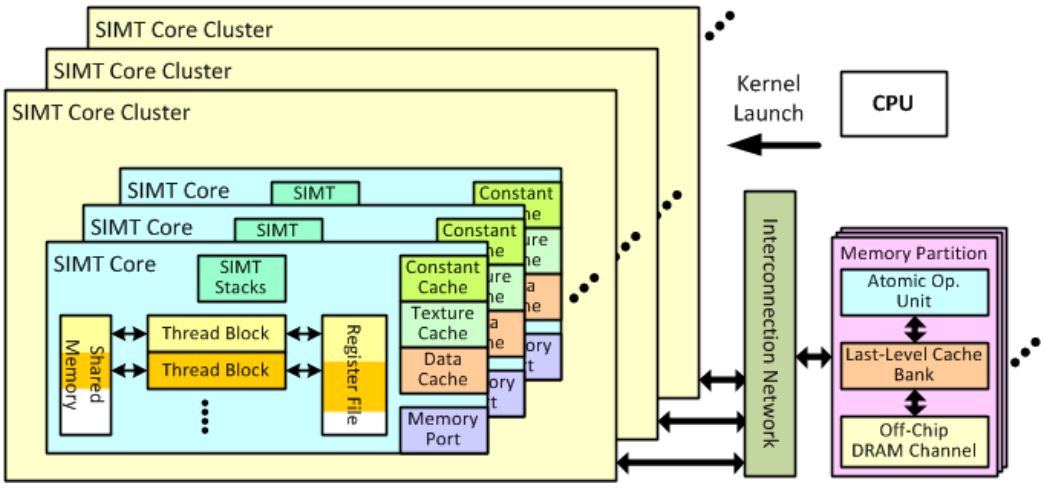


Figure 2.8: High level GPGPU-Sim architecture [123]

2.4 Simulation Tools

2.4.1 GPGPU-Sim

GPGPU-sim [12] is a constantly updated (version 4.2.0 at the time of writing this), state of the art, cycle accurate simulator modelling large scale, highly parallel modern General Purpose GPUs. It supports execution of Parallel Thread Execution (PTX) [103] code, a low level virtual machine and instruction set architecture for programming the GPU as if it was a general purpose data parallel computing device. It models the largest portion of the GPUs pipeline discussed in the previous subsection providing accurate results compared to those collected by actual hardware despite the fact that there exist known structures which are not modelled - or are modelled differently than they are actually implemented, in a degree due to the vast amount of undisclosed information from GPU Vendors. Although GPGPU-Sim closely estimates the performance of actual cards, design space exploration is also available via adjustable configuration files since the simulated hardware's parameters are - for the most part - read from a text file containing simulation options. This subsection will go in short detail of the modelling of various computational pipeline and memory structures for simulation purposes. It functions by interposing as the CUDA Runtime Library and redirecting calls to the CUDA Runtime API to its own functions thus requiring no modifications to a tested application's source code. The PTX of the application is extracted by the simulator when it is first invoked and the device code related to kernel execution is translated to basic operations (e.g. integer op-

eration) and the type, along other important characteristics, is stored in the simulators memory. It is important to note that GPGPU-Sim is a timing simulator and is distinct, but works in conjunction with the functional simulator CUDA-sim, that calculates the application's results.

Top Level Organization
Streaming Multiprocessors are simulated by structures abstractly named Single Instruction Multiple Thread cores (SIMT cores). Those structures can the be accumulated in larger groups in which SIMT cores share buffers to and from the interconnection network. The latter is implemented using the booksim [56] network simulator and on its other end attaches to memory partitions consisting of L2 cache (optionally) and DRAM. A high level representation of the simulated GPU architecture is presented in Figure 2.8. Those structures (SIMT cores, Interconnection Network, L2 cache and DRAM) all feature independent clock domains, bridged by buffers filled at the source domain's clock rate and drained at the destinations. All the information presented below is derived from the GPGPU-Sim 3.1.1 manual [123] along personal analysis of the simulators source code.

SIMT core
Each SIMT core comprises six main pipeline stages in the simulator: fetch, decode, issue, read_operands, execute and writeback. They are all abstractions of the actual hardware operations aimed to provide the same end result cycle-wise.

1. **Fetch:** The fetch stage preceding the decode simply gets the PC of a warp, either from the instruction cache or the memory, and places it in a fetch buffer, provided it has space and the same warp does not have pending instructions in the instruction buffer, to be found by the decode stage. The aforementioned instruction cache is modelled similarly to any other cache in the simulator and is highly configurable giving access to its size (block size, sets, associativity), replacement policy, number of miss status holding registers (MSHRs) and more.

2. **Decode:** The instruction buffer is capable of storing up to two instructions per warp (this number is configurable only at compile time) along with a flag regarding the eligibility of the instruction for issuing. This structure is being fed instructions by the decode stage by requesting the instruction, given its Program Counter (PC), from the functional simulator and places them in the instruction buffer provided it is not full of valid, yet to be executed instructions.

3. **Issue:** Issuing is handled by the warp schedulers. The number of warp schedulers within the core is configurable and along them is the sub-partitioning of each individual SM in sub-cores by GPGPU-Sim terminology. Sub-cores are the analogous of SM partitions with each warp scheduler being isolated and possessing its own register file and execution units. The warp scheduler first selects an available warp based on the scheduling policy specified in the configuration. During this step the instruction selected will be checked for various hazards, such as control hazards which are checked via the SIMT-stack (implemented as an array with each entry being the SIMT stack of a warp), data hazards against a scoreboard storing the registers to which a write is pending for every warp (sets for each warp) and finally structural hazards defined as a lack of space in the pipeline registers preceding the execution pipeline (The number of all pipeline registers mentioned here is configurable). Also a check on programmer specified barriers affecting the warps will be conducted before issuing. Once an instruction is successfully issued, meaning that it is forwarded to the pipeline registers of the next stage, its functional execution is triggered and the result becomes known facilitating the update of the SIMT-stack which happens in this stage as well. Also new register reservation entries are made in the scoreboard for the registers which the instruction will affect.

4. **Read Operands:** This stage involves the operand collector and all operations associated with it. In GPGPU-Sim those operations consist of allocating instructions to free collector units, processing register read requests that do not present with bank conflicts, forwarding register read results to the execution units and scheduling writebacks to the register file. In the simulator collector units are highly configurable exposing not only the number of collector units they contain along with the number of their ports but also their kind. Specifically collector units can either be specialized per instruction type or generalized for all instructions.

5. **Execute:** Execution units are objects of the same class of generalized pipelined SIMD functional unit. GPGPU-Sim models all functional units named in subsection 2.3.3 (integer, single/double precision floating point, special function units, tensor cores and a single load/store unit) as well as an abstract placeholder special functional unit for potential new operations. The quantity of each of the function units is configurable, allowing for some of them to even not exist at all by hav-

ing other overtake their function. As well as many other parameters their amount for tested configurations does not always match the actual GPU however the simulation results yielded correlate to those of the original hardware. Functional units receive instructions and operands from the collector unit via a set of pipeline registers and when finished executing write the results to another set of writeback pipeline registers. Instruction latencies (which are also configurable) are modeled via a set of chained pipeline registers shifted each cycle towards the end writeback pipeline register. Another common dispatch register is used to model instruction initiation latency blocking the issuing of following instructions as to model instruction throughput. The initiation latency is part of the total latency and the position of the piepeline registers in which an instruction that has finished waiting will be placed is adjusted based on the time spent waiting in the dispatch register.

6. **Writeback:** The writeback stage is common across execution units and in order to avoid conflicts a slot in its pipeline register has to be reserved ahead of instruction issuing to an execution unit (with the exception of the load/store unit which has its own writeback stage). Then the operand collector is responsible for allocating writes to the Register File.

Interconnection Network

The interconnection network is modelled as two separate networks one servicing traffic from the cores to the memory partitions and the other one traffic moving in the opposite direction. This is intended to eliminate the risk of circular dependencies causing deadlocks for packets in the network. This network is built off nodes connected at their start to input buffers with configurable size and at their ejection point to output buffers, one for each virtual channel. The unit of scheduling here is the flit, a subset of the packet. Packets are broken down to flits when moving from injection buffers to the network and are packed back when popped from the boundary buffers were they are placed after leaving the ejection buffers.

Memory Partitions

Figure 2.9 presents an abstraction of the simulated memory partitions of GPGPU-Sim. Memory requests received from the interconnection network are forwarded to the appropriate Memory Partition via a FIFO queue. A specialised simulated structure called the Raster Operations Pipeline is responsible for simulating the L2 latency. It is a queue preceding the main

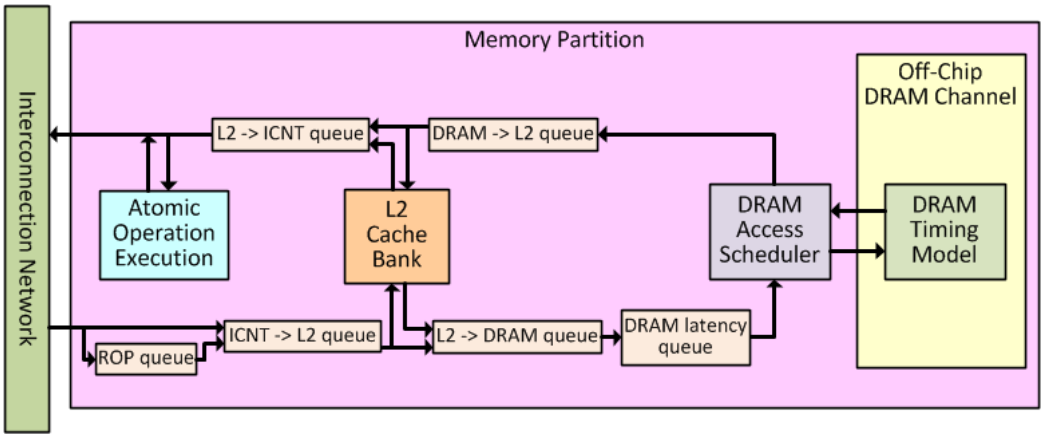


Figure 2.9: Memory partition architecture of GPGPU-sim [123]

interconnection to L2 queue. This structures function is different than that of a traditional GPUs ROP as described in 2.3.3. Cache banks pop one request from the last queue per cycle and responds via another queue going the other direction. Misses in the L2 are forwarded to another queue interposing between the L2 bank and the off-chip DRAM. Following this queue is another latency queue modelling DRAM latency and as with L2, responses from DRAM are placed in a queue to be written back to L2 and forwarded to the interconnection network. These latencies have been measured through micro-benchmarks and adjusted accordingly. A DRAM scheduler is responsible for popping requests from the L2 to DRAM queue and issuing them to the DRAM arrays. The simulator by default supports two schedulers a simple First-In-First-Out (FIFO) and a First-Row First Come First Serve scheduler which prioritizes requests that will access the previously accessed row. Inside the DRAM module the various DRAM timing configurable parameters are modelled based on the clock of the DRAM domain.

Block Scheduling Every cycle a thread block (if available) is chosen for issuing to each SIMT core provided there is enough space to house it. To select the SIMT core which will be assigned a block, the simulator goes through all the SIMT clusters and all SIMT cores within the cluster in a round robin fashion and issues one block. When a kernel finishes the next kernel's blocks issuing will continue where the last one's ended so that all SIMT cores have had at least a block to execute before one executes a second block.

2.4.2 Accel-Sim

Accel-Sim [67] is a simulation framework providing a front-end which facilitates trace-driven execution of the NVIDIA GPU's native machine Instruction Set Architecture while maintaining support for execution-driven simulation of the virtual ISA. The motivation behind this is the issue execution-driven simulation has to keep up with industrial changes so that research does not fall behind the state of the art assumptions. Industrial simulation tools are more often than not undisclosed to the public and as a result public researchers on the matter have to ensure that the assumptions their tools make hold true when testing newer hardware. This proves hard as although vendors provide binary compatibility across generation through the use of virtual ISAs they make drastic changes in the micro-architecture and the actual machine ISA which - even when documented - have to be implemented by open source simulators. Accel-Sim utilizes NVBit [128] to build a tool capable of producing machine ISA instructions from all CUDA binaries even those built using closed source highly optimized libraries like cuDNN [93], cuBLAS [85], cuFFT [87] and many others and then converting them to an internal representation which does not depend on the ISA. Accel-Sim [67] wraps around the latest co-developed version of GPGPU-Sim. It has been enhanced with the ability to output hardware data with direct counterparts in the data provided by NVIDIA profilers enabling another major feature of Accel-Sim [67] - the ability to automatically tune and validate a performance model for a newly released GPU by using a microbenchmark suite to detect changes in latencies, hardware and geometry.

2.4.3 AccelWattch

AccelWattch [63] is a GPU cycle-level power model validated against a NVIDIA Volta GPU displaying high accuracy. Many of its parameters are configurable much like GPGPU-Sim [12] from which it can acquire runtime performance statistics simulating the power consumption of the tested scenario. It reports dynamic, static and constant power consumption for the entire card while considering power-gating, control-flow divergence and Dynamic Voltage Frequency Scaling (DVFS) [9] if it exists. It is able to do so by using specialized micro-benchmarks with known hardware impact such as only powering on certain structures of a physical GPU, to obtain power measurements for components. Benchmarks run under varying clock frequencies are used to estimate constant power as static and dynamic power are affected by frequency either directly or through voltage under DVFS, which is employed by modern GPUs, so by mathematical analysis of the total

power results of such a procedure, constant power can be obtained. With constant power known, micro-benchmarks powering on specific number of lanes of a specific number of SMs are executed on the GPU and the activation power consumption results are used to power models revealing static power consumption aware of warp divergence. It is observed that static power can be accurately approximated with a linear model based on the number of active threads per warp. With a simple division the static power per SM is calculated which is then used for simulator calcualtions. Idle SM power is estimated by running micro-benchmarks occupying the entire GPU thus leaving no idle SMs to infer the summation of static plus dynamic power of a single SM by subtracting the constant from the total power and dividing by the number of SMs and then leaving some SMs idle and assuming that active SMs still consume the calculated amount of power so the rest of the consumption has to originate from the idle SMs. Finally dynamic power modelling by tuning scaling factors through an iterative approach involving the solution of an equation system based on execution metrics either on hardware or on a simulator and hardware power measurements. Once all those values have been put together and the power model is complete, execution statistics can be passed from Accel-Sim [67] to properly scale the calculated individual powers to receive an accurate power estimate of an applications execution. The power simulator ships pre-configured against the Volta GV100 and offers configurable parameters allowing for design space exploration.

2.5 Heterogeneous Computing

2.5.1 Introduction

Heterogeneous computing is a broad term encapsulating the use of more than one distinct types of processing elements with each of them having the capability to execute the task it performs better within a workflow. Those processors may reside on the same chip or on different chips connected by a network and they can share ISAs or feature entirely different ones [117]. The aim of such an effort is to improve performance and energy efficiency of systems that can otherwise be held back by the limitations of Moore's law.

2.5.2 Multiple-ISA Heterogeneous Systems

The traditional concept of heterogeneity involves a primary processor and co-processors often referred to as accelerators connected through a high band-

width interface. A common example of such a configuration is a CPU-GPU system where the main processor is usually the CPU, managing the workload and offloading certain parts to the GPU which serves as the accelerator. In the most common scenario the interface connecting the two is the PCIe [80] bus. CPU-FPGA systems are also common where FPGA are highly specialized for a specific task. Those are just a few specific examples since a system may include all three computing platforms (CPU, GPU, FPGA) and even more. A difficulty presented with such systems is that all those devices use different ISAs commonly requiring effort on the programmers side to write code differently for each part of their program meant to be executed on a different device. Multiple-ISA heterogeneous systems can feature their processors and co-processors on different chips such as the aforementioned CPU-GPU example interconnected by PCIe or co-exist on the same die with notable examples being integrated graphics processors such as Intel HD Graphics or AMD APUs [42, 14] and CPU-FPGA designs.

2.5.3 Single-ISA Heterogeneous Systems

Single-ISA heterogeneous systems are predominantly found in the form of Single-ISA heterogeneous multi-core architectures [69, 70, 68] which incorporate cores that share the same ISA but can be vastly different regarding their architecture. The widely varying demands of the large amount of applications executed on processing cores provides the motivation for an architecture which can adapt and best suite those needs. Applications which expose a high level of ILP would benefit from a wide superscalar Out-Of-Order processor however the same processor's complex and large pipeline would go to waste executing a memory bandwidth limited application or one that faces many control hazards, needlessly expending more power and under-utilizing the die. By having more than one type of cores within the chip and using appropriate scheduling mechanisms each application can be executed on the hardware which suits it best achieving potentially improvements in performance, energy and die utilization. A commercially available widely-used example of such an architecture is ARM's bigLITTLE [21].

Chapter 3

Related Work

3.1 Introduction

As far as our knowledge goes this work is pioneering in introducing tailored SM level heterogeneity on the same GPU chip, based on studying the needs of CUDA application templates, as well as a simulator that supports modeling a wide range of such architectures. In this chapter we will briefly discuss related work.

3.2 Heterogeneity in the CPU world

The big.LITTLE [10] architecture proposed and designed by ARM is an implementation of a single-ISA heterogeneous architecture as described in 2.5.3. Its original version comprises the ARM Cortex-A15, a multi-issue, out-of-

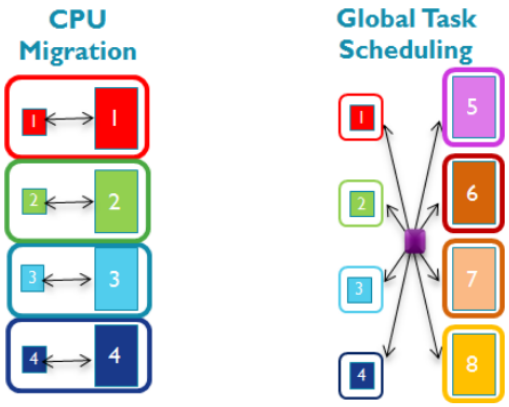


Figure 3.1: big.LITTLE Software Models [10]

order processor, and the ARM Cortex-A7 a simple eight stage in-order one. It is a design targeted at mobile devices, recognizing their inherent need for energy efficiency due to both the battery capacity limit as well as thermals. Applications executed on mobile devices present with large variability, ranging from intensive games to low intensity android system services. Optimizing for the most demanding application and using powerful, power hungry processors is detrimental to the energy efficiency of the device when executing commonly latent, persistent background services while optimizing optimizing for the latter would severely impact the performance of the first group. Both processors despite their micro-architectural differences share the same ISA with programmers having the ability to seamlessly program for such a processor as they would for any other. This is also facilitated by the fact that special care has been taken for big and LITTLE cores to be coherent on a hardware level. The first version of bigLITTLE includes two major software models for its operation: CPU Migration and Global Task Scheduling (GTS) as shown in Figure 3.1. CPU migration assumes an equal amount of big and LITTLE cores which are paired one to one. The system views these pairs as a single logical core and within the cluster one core is active at any time based on the load. The other alternative, GTS, is a more sophisticated and flexible model requiring a scheduler aware of the processors' compute capability alongside individual applications behaviour based on statistical execution data and heuristics. In this scheme all cores are visible as independent logical cores and the scheduler decides on thread level which core to use, with vacant cores being turned off to conserve power. The number of LITTLE cores does not have to match the number of big ones while no hardware goes unutilized when there is enough work placed on the system. Each thread's load is tracked by the scheduler and thresholds are set so that when the load in question crosses them it migrates to the appropriate core type. Commonly threads start on big cores and the thresholds are checked and migrations occur when they wake up after sleeping or periodically, in order to address threads that do not sleep very often or to utilize under-utilized big cores. Occasionally threads will be migrated downwards to LITTLE cores if they are left idle. An example showing three major migration mechanisms of big.LITTLE is depicted in Figure 3.2. The thread executing initially resides on a LITTLE core and migrates to a big after a periodic check during which the tracked load was above the up migration threshold (forced migration). After sleeping it resumes its execution on the big core as the threads' load metric before sleeping indicates. The threads behaviour has to be modified for a migration to occur (Wake migration). Finally the thread goes to sleep again and its last tracked load value before sleeping is below the down migration threshold so it is moved to a LITTLE core (Wake Migration).

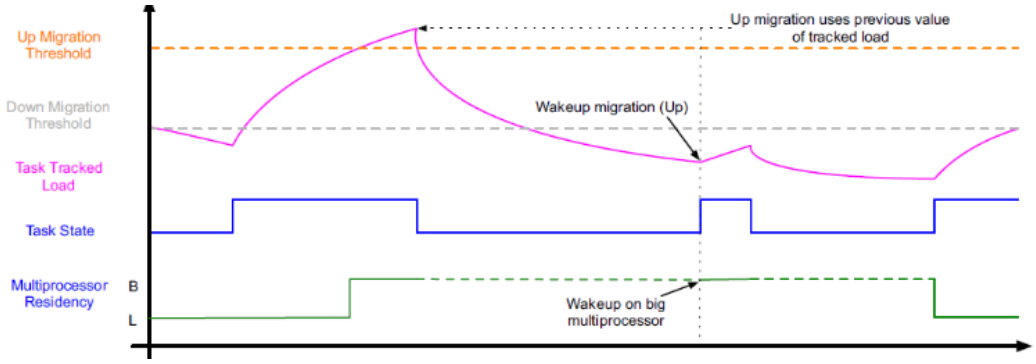


Figure 3.2: big.LITTLE Global Task Scheduling load tracking and migration mechanisms. [10]

More recently the concept of core heterogeneity was introduced to laptop and desktop computers with the release of Intel's Alder Lake architecture [114]. Although said systems may not be as constrained by power consumption as mobile devices their large Out-of-Order cores cannot be scaled out. This is why new efficient cores (E-cores) where introduced alongside stronger performance oriented ones (P-cores). The two types of cores differ in compute capability both due to micro-architectural disparities and feature support like the E-Cores not supporting Simultaneous MultiThreading [125] and clock frequencies. P-Cores also feature an enriched ISA, exploitable by heavier Artificial Intelligence and High Performance Computing applications. Although this makes the architecture heterogeneous-ISA by definition, both processor types support a common subset of ISA despite individual featured extensions [53]. The scheduler responsible for workload distribution between the core types involves cooperation of the Operating System and the underlying hardware. The latter tracks execution performance counters and uses them in an ML model to categorize the workloads. The categorization generated through this process, which leads to core recommendations, is communicated to the OS during context switching. The OS will then take Quality-of-Service constraints and user experience into account as well to decide the core to receive a specific workload.

3.3 GPU Specialization

As hinted in Section 1.3 specialization has been the go-to solution to further improve the performance of the popular GPU accelerators. Some of the solutions both researchers and vendors have proposed and implemented are briefly discussed below.

Tensor cores were added to GPUs, starting with Volta [96], by NVIDIA in an attempt to improve the performance of applications that rely on heavy computational matrix multiplication operations, namely machine learning training and inference. Thus kernels limited by such computational bottlenecks could utilize this new structure to achieve many times increased performance. Nevertheless kernels that do not rely as much on matrix operations are likely to leave tensor cores vacant. Specialization regarding ML has even escaped the boundaries of GPUs with various Deep Learning accelerators emerging utilizing tensor low-precision arithmetics as well as very high bandwidth memory implemented either on FPGAs [118] or as ASICS [47]. Google's Tensor Processing Units (TPUs) [62] are a prime example of an ASIC implemented accelerator, consisting of a systolic array of multiply and accumulate units. Its purpose is to provide better matrix operations performance to aid in Neural Network inference. Although such solutions do improve ML performance they lose the generality offered by GPU devices.

In literature **Light-Weight Out-Of-Order Execution (LOOG)** [51] improves the performance of long-stall-heavy kernels by introducing instruction level parallelism to mitigate the non-productive execution time when most thread groups are stalled waiting for a long latency operation wherever this originates from. Although the performance gains are significant when targeting such kernels, the edge over traditional GPUs is limited during the execution of compute heavy workloads which stress the functional units of the integrated circuit, since those kernels already manage to hide emerging latencies via TLP.

R-GPU [13] also attempts to improve utilization for kernels facing stalls due to high latency memory operations. It does so by re-configuring its SMs into specialized functional units configured to execute one specific operation while chaining them together to create a computational pipeline enabling better data locality exploitation, instruction processing and improving memory bandwidth. Nevertheless as noted by the authors this architecture will yield no benefit for compute bound kernels since R-GPU does not increase the devices compute capabilities so an entire class of kernels remains stagnant performance-wise.

3.4 GPU core breakdown and re-configuration

GPU Domain Specialization via Composable On-Package [40] Architecture suggests disaggregating the GPU monolithic core and creating one part con-

sisting of the SMs and caches up to L2 level as well as a switch and another part consisting of a third level cache and memory controllers. The aforementioned switch will connect the first part to the second if the GPU is specialized to be used for Deep Learning applications as Yaosheng Fu et al. [40] observed that those kernels rely a lot on caches to reduce DRAM traffic as well as high memory bandwidth in contrast with HPC kernels. When the GPU is specialized to run HPC applications the switch will be directly connected to the memory controllers, omitting the extra cache which will not be present in package. Although this work recognizes the different needs of kernel classes and proposes specialization as the solution it can only offer benefit to Machine Learning domain applications when necessary. Additionally despite the fact that their proposal allows for specialization with minimal design cost, it does so at production time and the final product targets one specific category of applications without being re configurable. Our heterogeneous GPU sustains flexibility which is exposed to the programmer at compile-time allowing clusters to better utilize the hardware based on the specific needs of individual kernels. Besides as we showed in detail in section 1.2 not all HPC kernels behave the same when it comes to memory requirements.

3.5 Fusing and Splitting Streaming Multiprocessors

AMOEBA [20] observes disparate kernel behavior regarding the scaling of resources and namely SMs. It suggests that having larger and fewer SMs (as opposed to more smaller SMs) could yield benefits for applications that cause intense loads on the interconnection network, since that load scales with the number of SMs, as well as applications with data locality which could utilize the larger L1 cache and shared memory with larger warps minimizing accesses to global memory and providing better coalescing. On the other hand more smaller SMs behave better with kernels who diverge on branches since larger SMs with wider pipelines suffer more from pipeline stalls due to control divergence. It uses an offline-trained model to predict the scalability of the kernel and fuse or split the SMs into bigger/smaller one respectively to best accommodate its needs. Splitting decisions can also be made dynamically during runtime based on the divergence status of the warp thus splitting highly divergent warps and re-fusing when needed. This imposes a heterogeneity scheme since fused and split SMs can coexist at runtime. However the scaled up SMs can only offer coarsely double all the resources at a bigger pipeline width not taking into account analogies or special structures like specialized

operand collectors resulting potentially in structures remaining unutilized or not exploiting the full potential of specialized targeted changes. Albeit they run smaller warps than AMOEBA suggests for its fused SMs, which could be a small L1 utilization bottleneck, as we observed in our case study of Chapter 5 HPC kernels do not utilize bigger L1 caches at all leading us to believe that the big fused L1 could not be exploited. We also noticed improvements for the Compute class of applications by using specialized operand collectors and GTO warp schedulers, changes which cannot be implemented by simply doubling the resources by joining two SMs.

3.6 One GPU for all

GPU sharing has been proposed and implemented in a variety of ways through the years of the wide adoption of GPUs as general purpose accelerators. In general it aims to improve utilization of hardware by assigning one works idle resources for another. Regarding already implemented solutions, NVIDIA CUDA Streams [104] allow for disparate independent work queues who serialize operations within but allow overlap across them. This enables concurrency between any type of CUDA operations including kernel execution while the API simultaneously provides explicit synchronization calls. NVIDIA since Kepler [100] also supports Multi-Process Service (MPS) [104]. MPS is a software layer interposing between the driver and the application exposing a single GPU context through which CUDA calls from multiple processes are routed. Through MPS concurrency is seamlessly integrated in parallel applications like those utilizing MPI processes. More recently - since Apere [107]- NVIDIA also supports Multi-Instance GPU (MIG) [94] a feature enabling the break down of the GPU in multiple partitions assignable to multiple users. Those partitions and the workloads executed on them do not intersect anywhere in the memory system providing increased isolation and security features to the users. In literature Adriaens et al. [2] observes the poor utilization patterns GPGPU applications present with and proposes statically partitioning the GPU across SMs and allowing different kernels, potentially from different applications, to utilize each portion. They show that such an execution scheme can offer a performance benefit while highlighting the interference penalties sharing of the memory system incurs. More fine-grained intra-SM parallelism has also been studied to use warps of different kernels to utilize structures within the SM itself [27].

Chapter 4

Implementation

4.1 Introduction

In this chapter we go through our work in extending GPGPU-Sim [12], packaged within Accel-Sim, to support simulation of Single-ISA heterogeneous GPUs, featuring two distinct types of SMs who share the same memory system. We also present the new version's capabilities and limitations while also modifying AccelWattch [63] to make it compatible with our design. We showcase our implementation of a CUDA API extension which enables the programmer to specify the type of SM their kernels should execute on at compile or run time. Finally we present the software model of the benchmarks designed to execute on this version of the simulator and a standardized and easily expandable benchmark suite.

4.2 Timing Model

4.2.1 Enabling the heterogeneity

In our efforts to enhance Accel-Sim [67] with the ability to model heterogeneous processing units inside a GPU, several modifications have been implemented as displayed in Figure 4.1. First and foremost, we have expanded its capabilities to handle multiple configuration files. Instead of a single configuration file, Accel-Sim now parses three distinct files (as shown in ①). One file outlines the common GPU structures and simulation options, while the other two are dedicated to each of the two core types, providing a detailed description of their internal micro-architectural configurations.

