National Technical University of Athens
School of Electrical And Computer Engineering
Division of Computer Science
Microprocessors and Digital Systems Laboratory

PhD Dissertation

---

# Large-Scale Software Optimization And Micro-Architectural Specialization for Accelerated High-Performance Computing

---

*Authored by Konstantinos Iliakis*

Co-Supervised by Prof. Dimitrios Soudris, NTUA
Dr. Helga Timko, CERN Staff Scientist

February 28, 2022

National Technical University of Athens
School of Electrical And Computer Engineering
Division of Computer Science
Microprocessors and Digital Systems Laboratory

PhD Dissertation

# Large-Scale Software Optimization And Micro-Architectural Specialization for Accelerated High-Performance Computing

*Authored by Konstantinos Iliakis*

Three-member Advisory Committee:
Prof. Dimitrios Soudris
Prof. Kiamal Pekmestzi
Dr. Helga Timko

Approved by the seven-member committee.

| | | |
|---|---|---|
| Dimitrios Soudris<br>Professor NTUA | Kiamal Pekmestzi<br>Professor NTUA | Dr. Helga Timko<br>CERN Staff Scientist |
| Sotirios Xydis<br>Asst. Prof. HUA | Dionisios Pnevmatikatos<br>Professor NTUA | Dr. Danilo Piparo<br>CERN Staff Scientist |
| | Michael Hübner<br>Professor BTU Cotbus, Germany | |

Athens, February 28, 2022

# Copyright Notice



Signature:

---

Date:

---

# *Abstract*

For almost half a century the performance of processors has been improving exponentially, closely following the observations made by Gordon Moore and Robert Dennard. One after the other, both predictions have come to a halt due to the increased complexity in transistor manufacturing, the power and thermal limitations at extremely small-scale technology nodes, and the implications of Amdahl's law to multiprocessing . Nowadays, more than ever, the path to high performance passes through meticulous software optimization, fine tuning, and the design of domain-specific hardware architectures and accelerators.

Within the scope of this thesis, we approach High-Performance Computing from two different standpoints. In the first part of the thesis, we bridge the gap between productivity-oriented, high-level programming languages and high-performance computing techniques. The domain of focus is particle accelerator physics, and more specifically beam dynamics. The state-of-art Beam Longitudinal Dynamics simulator *BLonD* was developed at CERN in 2014, and since then, *BLonD* has been driving the baseline choices for key-parameters related to the daily operation of the largest circular particle accelerators and their upgrades, as well the research for future machines. We develop a single-node optimized, multi-threaded version of *BLonD* to accommodate design-space exploration oriented simulation studies. Then, we build a hybrid, MPI-over-OpenMP version to bring the run-time of previously week-long or even month-long simulations down to a few hours. To achieve that, techniques such as intelligent dynamic load-balancing and approximate computing were employed. Finally, to anticipate the demand for ever-growing simulation workloads, we design a distributed, GPU-accelerated version of the code, which delivers more than two orders of magnitude improved latency and throughput compared to the previous state-of-art. All the above technologies and optimizations are developed in a user-friendly way. The dramatic reduction in execution time enables scientists to simulate beam longitudinal dynamics scenarios that combine more complex physics phenomena with finer resolution and larger number of simulated particles. These complex, accurate and fast simulations are essential in the field of beam dynamics to overcome current technological limitations, plan the upcoming upgrades of particle accelerators, and design future machines that will help science advance further.

The second part of the thesis is focused on hardware customization to accommodate the needs of modern applications. GPUs, once used for the acceleration of graphic workloads, have now become the dominant platform for general purpose application acceleration. Their processing power and cost-efficiency

have led to their adoption in almost every computing domain, including machine learning, scientific computing, and databases. Monitoring the behavior of multiple, GPU-accelerated workloads, the existence of a significant class of kernels was detected, which, due to limited data parallelism fail to support a large degree of Thread-Level Parallelism and hide the latency of memory operations. These kernels seek for more aggressive Instruction-Level Parallelism strategies to improve stall hiding and fill the execution pipeline. This inefficiency is addressed by designing a novel, light-weight Out-Of-Order GPU (LOOG) micro-architecture. LOOG is designed to re-use and re-purpose existing hardware components to minimize the power and area overheads. By exploiting Instruction-Level Parallelism to complement the existing Thread-Level Parallelism execution model, LOOG surpasses both traditional GPU platforms and other prior-art policies. A thorough discussion of LOOG internals and the key design tradeoffs that had to be considered are provided in the thesis. Moreover, an extensive design space exploration is performed to fine tune LOOG and demonstrate its effectiveness when applied on top of a variety of GPU platforms. The LOOG mechanism outperforms conventional platforms by 27.6% and 22.4% in terms of run-time and energy efficiency, respectively. This is a strong indication that LOOG is a promising alternative GPU micro-architecture, which is capable of expanding the applicability of future GPU platforms even further, to new application domains.

To summarize, this thesis proposes two approaches to improve the performance in terms of execution time and energy efficiency, anticipating the ever-increasing computing requirements of modern applications. Firstly, we discuss meticulous software customization to take advantage of existing multiprocessors and hardware accelerators, while providing an easy-to-use interface to the user-base. Secondly, we explore micro-architectural specializations to adjust to the needs of modern workloads.

**Keywords**: High Performance Computing, Distributed Computing, Approximate Calculations, Load Balancing, GPU Micro-Architecture.

# *Περίληψη*

Τον τελευταίο μισό αιώνα η απόδοση των επεξεργαστών αυξάνεται εκθετικά, ακολουθώντας πιστά τις προβλέψεις των Gordon Moore και Robert Dennard. Η μία μετά την άλλη, αμφότερες οι προβλέψεις έχουν σταματήσει να ισχύουν λόγω της αυξημένης πολυπλοκότητας στην κατασκευή τρανζίστορ, των θερμικών περιορισμών και περιορισμών ισχύος σε εξαιρετικά μικρές κλίμακες, καθώς και του νόμου του Amdahl σχετικά με την πολυεπεξεργασία. Σήμερα, περισσότερο από ποτέ άλλοτε, η πορεία για την υψηλή απόδοση συνοδεύεται από σχολαστική βελτιστοποίηση λογισμικού, ακριβή παραμετροποίηση και σχεδιασμό εξειδικευμένων αρχιτεκτονικών και επιταχυντών.

Στα πλαίσια αυτής της διατριβής, εξετάζεται ο Υπολογισμός Υψηλών Επιδόσεων από δύο σκοπιές. Στο πρώτο μέρος της διατριβής, γεφυρώνεται το χάσμα μεταξύ γλωσσών προγραμματισμού υψηλού επιπέδου με κύριο προσανατολισμό την παραγωγικότητα και τεχνικών υψηλής απόδοσης. Ο τομέας εστίασης είναι η φυσική των επιταχυντών σωματιδίων, και πιο συγκεκριμένα της φυσικής που περιγράφει την κίνηση μία δέσμης σωματιδίων μέσα σε κυκλικούς επιταχυντές σωματιδίων. Ο υπερσύγχρονος προσομοιωτής Beam Longitudinal Dynamics *BLonD* αναπτύχθηκε στον Ευρωπαϊκό Συμβούλιο Πυρηνικής Έρευνας (CERN) το 2014, και έκτοτε καθοδηγεί τον καθορισμό βασικών παραμέτρων σχετικά με την καθημερινή λειτουργία των μεγαλύτερων κυκλικών επιταχυντών, τις αναβαθμίσεις τους, καθώς και την έρευνα για μελλοντικά μηχανήματα. Αρχικά, αναπτύξαμε μία βελτιστοποιημένη για ένα κόμβο πολυνηματική έκδοση του *BLonD* για να φιλοξενήσει μελέτες προσομοίωσης προσανατολισμένες στην εξερεύνηση του χώρου σχεδιασμού. Στη συνέχεια, κατασκευάσαμε μια υβριδική, MPI-over-OpenMP έκδοση που επέτυχε την μείωση του χρόνο εκτέλεσης προσομοιώσεων που προηγουμένως διαρκούσαν μια εβδομάδα ή ακόμα και έναν μήνα σε μερικές ώρες. Για να επιτευχθεί αυτό, χρησιμοποιήθηκαν τεχνικές όπως η δυναμική εξισορρόπηση φορτίου και οι προσεγγιστικοί υπολογισμοί. Τέλος, λόγω της συνεχούς ζήτησης για συνεχώς αυξανόμενα μεγέθη προσομοίωσης, σχεδιάζουμε μια κατανεμημένη, επιταχυνόμενη με Μονάδες Επεξεργασίας Γραφικών (GPU) έκδοση του κώδικα, η οποία παρέχει περισσότερες από δύο τάξεις μεγέθους βελτιωμένη απόδοση σε σύγκριση με την προηγούμενη έκδοση. Όλες οι παραπάνω τεχνολογίες και βελτιστοποιήσεις αναπτύχθηκαν με φιλικό τρόπο προς τον χρήστη, ώστε να χρειάζονται ελάχιστες επεμβάσεις για την χρήση της κατανεμημένης ή GPU έκδοσης. Η δραματική μείωση του χρόνου εκτέλεσης επιτρέπει στους επιστήμονες να προσομοιώνουν σενάρια που συνδυάζουν πιο πολύπλοκα φυσικά φαινόμενα, με πιο λεπτομερή ανάλυση και μεγαλύτερο αριθμό σωματιδίων. Αυτές οι πολύπλοκες, ακριβείς και γρήγορες προσομοιώσεις είναι

απαραίτητες στον τομέα της φυσικής της κίνησης των δεσμών σωματιδίων σε κυκλικούς επιταχυντές για να ξεπεραστούν οι τρέχοντες τεχνολογικοί περιορισμοί, να υλοποιηθούν οι επερχόμενες αναβαθμίσεις των επιταχυντών σωματιδίων και να σχεδιαστούν μελλοντικές μηχανές που θα βοηθήσουν την επιστήμη να προοδεύσει περαιτέρω.

Το δεύτερο μέρος της διπλωματικής εργασίας επικεντρώνεται στην προσαρμογή και εξειδίκευση του υλικού με σκοπό να καλύψει τις ανάγκες των σύγχρονων εφαρμογών. Οι GPU, που κάποτε χρησιμοποιούνταν για την επιτάχυνση του εφαρμογών γραφικών, έχουν γίνει πλέον η κυρίαρχη πλατφόρμα για την επιτάχυνση εφαρμογών γενικού σκοπού. Η επεξεργαστική τους ισχύς, σε συνδυασμός με την υψηλή ενεργειακή αποδοτικότητα οδήγησαν στην υιοθέτησή τους σε σχεδόν κάθε υπολογιστικό τομέα, συμπεριλαμβανομένης της μηχανικής μάθησης, του επιστημονικών εφαρμογών και των βάσεων δεδομένων. Παρακολουθώντας την συμπεριφορά πολλαπλών εφαρμογών που χρησιμοποιούν GPU, εντοπίστηκε μια σημαντική συλλογή συναρτήσεων, οι οποίες, λόγω περιορισμένου παραλληλισμού δεδομένων, δεν υποστηρίζουν μεγάλο βαθμό παραλληλισμού νημάτων (Thread-Level Parallelism, TLP) και δεν καταφέρνουν να κρύψουν την καθυστέρηση των λειτουργιών μνήμης. Αυτές οι συναρτήσεις αναζητούν πιο επιθετικές στρατηγικές παραλληλισμού επιπέδου εντολών (Instruction-Level Parallelism, ILP) για τη βελτίωση της χρησιμοποίησης των πόρων κατά την εκτέλεση. Αυτή η προβληματική κατάσταση αντιμετωπίζεται με το σχεδιασμό μιας νέας μικροαρχιτεκτονικής GPU με δυνατότητα εκτέλεσης εντολών εκτός σειράς Out-Of-Order, που ονομάζεται LOOG. Το σύστημα LOOG έχει σχεδιαστεί ώστε να επαναχρησιμοποιεί υπάρχοντα εξαρτήματα υλικού και να ελαχιστοποιεί τα επιπρόσθετα κόστη σε ισχύ και μέγεθος επιφάνειας κυκλώματος. Με την εκμετάλλευση του Παραλληλισμού Επιπέδου Εντολών, το οποίο συμπληρώνει το υπάρχον μοντέλο εκτέλεσης Παραλληλισμού Επιπέδου Νήματος, το LOOG ξεπερνά τόσο τις παραδοσιακές πλατφόρμες GPU όσο και άλλες προγενέστερες υλοποιήσεις υψηλών επιδόσεων. Μια διεξοδική συζήτηση των λεπτομερειών του LOOG και των βασικών σχεδιαστικών αποφάσεων που έπρεπε να ληφθούν υπόψη παρέχονται στη διατριβή. Επιπλέον, εκτελείται μια εκτεταμένη εξερεύνηση του χώρου σχεδίασης για να ρυθμιστεί με ακρίβεια το LOOG και να αποδειχθεί η αποτελεσματικότητά του όταν εφαρμόζεται πάνω από μια ποικιλία από παραδοσιακές πλατφόρμες GPU. Ο μηχανισμός LOOG ξεπερνά τις συμβατικές πλατφόρμες κατά 27.6% και 22.4% όσον αφορά τον χρόνο εκτέλεσης και την ενεργειακή απόδοση, αντίστοιχα. Αυτή είναι μια ισχυρή ένδειξη ότι το LOOG είναι μια πολλά υποσχόμενη εναλλακτική μικροαρχιτεκτονική GPU, η οποία είναι ικανή να επεκτείνει την χρήση μελλοντικών συστημάτων GPU σε νέους τομείς εφαρμογών.

Συνοψίζοντας, αυτή η διατριβή προτείνει δύο προσεγγίσεις για τη μείωση του

χρόνου εκτέλεσης και της κατανάλωσης ενέργειας, προσαρμοσμένες στις συνεχώς αυξανόμενες υπολογιστικές απαιτήσεις των σύγχρονων εφαρμογών. Πρώτον, εστιάζουμε στην σχολαστική προσαρμογή λογισμικού για να αξιοποιήσουμε τους υπάρχοντες πολυεπεξεργαστές και επιταχυντές υλικού, παρέχοντας παράλληλα μια εύχρηστη διεπαφή στους χρήστες. Δεύτερον, διερευνούμε εξειδικευμένες μικροαρχιτεκτονικές με σκοπό την προσαρμογή στις ανάγκες των σύγχρονων εφαρμογών.

**Λέξεις Κλειδιά**: Υπολογισμός Υψηλών Επιδόσεων, Κατανεμημένη Επεξεργασία, Προσεγγιστικοί Υπολογισμοί, Δυναμική Εξισορρόπηση Φόρτου Εργασίας, Μικροαρχιτεκτονική Μονάδων Επεξεργασίας Γραφικών.

# *Acknowledgements*

This PhD dissertation concludes my post-graduate studies and research activities at the school of Electrical and Computer Engineering of the National Technical University of Athens.

I would like to express my profound and sincere thanks to Professor Dimitrios Soudris for giving me the opportunity to carry out my research under his supervision. His guidance and advises, alongside with his invaluable research experience provided me with motivation and inspiration to accomplish this work. I really appreciate the leadership qualities I was taught by his collaboration and supervision.

Dr. Helga Timko supervised my 3-year scholarship at CERN, and I had the chance to collaborate close with her during the whole duration of this PhD thesis. I deeply thank her for being consistently available to provide me with invaluable advises and actionable ideas. She encouraged me to expand my knowledge and skills in new areas, gave me enough space to develop my own ideas and initiatives, and gently suggested corrective actions when necessary. Finally, during our collaboration she taught me important lessons about effective teamwork and interdisciplinary cooperation.

In addition, I would like to thank Asst. Professor Sotirios Xydis, who advised and stood by me throughout the duration of this thesis with accurate observations and suggestions. He was always willing to support me in every difficulty I faced and share his knowledge with me. I feel grateful for having the opportunity to cooperate with him.

I am grateful to every member of the jury for participating in the defense of this thesis and provided me with insightful remarks and constructive feedback that improved the quality of this research work.

Furthermore, I am thankful to all my colleagues at the SY-RF (former BE-RF) group, CERN and Microlab, ECE, NTUA. The joyful and supporting office atmosphere they created and the lengthy discussions we had, motivated and shaped the implementation of some key research components included in this thesis.

I give my special thanks to all my friends who supported me throughout my studies. Finally, I like to thank my family who always encouraged and assisted me to achieve my goals.

# Contents

# List of Figures

# Chapter 1

# Introduction

## 1.1 Computer & Software Design in the Post-Moore's Law Era

Gordon Moore in 1965 made the observation that the number of transistors in dense integrated circuits doubles about every two years, and that there is no reason to believe that this rate will not remain fairly constant for at least the coming ten years [1]. Since then, Moore's prediction has been used in the semiconductor industry to drive long-term planning and set targets for research and development, thus functioning to some extent as a self-fulfilling prophecy. Indeed, looking at Fig. 1.1 that shows in a logarithmic scale the number of transistors of computer systems since 1970, we can see that the transistor count has been increasing exponentially for almost 50 years.

For many years, the increase in transistors had been accompanied with a similar increase in performance. In Fig. 1.2, we can see the processors' performance growth over a period of 40 years, relative to the VAX 11/780 processor [3]. The performance is measured by the SPEC CPU integer benchmark [4, 5, 6]. During the first 25 years, the growth in performance was mainly technology-driven, delivering performance improvement of about 25% per year.

The gradual improvement in performance per dollar led to the emergence of new classes of computers, such as personal computers and workstations. Additionally, improved semiconductor manufacturing guided the dominance of microprocessor-based computers across the entire range of computer design. These hardware innovations drove a renaissance in computer design, which emphasized both in architectural innovation and efficient use of technology improvements. The combination of computer design innovations and technological improvements resulted in an average 52% yearly performance improvement rate from 1986 to 2003.

Computer architecture is the organization of the components of a computer

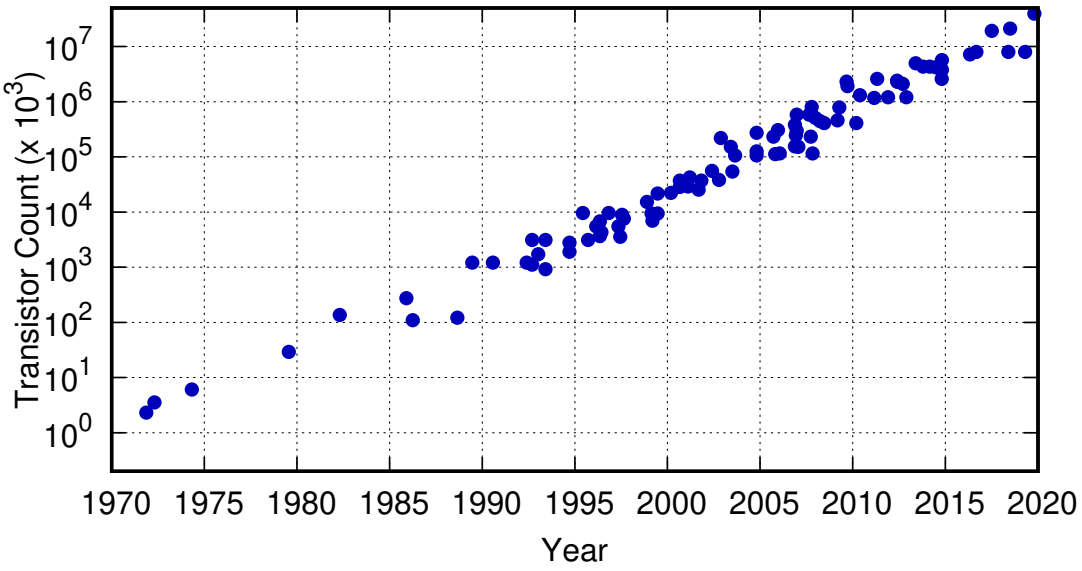



FIGURE 1.1: Evolution of transistor count in CPUs since 1970. Data source [2].

and the semantics of the operations that guide its function. The computer architecture governs the design of a family of computers and defines the logical interface that is used by programming languages and compilers. The organization determines the mix of functional units and their interconnectivity. The architecture semantics is the meaning of what the systems do under user direction and how their functional units are controlled to work together. The instruction set architecture (ISA) of the system represents the basic set of operations that a given architecture can perform. Compilers and interpreters are used to convert higher-level user programs into a sequence of such basic operations. This is true for all kinds of computers, from those in mobile phones and embedded devices to those making up the world's largest supercomputers. High-performance computer architecture is about designing and organizing computers specifically to deliver computational speed.

The third time period of Fig. 1.2 recorded an average annual performance improvement rate of 23%, and is characterized by what is referred to as the end of Dennard scaling [8]. In 1974 Robert Dennard observed that the power density of a given silicon area remained constant when increasing the number of transistors, because of the smaller dimensions of each transistor. This meant that processors could integrate more transistors without using more power. Dennard scaling ended around 2004. The primary reason cited for the breakdown is that at small sizes, current leakage poses greater challenges and also causes the chip to heat up, which creates a threat of thermal runaway and therefore further increases energy costs. This change forced the microprocessor industry to use multiple, small to medium size processors or cores, instead of a large

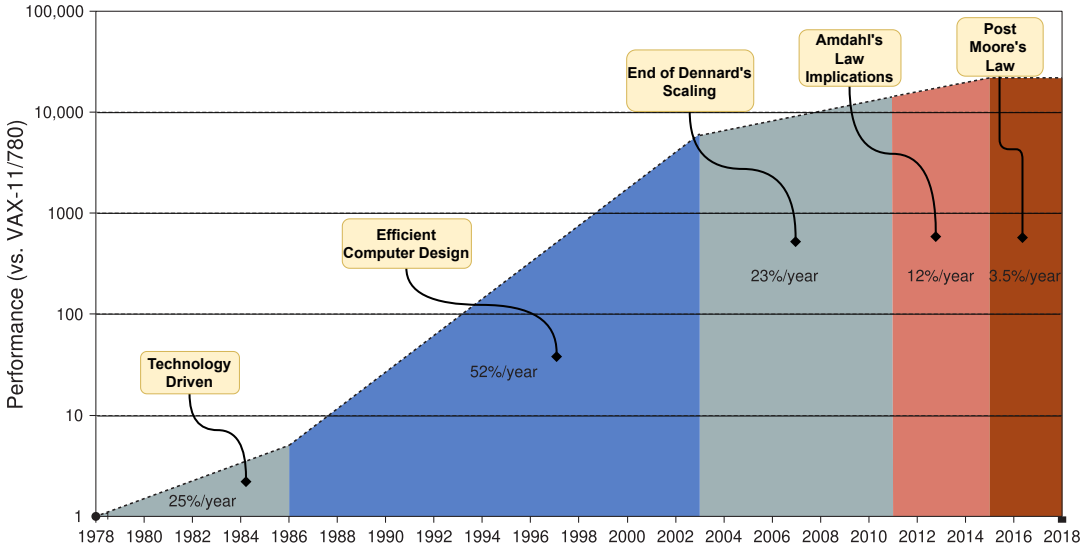



FIGURE 1.2: Performance of processors over 40 years according to the SPEC CPU benchmark [7].

monolithic processor. From 2004 and onward, the road to high-performance is via multi-core processors rather than via faster uniprocessors. This milestone signaled a switch from relying solely on instruction-level parallelism (ILP), to data-level parallelism (DLP) and thread-level parallelism (TLP).

The period from 2011 to 2015, the annual improvement rate dropped to around 12% per year. This is mostly attributed to the implications of Amdahl's Law [9]. Amdahl's law puts a theoretical upper bound to the maximum speedup that can be extracted from using multiple cores, and can be formulated as follows:

$$\text{Speedup} = \frac{1}{(1-P) + \frac{P}{N}} \tag{1.1}$$

where P is the proportion of the execution time that is parallelized and N is the number of cores. For example, if 10% of a program is serial, the speedup from parallelization cannot be more than 10 independently of the number of available cores in the system.

After Dennard's scaling, Moore's law has also come to an end recently. The combination of extreme complexity in transistor manufacturing, the limited power budget of modern processors, and Amdahl's law implications to multi-processing has caused the processors performance to improve at a rate of 3.5% per year, that is to double every two decades instead of every two years, as originally predicted by Moore.

The 50,000-fold performance improvement since 1978 (see Fig. 1.2) had a major impact in software development, since it allowed modern programmers to trade performance for productivity. In place of performance-oriented languages like C and C++, much more programming today is done in managed

programming languages like Java and Scala. Moreover, scripting languages like JavaScript and Python, which are even more productive, have gained in popularity. To maintain productivity and try to close the performance gap, interpreters with just-in-time compilers [10] are replacing the traditional compilers and linkers of the past. Software deployment is changing as well, with Software as a Service [11] (SaaS) used over the Internet replacing software that must be installed and run locally.

Living at the end of the line of the rapid performance improvement that lasted for half a century, there are no more low-hanging fruits. Nowadays, meticulous software optimization, fine tuning and hardware customization are more prevalent than ever before. In the scope of this thesis, we approach High-Performance Computing from two different viewpoints. At first, we bring cutting-edge high-performance computing techniques to the field of longitudinal beam dynamics, by entirely restructuring the state-of-art simulator library to match the hardware's features and take advantage of distributed computing and accelerator platforms. A brief outline of the steps taken towards the direction of high-performance software development can be seen in the top part of Fig. 1.3. Then, in the second part of the thesis we focus on architectural customization instead. Specifically, we target GPUs – the most prevalent platform for performance acceleration. We discuss, suggest and explore a novel, non-conventional GPU architecture paradigm, better tailored to the requirements of modern, general-purpose GPU accelerated workloads. The development steps, that will be discussed and evaluated in detail in the second part of this thesis are shown in the bottom of Fig. 1.3.

## 1.2 High-Performance Computing Strategies for Longitudinal Beam Dynamics

The first part of this thesis (see Fig. 1.3) bridges the gap between user-friendly scientific software and high-performance computing. Typically, scientific software is intended to be used by non-computer scientists. A high-level scripting language such as Python is widely popular among mathematicians, physicists, chemists, biologists, geologists, and others for simulation studies and data analysis. A scripting language like Python, allows for rapid development and prototyping, and seamless integration with powerful third-party libraries. However, high-performance software is usually developed in a compiled language such as Fortran, C and C++. In the scope of this thesis, we focus on the domain of beam dynamics, the field of physics that studies the beam motion in circular particle accelerators, also called synchrotrons. As shown later in Sec. 2,

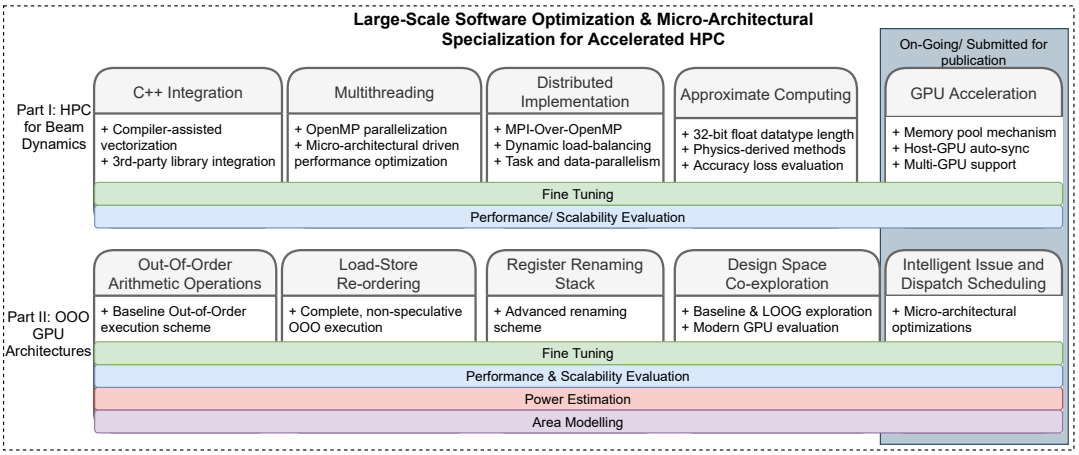



FIGURE 1.3: Thesis High-Level Organization.

the state-of-art Beam Longitudinal Dynamics code *BLonD*, was developed at CERN since 2014 by a team of experienced accelerator physicists in Python.

The first step to extract better single-node performance while maintaining an easy-to-use interface, was to combine Python with a C/C++ computational back-end. Chapter 4 shows how this was achieved, and in addition, how the computational core was able to effectively exploit modern processors' resources as well as intra-node, vertical scaling with the aid of the OpenMP [12] parallel programming framework. The single-node optimized version of *BLonD* is called *BLonD++*, and managed to provide on average 18× single-core speedup compared to the original Python-only *BLonD* version.

In beam dynamics, similarly to many other scientific domains, simulation studies may vary in terms of simulation count, i.e. the total number of simulations that need to be performed, and workload size. Figure 1.4 shows a qualitative diagram of the two-dimensional simulation space of *BLonD*, and how the different versions of the code that were developed in the scope of this thesis are better tailored for different simulation space regions. Below we explain the most typical simulation study scenarios, that motivated us to develop the different variations of *BLonD*:

1. The user is interested in scanning a parameter space of tens, hundreds or even thousands of slightly different configured simulations, to finely tune a set of input parameters. Since these distinct simulation runs are independent of each other, they can all run in-parallel given enough computing resources. Therefore, this type of studies is better tailored for efficient vertical-scalability to take full advantage of the available computing hardware. This is the purpose of the *BLonD++* code.

2. The user needs to examine certain phenomena with the highest possible accuracy, therefore a small set of large scale simulations has to be run.

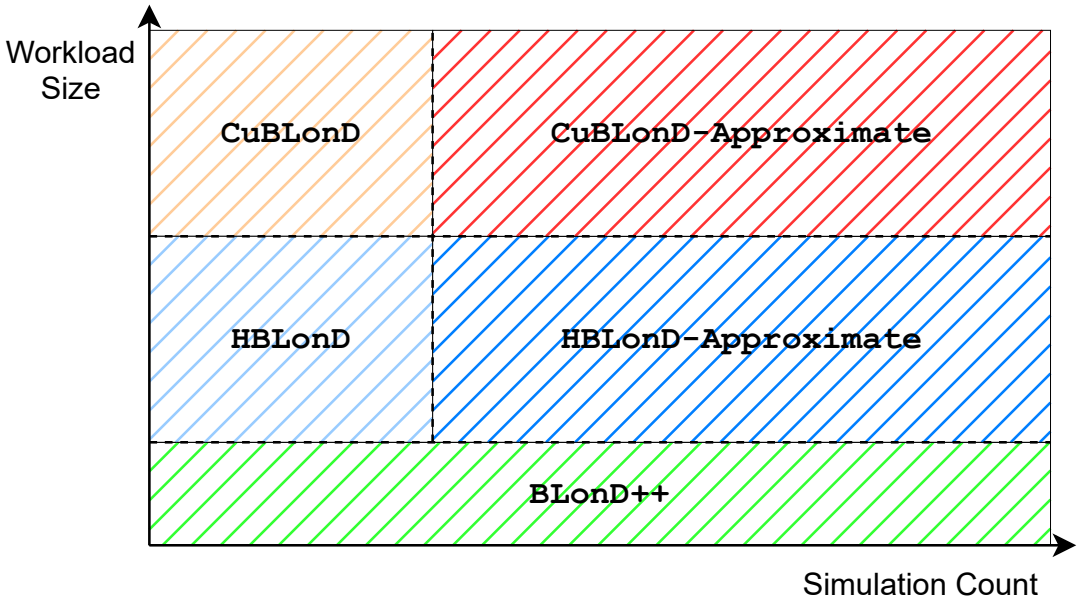



FIGURE 1.4: The three versions of *BLonD* developed in the scope of this thesis, and their recommended usage region.

These simulations can be very demanding in terms of execution time. In the case of *BLonD*, some simulations could take several weeks or even months to finish, making them impractical given that the outcome of the studies is needed to guide the on-going operation of real experiments. This type of studies can be only accommodated by horizontally scaling software, meaning that the simulation software must be capable of combining multiple computing nodes to cooperatively calculate a large-scale simulation in a shorter time frame. In Chapter 5, we will see the architecture of *HBLonD*, the MPI-over-OpenMP distributed implementation of *BLonD*, that is able to scale in more than 30 computing nodes and 600 cores. With the aid of a dynamic load balancing scheme and various approximate computing methods, *HBLonD* achieves an up to 58× faster execution time compared to a single-node instance of *BLonD++*.

The optimization of the operation of existing circular particle accelerators, the upgrade projects, and the design and research for future machines drive the need for larger, longer and more simulation studies. To anticipate this ever-growing need for computationally more and more demanding simulations, in our on-going work in Chapter 6 we combine MPI with the CUDA programming language to build a distributed, GPU-accelerated code for beam longitudinal dynamics, called *CuBLonD*. *CuBLonD* is designed with usability in mind, hence the Python front-end has been kept intact. Using an efficient, zero-overhead Python library, PyCUDA [13], we offload the most compute-intensive code regions to the GPU accelerator. Furthermore, a memory pool mechanism was

implemented to minimize memory allocation and de-allocation overheads. A CPU-GPU memory synchronization mechanism was developed to provide the user the view of a unified memory space with minimal performance overhead. *CuBLonD* delivers up to two orders of magnitude speedup w.r.t. *BLonD++*, using 32 GPU accelerators in 16 computing nodes. The dramatic reduction in execution time that is achieved by the highly optimized *BLonD++* and *HBLonD* codes, have enabled scientists to simulate beam longitudinal dynamics scenarios that combine more complex physics phenomena with finer resolution and larger number of simulated particles. These complex, accurate and fast simulations are essential in the field of beam dynamics to overcome current technological limitations, plan the upcoming upgrades of particle accelerators, and design future machines that will help science advance further.

## 1.3 Non-Conventional General-Purpose GPU Architectures

While the first part of this thesis discusses how to bridge the gap between high performance computing and scientific software simulators, the second part moves further towards the direction of accelerator platforms and hardware specialization. More specifically, it focuses on novel, non-traditional Graphics Processing Unit (GPU) accelerators. GPUs are nowadays the primary platform for general-purpose workload generation. Despite originally being designed for graphics processing in video games and other visual applications, over time, GPU programming for general-purpose applications became more practical. Their processing power and cost-efficiency have led to their adoption in a wide spectrum of computing domains, including among others machine learning [14, 15], scientific computing [16], and databases [17, 18].

By meticulously monitoring the characteristics of modern, GPU-accelerated workloads, we observe that certain classes of kernels, due to limited data parallelism, fail to support a large degree of Thread-Level Parallelism (TLP) and hide the latency of memory operations. Thus they suffer from excessive stalling time and sub-optimal resource utilization. These "irregular" kernels cannot effectively exploit the traditional TLP model due to low warp occupancy, and require more aggressive Instruction-Level Parallelism (ILP) strategies to improve stall hiding, tolerate periods of insufficient TLP, and provide the back-end with a flow of instructions even in the absence of a large number of active thread contexts. We address the aforementioned inefficiencies found in typical GPUs by re-purposing GPU micro-architectures towards a general-purpose, dynamic, Light-weight Out-of-Order GPU (LOOG) execution scheme, carefully designed

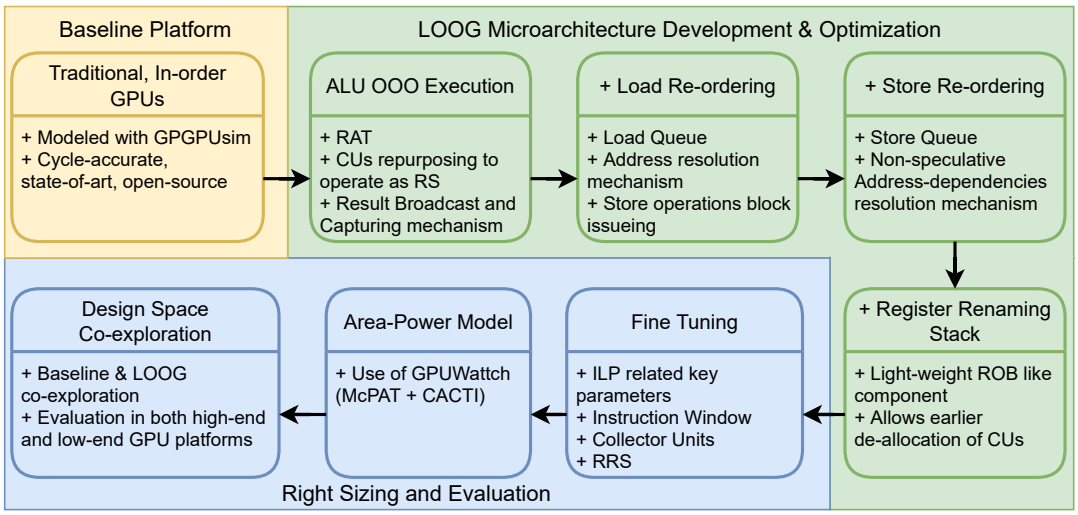



FIGURE 1.5: Development, Optimization and Evaluation methodology followed in the second part of this thesis.

to minimize the hardware overhead. LOOG surpasses prior state-of-art by exploiting ILP to complement the existing TLP and improve resource utilization of underperforming kernels.

Figure 1.5 shows the sequence of development, optimization, fine tuning and evaluation steps that are discussed in this thesis, regarding the Out-Of-Order execution for GPUs paradigm, LOOG. In Chapter 9, we present the micro-architectural mechanisms we implemented in order to enable Out-Of-Order execution in GPUs. A series of design tradeoffs had to be considered to achieve a balance between run-time and energy efficiency. The first LOOG implementation could re-order only arithmetic and not memory operations. However, this pose a great performance limiting factor. Therefore, with the introduction of a Load-Store Queue, we enabled the re-ordering of both arithmetic and memory operations. Then, we observed that the LOOG architecture was sensitive to the number of Collector Units, since they are used as reservation stations to implement register renaming. The addition of a light-weight structure, named Register Renaming Stack (RRS), managed to resolve this issue in a cost-efficient way, and resulted in the LOOG architecture being on average

In Chapter 10, we study the sensitivity of LOOG to key micro-architectural parameters that affect the Instruction-Level Parallelism potential. Furthermore, we perform an extensive design space exploration of traditional GPU architectures and the LOOG mechanism, to show that LOOG is capable of providing superior performance and energy efficiency when combined with both low-end and high-end GPU platforms. Finally, in Chapter 11, the evaluation of the fully optimized LOOG mechanism against conventional GPU architectures is presented, showing that LOOG offers 27.6% and 22.4% run-time and energy

efficiency, respectively. In conclusion, LOOG is a promising alternative GPU micro-architecture that can expand the applicability of future GPU platforms even further to new application domains.

## 1.4 Thesis Structure

The remainder of this thesis is organized in two parts. The first part, titled *High Performance Computing Strategies Applied to Longitudinal Beam Dynamics*, is organized as follows:

1. Chapter 2 provides a generic introduction to the role and scientific mission of CERN and highlights the need for computational beam dynamics studies.

2. A brief overview of the field of Longitudinal Beam Dynamics and the *BLonD* simulator is found in Chapter 3.

3. Chapter 4 discusses *BLonD++*, the single-node, optimized and multi-threaded version of *BLonD*.

4. In Chapter 5, we enable for the first time scale-out beam longitudinal dynamics simulations and provide the respective experimental evaluation.

5. Chapter 6 describes *CuBLonD*, the GPU accelerated version of *BLonD*, that provides superior runtime performance and scalability, and makes use of modern GPU architectures.

The second part of the thesis, titled *Towards Non-Conventional GP-GPU Micro-Architectures*, is outlined below:

1. Chapter 8 reviews related work on performance optimization strategies for GPUs, and provides a brief synopsis of the baseline GPU execution model and micro-architecture pipeline.

2. The in-depth implementation details of LOOG are discussed in Chapter 9.

3. The key design trade-offs that aroused during the development of LOOG and a thorough LOOG-oriented design space sensitivity analysis are discussed in Chapter 10.

4. The experimental evaluation in terms of run time performance and energy efficiency of LOOG compared to conventional GPU architectures takes place in Chapter 11.

Finally, Chapter 12 concludes this thesis.

# Part I

# High-Performance Computing Strategies for Longitudinal Beam Dynamics

# Chapter 2

# CERN's Role & Necessity for HPC Beam Dynamics Studies

## 2.1 CERN's Scientific Mission[1]

At CERN, physicists use large particle accelerators to accelerate and then collide high-energy particles in order to study the fundamental laws of particle physics. The Large Hadron Collider (LHC), operational since 2008, is currently the world's largest and most powerful particle accelerator and the latest addition to CERN's accelerator complex. The Standard Model of particle physics describes the weak, strong, and electromagnetic forces, but it does not describe gravity. In 1964, the Higgs boson was introduced by various scientists into the Standard Model in order to add a mechanism that explains the mass of particles [19, 20, 21, 22, 23]. The discovery of the Higgs boson in 2012 in the LHC machine and its two broad-purpose detectors ATLAS [24] and CMS [25] is one of the greatest discoveries of the LHC so far. Presently, the biggest challenge is to obtain hints for physics beyond the Standard Model, such as theories including gravity, the observed matter-antimatter asymmetry in nature, and the conservation of charge and parity in strong interactions, just to mention a few.

The higher-energy particle accelerators at CERN are circular machines, so-called synchrotrons. Charged particles are accelerated through a chain of synchrotrons, also called injectors, to increasingly higher energies, before being injected into two separate beam pipes to collide in the final synchrotron, also called collider. In the LHC, the two beams circulate in opposite directions and intersect at fixed locations in the heart of large particle detectors that can record and then analyse the collision traces. ATLAS [24] and CMS [25], two of the largest particle detectors on earth are two general-purpose detectors of the eight LHC experiments.

[1] The complete mission statement can be found at: home.cern/about/who-we-are/our-mission

CERN is not only exploiting its present facilities to a maximum potential, but it also participates in a wide range of accelerator physics R&D projects to prepare for the future. The accelerator physics community convenes regularly in forums like the European Strategy for Particle Physics [26, 27] and sets the priorities among different R&D projects. Discussed are so-called search machines that can probe a wide range of energies, and precision machines (so-called Higgs factories) that can probe the properties of the Higgs boson with high precision; both in order to explore physics and various theories beyond the Standard Model.

## 2.2 Beam Dynamic Studies Necessity

The LHC Injector Upgrade (LIU) [28], the upcoming High-Luminosity LHC project [29], and the studies of future machines such as the Future Circular Collider (FCC) [30, 31, 32] and the Compact Linear Collider (CLIC) [33, 34] are CERN's most important R&D projects at present. Despite of vast experience with the LHC and its injectors, upgraded and future machines cannot simply be scaled in size and energy to achieve the desired beam energy and intensity. Some limitations in future machines are known by design, but studies are required to explore previously unknown limitations, too. The studies are often very complex and require precision modelling in the domain of accelerator physics and specifically also beam dynamics simulations that can model the relevant, detailed physics phenomena and machine-specific features.

The simulator software has to be flexible enough to include a wide range of synchrotrons, energy regimes and particle types. To fulfill these critical requirements, the Beam Longitudinal Dynamics simulation suite (*BLonD*) [35, 36] was developed at CERN since 2014. As its name suggests, the field of longitudinal beam dynamics focuses on the longitudinal motion of the beam particles, and *BLonD* tracks the energy and time coordinates of beam particles in synchrotrons. It features a modular structure that allows the user to focus on different physics phenomena and combine different physics modules according the study requirements. *BLonD* is an open-source, cross-platform project that is increasingly gaining popularity among the world's largest accelerator laboratories.

Through an extensive range of applications [37, 38, 39, 40, 41, 42, 43, 44] and benchmarks [45] for beam dynamics studies, the confidence in the *BLonD* suite has grown among the scientific community. The benchmarks performed range from comparisons with theory and measurements, to other particle tracking codes and increased the trust in *BLonD* and its predictions. The outcome of

the simulation studies is continuously guiding the baseline choices for machine upgrades and future machines [42, 43]. In many cases the existing machines are being pushed beyond their design limits while the upgraded systems are designed with minimal margins in order to be as cost-efficient as possible. Thus, simulations need to be very accurate despite the complexity of the machines. Also, whenever new phenomena in operational machines are discovered, it is crucial to have a tool that can reproduce and explain observations.

## 2.3 High-performance Computing Simulations

Simulations of high-energy particle physics inside circular accelerators comprise a scientifically and computationally challenging task. These workloads are typically inherently parallel and fit naturally in a distributed-memory, weak scaling environment using MPI. Furthermore, strong-scaling within a node is also essential. Intra-node scaling and efficient utilization of the hardware resources is especially useful in studies that require large parameter scans. However, high-precision studies require simulating an extremely large number of macro-particles, that can take weeks or even months to execute. As a result, a hybrid, distributed-shared programming model combined with hardware-specific code tuning is required to squeeze every drop of performance out of modern CPUs. Optimizing and tuning scientific codes, while aided by compiler features, has become a highly manual endeavor that requires integrating awareness of the application specific features and the underlying system architecture: the memory subsystem, the execution pipeline, the concurrent threads, and the SIMD vector units [46].

To undertake this agenda, in this thesis we design, optimize and evaluate *HBLonD*, a hybrid, multi-parallel system for running large scale longitudinal beam dynamics simulations. Since the user-base of *BLonD* is mainly non-computer scientists, all the HPC techniques are integrated in the code in a completely transparently to the user way. We initially focus on improving intra-node performance of *BLonD*. We perform micro-architecture motivated optimizations by combining the TMAM [47] and Roofline model [48, 49] analyses. We identify and parallelize the computational core of *BLonD* using the OpenMP framework [12]. Then, we enable scale-out simulations by combining the OpenMP implementation with MPI. To minimize the inter-node communication and synchronization overhead, we apply various software optimisation techniques. At first, we develop a mixed data- and task-parallel model. Data parallelism is used across the nodes and task parallelism is used among

the intra-node MPI processes to profit from fast shared-memory communication. Then, we discuss and evaluate two traffic optimisation and relaxed synchronisation techniques, motivated by the modelled physics phenomena. We study the accuracy-performance trade-off when using those traffic optimization techniques, as well as single versus double floating point precision arithmetic. To deal with various sources of load imbalance, we develop a dynamic load-balancing (DLB) scheme that periodically re-distributes the workload to ensure that all the worker processes progress at the same rate. Finally, to satisfy the ever-growing need for larger workloads and longer simulation periods, guided by the research for future synchrotrons and the upgrades of the existing ones, we combine CUDA and MPI to build a GPU-accelerated, distributed version of *BLonD*, called *CuBLonD*. We evaluate *CuBLonD* in a GPU-enabled supercomputing infrastructure and demonstrate greater than two orders of magnitude speedups when using 16 or 32 GPU platforms.

# Chapter 3

# Background

## 3.1 Longitudinal Beam Dynamics in Synchrotrons

Beam dynamics is the field of physics that models the beam motion inside particle accelerators. Longitudinal beam dynamics focuses on the longitudinal plane of motion, i.e. the motion alongside the beam pipe, in contrast to transverse beam dynamics that focuses on the transverse plane, in the cross-section of the beam pipe. In this thesis, we will restrict the to longitudinal beam dynamics in synchrotrons, which the *BLonD* simulator suite is designed to simulate.

Synchrotrons are circular particle accelerators, in which Radio-Frequency (RF) cavities accelerate and magnets bend them beam; the frequency of the cavities and the bending field of the magnets are synchronized at any time such that charged particles are circulating more or less at a fixed reference orbit, independent of their energy. The angular RF frequency $\omega_{\mathsf{RF}}$ is therefore usually a harmonic of the angular revolution frequency $\omega_0$,

$$\omega_{\mathsf{RF}} = h\omega_0 \,, \tag{3.1}$$

where $h$ is an integer. A particle circulating on the synchronous orbit has the synchronous (relativistic) energy of $E_s$. If in addition the particle is arriving to the RF cavity exactly at the synchronous RF phase $\phi_s$, it will be perfectly on orbit also in the next turn; such a particle is referred to as the 'synchronous particle'.

The term 'beam' usually refers to the collection of all the particles of a given species of charged particles in the accelerator. In the presence of RF voltage, the beam is condensed around the centres of the RF potential wells, see Fig. 3.1, forming a train of several 'bunches'. Beam particles are described in a 2D phase space, typically by the longitudinal coordinate and its conjugate variable. Also, beam coordinates are usually described relative to the coordinates of the synchronous particle. A widely-used phase-space coordinate pair is $(\phi, \Delta E/\omega_0)$,

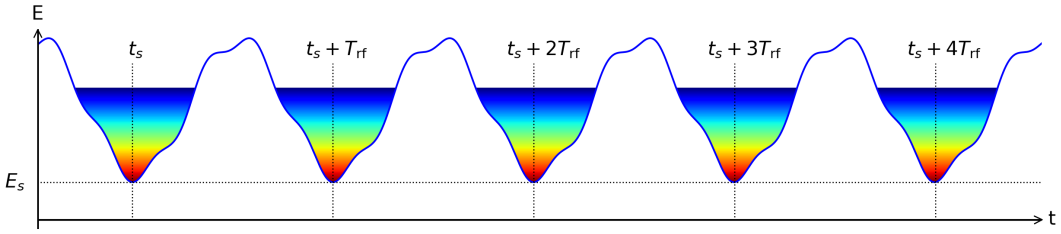



FIGURE 3.1: Example of a periodic potential well in a synchrotron, for several RF systems. In the absence of intensity effects, the synchronous point $(t_s, E_s)$ is periodic with $T_{\mathrm{rf}}$ along the ring. The beam particles fill part of the potential well.

where $\phi$ is the phase coordinate w.r.t. RF voltage wave and $\Delta E$ is the relativistic energy of the particle w.r.t. the energy $E_s$ of the synchronous particle. The longitudinal equations of motion can then be derived, for a single-RF system, to be [50]

$$\frac{d\phi}{dt} = \frac{h\omega_0^2\eta}{\beta^2 E}\left(\frac{\Delta E}{\omega_0}\right) \tag{3.2}$$

and

$$\frac{d}{dt}\left(\frac{\Delta E}{\omega_0}\right) = \frac{eV}{2\pi}(\sin\phi - \sin\phi_s)\,, \tag{3.3}$$

where $e$ is the unit charge, $V$ is the RF voltage amplitude and $\beta = v/c$ is the relativistic beta of the particle, with $v$ being the velocity and $c$ being the speed of light. The slippage factor $\eta$ is a property of the synchrotron, which describes how much RF phase slippage a particle will undergo after one turn with a given energy offset of $\Delta E$. With multiple RF systems installed in the machine, several voltage terms appear on the right-hand-side of Eq. 3.3.

In the above equations, all particles are moving independently in the machine. However, since the beam is a collection of charged particles, and the accelerator is composed of devices that are not perfect conductors, the beam particles can interact with its surroundings electromagnetically. This also leads to trailing particles being affected by leading particles, which we call collective effects. A given particle experiences thus an induced voltage by itself and the particles in front. If the overall impedance of the machine is $Z(\omega)$, we can define a corresponding wake field $W(t)$ as the Fourier transform of the impedance,

$$W(t) = \frac{1}{2\pi}\int_{-\infty}^{+\infty} d\omega e^{i\omega t} Z(\omega)\,. \tag{3.4}$$

A physical wake field is zero in front of the particle, $W(t > 0) = 0$. Denoting the longitudinal line density (or beam profile) of the particles with $\lambda(t)$, a particle

at the coordinate $\Delta t$ will experience the induced voltage

$$V_{\text{ind}}(\Delta t) = -q\, N_p \int_{-\infty}^{+\infty} \lambda(\tau)\, W(\Delta t - \tau)\, d\tau\,, \tag{3.5}$$

where $q$ is the particle charge, $N_p$ is the number of (real) particles in the beam and the profile is normalized to $\int_{-\infty}^{+\infty} \lambda(t) = 1$. Equivalently, the induced voltage can be calculated as the product of the beam spectrum $\Lambda(\omega) = \int_{-\infty}^{+\infty} \lambda(t)\, e^{-i\omega t} d\omega$ and the impedance,

$$V_{\text{ind}}(\Delta t) = -\frac{q\, N_p}{2\,\pi} \int_{-\infty}^{+\infty} Z(\omega)\, \Lambda(\omega)\, e^{i\omega \Delta t} d\omega\,. \tag{3.6}$$

The induced voltage is applied to the particles as an additional energy kick $E_{\text{ind}}(\Delta t) = q\, V_{\text{ind}}(\Delta t)$ added to Eq. 3.3. In a similar manner, energy loss due to synchrotron radiation, or any other phenomena affecting the particle energy, can also be added to the kick equation.

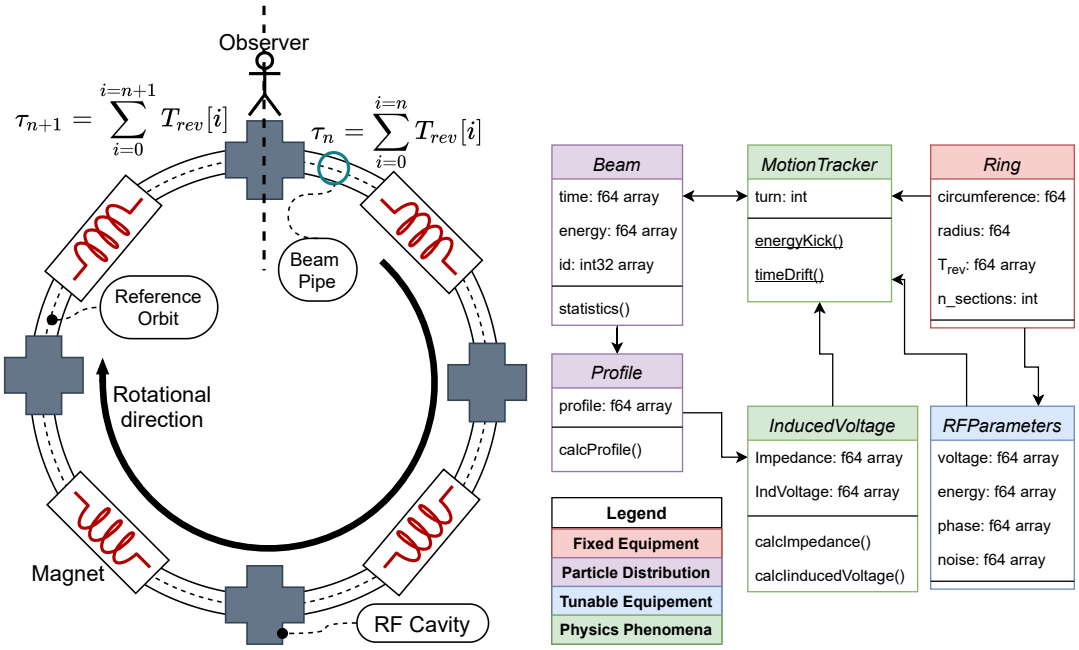
In addition, control systems used in the accelerators can be modeled with *BLonD*, too. So called beam-based or global feedbacks measure some beam observable, like the beam phase w.r.t. to RF bucket or the beam radial position, and feedback on the RF frequency. Cavity-based or local feedbacks control the RF voltage and phase bucket-by-bucket, adjusting the current of the amplifier that feeds the cavity so that the actual voltage is regulated to the desired voltage value. If enabled, the RF parameters are modified on a turn-by-turn basis.

## 3.2 The BLonD Simulator Suite

Fig. 3.2a shows a simple particle accelerator model containing the three main components modeled in *BLonD*: the synchrotron or 'ring', the Radio-Frequency (RF) cavities, and the beam that circulates inside the beam pipe. Fig. 3.2b shows a high-level class diagram of the core *BLonD* functionality together with the key class methods and class fields that interact in every typical *BLonD* simulation. The main code classes are the following:

- The `Ring` class. This class models the synchrotron, and contains machine-related parameters such as the machine's circumference, the type of particles to be accelerated and the number of RF sections.

- The `RFParameters` class. In this class are stored all the parameters related to the RF equipment that is attached around the beam pipe of the synchrotron. User-defined inputs such as the `voltage` and `momentum` arrays are stored in this class, and affect the motion of the simulated particles.




(A) Simplistic synchrotron model. The ring, the beam and the RF cavities are the main components.

(B) High-level class diagram of the *BLonD* simulator suite. The key methods and fields are depicted.

FIGURE 3.2: *BLonD* ring and class diagram.

- The `Beam` class. This class stores beam-related information, including the 2-D coordinates of all the simulated macro-particles. Thus, this class has the higher memory footprint in the *BLonD* code.

- The `Profile` class. This class is responsible for the beam profile, a histogram of the beam along the time axis, as well as other beam profile-related operations.

- The `InducedVoltage` class. This class describes the impedance of the machine, that is provided by external modeling tools and studies. The machines impedance interacts with the circulating particles, generating an induced voltage, which is also calculated by this class and is one of the most computationally heavy operations.

- The `MotionTracker` class. The actual tracking of the beam particles is performed in this class. This class contains methods related to the equations of motion that update turn-by-turn the 2-D particle coordinates. The evaluation of these equations is also one of the most computationally demanding operations in typical *BLonD* simulations.

More details about these clases, methods and their interactions are given below.

In reality, a bunch can contain trillions of particles, in simulations however, macro-particles are used which represent many real particles in order to reduce

the memory footprint. The user is responsible for determining the amount of macro-particles required to describe a certain physical phenomenon with sufficient resolution. The computational complexity of most operations in a *BLonD* simulation, as we can see from the following beam motion tracking equations, scales linearly with the number of simulated macro-particles, which typically ranges from a few millions to 100s of millions.

In *BLonD*, particle motion is described using the coordinates $(\Delta t_{(n)}, \Delta E_{(n)})$, which are the particle's arrival time and energy at the RF section with respect to the reference time $t_{d,(n)}$ and design energy $E_{d,(n)}$, respectively. The reference time is given by the revolution periods $T_{0,(n)} = 2\pi/\omega_{0,(n)}$ as follows:

$$t_{d,(0)} \equiv 0 \quad \text{and} \quad t_{d,(n)} \equiv \sum_{k=1}^{n} T_{0,(k)} \ \text{ for } \ n \geq 1. \tag{3.7}$$

On the other hand, the revolution periods are defined by the design orbit of radius $R_d$ and $\beta_{d,(n)}$, the relative speed of the design particle with respect to the speed of light $c$ on that orbit,

$$T_{0,(n)} = \frac{2\pi R_d}{\beta_{d,(n)} c}, \tag{3.8}$$

where the user can input $\beta_{d,(n)}$ implicitly via the corresponding design energy $E_{d,(n)}$ evolution over time. In *BLonD*, the reference time and design energy are therefore intrinsically connected. The reference time also serves as an external 'clock', to disentangle the equations describing the beam motion and the RF system, which are both tracked with respect to this external 'clock'. As a result, several beam, RF, and intensity effects can be included when modeling the beam motion.

RF sections are placed in fixed locations along the ring. The number of RF stations modeled along the ring depends on the case and ranges typically from one to a dozen. The so-called *kick* equation of motion is used to update the energy $\Delta E$ coordinate of a given particle from time step $n$ to $n+1$, based on the particle's $\Delta t_{(n)}$ coordinate and the RF voltage energy kicks $k$ received in the corresponding RF station,

$$\begin{aligned} \Delta E_{(n+1)} = \Delta E_{(n)} &+ \sum_{k=0}^{n_{\mathsf{rf}}} q V_{k,(n)} \sin(\omega_{\mathsf{rf},k,(n)} \Delta t_{(n)} + \varphi_{\mathsf{rf},k,(n)}) \\ &- (E_{\mathsf{d},(n+1)} - E_{\mathsf{d},(n)}) + E_{\mathsf{other},(n)}, \end{aligned} \tag{3.9}$$

where $q$ is the charge of the particle, $V_k$ the voltage amplitude, $\omega_{\mathsf{rf},k}$ the revolution frequency, and $\varphi_{\mathsf{rf},k}$ the phase of the RF system $k$, and $E_{\mathsf{d},(n+1)} - E_{\mathsf{d},(n)}$ the change of the design energy from one turn to another. The last term $E_{\mathsf{other},(n)}$ is

used to model energy changes due to intensity effects or synchrotron radiation, for instance.