Furthermore, a modification has been made to how the simulator initializes GPU structures ②. Since the simulator is built in an object-oriented

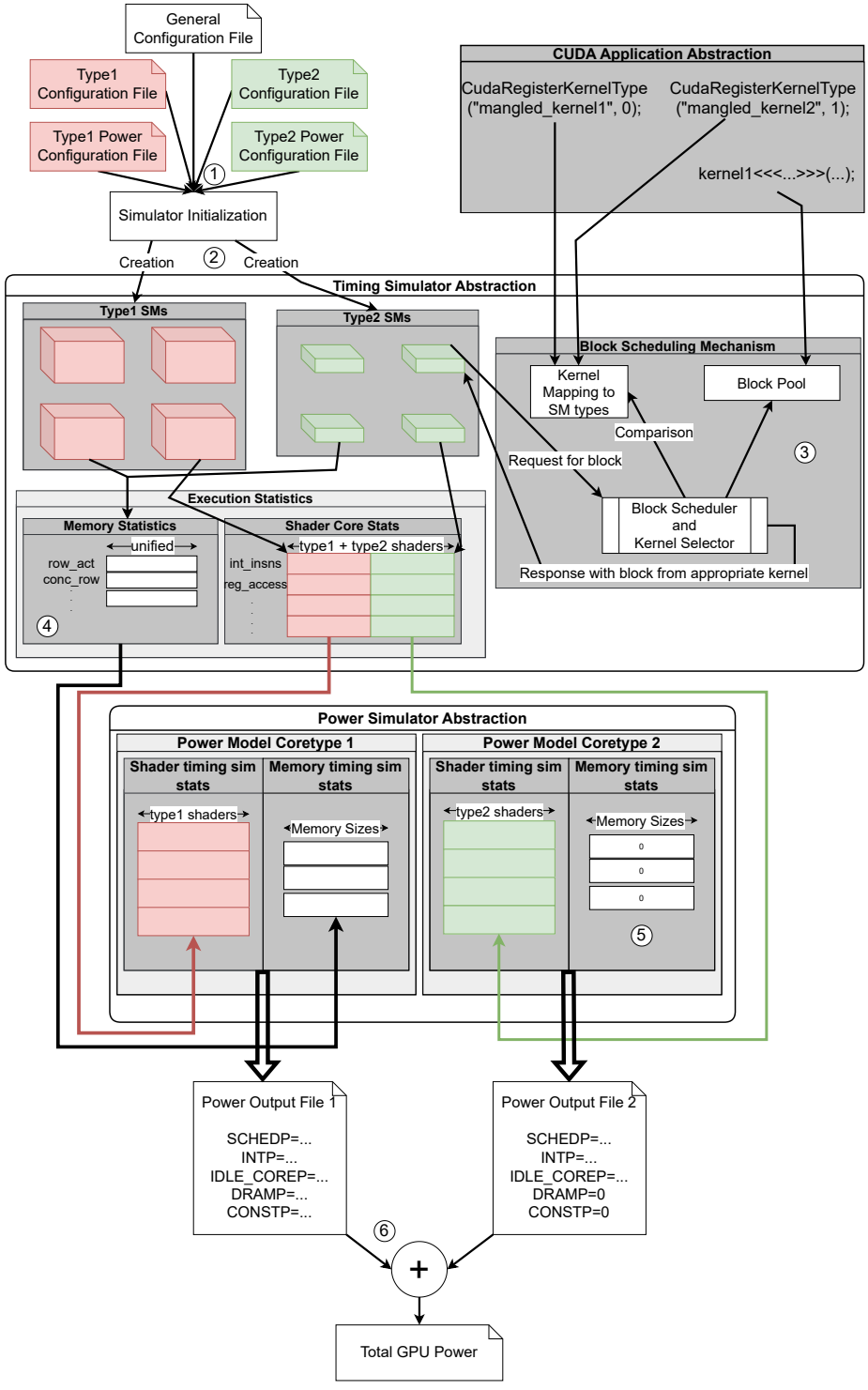


Figure 4.1: Simulator modifications to support heterogeneous architectures.

fashion and each SM cluster and the SMs it contains are separate objects with proper adjustments the heterogeneous cores can still appear to be SIMT clusters of the same class despite their internal disparities. This change allows for heterogeneity between the cores, ensuring that differences between the SMs remain transparent to the rest of the system. Importantly, there is no need to alter the internal processes of each SM, except for certain addressing aspects that depend on the total number of SMs, as the heterogeneity is seamlessly integrated into the system.

The heterogeneous SMs should be capable of running at different clock speeds hence the proposed version uses two separate, independent core clocks to support this feature. However in Accel-Sim the core clock is the simulators global clock, used to calculate several latencies and keep track of execution time, thus with two different SM clocks there is a lack of a global clock which could result in a discrepancy on a time basis between latencies of operations issued by the different type cores. For example the DRAM latency mentioned in subsection 2.4.1 is set to 100 cycles for the NVIDIA V100 GPU [97] whose core is clocked at 1447MHz. This implies a latency on requests attempting to access the DRAM array of approximately 69 ns. If we were to add another type of SM to this card and clock it at 1000MHz while leaving the other type at 1447MHz the 100 clock latency on the DRAM would mean that requests from the faster core type would still take 69 ns but those originating from the new one would spend 100 ns waiting for DRAM to start their processing. This is not sensible computationally wise since DRAM is an off chip structure with a disparate clock from the SMs so latencies in its operation should be independent of each SMs running frequency. The same applies to other global latencies in the simulator. To address this issue we introduce a third artificial simulation clock whose frequency is the lowest common multiple of the frequencies of the two core clocks (Equation 4.1). With this as the global clock latencies that should be independent on core frequency but are modeled based on the global clock, can be scaled accordingly and remain constant on a time basis for all SMs, while intra-SM operations happen at each core type's defined frequency.

$$clock_{global} = lcm(clock_{coreType1}, clock_{coreType2}) \tag{4.1}$$

Our approach also requires modification to the block scheduling mechanism ③. In Accel-Sim, each SM that is not currently engaged in executing a thread block requests a block for execution as described in subsection 2.4.1. This is where the kernel selector comes in which selects a block from the available pool of common thread arrays (CTAs) as it has been defined by previous kernel launches. We introduce a feature wherein this selector exam-

ines the source of the request, specifically identifying the type of SM making the request. Subsequently, it compares the candidate blocks for selection with a list of all kernels registered by the CUDA application programmer as shown in section 4.5 and ③ to execute on that specific type of SM. If a compatible and available block aligns with the SM type, it is designated for execution. Further details regarding the categorization of kernels will be discussed subsequently in Section 4.5.

4.2.2 Timing statistics

In order to provide future researchers the capability of clearly observing the behavior of each of their designed SM types, statistics reporting needs to be altered from its current, accumulated over all SMs, form. Serving that purpose accumulated statistics over the whole GPU should sometimes also be accumulated over the cores of each type separately. This version of the simulator offers a wide variety of per-core-type statistics like instructions per clock, memory reads and writes issued, number of stall cycles due to shared memory conflicts, non-coalesced accesses or serialized constant memory accesses and warp scheduler stalls in its reporting among many others to assist the user in understanding the details of his concurrent kernel execution. The latency reporting for accesses to structures outside the SM like the DRAM has also been enriched with individual reporting for each core type (e.g. maximum memory fetch latency for requests issued by the first type of SMs) allowing for better bottleneck analysis in the common structures as well as inter-kernel interference analysis (more on that in Section 6.2). This is made possible by tagging memory requests exiting the cores with information on the type of SM that issued them. Additionally some new statistics have been implemented, that were not previously supported in Accel-Sim like breakdown of scoreboard stalls as per the type of instruction that caused them.

The reporting has to be backed by modifications in the way those statistics are tracked inside Accel-Sim. Some of them are also passed to AccelWattch [63] as performance counters so counting them appropriately is crucial. By default on Accel-Sim most core statistics are tracked on a per-core basis inside of a single core statistics object. Our approach ④ involves using a single statistics object as well that encompasses all core types, keeping a comprehensive record of simulation statistics per core. For statistics that are not specific to each core but are tracked across all SMs (e.g. warp issue distribution), we have added an extra dimension to the statistics arrays representing the core type. When time comes to report statistics, accumulation occurs differentiated by core type when needed. The separate statistics

object used for tracking memory operations and isolated memory statistics are appropriately resized as well when needed, with some memory statistics being differentiated per the core type associated with them while others like low level DRAM statistics which are used for power estimations are tracked for the entire system.

4.3 Power Model

In order to model the power consumption, we've introduced two separate power models ⑤ which use the pre-implemented AccelWattch [63]. One model is dedicated to estimating the dynamic and static power consumption of the first type of SMs and the components of the shared memory partitions as well as the constant power of the entire card. The second power model focuses on modeling the dynamic and static power consumption of the second type of SMs only. Consequently, there is a requirement to parse two distinct power configuration files ① to support this comprehensive power modeling approach. To calculate the total dynamic power consumption for the GPU excluding the memory partitions, we leverage the superposition principle [46], which states that for a linear system the net response caused by two or more stimuli is the sum of the responses that would have been caused by each stimulus individually . In our case the linear system is the sum of all SMs power contributions, the two different stimuli are the two kernels executing concurrently and the net response is the total dynamic power consumption of type-1 SMs plus that of type-2 SMs. To that we add the dynamic power consumption of the memory subsystem which is calculated for the whole card at once in the first power model taking into account all accesses to the L2 and DRAM regardless of the type of the SM it originated from. Through this process the dynamic power consumption for the entire card is calculated ⑥ as shown in 4.2.

$$P_{dyn,Total} = P_{dyn,SMsType1} + P_{dyn,SMsType2} + P_{dyn,ICNTfromType1SMs} + P_{dyn,ICNTfromType2SMs} + P_{dyn,DRAM} \quad (4.2)$$

This is made possible because when modeling dynamic power consumption AccelWattch [63] uses the per core timing simulation statistics mentioned before, which can be separated based on core type and passed to the appropriate power model along with some other fundamental statistics such as the per type instructions executed. Statistics regarding the memory subsystem below the interconnection network are passed as a whole to the first model

only, which as mentioned in the previous paragraph, is responsible for calculating the entire dynamic power consumption for the out-of-SM memory subsystem.

When addressing static power, AccelWattch's [63] calculations are based on measurements taken from simulated hardware for various structures and lanes activations as mentioned previously in subsection 2.4.3. It is not modeled analytically (per component) but only as a single numeric value based on the types of lanes that are activated which makes it challenging for AccelWatch to accurately report static power when exploring the design space since it is expected to fluctuate when testing different configurations, something that is not modeled. However we model the power consumption by AccelWattch [63] standards for our heterogeneous system since it is considered reliable for design space exploration [63]. The total static power was originally calculated by adding the configuration's measured baseline static power - which corresponds to the first lane activation - plus the extra static power added for each additional lane activation multiplied by the average number of active threads within the warp during the kernel's execution. Since the activation values used in the configurations were measured when all the SMs of the card were active, AccelWattch [63] applies a scaling factor based on the number of active SMs during the simulation. Now we change the scaling factor applied on this measurement to be

$$\frac{ActiveSMsPerCoreType}{\#AllSMs}$$

. By doing so we calculate the portion of static power the SMs of each core type contribute in each of our two models, once again adding them to obtain the total static power.

Constant power on the other hand, which represents the power consumed by peripheral components such as fans and is independent of the SM and memory system configuration [67], has already been determined as a fixed number for each validated GPU in AccelWattch [63]. This value is reported alongside the rest of the power calculations and is incorporated into the first power model thus reported only once.

It is important to highlight that the two power models operate independently and collectively account for all components of the GPU. By using the superposition principle as mentioned previously, we can add the results from both models for a fixed time window (usually that between the end of two kernels execution). This approach effectively yields the total power consumption for the entire graphics card. The simulator separately reports the results of the two models providing the end user with more insight to

which part of their configuration is responsible for what part of the total power consumption.

4.4 Supported Features

The aforementioned extensions and modifications afford users the capability to introduce heterogeneity between the two core types across nearly every architectural feature within the SM that Accel-Sim already accommodates for modification. Specifically, users have the flexibility to differentiate the two core types in the following dimensions:

- **Clock gating on lanes and register file**
- **Maximum number of threads/warps and blocks per SM**
- **Maximum fetch width**
- **Sub-core model:** Accel-Sim supports this mode for Volta/Pascal architectures, where schedulers are isolated in SM sub-partitions (subsections 2.3.3 and 2.4.1) with their own register file and execution units
- **Full within-SM caches configuration (except for cache line size)**: Namely L1 (instruction and data), as well as texture and constant caches configuration (number of sets, block size, associativity, replacement policy, sectored or non sectored cache and sector sizes in case of sectored cache, write policy, allocation policy, write-allocation policy, set index hash function, miss status holding registers configuration, miss queue, port width, access latency, etc.). L1 data cache also supports different write ratios, number of banks as well as bank interleaving and hashing policy.
- **Shared memory:** size, number of ports, access latency, maximum shared memory per block, toggle for unification with L1 cache
- **Register File:** size, banks, port throughput, maximum registers per block.
- **Operand Collector:** Number of collector units and their ports, as well as their type. Accel-Sim supports both generalized and specialized (per execution unit type) operand collectors
- **Warp Schedulers:** Number per core and their scheduling policy (e.g., loose round robin or greedy then oldest), ability for dual issue when the two instructions use different function units

- **Number of pipeline registers for all execution pipeline stages**
- **Number of execution units and their types per SM** (e.g., one group of cores lacking tensor cores or single precision units while the other supports them)

Beyond the inside of the SMs, heterogeneity can be enhanced by assigning different clocks with different frequencies to the two core types and by altering the number of SMs of each type. Certain features are constrained to be identical between the different core types. Specifically:

- **Warp size**: since it is employed during the parsing of the PTX code of the entire application binary before any of the different structures are initialized. While modifying the mechanism to support parsing each kernel separately in the PTX could allow for different warp sizes, this would still impose a restriction for the programmer, as a kernel's type could not be changed dynamically and must be known at load time. Given that 32 is a standard warp size with NVIDIA GPUs, this heterogeneity restriction is considered negligible
- **Texture Cache line size**: as it is used in the functional simulation of 2D texture accesses. Inclusive caches commonly have the same cache line size across all levels in order to avoid multiple snoops in the L1 due to L2 evictions. However since GPUs employ non-inclusive non-exclusive caches [19] it is plausible to attempt to differentiate cache line size between the two core types and consequently between the L1 and L2 cache. It should be noted that changing the line size even on the default version of Accel-Sim is tedious when using sectored caches since the number of sectors that make up a line and the sector size are defined at compile-time within the simulator and they explicitly define the cache line size.
- **Reconvergence Strategy**: as it is utilized during the parsing of the PTX to identify reconvergence points for branches within kernels. While NVIDIA GPUs, to the best of our knowledge, consistently employ post-dominator strategies where diverging threads execution is serialized up until the immediate post-dominator of the branch which led to this behavior [1], there exists literature supporting different strategies being beneficial for different applications [24, 30, 41, 28]. Future versions should allow this strategy to differ between the two core types
- **Instruction Latencies**: as they are encoded within the instructions during parsing of the PTX.

Notably the limitations are scarce compared to the degrees of freedom offered rendering the proposed version of the simulator a robust tool for design space exploration.

4.5 Handing Control to the Programmer

We have successfully enabled heterogeneity in the proposed version of the simulator however it has not yet been determined how the end users can exploit it by assigning kernels to targeted hardware. It is necessary to provide them with the ability to denote the type of SMs each kernel should run on. We introduce a novel feature in our simulator, enabling the explicit definition of a kernel's type. This allows programmers to specify the target type of SMs for a kernel's execution through a newly introduced API call named **cudaRegisterKernelType**. This API call takes two arguments as shown in the function's signature displayed in Listing 4.1: the symbol name of the kernel and an integer (0 or 1) that signifies the type of SMs where the kernel should be executed. In the simulator context, the symbol name of the kernel (i.e. the symbol name of the device function being offloaded to the GPU) is written to a set corresponding to one of the two types of SMs, denoted by the second integer parameter. The block scheduler will then, for each block it is considering to issue to a SM, compare the name of the kernel the block belongs to against the set containing the names of kernels registered for execution on the type of the SM in question. If a match is found and the block is otherwise eligible to reside in that SM, a block issue occurs. This innovation grants programmers the flexibility to categorize their kernels directly within the source code, obviating the need for any modifications to the simulator's source code or configuration files based on the executed benchmark.

```
1 __host__ __cudart_builtin__ cudaError_t CUDARTAPI
    cudaRegisterKernelType(char *kernelName, int type);
```

Listing 4.1: cudaRegisterKernelType function signature. It is a host function meaning that it is called and executed on the host device. Its first argument is the targeted kernel's symbol name (will be mangled during C++ compilation) and the second one is the type of SM the previously specified kernel should run on. Currently supported values for the second argument are 0 and 1 corresponding to the first and second type of SMs

This feature has been integrated into the proposed version of the simulator. However the NVIDIA CUDA compiler does not inherently implement this functionality. Therefore, to utilize it, programmers need to take triv-

ial, nevertheless additional, steps when compiling applications intended to be run on the heterogeneous platform to ensure successful compilation and linking. First and foremost in order for compilation of an application which is intended to be studied on a heterogeneous GPU and includes the cudaRegisterKernelType call to succeed the name of the function has to be defined. This version of the simulator includes an install script (coupled with the uninstall counterpart) which will install a header file containing this definition among the default CUDA header files shipping with the CUDA installation. This way the header file can be included as any other CUDA library file would be included. The next step is to ensure appropriate linking. The simulator operates by interposing as the CUDA runtime library, which should be dynamically linked in the benchmark binary, and servicing those API calls internally. Our simulator contains the implementation for simulating this new API call however the original CUDA Runtime Library against which nvcc (NVIDIA's default CUDA compiler) compiles obviously does not and a linking error will arise during the linking phase of the binary. With the use of certain specific linker flags this error is converted into a warning and despite the symbol not being found during static linking, an entry is reserved for it in the dynamic symbol table of the binary, ensuring correct dynamic linking.

This implementation is specifically designed to suit the needs and be compatible with the simulation environment. In actual hardware there is no necessity of associating a kernel's type with its name and the desired execution hardware can be specified as part of the call configuration, requiring minimal hardware expansion on the GPU - essentially a bit per block denoting the type of the kernel it belongs to and an XOR gate to match the kernel's type with the SM type, which can also be indicated by a bit, when it is considered for issuing. However since we do not have access to the internals of nvcc which transforms the `<<<...>>>` configuration syntax to actual CUDA runtime calls which are then passed to the simulator, we implement a work around using the symbol name of the kernel as its identification.

Actual implemented hardware could also eliminate the need for the programmer to statically assign a kernel to a type of SMs by using either hardware or software predictors to analyze each kernel issued to the GPU and determine the best type of core for its execution. However this static assignment is better suited in the context of providing the user with full control over tested models and other assignment policies are better suited for studying after a final heterogeneous configuration has been devised.

4.6 Concurrent Kernel Applications Modeling

Especially in the context of this research, the simulated scenario necessitates the concurrent execution of two or more kernels of distinct types on a GPU to accommodate kernels of different types to both SM classes (further discussed in Chapter 5). To achieve this, we employ the strategy of executing two separate CUDA applications concurrently. Since Accel-Sim does not currently support Multi-Process Service (MPS), in order to enable simultaneous kernel execution within the simulator, we harness the Streams API, which has been pre-implemented in Accel-Sim. By launching the dissimilar kernels in separate streams, we enable their consideration for concurrent scheduling.

The simulator is initiated through the launch of a CUDA application, which leads to the loading of libcudart (the CUDA runtime library the benchmark binary is linked against which is overridden by the simulator) and the creation of a GPU context. Consequently, each launch of a CUDA application initializes an individual instance of the simulator. To ensure seamless execution, it is imperative that all kernels of a single benchmark are launched from a single process, which inherently uses the same GPU context and thus the same simulator instance. Furthermore, it is essential for the two applications to operate independently, with regard to CPU execution, to prevent GPU inactivity resulting from unwarranted serialization induced by the combination of the two applications.

To address these requirements, a single driver process is utilized, which is responsible for parsing the arguments for both applications, creating two distinct CUDA streams, and launching two threads, each dedicated to an individual application. These threads are equipped with the respective application's arguments, one of the two streams, and any additional essential options.

In the interest of streamlining concurrent execution and preventing unnecessary GPU serialization, application threads are advised to employ asynchronous GPU operations and avoid their blocking counterparts (e.g. cudaMemcpyAsync instead of cudaMemcpy). It is noteworthy that despite being a traditionally blocking operation, cudaMalloc calls do not pose an issue in Accel-Sim as they are not implemented as such. Explicit synchronization operations invoked by the programmer should also be replaced with their per-stream counterparts. Moreover, application threads should initiate the execution of their first kernels in a synchronized manner, thereby mitigating potential race conditions and non-deterministic behavior in the simulation results due to high CPU times attributed to input reading etc. and maintain

low CPU times between additional kernel issuing. This synchronization can be readily achieved through the implementation of appropriate barriers. Listing 4.2 showcases a very basic benchmark example based on the principles discussed above. In this example since there is no significant CPU code and the only serialization between the two kernel launches is the minimal overhead required to asynchronously offload the kernel to the simulated GPU, a single thread is utilized.

```
cudaStream_t stream1, stream2;
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);

cudaRegisterKernelType("mangled_kernel1", 0);
cudaRegisterKernelType("mangled_kernel2", 1);

kernel1<<<grid1,block1,smem1,stream1 >>>(...);
kernel2<<<grid2,block2,smem2,stream2>>>(...);

cudaStreamSynchronize(stream1);
cudaStreamSynchronize(stream2);
```

Listing 4.2: Simple benchmark abstraction consisting of two kernels intended to execute concurrently in the proposed version of the simulator. Kernels are launched on separate streams after their designated execution core type is registered, with no serialization present.

Acknowledging the aforementioned benchmark creation instructions, we developed a benchmark suite fit for execution with this version of the simulator, which ships with it, to assist in research in this field. It contains ten benchmark sources related to HPC (the target scenario is further elaborated on in Chapter 5) and combinations among those with different characteristics. It offers a standardization through which arbitrary benchmarks (designed as benchmarks for the original Accel-Sim or as normal CUDA applications) can be added to it and with trivial changes run along other kernels concurrently. This is facilitated by a standard driver process employing Pthreads [84] which as explained above contains the code to create the two dissimilar CUDA streams, create and initialize potentially useful locks and barriers and parse the arguments necessary for the execution of both applications while separating them appropriately. A pre-implemented structure is responsible for supplying each benchmark with its arguments, one of the streams and its locks, if needed, before the actual simulation threads are launched. We also opt to declare the designated execution cores (via cudaRegisterKernelType) in the driver process for reasons that will be made obvious shortly. The two

simulation threads contain actual benchmark or standard CUDA application source code with minimal changes, compared to its base version, based on the above instructions. Specifically for the user to add a new benchmark to the suite it is required to adjust the benchmarks source code in the following ways:

- replace the main function of the benchmark with a thread subroutine (essentially changing its type, arguments and return value) since the benchmark will be launched from the driver process as a thread
- parse the structure provided by the driver process to obtain its arguments which would otherwise be read from standard input, its CUDA Stream and any additional required items (e.g. locks)
- replace CUDA blocking calls and global synchronizations with their asynchronous and stream counterparts respectively
- add the CUDA stream as an operand to the kernel launch calls
- add barriers if necessary before the first kernel launch of a benchmark to eliminate CPU time of a benchmark from interfering with the simulation results by altering the kernel offload time

Once a benchmark has been successfully added it can be combined with as many of the other added benchmarks as the user deems necessary by only creating new driver programs without changing the code of each benchmark. This is possible because each thread code is independent of the coupled benchmark. The driver process is a fixed piece of code where the only changes necessary to it in order to create a new combination of benchmarks are:

- providing the correct thread routines names to start the desired benchmarks
- defining the appropriate hardware for the execution of the simulated kernels. By doing so in the driver process kernels are eligible to be issued to different SM types when participating in different combinations or for any other reason without affecting the benchmark source code itself.
- defining the number of arguments each application should receive

As such we have resolved the issue arising from the lack of benchmarks available to study such an architecture while inflicting minimal overhead to the

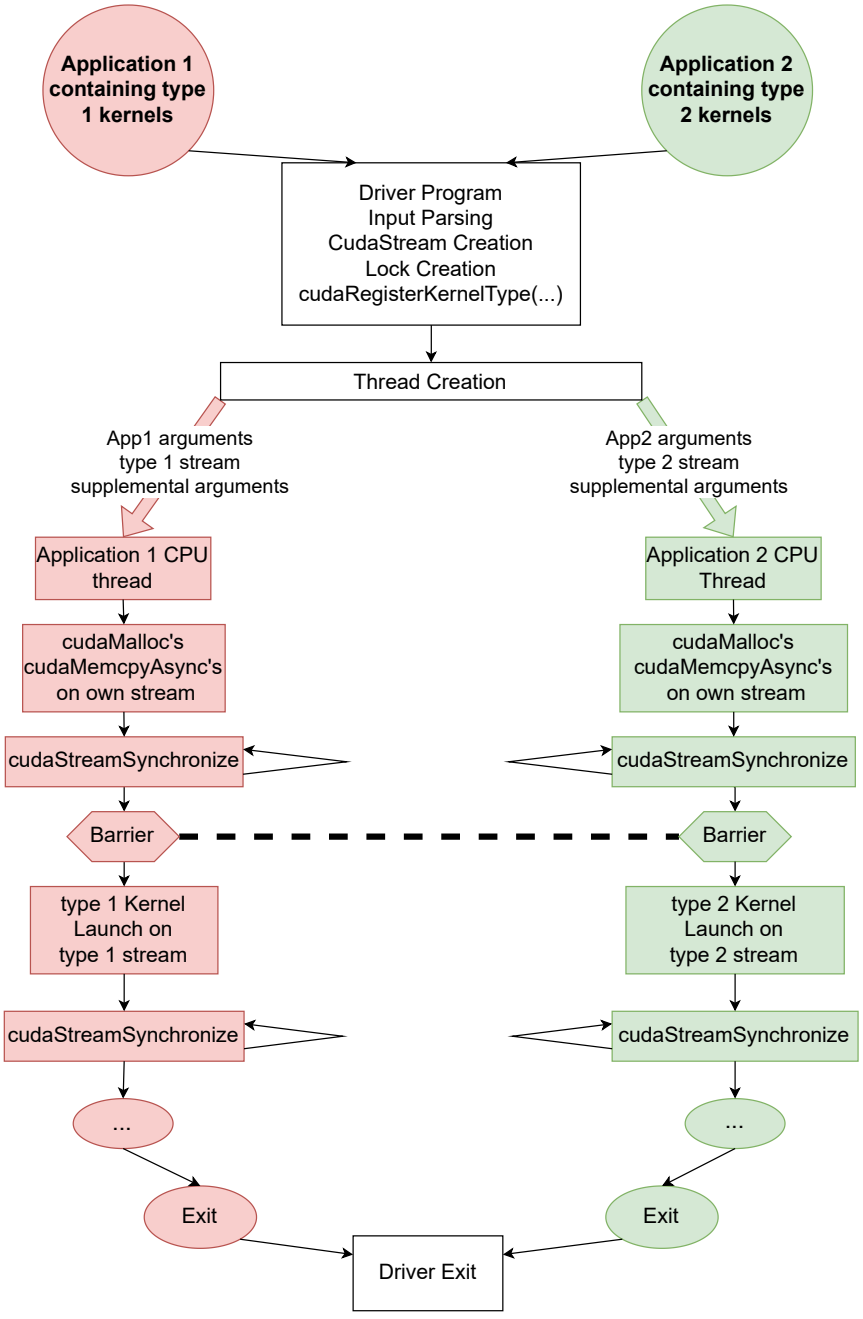


Figure 4.2: Concurrent Heterogeneous Kernel Application Example intended to be tested on the Single-ISA Heterogeneous GPU simulator and designed as part of the benchmark suite accompanying it. A driver process reads both applications arguments and launches them as threads each using its own CUDA stream and being able to launch kernels concurrently on the simulated GPU.

end user for adapting any benchmark of interest to be executable in this version of the simulator.

Figure 4.2 showcases an execution example of an application consisting of a benchmark containing solely type-1 designated kernels and another containing type-2 (although such exclusivity is not mandatory) designed with the standards described above.

In case the user is interested in the concurrent execution of specific kernels from benchmarks containing multiple, simple synchronization mechanisms dependent on command line input are available in the standard, offering launch-time configuration of which kernels will be executed concurrently for each test resulting in multiple benchmarks stemming from a single binary. This implies that concurrent execution testing of kernels originating from the same application is possible, by combining an application with itself involving the same steps as combining it with any other.

Chapter 5

Case Study

5.1 Scenario description

The application scenario, which we study, for the single-ISA heterogeneous GPU adaptation is the execution of scientific computing applications on High Performance Computing (HPC) Clusters. Typically a research center with scientific computing needs is assigned a large HPC Cluster which can contain multiple hosts and nodes [127]. These nodes often are equipped with GPUs [15] which they use to exploit the parallelism exhibited in such applications [79]. Although development in GPUs has made tremendous progress in the Machine Learning domain HPC has lagged behind making it an appealing target domain.

The aim of this work is to allow for kernels with different execution characteristics to run in parallel within the same GPU featuring heterogeneous processing cores. CUDA HPC workloads tend to use multiple CUDA kernels ran sequentially each consuming the data of the previous to calculate the end result. Since HPC nodes tend to run a single type of application at a time (e.g. a specific fluid dynamics application) and its kernels are ran in a strict serialized fashion within the processes due to algorithm-imposed data dependencies one would reasonably come to the conclusion that there is no room for kernel level parallelism in this scenario. However both vendors and literature suggest sharing of GPUs in process/thread level [75, 130, 57, 16, 104, 94], with HPC applications commonly employing said parallelism in CPU level. A real world example of this form of GPU sharing is that among MPI [23] processes, which HPC jobs typically comprise, a practice strongly facilitated by MPS [90] which can allow seamless GPU sharing between MPI processes. The aim of sharing is to increase hardware utilization and reduce total energy consumption as mentioned in Section 1.4. Literature also suggests sharing in HPC node granularity [111] making kernel level parallelism

even more apparent. Additionally cluster users customarily schedule multiple jobs even if they contain only one application (batches containing multiple runs of a specific test) which would be scheduled on the same nodes as is common in HPC clusters. Thus those instances of the application could be executed concurrently and given that they are independent, At a fixed time point they will most likely be at different points in execution and they will have different kernels to execute which can in turn be scheduled in parallel on the GPU. So inter-job kernel parallelism is also exploitable.

These suggestions are also applicable to other fields such as Neural Networks (NN) wherein a continually learning NN [110] could execute kernels, regarding training and inference respectively, simultaneously on the same specially configured heterogeneous GPU or Cloud Computing where providers nowadays commonly offer GPU instances and for which GPU virtualization and sharing between multiple Virtual Machines has been proposed [112]. However the focus of this work is on the HPC scientific computing scenario. The goal of this section is to assemble a proof of concept heterogeneous GPU with the SMs of each category tailored to kernels of certain characteristics whose performance will be later evaluated to showcase the benefits concurrent kernel execution on it, can provide in the target scenario. To do so we will collect a set of kernels representing various types of scientific computations, break them down in two groups and perform a crude yet detailed analysis on major micro-architectural reconfiguration axes to arrive at the final model of our heterogeneous GPU which we will evaluate in the next chapter.