The beam motion from one RF station to another is modeled by the *drift* equation of motion that updates the time coordinate using the updated energy of the particle. In *BLonD*, the machine-dependent slippage factor $\eta(\Delta E)$ used in Eq. 3.3 is replaced by the momentum compaction factor $\alpha$ of at least zeroth, and up to second order,

$$\Delta t_{(n+1)} = \Delta t_{(n)} + T_{\mathrm{rev},(n+1)} \left[ \left( 1 + \alpha_{0,(n+1)} \delta_{(n+1)} + \alpha_{1,(n+1)} \delta_{(n+1)}^2 + \alpha_{2,(n+1)} \delta_{(n+1)}^3 \right) \frac{1 + \frac{\Delta E_{(n+1)}}{E_{d,(n+1)}}}{1 + \delta_{(n+1)}} - 1 \right], \tag{3.10}$$

where $T_{\mathsf{rev}}$ is the revolution period and $\delta_{(n)} = \frac{\Delta p_{(n)}}{p_{d,(n)}} = \frac{\Delta E_{(n)}}{\beta_d^2 E_{d,(n)}}$ is the relative momentum offset. A full cycle of updating the beam coordinates corresponds to a single simulation iteration. The number of iterations required for a given testcase can range from a few thousands to a few millions.

While the kick and drift equations act on the particles one by one, the collective effects described in Eqs. 3.5 and 3.6 couple the particles and limit the exploitable parallelism degree of the code. In frequency domain, the induced voltage is calculated by discretizing Eq. 3.6 as

$$V_{\mathrm{ind}}[n] = -q\, N_p\, \mathrm{IDFT}\left( Z[k]\ \Lambda[k] \right), \tag{3.11}$$

where IDFT is the Inverse Discrete Fourier Transform and $\Lambda[k]$ is the Discrete Fourier Transform (DFT) of the line density $\Lambda[k] = \mathrm{DFT}\left(\lambda[n]\right)$.

In time domain, the induced voltage (Eq. 3.5) is calculated as a discrete convolution, where for run-time efficiency, the circular convolution theorem is being applied,

$$V_{\mathrm{ind}}[n] = -q\, N_p\, \mathrm{IDFT}\left\{ \mathrm{DFT}\left(W[n]\right)\ \mathrm{DFT}\left(\lambda[n]\right) \right\}. \tag{3.12}$$

In order to obtain a linear convolution in the end, both $W[n]$ and $\lambda[n]$ have to be suitably zero-padded to the length $L = N + M - 1$, where $N$ is the length of $W[n]$ and $M$ is the length of $\lambda[n]$. The complexity of the time-domain algorithm that uses FFTs is $O(L \log L)$, while that of the direct convolution is $O(NM)$.

Since both Eqs. 3.11 and 3.12 apply and IDFT, BLonD has a single implementation for both frequency- and time-domain methods. The only difference is that, in time-domain calculations, a pseudo-impedance and beam spectrum are defined as $Z^*[k] = \mathrm{DFT}\left(W^*[n]\right)$ and $\Lambda^*[k] = \mathrm{DFT}\left(\lambda^*[n]\right)$, respectively, where

the * represents zero-padding of the signal.

Just as in particle-in-cell simulations [51], induced voltage is calculated on a discrete histogram of the beam line density to optimise the computation time. Particles are grouped in bins. The amplitude corresponding to the bin is given by the number of particles inside and a linear interpolation is applied between bins. The histogram acts as a frequency filter and cuts high-frequency numerical noise, but also physical contributions if the resolution is not sufficient. The minimum amount of bins required is fixed by the Nyqvist sampling theorem, and depending on the phenomena modelled, the user has to choose the adequate amount of bins.

*BLonD* is a modular and flexible library. Other physics phenomena, such as space-charge effects, synchrotron radiation, or impedance-reducing control circuits can be included in a given simulation. A *BLonD* simulation scenario is an assembly of components which in turn can be composed of smaller sub-parts. The user, knowing which physics effects are essential for a given study, initializes the relevant components. Then, the user wires the components to form a pipeline of physics transformations that will be computed on a turn-by-turn basis. Some optional features of *BLonD* include a complete tool-set for data analysis, storage and plotting.

## 3.3 Related Work

The *BLonD* simulator suite emerged in 2014 to respond to the need for a highly customisable tool required by scientists at CERN driving a large amount of longitudinal beam dynamics studies. Some of the features modeled in *BLonD* were unique and had not been implemented by any other prior code.

ESME [52, 53], a code developed at Fermilab since 1984, was widely used prior to *BLonD*. The code is written in Fortran and is compatible with certain Unix-based operating systems like Solaris. The lack of support and maintenance led the ESME project to slowly disappear from being used in the scientific community.

Py-Orbit [54, 55] and Elegant [56, 57] are two other alternatives to *BLonD*. They are Particle-In-Cell, 6D tracker codes, modelling both transverse and longitudinal beam dynamics. For longitudinal purposes, the computational complexity of a Py-Orbit or Elegant simulation is significantly heavier than that of a *BLonD* simulation. Good agreement between Py-Orbit and *BLonD* has been reported in various studies [45, 38]. Being more general-purpose oriented codes, both Py-Orbit and Elegant lack many of the specific longitudinal and RF features available in *BLonD*.

The *BLonD* suite differs from the aforementioned codes in numerous ways. Using a python front-end makes *BLonD* easy-to-use and particularly attractive to new users. The modular structure allows rapid prototyping of new features that extend its capabilities. Contrary to Py-Orbit and Elegant, *BLonD* specializes in the longitudinal plane and as a consequence, it contains more detailed physics models and is computationally less heavy, resulting in shorter simulation times. In terms of flexibility and precision, *BLonD* has been tested successfully on a wide range of real-world simulation scenarios and it is generic enough to cover a wide spectrum of beam dynamics simulation scenarios ranging from existing to future machines and from relatively small to very large synchrotrons accelerating protons, electrons or ions. Finally, as we will see later in this thesis, *BLonD* allows for efficient horizontal and vertical scaling across hundreds of cores, as well as heterogeneous simulations using CPU and GPU platforms.

# Chapter 4

# Intra-node Optimizations

## 4.1 Optimization Methodology

To cope with the challenging simulation complexity and prediction accuracy needed in the field of longitudinal beam dynamics, single-node, run-time optimizations were introduced in the *BLonD* simulation suite, and are described in this section. Our analysis and experimental evaluation is based on four real-world simulation scenarios, representative of typical *BLonD* simulations scenarios, each concerning a different particle accelerator. As a first approach to performance optimization, the most compute-intensive code regions are ported to C++. Then, the Top-Down method [47] is applied on the testcases to provide an in-depth micro-architectural insight of *BLonD*. Based on the analysis outcome, we identify a series of bottlenecks and proceed to mitigate them through compiler tuning, the use of high-performance scientific libraries and other software optimization techniques. We employ the Roofline model [48, 49] to verify the efficiency of *BLonD*'s computation core. In addition, we employ OpenMP [12] to parallelize the compute-intensive code regions. Finally, we evaluate the single-core run-time performance of *BLonD++* [58] as well its scalability in a multi-core Intel Haswell [59] server platform. The proposed implementation, *BLonD++*, demonstrates an up to 23× single-core run-time speedup compared to the previous python-only *BLonD* library. By dramatically reducing the duration of a week-long simulation to below 9 hours, *BLonD++* has enabled users to perform several beam dynamics studies that were previously unfeasible due to run-time, memory, and CPU limitations.

We adopted a straightforward performance optimization methodology to improve the intra-node performance of the *BLonD* code. Our methodology can be easily adapted to the needs of other similar high-performance scientific simulators. The suggested methodology is depicted in Fig. 4.1. Initially, we identify the most time-consuming regions, by profiling four representative, large-scale simulation testcases. We then port these identified regions, also referred to as *benchmarks*, from Python to C++ in order to profit from compiler and

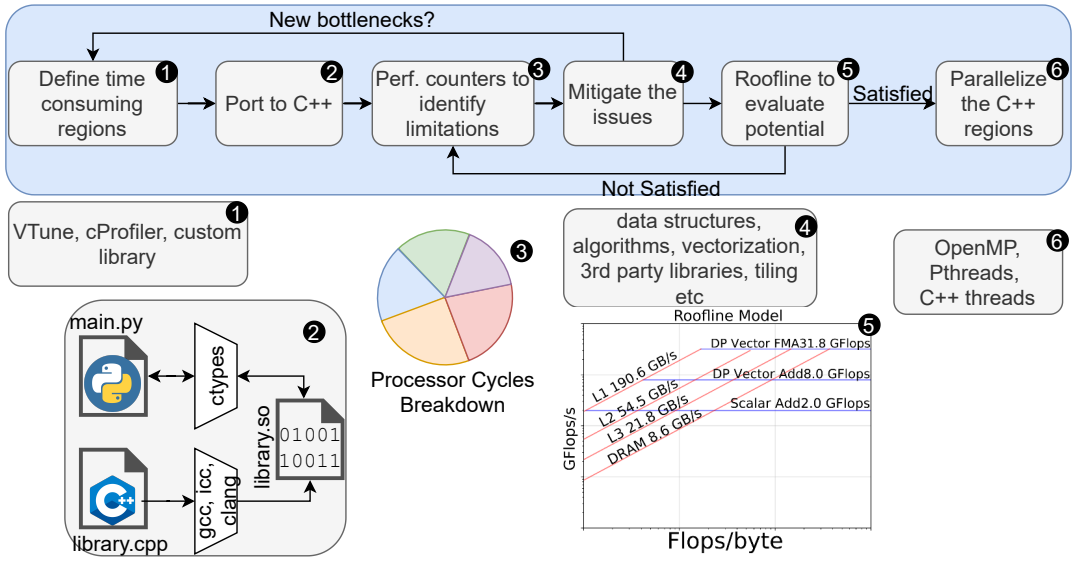



FIGURE 4.1: *BLonD++* performance optimization methodology.

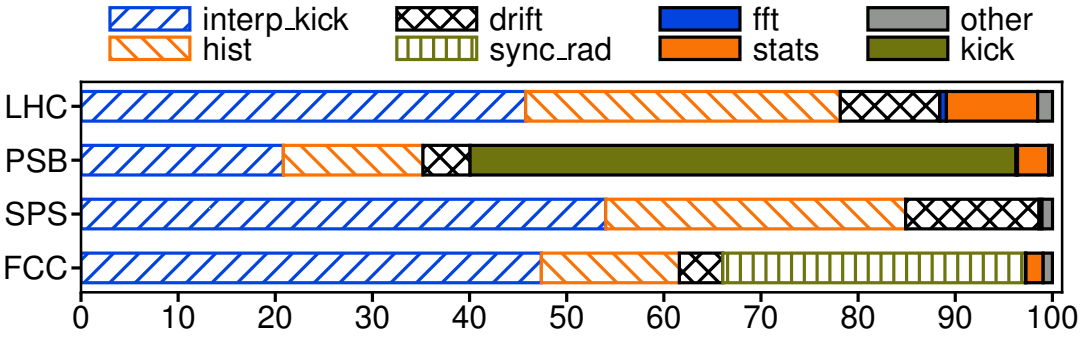



FIGURE 4.2: Run-time breakdown of the four target testcases with the initial, python-only *BLonD* version. The seven tagged methods are responsible for 99% of the run-time on average.

other low-level optimizations. Then, we use a collection of hardware counters to discover various micro-architecture limitations. Subsequently, we mitigate these limitations by combining a series of typical High-Performance Computing (HPC) strategies. To evaluate and verify the performance of our benchmarks, we apply the Roofline model [48, 49] analysis. Finally, when we are satisfied with the achieved performance, we parallelize the computational core of *BLonD*, that is, the identified set of benchmarks, using the OpenMP [12] framework.

## 4.2 C++ Computational Core

*BLonD* started originally as a pure Python code. Python is a widely adopted language for various reasons; to name a few, Python combines rapid prototyping and development, ease-of-use, object-oriented design principles, seamless

cross-platform support, and simple integration with a huge body of third-party libraries [60, 61].

To identify the most time-consuming regions of the *BLonD* code, we built a lightweight time profiler. In Fig. 4.2, we can see the run-time breakdown of four representative *BLonD* simulation cases. The seven tagged methods, namely `LIkick()`, `drift()`, `hist()`, `fft()`, `stats()`, `SR()` and `kick()` aggregate 99% of the simulation time. Thus, in the remainder of this section wer refer to them as benchmarks, and focus our efforts on optimizing their performance individually but also collectively, whenever possible.

As a first approach to reduce the run-time, the selected benchmarks were ported to C++; a programming language that is well suited for performance-critical applications. To interface the existing Python code with the C++ extensions, the C++ sources are compiled into a shared library which is exposed to Python via the `ctypes` module. This hybrid implementation combines the best of both programming languages; the usability of Python in the front-end and the efficiency of C++ in the compute-intensive back-end. Furthermore, it allows for fine control over the compilation process of the C++ sources.

A noticeable speedup was extracted by porting the computationally intensive core to C++ and is reported in Fig. 4.3. The first bar of every group shows the overall speedup of the testcase specified in the x-axis and the remaining bars of every group show the speedup of each individual benchmark in that testcase. The run-time ranges from 3.3× to 12.5×, or 7.5× on average. The `SR()` method of the FCC testcase used the Boost library [62] for the Pseudo-Random Number Generation (PRNG) as it was noticed at this early stage that the STD library PRNGs performed worse. The `fft()` benchmark has not been optimized with respect to the Python-only version as at that moment, as `fft()` was allocating a very slight percentage of the run-time. However, as we will see below, the `fft()` performance will be revised in Sec. 4.5.

## 4.3 Probing Performance Counters for Optimization

Software developers utilize performance counters extracted with the aid of the Performance Monitoring Units (PMUs) to better understand workload bottlenecks and act accordingly. Due to the increased micro-architecture complexity of modern processors, and the large number of performance counters, this process can be cumbersome and error prone. To deal with these challenges, the Top-down Micro-architecture Analysis Method (TMAM) [47] has emerged, and has been integrated in widely-used profiling tools like the Intel VTune [63]

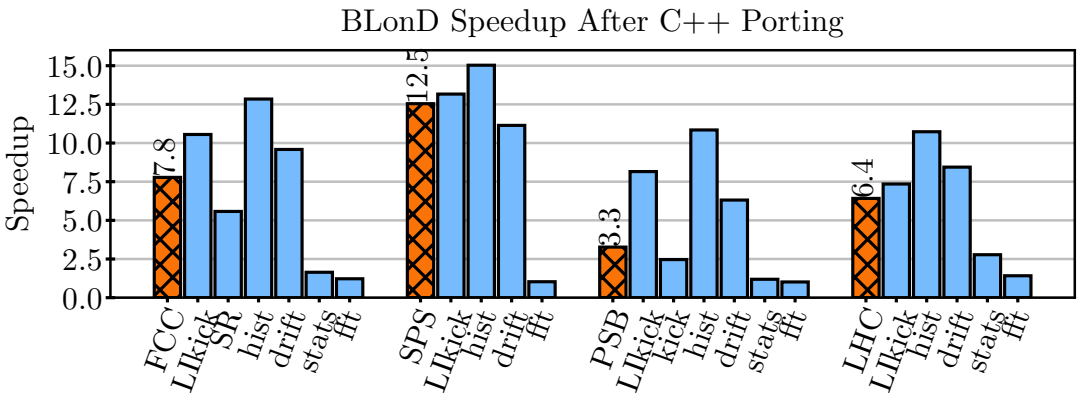



FIGURE 4.3: Per-testcase and per-benchmark speedup of the first *BLonD++* revision compared to the initial, Python-only version.

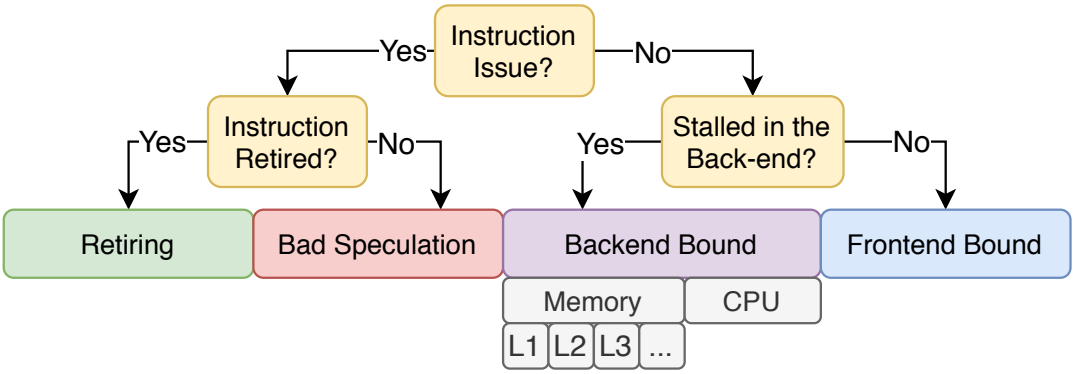



FIGURE 4.4: Top-Down breakdown of pipeline slots into different categories.

and the Linux Perf [64]. TMAM is a low-overhead, hierarchical method that effectively pinpoints performance limitations in modern, out-of-order (OOO), super-scalar processors. TMAM divides the total number of processor pipeline slots into four main categories:

**Bad Speculation (BS)** denotes slots wasted due to all aspects of incorrect speculations like mis-predicted branches.

**Retiring (RET)** denotes slots utilized by "useful operations". Ideally, all slots should be attributed here. However, a high retiring fraction does not necessary mean that there is no room for improvement.

**Front-End Bound (FEB)** denotes stalled slots because the pipeline's front-end undersupplies the back-end. The front-end is the portion of the pipeline responsible for fetching the next instruction from the ICache and decoding it into micro-operations to be executed by the back-end.

**Back-End Bound** denotes stalled slots due to lack of resources to accept new operations. It is further divided into: **Memory bound (MB)** which

reflects execution stalls due to the cache and memory subsystems, and **Core bound (CB)** which reflects either pressure on the execution units or lack of Instruction Level Parallelism (ILP).

The above categories are then further broken down into sub-categories (see Fig. 4.4), that are more closely associated with one or a few possible bottlenecks. Each category is assigned a weight, and the category with the highest value is flagged as the one limiting the workload's performance, therefore the developer can safely ignore all the other categories and focus only in one. Compared to other approaches, TMAM is generic enough to be applied to any modern O3 processor, it induces a low time overhead, and it offers clear insights on micro-architecture related performance bottlenecks.

Following the TMAM approach, we analyze our target benchmarks and identify the most critical performance issues for each of them.

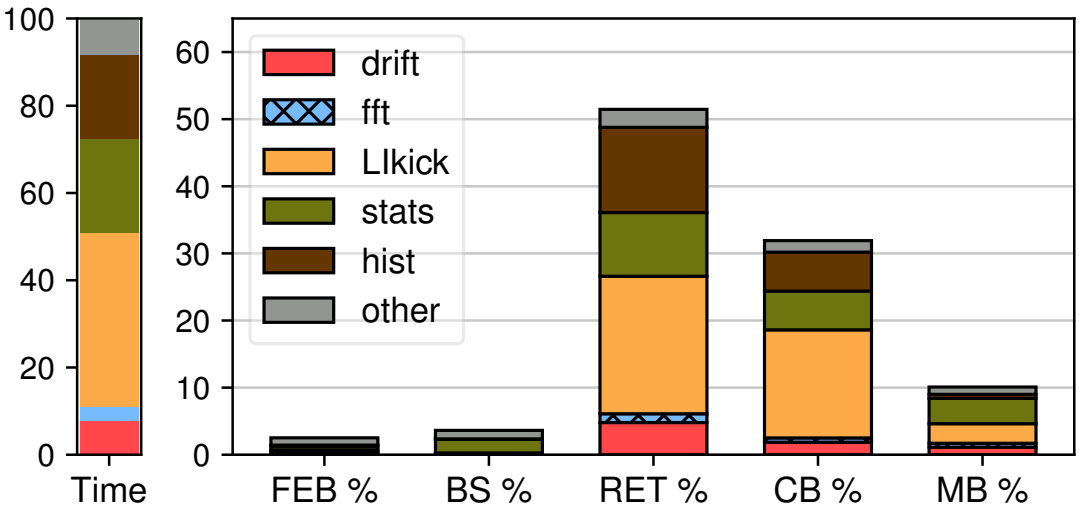



FIGURE 4.5: LHC TMAM cycles breakdown.

In Sec. 4.2, a compelling speedup was reported by porting the computation core of the *BLonD* code to C++. To go even further, we need to identify the performance limitations of the new code. In this section, the TMAM analysis is applied to the targeted testcases in order to better understand and eventually tackle the micro-architectural bottlenecks of the *BLonD* suite.

To reproduce the TMAM breakdown described earlier in this section, a collection of approximately 90 hardware counters is needed. To facilitate the automation of the collection process, the command-line interface of the Intel VTune Amplifier [63] was used. The Instrumentation and Tracing Technology (ITT) API [65] was utilized to localize the event collection around the regions of interest as well as to enable a more fine-grained, per-benchmark event grouping. Finally, after the collection and grouping of the events was completed,

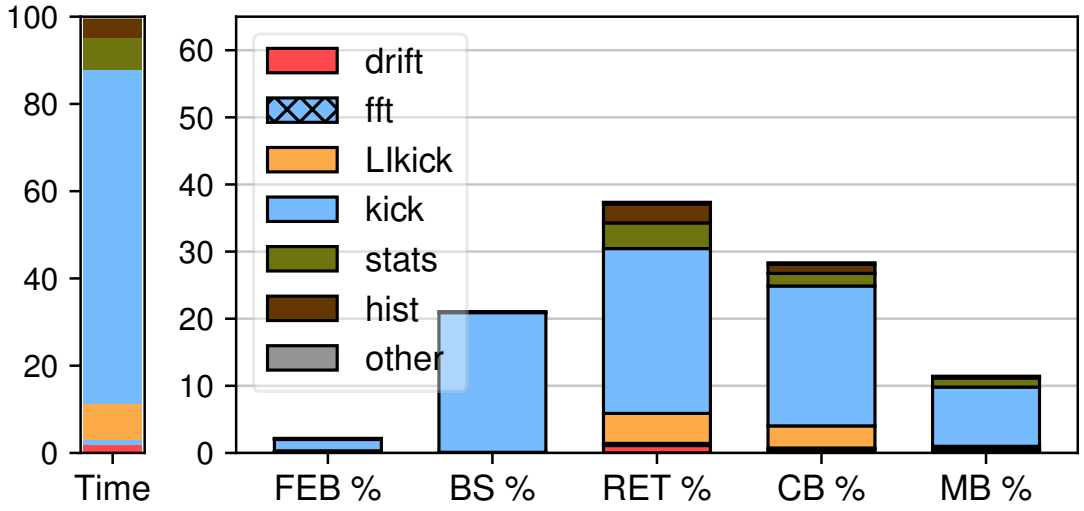



FIGURE 4.6: PSB TMAM cycles breakdown.

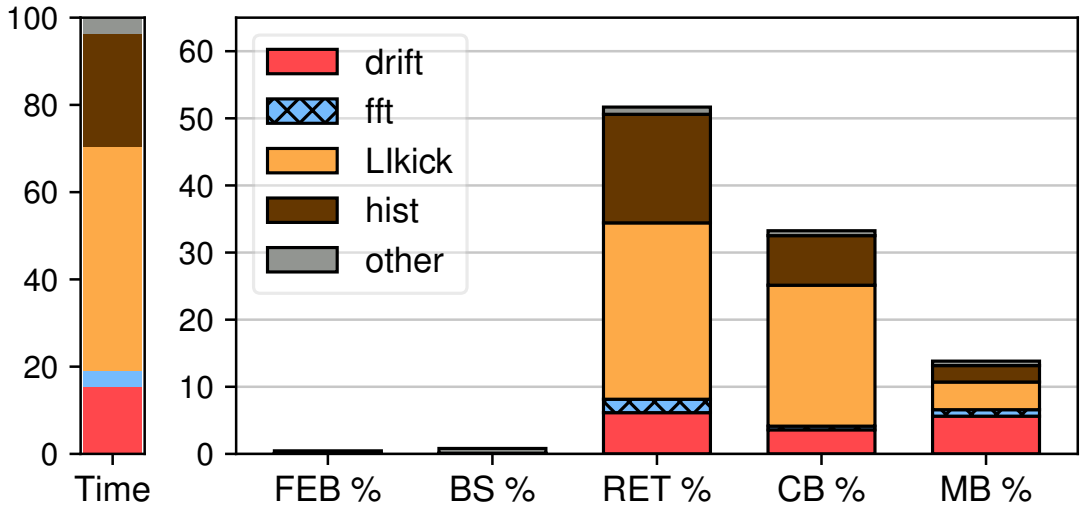



FIGURE 4.7: SPS TMAM cycles breakdown.

the formulas suggested by TMAM were used to breakdown the total available processor pipeline slots into the following categories: FEB, BS, RET, CB, and MB.

Figs. 4.5, 4.7, 4.6, and 4.8 show this breakdown for each of the targeted testcases displaying the percentage of the total available pipeline slots dedicated to each of the above-mentioned categories on a per-benchmark base. The bars on the left each plot show the time contribution of the considered benchmarks to the run-time. Based on these cycle breakdown figures, the following five performance inefficiencies were identified.

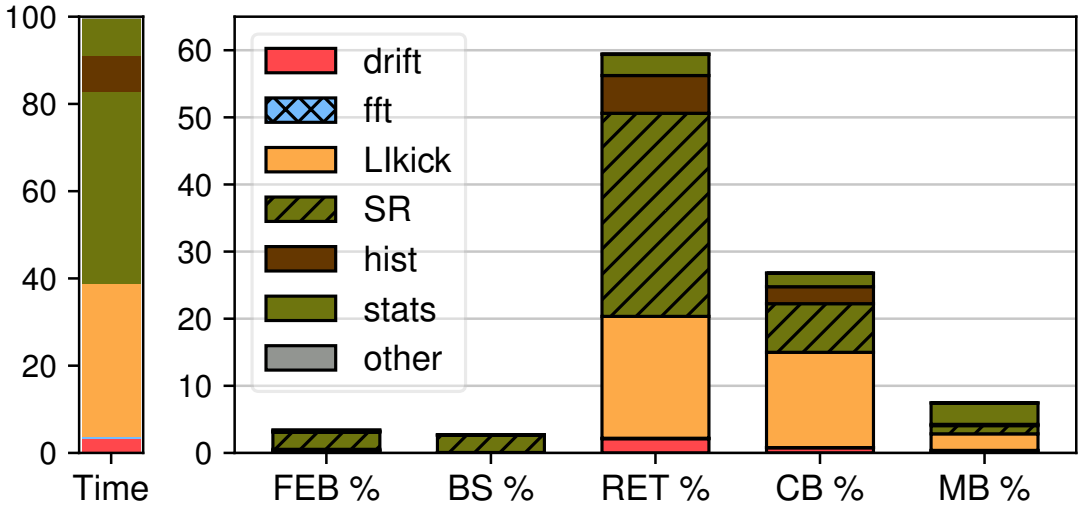



FIGURE 4.8: FCC TMAM cycles breakdown.

### 4.3.1 Core-bounded `LIkick()`.

The `LIkick()` benchmark is in overall responsible for a big portion of wasted cycles due to core-related stalls. Core-related stalls are usually caused by sequences of dependent instructions or unbalanced use of the execution units that leads to instruction serialization and ILP deterioration. Furthermore, `LIkick()` contributes significantly to the retiring part. As mentioned in Sec. 4.3, a high retiring percentage does not necessarily mean that there is no space for improvement. In particular, vectorization is a technique that lets more operations to be executed by a single instruction, thus decreasing the retiring percentage and speeding up the execution at the same time.

The inefficiency spotted in `LIkick()` was tackled in two phases. The `LIkick()` function computes the energy transferred to each macro-particle, every time the beam passes through an acceleration cavity. At first, it was noticed that a portion of the main computation of `LIkick()` was independent of the particle index and was determined only by the particle distribution bin to which the particle belonged to. As a result, two auxiliary arrays of a size equal to the number of bins ($\sim 10^3$) were pre-calculated and then used as look-up tables in the main loop ($\sim 10^6 iterations$) saving expensive computations. Furthermore, the main loop was unrolled and partially vectorized.

### 4.3.2 Memory-bounded `drift()`.

The second identified issue is related to the `drift()` benchmark. Fig. 4.9 shows that `drift()` suffers from frequent memory stalls. To mitigate this pathogenic behavior, we noticed that the calculation of `drift()` and `LIkick()` can be

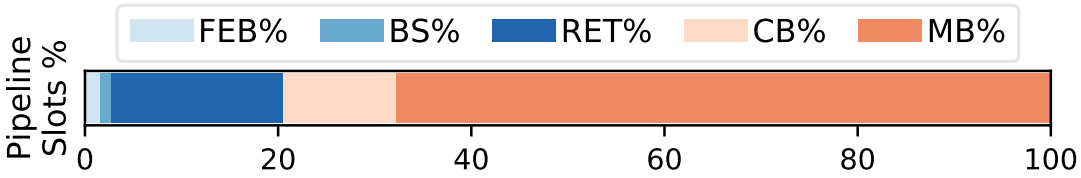



FIGURE 4.9: TMAM breakdown for the `drift()` benchmark. 68% of the total pipeline slots is wasted due to memory-related stalls. The input size was 1M particles.

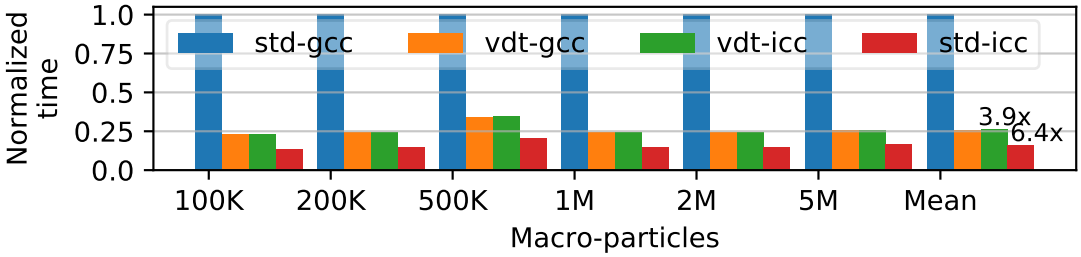



FIGURE 4.10: Comparison of STD and VDT libraries with the gcc and icc compilers in `kick()`. The STD-icc configuration is on average 6.4× faster than the STD-gcc configuration.

interleaved. By doing so, the memory loads are reduced roughly by 50% and therefore the pressure to the memory subsystem is reduced.

### 4.3.3 Inefficient `kick()` implementation.

The third issue concerns the `kick()` benchmark that dominates the run-time of the PSB testcase. The most time consuming part of `kick()` is the calculation of the `sin()` function. In Fig. 4.10, the performance of `kick()` is evaluated with the C++ Standard Library (STD) [66] and the VDT Library [67] compiled with the `gcc` and `icc` compilers. The values on the y-axis are normalized to the `STD-gcc` configuration. The fastest configuration appears to be the use of the STD library compiled with the `icc` compiler which is on-average 6.4× faster than the `STD-gcc` configuration.

### 4.3.4 Inefficient `SR()` Implementation.

The `SR()` benchmark is responsible for the fourth bottleneck. In the FCC testcase, `SR()` dominates in the RET, FEB, BS categories and the run-time. The most time-consuming task of the `SR()` benchmark is the pseudo-random number generation (PRNG). In Fig. 4.11, the performance of three different PRNG libraries is evaluated: STD [66], Boost [62] and Intel MKL [68]. The latter is the most efficient and outperforms the STD library by 11.4× on average across a range of input sizes.

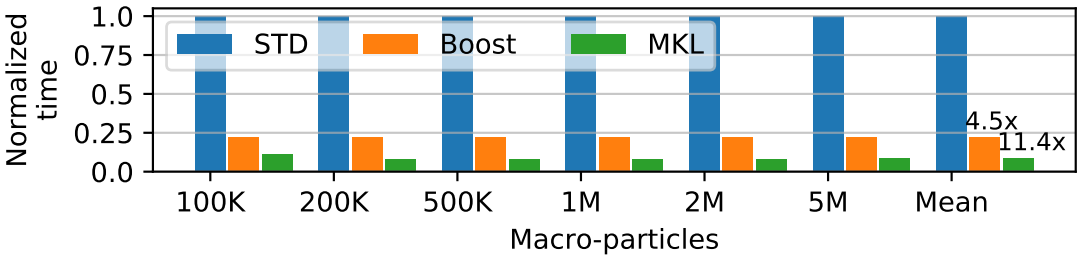



FIGURE 4.11: Benchmarking the STD, Boost and MKL libraries for the PRNG methods in the `SR()` function. The MKL PRNG method is on average 11.4× faster than the STD counterpart.

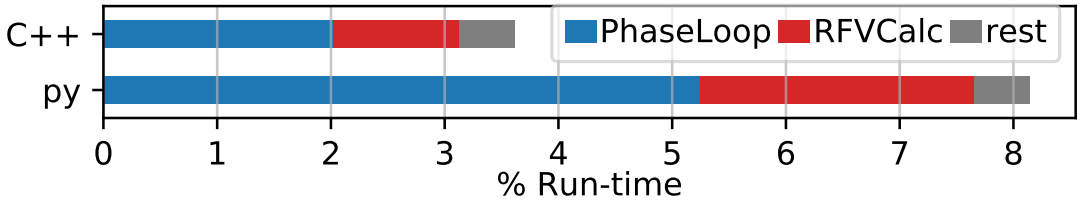



FIGURE 4.12: Breakdown of the "other" part of the LHC testcase before and after porting `PhaseLoop()` and `RFVCalc()` to C++.

### 4.3.5 Large Contribution of "other" in LHC Testcase.

The final underlined limitation appears in the LHC testcase. The "other" part, which represents the code that does not belong to any of the considered benchmarks, allocates 8% of the run-time. While this might seem as a minor issue, it is crucial to reduce the contribution of the serial parts as they greatly affect the overall scalability of the code. With detailed profiling, we discovered the two most significant methods of the "other" part: `RFVCalc()` and `PhaseLoop()`. By parallelizing and porting them to C++ the contribution of the "other" part to the overall run-time dropped to 3.5%. Fig. 4.12 summarizes the run-time breakdown of the "other" part in the LHC testcase, before and after the porting to C++.

Tackling the issues mentioned above with the suggested techniques lead to the next generation beam longitudinal dynamics simulator suite, *BLonD++*. The evaluation of the single-core performance of *BLonD++* is given in Sec. 4.6.2.

## 4.4 Roofline Model Assisted Analysis

The vast complexity and variability of modern applications have motivated CPU architects to incorporate a plethora of hardware components in CPU processors, e.g. Fused Multiply Add (FMA) units, multiple cache levels, wide vector units, multiple hyper-threads per CPU core, among others. This has resulted

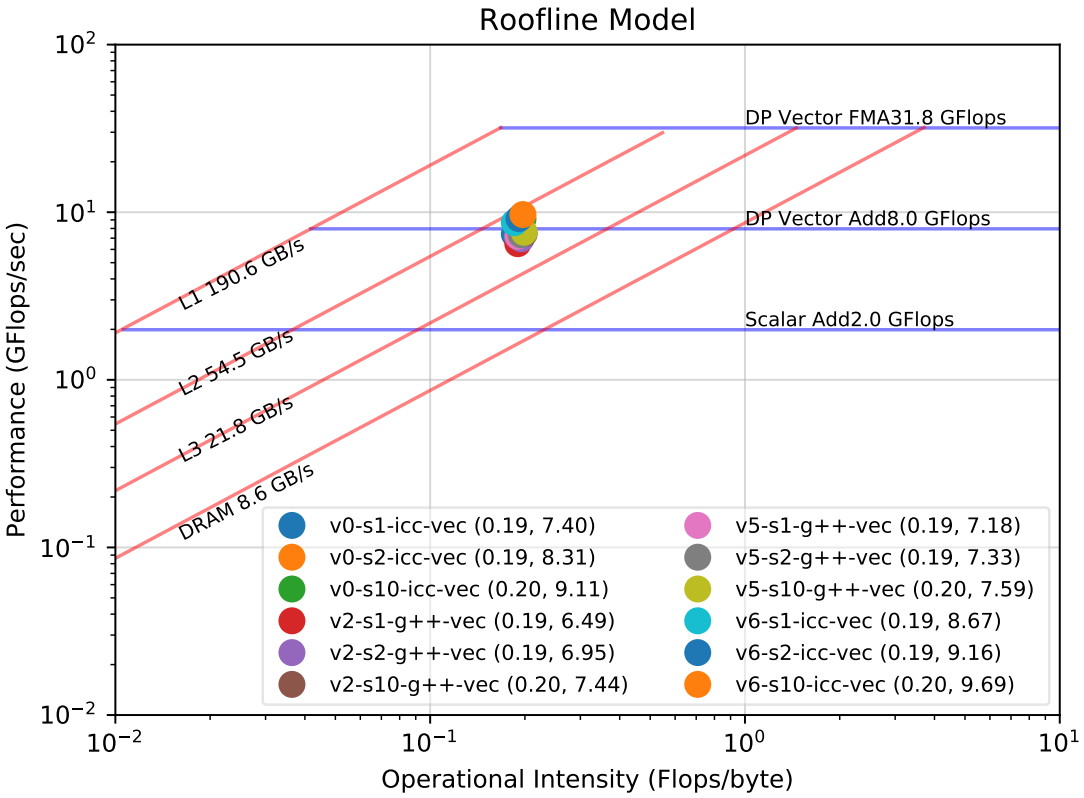



FIGURE 4.13: Roofline model applied in the `kick()` benchmark. The best performing variation (v6-s10-icc-vec) is bounded by L2 cache bandwidth. To break through this ceiling, we need to fit the entire data-set in the L1 cache.

in a complex processor design, which makes it challenging to exploit to its full potential by application developers.

Usually, when executing an application in a given CPU architecture, one or a few resources are those that limit the application's performance. For example, memory-intensive workloads are limited by the cache capacity and memory bandwidth, while CPU-intensive workloads are less sensitive in terms of memory capacity but are prone to congestion in the integer or floating point execution units. Knowing which resource is limiting the application's performance is a valuable insight and can motivate certain optimization strategies.

The Roofline model [48, 49], is a graphical representation of the theoretical upper-bound performance of a computer architecture. By taking into consideration the bandwidth of the memory and cache components, as well as the throughput of various compute units it offers useful insights and guidelines for improving applications, and also evaluating how close to their maximum potential they perform.

In Fig. 4.13, we can see an application of the Roofline model for the purposes of optimization of the `kick()` benchmark. The inclined lines correspond to the

bandwidths of the various memory structures, while the horizontal ceilings correspond to execution units related optimizations. Each bullet point represents a different variation of the kick algorithm. As we can see, the operational intensity is only slightly affected, since it is a property of the algorithm. The attained performance of the best variation (v6-s10-icc-vec) has hit the L2 memory bandwidth ceiling. Therefore, this is an indication that in order to extract more flops/sec, we need to modify our algorithm so that the dataset can mostly fit in the L1 cache. Unfortunately, this is not always possible. However, we can safely stop optimizing this particular algorithm, since it has reached its full potential in this specific evaluation platform.

## 4.5 Code Parallelization

To anticipate forthcoming computational challenges in the field of beam dynamics, the seven considered benchmarks were parallelized.

The framework used to express parallelism is the OpenMP [12]. Simplicity, maturity, compiler support, and scalable performance are some of the assets that made OpenMP the most popular shared-memory parallelization framework. In general, most benchmarks were parallelized using the parallel loop `pragmas`. Some benchmarks, like `hist()` were not inherently parallel as they need to update shared data structures. In this case, to avoid atomic operations or other means of synchronization, each thread computes a private histogram and in the end the private histograms are reduced to a global one. For the `fft()` benchmark, the multi-threaded version of the FFTW library [69] was used. We performed a deep evaluation and analysis of *BLonD++* scalability, reported in Sec. 4.6.3.

## 4.6 Experimental Evaluation

### 4.6.1 Experimental Setup

The proposed *BLonD++* library is evaluated experimentally on a Non-Uniform Memory Access (NUMA), multi-core server platform. Table 4.1 summarizes the hardware set-up. The Intel Turbo Boost technology and the hyper-threading feature were disabled for more stable and reproducible measurements. The standard deviation of the reported results is $\approx 1\%$.

TABLE 4.1: Hardware Set-up.

| Model | Intel Xeon Haswell E5-2683v3 @ 2.00GHz | OS | CentOS Linux 7.4 kernel 3.10 |
|---|---|---|---|
| Slots | 2 | Cores/Slot | 14 (Total: 28) |
| L1I & L1D | 32KB/core (Total: 896KB) | L2 | 256KB/core (Total: 7MB) |
| L3 | 35MB/slot (Total: 70MB) | DRAM | 64GB |
| Compiler | gcc 5.3 & icc 18.0 | Flags | -O3 -ffast-math -mtune=native |
| Turbo Boost | Off | Hyper-Threads | Off |

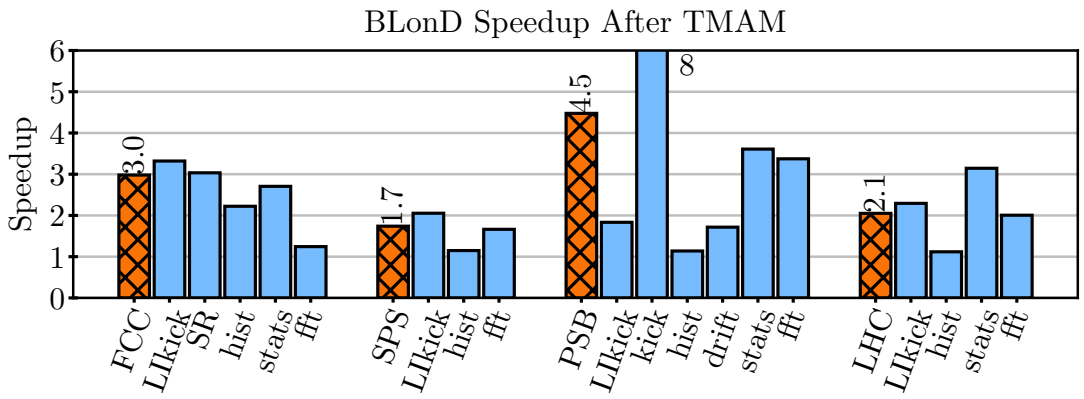



FIGURE 4.14: Single-core execution time evaluation of *BLonD++* after mitigating the identified issues with TMAM.

### 4.6.2 *BLonD++* Single-core Performance

Fig. 4.14 presents the speedup gained by mitigating the identified bottlenecks described in Sec. 4.3.

The `Likick()` that now includes `drift()` in all testcases except PSB, has been improved by a factor of 2× to 3× in terms of run-time. This is mainly the result of saving computations by utilizing the look-up tables described in Sec. 4.3, reducing memory loads by overlapping `LIkick()` with `drift()`, and finally employing auto-vectorization. The `kick()` benchmark that was dominating the PSB testcase is 8× faster in the fully optimized version. In the FCC testcase, the `SR()` has been improved by a factor of 3× due to the use of the random number generation functions from the MKL [68] library. Note that in the previous, un-optimized version, the Boost [62] library was used and not the C++ STD [66] library. In the `fft()` benchmark, the Scipy [60] FFTs have been replaced by the more efficient FFTW [69] library. The `fft()` benchmark demonstrates a speedup of up to 3.3×.

In Fig. 4.15, the cumulative speedup of the final *BLonD++* version against the initial Python-only version is presented. *BLonD++* achieved a 18× speedup in the four representative testcases on average. This means that a previously day-long, single-core simulation, can now be completed in 80 minutes while a

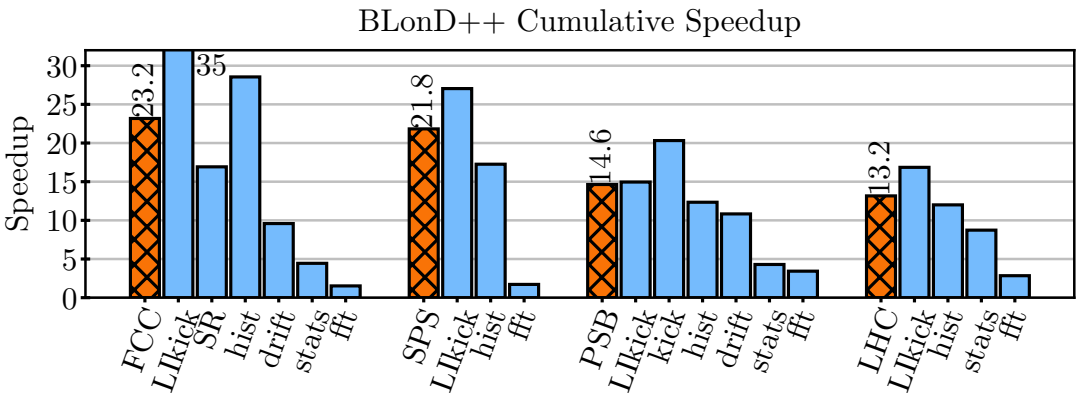



FIGURE 4.15: Single-core cumulative speedup of *BLonD++* over the initial *BLonD* version. *BLonD++* demonstrates a 18× speedup on average.

TABLE 4.2: *BLonD++* scalability efficiency.

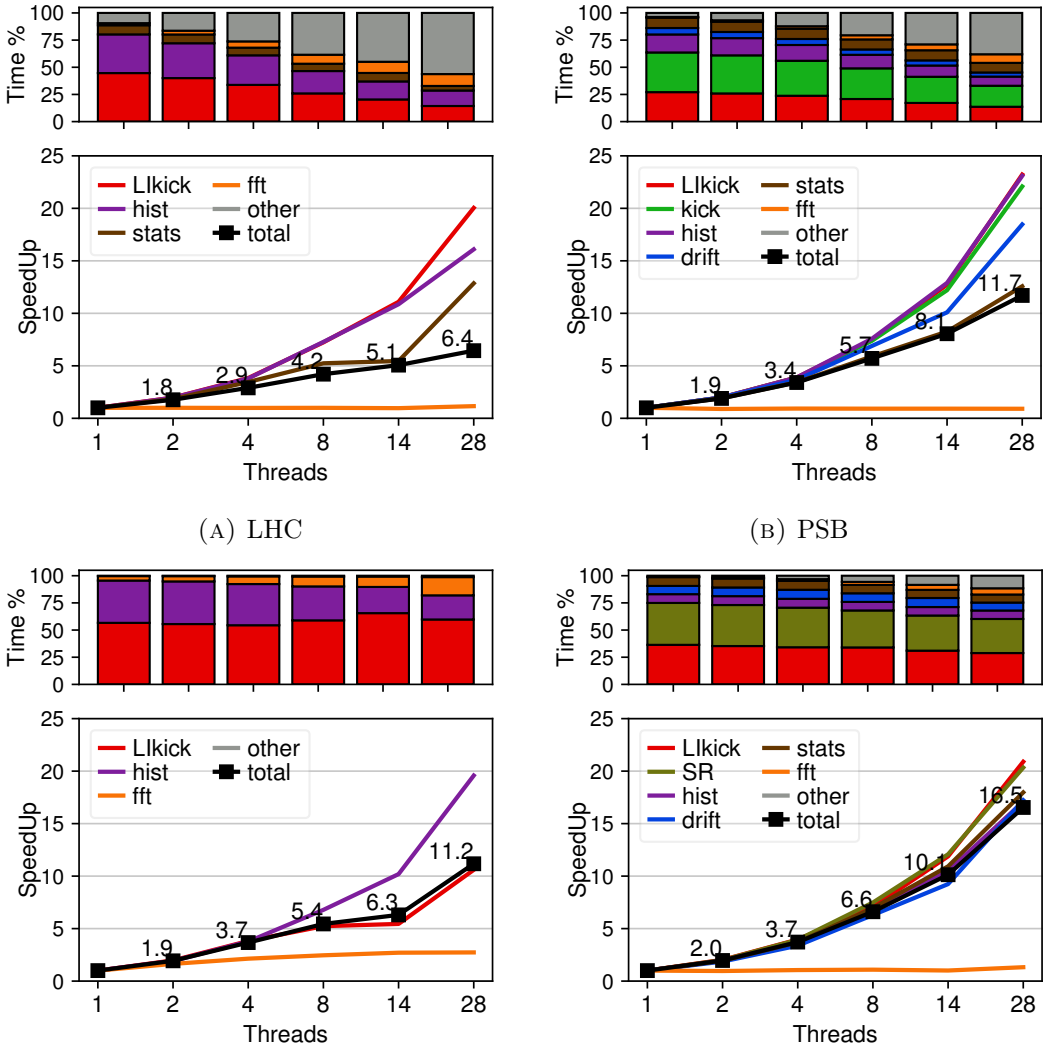
| Testcase | Non-parallel (%) | Efficiency (%) | | |
|---|---|---|---|---|
| | | 4 thr. | 14 thr. | 28 thr. |
| FCC | 0.85 | 95 | 80 | 73 |
| PSB | 3.73 | 95 | 85 | 84 |
| SPS | 0.35 | 93 | 47 | 44 |
| LHC | 9.40 | 93 | 80 | 81 |
| Mean | 3.58 | 94 | 73 | 71 |

week-long simulation needs only 9 hours to complete. Furthermore, this dramatic reduction in execution time has enabled the scientists using *BLonD* to simulate scenarios that combine more complex physics phenomena with finer resolution. For instance, in the SPS, modeling 144 bunches was crucial to get more accurate predictions for the upgrade of the machine, since bunches are coupled through intensity effects. This only became possible with the optimized *BLonD++* version.

### 4.6.3 Scalability Analysis

The dramatic single-core speedup reported in the previous section is sufficient to enable studies of physics effects that were previously unfeasible due to run-time limitations in deeper detail. Nevertheless, as future challenges are anticipated, the considered benchmarks were parallelized to provide even greater speedups.

As mentioned in Sec. 4.5, OpenMP was used to express parallelism. The multi-threaded version of FFTW [69] was used in the `fft()` benchmark. FFTW enables multi-threading only after a certain threshold of input points. This is why the `fft()` benchmark shows scalable behavior only in the SPS testcase




(A) LHC (B) PSB

(C) SPS (D) FCC

FIGURE 4.16: Intra-node scalability stressing of four real-world test-cases.

where the problem size is large enough. The FFTW library also defines the exact number of utilized threads depending on the input size.

Fig. 4.16 summarizes the scalability analysis for the four selected testcases. The top subplot of each sub-figure shows how the contribution of every benchmark to the total execution time changes with the increase of the thread count. The contribution of the multi-threaded parts decreases with the increase of the threads, and correspondingly, the non-parallelized parts become more significant as the thread count increases. This means that for a testcase to be scalable as a whole, all of its sub-parts need to be sufficiently scalable. The bottom subplot of each sub-figure shows the speedup of each benchmark as well as the speedup of the whole testcase, compared to the single threaded execution.

In general, most of the benchmarks demonstrate decent scalability. However,

not all testcases scale well as a whole. In the LHC and PSB testcases the "other" part, dominates the run-time when the thread count increases. The "other" part needs to be further broken down into sub-parts, analyze them and explore new optimization opportunities. In the SPS testcase, `LIkick()` does not demonstrate a very scalable behavior. This is mainly due to the big input size which requires more than 1.2 GB of memory. As the thread count increases, the overall performance is limited by the memory bandwidth. This explains the speedup jump from 14 to 28 threads. The first 14 threads are allocated in the first node to avoid expensive inter-thread communication via the main memory. The second group of threads is scheduled in the second node, which unlocks an extra 35 MB of L3 cache. As a result, the performance of this testcase scales ideally from 14 to 28 threads. Finally, the FCC testcase demonstrated the most scalable behavior among the testcases.

Table 4.2 shows how efficiently each testcase scales. The "Non-parallel%" column shows what percentage of the run-time is spent on serial code. The efficiency has been calculated as the ratio of the measured speedup to the theoretical speedup according to Amdahl's law. All the reported values are percentages. We show the efficiency for four, 14 and 28 threads. Four threads is a typical number of cores that a desktop computer has, 14 is the number of cores in each node of the platform used for the experimental evaluation and 28 is the total number of available cores in the experimental platform. On average, *BLonD++* achieved near-optimal, 94% scalability efficiency with four threads. This indicates that the multi-threaded benchmarks are indeed efficiently parallelized, successfully avoiding harmful effects of synchronization and load imbalance. Moreover, *BLonD++* achieved over 70% efficiency with 14 and 28 cores, which is acceptable considering the system's configuration. Nonetheless, there is still room for improvement.

## 4.7 Memory-Bound Limitation

Despite our efforts to parallelize the most computationally intensive regions of our code, we saw that in large thread counts, *BLonD++* is constrained by the memory hierarchy. This limitation appears quite often in modern parallel applications. In the coming section, we will see how we employ distributed computing and horizontal scaling to allow multiple computing nodes to collectively run *BLonD* simulations. This not only addresses the memory-bound limitation, since the memory bandwidth is effectively multiplied by the number of available computing nodes, but also allows our code to scale across multiple computing nodes.

# Chapter 5

# Scale-Out Beam Dynamics

## 5.1 Ever-Growing Simulation Sizes and Horizontal Scalability

The single-node optimized *BLonD++* code managed to efficiently reduce the run-time of beam dynamics simulations and push the boundaries of practicable simulation scenarios further. However, the past and upcoming synchrotron upgrades, including the LHC Injector Upgrade (LIU) project [28], the High-Luminosity LHC project (HL-LHC) [29], and the studies of future machines, like the FCC [31], keep pushing the current technology boundaries and call for larger and more extensive simulation studies.

As shown in Chapter 4, *BLonD++* is constrained by the memory bandwidth when increasing the thread count. As a consequence, it cannot exploit existing or future multi-/many-core processors to their full potential. Furthermore, even if this issue was mitigated, our server infrastructure would have to be updated with newer, faster processors in order to be able to continue scaling the simulation sizes and maintain the execution time under a feasible extent.

Apart from *BLonD++* scalability limitations, simulations of high-energy particle physics inside circular accelerators comprise a scientifically and computationally challenging task. These workloads are typically inherently parallel and fit naturally in a distributed-memory, weak scaling environment using MPI [70] (Message Passing Interface). Furthermore, strong-scaling within a node is also essential. As a result, a hybrid, distributed-shared programming model combined with hardware-specific code tuning is required to extract maximum performance out of modern CPUs. Optimizing and tuning scientific codes, while aided by compiler features, has become a highly manual endeavor that requires integrating awareness of the application-specific features and the underlying system architecture: the memory subsystem, the execution pipeline, the concurrent threads, and the SIMD vector units [46].

To undertake this agenda, in this section we design, optimize and evaluate

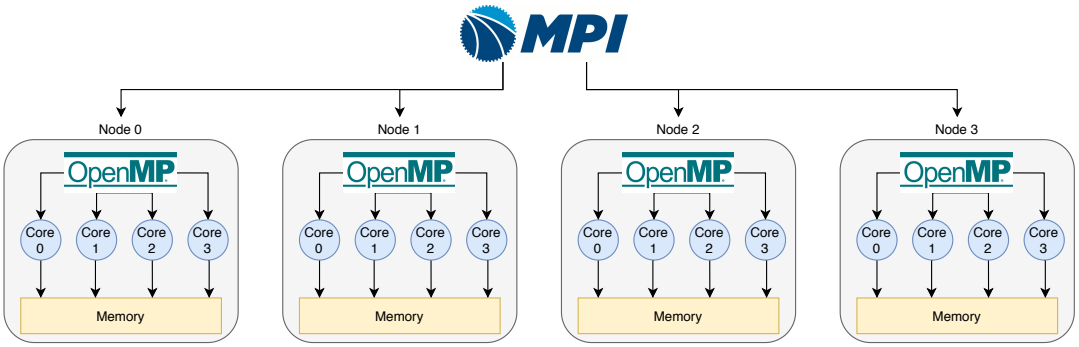



FIGURE 5.1: The hybrid, MPI-over-OpenMP *HBLonD* architecture.

*HBLonD*, a hybrid, MPI-over-OpenMP system for running longitudinal beam dynamics simulations. As seen in Fig. 5.1, *HBLonD* relies on MPI [70] to enable inter-node communication and cooperation between remote computing nodes. *HBLonD* is built on top of *BLonD++*, therefore OpenMP [12] is used for intra-node parallelization. We apply three software optimisation techniques to minimize the inter-node communication and synchronization overhead. At first, we develop a mixed data- and task-parallel model. Data parallelism is used across the nodes and task parallelism is used among the intra-node MPI processes to profit from fast shared-memory communication. Then, we discuss and evaluate two traffic optimisation and relaxed synchronisation techniques, motivated by the modelled physics phenomena. Finally, we develop a dynamic load-balancing scheme that periodically re-shuffles the workload to ensure that all the worker processes progress at the same rate. *HBLonD* is evaluated against the previous state-of-the-art *BLonD++* [58], using three real-word simulation scenarios, each targeting a different particle accelerator. We show that *HBLonD* can successfully scale up to 32 nodes or 640 cores, showing an average 40.5x speedup against *BLonD++*.

### 5.1.1 Real-World Testcases

Three real-world testcases, representative of typical *BLonD* workloads were used for the performance evaluation and optimization of the *HBLonD* MPI-over-OpenMP distributed system. Each testcase targets a different synchrotron in the CERN complex, and has its own distinct characteristics. The execution time of a simulation highly depends on the modelled physics, and the modules the user chooses to combine. A short description of the three real-world testcases follows.

**The Proton Synchrotron (PS) testcase** The PS is the second synchrotron of the LHC injector chain. It is CERN's oldest synchrotron and has a circumference of 628 m. The PS can accelerate protons up to an energy of 26 GeV, before injecting them into the SPS. Due to its relatively small size and closely spaced bunches, the PS is dominated by collective effects, meaning that leading particles of the beam affect the dynamics of trailing particles. The PS can accelerate up to 18 bunches simultaneously. The aim of the PS testcase we benchmarked is to study and control the beam instabilities that can manifest due to collective effects. The characteristic timescale of the beam motion in the PS is longer than in the other two real-world test cases, resulting in a slower moving, more "rigid" dynamics.

**The Super Proton Synchrotron (SPS) testcase** The SPS receives bunches of charged particles from the PS, accelerates them to 450 GeV and delivers them to the LHC or other experiments. With a circumference of 7 km it is the second-largest machine in the CERN's accelerator complex and one of the largest machines worldwide. For the LHC-type beam, the SPS can receive up to four batches of 72 bunches, or 288 bunches in total. In the testcase used for the evaluation [43], due to the very high number of particles and bunches in the machine, collective effects were an important limitation. Furthermore, the detailed impedance model requires very fine frequency sampling rate, calling for time-consuming FFT operations on large input arrays. Finally, the beam phase loop, a dynamic feedback system required to correct the bunch phase with respect to the RF system, adds to the overall simulation complexity.

**The Large Hadron Collider (LHC) testcase** With a circumference of 27 km and a collision energy at 13 TeV, the LHC is the world's largest and most powerful particle collider. The LHC can fit up to 2808 bunches of charged particles. The testcase used in the evaluation [71, 45] describes one of the critical elements of machine operation, which is the controlled emittance blow-up during the acceleration ramp, required for beam stability. An RF noise is generated and applied turn by turn for the so-called controlled emittance blow-up, which is a stochastic process that increases the bunch size and largely affects the simulation results. As the LHC uses super-conducting magnets, the acceleration ramp to 6.5 TeV takes about 13 million simulation steps, making the simulation extremely time consuming, even for the smallest sizes using a single bunch.

## 5.2 Base Distributed Implementation

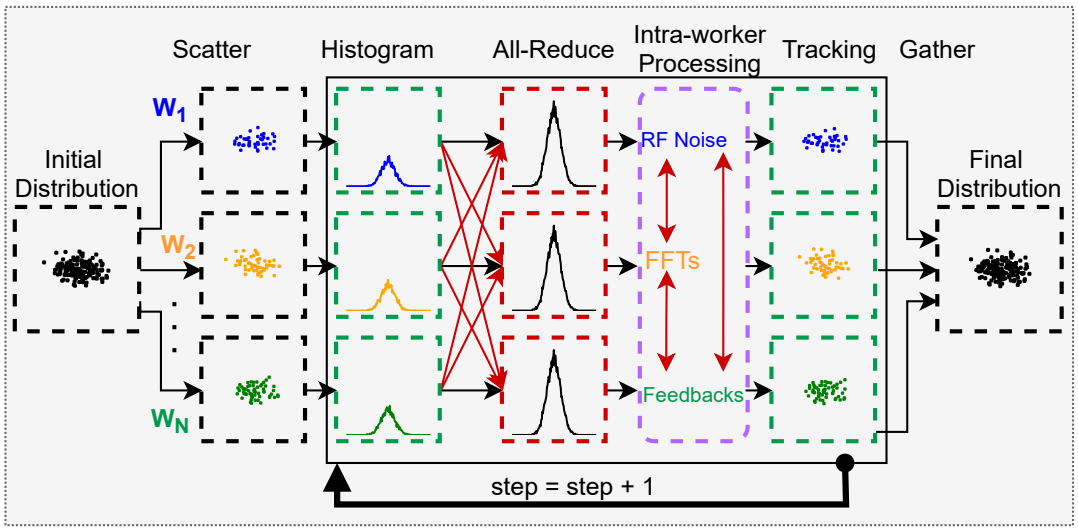



FIGURE 5.2: The baseline *HBLonD* workflow. A weak-scaling, data-parallel model is used to scatter the workload among the available MPI processes.