5.2 Workload Identification

5.2.1 Formulating the test set

According to our assumed scenario, we have selected a plethora of scientific computation applications whose kernels are candidate to be run on an HPC Cluster that utilizes GPUs. We feature applications from various domains as shown in Table 5.1. More specifically we include:

- the five main components of the Beam Longitudinal Dynamics (BLonD) code [52], which CERN employs for the simulation of longitudinal beam dynamics in synchrotrons, and exist as individual, standalone benchmarks

- LULESH [50] [64] that is a proxy application whose purpose is to solve a simple Sedov blast problem [116] but nevertheless represents the nu-

Benchmark Name	Benchmark Suite	Domain	#kernels
convolution	BLonD	Particle Physics	1
histogram	BLonD	Particle Physics	1
kick	BLonD	Particle Physics	1
interpolation-kick	BLonD	Particle Physics	2
drift	BLonD	Particle Physics	1
LULESH	LLNL-proxy-apps	Hydrodynamics	26
hotspot	rodinia	Thermal Physics	1
GASAL2.0	(Standalone)	Bionformatics	5
gramschmidt	polybench-gpu	Linear algebra	3
bicg	polybench-gpu	Linear algebra	2

Table 5.1: Scientific Computing Benchmarks used for testing and evaluation

merical algorithms, data motion, and programming style typical in scientific C or C++ based applications

- hotspot, a thermal simulation tool to estimate processor temperature shipped as part of the rodinia-3.1 suite [18]

- GASAL2 [3] a CUDA library containing sequence alignment algorithms regarded as one of the best samples of existing CUDA genomics algorithms [5] and its test case featured in the Genomics-GPU benchmark suite [77]. Through the benchmark four different alignment algorithms can be tested, namely a global alignment algorithm, a semi-global alignment algorithm and two local alignment algorithms (ksw and GASAL's own implementation of a local alignment algorithm)

- two basic linear algebra operation applications from the polybench benchmark suite [44]: a biconjugate gradient method (BiCG) algorithm [108] and a Gram-Schmidt process [74] respective implementations. Linear algebra operations commonly partake in various scientific computation applications and should be represented in our test set.

,

5.2.2 Experimental Methodology

Given the kernels of the above applications we attempt to group them in two large groups based on their SIMD execution patterns by identifying their

bottlenecks and improvement leeways. By creating two broad groups in which kernels can fall into, we will be able to tailor the two heterogeneous SMs types to mend to their specific needs. We propose the following categorization:

- **Compute-Intensive Kernels:** This category encapsulates kernels heavily stressing the computational backend of the SMs pipeline, who could exploit even further thread-level parallelism within the SM but their performance is bounded by the lack of available compute resources.

- **Low-Utilization Kernels:** Kernels placed in this group will under utilize the computational pipeline during their execution essentially wasting resources throughout the GPU without leaving room for much performance improvement due to their insensitivity to micro-architectural adjustments. Common reasons behind that insensitivity can be that the primary performance limiter either resides outside the SM (e.g. DRAM) or that the kernel's programming itself does not allow for high hardware utilization (e.g. small kernel grid size)

With the two target kernel groups now defined we must analyze the HPC scientific computing representative applications in order to create a benchmark set and classify its kernels in the two aforementioned groups.

Studying compile-time kernel characteristics can only reveal limited information regarding their hardware utilization. For example we can determine that a kernel launch with a small grid or a kernel launch with a sufficiently large grid but high register or shared memory usage will limit the available parallelism on the hardware by not filling out the SMs with their maximum supported threads leading to low occupancy. Nevertheless parallelism could be limited for more obscure reasons, such as high memory bus contention, which are not always readily apparent just by studying the source code. Acknowledging the aforementioned observation we approach the kernel categorization as follows. We extensively test the kernels at hand on targeted modified versions of the V100 Volta architecture GPU [97] - comes tested and verified with the Accel-Sim [67] package - which serves as our baseline. Table 5.2 presents the major hardware characteristics of our baseline GPU which are the values used in the simulator but do not necessarily correlate 1:1 to the actual GPU as they can be altered by the developers to provide runtime behavior as close to that of the actual hardware through the simulator. The tests are executed in the PTX simulation environment offered by Accel-Sim [67] and reported timing simulation statistics are collected along with power measurements from AccelWattch [63]. In the bar charts to follow

each couple of bars for a specific kernel represents a tested configuration. The blue hue bars represent the speedup with the corresponding configuration compared to the baseline (first set of bars) which is calculated based on the total execution cycles reported by Accel-Sim [67], while the orange hue bars represent the, relative to the baseline, average power consumption throughout the execution of the kernel. Both baseline and tested configurations will feature 40 SMs (half of the original 80), unless mentioned otherwise, since the grouping will be aimed towards tailoring each category's respective SMs for this new architecture which will feature a fraction of the original SMs for each core type, attempting to stay in par with the original size of an actual, hardware-implemented scientific computation oriented GPU. Additionally the simulated kernel launch latency (latency between issuing of a kernel to the GPU for execution and block execution initiation) will be set to zero on all configurations, since this is modelled as a constant added to any kernels execution time, independent of the specific features of the simulated hardware, merely obscuring performance fluctuations between the tested configurations. Those two changes are the only ones incorporated in the baseline model of our testing, differentiating it from the tested V100 configuration. We compare the performance on the modified hardware to the baseline for various modifications and based on the collected results we choose the category into which each kernel falls. Once the groups in which each kernel falls have been identified more specific targeted hardware changes can be attempted and tested in the same manner leading to the final cores configuration.

At this point it is important to note the limitations of the power model employed in AccelWattch [63]. Specifically AccelWattch [63] operates by obtaining performance counters from Accel-Sim's [67] timing model, regarding accesses to specific components, and then using pre-calculated scaling factors, obtained by running micro-benchmarks on the actual physical card and performing regression on the scaling factors based on the results, to estimate the power consumption. Static and constant powers are also calculated based on measurements from the actual GPU being modelled as explained in Chapter 2. Although this results in accurate power simulation results when testing applications against validated configurations for existing real world GPUs, it provides poor design space exploration as the scaling factors and pre-calculated values for static and constant power cannot be confidently adjusted when modifying the GPU's micro-architectural configuration. Nevertheless AccelWattch [63] and Accel-Sim [67] likewise are the state of the art open source GPU simulation tools and although power measurements may not truly reflect the simulated hardware when changes are applied, they may provide a helpful approximative upper boundary, since through the use

Cores	40
Core Clock	1447MHz
RF Size	256KB/Core
RF Banks/Core	16
Max Threads/Core	2048
Warp Size	32
INT,DP,SP,SFU,Tensor Units/Core	4
Warp Scedulers/Core	4
Issue Width	1
Collector Units/Core	8 8x8ports
Unified L1/Shared mem/Core	32KB/96KB
Shared mem banks/Shared	32
L2 cache	6MB (96KB per partition)

Table 5.2: Specifications of the baseline V100 used in our testing.

of performance counters, increased utilization of potentially expanded hardware will translate to higher power consumption.

5.2.3 Kernel Elimination

We choose to study kernels and not the entire applications as a whole at the time since those will serve as the unit of scheduling to each type of SMs on the targeted scenario. We have to handpick the targeted kernels of the above applications in order to keep the study group reasonably sized and observable. We proceed to do so by including at least one kernel from each of the applications above in order to sustain a wide range of kernels representative of the diverse scientific computing field while omitting kernels which do not take up important portions of an applications total execution time. When measuring execution time for this purpose we put our emphasis on our baseline model, executing benchmarks to completion (with the exception of GASAL being limited to 100 million thread instructions due to its very long simulation time considering the simulator imposes a slowdown in the ranks of millions compared to actual GPU execution) and comparing each kernels execution time to the total execution time of its corresponding benchmark.

Starting with the GASAL benchmarks as observed in Figure 5.1, regardless of the chosen alignment algorithm, the kernel involved in packing the query and target sequences (abbr. GApa) takes up a mean 12.82% of the total, instruction-limited, execution with very low deviation. Since each of these four tests contains only two kernels this pack kernel is considered important and will be included in our study along the four main alignment

Kernel Name	Abbreviation	Kernel Name	Abbreviation
CalcHourglassControlForElems	LUL1	CalcSoundSpeedForElems	LULt
CalcFBHourglassForceForElems	LUL2	UpdateVolumesForElems	LULu
CalcKinematicsForElems	LUL3	CalcCourantConstraintForElems	LULv
InitStressTermsForElems	LULa	CalcHydroConstraintForElems	LULw
IntegrateStressForElems	LULb	convolution	BLco
AddNodeForcesFromElems	LULc	histogram	BLhi
AddNodeForcesFromElems2	LULd	drift	BLdr
CalcAccelerationForNodes	LULe	kick	BLki
ApplyAccelerationBoundaryConditionsForNodes	LULf	precalc_interp_kick	BLik_pre
CalcVelocityForNodes	LULg	linear_interp_kick	BLik
CalcPositionForNodes	LULh	calculate_temp	HOT
CalcLagrangeElementsPart2	LULi	bicg_kernel1	BIC1
CalcMonotonicQGradientsForElems	LULj	bicg_kernel2	BIC2
CalcMonotonicQRegionForElems	LULk	gramschmidt_kernel1	GRA1
ApplyMaterialPropertiesForElemsPart1	LULl	gramschmidt_kernel2	GRA2
EvalEOSForElemsPart1	LULm	gramschmidt_kernel3	GRA3
CalcEnergyForElemsPart1	LULn	gasal_pack	GApa
CalcPressureForElems	LULo	gasal_ksw	GAksw
CalcEnergyForElemsPart2	LULp	gasal_local	GAlo
CalcEnergyForElemsPart3	LULq	gasal_semi_global	GAsg
CalcEnergyForElemsPart4	LULr	gasal_global	GAgl
EvalEOSForElemsPart2	LULs		

Table 5.3: Kernel name abbreviations

kernels. Within each benchmark all kernels are important.

The two linear algebra operation applications Gramschmidt and BICG feature two and three kernels respectively. Figure 5.2 indicates that all of them are important in determining the applications performance except from Gramschmidt's intermediate kernel which takes up less than one percent of the total execution time and is subsequently eliminated from further analysis.

The only benchmark of the BLonD computational pipeline consisting of more than one kernel is interpolation-kick. However the first of its kernels is an auxiliary one setting the ground for the execution of the main workload which is the ultimate determinant of the component's performance accounting for more than 98% of the total execution time (Figure 5.3. Lulesh is by far the most complex benchmark consisting of 26 kernels. In order to draw a large enough sample from this application we will include its three most important kernels (named LUL1, LUL2, LUL3 in Figure 5.3). Those three are considered a good sample since in total they take up nearly half the applications execution time (48.77%) and the next largest kernel execution-time wise (called LULb in this context) takes 46% less cycles to complete compared to the fastest of the three selected ones.

After this preliminary analysis our study group consists of eighteen kernels presented in 5.4 along with their shortened names used in our analysis plots, simulated data sizes and launch grid and block size configurations with the two last ones being known - worst case scenario - at run time of the appli-

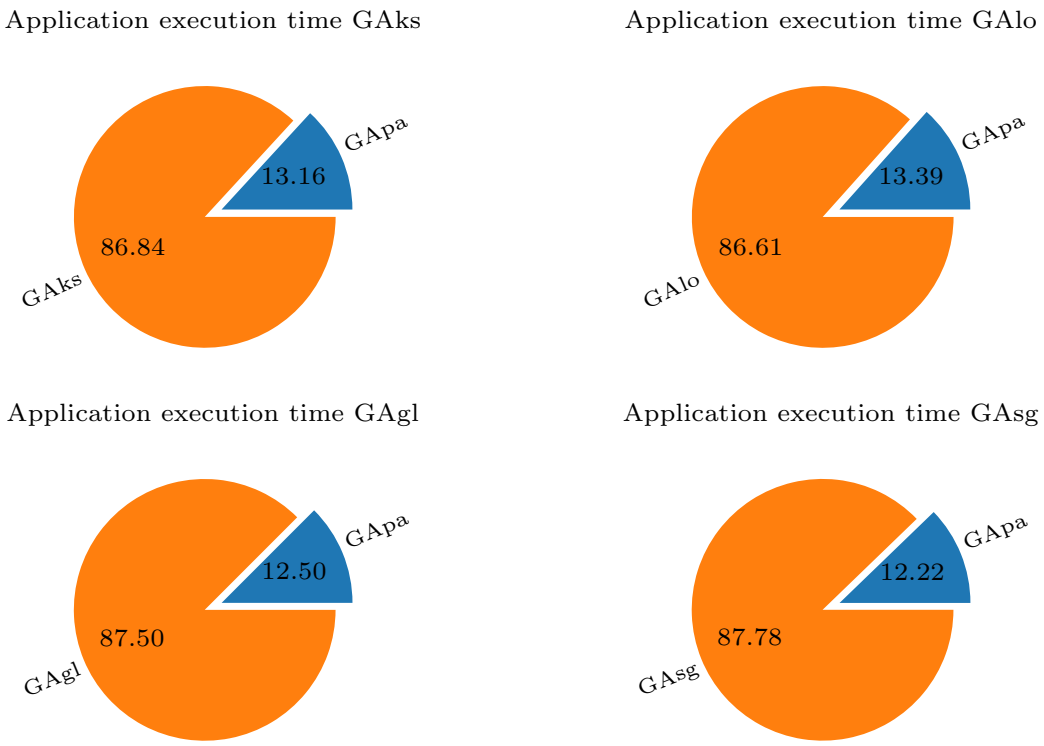


Figure 5.1: Execution time distribution, as measured on our baseline model, among main kernels of GASAL benchmarks. GASAL provides four benchmarks for its four supported alignment methods

cation. In the following subsections we will attempt to classify those eighteen kernels into two groups in order to tailor the two dissimilar SM types towards their specific needs.

5.2.4 Classifying Memory and TLP bound kernels

Having fully determined the test set, the first step we take is to inspect the instruction mixes of our kernels, signifying the percentage of PTX instructions that are requests to memory and the percentage executed on different types of computational units. The functional units that Accel-Sim [67] distinguishes from are integer units (INT) single precision units (SP), double precision units (DP), special function units (SFU) tensor cores (Tensor) and Load/Store Units (LD/ST). By executing the applications on the simulator we precisely measure the number of instructions executed on each of the aforementioned six types of functional units with the results being presented in Figure 5.4 providing us with a first insight as to how each kernel behaves

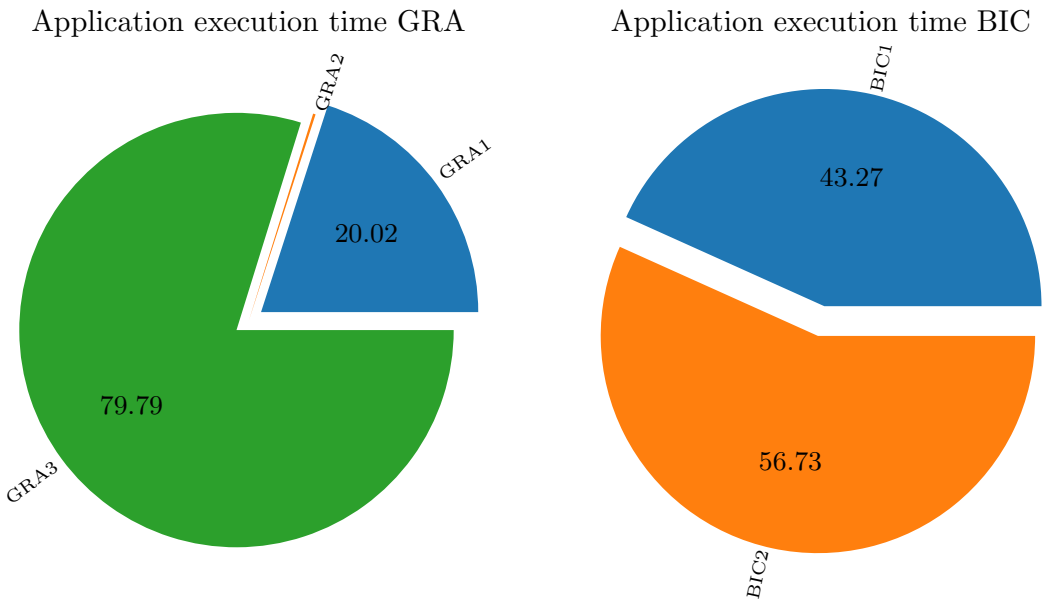


Figure 5.2: Execution time distribution, as measured on our baseline model, of the two linear algebra representative applications. GRA2 takes up negligible execution time

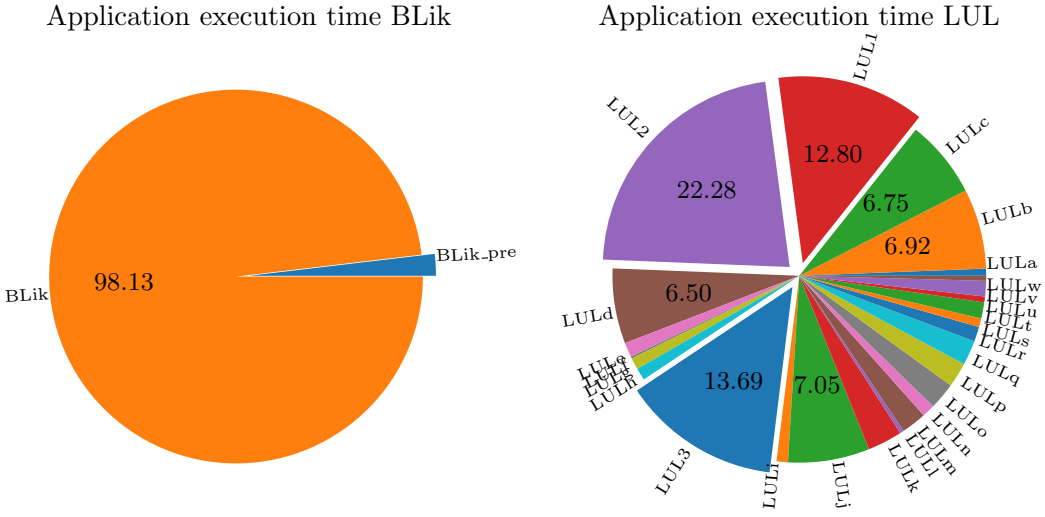


Figure 5.3: Execution time distribution, as measured on our baseline model, of interpolation-kick and lulesh benchmarks. Both of those benchmarks contain kernels taking up an insignificant portion of the total execution time.

and can be targeted for optimization. The instructions measured here are

Kernel Name	Abbreviation	Data size	Block Size	Grid Size
CalcHourglassControlForElems	LUL1	46 edge nodes	(256,1,1)	(2848,1,1)
CalcFBHourglassForceForElems	LUL2	46 edge nodes	(256,1,1)	(2848,1,1)
CalcKinematicsForElems	LUL3	46 edge nodes	(256,1,1)	(356,1,1)
convolution	BLco	(nsignal, nkernel)=(1000, 1000)	(64,1,1)	(512,1,1)
histogram	BLhi	nparticles=1000000	(1024,1,1)	(512,1,1)
drift	BLdr	nparticles=1000000	(64,1,1)	(512,1,1)
kick	BLki	nparticles=1000000	(64,1,1)	(512,1,1)
linear_interp_kick	BLik	nparticles=1000000	(1024,1,1)	(512,1,1)
calculate_temp	HOT	grid=(512,512)	(16,16,1)	(43,43,1)
bicg_kernel1	BIC1	grid=(1024,1024)	(256,1,1)	(4,1,1)
bicg_kernel2	BIC2	grid=(1024,1024)	(256,1,1)	(4,1,1)
gramschmidt_kernel1	GRA1	grid=(512,512)	(256,1,1)	(1,1,1)
gramschmidt_kernel3	GRA3	grid=(512,512)	(256,1,1)	(2,1,1)
gasal_pack	GApa	default query and target [3]	(128,1,1)	(40,1,1)
gasal_ksw	GAksw	default query and target [3]	(128,1,1)	(40,1,1)
gasal_local	GAlo	default query and target [3]	(128,1,1)	(40,1,1)
gasal_semi_global	GAsg	default query and target [3]	(128,1,1)	(40,1,1)
gasal_global	GAgl	default query and target [3]	(128,1,1)	(40,1,1)

Table 5.4: Test set of eighteen kernels spanning multiple applications and fields of scientific computation

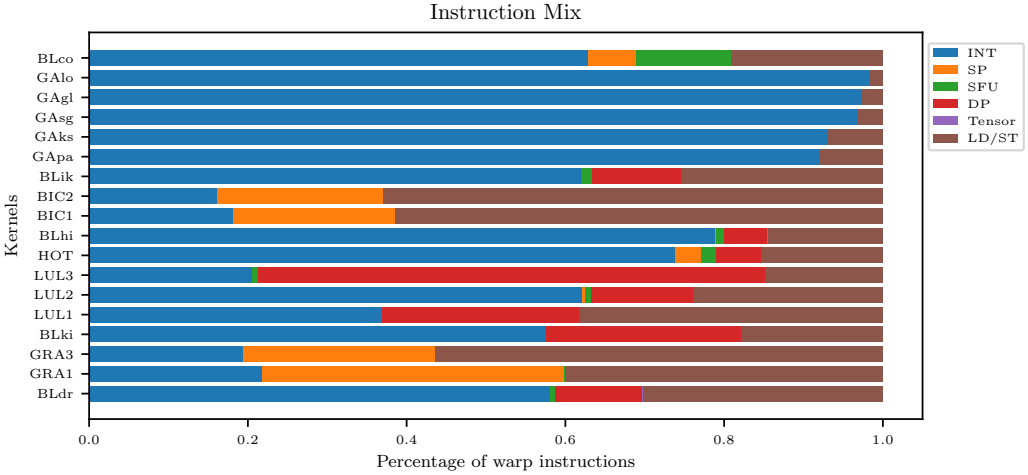


Figure 5.4: Instruction Mix of the eighteen studied HPC kernels

warp instructions, meaning that regardless of how many threads within the warp are active, when a warp is issued for execution the warp instruction counter for the specific type of instruction to be currently executed in lock-step is incremented by one. This provides more insight than measuring per active thread instructions since the warp will take up the entire execution unit regardless of how many of its threads are active. With regards to the instructions which stress any part of the memory system (instructions which are executed by the Load/Store unit) via either loads or stores three of the

eighteen kernels consist of at least 55% memory instructions and another five above 25%. The higher the portion of a kernel memory instructions take, the more likely this kernel is to be bound by memory limitations such as bandwidth or latency.

For the kernels who we previously determined are not memory request saturated (those featuring less than 55% memory instructions), it is evident that the dominant type of instruction is the integer-type instruction. This is understandable considering that all control related instructions, including loop index incrementing, branch instructions etc, are executed on the integer pipeline. The combination of those along with integer operations on the programs actual data path makes up for the majority of the program in almost all studied cases. Trailing behind are double precision operations. Taking the fact that the selected kernels belong to the HPC domain into consideration, it is expected kernels will offload large parts of their execution on the DP pipeline since scientific computing commonly involves high precision floating point arithmetics [11]. SP instructions are only significant in one non-memory-saturated instance and SFU instructions take up a negligible percentage of the total instructions while tensor ones are completely absent. Since they do not appear in any of the tested kernels they will be ignored in the rest of this thesis.

With this knowledge at hand we can better focus our efforts in identifying kernel bottlenecks in the vast architectural design space of the GPU through performance simulation data. We start by measuring for memory bottlenecks in order to confirm that kernels comprising large amounts of memory instructions are actually limited by the memory system and more specifically the off-SM one while at the same time uncovering such dependencies not made obvious by the instruction mix alone. We do so by utilizing the perfect memory configuration, same method we used in Section 1.2. This is a completely unrealistic approach implementation-wise as it almost entirely eliminates all memory limitations hence we set the bar high regarding the performance increase needed to justify a kernels classification as memory bound. Specifically kernels that improve with this configuration by at least 40% will be classified as moderately memory bound and those that double their performance will be deemed strongly memory bound. As we observe in Figure 5.5 BIC1 along both Gramschmidt kernels, BLdr plus BLco and LUL3 at least double their performance with this configuration and thus should be placed in the second category as they are not expected to improve with more compute resources. This observation is partially on par with our previous one regarding the instruction mix with exceptions being BIC2 which shows high percentage of memory instructions but no improvement with perfect memory and GRA1 and LUL3 which show the opposite behavior. These ex-

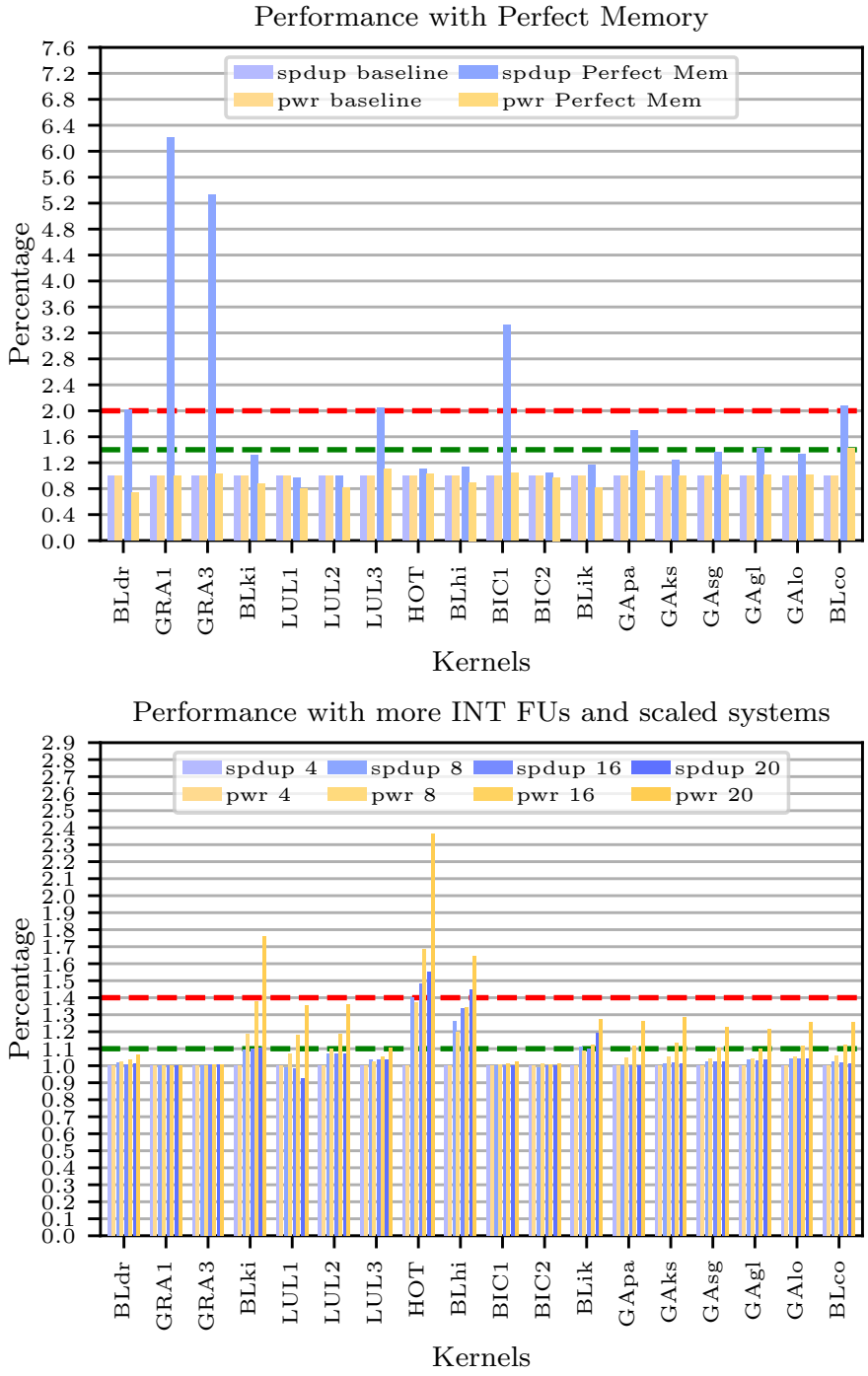


Figure 5.5: Kernel categorization benchmarks with perfect memory and with larger computational pipeline focused on integer operations. Green line represents what is considered weak improvement for each test while the red line represents major improvement

ceptions stress the need for performance runtime profiling. Meanwhile four of the five GASAL kernels and BLki exhibit weak improvement which does not allow us to make definitive decisions yet without further analysis. The rest will not be considered memory bound. Kernels experiencing memory related bottlenecks in GPUs do not utilize the execution pipeline efficiently and reverting this through micro-architectural modifications to the SM's inside is especially challenging even with increases in data cache capacities as we will see later in this section. Hence memory bound applications will be considered Low-Utilization unless another axis for improvement is identified through further analysis.

The default sub-core model of the V100 limits our degrees of freedom when it comes to modifications within the SM, since - at least in the simulator's context, requires each sub-partition of the SM to be complete, featuring all types of functional units the other do. More strictly it requires the functional units to be at least of the same quantity as the warp schedulers while adding more is poinless since they will not be used in the simulator. It also requires that each of them has access to identically sized portions of connected structures. This means that in this case when we scaled the warp schedulers alongside the integer function units we would have to scale every other function unit as well for each warp scheduler to have access to one. Similarly the collector units and register file banks would have to be divisible by the number of warp schedulers thus limiting the individual modifications we can benchmark. Those are only some of the limitations the model itself and its implementation in Accel-Sim enforce. For that reason sub-core model will be turned off when testing modifications which would introduce heterogeneity within the SM itself enabling each warp scheduler to access the entire pipeline. The implications of this change will be further discussed in Section 5.3.

As a next step we seek compute-hungry kernels. Since we have already determined that in almost all kernels the dominant instruction type is the integer operation we increase the integer functional units (INT FUs). Simultaneously we scale other compute related resources so as not to introduce new bottlenecks. Those resources span the execution front-end in the form of warp schedulers as well as parts of the back-end like collector units and register file banks all of which we scale by the same factor as the FUs. Specifically we increase INT FUs by four eight and sixteen with regard to the original four. We also increase the pipeline registers (in the simulator they link the instruction buffer to the operand collectors and the operand collectors to the execution units) as well as the write-back stage (which is also simulated by pipeline registers) to match those functional units. This way we introduce completely independent execution pipelines that do not share resources while

executing the instructions. Along with the FUs we scale the collector units and register file banks by the same factor so that the extra warp schedulers will not congest those structures. We employ these increases knowing that the biggest of them are most likely overkill as now in the process of categorization we want to be definitive when we decide the execution characteristics of a kernel. Figure 5.5 reveals that four of our kernels experience a noticeable performance increase when ran with this configuration. Since this modification is not nearly as aggressive as a memory subsystem that answers all requests in one cycle we consider a 40% and above speedup major while we set the minor limit at 10%. Kernels experiencing such performance increases with scaled execution pipelines are by definition Compute-Intensive. This barrier sets HOT and BLhi as strongly Compute-Intensive while BLki and BLik fall in the same category but weakly.

5.2.5 Classifying Kernels Based on Hardware-Independent Characteristics

We further investigate kernels whose performance analysis provided inconclusive results regarding their bottlenecks such as the GASAL kernels as well as kernels where the result fell far from the expected like the LUL3 whose performance doubled with perfect memory albeit it only consisting fifteen percent of memory instructions. This time we take a step back and emphasize on compile-time known characteristics which could shed light on some of our obscure performance analysis observations. We observe that the GASAL kernels as well as BIC2 have very small grids issuing between 4 and 40 blocks. Obviously these numbers, also taking into account the defined block size, lead to low occupancy and do not leave much room for parallelism to be exploited as well as limit the GPUs basic latency hiding techniques. Such kernels are not likely to improve their performance with additional hardware and are thus placed in the second category of Low-Utilization kernels.