In Fig. 5.2, the baseline *HBLonD* execution flow is displayed. A typical simulation begins with the generation of the 2-D particle distribution. For this purpose, an extensive collection of particle distribution generation methods is included in *BLonD*. Since this initialization step is executed only once, it is not parallelized with MPI. The particle distribution is then scattered equally among the available MPI workers which reside in the same or remote nodes. As seen in Sec. 5.6.3, one worker per NUMA socket is providing the best and most stable performance.

Each worker runs a histogram operation to calculate the so-called beam profile of its assigned particles. All the local profiles are summed in order to generate the global beam profile that is needed for the subsequent stages. The global beam profile is used as input to the intra-worker processing stage. This stage is composed of a collection of operations that can only be parallelized among the threads of a worker but not across workers. Every worker is running the same sequence of operations on the same input, producing the exact same output. Even though it might seem as a redundant operation and a waste of resources, it is actually more efficient than having only one worker to calculate these tasks and then broadcast the result, since the remaining workers would be idling and also they would have to communicate.

Then, the workers perform the particle tracking, which essentially updates the particle coordinates according to the equations of motion, for all their assigned particles. The *kick* and *drift* are the two routines that absolutely need

to be applied during the tracking stage, as discussed in Sec. 3.2. Their computational complexity is linear w.r.t. the number of simulated macro-particles. This succession of operations from histogram to tracking is repeated hundreds of thousands or millions times, before the final gather operation that assembles the resulting particle distribution that is one of the main outcomes of a *BLonD* simulation. The percentage that each of the above stages allocate in typical simulation scenarios is discussed in Sec. 5.7.

In *HBLonD*, a series of optimization techniques were developed with the aim of minimizing the communication and intra-worker processing stage time. All those techniques are compatible with each other and their activation can be easily controlled via a set of command-line arguments.

## 5.3 Mixed Data and Task Parallelism

*HBLonD* is using a data-parallel model that distributes the input data (particle coordinates) across the available workers and lets every worker operate mainly on their local data. Generally, the stages of the main execution pipeline cannot profit from task parallelism since the output of each stage is required before the subsequent stage can begin executing.

However, among the various tasks of the intra-worker processing stage, there is some degree of task-parallelism that can be exploited. Typically, three or more tasks need to be calculated during this stage, depending on the use-case. The tasks are divided into two groups and assigned to every pair of MPI processes sharing the same node, as seen in Fig. 5.3. When both processes finish calculating the assigned tasks, they exchange the calculated data using fast shared-memory message passing, since they share the same computing node and memory hierarchy. The mixed data- and task-parallel model is evaluated experimentally in Sec. 5.6.4.

## 5.4 Approximate Computing Techniques

### 5.4.1 Description

To further optimize the inter-node network traffic and extract higher scalability and speedups, we developed two computing techniques motivated by the physics of the simulated phenomena. These techniques trade off prediction accuracy, which is strongly case dependent, for performance gain. Therefore, they

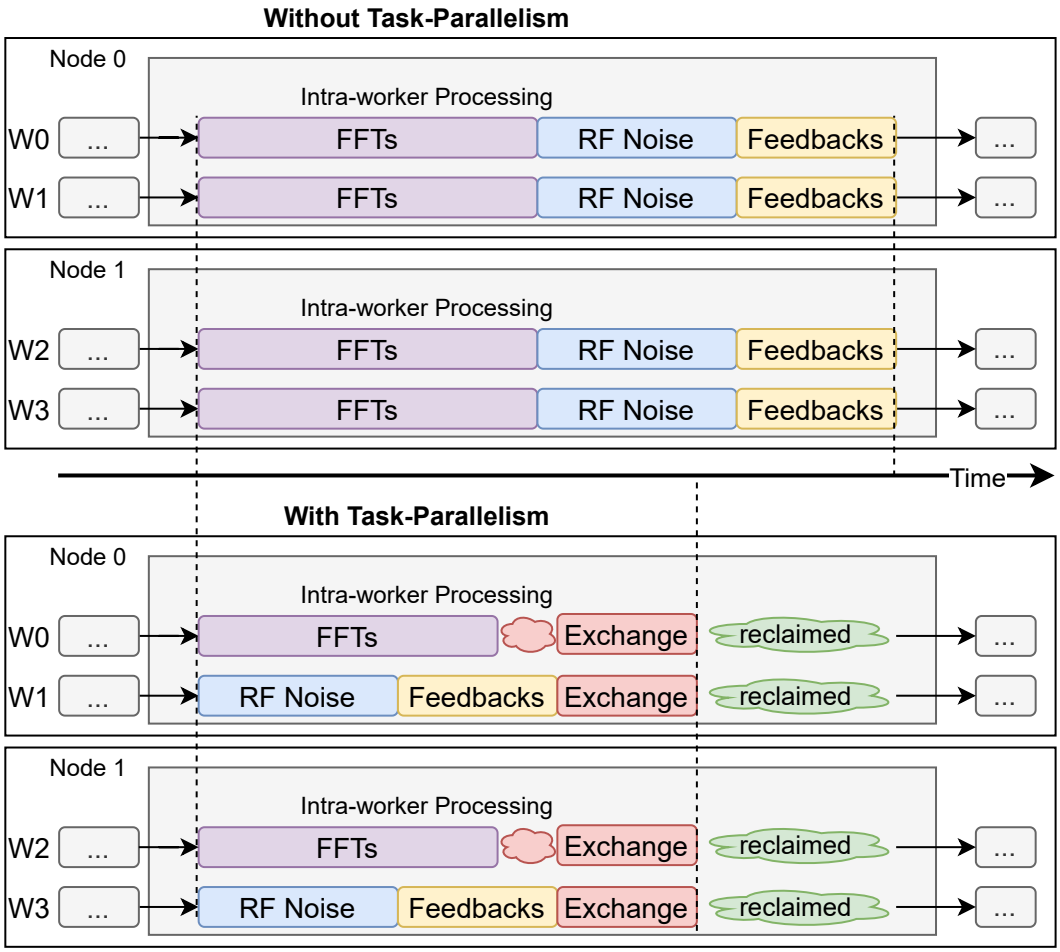



FIGURE 5.3: Task-parallelism exploited by neighbouring MPI processes during the intra-worker processing stage.

need to be controlled by experienced scientists that deeply understand the simulated problem and can decide which measure of error should be calculated and whether they can afford a loss in prediction accuracy for improved performance.

**Representative Distribution Subset (RDS)** In the beam distribution scatter phase, a large number of particles, in the order of 100s of millions, is distributed among a small number of MPI workers. Consequently, every worker is assigned 10s or 100s of million particles. This traffic optimisation method is based on the assumption that each worker has been assigned a representative subset of the particles that describes the overall distribution adequately. To ensure that the subset of particles assigned to each worker represents the features of the original particle distribution, the particles are scattered among the available workers randomly. As seen in Fig. 5.4 and expressed in Eq. 5.1 to approximate the global beam profile, worker $i$ can simply scale up the local beam profile by the ratio of all particles to the assigned particles:

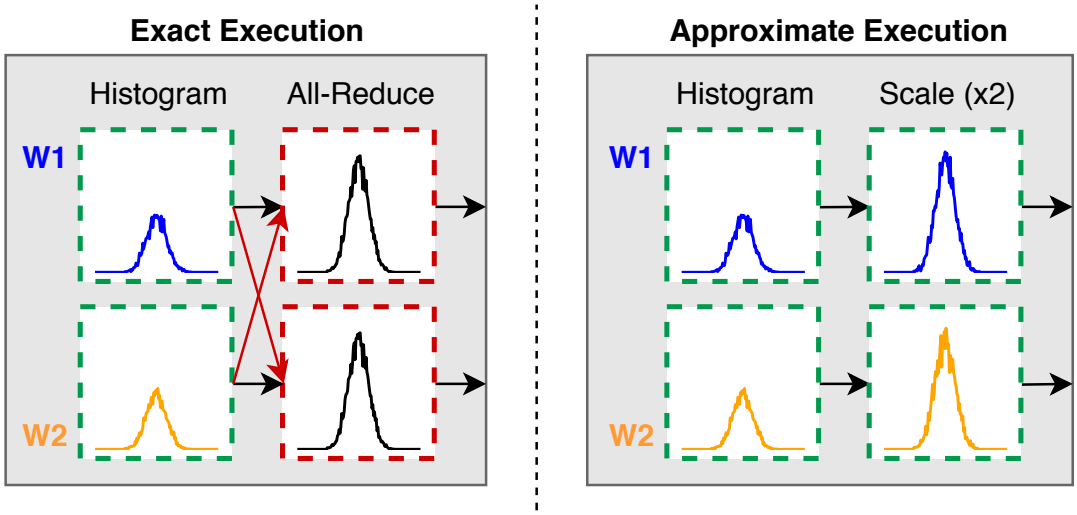



FIGURE 5.4: RDS method: Assuming that each MPI worker is assigned subset of the particle distribution that is representative of the whole, the all-to-all global reduction can be replaced by a less costly scale-up operation.

$$globalProfile \equiv \sum_{i=0}^{N} profile_i \simeq profile_k \times \frac{\sum_{i=0}^{N} particles_i}{particles_i} \tag{5.1}$$

where $N$ is the total number of workers, $particles_i$ are the particles assigned to worker $i$, and $profile_i$ is the local beam profile of worker $i$. As a consequence, the costly all-to-all beam profile reduction is avoided, the workers' execution is disengaged, and the time needed for communication and synchronization among them is greatly reduced.

**Smoothly Revolving Profile (SRP)** In the 2D time and energy phase space, the particles are slowly revolving around the synchronous point with every simulation step. This approximation method is based on the assumption that the beam profile is not changing rapidly between consecutive steps, which is generally true for the slow synchrotron motion of particles, in the absence of specific scenarios e.g. fast beam instabilities. Indeed, as depicted in Fig. 5.5, in a real-world scenario with a sufficient number of simulated particles per histogram bin, the per-turn variation of the beam profile is limited to 1.3% on average. This condition needs to be verified by the user before enabling the approximation. SRP updates the beam profile every K iterations as shown in Eq. 5.2, which is generally tolerable in terms of precision for small K values, e.g. two or three.

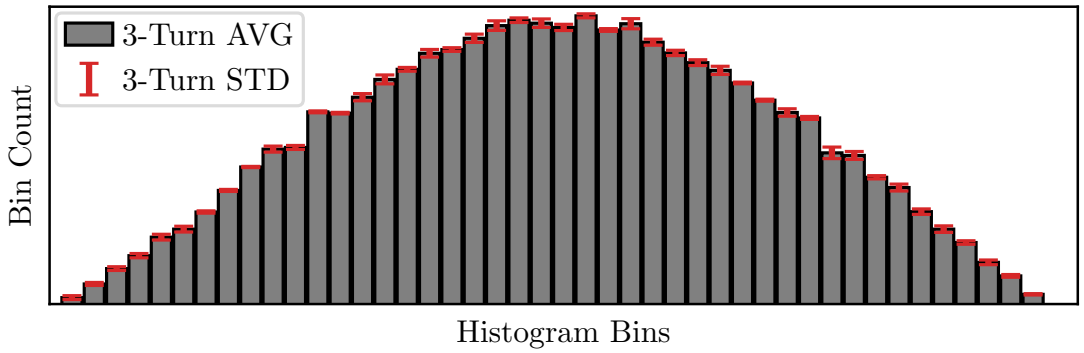



FIGURE 5.5: The beam profile is not rapidly changing between consecutive turns. Therefore, applying the SRP method and updating the profile every two or three iterations is affordable in terms of simulation accuracy.

$$profile^t = profile^{t+1} = \cdots = profile^{t+K-1}, t \bmod K = 0, \tag{5.2}$$

where $profile^t$ is the beam profile of turn $t$. By doing so, the need to perform the histogram calculation and the costly, communication-heavy global reduction in every turn is eliminated, reducing the simulation latency significantly.

**Floating Point Arithmetic Precision** Another typical performance-accuracy trade-off lies in the floating point datatype size. *BLonD* traditionally uses 64-bit floating point numbers to ensure maximum accuracy in the calculations involved in *BLonD* simulations. However, using 32-bit floating point numbers can provide significant latency gains, without major accuracy loss. In the 32-bit *HBLonD* version, the datatype length of all floating point variables is reduced from 64-bit to 32-bit. Additionally, all the routines that were previously operating on 64-bit floats have been adjusted to operate on shorter, 32-bit wide floats. The two previous approximate computing techniques, i.e. RDS and SRP, can provide additional execution latency gain when combined with the 32-bit floating point datatype, as shown in Fig. 5.15.

### 5.4.2 Accuracy Loss Evaluation

In this section we evaluate the approximate computing techniques detailed above in terms of accuracy loss. The first two techniques, namely RDS and SRP, can be combined with 32-bit floating point size (f32), therefore in total five different approximate variations are considered: SRP, SRP-f32, RDS, RDS-f32 and f32.

Naturally, the five methods described above produce different simulation results compared to the baseline, approximate-free version. To be able to evaluate the magnitude of this difference, we put it in perspective with the statistical fluctuations induced by altering the input seed of the random number generator function used during the particle distribution instantiation. This statistical fluctuation is considered acceptable, when using a large enough number of simulated macro-particles.

In Figs. 5.6 and 5.7, there are three data-points in addition to the five approximate variations. *Base* corresponds to the approximate-free, exact simulation. The *Ref1* and *Ref2* data-points are also non-approximate, but have slightly altered initial particle distribution. More specifically, as part of the particle coordinates generation process, a Gaussian random number generator is used. Passing a different seed value to the generator results in a distinct set of particle coordinates, which follows the same Gaussian distribution. By using a large enough number of particles, the statistical deviation of two particle distributions with a different seed is minimized. The only difference between *Ref1*, *Ref2* and *Base* is the seed value used for the generation of the input particle distribution. To limit the statistical fluctuation deriving from the exact instantiation of the particle coordinates, large numbers of particles were used in the experiments shown in Figs. 5.6 and 5.7, i.e. four million.

Fig. 5.6 shows the accuracy loss in the energy coordinate while Fig. 5.6 shows the accuracy loss in the time coordinate. In both figures, the cross points correspond to the distribution's average after a relatively large period of simulated turns (i.e. approximately 40 thousand). The errorbars show the per-turn fluctuation of the distribution's average in the last few turns. As described above, the *Base, Ref1*, and *Ref2* datapoints are basically equivalent, and their differences reflect the statistical fluctuation noise.

The best agreement can be observed in the PS testcase, where both the average values and the errorbars of the base, reference and approximate datapoints are essentially identical. This behavior derives from the fact that in the PS simulation scenario the particle distrubution is slowly moving and more resilient to micro-approximations. As a consequence, the PS testcase is a good candidate for more aggressive approximations in order to save execution time.

On the other hand, in the time coordinate evaluation of the LHC testcase (Fig. 5.6), we observe the greatest variation between the base, reference and approximate values. This is related to the RF noise diffusion phenomenon present in this specific testcase. The RF noise is also generated based on a random number generator, and the final particle distribution depends on the generated noise sequence. The effect of the applied noise largely affects the simulation's

dynamics. As a consequence, it overshadows the final result variation coming from both the seed derived statistical fluctuation, and the approximate techniques. Moreover, by observing the error bars of all three approximate versions using 32-bit float (i.e. *SRP-f32*, *RDS-f32*, and *f32*), we see that reducing the data-type precision introduces a significant loss. This is due to the fact that 32-bit floats are used for the RF noise, which as said above, largely affects the energy and time coordinates. Nevertheless, Fig. 5.6 shows that the base point is closer to the approximate points than the two reference points for the LHC testcase. This means that due to the presence of the RF noise, the LHC testcase is more sensitive to the input particle distribution, and that the deviation deriving from the approximations is less significant than the deviation deriving from the particle distribution generation.

Finally, the SPS testcase falls within the two extremes, and shows mostly good agreement between the approximate and reference points, and the base simulation results. In the SPS testcase, it can be seen that the *RDS* approximation method reduces the per-turn variation errorbars both in terms of energy and time. The *RDS* method decouples the MPI workers and lets them approximate the global beam profile based on their local beam profile. This effectively reduces the per-worker distribution size, which is possibly the reason why the pet-turn variation errorbars appear to be narrower.

In conclusion, all the five approximate computing techniques presented above provide acceptable agreement with the exact, non-approximate simulations. Their error magnitude is strongly testcase dependent. Approximate computing is therefore a valuable tool for the expert users of *BLonD*, and can be used to reduce the run-time of lengthy simulations, as will be shown later in Sec. 5.6.5.

## 5.5 Dynamic Load Balancing

During the evaluation process, an occasional increase in the latency of some nodes was noticed, despite the fact that the computing cluster used for development and evaluation was composed of homogeneous hardware. The delayed nodes would require significantly more time than others to execute the same amount of workload. This increased latency appeared to be spontaneous and temporal, but in most of the cases it would persist for small or medium periods of time. As a result, the execution was imbalanced. Since in every iteration all the workers need to synchronize (upper half of Fig. 5.8), the latency of the slowest worker is experienced by all workers in a given simulation step. The load imbalance was not coming from the characteristics of the data, since the

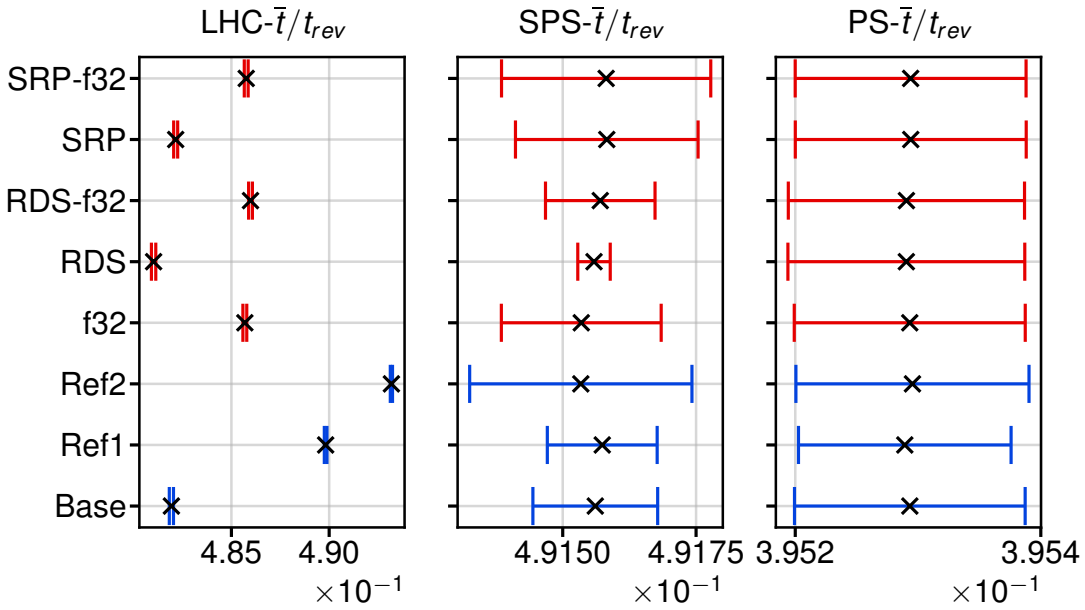



FIGURE 5.6: Accuracy loss evaluation of the approximate computing techniques. The approximate-free baseline is the *base* data-point. The *Ref1* and *Ref2* data-points are used as reference points for the approximate techniques.

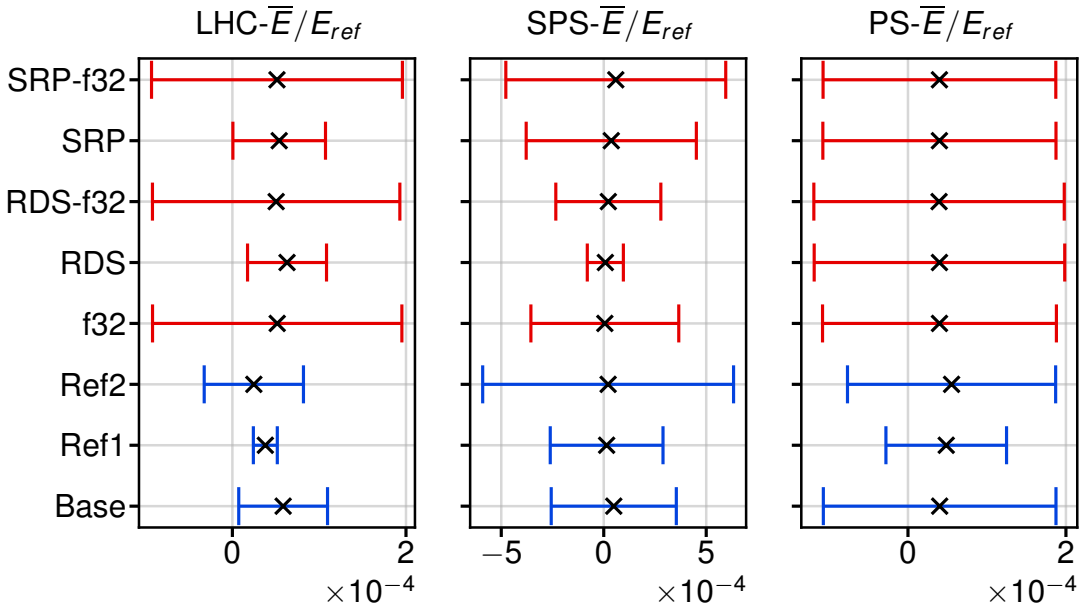



FIGURE 5.7: Accuracy loss evaluation of the approximate computing techniques. The approximate-free baseline is the *base* data-point. The *Ref1* and *Ref2* data-points are used as reference points for the approximate techniques.

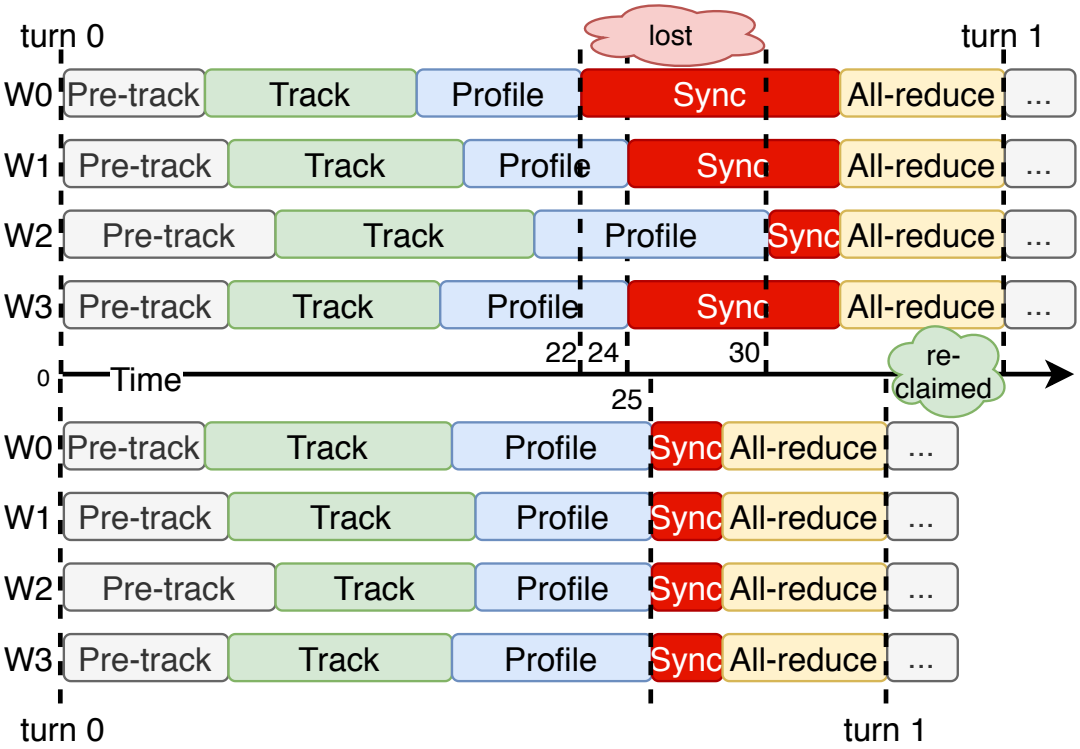
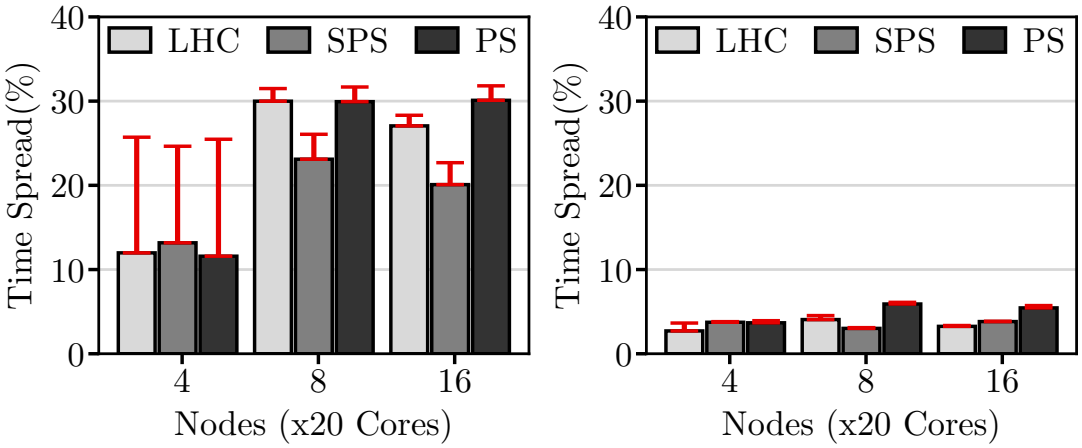



FIGURE 5.8: The observed per-turn latency is defined by the slowest worker, therefore a slow worker is enough to reduce the performance of the whole system.

same operations are applied universally to all available data, but from the characteristics of the cluster and the nodes.

Slow workers would experience near-zero waiting times at the synchronization barriers while faster workers would experience a larger waiting time. By measuring the time the workers spent synchronizing, we can calculate the spread of the workers' execution time. Using this metric we can evaluate if and how imbalanced a workload is. Figure 5.9a shows the time spread, normalized to the total execution time for three *HBLonD* testcases (described in Sec. 5.1.1) when running two-, four-, eight- or 16-node simulations. Multiple runs per testcase and number of nodes were executed, and the red capped lines show the standard deviation of these runs. The time spread ranges from 11.6% to 30.1%. Furthermore, we observe that when using more nodes, the time spread is increasing since, statistically, the chances that one or more workers will undergo a delayed phase increase. On the other hand, while in the smaller node configurations the average time spread is low, the standard deviation is much larger. This means that most of the runs did not experience any increased latency phases, but there were still some few cases with severe load imbalance.

To mitigate this issue, we developed a Dynamic Load-Balancing (DLB) scheme. The proposed DLB scheme is generic enough to alleviate imbalances




(A) Time spread without DLB.

(B) Time spread with DLB.

FIGURE 5.9: The difference in run time among the workers, normalized to the total execution time. Without DLB, the time spread ranges from 11.6%-30.1%. With DLB, it is limited to 2.7%-5.9%.

that originate from various sources such as the cluster's topology, the interconnection network, and the involvement of heterogeneous hardware. The proposed DLB scheme has been customized to the following *HBLonD*'s specific workload properties:

1. $tComp_i = p_i \times m_i + c_i$: The computation time of worker $i$ is linearly associated with the number of particles ($p_i$). The computation time is considered the time needed to perform the particle tracking (*kick* and *drift* methods) as well as to calculate the beam profile (*histogram* operation). All these methods have a linear time complexity with the number of input particles. The first subplot of Fig. 5.10 validates experimentally this assumption.

2. $tComm_i = const, tIntra_i = const$: The nodes only need to communicate the beam profile and not the beam coordinates, thus the communication ($tComm$) and intra-node ($tIntra$) processing time are independent of the number of particles. The length of the beam profile is equal to the number of histogram bins, which is typically much smaller, of the order of 1‰, than the number of particles. The second and third subplots of Fig. 5.10 shows that the communication and intra-node processing time are basically independent of the particles' size.

3. Perfect load balance $\Leftrightarrow tSync_i \to 0$: In a perfectly load-balanced scenario, all workers are in sync and experience near-zero synchronization time ($tSync_i$).

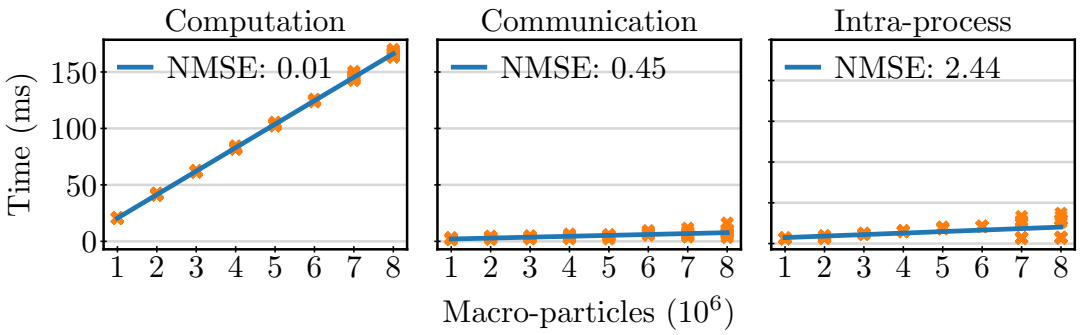



FIGURE 5.10: Experimental verification of the two first workload assumptions on which the DLB scheme is based.

4. Once a worker enters a slower than usual execution phase, it maintains this status for a medium or long period of time.

According to the first property, the workers by measuring their computation time, they can apply a least-squares, 1st-degree polynomial fit method to calculate the slope ($m_i$) and y-intersect ($c_i$) coefficients. A weighted polynomial solver is used, that takes into account a fixed number of past measurements. To allow for swift reaction to developing imbalances, greater weight is given to the more recent data points. An exponential decay function is used to calculate the weights $W$, given by the following equation:

$$W[k] = e^{-(H-k)/DC}, k \in \{1...H\}\,, \tag{5.3}$$

where $H$ is the number of historic points considered and $DC$ is the decay coefficient.