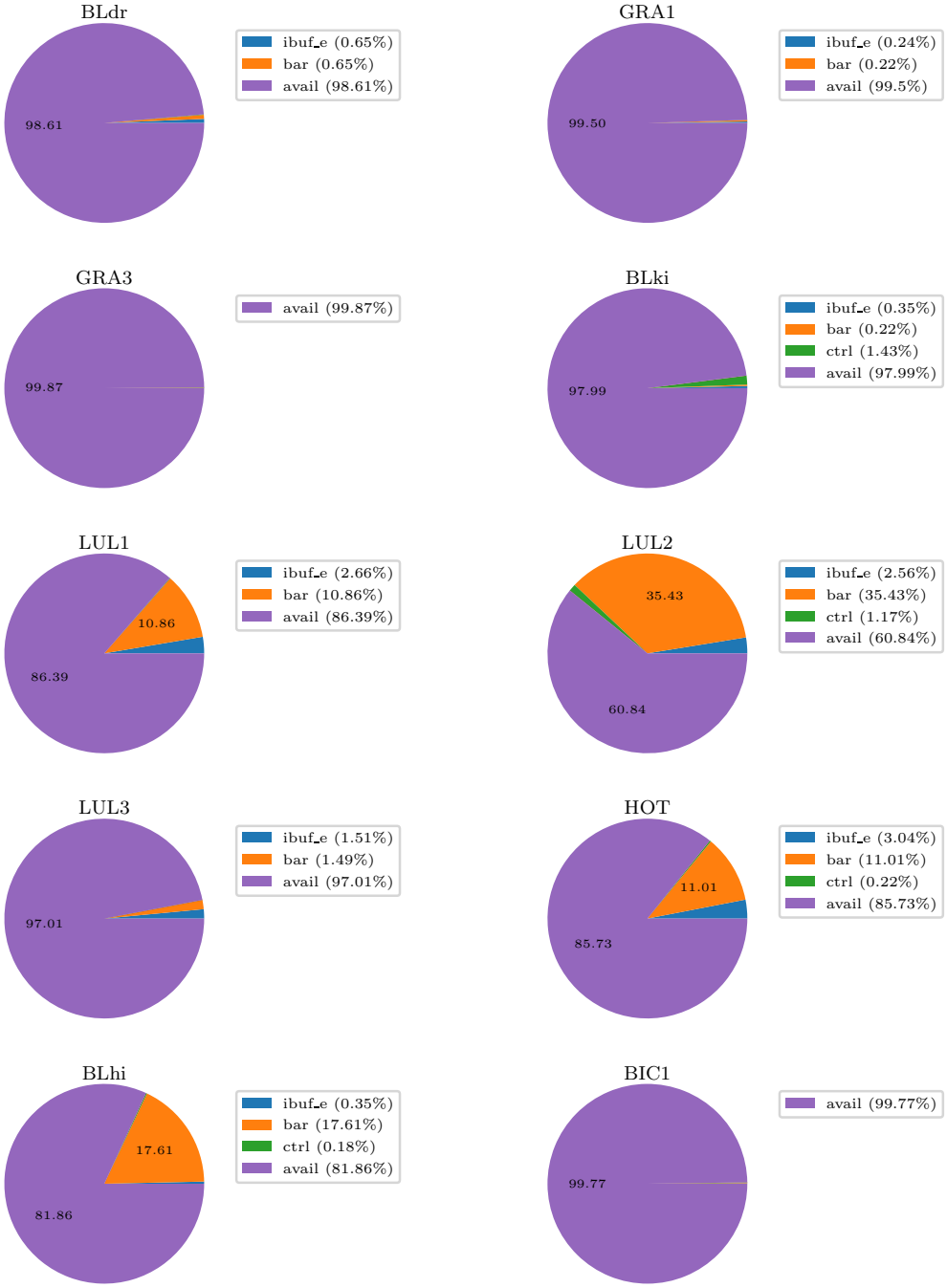
For LUL3 we observe that the calculated GPU occupancy of the kernel is extremely low at 12.5% despite the fact that the kernel features sufficient warps (determined by grid and block size displayed in Table 5.4) to reasonably fill out the SMs. This is caused by a very high register file usage (156 registers per thread) constraining each SM to house only a single block at a time out of the maximum of eight blocks of 256 threads it could due to the limit of 2048 threads per SM. In that case the SMs ability to hide long memory latencies is severely impacted by the lack of available warps to issue making otherwise negligible slowdowns imposed by waiting for few memory results much more important in the total execution time. Other previously

determined Compute-Heavy kernels also present with high register file usage sometimes limiting the occupancy as well (as is the case for HOT which can have seven blocks active within an SM at a time instead of eight). It is common for kernels featuring lots of computations to occupy a vast amount of registers as all those computation instructions could serve towards providing one end result with the register file serving as the housing for all the intermediately computed and later re-used values. For those benchmarks, providing a larger register file could improve their utilization and hence their performance, and considering how tightly coupled the register file is to the compute units via the operand collectors as well as the tendency of Compute-Intensive kernels to take up many registers per thread, LUL3 will be placed in the Compute-Intensive kernel category.

5.2.6 Classifying Inconclusive Kernels

Similarly to LUL3 and HOT, lulesh's other two kernels fail to fully occupy the SMs due to high register file usage however this does not impede their performance as is demonstrated later in Figure 5.16 where we see that although increasing their register to sizes which lead to the maximum possible occupancy (limited only by the maximum number of warps/SM) no speedup is observed - quite the contrary, their performance drops. One of the reasons for this lies in the way the program itself is written. Both kernels employ a plethora of barriers, imposing synchronization between threads of the same block. As a result fewer warps are available for scheduling at any given time since many of them are stuck waiting at barriers, reducing latency masking potential while imposing synchronization overheads. Making matters worse LUL2 kernel features these barriers as part of reduction operations. Reduction operations utilize progressively less threads of the block as they accumulate results, synchronizing with barriers between accumulation steps. This forces warps, regardless of the sequence of their execution into the aforementioned barriers leaving them inactive [105] while the progressive nature of reductions means that near its end almost all warps of a block will be inactive. Figures 5.6, 5.6 display the percentage of all warps residing within the SM available to the warp schedulers to consider for execution on average throughout the kernels runtime. We determine a warp as unavailable for scheduling when it is unable to issue the next instruction for reasons which would eliminate it from consideration before any data or structural hazards would be checked. This inability can be caused by an empty instruction buffer (denoted ibuf_e in Figures 5.6, 5.6), a warp idling on a barrier (denoted bar) and control hazards (ctrl) originating from divergence. Available warps (avail) are considered by the schedulers but could be stalled later be-

Warp Availability



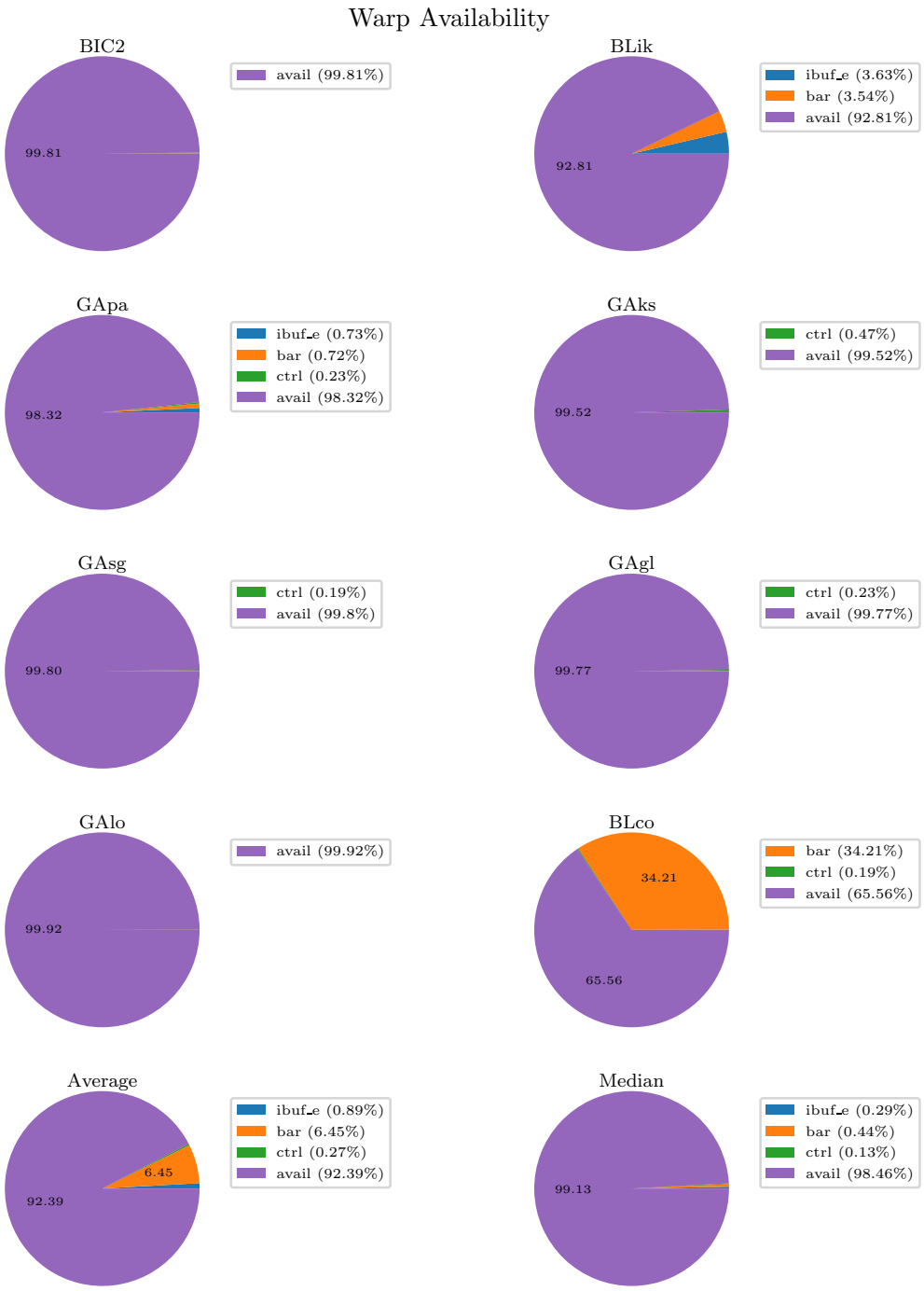
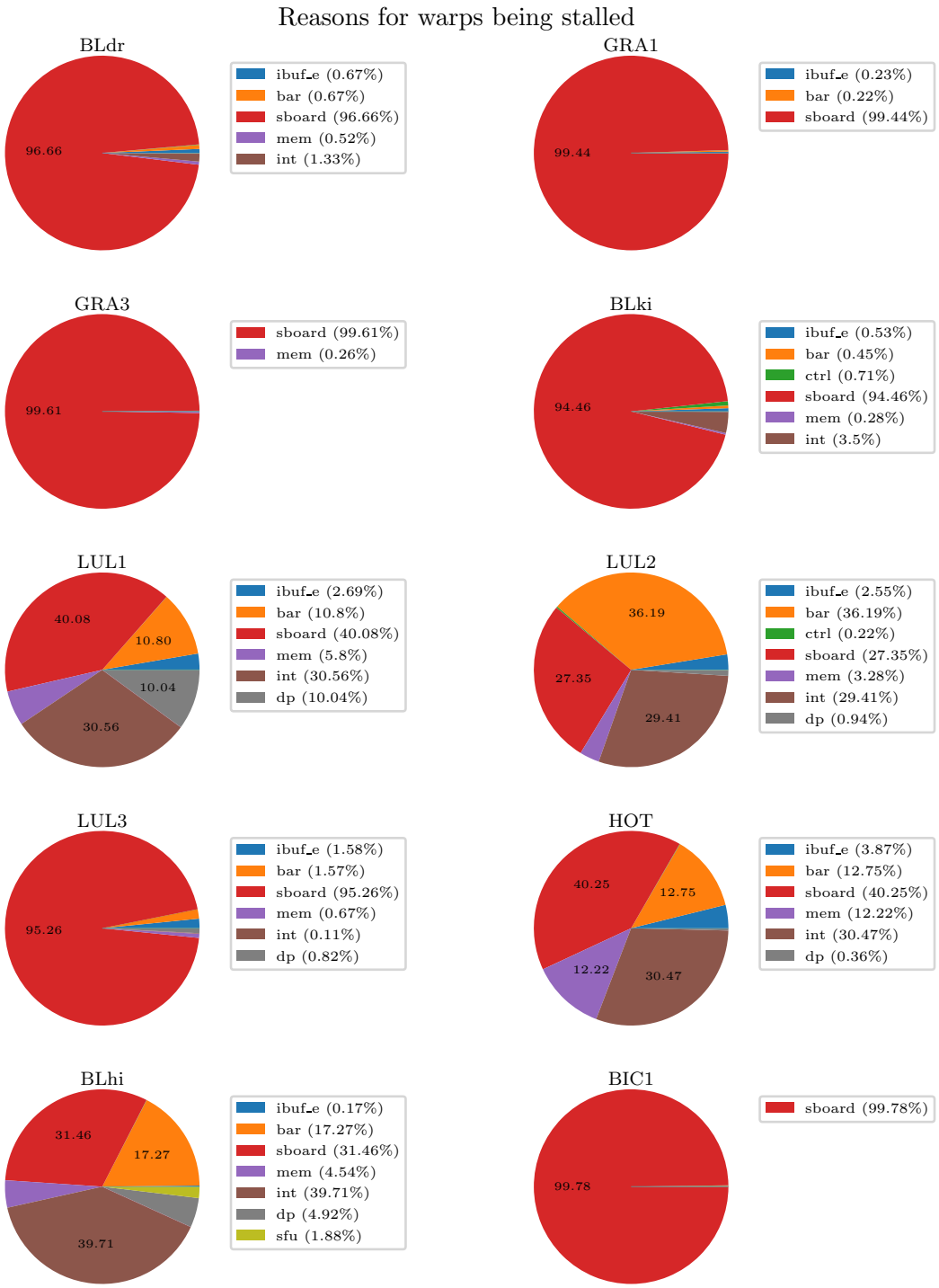


Figure 5.6: Average number of warps considered for scheduling throughout the kernel's execution

fore issuing in case of unresolved data dependencies or unavailable execution hardware. As it is observed in Figure 5.6 for LUL2 at any given time on average only 60.84% of its warps are available. Coupled with the fact that each SM only houses six warps instead of the maximum of eight due to high register file usage, the estimated runtime occupancy of the card at any time is lower than 46%. A close enough percentage of warps are stalled on barriers as well for BLco enhancing its already observed memory dependency. There is little room for improvement of kernels with few active warps caused by algorithmic requirements for barriers or small grids by simply modifying micro-architectural features of the SM without inflicting drastic changes in the execution model which is not in the scope of the current work. As such lulesh's second kernel will be named Low-Utilization as well. LUL1's 10.86% barrier-stalled warps, although high compared to the average and especially the median kernel, is on par with other kernels such as HOT and BLhi which we have already determined are compute bound. So although its lower warp availability surely limits its potential for performance gains it is not what inhibits its sensitivity to more compute resources.

In order to really be certain about the group we will place it in we take a further step and inspect the reasons for its stalls which are the throughput limiting factor. Specifically, again throughout the execution of a kernel, for all warp schedulers, each warp examined for issuing which failed to issue reports the reason why. When a warp scheduler cannot find an eligible warp to schedule and its cycle is wasted we collect the reasons causing each of the examined warps to be rejected by the warp scheduler. This could be any of the reasons making a warp unavailable and examined above (empty instruction buffer, barriers, control hazard), a data hazard represented by sboard in Figures 5.7, 5.7 or any structural hazard due to lack of execution resources. Structural hazards are separated by type and are the result of any part of the corresponding execution pipeline being filled up, from the operand collectors to the write back stage. It is expected that the main stall reason for a kernel will be scoreboard stalls when it cannot take advantage of additional hardware resources to execute more warps in parallel as it will all come down to data dependencies forced by the instructions of the kernel and their latencies themselves. For three of our Compute-Intensive kernels (HOT, BLik, BLhi) the percentage of stalls due to overworking the integer pipeline is greater than 30% inlining with our categorization however so is it for the lulesh kernel in question and nearly as much for LUL3. Figures 5.8, 5.8 reveal what happens to these stalls when quintupling the Integer functional units, collector units and register file banks but leaving the warp schedulers at the original four. We see that for LUL3 structural hazard stalls in the integer pipeline are practically eliminated. BLhi and HOT retain a higher percentage


Reasons for warps being stalled
BLdr
96.66
ibuf_e (0.67%)
bar (0.67%)
sboard (96.66%)
mem (0.52%)
int (1.33%)
GRA1
99.44
ibuf_e (0.23%)
bar (0.22%)
sboard (99.44%)
GRA3
99.61
sboard (99.61%)
mem (0.26%)
BLki
94.46
ibuf_e (0.53%)
bar (0.45%)
ctrl (0.71%)
sboard (94.46%)
mem (0.28%)
int (3.5%)
LUL1
40.08
10.80
10.04
30.56
ibuf_e (2.69%)
bar (10.8%)
sboard (40.08%)
mem (5.8%)
int (30.56%)
dp (10.04%)
LUL2
36.19
27.35
29.41
ibuf_e (2.55%)
bar (36.19%)
ctrl (0.22%)
sboard (27.35%)
mem (3.28%)
int (29.41%)
dp (0.94%)
LUL3
95.26
ibuf_e (1.58%)
bar (1.57%)
sboard (95.26%)
mem (0.67%)
int (0.11%)
dp (0.82%)
HOT
40.25
12.75
12.22
30.47
ibuf_e (3.87%)
bar (12.75%)
sboard (40.25%)
mem (12.22%)
int (30.47%)
dp (0.36%)
BLhi
31.46
17.27
39.71
ibuf_e (0.17%)
bar (17.27%)
sboard (31.46%)
mem (4.54%)
int (39.71%)
dp (4.92%)
sfu (1.88%)
BIC1
99.78
sboard (99.78%)

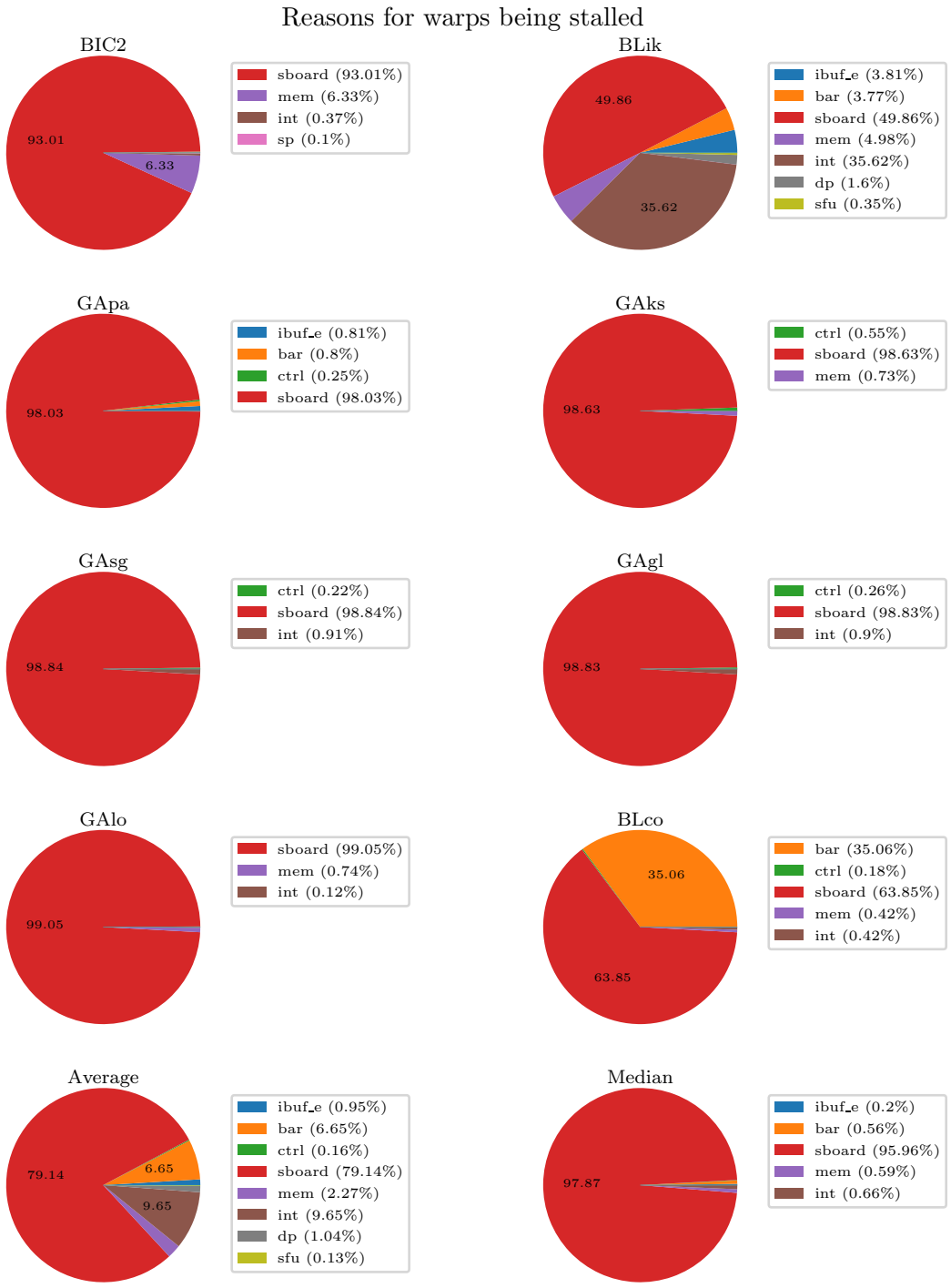
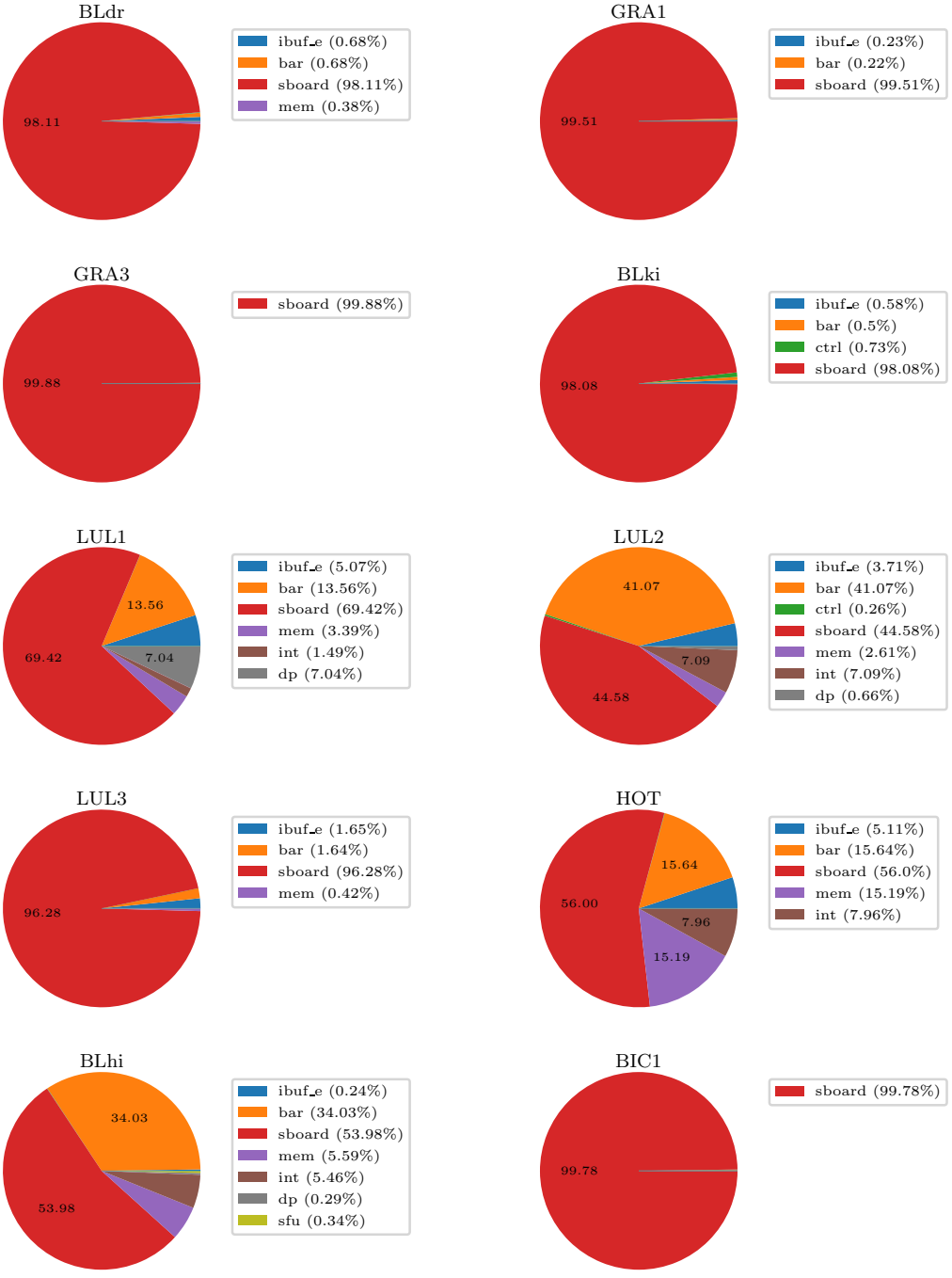


Figure 5.7: Reasons of warp stalls

Reasons for warps being stalled with larger Integer Pipelines



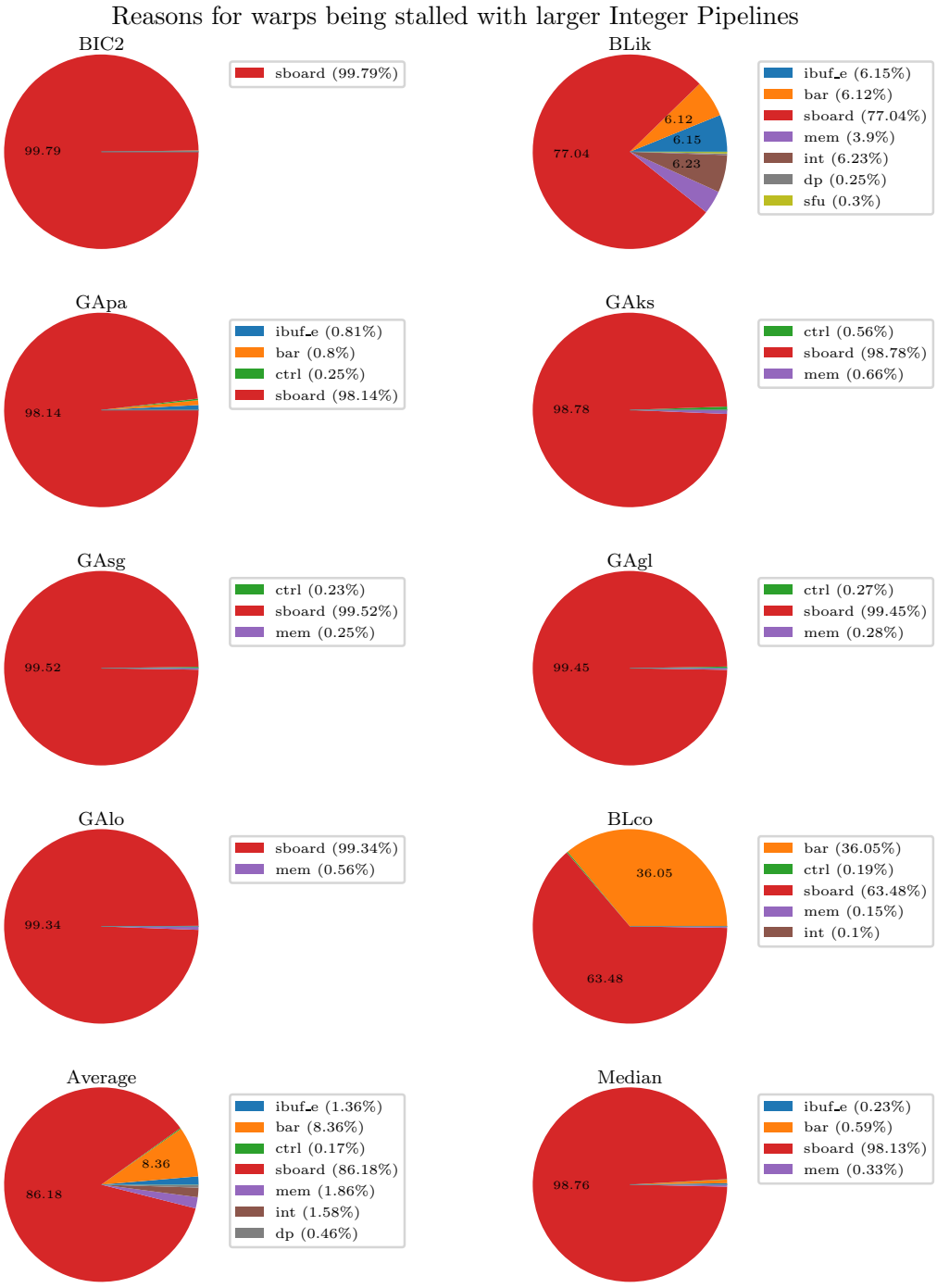


Figure 5.8: Reasons of warp with five times more Integer functional units, collector units and register file banks stalls

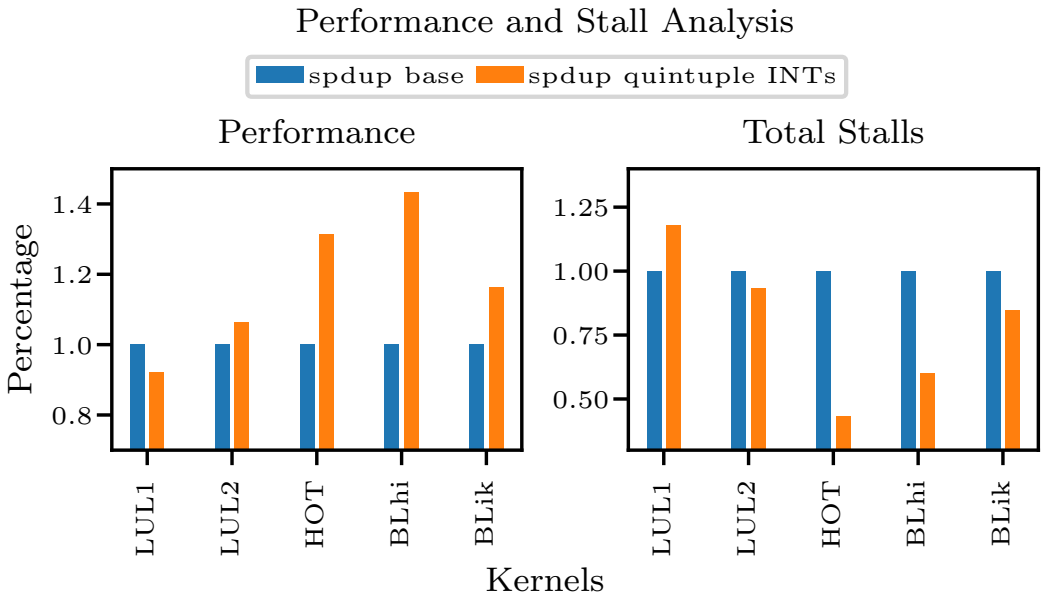


Figure 5.9: Performance and total stalls analysis of five kernels of interest. Resolving the structural stalls of LUL1, LUL2 offers little to no gain in performance.

however significantly lower than the baseline. The caveat for LUL1 is that although integer structural hazard stalls decrease, the total number of stalls does not even remain unaltered but is increased in contrast with the compute benchmarks for whom the integer stall reduction translates to a total stall reduction leading to better performance (Figure 5.9). Similar but not as bad behavior is observed for LUL2. This is indicative of the fact that for the two lulesh kernels are not really limited by the width of the integer pipeline and that the system's bottleneck is somewhere else, since when resolving these stalls they are balanced out mainly by scoreboard but barrier stalls as well. The latency of the various operations such as memory accesses to the various caches or even the double precision operations themselves cannot be decreased by modifying the micro-architectural configuration and requires more drastic modifications of the internals of those structures which is a separate field of work. Concluding this analysis LUL1 is also categorized as Low-Utilization.