Due to the synchronization barriers before the communication phase, all workers will experience the same per-turn latency $T$:

$$tComp_i + tIntra_i + tSync_i = T, \xrightarrow{\text{property 1}} \tag{5.4}$$

$$p_i \times m_i + c_i + tIntra_i + tSync_i = T. \tag{5.5}$$

The $tIntra_i$ can be also measured by each worker, and since it is a constant independent of the particles, let us incorporate it in $c_i$:

$$p_i \times m_i + c_i + tSync_i = T. \tag{5.6}$$

Faster workers will experience longer synchronization time so that the turn latency of all workers is equal to $T$. According to the 3rd property, we want to

TABLE 5.1: Dynamic-Load Balancing Scheme Configurable Parameters.

| Name | Default | Description |
|---|---|---|
| Rebalance Period | 1000 | Number of turns between two re-balancing operations. |
| $Tx_{min}$ | 3% | Min. transaction size as a percentage of the total particles. |
| $P_{min}$ | 10% | Min. percentage of particles assigned to a worker. |
| History length | 20 | Number of points to consider in polynomial fitting. |
| Decay Coefficient | 5 | Decay coefficient used in the exponential weight function. |

find a new set of $(p_i^{'}, T^{'})$ that makes $tSync_i^{'} = 0$:

$$p_i^{'} \times m_i + c_i = T^{'}, \text{ and} \tag{5.7}$$

$$P = \sum_{i=0}^{N} p_i^i, \tag{5.8}$$

where $P$ is the total number of particles that remains constant, and $N$ is the number of workers. This is a set of $N+1$ equations with $N+1$ variables. By solving this set of equations we get:

$$p_i^{'} = \frac{P + sum_1 - c_i \times sum_2}{m_i \times sum_2}, \text{ where} \tag{5.9}$$

$$sum_1 = \sum_{i=0}^{N} \frac{c_i}{m_i}, \text{and } sum_2 = \sum_{i=0}^{N} \frac{1}{m_i}. \tag{5.10}$$

Knowing the number of particles each worker should be assigned in order to simultaneously arrive at the synchronization barrier (bottom half of Fig. 5.8), slower-than-average workers have to offload a portion of their workload to faster-than-average workers. This set of transactions is calculated by minimizing data traffic[72] and prioritizing transactions within the same node. The workers calculate the particles they need to send or receive from other workers. After completing these transactions, the workers continue with the next iteration of the simulation. This process is repeated periodically. The DLB scheme is highly customizable, and able to cover a wide range of load-imbalance scenarios. The key configurable parameters and their default values, that were identified by exhaustive exploration, are summarized in Table 5.1. For optimal and portable performance, these DLB parameters must be fine-tuned under a specific workload, load imbalance scenario, and cluster configuration.

Figure 5.9b shows the normalized spread in time among the MPI workers when the DLB mechanism is enabled. It is evident that the imbalance among the workers has been minimized both in smaller and larger node configurations. The time spread is limited to 2.7% - 5.9%, i.e. 5× lower than without the DLB mechanism. Furthermore, the more balanced workload brings 17% gain

in execution time on average across three real-world simulation cases. The overhead of the DLB scheme is limited to 0.4% for the polynomial fit and $p_i^{'}$ calculation, and 1.1% for the particle transactions, so in total 1.5% of the total execution time.

## 5.6 Fine-Tuning and Sensitivity Analysis

### 5.6.1 Experimental Setup Configuration

All reported experiments took place in an HPC cluster hosted at CERN. Every node contains two sockets, equipped with a 2.2 GHz 10-core Intel Xeon E5-2630v4, Broadwell micro-architecture [73] processor and 64 GB of RAM. The nodes are connected using Infiniband [74] and run CentOS Linux 7. Different configuration containing up to 32 such nodes, or 640 cores were evaluated. All the data points appearing in this section's figures represent the average value of ten identical runs. The standard deviation was most of the time less than one percent.

### 5.6.2 Benchmarking MPI Implementations

In this section, we benchmark three of the most widely used MPI flavours: i) MVAPICH2 [75], ii) OPENMPI3 [76] and iii) MPICH3 [70], in order to determine which is performing best in terms of elapsed time for our testcases and cluster configuration. Figure 5.11 shows the run time of each testcase, using each of the three different MPI implementations, normalized to the run time of MVAPICH2, thus larger values correspond to longer execution times. The main MPI operations used in HBLonD are the point-to-point *Isend()*, *Ireceive()* and *sendrecv()*, as well as the collective *allreduce()* and *allgather()*.

The rightmost group of bars in Fig. 5.11 shows the average, normalized run time values for all three testcases. We see that MPICH3 is on average 11% slower than MVAPICH2, and OPENMPI3 is 17% slower than MVAPICH2. In addition, MVAPICH2 provided the most stable and reproducible run-times. Thus, in the following experiments of this section and Sec. 5.7, the MVAPICH2 MPI implementation will be used.

### 5.6.3 Workers-per-Node Sensitivity Analysis

Apart from the MPI implementation, the number of MPI workers-per-node (WPN) is another important configurable value that can greatly affect the performance. In Fig. 5.12, we executed all three testcases using a single, two, four

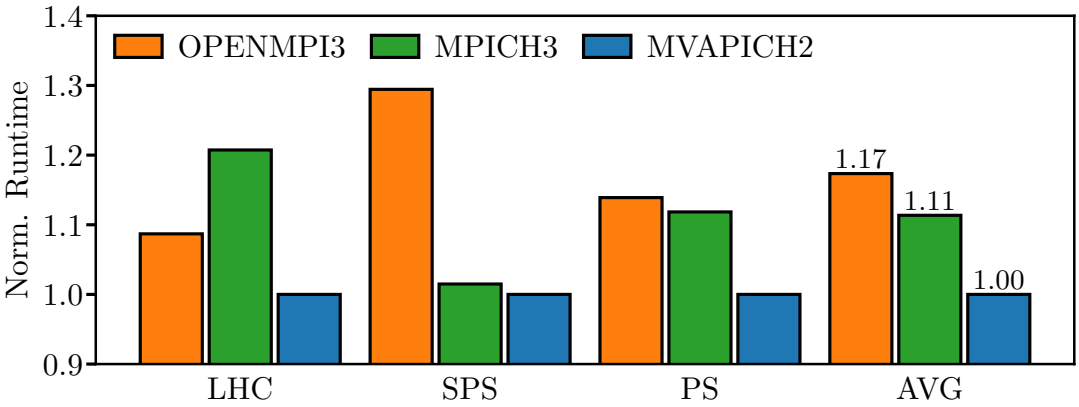



FIGURE 5.11: Performance benchmarking of three MPI implementations: openmpi3, mpich3 and mvapich2. The run times are normalized to the mvapich2 run time.

or ten WPN, with 20, ten, five or two hardware threads available to each worker, respectively, since every node contains 20 cores in total. Generally, a very large number of workers might increase the communication time and negatively affect the performance. On the other hand, a very small number of workers might also perform poorly due to sub-optimal memory usage [77].

The cluster used in the experimental evaluation is composed of dual-socket servers. Each socket forms a separate NUMA domain or locality group [78], therefore we chose WPN values that allow for all the threads that are assigned to a worker to remain in the same locality group. For instance, a value of five WPN would mean that each worker would spawn four threads and one of them would have two threads in each locality group. This can hurt the performance and lead to a load imbalance among the workers within a node [79, 80]. In Fig. 5.12, the reported run times are normalized to the run time of the configuration using ten MPI WPN. We observe that the outcome of all testcases is qualitatively consistent; the dual WPN (or one worker per locality group) performs best, followed by the four WPN configuration, then the single WPN and finally the ten WPN configuration. Since using one worker per locality group consistently performs best, in the following experiments we will be always using one MPI process per locality group.

### 5.6.4 Task-Parallelism and Load-Balancing Evaluation

As described earlier in Sec. 5.5, a mixed data- and task-parallel model is used in *HBLonD* to reduce the time spent in the intra-worker processing stage. Furthermore, to deal with the spontaneous load imbalance that was observed, a dynamic load-balancing scheme was integrated, that periodically evaluates the processing rate of the workers and re-distributes the particles assigned to them

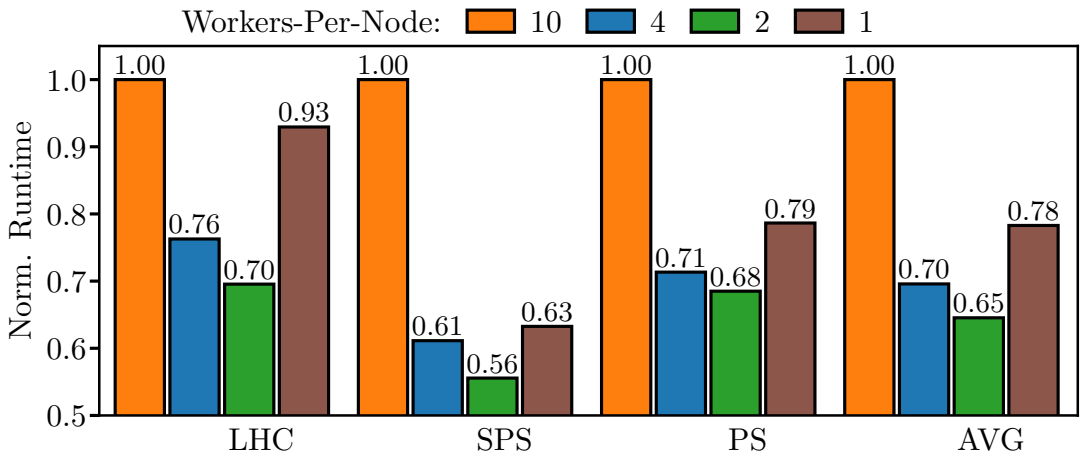



FIGURE 5.12: Sensitivity to the number of MPI workers-per-node (WPN). Two WPN provide the best performance across all testcases. The run times are normalized to that of the ten WPN configuration.

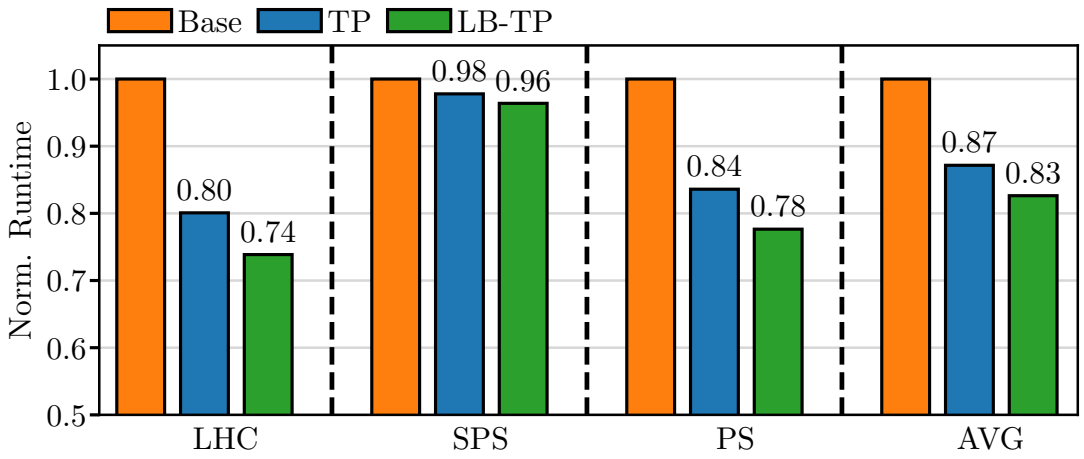



FIGURE 5.13: Performance gains of the individual optimization techniques. Run times are normalized to the run time of the baseline *HBLonD*.

accordingly, in order to ensure minimum waiting time at the synchronization barriers.

The effect these techniques have on the run time can be seen in Fig. 5.13. In this figure, *base* corresponds to the baseline *HBLonD* without any additional optimization. *LB* stands for Load-Balance and *TP* stands for Task-Parallelism. Hence, *LB-TP* is the exact, fully optimized *HBLonD*. In Fig. 5.13, all the results were collected using eight computing nodes or 160 cores. With fewer nodes, the advantageous effects of some of the optimisation techniques were not evident. This was anticipated since our techniques aim to reduce the time lost in communication and non-parallelizable regions, which gets more significant in larger configurations. The run times have been normalized to the run time of the baseline *HBLonD*, thus lower values signify faster execution times.

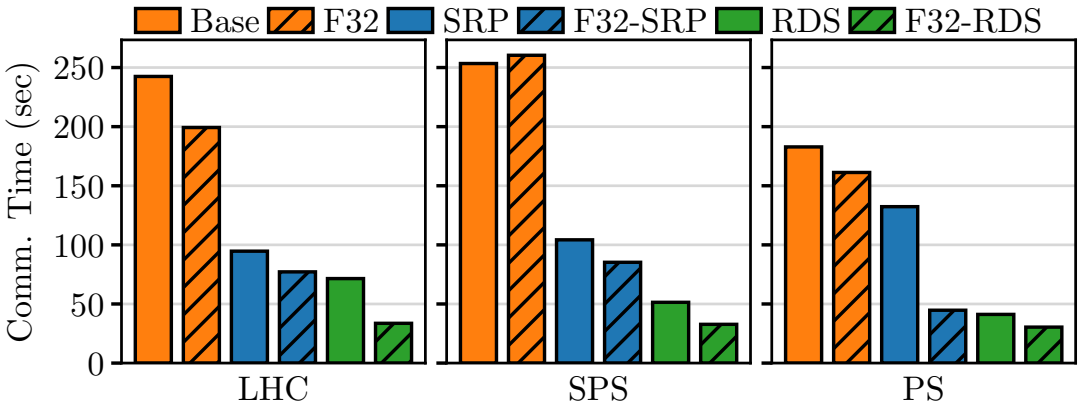



FIGURE 5.14: Time spend for inter-node communication with the various approximate computing techniques.

From a qualitative point of view, a consistent behavior across the three real-world testcases is observed. *HBLonD* with *TP* and *LB-TP* is on average faster than the baseline by 13% and 17%, respectively. This is a significant improvement considering that *HBLonD* has been based on *BLonD++* [58] – a well optimized software.

### 5.6.5 Approximate Computing Evaluation

In Sec. 5.4.2, we described three approximate computing techniques that were designed to reduce the time spent on inter-node communication and other non-parallelized regions. The Smoothly Revolving Profile (SRP) and Representative Distribution Subset (RDS) techniques are motivated by the physics of the simulated process, and the characteristics of the specific test-case. The third technique, reducing the floating point data-type length, is a typical technique used to accelerate floating point arithmetic operations and reduce the application's memory footprint. Figure 5.14 demonstrates the total time allocated for inter-node data exchange in each of the three test-cases when applying the aforementioned approximation techniques, as well as the combination of SRP and F32 (F32-SRP) and RDS with F32 (F32-RDS). Eight nodes with 160 cores in total were used for this experiment. Using smaller datatype size is not affecting greatly the communication time, since most of the traffic has to do with the beam profile, which is an array of 32-bit integers. The SRP and RDS approximations reduce the communication time by a factor of 2.5×-5×. As expected, the RDS effects more the inter-node traffic since it eliminates the need for the costly all-to-all collective operation, while the SRP technique still applies the all-to-all operation, but less frequently than in the non-approximate baseline.

Fig. 5.15, summarizes the gain in runtime when applying the aforementioned

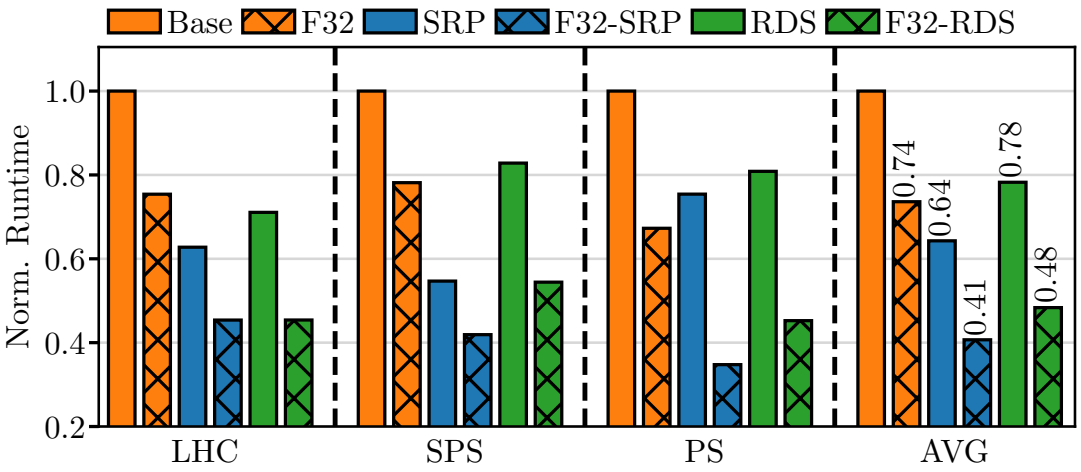



FIGURE 5.15: Approximate computing performance evaluation. 8 nodes or 160 cores were used.

approximation techniques, as well as the combination of SRP and F32 (F32-SRP) and RDS with F32 (F32-RDS). Eight nodes with 160 cores in total were used to perform this experiment. The y-axis shows the runtime, normalized to the base *HBLonD*, i.e. without any approximation applied. Lower values correspond to greater gains in latency. As expected, the combined approximate versions, F32-SRP and F32-RDS demonstrate the greatest gains in latency, 59% and 52% on average, respectively. SRP calculates the beam profile, which is one of the most time-consuming operations, only once every three turns. Furthermore, the costly all-to-all beam-profile reduction is only performed when the beam profile is updated, i.e. once in three turns.

We observe that the characteristics of the simulated scenario affect the latency gain provided by the various approximate techniques. In the PS testcase, for instance, using 32-bit floats instead of 64-bit floats provides 33% reduction in runtime. This is due to the fact that the particle tracking related operations, such as kick and drift, allocate a large percentage of the total execution time, and these operations profit the most from the reduced data-type length. In the SPS testcase, the runtime is reduced by 45% when updating the beam-profile periodically instead of in every turn. This is explained by the fact that the all-reduce and the induced voltage operations are calculated periodically, which jointly account for 40% of the execution time in the SPS testcase. Finally, in the LHC testcase the RDS method offers 21% latency gain. The RDS method allows the workers to operate independently by approximating the global beam profile based on their local beam profile. This completely eliminates the cost of the all-to-all reduction and any time lost in synchronization among the workers. In the LHC testcase, these two categories were allocating 39% of the total execution time.

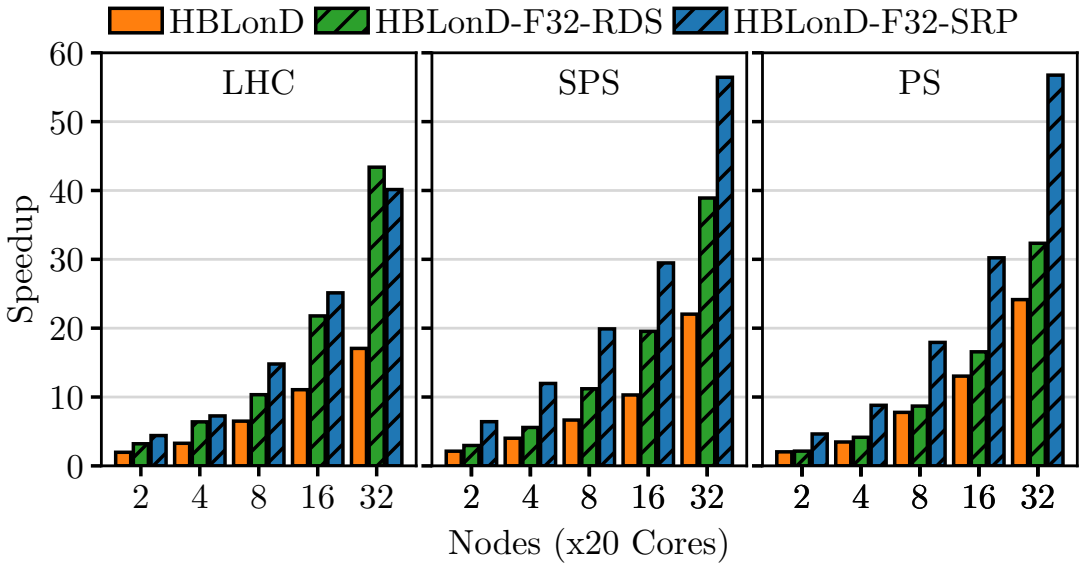



FIGURE 5.16: *HBLonD* strong scalability. The speedup shown is w.r.t a single-node, 20-core *BLonD++* instance.

## 5.7 Scalability Stressing

In this section, we present *HBLonD* performance under weak- and strong-scaling workloads. In strong-scaling experiments, the input size is held constant while the number of computing resources increases. The purpose of this study is to determine whether the code can effectively handle the additional resources or if the scalability saturates quickly, meaning that adding more resources cannot provide any further performance speedup. This usually happens when a portion of the workload is not parallelized, thus the speedup is upper-bounded by Amdahl's law [9].

Fig. 5.16 summarizes the strong-scaling results for the three real-world testcases. For every testcase, there are six lines, one for the baseline, optimized exact version, and one for every of the five optimized approximate versions. The number of nodes are shown on the x-axis and range from one to 16 nodes, with each node containing 20 cores. The speedup shown on the y-axis is w.r.t. to a single node *BLonD++* instance with 20 cores. We observe that up to 8 nodes, all versions including the exact *HBLonD*, demonstrate near-linear scalability. In the largest, 16 and 32 node configurations the speedup seems to saturate in the exact version. As previously stated, this is a direct product of Amdahl's law. However, with the aid of the approximate computing techniques, a speedup of 43.4×, 56.4× and 56.7× is extracted in the LHC, SPS and PS testcase, respectively. In some of the fewer-node configurations, we observe super-linear speedups of *HBLonD* compared to *BLonD++*. This is a product of the task-parallelism exploited by neighboring workers. With task-parallelism,

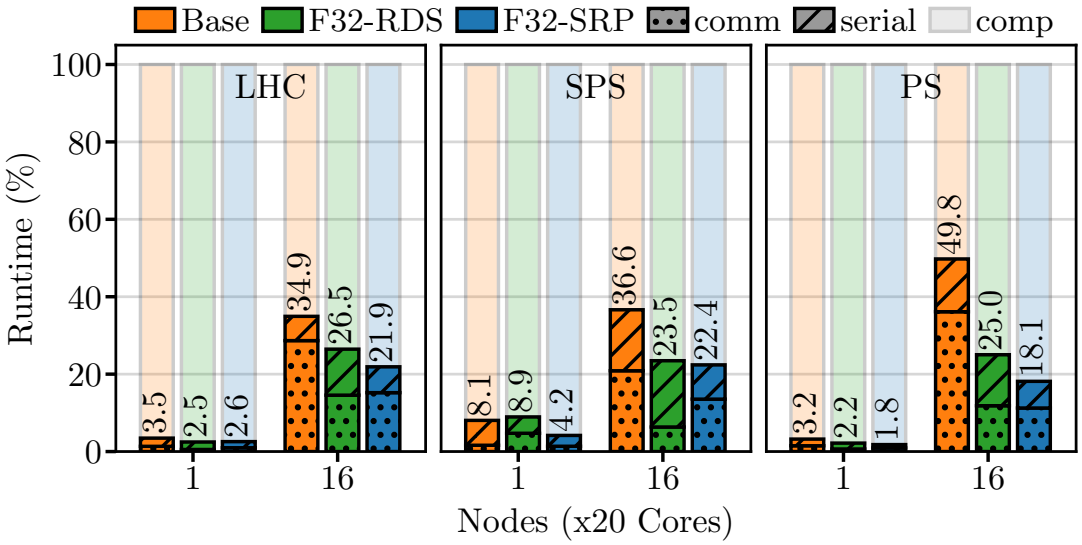



FIGURE 5.17: *HBLonD* execution time breakdown with and without approximate computing. As the node count increases, less time is spent on useful, fully parallelized computations, limiting the scalability.

each worker calculates a subset of all the non-parallelizable with MPI operations, and then neighboring workers exchange the calculated results profiting from fast, shared memory communication. These tasks demonstrate limited scalability, due to the nature of the algorithm or the input size. It is therefore most preferable to assign less cores to each task and calculate them in parallel, as is done in *HBLonD*, rather than using all the available cores in every task and calculating the various tasks sequentially, as in *BLonD++*.

In order to study the factors that limit the code's scalability, we have to compare the run-time breakdown of the *Base* implementation with the *F32-SRP* and *F32-RDS* implementations. In Fig. 5.17, the total runtime is divided into three categories: communication time (*comm*), time spent on operations that are parallelized only intra-node and not across nodes using MPI (*serial*), and the remaining is the computation time (*comp*) that scales with the number of particles and is parallelized both intra and inter-node.

Ideally, the MPI workers should mostly run fully parallelized computations and only spend a minimum percentage of their time for communication and intra-node processing. When using only one computing node, less than 9% of the run time of all testcases is spent for communication and intra-node processing. When the number of computing nodes increases to 16, approximately 21% to 36% of the run time of the *Base* version is used for communication and synchronization among the remote MPI workers. The percentage of the run time spent for intra-node processing in the *Base* version, ranges from 6% to 16%. The most time consuming intra-node processing task are the FFT computations that

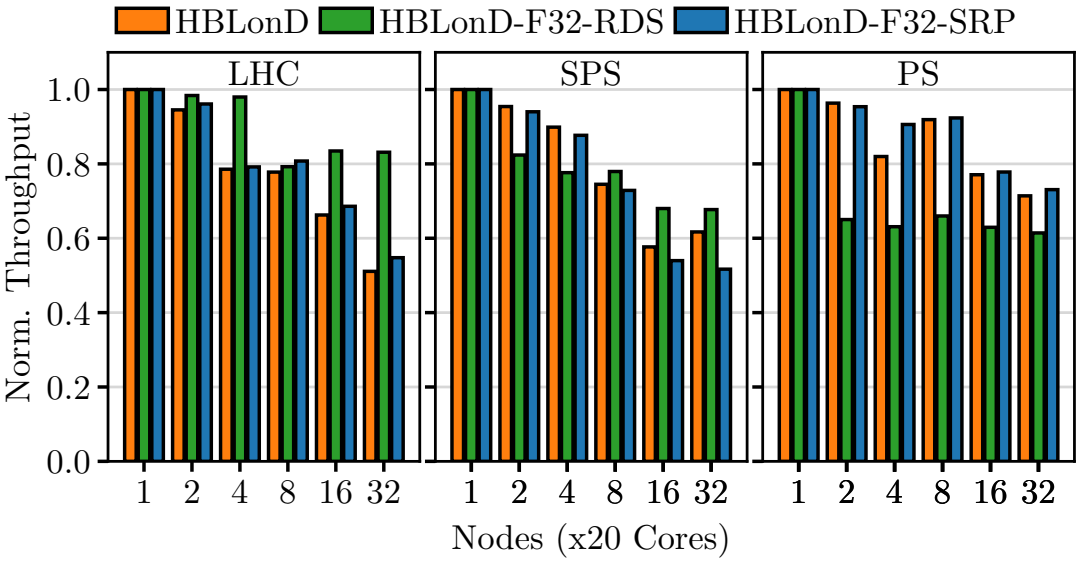



FIGURE 5.18: *HBLonD* weak scalability. The throughput is normalized to the number of nodes used.

take place during the induced voltage calculation. In the PS testcase, about half of the total run time is spent running useful parallel operation, and the other half is used for cooperation and non-scalable with MPI operations. This is limiting the scalability that can be exploited as the node count increases, as shown previously in Fig. 5.16. When the approximate variations are switched on, the communication time is effectively reduced to the range of 6% to 15%, much lower than in the baseline. The intra-node processing time remains at the same levels as in the baseline, ranging from 6.7% to 17%. Since not more than a quarter of the total run time is spent for non-parallelizable operation, we can expect the approximate variations to scale with a further increase in the number of computing nodes.

To further study *HBLonD*'s scalability, we stress its performance under a weak-scaling experiment. In weak-scaling experiments, the workload per node is kept constant as the number of nodes increases. In Fig. 5.18, the number of nodes ranges from one to 32, and each time the number of nodes is doubled, the input size is also doubled so that the workload per worker is constant. The y-axis shows the throughput per computational node, normalized to the throughput of the smallest configuration, that is the one using a single node. Ideally, the throughput per node should remain constant as the node count increases. However, this is not happening since the communication and synchronization time are increasing as more nodes are added. Therefore, with 32 nodes, the normalized throughput per node drops to 40-80%. Again, this is a consequence of Amdahl's law, the existence of non-parallelized code regions, as

TABLE 5.2: End-to-end real-world case studies run time with *BLonD++* on one node and *HBLonD* on 32 nodes.

| Case | Turns | Particles | *BLonD++* Run Time | *HBLonD* Run Time | Speedup 32 Nodes |
|---|---|---|---|---|---|
| LHC | 14M | 192×4M | 110 days | 61 hours | 43.4× |
| SPS | 430K | 288×6M | 8.8 days | 3.7 hours | 56.4× |
| PS | 380K | 21×32M | 2.3 days | 1 hour | 56.7× |

well as communication and synchronization overheads that dominate the execution as more nodes are added. Nevertheless, the RDS version maintains around 60-80% normalized throughput efficiency, even in the largest, 32-node configuration. The RDS technique decouples the MPI workers, by assuming that their locally calculated beam profile is a representative sample of the global beam profile. Therefore, with RDS, the communication and synchronization overheads are minimized, which explains why the throughput per node remains solid even in large node configurations.

### 5.7.1 Exploring New Parameter Spaces

In this section, we touch the importance of the speedup recorded with *HBLonD* as seen by the user base experience – a team of experienced accelerator physicists working at CERN, one of world's largest research facility.

The execution time of a test-case is mainly determined by the number of simulated macro-particles and synchrotron revolutions (simulation turns) required. These parameters vary according to the phenomena under study and the physical dimensions of the accelerator. Usually, for a complete study, thousands of simulations are run in order to explore the parameter space and discover solutions that optimally satisfy the requirements. The exploration of this parameter space is an iterative process, and thus run times beyond a few days to a week are impractical. Users often have to simplify their models to reduce the run time requirements. Furthermore, simulations lasting longer than a week have a higher probability of failure due to the imperfect cluster reliability, the scheduler's quota limitations, scheduled maintenance and other unplanned outages.