At last we have found the most fitting category for each kernel in our test group with table 5.5 providing a brief summary of the category each kernel fell into. The kernel abbreviation names are used.

It is quite evident from the above analysis that many features contribute in the final execution characteristics of a kernel. However classification of ker-

Compute-Intensive	Low-Utilization		
HOT	BLdr	BIC2	GAlo
BLhi	BLco	GRA1	GAsg
BLik	LUL1	GRA3	GAgl
BLki	LUL2	GApa	
LUL3	BIC1	GAks	

Table 5.5: Final kernel categorization

nels by their developers is possible both through experience and knowledge which would allow a programmer to identify that a kernel with small dimensions or many reduction operations would most likely not be bottlenecked by the computational pipeline, and for cases where it is not as obvious, through profiling which can be done with advanced tools such as NVIDIA Nsight Compute [88] to reveal concealed intrinsic information regarding the lower level execution of machine code. Most importantly when the final model has been decided and the configurations for the two SM types are available, kernels can be directly tested on them to determine which one suits them best.

5.3 Configuration Study

5.3.1 Kernel Sensitivity Analysis

The next step is realizing how to treat each class in order to adjust the heterogeneous SMs. As we have already observed increasing INT FUs, collector units and register file banks improves performance in Compute-Intensive kernels. However it does so inefficiently power wise when done recklessly. We also have not explored other options such as the different types of functional units and the appropriate sizings. Likewise the Low-Utilization kernels leave SM resources completely untouched allowing them to be trimmed, conserving static power for non power-gated components as well as area without harming the performance. The aforementioned changes along others will be discussed and tested in this section in an attempt to engineer a crude yet representative heterogeneous GPU and showcasing its benefits in concurrent scientific kernel execution by GPUs.

- **Number of SMs:** A most important part of the analysis involves realizing the importance the number of compute cores plays in the performance of each kernel. Since each core type will end up featuring

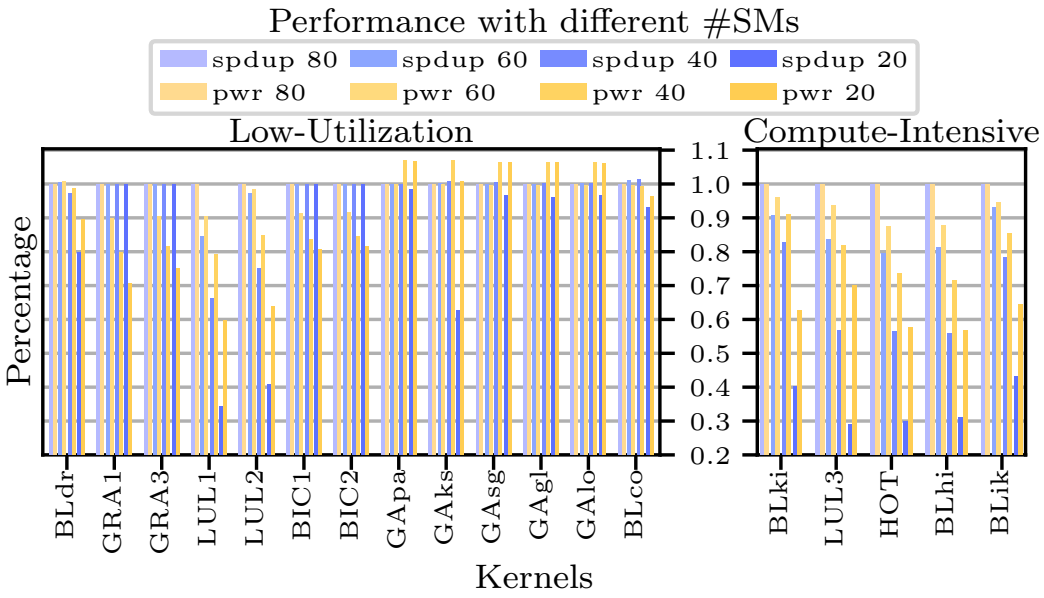


Figure 5.10: Kernel behavior with different number of homogeneous SMs

only a subset of the original 80SMs of the V100 it is crucial to study the performance impact limiting the parallelism at the highest level has on performance. As observed in Figure 5.10 Compute-Intensive tasks obviously suffer more from decreasing the SMs as they feature huge thread level parallelism and are not limited by outside-SM bottlenecks like memory bandwidth which would worsen with pressure from more SMs. As for Low-Utilization benchmarks many of them who were limited by memory bandwidth (e.g. BLdr or BLco) are quite insensitive to the number of SMs while others who failed to fill up all the SMs to the maximum obviously are not really affected. Most importantly we observe sub-linear performance decrease against the reduction of computing cores for all benchmarks regardless of their type. This is crucial in supporting our concept as the increasing number of SMs appears to reach a point of diminishing returns for a single kernel indicating that the GPU could be better utilized by being split and assigned the execution of multiple kernels in subsets of the original computational powerhouses.

- **sub-core Model:** As already discussed differentiating the quantities of components within the SM would inevitably result in non-homogeneous sub-cores by NVIDIA definition. Since this is not allowed in the simulator and the setting has to be switched off we are obligated to study how this change affects the results in order to have a more clear under-

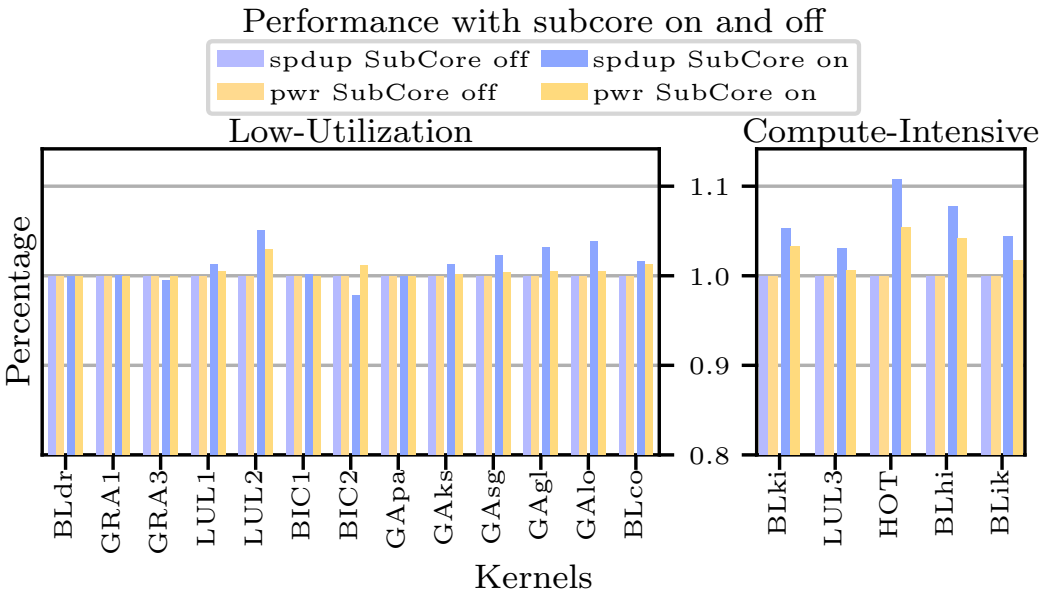


Figure 5.11: Kernel behavior with enabled(baseline) and disabled sub-core model

standing of the results which follow. Disabling this core configuration means that all warp schedulers will now share the entirety of structures like the operand collector and execution pipelines. This can lead to better resource utilization by avoiding certain structural hazard stalls and bank conflicts but requires a higher level of interconnection resulting to increases in area and power. In figure 5.11 we can see that as expected the Compute-Intensive benchmarks are the main beneficiaries from this change improving by an average of 6.3% while the mean performance increase for the Low-Utilization class is 1.3% with a maximum increase of 5.1% for LUL2. In all cases save two, which experience performance deterioration without scheduler isolation, the spike in power caused by the higher utilization of SM components due to this change is smaller than the increase in performance making this necessary change for our configuration, bearable as well.

- **Integer functional units:** We have already looked at what kernels are benefited by expanding the Integer Pipeline. Studying Figure 5.5 under a different prism we observe that there is a point of diminishing returns around the tripling of the integer functional units where the increase in power outweighs the speedup which increases minimally with more execution units. Although the functional units themselves consume power the main culprit for this jump are the register file banks which

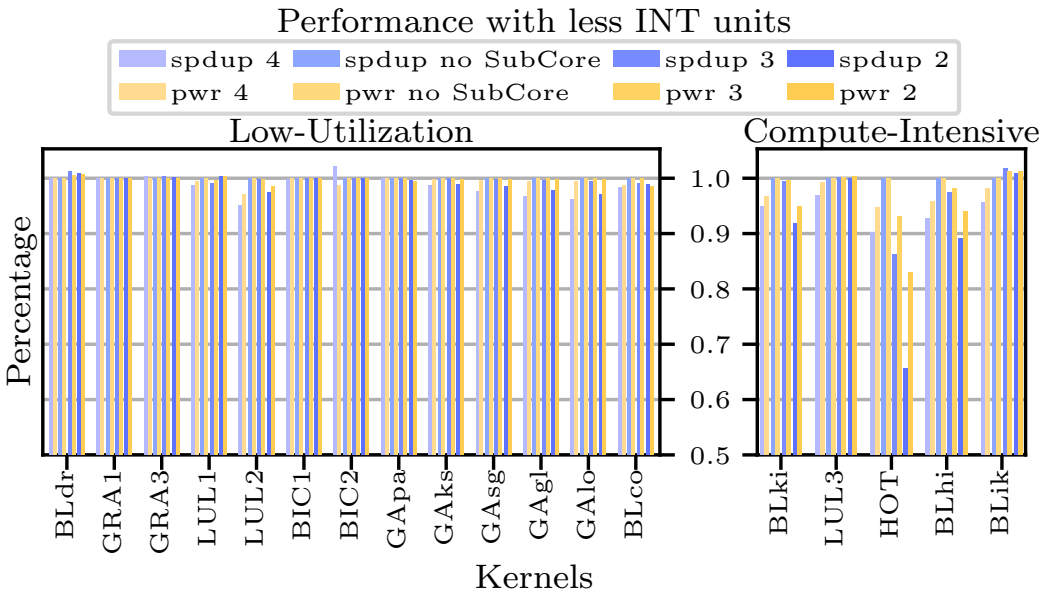


Figure 5.12: Kernel behavior with fewer Integer execution units

we scaled as much as the functional units to avoid potential bottlenecks in this configuration. This will be confirmed in a following separate analysis. The main takeaway from this analysis is that by definition Compute-Intensive benchmarks improve with this scaling and caution should be taken regarding the increase we employ.

Now we have to also examine the effects of reducing them since in the less powerful, Low-Utilization specialised cores we will attempt to trim compute resources. Figure 5.12 displays those effects when testing our kernels on two new configurations with three and two Integer Units respectively. The reduction of the function units is the only implemented change here beside the necessary turning off of the sub-core model - the rest of the pipeline is not altered contrary to when incrementing them as we are trying to introduce a single bottleneck and observe its importance. Since switching off the sub-core model incurs performance changes (usually a boost) in simulated kernels a second set of bars next to the baseline presents the performance impact already seen in Figure 5.11 for a clearer comparison. It is adamant that reducing them even by half does not hurt the performance of the Low-Utilization kernels however three of our Compute-Intensive kernels show significant deterioration for such a reduction with HOT which is the most affected one suffering a slowdown of 27.2% compared to the baseline and 34.3% compared to the no sub-core model. The notable exception to the rule

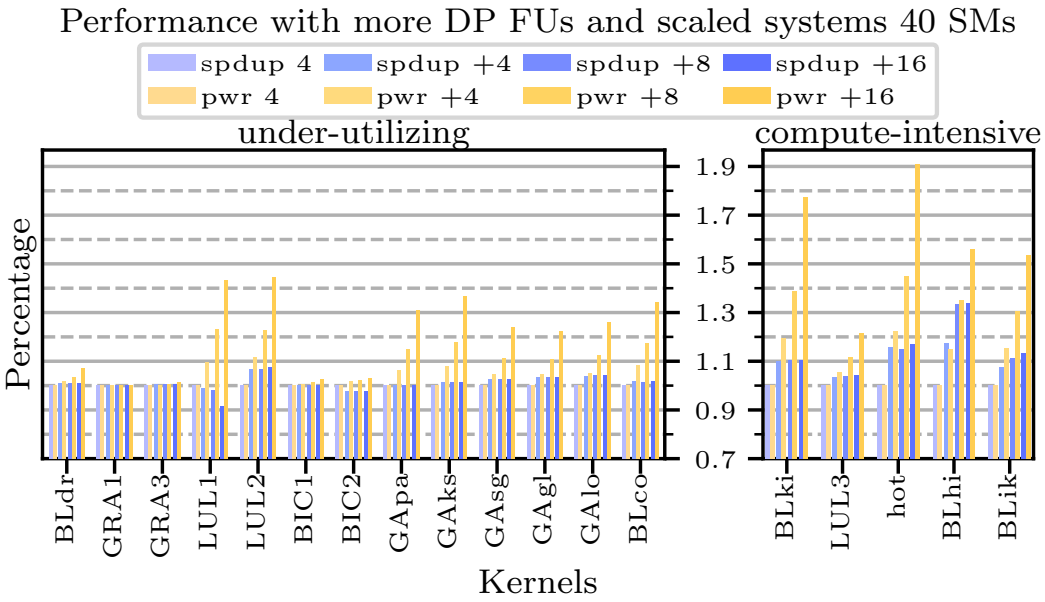


Figure 5.13: Kernel behavior with larger computational pipeline focused on double precision operations

is BLik whose performance increases with less Integer execution units available. This contradicts the performance increases we noticed when scaling the pipeline and may appear unreasonable at first glance however a higher degree of concurrency in Integer or any type of instruction execution can increase contention for shared resources such as the register file and hurt performance that way. When exploring scaling the entire pipeline this does not pose a problem.

- **DP functional units:** Being the second most used operation it makes sense to explore adding more DP FUs next to the INT to the cores customized to handle the Compute-Intensive workloads. We configure the cores with more DP units in the same manner we configured them with more INT i.e. scaling other systems as well and disabling sub-core model. As shown in Figure 5.13 Compute-Intensive kernels do benefit from such an increase albeit this increase is smaller than that with INT FUs and can largely be attributed to the rest of the pipeline being scaled since DP instructions only make up for 6% and 7% of the instructions of the two most improved kernels, HOT and BLhi respectively. Although DP units do contribute in kernels like BLhi and BLik facing noticeable yet few structural hazards in the DP pipeline as observed in Figures 5.7 5.7, it is evident that more collector units and/or register file banks are what unlocks the most performance ben-

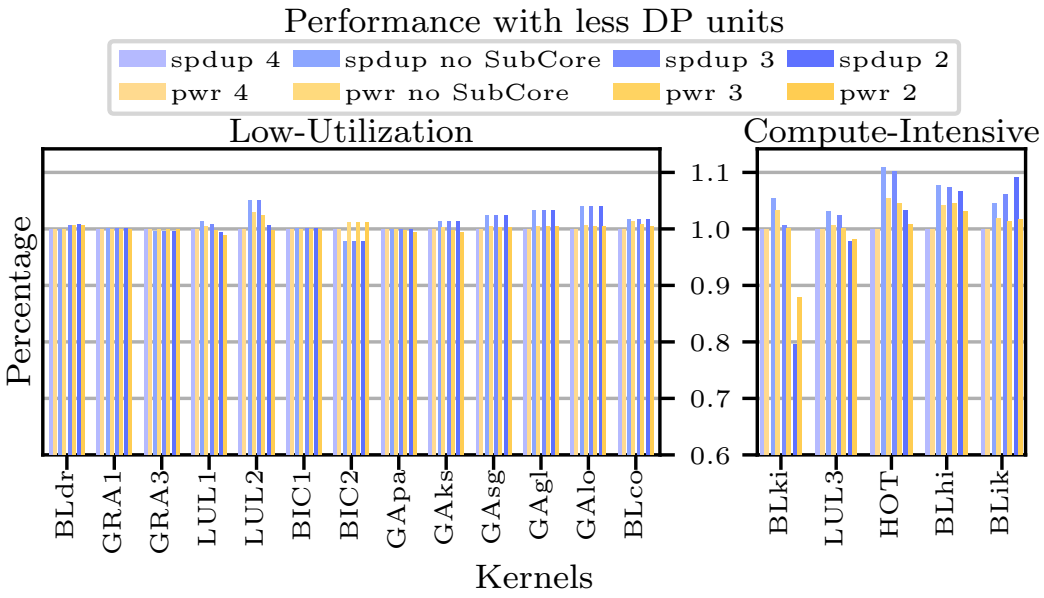


Figure 5.14: Kernel behavior with fewer double precision execution units

efits for the compute class of kernels in this case. Points discussed about power increase and detectable points of diminishing returns hold true for DP scaling as well although here the point of no improvement appears earlier than where it appeared when examining integer scaling, reasonably so taking into account the amount of instructions of each type and the structural stalls caused by each as examined earlier. As expected Low-Utilization kernels remain unfazed with their most improved kernel being LUL2 showing a 7.6% speedup with the largest tested increase, highest portion of which can be attributed to the disablement of sub-core configuration which increased its performance by 5.1% based on our previous observations.

We also test how the reduction of those execution units affects our kernels in order to make a clearer decision on how much we will reduce them in the second type of cores, those executing Low-Utilization kernels. We do not expect any significant performance deterioration in the target kernels as they do not fully utilize the compute pipeline anyways and only the lulesh kernels among them experience any stalls due to filled up DP pipelines when executed on our baseline configuration (Figures 5.7 5.7) and for those we have already determined that the bottleneck does not reside in the width of the execution pipeline. Figure 5.14 visualizes the performance impact of reducing only the functional units. Again taking into consideration the effects of not im-

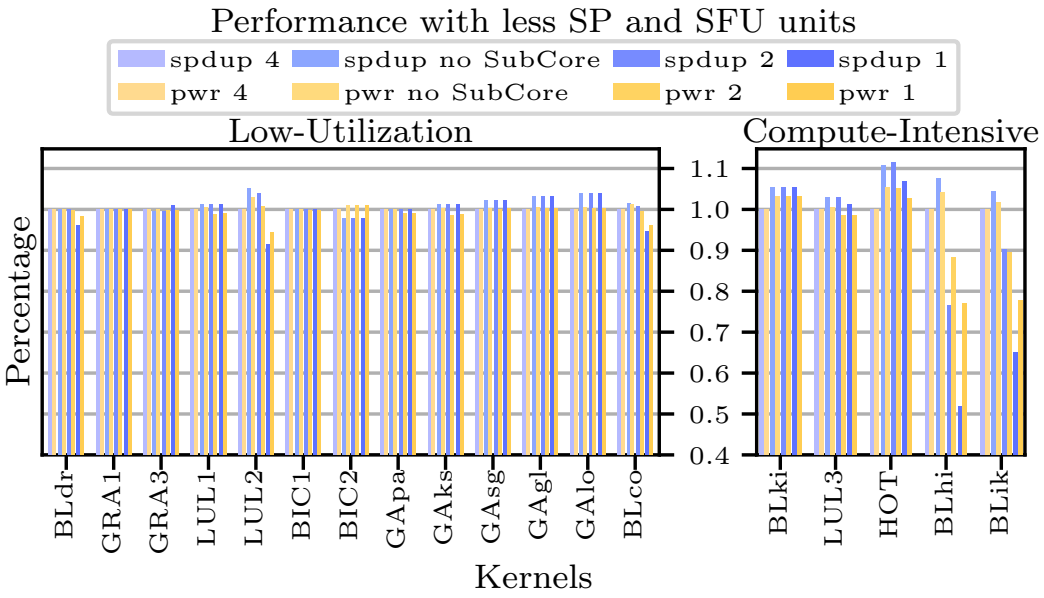


Figure 5.15: Kernel behavior with fewer single precision and special function execution units

plementing the sub-core model represented by the second set of bars in this figure, reducing the double precision functional units by one does not deteriorate the performance of any kernel while halving them down to two only worsens LUL2's performance and only slightly. On the other hand four out of the five Compute-Intensive kernels experience some degree of slowdown when executed on trimmed hardware with the exception once again of BLik whose performance increases the less DP execution units it has available. The explanation for this behaviour when reducing Integer functional Units holds true for DP Units as well.

- **SP/SFU functional units:** The single precision and special functions (such as trigonomteric functions) are the two least encountered types of instructions in our kernels (omitting the non-existent tensor operations) and as Figures 5.7 5.7 show, only cause structural hazard stalls in one of our Low-Utilization kernels when benchmarked with the baseline configuration. Reducing them and only them from the starting four down to even one yields an average 0.5% and a maximum 8.6% performance hit in Low-Utilization kernels compared to the baseline V100 and 2.7% and 13% respectively compared to the no sub-core model (Figure 5.15) for reducing the amount of two types of execution units by 75%. Compute applications as expected suffer more from this

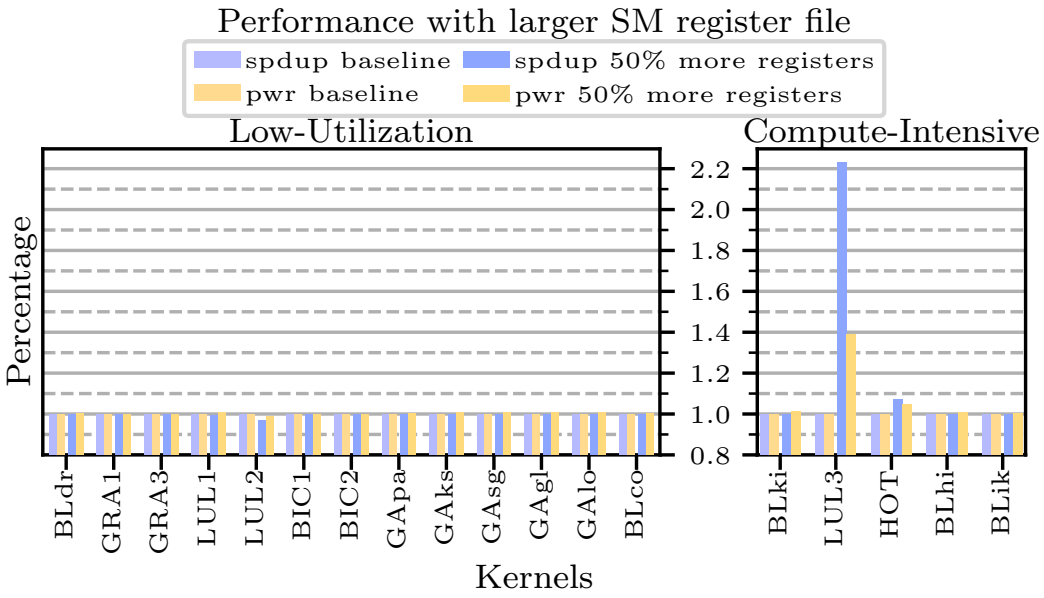


Figure 5.16: Kernel behavior with larger Register File

reduction as well, with the most drastic decrease nearly halving the performance of BLhi compared to the V100 and reducing overall performance against the no sub-core model by 19% on average and 51.8% maximum. Since no applications experienced any significant structural hazard stalls due to lack of space on any of the specific two pipelines discussed here, we will not study the effects of increasing them.

- **LD/ST units:** Accel-Sim [67] simulates the memory pipeline stage through a single load/store providing no flexibility for testing configurations with more/less such units.

- **Register File Size:** As mentioned in subsection 5.2.5 some of the studied kernels in both groups despite having enough warps based on their configuration they do not assign them to an SM since there do not exist enough registers to accommodate all of the threads' data until their execution is complete. NVIDIA GPUs employ register spilling only at compile time, through specific instructions which do just that, without knowledge of the hardware the kernel will execute on and primarily focusing on the performance impact those spills will have, as evicted registers are placed in local memory - a logical memory space residing in the high latency global device memory and backed by the main data caches [106, 89]. It is evident why a spillage transforming a register read latency to a global memory read latency has important

drawbacks and performing such decisions at runtime could result in highly unpredictable performance hits. That is why the decision on the number of used registers per thread is finalized at compile time and the GPU will restrict the active threads of an SM at runtime if it cannot comply with the requested amount rather than trying to pack as many warps as the computational system can fit and take steps back by offloading register data to memory when the register file cannot keep up with the demand. Increasing the Register File size is the only way for kernels to increase occupancy when the limiting factor is high registers/warps ratio. Figure 5.16 shows that doing so by 50% massively benefits the performance of the most restricted kernel in our test set, LUL3 which was limited to one warp per SM at a time instead of the maximum of eight it could house before hitting the maximum threads per SM limit of 2048 while also improving the performance of the HOT kernel which achieved seven out of the maximum eight warps per SM. All of the kernels categorized as Low-Utilization do not improve with larger register file. This is readily apparent for most of them as given their nature and low amount of computations they were well within the original Register File's limitations however for kernels such as LUL1 and LUL2 who faced the aforementioned limitation the increased occupancy resulting from that change did not incur performance increase indicating that higher occupancy does not always imply higher performance. All things considered amplifying the register file size is a change better suited for Compute-Intensive Kernels.

- **Register File banks.** Loaded compute units are the result of warp schedulers issuing many compute instructions first to the collector units whose job is to probe the Register File and retrieve the necessary values for the instruction to operate upon. Although their scheduling mechanism minimizes collisions, when the workloads are heavy enough it is inevitable that two requests will end up accessing the same bank in which case they have to be serialized and served in multiple cycles. This is why we study increasing those banks however the results are disappointing with only minuscule performance increases but a large spike in power consumption. Only HOT achieves a 6.2% performance increase when doubling the register file banks and the speedup of all the other kernels is less than 3% for all configurations. As such adding a high number amount of RF banks should not be considered unless required to balance out increased warp level parallelism when scaling the pipeline as we did in the configurations with more Integer and DP function units. Even then it should be done conservatively as we observe

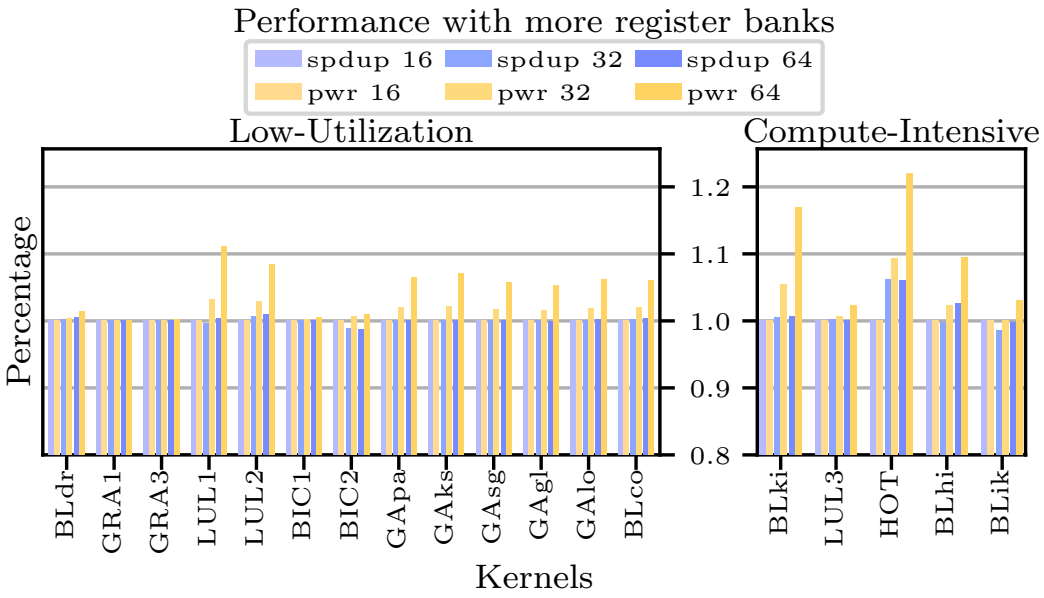


Figure 5.17: Kernel behavior with more Register File banks

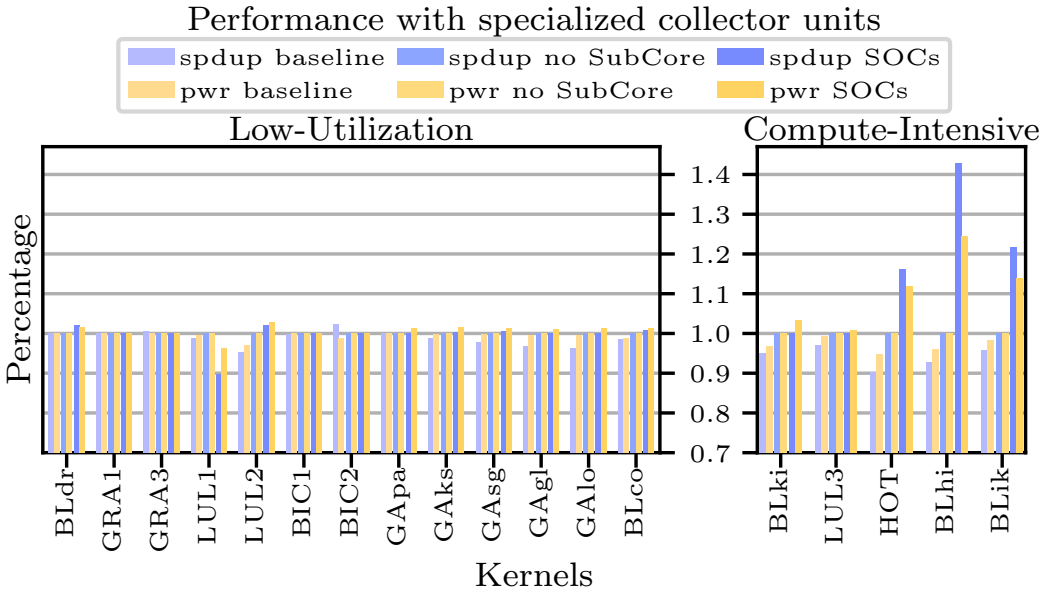


Figure 5.18: Kernel behavior with eight general purpose eight-port collector units vs with adjusted analogies of specialized collector units

diminishing returns in power consumption when more than doubling them.