*HBLonD* alleviates the aforementioned limitations and enables simulation of a wide range of new studies, becoming and invaluable tool for accelerator physicists. Table 5.2 contains the basic input configuration for the three target test-cases, as well as the estimated execution time with *BLonD++* [58] and *HBLonD* on 32 nodes, respectively. With *BLonD++*, running the LHC case would be practically impossible. For the SPS, simulating the real amount

of bunches was impractical and therefore omitted. The speedup brought by *HBLonD* enables users to simulate the real operational scenario and discover some multi-bunched effects (like beam instabilities) that would be impossible to observe with a reduced number of bunches. Some of these results lead to hardware design modifications [42, 43] for the ongoing machine upgrades [28], which would otherwise not have been realised. With *HBLonD* the LHC and SPS cases require less than 3 days and 4 hours, respectively.

# Chapter 6

# GPU-Accelerated Beam Longitudinal Dynamics Studies

## 6.1 *CuBLonD*: Multi-GPU *BLonD* Experiments

Graphic Processing Units (GPUs) were originally designed for efficient image and video processing in computer graphics applications. Over time, their massive computing capacity as well as the emergence of intuitive programming models, such as CUDA [81] and OpenCL [82], lead to the widespread use of GPUs for general purpose applications [83, 16]. GPUs are throughput oriented machines, meaning that they are optimized for processing large data sets at the expense of increased latency in smaller-scale data sets.

*HBLonD* made possible the simulation of a very large number of macro-particles by harnessing the power of distributed computing, combined with efficient intra-node scaling. Despite this major breakthrough in longitudinal beam dynamics simulation studies, the design of future machines, such as the FCC [31], the upcoming LHC luminosity upgrade project [29], and the discovery of new phenomena in multi-bunch simulations, are constantly pushing for larger, longer and more complex scenarios. Furthermore, living in the post Moore's law [1] era, the use of accelerator hardware is mandatory to achieve high performance and energy efficiency. These are the main driving forces that lead us to the development of a GPU accelerated version of *BLonD*, named *CuBLonD*, that will combine the existing MPI infrastructure with CUDA [84] kernels, to reduce further the executing time of extremely large simulation cases. This chapter describes the implementation and experimental evaluation of *CuBLonD*.

Not every application is well-fitted for GPU acceleration. GPUs are tailored for applications with wide data-level parallelism. *BLonD* simulations can reach up to 1 billion of simulated macro-particles, providing enough work to the GPU hardware to sustain high levels of data-parallelism. Furthermore, frequent CPU-GPU communication is often a performance limiting factor for GPU accelerated applications. In *CuBLonD*, the largest structures, which are the energy and

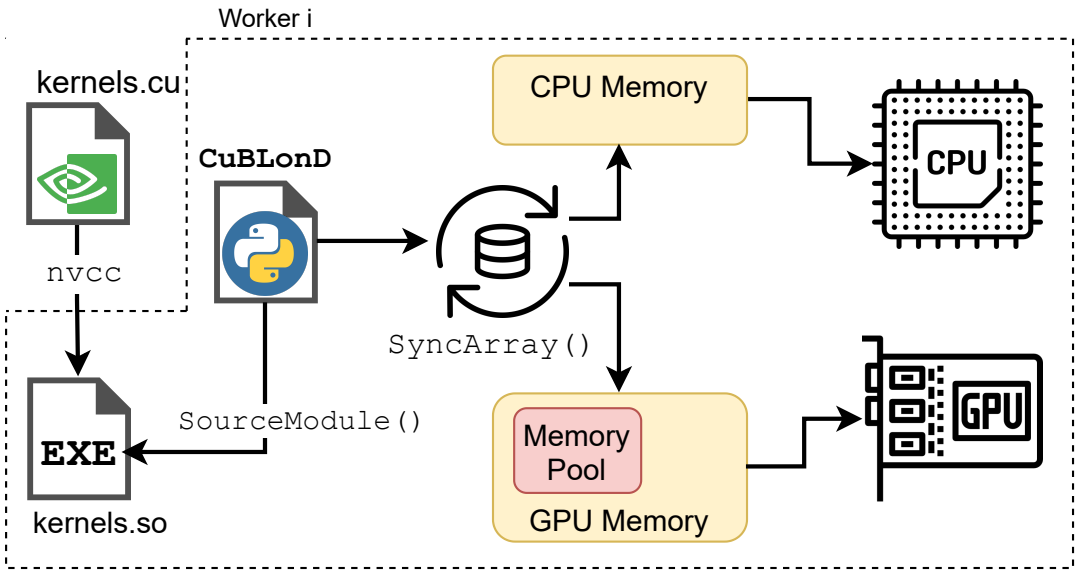



FIGURE 6.1: High-level software architecture of the GPU-accelerated *BLonD* code, *CuBLonD*.

time coordinate arrays, are transferred once from the CPU to the GPU memory, where they reside for the entire simulation. Only smaller arrays, such as the beam profile, are transferred more often between the host and the GPU. Finally, one of the challenges in a code like *BLonD*, is the variety of kernels required to model all the physics effects in typical simulation scenarios. Unless a significant portion of the code is accelerated by the GPU, the gains in execution time will be severely reduced. Therefore, *CuBLonD* accelerates almost the entirety of operations that take place during the main computational loop (see Fig. 5.2), to limit the potential communication overhead that would be otherwise required. Overall, more than one hundred functions had to be converted to GPU kernels for the efficient acceleration of the *BLonD* suite.

## 6.2 Seamless CUDA Integration

The CUDA enabled variant of *HBLonD*, *CuBLonD* was designed with performance as well as ease-of-use in mind. The PyCUDA [13] and Scikit-CUDA libraries were used to simplify the GPU memory allocation and the integration of native CUDA code in Python. In Fig. 6.1, we can see the high-level architecture of *CuBLonD*. The CUDA kernels are merged in one source file (`kernels.cu`) which is then compiled using the Nvidia C compiler to a shared library (`kernels.so`). This compiled library is exposed to the Python front-end using the `SourceModule()` method of PyCUDA, which allows for direct calls to native CUDA code with seemingly zero performance overhead.

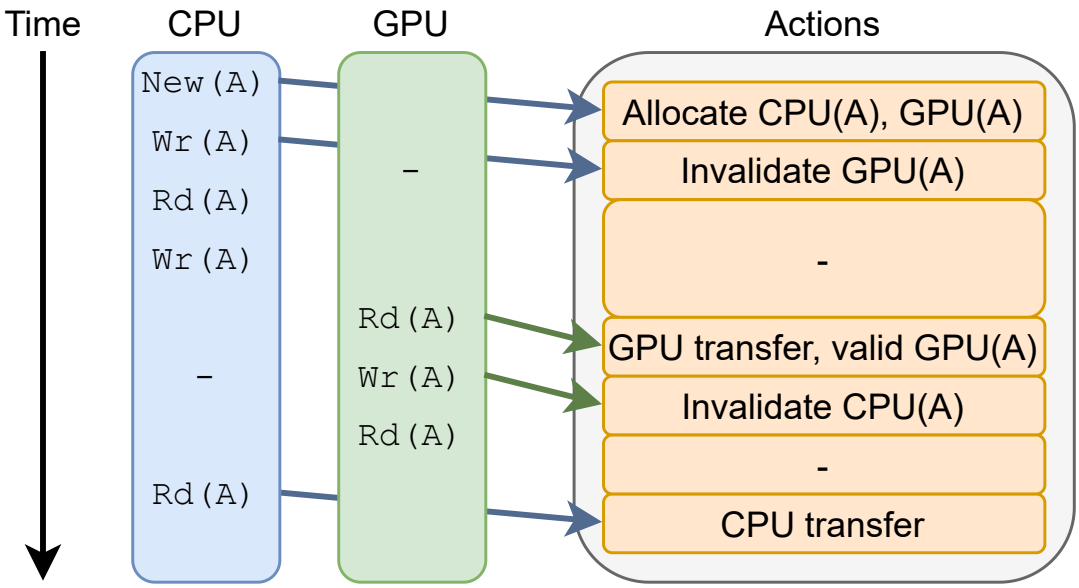



FIGURE 6.2: The `SyncArray()` class provides automatic synchronization between the view of array A in the CPU and GPU side.

To simplify the usability of the code, we developed the `SyncArray()` class. This class extends the Numpy [61] array interface, and basically provides to the user a single array reference that can be used as is by CPU and GPU code. The operations that take place under the hood when operating on a `SyncArray()` object can be seen in Fig. 6.2. The array needs to be allocated both in the CPU and GPU memory. When a modification to the CPU copy is made, the GPU copy is invalidated, and vise-versa. The invalidated copy is updated lazily, so that consecutive modifications will cause only a single data movement, when the invalidated copy will be first accessed. Since the greatest part of computing operations has been ported to the GPU, this process rarely needs to happen, especially inside the main computation loop. However, there are scenarios, where a user of *BLonD* would need to access data stored in GPU for reporting, plotting, storing in files. In this case, the `SyncArray()` mechanism hides the underlying complexity from the end-user, while ensuring the correctness of the simulation.

Another mechanism we developed is the GPU memory pool. There are certain operations in a *BLonD* simulation, such as the evaluation of multiple FFTs, that require the allocation of temporary memory regions. The size of these memory regions often remains constant for long simulation periods. To avoid frequent allocation/de-allocation of these memory regions, we designed a software-managed memory pool, on top of the `SyncArray()` class. This means that the end-user does not need to be aware of the memory pooling mechanism.

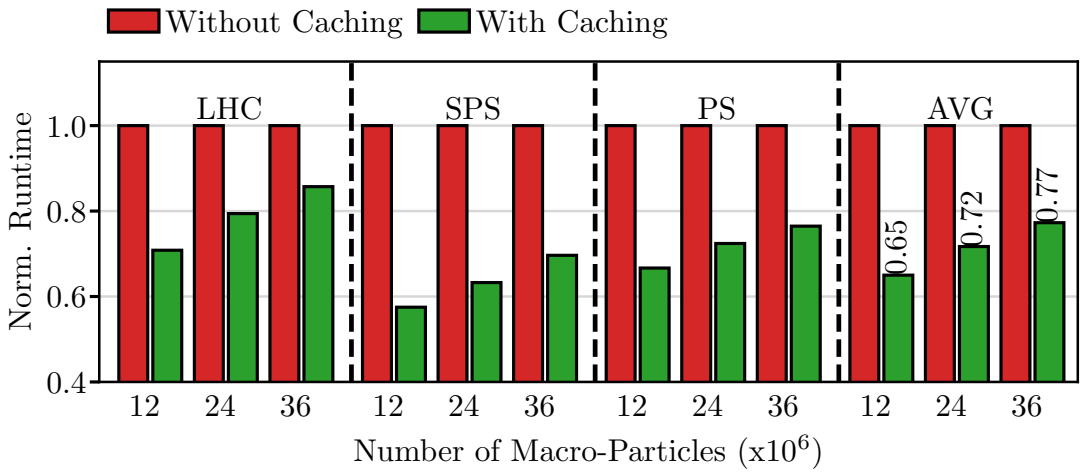



FIGURE 6.3: Effect of caching common arrays in GPU main memory.

The memory pool basically caches frequently used memory structures, and returns them when requested. Since the memory pool manages a fixed amount of memory, it uses a Least Recently Used policy to replace old memory allocations with recent ones. In Fig. 6.3, we can see the effect on the run time of the GPU pooling mechanism with varying input sizes. The run time is normalized to that of *CuBLonD* without pooling, therefore lower values correspond to greater performance gains. We observe that the performance gain decreases with larger input sizes. This happens since the memory allocation/de-allocation time is amortized across lengthier calculations, due to the higher number of particles. Furthermore, we see that the SPS testcase profits the most from memory pooling. The SPS testcase runs the most FFTs among the three testcases, and subsequently has the most allocation/de-allocation requests among the three testcases. On average, we see a 23% to 35% performance gain from the memory pooling mechanism.

With the use of CUDA libraries like CuRand and CuFFT, porting most of the computationally heavy code regions from C++ to CUDA was a straightforward process. Two benchmarks in particular, the `histogram()` that generates the beam profile and the `linear_interpolation_kick()` required adjustments to make proper use of the shared memory. More specifically, to avoid costly atomic operations on global memory structures, the histogram kernel first allocates a thread-block private beam profile. Then the thread-block private profiles are accumulated to generate the global beam profile (or worker-wide beam profile). However, since the shared memory is a limited resource, in large simulations only a portion of the beam profile fits in the shared memory. In this case, we take advantage of the Gaussian-like shape of the beam, to store in

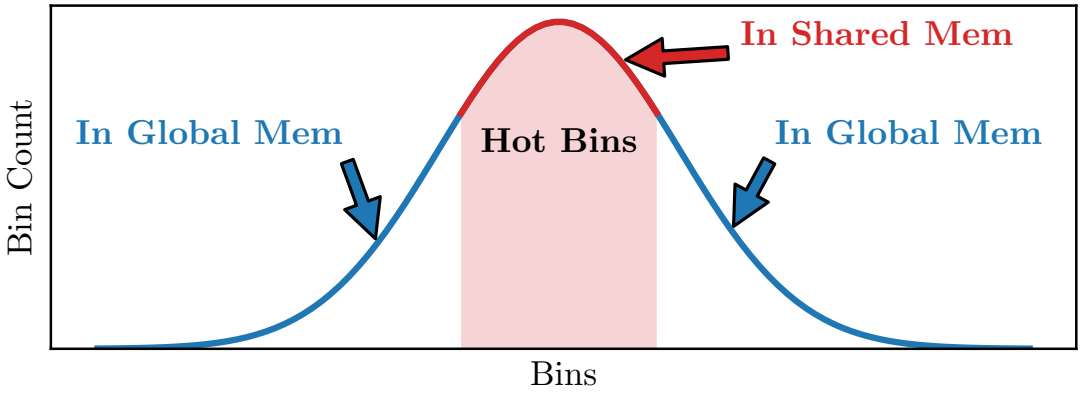



FIGURE 6.4: Using the thread block shared memory to cache the most frequently accessed histogram bins.

memory only the "hottest" bins, those around the center of the Gaussian distribution, as can be seen in Fig. 6.4. The remaining bins reside in global memory and are updated using atomic operations. Since the bins around the center of the distribution are accessed more frequently than the others, the average memory access latency approaches the latency of the fast shared memory.

The performance gain of this hybrid implementation is shown in Fig. 6.5. The y-axis shows the run time normalized to the global memory only implementation, therefore lower values correspond to greater run time gains. The x-axis shows different numbers of input particles. We observe that the performance gain gradually increases from 27% with 1 million particles, to 51% with 64 million particles. Then, it reaches a saturation point, where the portion of the beam profile that fits in the shared memory is too small to capture the hottest bins, therefore the performance gain is reduced to 29%. We need to note that 100 million macro-particles is the largest input size that we expect to use per GPU platform, since for very large simulations we use multiple GPU platforms and computing nodes. Nevertheless, the latency gain of utilizing the shared memory both in the `histogram()` and `linear_interpolation_kick()` operations ranges from 29% to 51%.

## 6.3 *CuBLonD* Single-Node Performance Evaluation

This section evaluates the single-node performance of *CuBLonD* in comparison with *HBLonD*. Later, in Sec. 6.5, the scalability of the code is examined. The specifications of the CPU server used for *HBLonD* are shown in the last column of Table 6.1, while the specifications of the two GPU platforms evaluated are shown on the second and third column of Table 6.1.

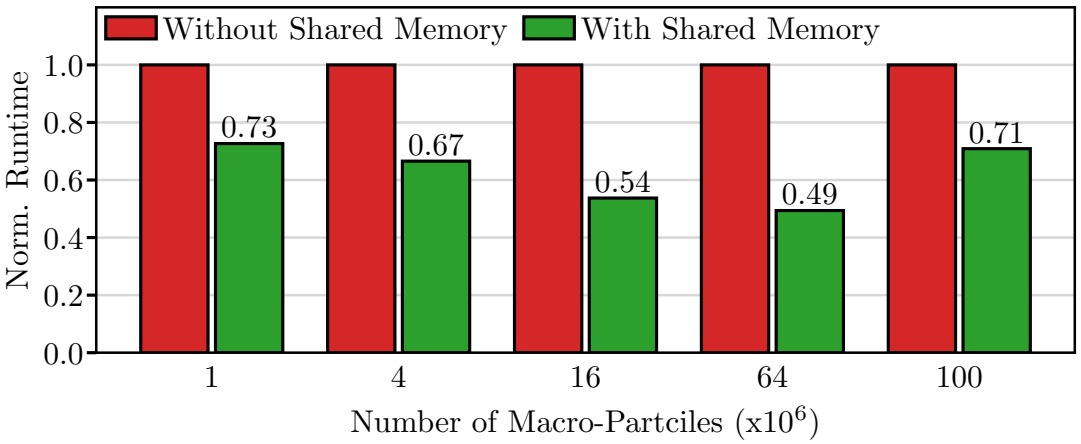



FIGURE 6.5: Latency gain by making use of the thread block shared memory in the `histogram` operation.

TABLE 6.1: Benchmarking Platforms' Specifications.

| Model | Nvidia Tesla K40 | Nvidia Tesla V100 | Intel Xeon E5-2630v4 |
|---|---|---|---|
| Generation | Kepler (2013) | Volta (2018) | Broadwell (2016) |
| Process size | 28 nm | 12 nm | 14 nm |
| RAM | 12 GB GDDR5 | 32 GB HBM2 | 2x64 GB DDR4 |
| Bandwidth | 288 GB/s | 897 GB/s | 2x68 GB/s |
| Cores | 15 | 80 | 2x10 |
| Frequency | 0.75 GHz | 1.2 GHz | 2.2 GHz |
| Cache | 1.5 MB (L2) | 6 MB (L2) | 2x25 MB (L3) |

The y-axis of Fig. 6.6, shows the throughput of *CuBLonD* using one GPU platform (*CuBLonD-1PN*) or two GPU platforms (*CuBLonD-2PN*), normalized to the throughput of *HBLonD* on a single 20-core node. Higher values correspond to greater speedups. The GPU that was used is the Nvidia K40 and the CPU is the Intel Xeon E5v4 (see Table 6.1). *CuBLonD* with one K40 provides approximately five times higher throughput than the CPU-only *HBLonD*. We observe varying speedup values for the three testcases, since each testcase is a unique application with specific characteristics. By utilizing both GPU platforms, the average speedup is 9.3x compared to the CPU baseline, or 1.82x higher than when using one platform. The speedup is not doubled when going from one to two GPU platforms, mainly because the code is not entirely GPU-accelerated and parallelized. In fact, on average 96.2% of the code runs on the GPU and is also parallelized with MPI. In addition, using two GPU platforms instead of one adds a communication and synchronization overhead to the total execution time.

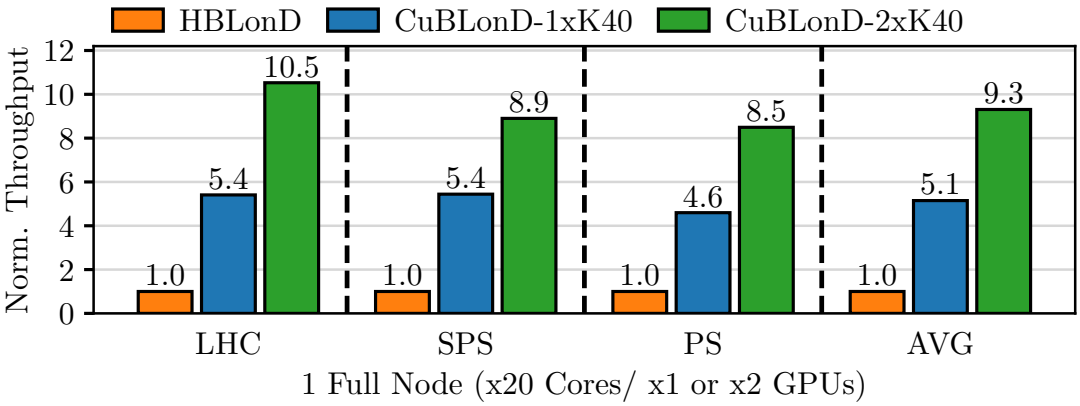



FIGURE 6.6: *CuBLonD* - *HBLonD* single-node throughput comparison with one and two K40 GPUs, in one node.

## 6.4 Future-proof High Performance Simulator

Scalability stressing experiments were performed on a system that contains Nvidia K40 GPUs. When the K40 was released in the end of 2013, it was a high-end, server class GPU. Nowadays it is considered outdated and has been superseded by newer generations. In this section, we evaluate the performance of *CuBLonD* in a more recent GPU generation – the Nvidia Tesla V100. The model's specifications can be found in Table 6.1. In Fig. 6.7, the y-axis shows the speedup achieved using one K40 or V100 GPU platform w.r.t. a 20-core CPU node. The bars correspond to the double precision, non-approximate version (*F64*), the single precision, non-approximate version (*F32*), and the single-precision SRP approximate (*F32-SRP*) version. On average, the V100 GPU is 3x faster compared to the K40 GPU, and provides 23x to 46x faster execution compared to a 20-core CPU node, depending on the approximation method applied. These results demonstrate the potential of *CuBLonD* to efficiently take advantage of both older and more recent GPU platforms and provide great speedups w.r.t. the previous state-of-art CPU-only implementation.

## 6.5 Stressing *CuBLonD* Multi-Node Scalability

This section evaluates the scalability of *CuBLonD* under strong-scaling workload scenarios. In strong-scaling experiments, the input size is held constant while the number of computing resources increases. The purpose of this study is to determine whether the code can effectively scale in multiple computing nodes and discover the saturation point, i.e. the point after which the addition

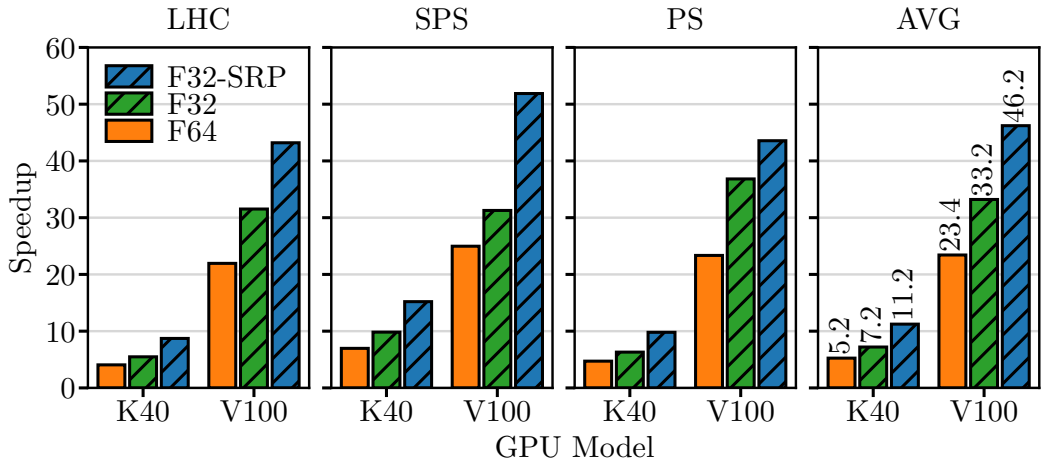



FIGURE 6.7: Single-node performance comparison of the Nvidia Tesla K40 and Nvidia Tesla V100 platforms.

of more resources is not increasing the performance. As formulated by Amdahl's law [9], this happens when a portion of the workload is not parallelized.

Each of the three columns of Fig. 6.8 corresponds to one of the three real-world testcases described in Sec. 5.1.1. The top row shows the performance of *CuBLonD*, when using one GPU card per node and up to 16 nodes or 16 GPU cards, while the bottom row shows the scalability of *CuBLonD* with two GPU cards per node and up to 16 nodes or 32 GPU cards. Each sub-figure of Fig. 6.8 demonstrates the scalability of the baseline non-approximate version, the 32-bit RDS approximation and the 32-bit SRP approximation. These approximate computing techniques have been described in detail in Sec. 5.4.

The y-axis of all sub-figures shows the speedup w.r.t. a 20-core instance of *BLonD++* [58]. We observe that without enabling any of the approximate computing variations, the scalability of the code saturates around eight computing nodes, due to the excessive communication time. Similarly to what was observed in *HBLonD*, the approximate computing techniques allow for far greater performance scalability compared to the non-approximate baseline. In the top row of Fig. 6.8, the performance gain the SRP and RDS techniques is comparable, with SRP being slightly better, demonstrating speedups of 50x, 78x and 72x in the LHC, SPS and PS testcase, respectively. Finally, in the bottom row we can see the performance of *CuBLonD* when using both GPU devices available in every computing node. The largest configuration uses 32 GPUs in 16 nodes. In this case, the speedup compared to a 20-core reaches or surpasses two-orders of magnitude in the SPS and PS testcase, i.e. 122x and 108x, respectively. In the LHC case, the larger and more time consuming FFT operations during the induced voltage calculation, that are not parallelized across the MPI workers

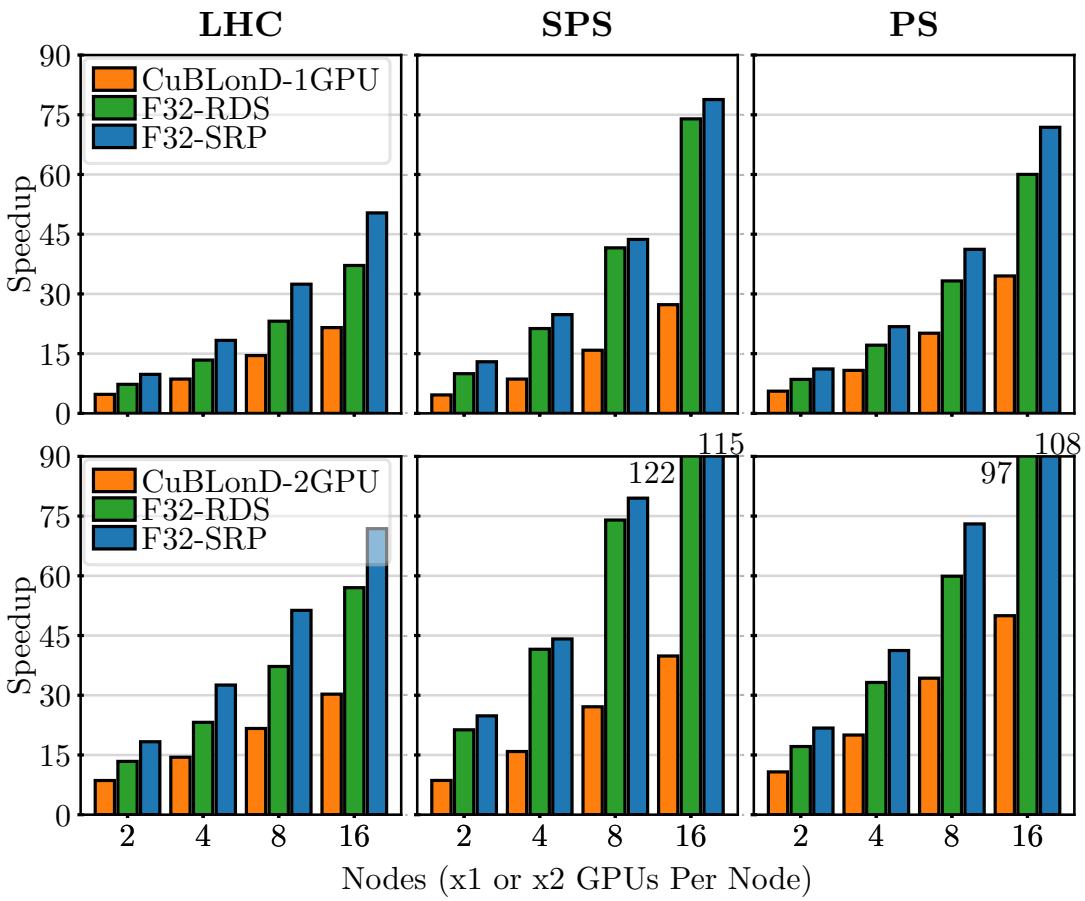



FIGURE 6.8: Multi-node scalability of Cublond.

but only within every worker, the achieved speedup is slightly lower, at 72x. The massive computational capacity of *CuBLonD* will play a key role in the upcoming, large-scale *BLonD* studies that include:

- End-to-end simulations, i.e. beam dynamics simulations that start from the injection of the charged particles to the smallest synchrotron (Proton Synchrotron Booster), the gradual energy ramp-up through a sequence of increasingly larger machines, until the injection to the LHC and the moment before the first collisions.

- High-precision, future machine studies that integrate multi-bunch collective effects only observable with high number of beam bunches and macro-particles.

To better understand the factors that limit the code's scalability, especially when using the approximate-free version, we examine the run-time breakdown of *CuBLonD* in three categories: communication and synchronization time (*comm*), time spent in regions not globally parallelized with MPI, but only locally with CUDA (*serial*), and the remaining fully parallelized computation

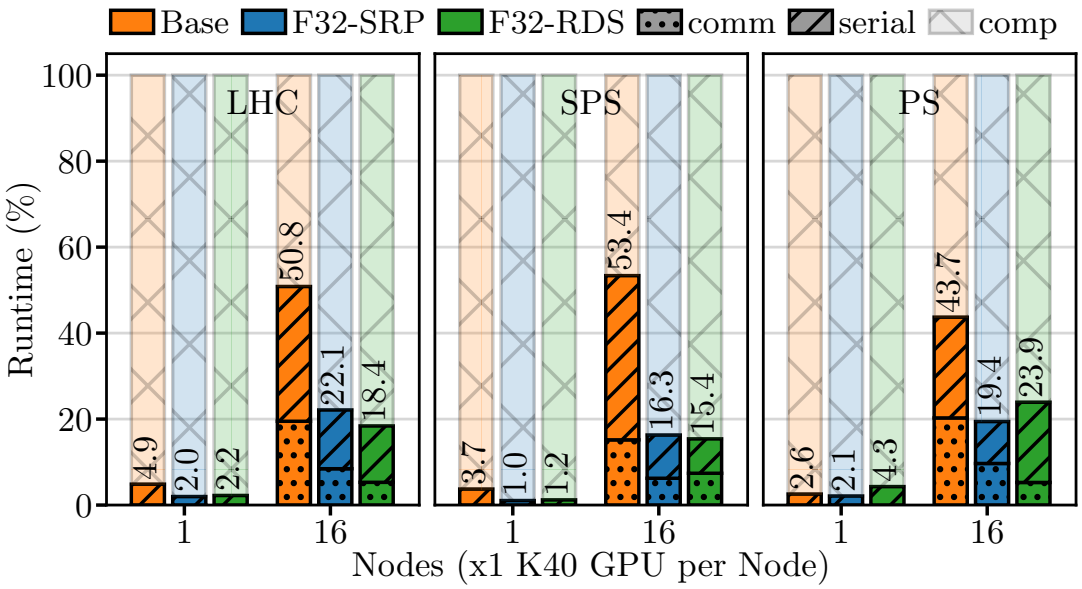



FIGURE 6.9: *CuBLonD* execution time breakdown. As the node count increases, less time is spent on fully parallelized computations, limiting the scalability.

time (*comp*). Ideally, the MPI workers should use their time mostly on the fully parallelized code regions, and only spend a minimum percentage of their time for communication and other non-globally parallelized calculations.