- **Specialized Operand Collectors:** The V100 contains eight gener-

alized collector units inside an operand collector with eight input and output ports. The importance of the operand collector in organizing register file accesses and simultaneously fetching multiple operands by enhancing bank level parallelism has already been stressed in Chapter 2. Once all operands of an instruction have been fetched it can be dispatched for execution. Since we observed in our benchmarks that the most widely used instructions are those executed in the Integer and secondarily in the DP Pipeline while Single Precision floating point and Special Function Unit operations are scarce it would be reasonable to use specialized collector units (CUs) providing more space and bank parallelism potential to the most used instructions while constraining less frequent ones to keep the size of the entire Operand Collector reasonable. This way more instructions will be able to concurrently collect operands and issue for execution in parallel and the total waiting time in the Operand Collector stage will be decreased. This change is expected to assist kernels with high warp level parallelism who require a wide pipeline and concurrent accesses to fetch multiple operands. In the tested configuration the general purpose CUs were replaced by sixteen collector units dedicated to integer operations featuring four input and four output ports, eight single ported memory instruction collector units, four single ported (one input, one output) single precision ones, another four single ported SFU ones, and six two-input two-output ported double precision collector units In order to make this change possible we once again had to disable the sub-core model of the GPU as warp schedulers would end up with uneven collector units within each sub-core. Figure 5.18 depicts that the performance increase is significant in the three kernels categorized as the strongest Compute-Intensive ones by our analysis so far (36.5% average, 53.8% maximum compared to baseline and 26.8%, 42.7% respectively compared to baseline with sub-core support) while the weaker improvements observed in the other two of this group are almost entirely attributed to fully interconnected SM pipelines. Minuscule performance increases are also present in some Low-Utilization kernels not enough however to justify the increase in hardware which will stem from the increase in the crossbar necessary to interconnect the operand collector ports with the register file and execution units.

- **Warp schedulers:** We have already seen in the study of Integer and Double Precision functional Units that increasing the amount of warp schedulers contributes to increased warp level parallelism through a bigger issue width. However not only the quantity but also the type

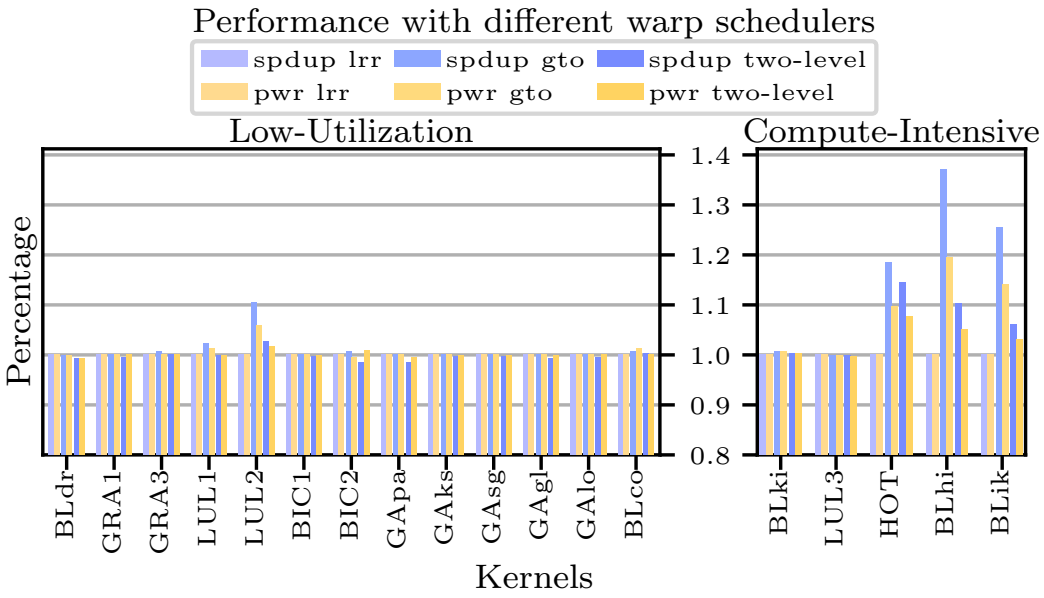


Figure 5.19: Kernel behavior between Loose Round Robin (LRR) warp scheduler, Greedy then Oldest (GTO) and two-level

of warp scheduler plays a role. The V100 GPU by default uses Loose Round Robin (LRR) warp schedulers. This scheduler cycles through warps in a round robin fashion skipping warps ineligible to issue instructions for execution. This scheme offers fairness and aims to provide even execution times for all warps. Another studied warp scheduler is the Greedy Then Oldest (GTO) warp scheduler which greedily prioritizes the same warp for execution up to the point of a stall when it then switches to the oldest one based on core assignment time and thread's id. The purpose of such a policy is to allow warps more exclusive access to cache structures reducing thrashing and evictions and increasing locality [113]. Additionally it may allow for better performance by avoiding structural hazards occurring when using its LRR counterpart, whose purpose of equal progress among warps can lead to consecutively scheduled warps to be issuing the same instruction promoting contention for the appropriate execution units [76]. A two-level [82] scheduler is also tested where warps are split into two groups. The first group contains warps stalled on instructions with estimated long latency such as memory fetches while the other one should contain warps available to execute. This scheduling scheme functions by scheduling in LRR fashion inside the group of active warps, demoting any encountered warp waiting on a long latency instruction, while the

first group replaces demoted warps in the active group. The purpose of this scheduler is to enforce the overlap of memory instruction and computation instruction execution [122]. The tested two level scheduler comprises two groups of eight warps each making up for the total of sixteen warps each of the four schedulers cycles through for the total of 64 warps per SM. As Figure 5.19 depicts three of the five selected Compute-Intensive kernels reap the benefits of the GTO scheduler improving by 18.4%, 25.5%, 37.1% ranked from lowest to highest when it used in place of the baseline's LRR one while only LUL2 improves noteably by 10.4% with it from the Low-Utilization group. The two-level scheduler improves the same kernels the GTO one does however by less. This can be attributed to the fact that this scheduler can imitate a greedy policy by cycling through the same limited warps in case they do not stall for long latency memory requests. Nevertheless its innermost group is still scheduled in LRR fashion making it a middle ground solution. It is also easier for kernels featuring many lower latency computations (as our Compute-Intensive kernels tend to do) to provide available warps for the inner-most level of scheduling to mask the stalled ones on the other group, where on the other hand kernels with few such warps cannot efficiently use this mechanisms, in some cases even taking slight performance hits as we see with some of the GASAL kernels.

- **L1 Data Cache:** The size of the L1 cache is another possible reconfiguration axis. The V100 features a 32KB pure data cache and 96KB of shared memory which are unified. This means that when data is not cached in the shared memory explicitly by the programmer its redundant space is used as L1 cache. We also try a configuration where this adaptive cache mechanism is turned off limiting the SM to the original 32KB of L1 Data cache and gradually scale it up to 192KB by changing its associativity appropriately. In Figure 5.20 we observe little to no sensitivity in almost all our benchmarks when the L1 is not pushed to extremely low sizes while increases hardly benefit any kernel. This can be explained by the fact that GPUs work on the basis of massive multithreading leading to thousands of threads accessing the L1 cache at a certain time frame. As a result the cache size per thread is very limited making it hard for warps to exploit data locality when accesses from different threads cause evictions and thrashing. This cache is better suited towards serving smaller footprint operation like register spilling and increasing throughput of irregular kernels featuring potentially non coalesced accesses [66, 113].

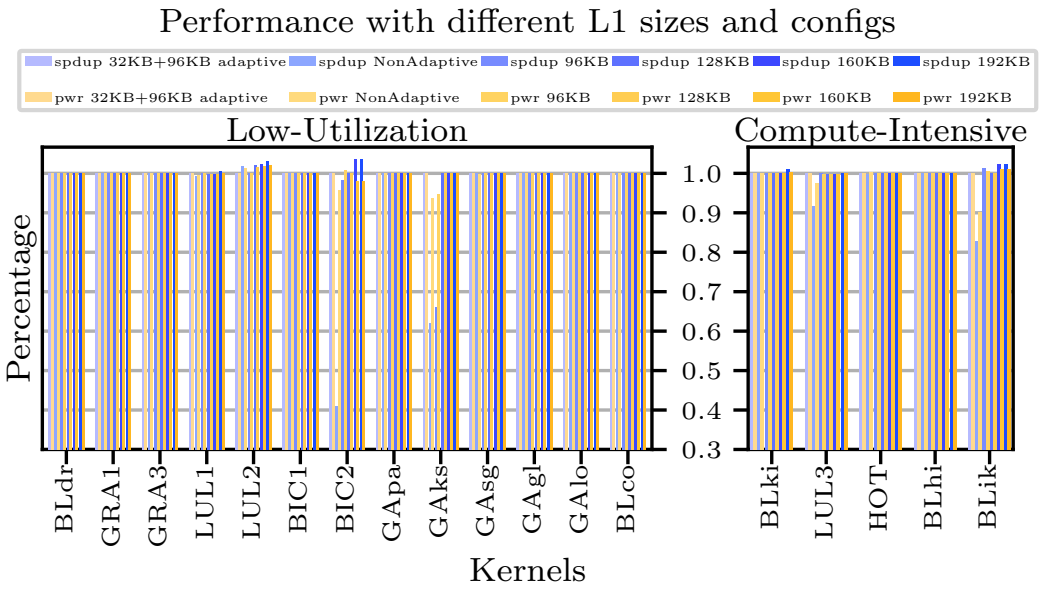


Figure 5.20: Kernel behavior with unified L1 and shared memory as well as with exclusive L1 of variable sizes

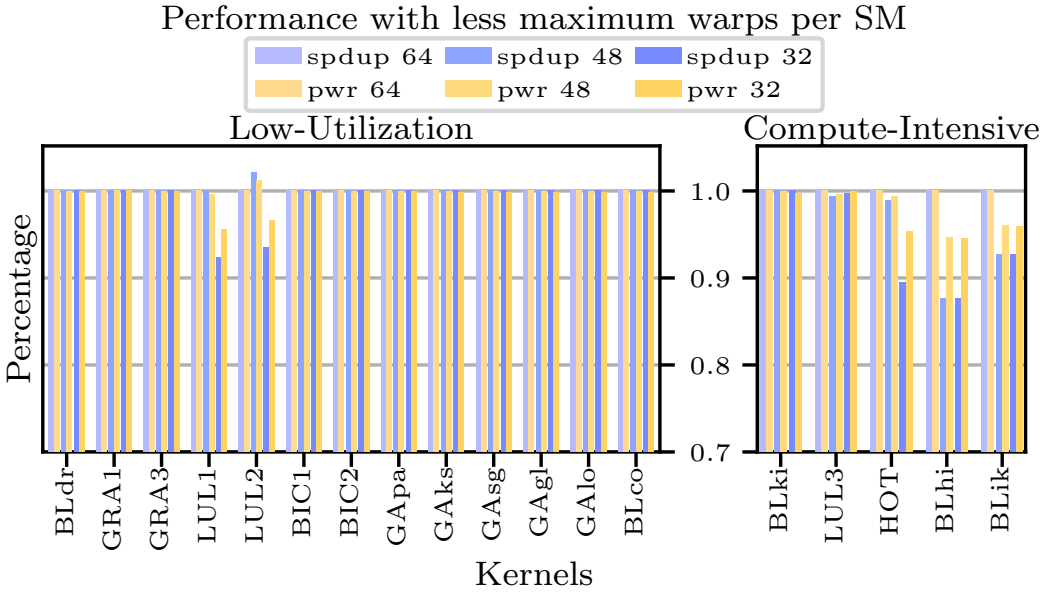


Figure 5.21: Kernel behavior with different maximum number of concurrently running warps within the SM

- **Maximum number of warps per SM:** As we have already seen in subsection 5.2.6 a low number of available warps in the SM can hurt performance by reducing available parallelism and inhibiting latency

hiding. However as we observed some benchmarks which did not fully occupy the SM due to register file usage did not benefit when the latter was made large enough to allow them to and literature suggests that sometimes limiting the maximum allowed number of active threads per SM can even increase performance by decreasing contention for shared resources such as the L1 cache examined above [113, 65]. Also decreasing the maximum number of warps per SM will decrease the size of front-end structures like the instruction buffers and warp schedulers. V100 allows for up to 64 active warps per SM which we try to decrease to 48 and 32 with the results appearing in Figure 5.21. All Low-Utilization benchmarks do not suffer any performance losses with the most mild decrease and only two of them drop by nearly 9% in performance with half the number of maximum warps. This is to be expected as many of them did not fill up the SMs in the first place as already observed and those who did, did not exploit high thread level parallelism. On the other hand Compute-Intensive benchmarks suffer more direct performance hits even with 16 less maximum active warps and limiting their TLP appears detrimental with the power gain not being as significant.

5.3.2 Assembling the final model

Provided the kernel analysis of section 5.2 and the exploration of improvement axes in the GPU micro architecture space of section 5.3, we are equipped to devise the two heterogeneous SM types our GPU will feature. The most common feature we detected in the Compute-Intensive kernels is the lack of one or more hardware resources that limit the execution. Those bottlenecks have been identified and most importantly can be addressed mainly by supplying the compute cores with the required hardware without going overboard with drastic changes. The aim of this is achieving better performance while keeping the consumed power reasonably low so as to increase the GPU's efficiency. These cores will be called High-Performance (HP) from now on. For the Low-Utilization kernels we follow the inverse approach. Their bottlenecks reside outside the SM or are imposed by the algorithm, in a way that the GPU core cannot be readily adapted to handle, and lead to overall sub-optimal utilization of the cards computational cores and especially their execution units. For that reason we consider reducing their execution pipeline under the assumption that their performance will not be greatly affected. However the otherwise unused components removed will no longer consume power when they are rarely activated and switched on and off in the case of power gating and will free up die area allowing us to scale up our

compute focused cores. Overall the goal is to develop an efficient platform which will allow Compute-Intensive kernels to exploit their innate warp level parallelism and run faster while the Low-Utilization ones will perform nearly as well as the baseline model however more economically so, leading both to faster overall execution and reduced energy consumption for the entire GPU. The Low-Utilization focused cores will be called Low-Power (LP)

We will initially look at how the functional units of each core will be treated. With regards to the first class of SMs - the High-Performance ones - the fist and major part of the pipeline we have to scale is the integer pipeline. We observed major improvement by doubling the integer functional units and room for improvement with some additional expansion up to three times the original without going overboard in terms of power relative to the baseline. Judging by the percentage integer instructions take up in the entire kernel and the structural integer stalls caused by lack of resources to execute them in parallel it makes sense to go for a higher amount of execution units and as such we will use 12 of them in the High-Performance cores. For double precision units the benefits from adding more were slighter and since DP instructions are less popular in the kernels causing fewer structural hazard stalls we will increase them more conservatively by 50% from four to six. Since no structural hazards in the SFU or SP pipeline were observed and the instruction mix suggests those instructions are scarce we will not increase the corresponding functional units. Also since we are in the "heavier" class of SMs we will not reduce them either, from their baseline values, sustaining both the performance of kernels who have even a little bit of those instructions such as BLhi and BLik who suffer from drastic cuts on those units and the core's capabilities in those areas in case an outlier kernel with more instructions of those types arises. For the increased functional units we scale their pipeline stages proportionally just as we did in the testing stage previously. In order to support the higher instruction throughput we must also increase the pipeline stages memory instructions follow which we double and the common writeback stage all execution lanes lead to which originally was eight registers long and now is configured to be quadruple that to account for all the other increases in the execution pipeline converging to that stage. Contrarily for the second type of cores, the Low-Power ones, aimed at Low-Utilization kernels we will attempt to reduce the functional units, making compromises performance wise to save in power and size compensating for the increases on the other core type. As we observed cutting SFU and SP units even down to just one minimally increases execution time on the average case while reducing the cards original hardware by a significant percentage. That is why our small SMs will contain just one SFU and one SP unit. Benchmarking results on our test set indicate that decreasing DP

functional units from the original four down to two had minimal performance impact on our test set of Low-Utilization kernels. DP instructions are more popular in the studied field of scientific computing in HPC and should be approached more conservatively in order to implement a core type capable of handling the variety of kernels potentially thrown at it. For that reason dropping their FUs down to one as well is not an option and the final Low-Power core type will feature two such units for which the performance compromise is minimal. The need for a higher threshold is even more necessary for Integer function units which we only reduce by one despite our study showing that decreasing them by two does not have much of a performance penalty either. We do so knowing that we have successfully reduced other types of functional units significantly so there is little reason in being aggressive with the one serving the most common type of instruction.

Then we take it one step back focusing on the Operand Collector supplying the execution units with instruction. As our analysis indicated, using specialized, per instruction type, operand collector units and adjusting their sizes for each instruction type based on the demands on the corresponding pipeline can dramatically enhance the performance of Compute-Intensive kernels. For that reason we choose to use the tested and effective configuration studied in 5.3.1 for our High-Performance cores while we will keep the generalized collector units in our Low-Power cores, since having too few specialized ones for certain pipelines could result in very poor utilization of the register file by eliminating parallel accesses when instructions aimed towards this pipeline arise. However due to the reduced parallelism observed in Low-Utilization kernels which will run on these SMs we will reduce them by 25% down to a total of six generalized operand collector units and we will reduce both their input and output ports by the same amount.

When it comes to the register file accessed by those operand collector units two parameters have to be considered. Its size and its banking level. We have already determined that an increased size allows register-hungry Compute-Intensive kernels to achieve higher occupancy often leading to significantly higher performance. For that reason we will increase the register file size by 50% to 384KB for our High-Performance cores. The performance of Low-Utilization kernels is not affected when increasing the available registers per thread however we choose not to decrease its size in the rest of the cores, so as not to worsen the already observed occupancy problems many such kernels face. Register file banks have to be cautiously modified since we observed they play a key role in power consumption. Although on their own they did not provide much benefit for our Compute-Intensive kernels, now with our High-Performance cores providing more parallelism and the increased number of operand requests instigated by the specialized operand collector

configuration, a higher level of parallel request servicing is required from the register file as well. Thirty two register file banks compared to the original sixteen will suite this purpose. Inversely for the Low-Power SMs, since their maximum available parallelism is decreased execution-wise it should also be decreased in the register file by dropping the banks to twelve.

For the warp schedulers responsible for deciding which warp will be executed we have two degrees of freedom when adjusting them for each core type. The first one is their type. The GTO scheduler clearly outperforms the other two when issuing warps from Compute-Intensive applications making it clear that we should select the GTO scheduler for the cores adjusted to handle such workloads while for Low-Utilization applications the choice appears to be indifferent performance wise. Under those circumstances we will sustain the LRR scheduler for the Low-Power cores to promote even progress among the warps, making sure not to worsen the issue of low warp availability as the execution of the kernel progresses. The second choice we have to make regarding warp schedulers configuration is the number of them. The increased level of parallelism we attempt to exploit in our stronger cores calls for an increased amount of warp schedulers so that more warps can be issued in parallel to the increased number of collector and subsequently execution units. However scaling them up to the size of the largest execution pipeline, in our case the Integer one with twelve functional units, as we did in the testing phase, would be unreasonable since the increased parallelism at that level could not be exploited by the rest of the pipelines and would rarely be, even by the Integer one itself. Warp schedulers issue one warp instruction per cycle for the V100 however an issued instruction commonly takes more than one cycle to initiate and execute on the pipeline stages of the appropriate unit so a downtime is introduced during which the execution pipeline cannot accept a new instruction. As such fewer warp schedulers would still mask this downtime just as efficiently by issuing on another vacant execution pipeline to which they did not previously issue. For this reason we only increase the warp schedulers per SM by two leading to a total of six. For Low-Utilization emphasized cores who as mentioned previously will have smaller pipelines for all types of instructions, following the same logic it is irrational to keep the warp schedulers at four so we reduce them down to two.

Obviously all the aforementioned changes require the lack of the sub-core model for the reasons explained in subsection 5.3.1.

As for the L1 size and the maximum number of active warps inside the SM, although these reconfiguration axes were tested in our previous analysis we opted to leave them out of the final model without this meaning that there is no point in altering them. Specifically regarding the L1 size we showed that increasing it does not significantly benefit any of our kernels

however decreasing it could make sense for some applications. Its unification with the shared memory though and the way it is implemented in the simulator provides little room for changes without requiring decoupling the two memories. Doing so would increase the total SRAM array size required to maintain the initial capacity of the shared memory and the new L1 capacity which would be larger than the original exclusively-L1-used 32KBs who on their own displayed significant performance decreases for three of our kernels. As for the maximum number of warps per SM although it would make sense to slightly reduce it for the SMs handling Low-Utilization kernels, we have already significantly reduced their available parallelism and limiting their available warps could take a heavier toll than initially appears in the isolated analysis. We believe it is a change better suited for exploration against an established baseline heterogeneous model and as such we leave it out of our proof of concept model.

Finally for the number of SMs of each type and their respective frequencies, the analysis required is better suited after the SM micro-architectural configuration has been established and is of higher importance to be conducted on the heterogeneous platform since more emphasis should be put on the way these parameters affect the entire system by interactions of the SMs to the directly connected and shared between the core types, off-SM memory interface consisting of the NOC, L2, and DRAM as well as the relationships between the timings of the aforementioned structures and the two types of SM. As such those axes elude the scope of this study.

Summarizing, the major characteristics of the two types of SMs are provided in table 5.6 alongside the V100 ones for comparison. Systems outside the SM such as the NOC, L2 cache and DRAM are left identical to the V100 configuration.

	High Performance (HP)	V100	Low-Power (LP)
SMs	40	80	40
Core Clock	1447MHz	1447MHz	1447MHz
RF Size	384KB	256KB	256KB
RF Banks	32	16	12
Max Threads/Core	2048	2048	2048
INT Units/Core	12	4	3
DP Units/Core	6	4	2
SP Units/Core	4	4	1
SFU Units/Core	4	4	1
Warp Scedulers/Core	6	4	2
Warp Scedulers Type	GTO	LRR	LRR
Collector Units	INT 16 4x4 DP 6 2x2 SP 4 1x1 SFU 4 1x1 MEM 8 1x1	General Purpose 8 8x8	General Purpose 6 6x6
Unified L1/Shared mem	32KB/96KB	32KB/96KB	32KB/96KB

Table 5.6: Final configuration of the two types of SMs featured in our prototype proof-of-concept heterogeneous GPU

Chapter 6

Evaluation

6.1 Introduction

In Section 5.3 we established the first revision of a single-ISA heterogeneous GPU which we are going to test in this Chapter. We will simulate the concurrent execution of kernels from the two groups identified previously in Sections 4.2 and 4.3 in order to evaluate its performance against multiple baselines. This concurrent execution will be facilitated by the developed benchmark suite described in Section 4.6 containing all the previously studied kernels.

6.2 Experimental Setup

The testing units involve the concurrent execution of two CUDA applications of comparable GPU execution-time to one another, each of them containing kernels of one of the two defined kernel classes.

The real-world targeted scenario involves streams of multiple kernels winding up in the same GPU and their execution being serialized per core type. Attempting to simulate an actual flow of kernels by sequentially issuing all our analyzed kernels on the two types of SMs would be problematic with many configuration parameters regarding the ordering of the kernels and their sizing, introducing bias to our results based on the choice we made, or forcing us to sample results in a random non-deterministic manner. Making a choice or using randomization would be enforced as otherwise for our relatively small set of five Compute-Intensive and thirteen Low-Utilization kernels there exist $5! \times 13! = 747242496000$ possible orderings for the two types and even if we assumed kernels of the same application and type would run sequentially it would only bring us down to $5! \times 10! = 435456000$ orderings considering

fixed problem sizes. Simulation of such an amount of configurations is not practical. We choose to test combinations in application level and not kernel level granularity since this method allows for a closer approximation of the targeted scenario where streams of potentially multiple kernels in a strict application-defined order wind up in the same GPU and their execution is serialized per core type. Many of our applications are originally designed solely as benchmarks and for that reason they are more likely to comprise only a single kernel. Those featuring more than one will serve to bring our results closer to a real world use case which will not be limited only to two kernels executing concurrently from the same starting point but will feature an application with multiple kernels offloading them sequentially as it was designed to. Those who do not will provide clearer results by isolating the interactions between specific kernels of certain known characteristics all the while remaining true to the basis of our scenario which is concurrent execution. In that fashion all combinations of applications can be tested exploring nearly all concurrent kernel interactions eliminating bias from our evaluation. Testing application level combinations is facilitated by the fact that multiple, execution-time-important as defined in subsection 5.2.3, kernels making up a single benchmark are almost always of the same type. This is important for our tests since we only have two applications issuing kernels and if they both attempted to issue to the same SM type the other one would be left vacant obscuring our test results. With our approach we promote concurrent execution up until an entire application has finished its execution and the other one is left to complete its own.

Continuing our previous point, for the same reason of maximizing concurrency, it is desired for our two applications to execute on the GPU for comparable amounts of time otherwise only a small portion of the execution on the proposed GPU would have two kernels running concurrently on the two types of cores and the rest would have half the card idling waiting for more work. In the application scenario high idling periods are not likely to occur when dealing with streams of multiple kernels. The execution times of the applications are adjustable through their inputs or configuration parameters which we tune to serve that purpose when feasible. For the combinations containing the GASAL benchmarks which are not run to completion due to their excessive execution time, we only study the portion for which the two applications executed simultaneously on the platform skipping the rest of GASAL's execution. For the baselines to be comparable to the results acquired in this way we measure the instructions executed from each application up to the stopping point of the simulation in the heterogeneous platform and execute the same amount for all other configurations. For the rest of the applications who are always run to completion no such

issue arises. The exception to the rule of a benchmark featuring only one type of time-critical kernels for our test set is lulesh, whose analyzed kernels comprise two Low-Utilization and one Compute-Intensive. Consequently it will be broken down and LUL1, LUL2 and LUL3 will be treated as separate applications in the study to follow. Kernels who take up a negligible portion of an applications execution time (like GRA2 or BLik_pre) will not affect the end results notably regardless of the core type they are executed on, so they are offloaded to the same SMs the rest of the application's kernels are. In order to execute the applications concurrently we treat each of the two otherwise standalone benchmarks as a thread of a common parent application which provides them both with their appropriate input arguments, separate CUDA streams and other necessary metadata as described in Section 4.6.

With this approach of combining applications containing Compute-Intensive kernels with applications containing Low-Utilization kernels we obtain a total of $5 \times 10 = 50$ combinations, presented in Table 6.1 alongside the problem sizes.

These applications are run on the single-ISA heterogeneous GPU simulator for the core type configuration of Table 5.6. The remainder of the card (i.e. not mentioned SM characteristics, memory system, interconnection network) is left unaltered compared to the V100. We also execute them on this version of the simulator for another configuration, that of the V100 itself but split into two 40 SM partitions just like our heterogeneous GPU. The purpose behind this is to isolate the effect partitioning the GPU and concurrently executing the different type kernels on the two partitions has on performance, without any differences between the SMs. The baseline against which performance will be measured is the original version of the V100 featuring 80 homogeneous SMs which will execute the two applications sequentially, modelled by simulating the execution of both applications separately on the original simulator and adding the execution times measured in core cycles. In the same manner we will test with the original simulator two other configurations both with 80 SMs but for the first one they will all have the configuration of the High-Performance core type while for the other one that of the Low-Power. All configurations feature the same memory system outside the SMs, that of the V100. For all models execution time performance is measured in core clock cycles reported by the timing simulators. Since all our cores operate on the same frequency of 1447 MHz each cycle corresponds to the same time unit of approximately 0.6911ns. To estimate the total energy we sum over all kernels, the energy calculated when a kernel finishes by multiplying the average power consumed throughout the execution window from the previous finished kernel (or the start of the entire execution if it is the first kernel to finish) up to its own finish point, as reported by the power

Low-Utilization App	Compute-Intensive App	Low-Utilization Size	Compute-Intensive Size
Polybench BICG (BIC1,BIC2)	BLonD Histogram (BLhi)	grid=(1024,1024)	nparticles=40000000
Polybench BICG (BIC1,BIC2)	Rodinia Hotspot (HOT)	grid=(1024,1024)	grid=(2048,2048)
Polybench BICG (BIC1,BIC2)	BLonD Interpolation-Kick (BLik_pre,BLik)	grid=(1024,1024)	nparticles=10000000
Polybench BICG (BIC1,BIC2)	BLonD Kick (BLki)	grid=(1024,1024)	nparticles=6000000
Polybench BICG (BIC1,BIC2)	lulesh (LUL3)	grid=(1024,1024)	46 edge nodes
BLonD Convolution (BLco)	BLonD Histogram (BLhi)	(nsignal,nkernel)=(1000,1000)	nparticles=1000000
BLonD Convolution (BLco)	Rodinia Hotspot (HOT)	(nsignal,nkernel)=(1000,1000)	grid=(512,512)
BLonD Convolution (BLco)	BLonD Interpolation-Kick (BLik_pre,BLik)	(nsignal,nkernel)=(1000,1000)	nparticles=1000000
BLonD Convolution (BLco)	BLonD Kick (BLki)	(nsignal,nkernel)=(1000,1000)	nparticles=1000000
BLonD Convolution (BLco)	lulesh (LUL3)	(nsignal,nkernel)=(1000,1000)	26 edge nodes
BLonD Drift (BLdr)	BLonD Histogram (BLhi)	nparticles=1000000	nparticles=1000000
BLonD Drift (BLdr)	Rodinia Hotspot (HOT)	nparticles=1000000	grid=(512,512)
BLonD Drift (BLdr)	BLonD Interpolation-Kick (BLik_pre,BLik)	nparticles=1000000	nparticles=1000000
BLonD Drift (BLdr)	BLonD Kick (BLki)	nparticles=1000000	nparticles=1000000
BLonD Drift (BLdr)	lulesh (LUL3)	nparticles=1000000	26 edge nodes
Polybench Gramschmidt (GRA1,GRA2,GRA3)	BLonD Histogram (BLhi)	grid=(512,512)	nparticles=40000000
Polybench Gramschmidt (GRA1,GRA2,GRA3)	Rodinia Hotspot (HOT)	grid=(512,512)	grid=(2048,2048)
Polybench Gramschmidt (GRA1,GRA2,GRA3)	BLonD Interpolation-Kick (BLik_pre, BLik)	grid=(512,512)	nparticles=10000000
Polybench Gramschmidt (GRA1,GRA2,GRA3)	BLonD Kick (BLki)	grid=(512,512)	nparticles=7000000
Polybench Gramschmidt (GRA1,GRA2,GRA3)	lulesh (LUL3)	grid=(512,512)	46 edge nodes
GASAL ksw (GApa,GAks)	BLonD Histogram (BLhi)	default query and target [3]	nparticles=5000000
GASAL ksw (GApa,GAks)	Rodinia Hotspot (HOT)	default query and target [3]	grid=(1024,1024)
GASAL ksw (GApa,GAks)	BLonD Interpolation-Kick (BLik_pre,BLik)	default query and target [3]	nparticles=2000000
GASAL ksw (GApa,GAks)	BLonD Kick (BLki)	default query and target [3]	nparticles=1000000
GASAL ksw (GApa,GAks)	lulesh (LUL3)	default query and target [3]	46 edge nodes
GASAL local (GApa,GAlo)	BLonD Histogram (BLhi)	default query and target [3]	nparticles=5000000
GASAL local (GApa,GAlo)	Rodinia Hotspot (HOT)	default query and target [3]	grid=(1024,1024)
GASAL local (GApa,GAlo)	BLonD Interpolation-Kick (BLik_pre,BLik)	default query and target [3]	nparticles=2000000
GASAL local (GApa,GAlo)	BLonD Kick (BLki)	default query and target [3]	nparticles=1000000
GASAL local (GApa,GAlo)	lulesh (LUL3)	default query and target [3]	46 edge nodes
GASAL global (GApa,GAgl)	BLonD Histogram (BLhi)	default query and target [3]	nparticles=5000000
GASAL global (GApa,GAgl)	Rodinia Hotspot (HOT)	default query and target [3]	grid=(1024,1024)
GASAL global (GApa,GAgl)	BLonD Interpolation-Kick (BLik_pre,BLik)	default query and target [3]	nparticles=2000000
GASAL global (GApa,GAgl)	BLonD Kick (BLki)	default query and target [3]	nparticles=1000000
GASAL global (GApa,GAgl)	lulesh (LUL3)	default query and target [3]	46 edge nodes
GASAL semi global (GApa,GAsg)	BLonD Histogram (BLhi)	default query and target [3]	nparticles=5000000
GASAL semi global (GApa,GAsg)	Rodinia Hotspot (HOT)	default query and target [3]	grid=(1024,1024)
GASAL semi global (GApa,GAsg)	BLonD Interpolation-Kick (BLik_pre,BLik)	default query and target [3]	nparticles=2000000
GASAL semi global (GApa,GAsg)	BLonD Kick (BLki)	default query and target [3]	nparticles=1000000
GASAL semi global (GApa,GAsg)	lulesh (LUL3)	default query and target [3]	46 edge nodes
lulesh (LUL1)	BLonD Histogram (BLhi)	23 edge nodes	nparticles=1000000
lulesh (LUL1)	Rodinia Hotspot (HOT)	23 edge nodes	grid=(512,512)
lulesh (LUL1)	BLonD Interpolation-Kick (BLik_pre,BLik)	23 edge nodes	nparticles=1000000
lulesh (LUL1)	BLonD Kick (BLki)	23 edge nodes	nparticles=1000000
lulesh (LUL1)	lulesh (LUL3)	23 edge nodes	23 edge nodes
lulesh (LUL2)	BLonD Histogram (BLhi)	23 edge nodes	nparticles=1000000
lulesh (LUL2)	Rodinia Hotspot (HOT)	23 edge nodes	grid=(512,512)
lulesh (LUL2)	BLonD Interpolation-Kick (BLik_pre,BLik)	23 edge nodes	nparticles=1000000
lulesh (LUL2)	BLonD Kick (BLki)	23 edge nodes	nparticles=1000000
lulesh (LUL2)	lulesh (LUL3)	23 edge nodes	23 edge nodes

Table 6.1: Evaluation set of the single-ISA heterogeneous architecture consisting of 50 combinations of Low-Utilization and Compute-Intensive Benchmarks alongside their main size determining parameters.

simulators, with the corresponding execution time reported in cycles by the timing simulators.

Finally the concurrent kernel execution on the GPU is expected to incur an interference penalty in the shared subsystems like the NOC, L2, and DRAM. Memory requests from two distinct kernels will be routed through the interconnection network to the various memory partitions causing increased response latencies through otherwise non-existent inter-kernel conflicts in all of those structures, and serialization of responses due to limited bandwidth [58, 25, 26]. This can degrade the performance of one or both kernels. In order to estimate the performance lost due to this effect we will

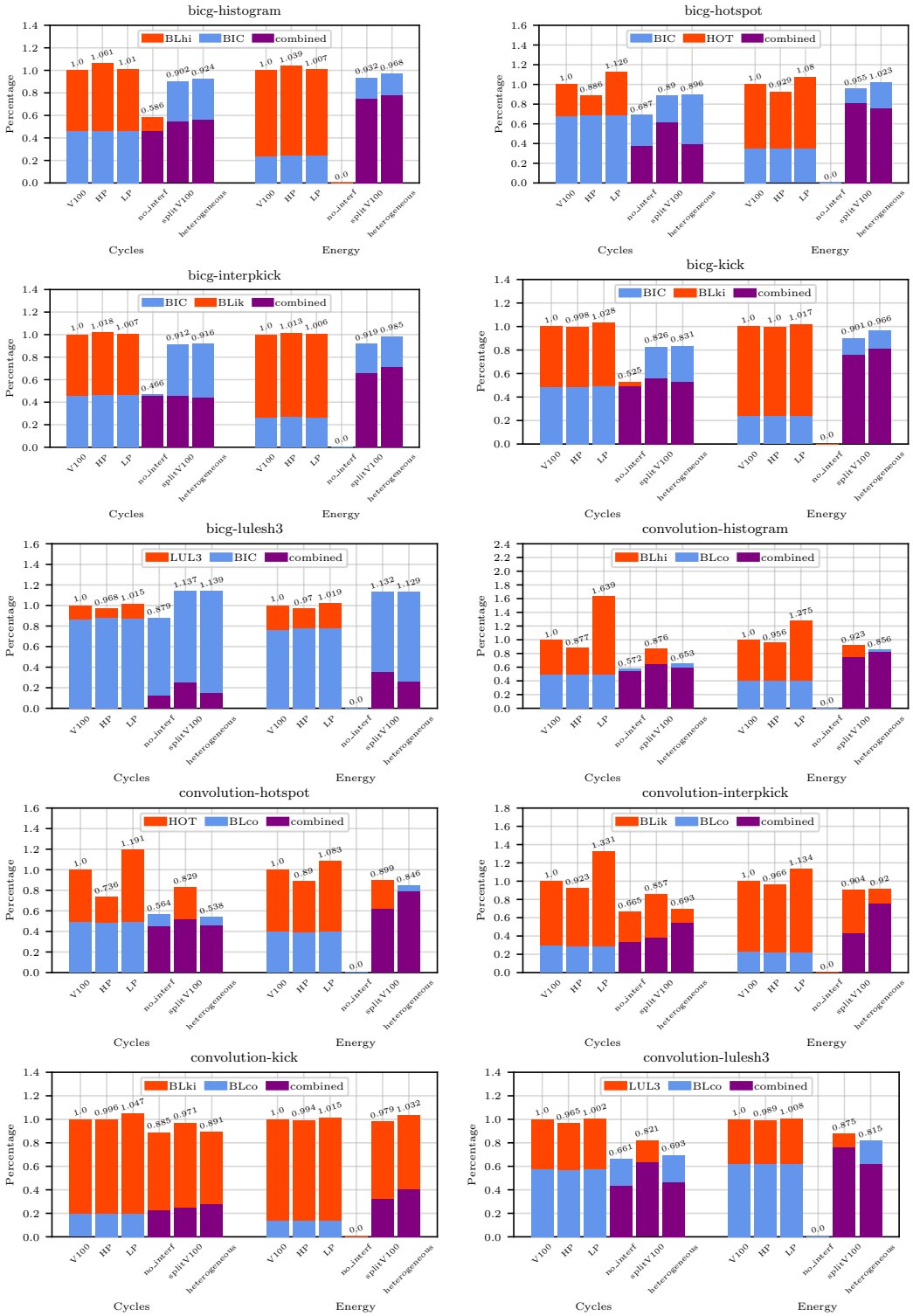
include one more analysis in our performance evaluation. This will involve executing both applications on the original simulator, one at a time, on a configuration featuring 40 SMs of the type it was meant to run on. This way each of the kernels has the exact same resources it did on the heterogeneous platform available to it, however with the entirety off-SM memory system at its disposal. Then by overlapping the timing results of the executions of the two kernels we can make an estimate on how they would perform on the heterogeneous platform if there was no interference present with the total execution time in this case represented by the maximum execution time of the two applications. Making power and energy estimates for this ideal scenario would be highly inaccurate since the DRAM power modelled by Accel-Sim [67] for the two configurations through detailed performance counters regarding low level memory accesses cannot be simply combined for the two executions into a unified memory system power consumption. For that reason energy estimates will be skipped for this configuration.

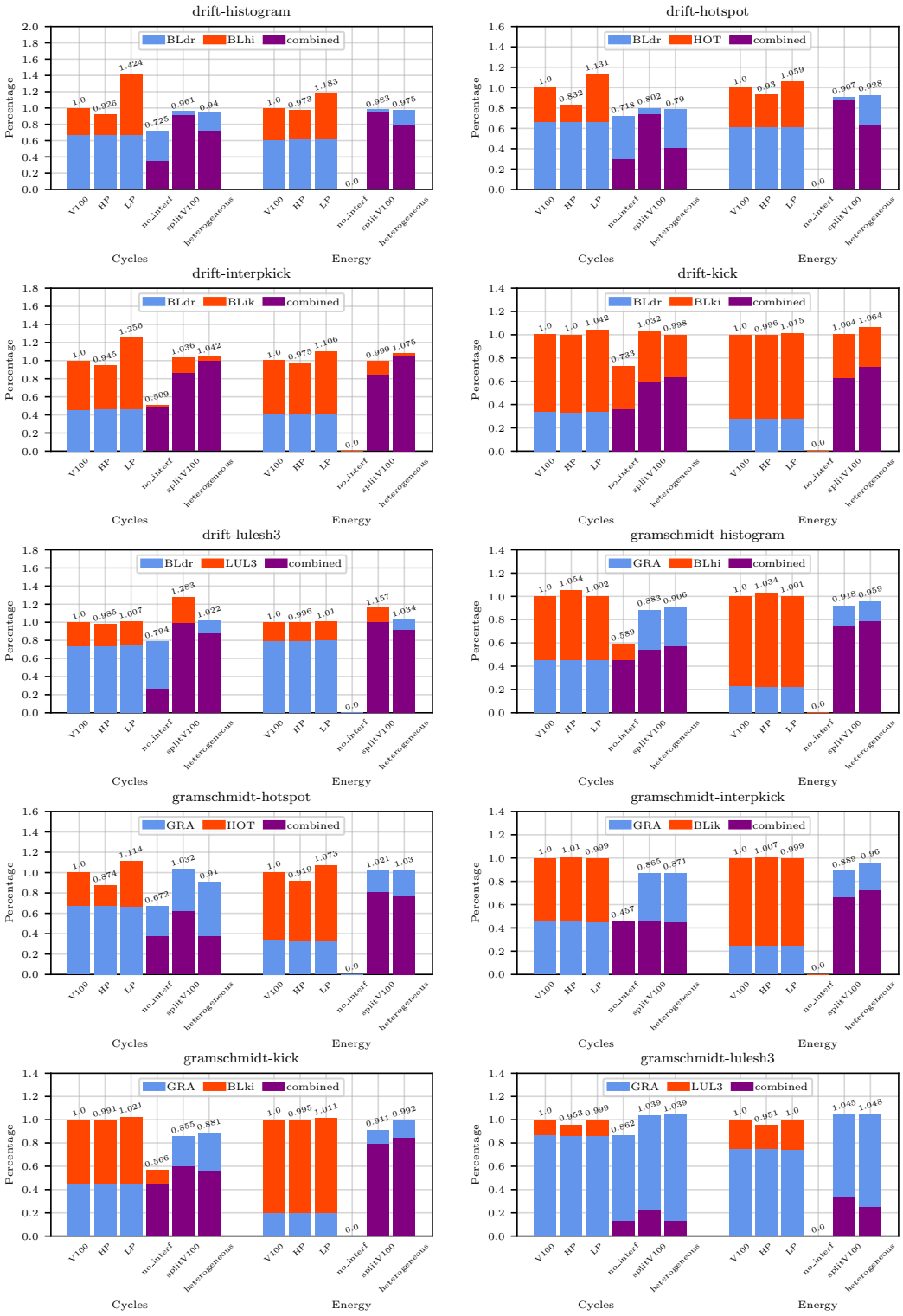
6.3 Results

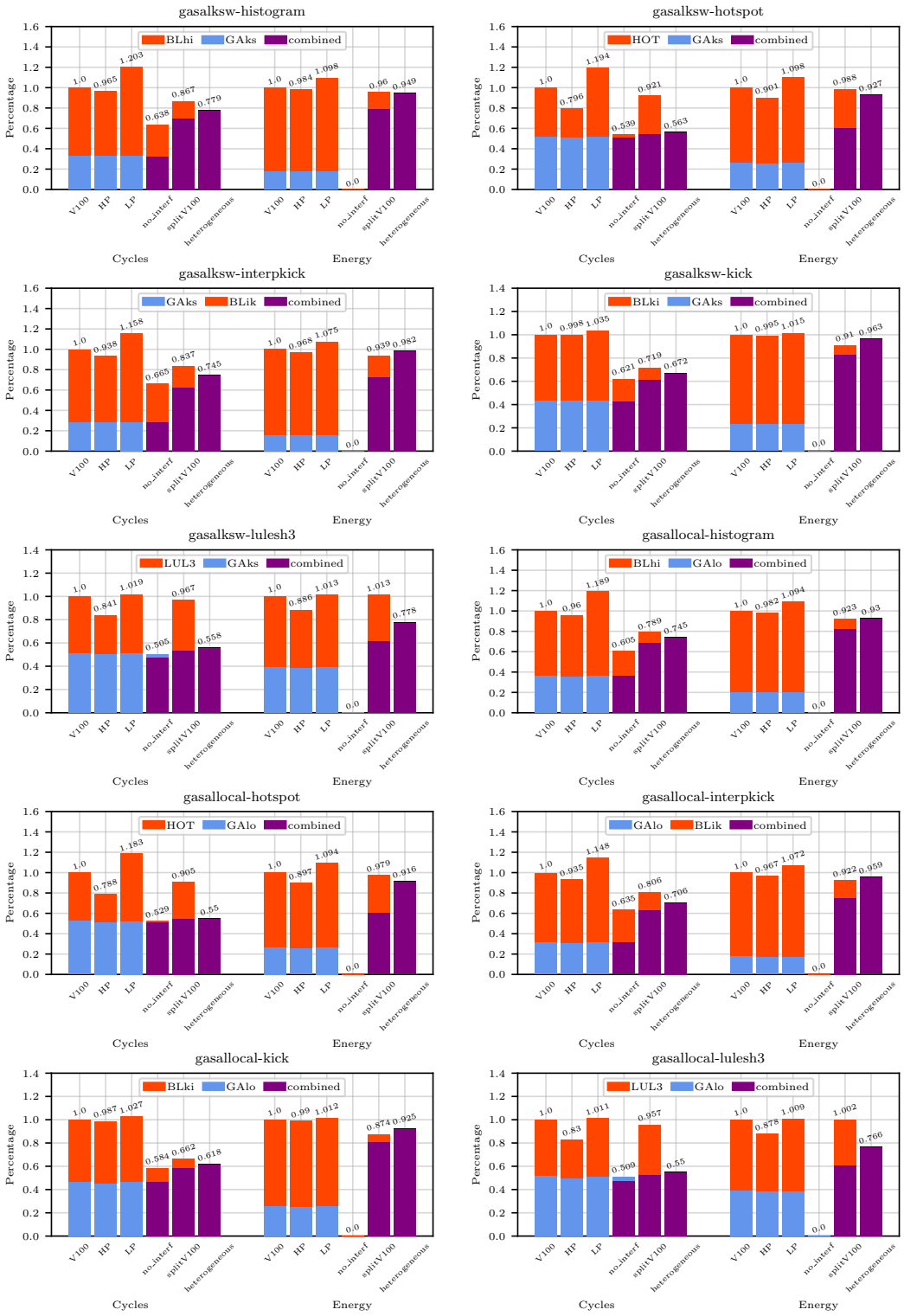
6.3.1 Results Overview

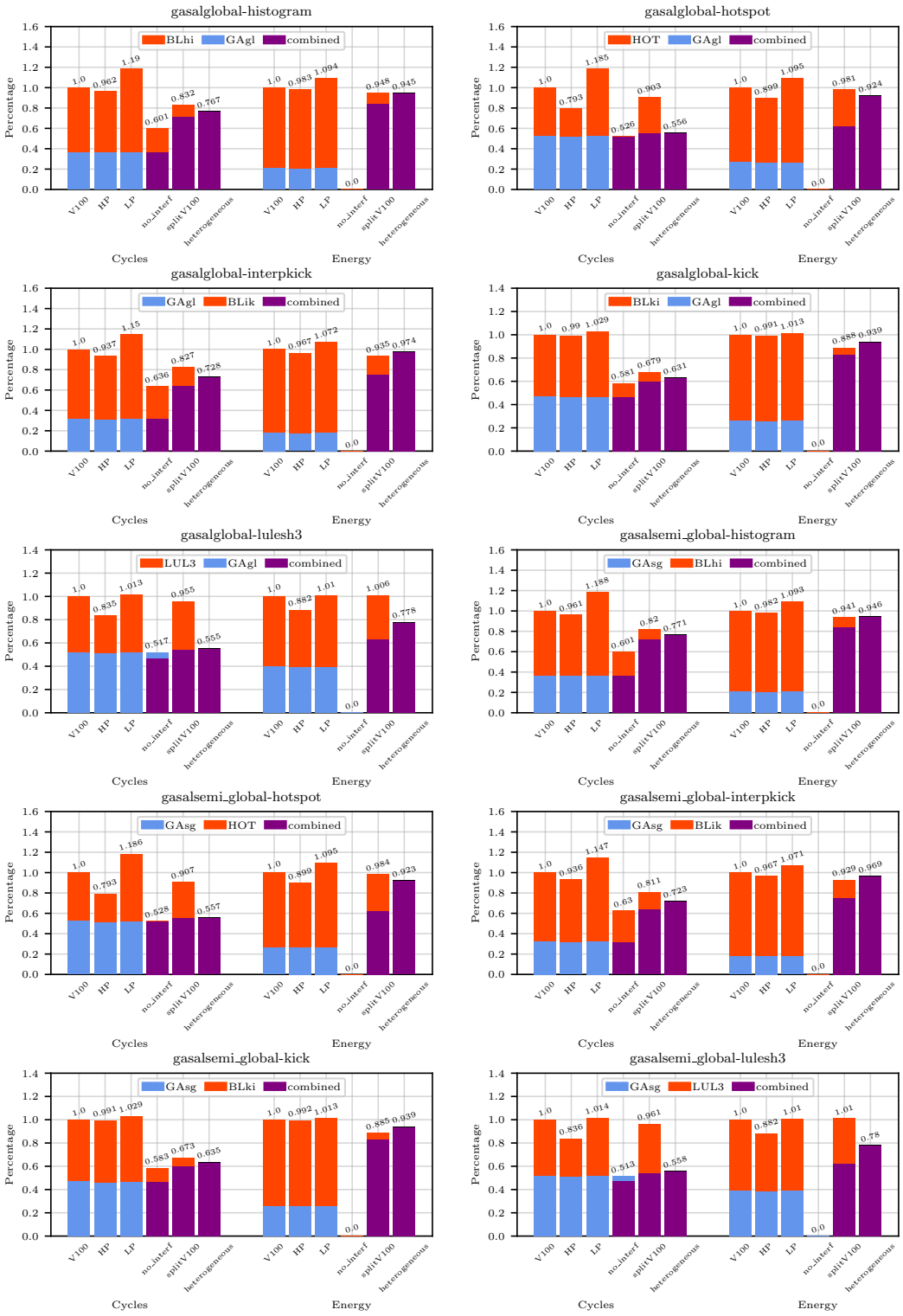
The results of each of the combined applications execution on the platforms described above are presented in Figure 6.1. All results, both for power and execution time respectively, are normalized to those of the sequential execution on the original V100 represented by the first bar. The second bar refers to the sequential execution on the GPU with 80 Compute-Intensive emphasized SMs and the third to that with 80 Low-Utilization emphasized SMs. Fourth (no_interf) corresponds to the configuration where kernels have exclusive access to shared off-SM resources with no interference while the fifth represents the homogeneous yet segmented V100, simulated on the new proposed version of Accel-Sim [67], with 80 SMs where dissimilar kernels can execute concurrently with 40 SMs available to each. Finally the last bar (heterogeneous) showcases the results achieved by our case of study heterogeneous GPU on the concurrent execution of the two scientific compute applications. In all configurations, the portion of the results attributed to applications featuring Low-Utilization kernels is colored in blue while that corresponding to the execution of Compute-Intensive ones is painted orange. The purple portion of the bars indicates concurrent execution of kernels from both applications for the configurations to which it is available.

A summary of the results across all 50 benchmarks is displayed in Figure 6.2. This evaluation analysis will focus on the geometric mean of the


bicg-histogram
BLhi
BIC
combined
Percentage
Cycles
Energy
V100
HP
LP
no_interf
splitV100
heterogeneous
bicg-hotspot
BIC
HOT
combined
bicg-interpkick
BIC
BLik
combined
bicg-kick
BIC
BLki
combined
bicg-lulesh3
LUL3
BIC
combined
convolution-histogram
BLhi
BLco
combined
convolution-hotspot
HOT
BLco
combined
convolution-interpkick
BLik
BLco
combined
convolution-kick
BLki
BLco
combined
convolution-lulesh3
LUL3
BLco
combined
Percentage


drift-histogram
BLdr
BLhi
combined
drift-hotspot
HOT
drift-interpkick
BLik
drift-kick
BLki
drift-lulesh3
LUL3
gramschmidt-histogram
GRA
gramschmidt-hotspot
gramschmidt-interpkick
gramschmidt-kick
gramschmidt-lulesh3
Percentage
V100
HP
LP
no_interf
splitV100
heterogeneous
Cycles
Energy


gasalksw-histogram
BLhi GAks combined
gasalksw-hotspot
HOT GAks combined
gasalksw-interpkick
GAks BLik combined
gasalksw-kick
BLki GAks combined
gasalksw-lulesh3
LUL3 GAks combined
gasallocal-histogram
BLhi GAlo combined
gasallocal-hotspot
HOT GAlo combined
gasallocal-interpkick
GAlo BLik combined
gasallocal-kick
BLki GAlo combined
gasallocal-lulesh3
LUL3 GAlo combined
Percentage
V100 HP LP no_interf splitV100 heterogeneous
Cycles
Energy


gasalglobal-histogram
BLhi GAgl combined
gasalglobal-hotspot
HOT GAgl combined
gasalglobal-interpkick
GAgl BLik combined
gasalglobal-kick
BLki GAgl combined
gasalglobal-lulesh3
LUL3 GAgl combined
gasalsemi_global-histogram
GAsg BLhi combined
gasalsemi_global-hotspot
GAsg HOT combined
gasalsemi_global-interpkick
GAsg BLik combined
gasalsemi_global-kick
GAsg BLki combined
gasalsemi_global-lulesh3
GAsg LUL3 combined
Percentage
V100 HP LP no_interf splitV100 heterogeneous
Cycles
Energy

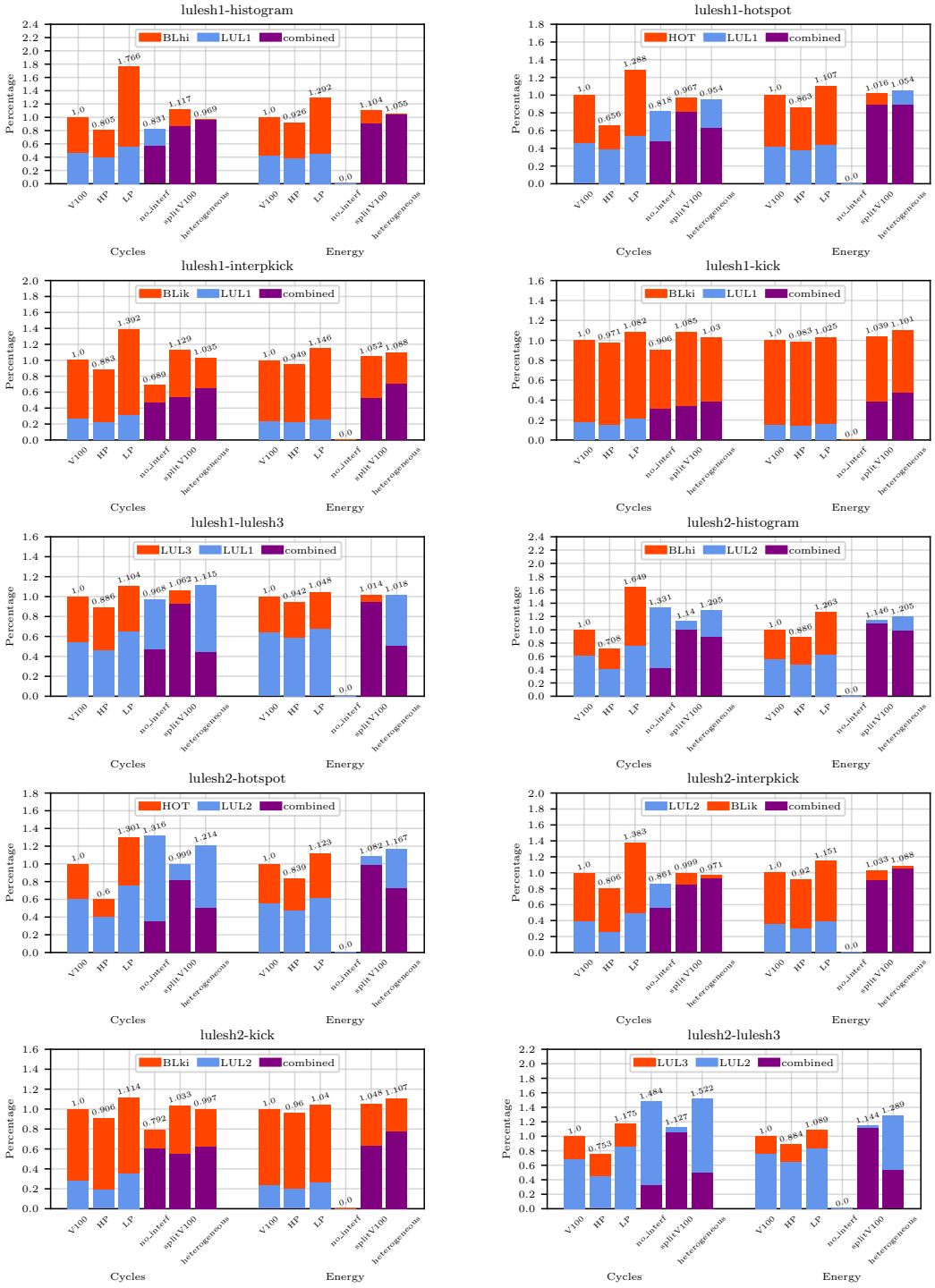


Figure 6.1: Evaluation of concurrent execution of Compute-Intensive and Low-Utilization Kernels on modelled single-ISA heterogeneous GPU for 50 distinct combinations of applications from the two classes

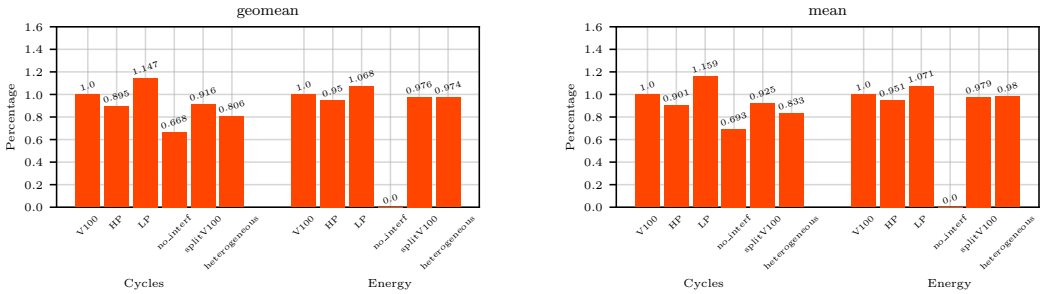


Figure 6.2: Summarized evaluation of concurrent execution of Compute-Intensive and Low-Utilization Kernels on modeled single-ISA heterogeneous GPU for 50 distinct combinations of applications from the two classes

results since the arithmetic mean is considered unreliable when summarizing the results of normalized performance values across machines [37]. However we still provide it for completeness. As we can see on average our model reduces total execution time by 19.4% compared to traditional sequential execution of the two applications on the baseline V100 model. Simply partitioning the card without employing any heterogeneity also performs better than the baseline but the heterogeneous model is still faster than it taking 12.1% less cycles to complete execution. The toll interference takes on the heterogeneous model is made obvious by the speedup of 49.7% on average the simulated no-interference approach achieves relative to the V100 compared to the 24.1% our realistic heterogeneous model achieves. Obviously the model with the 80 Compute-Intensive focused cores performs better than the standard V100 although by less than the heterogeneous model due to lacking the gain attributed to concurrent execution and the inability of the Low-Utilization kernels to exploit the stronger cores while inversely the model with the 80 smaller Low-Power SMs performs worse.

Unfortunately regarding energy consumption the very small fluctuations observed between the different models tested, render the fact that we are limited by Accelwattch's [63] power model as noted in subsection 5.2.2 quite evident. Since with this power model structures who were originally not utilized and were trimmed down for the Low-Utilization cores will not affect the results as would be expected, due to the performance counters remaining roughly the same between the configurations, we cannot visualize the gain execution on those cores offers us in terms of power. However the heterogeneous model still provides an indication towards its energy efficiency since the extra utilization of - and because of - the additional hardware on the High-Performance cores, which is taken into account through the performance counters, does not induce a spike in its energy consumption, quite

the contrary our design performs marginally better in terms of total energy consumption compared to the original V100 by 2.6% on average. This can be largely attributed to the faster execution times reducing the power-on time of the GPU.

6.3.2 Notable Cases

In this section we will examine some characteristic application combinations representative of distinct behaviours observed among the various applications showcased in Figure 6.1.

- **GASAL local - Hotspot:** It belongs to the class of the best available combinations. Hotspot responds really well to the modified High-Performance cores on which it executes while GASAL's local alignment performance is not affected by the down scaling of the SMs available to it. Simultaneously there is a small degree of interference between the two kernels featured in those benchmarks as is evident from the small difference between the execution time on the simulated new architecture and the no interference approximation. For those reasons plus the fact that performance of those kernels does not degrade linearly with the reduction of the SMs they run on by half, as observed in subsection 5.3.1, our proposed GPU performs excellently at running those applications concurrently achieving a 45% reduction in execution time. It is accompanied by an 8.4% energy gain.

- **Polybench BICG - Hotspot:** This combination's total execution time is dominated by the Low-Utilization application. Since all steps taken towards tailoring the SMs focused on Low-Utilization kernels were aimed towards improving their power consumption and not their performance we do not expect to observe any performance gains here. This is why although hotspot's kernel completes its execution faster than it does on the segmented V100 the total performance of the combination is not improved. The only way such a test could receive a performance boost from the introduced heterogeneity is by implicitly reducing the inter-kernel interference by having the Compute-Intensive application complete its execution earlier. Here it is not the case as the less time spent concurrently executing kernels is counter-balanced by the increased interference caused by the more aggressive execution of hotspot.

- **BLonD drift - BLonD interpolation-kick**: In this case we observe the heterogeneous platform performing 4.2% worse than our baseline

despite there being potential for performance gain in interpolation-kick based on the observed speedup with 80 compute-focused SMs. The reason for this is made clear by observing the performance of the no-interference model for which the execution time is reduced by 49.1%. It is the interference in the memory system from the interconnection network and below that bottlenecks the execution of those two kernels and leads to such a strong performance penalty for two benchmarks which in theory would perform excellently on such a platform if one was to look at the performance gain of interpolation-kick with High-Performance cores and the small loss of performance of both kernels by reducing the SMs in half. Another interesting observation is that the heterogeneous model performs even worse than the split V100. The extra pressure applied on the memory system by the wider pipelines executing interpolation-kick slows drift even more and in a cascading effect the longer execution time of drift causes prolonged interference to interpolation-kick.

- **Polybench BICG - BLonD Histogram:** For this combination of kernels we can see the applications performing worse than the baseline on the platform featuring 80 High-Performance SMs and histogram in particular whose kernel was earlier categorized as Compute-Intensive. The caveat is its input size. In order for the cycles it takes to complete to be comparable to those BICG took for a grid of 1024x1024 its input size was increased by 40 times, from one million to forty million particles, compared to the executed version of section 5.3. This impacts its behavior making it more memory bandwidth bound and losing its Compute-Intensive characteristics. This is the reason why the performance of this combination on our heterogeneous GPU is marginally worse than that when executed on the partitioned V100. A high degree of interference is also observed between the two bandwidth dependent applications. This indicates that a kernel's behavior is not immune to its input size and does not depend entirely on the source code and launch configuration.

- **Lulesh CalcFBHourglassForElems kernel - Hotspot** In this case we see a paradoxical behavior where the no-interference performance is worse than that of the actual heterogeneous system and specifically lulesh's kernel performance. Although this behavior may appear strange the interference caused by the concurrent execution of kernels in the GPU is capable of being beneficial in cases where it leads to better memory access patterns or warp scheduling due to differences in

the warps being stalled. The fact that more than one applications allocate memory regions also affects the end results. Application structures will be mapped to addresses differently compared to when one application exclusively occupies the GPU due to interleaving allocations from both applications. This can potentially lead to more favorable memory accesses as is the case for this benchmark combination where LUL3 encounters less global memory bank conflicts when run alongside Hotspot. It should be noted that this phenomenon is a double-edged sword, since although one application's performance improves the other one's is decreased more often than not. Here the application finishing last is the one which benefits, hence the total performance gain. This behavior is observed in some other tested combinations as well although not nearly as evident. For this to occur the application performing better on the actual heterogeneous model has to not be heavily affected by the bandwidth sharing. Finally although there are cases exposing this phenomenon it is far from the rule as we have already observed that on average applications execute 20.7% faster when they do not share memory resources.

Chapter 7

Summary and Future Directions

7.1 Summary

We present the idea of bringing single-ISA heterogeneity on a core level in the world of GPUs which is a sensible future direction considering the general path towards specialization and heterogeneity followed in computing. Enabling the study of such architectures, we expand the state of the art cycle accurate simulators Accel-Sim [67] and AccelWattch [63] to support configurations featuring SMs of different compositions and concurrently execute kernels on them. We extend the CUDA API with a new call giving the end users the ability to denote the type of SM they want each of their kernels to run on within the source code of the benchmarks allowing for portability of the benchmarks without needing simulation configurations to denote the preferred core type. To facilitate the study of concurrent kernel execution on said architectures we develop an expandable benchmark suite pre-equipped with multiple benchmarks to which more can be added through a standardized process.

To showcase the significance of our proposal we perform a case study involving improving the performance of scientific computation applications executed within HPC Clusters featuring GPUs. Multiple batches of application runs along with GPU sharing and virtualization techniques employed in such clusters provide the ground for kernel level parallelism. We select a representative set of scientific computation involved kernels and study their compile-time as well as their execution characteristics on Accel-Sim's [67] PTX environment showing that they expose different patters of hardware utilization and bottleneck points, based on which they are placed in two different groups. We then attempt to tailor the cores of the GPU starting from the Volta V100 to create two new configurations each better suited towards executing one of the two groups of kernels by testing how the studied kernels

respond to specific micro-architectural changes.

We then proceed to evaluate this new architecture by concurrently executing benchmarks containing kernels from the two previously identified groups, showing a considerable 24.1% speedup over sequential execution on the baseline V100, without energy compromises but with a 2.6% gain.

7.2 Future Work

- Further extension and testing of Accel-Sim [67] to support trace-based simulation on the heterogeneous platform providing more accurate simulation results and exposure to more intrinsic execution characteristics, not simulated when using the PTX virtual ISA.

- Enabling support for currently not supported differentiations between the core types such as re-convergence strategy and texture cache line width.

- Development of a power model capable of fully adapting to hardware re-configuration and does not solely rely on mathematically tuned coefficients based on measurements of real, implemented hardware.

- Enabling support for drastic architectural changes between the two core types not yet available in Accel-Sim [67]. For example we observed kernels who presented with limited active warp availability whose performance we were not able to improve through available micro-architectural changes and were deemed Low-Utilization. However literature suggests that there exist micro-architectures capable of speeding such kernels up like LOOG [51] which introduces light-weight out of order execution to the GPU enabling ILP to hide the latencies execution of different warps can not. Although such a scheme could benefit any kernel, it has a specific target group, as is commonplace for a vast variety of work on GPUs considering the trend on specialization. SM level heterogeneity allows for such architectural models to be applied exclusively to the beneficiary kernels, allowing the rest of the applications to be executed on SMs tailored for their dissimilar needs. By implementing specialized architectures studied in literature and allowing combination of them we open up a much broader design space allowing for even better results.

- Deeper studying of characteristics outside the SM like the number of SMs or the clock frequencies they operate on

- Implementing mechanisms to boost performance by alleviating the interference in the common memory system. Literature has already explored the effect co-location of kernels on the GPU has regarding memory inefficiencies due to traffic from more than one kernels and potential solutions have been proposed such as alternative DRAM schedulers [58], balancing memory requests from the kernels or limiting them [26], and assigning addresses differently to isolate the different kernels within the memory partition in the same manner MIG [94] does.

- Implementation of security mechanisms to ensure kernel isolation in the memory system when necessary and provide fault tolerance between applications concurrently issuing workloads to the GPU.

Appendix A

Source Code

A.1 Source Code repositories

The source code for the simulator and benchmark suite implementations of Chapter 4 can be found at:
https://github.com/Moiralex/bigLITTLE_gpgpu-sim_distribution
https://github.com/Moiralex/bigLITTLE_GPUapps

A.2 Original License

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

BSD 2-Clause License

Copyright (c) 2018-2021, Vijay Kandiah, Junrui Pan, Mahmoud Khairy, Scott Peverelle, Timothy Rogers, Tor M. Aamodt, Nikos Hardavellas Northwestern University, Purdue University, The University of British Columbia All rights reserved.

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

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer; 2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution; 3. Neither the names of Northwestern University, Purdue University, The University of British Columbia nor the names of their contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS

Bibliography

[1] Tor M. Aamodt, Wilson Wai Lun Fung, and Timothy G. Rogers. *General-purpose Graphics Processor Architectures*. Morgan & Claypool Publishers, 2018. ISBN: 1627059237.

[2] Jacob T Adriaens et al. "The case for GPGPU spatial multitasking". In: *IEEE International Symposium on High-Performance Comp Architecture*. IEEE. 2012, pp. 1–12.

[3] Nauman Ahmed et al. "GASAL2: a GPU accelerated sequence alignment library for high-throughput NGS data". In: *BMC bioinformatics* 20 (2019), pp. 1–20.

[4] Jaeguk Ahn et al. "Network-on-chip microarchitecture-based covert channel in GPUs". In: *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 2021, pp. 565–577.

[5] Mohammed Alser et al. "Accelerating genome analysis: A primer on an ongoing journey". In: *IEEE Micro* 40.5 (2020), pp. 65–75.

[6] AMD. *PUSHING GRAPHICS BANDWIDTH PERFORMANCE FURTHER, AMD Infinity Cache™ Technology Explained*. URL: https://www.amd.com/system/files/documents/infinity-cache-technology-explained.pdf.

[7] AMD. *ROCm HIP*. URL: https://github.com/ROCm/HIP.

[8] ARM. *Arm Big.LITTLE*. URL: https://www.arm.com/technologies/big-little.

[9] ARM. *ARMv8-A Power management*. URL: https://developer.arm.com/documentation/100960/0100/Dynamic-Voltage-and-Frequency-Scaling.

[10] ARM. *big.LITTLE Technology: The Future of Mobile*. URL: https://armkeil.blob.core.windows.net/developer/Files/pdf/white-paper/big-little-technology-the-future-of-mobile.pdf.

[11] David H Bailey. “High-precision floating-point arithmetic in scientific computation”. In: *Computing in science & engineering* 7.3 (2005), pp. 54–61.

[12] Ali Bakhoda et al. “Analyzing CUDA workloads using a detailed GPU simulator”. In: *2009 IEEE international symposium on performance analysis of systems and software*. IEEE. 2009, pp. 163–174.

[13] Gert-Jan Van Den Braak and Henk Corporaal. “R-gpu: A reconfigurable gpu architecture”. In: *ACM Transactions on Architecture and Code Optimization (TACO)* 13.1 (2016), pp. 1–24.

[14] Alexander Branover, Denis Foley, and Maurice Steinman. “Amd fusion apu: Llano”. In: *Ieee Micro* 32.2 (2012), pp. 28–37.

[15] André R Brodtkorb, Trond R Hagen, and Martin L Sætra. “Graphics processing unit (GPU) programming strategies and trends in GPU computing”. In: *Journal of Parallel and Distributed Computing* 73.1 (2013), pp. 4–13.

[16] Pablo Carvalho et al. “,” in: *Future Generation Computer Systems* 113 (2020), pp. 528–540.

[17] Niladrish Chatterjee et al. “Managing DRAM latency divergence in irregular GPGPU applications”. In: *SC’14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2014, pp. 128–139.

[18] Shuai Che et al. “Rodinia: A benchmark suite for heterogeneous computing”. In: *IISWC*. IEEE. 2009, pp. 44–54.

[19] Xuhao Chen et al. “Adaptive cache management for energy-efficient GPU computing”. In: *2014 47th Annual IEEE/ACM international symposium on microarchitecture*. IEEE. 2014, pp. 343–355.

[20] Xianwei Cheng et al. “AMOEBA: a coarse grained reconfigurable architecture for dynamic GPU scaling”. In: *Proceedings of the 34th ACM International Conference on Supercomputing*. 2020, pp. 1–13.

[21] Hongsuk Chung, Munsik Kang, and Hyun-Duk Cho. “Heterogeneous multi-processing solution of Exynos 5 Octa with ARM big. LITTLE technology”. In: *Samsung White Paper* (2012).

[22] University of Cincinaty. *Task Level Parallelism*. URL: https://eecs.ceas.uc.edu/~wilseypa/classes/eece7095/lectureNotes/parallelism/taskLevelParallelism.pdf.

[23] Lyndon Clarke, Ian Glendinning, and Rolf Hempel. "The MPI message passing interface standard". In: *Programming Environments for Massively Parallel Distributed Systems: Working Conference of the IFIP WG 10.3, April 25–29, 1994*. Springer. 1994, pp. 213–218.

[24] Brett W Coon and JE Lindholm. *United States Patent# 7,353,369: System and Method for Managing Divergent Threads in a SIMD Architecture (Assignee NVIDIA Corp.)* 2008.

[25] Rommel Cruz et al. "Analyzing and estimating the performance of concurrent kernels execution on GPUs". In: *XVIII Simpósio em Sistemas Computacionais de Alto Desempenho-WSCAD* (2017).

[26] Hongwen Dai et al. "Accelerate GPU concurrent kernel execution by mitigating memory pipeline stalls". In: *2018 IEEE international symposium on high performance computer architecture (HPCA)*. IEEE. 2018, pp. 208–220.

[27] Hongwen Dai et al. "Accelerate GPU concurrent kernel execution by mitigating memory pipeline stalls". In: *2018 IEEE international symposium on high performance computer architecture (HPCA)*. IEEE. 2018, pp. 208–220.

[28] Sana Damani et al. "Speculative reconvergence for improved SIMT efficiency". In: *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*. 2020, pp. 121–132.

[29] MAYANK DHAM. *Instruction level parallelism(ILP)*. URL: https://www.prepbytes.com/blog/computer-architecture/instruction-level-parallelismilp/.

[30] Gregory Diamos et al. "SIMD re-convergence at thread frontiers". In: *Proceedings of the 44th annual ieee/acm international symposium on microarchitecture*. 2011, pp. 477–488.

[31] Gregory Frederick Diamos et al. *Execution of divergent threads using a convergence barrier*. US Patent 10,067,768. 2018.

[32] Michael Doggett. "Texture caches". In: *IEEE Micro* 32.3 (2012), pp. 136–141.

[33] Ralph Duncan. "A survey of parallel computer architectures". In: *Computer* 23.2 (1990), pp. 5–16.

[34] John H Edmondson and James M Van Dyke. *Memory addressing scheme using partition strides*. US Patent 7,872,657. 2011.

[35] Ahmed ElTantawy and Tor M Aamodt. "MIMD synchronization on SIMT architectures". In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2016, pp. 1–14.

[36] Juan Fang, Zelin Wei, and Huijing Yang. "Locality-based cache management and warp scheduling for reducing cache contention in GPU". In: *Micromachines* 12.10 (2021), p. 1262.

[37] Philip J Fleming and John J Wallace. "How not to lie with statistics: the correct way to summarize benchmark results". In: *Communications of the ACM* 29.3 (1986), pp. 218–221.

[38] Michael J Flynn. "Some computer organizations and their effectiveness". In: *IEEE transactions on computers* 100.9 (1972), pp. 948–960.

[39] Michael J Flynn. "Very high-speed computing systems". In: *Proceedings of the IEEE* 54.12 (1966), pp. 1901–1909.

[40] Yaosheng Fu et al. "GPU domain specialization via composable on-package architecture". In: *ACM Transactions on Architecture and Code Optimization (TACO)* 19.1 (2021), pp. 1–23.

[41] Wilson WL Fung and Tor M Aamodt. "Thread block compaction for efficient SIMT control flow". In: *2011 IEEE 17th international symposium on high performance computer architecture*. IEEE. 2011, pp. 25–36.

[42] Prasun Gera et al. "Performance characterisation and simulation of Intel's integrated GPU architecture". In: *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE. 2018, pp. 139–148.

[43] David B Glasco et al. *Virtual channels for effective packet transfer*. US Patent 8,539,130. 2013.

[44] Scott Grauer-Gray et al. "Auto-tuning a high-level language targeted to GPU codes". In: *2012 innovative parallel computing (InPar)*. Ieee. 2012, pp. 1–10.

[45] Khronos Group. *OpenCL*. URL: https://www.khronos.org/opencl/.

[46] Stanley P Gudder. "A superposition principle in physics". In: *Journal of Mathematical Physics* 11.3 (1970), pp. 1037–1040.

[47] Linley Gwennap. "Groq rocks neural networks". In: *Microprocessor Report, Tech. Rep., jan* (2020).

[48] Ziyad S Hakura and Anoop Gupta. "The design and analysis of a cache architecture for texture mapping". In: *Proceedings of the 24th annual international symposium on Computer architecture*. 1997, pp. 108–120.

[49] HEAVY.AI. *What is Parallel Computing? Definition and FAQs*. URL: https://www.heavy.ai/technical-glossary/parallel-computing.

[50] *Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory*. Tech. rep. LLNL-TR-490254. Livermore, CA, pp. 1–17.

[51] 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.

[52] Konstantinos Iliakis et al. "Enabling large scale simulations for particle accelerators". In: *IEEE Transactions on Parallel and Distributed Systems* 33.12 (2022), pp. 4425–4439.

[53] Intel. *Intel performance hybrid architecture & software optimizations*. URL: https://cdrdv2-public.intel.com/685861/211115_Hybrid_WP_1_Introduction_v1.2.pdf.

[54] Saksham Jain et al. "Fractional GPUs: Software-based compute and memory bandwidth reservation for GPUs". In: *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE. 2019, pp. 29–41.

[55] JEDEC JESD250. "Graphics double data rate 6 (GDDR6) SGRAM standard". In: *JEDEC Solid State Technology Association* (2017).

[56] Nan Jiang et al. "A detailed and flexible cycle-accurate network-on-chip simulator". In: *2013 IEEE international symposium on performance analysis of systems and software (ISPASS)*. IEEE. 2013, pp. 86–96.

[57] Qing Jiao et al. "Improving GPGPU energy-efficiency through concurrent kernel execution and DVFS". In: *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2015, pp. 1–11.

[58] Adwait Jog et al. "Anatomy of gpu memory system for multi-application execution". In: *Proceedings of the 2015 International Symposium on Memory Systems*. 2015, pp. 223–234.

[59] Adwait Jog et al. “Anatomy of gpu memory system for multi-application execution”. In: *Proceedings of the 2015 International Symposium on Memory Systems*. 2015, pp. 223–234.

[60] Adwait Jog et al. “OWL: cooperative thread array aware scheduling techniques for improving GPGPU performance”. In: *SIGPLAN*. ACM. 2013, pp. 395–406.

[61] Eric E Johnson. “Completing an MIMD multiprocessor taxonomy”. In: *ACM SIGARCH Computer Architecture News* 16.3 (1988), pp. 44–47.

[62] Norman P Jouppi et al. “In-datacenter performance analysis of a tensor processing unit”. In: *Proceedings of the 44th annual international symposium on computer architecture*. 2017, pp. 1–12.

[63] Vijay Kandiah et al. “AccelWattch: A power modeling framework for modern GPUs”. In: *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 2021, pp. 738–753.

[64] Ian Karlin et al. *LULESH Programming Model and Performance Ports Overview*. Tech. rep. LLNL-TR-608824. Livermore, CA, Dec. 2012, pp. 1–17.

[65] Onur Kayıran et al. “Neither more nor less: Optimizing thread-level parallelism for GPGPUs”. In: *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*. IEEE. 2013, pp. 157–166.

[66] Mahmoud Khairy, Mohamed Zahran, and Amr G Wassal. “Efficient utilization of gpgpu cache hierarchy”. In: *Proceedings of the 8th Workshop on General Purpose Processing using GPUS*. 2015, pp. 36–47.

[67] Mahmoud Khairy et al. “Accel-Sim: An Extensible Simulation Frameworkfor Validated GPU Modeling”. In: *2020 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2020.

[68] Rakesh Kumar et al. “Processor power reduction via single-ISA heterogeneous multi-core architectures”. In: *IEEE Computer Architecture Letters* 2.1 (2003), pp. 2–2.

[69] Rakesh Kumar et al. “Single-ISA heterogeneous multi-core architectures for multithreaded workload performance”. In: *ACM SIGARCH Computer Architecture News* 32.2 (2004), p. 64.

[70] Rakesh Kumar et al. “Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction”. In: *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.* IEEE. 2003, pp. 81–92.

[71] Nagesh B Lakshminarayana et al. “DRAM scheduling policy for GPGPU architectures based on a potential function”. In: *IEEE Computer Architecture Letters* 11.2 (2011), pp. 33–36.

[72] Samuel Larsen and Saman Amarasinghe. “Exploiting superword level parallelism with multimedia instruction sets”. In: *Acm Sigplan Notices* 35.5 (2000), pp. 145–156.

[73] Ahmad Lashgar, Ebad Salehi, and Amirali Baniasadi. “A case study in reverse engineering gpgpus: Outstanding memory handling resources”. In: *ACM SIGARCH Computer Architecture News* 43.4 (2016), pp. 15–21.

[74] Steven J Leon, Åke Björck, and Walter Gander. “Gram-Schmidt orthogonalization: 100 years and more”. In: *Numerical Linear Algebra with Applications* 20.3 (2013), pp. 492–532.

[75] 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.

[76] Yuxi Liu et al. “Barrier-aware warp scheduling for throughput processors”. In: *Proceedings of the 2016 International Conference on Supercomputing.* 2016, pp. 1–12.

[77] Zhuren Liu et al. “Genomics-GPU: A Benchmark Suite for GPU-accelerated Genome Analysis”. In: *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS).* IEEE. 2023, pp. 178–188.

[78] David Loshin. *Data Parallelism.* URL: https://www.sciencedirect.com/topics/computer-science/data-parallelism.

[79] David Luebke. “CUDA: Scalable parallel programming for high-performance scientific computing”. In: *2008 5th IEEE international symposium on biomedical imaging: from nano to macro.* IEEE. 2008, pp. 836–838.

[80] David Mayhew and Venkata Krishnan. “PCI Express and Advanced Switching: Evolutionary path to building next generation interconnects”. In: *11th Symposium on High Performance Interconnects, 2003. Proceedings.* IEEE. 2003, pp. 21–29.

[81] MIT. *Lecture: Graphics pipeline and animation*. URL: https://web.archive.org/web/20171207095603/http://goanna.cs.rmit.edu.au/~gl/teaching/rtr%263dgp/notes/pipeline.html.

[82] Veynu Narasiman et al. "Improving GPU performance via large warps and two-level warp scheduling". In: *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. 2011, pp. 308–317.

[83] Mohammed Al-Neama, Naglaa Reda, and Fayed Ghaleb. *A study of parallel algorithms for multiple sequence alignment*. May 2016. ISBN: 3659883786.

[84] Bradford Nichols, Dick Buttlar, and Jacqueline Farrell. *Pthreads programming: A POSIX standard for better multiprocessing*. " O'Reilly Media, Inc.", 1996.

[85] NVIDIA. *cuBLAS*. URL: https://docs.nvidia.com/cuda/cublas/index.html.

[86] NVIDIA. *CUDA C++ Programming Guide*. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#.

[87] NVIDIA. *cuFFT API Reference*. URL: https://docs.nvidia.com/cuda/cufft/index.html.

[88] NVIDIA. *Kernel Profiling Guide*. URL: https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html#metric-collection.

[89] NVIDIA. *Memory Statistics - Local*. URL: https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/memorystatisticslocal.htm.

[90] NVIDIA. *MULTI-PROCESS SERVICE*. URL: https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf.

[91] NVIDIA. *NVIDIA ADA GPU ARCHITECTURE*. 2022. URL: https://images.nvidia.com/aem-dam/Solutions/Data-Center/l4/nvidia-ada-gpu-architecture-whitepaper-v2.1.pdf.

[92] NVIDIA. *NVIDIA AMPERE GA102 GPU ARCHITECTURE*. URL: https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf.

[93] NVIDIA. *NVIDIA cuDNN*. URL: https://docs.nvidia.com/deeplearning/cudnn/.

[94] NVIDIA. *NVIDIA Multi-Instance GPU User Guide*. URL: https://docs.nvidia.com/datacenter/tesla/mig-user-guide/index.html.

[95] NVIDIA. *NVIDIA TESLA P100*. 2016. URL: https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf.

[96] NVIDIA. "NVIDIA Tesla V100 Architecture". In: *NVIDIA White Paper* (2017).

[97] NVIDIA. *NVIDIA TESLA V100 GPU ARCHITECTURE*. 2017. URL: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[98] NVIDIA. *NVIDIA TURING GPU ARCHITECTURE*. 2018. URL: https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf.

[99] NVIDIA. *NVIDIA's Next Generation CUDA Compute Architecture: Fermi*. URL: https://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf.

[100] NVIDIA. "NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110/210". In: *NVIDIA White Paper* (2012).

[101] NVIDIA. *NVIDIA® NVLink TM High-Speed Interconnect: Application Performance*. URL: https://info.nvidianews.com/rs/nvidia/images/NVIDIA%20NVLink%20High-Speed%20Interconnect%20Application%20Performance%20Brief.pdf.

[102] NVIDIA. *NVLink and NVLink Switch*. URL: https://www.nvidia.com/en-us/data-center/nvlink/.

[103] NVIDIA. *PTX ISA 8.3*. URL: https://docs.nvidia.com/cuda/parallel-thread-execution/index.html.

[104] Justin Luitjens - NVIDIA. *CUDA STREAMS BEST PRACTICES AND COMMON PITFALLS*. URL: https://on-demand.gputechconf.com/gtc/2014/presentations/S4158-cuda-streams-best-practices-common-pitfalls.pdf.

[105] Mark Harris NVIDIA. *Optimizing Parallel Reduction in CUDA*. URL: https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf.

[106] Paulius Micikevicius NVIDIA. *Local Memory and Register Spilling*. URL: https://developer.download.nvidia.com/CUDA/training/register_spilling.pdf.

[107] NVIDIA Corporation. *NVIDIA A100 Tensor Core GPU Architecture*. 2020. URL: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf.

[108] G Ortega et al. "The BiConjugate gradient method on GPUs". In: *The Journal of Supercomputing* 64 (2013), pp. 49–58.

[109] John D Owens et al. "GPU computing". In: *Proceedings of the IEEE* 96.5 (2008), pp. 879–899.

[110] German I Parisi et al. "Continual lifelong learning with neural networks: A review". In: *Neural networks* 113 (2019), pp. 54–71.

[111] Antonio J Pena et al. "A complete and efficient CUDA-sharing solution for HPC clusters". In: *Parallel Computing* 40.10 (2014), pp. 574–588.

[112] Vignesh T Ravi et al. "Supporting GPU sharing in cloud environments with a transparent runtime consolidation framework". In: *Proceedings of the 20th international symposium on High performance distributed computing*. 2011, pp. 217–228.

[113] Timothy G Rogers, Mike O'Connor, and Tor M Aamodt. "Cache-conscious wavefront scheduling". In: *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE. 2012, pp. 72–83.

[114] Efraim Rotem et al. "Intel alder lake cpu architectures". In: *IEEE Micro* 42.3 (2022), pp. 13–19.

[115] Jeffrey B Rothman and Alan Jay Smith. "Sector cache design and performance". In: *Proceedings 8th international symposium on modeling, analysis and simulation of computer and telecommunication systems (cat. no. pr00728)*. IEEE. 2000, pp. 124–133.

[116] Leonid Ivanovich Sedov. "Propagation of strong shock waves". In: *Journal of Applied Mathematics and Mechanics* 10 (1946), pp. 241–250.

[117] Amar Shan. "Heterogeneous processing: a strategy for augmenting moore's law". In: *Linux Journal* 2006.142 (2006), p. 7.

[118] Ahmad Shawahna, Sadiq M Sait, and Aiman El-Maleh. "FPGA-based accelerators of deep learning networks for learning and classification: A review". In: *ieee Access* 7 (2018), pp. 7823–7859.

[119] James E Smith and Gurindar S Sohi. "The microarchitecture of superscalar processors". In: *Proceedings of the IEEE* 83.12 (1995), pp. 1609–1624.

[120] JEDEC Standard. "Graphics Double Data Rate (GDDR5) SGRAM Standard". In: *JESD212C, February* (2016).

[121] Yingying Tian et al. "Adaptive GPU cache bypassing". In: *Proceedings of the 8th Workshop on General Purpose Processing using GPUS*. 2015, pp. 25–35.

[122] Tor M. Aamodt Timothy G. Rogers Mike O'Connor. *Warp Scheduling Basics*. URL: https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&ved=2ahUKEwjnw4fD1OqEAxXwgf0HHWjJDpoQFnoECBcQAQ&url=https%3A%2F%2Fwww.cs.ucr.edu%2F~nael%2F217-f19%2Flectures%2FWarpScheduling.pptx&usg=AOvVaw2O15IR4nW8Z3IPl12jzOgz&opi=89978449.

[123] Tayler H. Hetherington Tor M. Aamodt Wilson W.L. Fung. *GPGPU-SIM Manual*. URL: http://gpgpu-sim.org/manual/index.php/Main_Page.

[124] Sean J Treichler et al. *Consolidated crossbar that supports a multitude of traffic types*. US Patent 9,098,383. 2015.

[125] Dean M Tullsen, Susan J Eggers, and Henry M Levy. "Simultaneous multithreading: Maximizing on-chip parallelism". In: *Proceedings of the 22nd annual international symposium on Computer architecture*. 1995, pp. 392–403.

[126] Ghent University. *The Quasar Computation System: Quick Reference Manual*. URL: https://quasar.ugent.be/files/doc/CUDA-Constant-Memory.html.

[127] Sébastien Varrette et al. "Management of an academic HPC cluster: The UL experience". In: *2014 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE. 2014, pp. 959–967.

[128] Oreste Villa et al. "Nvbit: A dynamic binary instrumentation framework for nvidia gpus". In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 2019, pp. 372–383.

[129] Dana Vrajitoru. *Embarrassingly Parallel Algorithms*. URL: https://www.cs.iusb.edu/~danav/teach/b424/b424_23_embpar.html.

[130] Lingyuan Wang, Miaoqing Huang, and Tarek El-Ghazawi. "Exploiting concurrent kernel execution on graphic processing units". In: *2011 International Conference on High Performance Computing & Simulation*. IEEE. 2011, pp. 24–32.

[131] Zhenning Wang et al. "Simultaneous multikernel GPU: Multi-tasking throughput processors via fine-grained sharing". In: *2016 IEEE international symposium on high performance computer architecture (HPCA)*. IEEE. 2016, pp. 358–369.

[132] Wikipedia contributors. *Bit-level parallelism — Wikipedia, The Free Encyclopedia*. 2023. URL: https://en.wikipedia.org/w/index.php?title=Bit-level_parallelism&oldid=1174491081.