Figure 6.9 shows time breakdown of *CuBLonD* for the non-approximate *Base*, and the *F32-SRP* and *F32-RDS* approximations. In the single-node configuration, no time is spent on communicating, since there is only a single worker, and less than 5% of the execution time of each testcase is spent for serial processing. When the number of computing nodes increases to 16, approximately 15% to 20% of the *Base* version run time is used for communication and synchronization among the remote MPI workers, while the percentage of the run time spent for serial processing ranges from 23% to 38%. The most time consuming, non-globally parallelized computation is spent on the FFTs that take place during the induced voltage calculation. These are one-dimensional FFTs that do not scale efficiently on a distributed environment due to the nature of the algorithm. In all three testcases, without applying approximate computing, about half of the total run time is spent running useful parallel operation, and the other half is used for communication and non-scalable operations. This is limiting the scalability that can be exploited as the node count increases, as shown previously in Fig. 6.8. The RDS approximation relaxes the synchronization required among workers, therefore reducing the communication time to 5-7%. The serial processing time is also reduced greatly with the approximate versions, ranging from 10-18%. Since not more than a quarter of the total run time is spent in non-parallelizable operation, we expect the performance

of the approximate variants to scale with a further increase in the number of computing nodes.

# Part II

# Non-Conventional General-Purpose GPU Architectures

# Chapter 7

# Motivational Observations

## 7.1 Traditional GPUs and Thread-Level Parallelism

Graphic Processing Units (GPUs) are nowadays the dominant platform for general-purpose workload acceleration. While originally designed for graphics processing in video games and other visual applications, over time, with the introduction of comprehensive programming interfaces [81, 85], GPU programming for general-purpose applications became more practical. Their processing power and intuitive programming model led to their adoption in a wide spectrum of computing domains [86, 87]. GPUs are throughput-oriented machines; they employ massive multi-threading and fast context switching to achieve high computational throughput [7]. A modern GPU holds up to 64 active thread contexts per core at a time[88, 89, 90]; much more than what is seen in modern CPUs with one to four hyper-threads per core. To allow for single-cycle context switch, every GPU thread context has its own set of registers. This results in large register files that have been the focal point of research efforts [91, 92, 93].

In modern GPUs [94, 95], a front-end issue scheduler is responsible for selecting one or more instructions to issue to the back-end pipeline every turn. Although GPU manufacturers do not disclose all micro-architecture related details [96] of their platforms, including the front-end issue scheduler, various implementations have been proposed like the Loose Round-Robin (LRR), the Greedy-Then-Oldest (GTO) [97] and the Two-Level-Scheduler (TLS) [98]. The fundamental idea is that when a thread context, also called *warp* or *wavefront*, encounters a stall, the scheduler will continue to issue instructions from the remaining warps. Given enough active warps, the back-end execution pipeline will remain active until the stalled operation will be serviced, and the stalled warp will be considered for issuing again. Consequently, for most GPU functions, also called *kernels*, and GPU software developers, maintaining a large number of active warps, or high *warp occupancy*, is inherent to extracting maximum

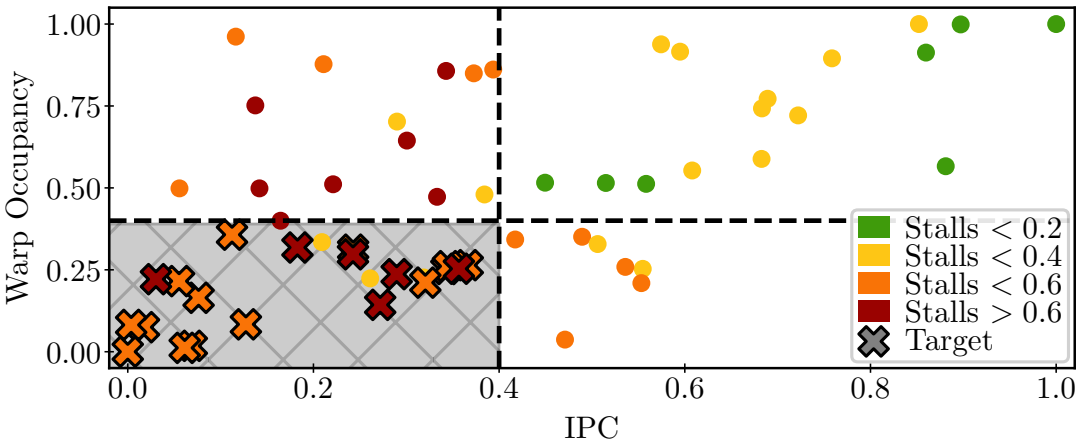



FIGURE 7.1: Warp occupancy, IPC and stalls correlation for 60 general-purpose GPU kernels.

performance off a GPU platform. Reportedly, cache-sensitive workloads are an exception to this rule, since they require a larger share of the cache and memory resources and thus they profit from a lower warp occupancy [97, 99, 100].

The rise of deep learning and big data analytics applications [101, 102, 83, 103], with extremely large input datasets, has motivated Multi-GPU system designs [104, 105, 106], to allow for scaling beyond the scope of a single GPU platform. At the same time, NVIDIA recently announced the Multi-Instance GPU (MIG) architecture [107], allowing up to seven clients to simultaneously share a GPU, in order to accommodate smaller scale applications effectively. As GPU acceleration has become more prevalent than ever, we see active development towards both directions of the workload scale spectrum.

## 7.2 Low-Occupancy, Underperforming Kernels

Among the increasing range of GPU-accelerated applications, we observe the existence of a class of kernels, which, due to limited data parallelism, fail to support a large degree of Thread-Level Parallelism (TLP) and hide the latency of memory operations. Thus they suffer from excessive stalling time and sub-optimal resource utilization. As shown in Fig. 7.1, from a collection of 60 general-purpose kernels originating from three well known benchmark suites [108, 109, 84], a noticeable percentage of kernels fail to maintain enough active warps and are prone to excessive stalling time. Fig. 7.1 depicts the distribution of warp occupancy, IPC, and stalled cycles[1], where both axes are normalized to the range of observed values. A cycle counts as stalled if no warp was issued in

[1]Our baseline GPU architecture throughout the thesis is modelled in GPGPU-Sim [110] after the NVIDIA Pascal [89, 88] GeForce GTX1080Ti model, detailed in Sec. 10.2

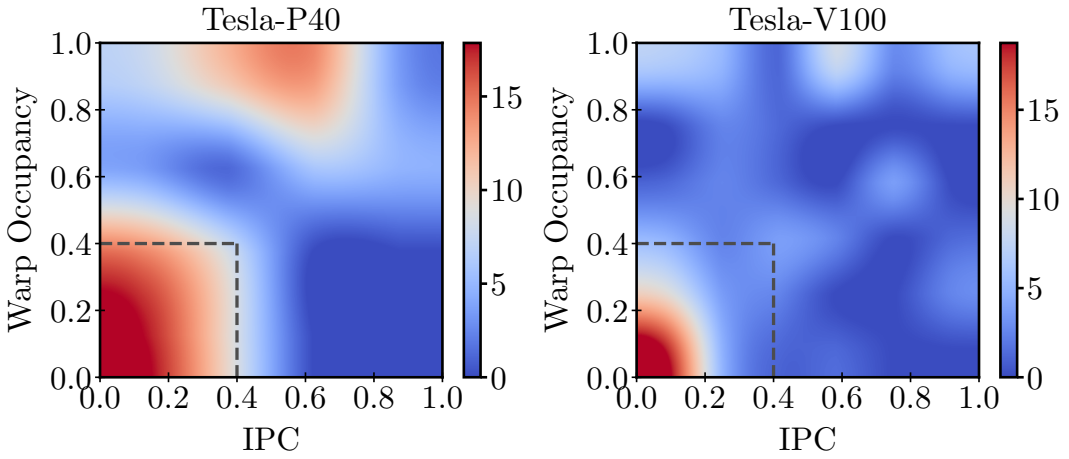



FIGURE 7.2: Warp occupancy - IPC distribution for 115 general-purpose kernels in two physical GPU platforms.

this cycle. There are three stall sources: control hazards, register dependencies, and no available execution units. In practice, dependencies on long-latency memory operations are the most common source of stalls. As shown in Fig. 7.1, a significant number of kernels, i.e. 20/60 (33.3%), demonstrate low (less than 40% of the maximum) warp occupancy, IPC deterioration, and stall more than 40% of their runtime. These "not-well-fitted" kernels (Table 10.1) cannot exploit effectively the traditional TLP model due to low warp occupancy. They seek for more aggressive Instruction-Level Parallelism (ILP) strategies to improve stall hiding, tolerate periods of insufficient TLP, and provide the back-end with a flow of instructions even in the absence of a large number of active thread contexts. In the remainder of this thesis we will refer to these low-TLP kernels as *target* kernels, to distinguish them from the rest *non-target* kernels.

The above observation forms an ill-pattern for a large set of GPU applications. This pattern is further confirmed when examining a wider set of applications (115 kernels in total) from [108, 109, 84] and their execution footprint onto two physical, not simulated, GPU platforms. Fig. 7.2 shows the warp occupancy - IPC distribution for the NVIDIA Tesla P40 [89] and NVIDIA Tesla V100 [90] GPUs, respectively. In the figure, both axes are normalized to the range of observed values. In both GPUs, it is evident that a large amount of kernels (i.e. 35.7% and 42.5% for Tesla P40 and Tesla V100, respectively) is concentrated in the lower left region, characterised by limited warp occupancy, hence limited TLP, and low IPC, hence sub-optimal execution efficiency. While several research efforts have already focused on resource utilization improvements [111, 112], they are either tailored to specific workload scenarios [97, 99] or rely on static information and lack flexibility [113, 114].

## 7.3 Instruction-Level Parallelism Exploitation

In the second part of this thesis, we address the aforementioned inefficiencies found in typical GPUs by re-purposing GPU micro-architectures towards a general-purpose, dynamic, Light-weight Out-of-Order GPU (LOOG) execution scheme, carefully designed to minimize the hardware overhead. LOOG surpasses prior state-of-art by exploiting ILP to complement the existing TLP and improve resource utilization of underperforming kernels. To limit the area and power overhead to a feasible extent, most resources are left intact and some are repurposed to the needs of Out-Of-Order execution. LOOG is a fully-functional, non-speculative Out-Of-Order (OOO) architecture, capable of reordering both arithmetic and memory operations, that can alleviate a wide variety of pathogenic scenarios that influence intra-core resource utilization. Being a dynamic system, LOOG boosts ILP when needed according to the dynamic instruction mix, that varies largely compared to the static instruction flow. We analyse LOOG's features, by exploring its sensitivity to various key micro-architecture components, demonstrating significant performance, area and power benefits from fine-tuning the LOOG architecture. In addition, a systematic two-level exploration of its key micro-architectural parameters is followed. At first, a sensitivity analysis on the sizing of the Instruction Window (I-Window) and the number of Operand Collectors is performed, identifying the configuration that provide good performance with minimal hardware overheads. Then, an extensive resource-aware design space exploration (DSE) for both typical GPU platforms and their LOOG-based projections is conducted, to study OOO GPUs' sensitivity and efficiency to scaled resource allocation scenarios. Historically, newer GPU generations contain more resources in terms of Execution Units (EXUs) and L2 Cache (L2C) per GPU core. Therefore, we vary these two resources, generating multiple baseline in-order GPU variants, and fine-tune LOOG for each of these variants.

LOOG is extensively evaluated and benchmarked against conventional in-order GPU architectures, as well as prior-art implementations of Out-Of-Order execution GPUs. To establish LOOG's potential as a realistic alternative GPU architecture, we evaluate LOOG's efficiency over a generic collection of 60 kernels in comparison with state-of-art GPU configurations. Finally, we provide an in-depth analysis of LOOG's performance and power efficiency, with respect to compiler-based optimizations, up-sized workloads, and modern micro-architectural configurations. We show that LOOG's OOO execution delivers average latency gains of 27.6%, and energy gains of 22.4%.

# Chapter 8

# Prior-Art & Background Knowledge

## 8.1 Prior-Art

In this section, we discuss LOOG's positioning and qualitative features with respect to prior-art. Several research efforts have pinpointed the inherent issues of resource utilization inefficiencies of modern GPU architectures, proposing either software or hardware approaches to effectively address them. Warped-P [113] is such a hardware-based extension that, in the shadow of long-latency stalls, continues to fetch, decode and execute independent instructions to keep the pipeline busy. The often underutilized [115, 116] RF is leveraged to store the speculatively pre-calculated results, from where they are restored when a stalled warp resumes normal execution. HAWS [114] follows a similar approach, but is based on compiler-generated hints to minimize the overall area overhead. Both HAWS and Warped-P cannot re-order store instructions since they cannot handle address dependencies. LOOG on the contrary, uses a load-store queue to resolve address dependencies and allow for load-store re-ordering. Furthermore, in Warped-P, loads do not modify the RF but only warm up the cache. There are no guarantees, however, that the prefetched data will not be evicted from the cache by the time the stalled warp will resume normal execution and will try to access them. LOOG fully executes loads in an OOO fashion, in the absence of address dependencies. While Warped-P and HAWS mitigate only a specific pathological scenario: keeping the pipeline busy in the shadow of long-latency global memory operations, LOOG introduces a more generic, dynamic scheme that can additionally mitigate other pathological scenarios such as stalls due to low-occupancy kernels, long-delay special function unit (SFU) operations, and excessive control and structural hazards. Thus, LOOG forms a more holistic OOO scheme and as such, it has greater performance improvement potential.

Twin-Kernel (TK) [117] follows a compiler-based approach that better distributes the memory requests in time by re-organising the static instruction

stream. When compiling a kernel, a set of different versions with slightly altered memory access pattern is generated. Then, during the kernel launch, the altered versions are combined into pairs to utilize the GPU resources more efficiently. However, TK is a static approach and cannot properly handle the dynamic instruction mix characteristics. In addition, it requires that two complementary versions can be formed from each kernel, which is not always the case. LOOG on the other hand makes no assumptions about the workload characteristics and exploits ILP based on the dynamic instruction stream, and the available memory and compute resources. Similarly to TK, HyperQ [118] enables multiple kernels to run concurrently on the GPU, by assigning leftover shaders to subsequent kernels, assuming that there are independent kernels waiting for resources. While it improves the overall GPU throughput, HyperQ does not allow for shader sharing and is unable to improve the latency of kernels that suffer from frequent stalling. Nevertheless, the LOOG runtime is compatible with HyperQ.

A plethora of alternative warp scheduling techniques have been proposed to improve GPU resource utilization and avoid stalling. Narasiman et al. [98] propose a combined two-level scheduler (TLS) with a large warp micro-architecture (LWM). TLS aims to better distribute in time the long-latency stall operations and improve utilization, while LWM forms dynamically SIMD-width sub-warps to better utilize the EXUs in the presence of branch divergence. Mascar [119] implements a memory aware, warp prioritization scheduler, intertwined with a cache access re-execution mechanism to better overlap computation with memory accesses. RLWS [120] proposes a reinforcement learning based warp scheduler that can learn and adapt to various sorts of workloads, demonstrating performance gains compared to prior static scheduling methods [97, 98]. All these schedulers are better tailored for specific workloads, i.e. memory or compute intensive, with or without inter-thread data locality. LOOG features a simple, in-order warp issue scheduler, avoiding the complexity of an intelligent scheduler. After issuing, instructions dispatch to the EXUs in an OOO manner, as their source operands become available.

A variety of prior studies observe that the interference of multiple warps sharing the L1D cache, the network and the DRAM bandwidth can lead to cache thrashing and performance deterioration. Thus, these studies suggest either limiting the maximum number of active warps at a time (TLP throttling) or finely controlling the warps' access priority to the shared resources. Rogers et al. proposed CCWS [97], a scoring system to identify warps that re-reference their data in the L1D cache, and gives these warps more exclusive cache access.

In Dyncta [99], a CTA scheduling mechanism based on the applications characteristics is proposed, that limits the number of CTAs assigned to a shader and alleviates the contention to shared resources. LCS [121] and OWL [100] both develop coordinated CTA and warp scheduling techniques to throttle the TLP and better utilize the memory resources. Interestingly the authors of PCAL [112] observed that although TLP throttling techniques can improve the L1 cache utilization, they can leave other resources underutilized. Thus they proposed a coupled warp scheduling and cache management mechanism which prioritizes a subset of warps to access the cache while allowing low-priority warps to use other GPU resources. Recently, Dublish et al. introduced Poise [122], which is a machine learning framework combined with a runtime inference engine to predict the warp scheduling decision that will lead to a balanced TLP and memory system performance. Contrary to TLP limiting [99, 97, 112], Virtual Threads have been proposed in [111] suggesting that increasing the maximum number of concurrent warps beyond the scheduling limit can favor some applications. In conclusion, TLP throttling approaches can reduce cache congestion but can cause under-utilization of other GPU resources. In LOOG, we take a different approach to improve resource utilization by exploiting hidden ILP at the micro-architecture level and thus improving performance of both regular (data-parallel) or irregular workloads. As the ILP complements the existing TLP model, LOOG is orthogonal to TLP tuning techniques.

LOOG highly differentiates from prior-art by repurposing GPU microarchitecture through a light-weight Out-Of-Order execution scheme. Although LOOG's motivation is to assist kernels with low occupancy, in principle every workload can profit from OOO execution, compared to in-order only execution. Furthermore, LOOG exploits ILP among arithmetic and memory operations, therefore tackling a wide variety of sub-optimal resource utilization. Limiting the area overhead of an OOO scheme is crucial. We manage that by re-using the existing CUs as RSs and adopting a light-weight non-speculative pipeline. To the best of our knowledge, this is the first work that suggests, implements and evaluates a hardware-only, general-purpose, fully featured OOO execution scheme for GPU platforms.

## 8.2 GPU U-Arch Internals

A typical GP-GPU application begins executing on the host CPU side. Selected functions, called *kernels*, are offloaded to the GPU. A kernel is composed of hundreds or thousands of lightweight threads, called *work items* or simply threads, executing the same stream of instructions on different data, according

to the Single-Instruction Multiple-Thread (*SIMT*) model [81, 123]. Threads are organised into a grid of *Cooperative-Thread-Arrays* (CTA), consisting of *warps* or *wavefronts* that contain a group of 32 or 64 consecutive work items [85, 124]. Multiple CTAs are assigned to each GPU computing core, also called *Streaming Multiprocessor* (SM) or *shader* core. Each work item disposes a unique identifier (ID), that is typically used to identify its share of input data to process. Wavefronts execute in lockstep using a single Program Counter (PC) for all their work items. This is essential to reduce the fetch and decode resources with such a large number of thread contexts. When some of the work items of a wavefront follow different directions of a branch, they encounter a *branch* or *warp divergence*. The performance deteriorates, since both branch paths are executed sequentially by all work items, masking-off the unneeded calculations. A rich body of research work attacks the performance slowdown caused by branch divergence [125, 126, 98].

Wavefronts have access to a plethora of on- and off-chip memory structures [7, 123], namely the *local* memory, the *global* memory, the *shared* memory, and the read-only *constant* and *texture* memories. In addition, each thread has exclusive access to a limited number of registers. Having a separate set of registers per thread allows for fast and efficient context switching. Due to the very large number of active contexts (warps), the capacity of the register file tends to be substantial and a lot of effort has been put into optimising it in terms of efficiency, area and power consumption [116, 93, 127, 92].

GPU exposes three synchronization mechanisms. Threads within the same warp execute in lockstep, thus they are always in-sync. Threads belonging to the same CTA can synchronise using a hardware supported barrier instruction. Finally, a special host-side function call can be invoked to ensure that all CTAs of a kernel have completed. The programmer can make no assumptions about the execution order across CTAs or even warps belonging to the same CTA.

The number of concurrent CTAs assigned to a shader core normalized to the maximum number of CTAs supported by the hardware, also called *warp occupancy*, is calculated by the following Eq. 8.1:

$$\text{occupancy} = \frac{\text{Min}\left(\frac{\text{MaxWarpsPerSM}}{\text{WarpsPerBlk}}, \frac{\text{MaxRegsPerSM}}{\text{RegsPerBlk}}, \frac{\text{MaxShMemPerSM}}{\text{ShMemPerBlk}}\right)}{\text{MaxWarpsPerSM}} \tag{8.1}$$

During the kernel launch, the programmer specifies the number of warps per block, as well the amount of shared memory that will be allocated to each block. The number of registers needed per block is calculated by the compiler and depends mostly on the kernel code. The maximum values for these three

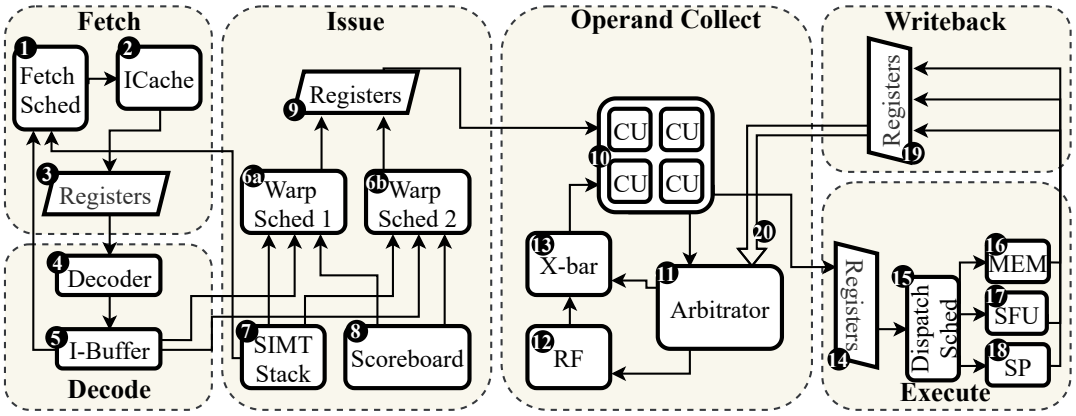



FIGURE 8.1: The baseline GPU architecture pipeline.

variables are architecture-dependent. Traditionally, high warp occupancy is desired, however there are specific cases where a higher number of active warps can hurt performance, mainly due to sub-optimal cache usage [97, 99, 112].

This work adopts the shader core pipeline of GPGPU-Sim [110], the state-of-the-art, open-source GPU simulator. The pipeline is composed of six stages: fetch, decode, issue, operand collect, execute and writeback. The main micro-architecture modules of each stage and their inter-connections are depicted in Fig. 8.1. The circled numbers in Fig. 8.1 indicate the instruction flow from fetch to writeback. A brief description of every stage follows.

*Fetch and Decode.* In the fetch stage, a round-robin scheduler selects a warp to issue a load operation from the I-Cache. A warp is considered for fetching only if it has no valid instructions in the I-Buffer, the structure that holds the decoded, ready-to-issue instructions of every warp. By default, two instructions are copied from the I-Cache to the fetch-decode pipeline register, except if the fetch address is situated at the end of the cache line, where only a single instruction is forwarded to the register. The decoder decodes the content of the fetch-decode register and stores it in the I-Buffer.

*Instruction Issue.* The warps are generally divided into multiple groups and each of the groups is assigned to an issue scheduler. Separate schedulers can use different scheduling strategies to avoid being too biased towards a specific workload scenario. Based on the scheduling policy, the scheduler selects one warp to issue up to two instructions per cycle. For a warp instruction to issue, it must not be blocked in a synchronization barrier, pass the scoreboard check and are find free issue-pipeline registers. Typically a GPU contains at least one memory unit (MEM), one arithmetic unit, also called scalar processor (SP), and one special function unit (SFU). After the instruction issues, the SIMT stack [128], and the scoreboard (SB) are updated.

The SB is a simple dependency-tracking mechanism, mainly used in in-order

processors [129]. In typical GPU micro-architectures, the scoreboard maintains a vector of write bits for every warp. The size of the vector is equal to the maximum number of registers that a warp can use. When instructions issue, they set the write bit of their destination register. The issue scheduler, before issuing an instruction to the operand collect stage, ensures that all the source and destination operands, have their write bits cleared in the SB. This way, RAW and WAW hazards are avoided. As we will see later in Sec. 10.1.4, WAR hazards are wrongly ignored in the base architecture. In GPU workloads, threads within a warp mostly execute the same flow of operations in lockstep. However, given that each thread operates on separate data, a warp can encounter a *branch* divergence, which deteriorates the performance [125, 126, 98]. The SIMT stack is a warp private structure used to handle branch divergence by serializing the execution of the divergent control flow paths [130]. Every SIMT stack entry holds a bit vector representing the active warp-items, called *active mask*, the PC address of the next instruction to execute and the reconvergence address. We consider the reconvergence address to be the immediate post-dominator (IPDOM) of the divergent branch, i.e. the earliest guaranteed point in the program that the instruction flow of all threads re-converges. Mechanisms that can, under circumstances, discover earlier re-convergence points than IPDOM have been proposed [126, 128, 130].

*Operand Collect (OC).* To allow for single-cycle context switching, every warp needs to have access to a separate set of registers. Due to the large number of active warps, the RF in modern GPUs can be as large as 256 KB [91]. Since the SRAM memory area scales with the number of read/write ports, a RF composed of multiple single-ported banks is less demanding in terms of area and energy resources, and is preferred in GPUs [131, 132]. After issuing, instructions allocate a CU and issue a read request for every source operand to the RF arbitrator. Instructions wait in the CUs until their source operands are read from the RF and are ready to dispatch to the execution units. Arbitration logic is used to handle RF bank conflicts and maximize the RF read throughput [133]. A crossbar (X-bar) is used to wire the CUs with the RF ports. There is a dedicated set of CUs per execution unit.

*Execute and Writeback.* As previously mentioned, there are multiple types of execution units. The SPs and SFUs are pipelined and SIMD-vectorized. Memory operations are forwarded to the MEM unit. The SFUs calculate transcendental instructions and the remaining arithmetic operations are calculated by the SPs. The dispatch scheduler selects, from the pool of ready to execute instructions, up to one instruction per EXU to dispatch, prioritizing older instructions. After executing, the instructions and the values of the destination

registers are moved to a set of execute-writeback registers. There, they issue a write request to the RF arbitrator and then retire from the pipeline. Multiple instructions can writeback per cycle.

# Chapter 9

# LOOG Internals

## 9.1 From GPU to LOOG uarch

This section describes the micro-architectural modifications required to enable out-of-order execution and exploit instruction-level parallelism. In addition to Fig. 8.1 that was used in Sec. 8.2 to show the baseline GPU pipeline components, Fig. 9.1 shows the LOOG-specific modifications and components introduced. While LOOG modifies several parts of the datapath, reviewed in the remainder of this section, it mainly extends the OC phase by i) advancing the CUs to implement functionality of reservation stations, and ii) introducing a Register Alias Table (RAT) structure that replaces the SB and eliminates false dependencies with register renaming.

**Wider fetch and decode bandwidth.** In LOOG, when a cache line is fetched from the I-Cache, 16 bytes (two instructions) to 128 bytes (16 instruction) are forwarded to the fetch-decode register. If the requested address is close to the end of the cache line, only the remaining bytes in the cache line will be fetched even if less than initially requested. Since warps can still fetch more data only after there are no more valid instructions in the I-Buffer, the I-Window is quantised into chunks. Although this may limit the ILP opportunities when approaching the chunk's end, the overall design is simplified. The I-Window size defines the register width, the decoder capacity per cycle and the width of every I-Buffer row. As seen in Sec. 10.2.3, the I-Window size greatly affects the area and power resources.

**Simplified issue stage.** LOOG's issue stage has been simplified compared to the baseline. Since the RAW dependencies are resolved in the CUs and the false dependencies are eliminated with register renaming, described in detail in the next paragraph, the SB has been removed. Secondly, traditional GPUs often use two or more issue schedulers, implementing a different scheduling policies. This intelligent scheduling logic is power hungry [116]. In LOOG, only a single round-robin scheduler is used, reducing the scheduling complexity, without performance loss. LOOG is insensitive to the warp issue policy. In

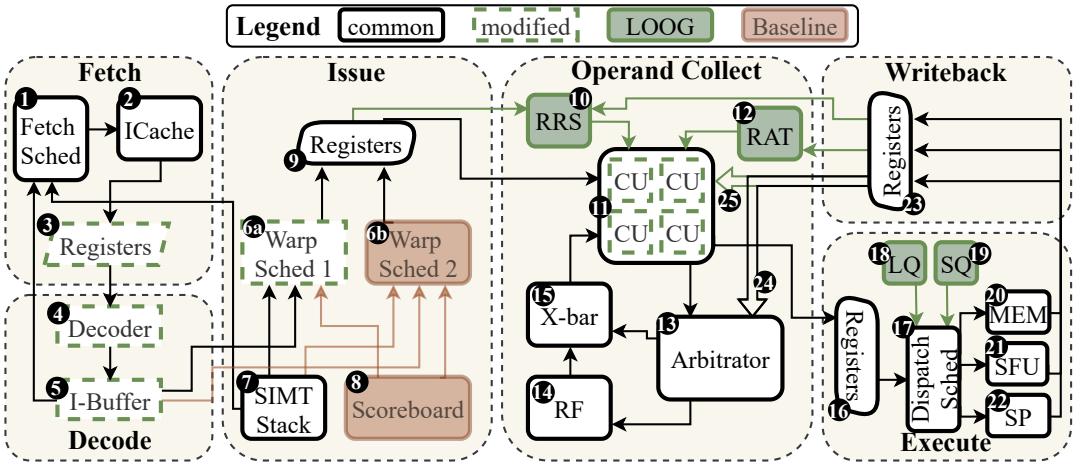



FIGURE 9.1: The LOOG modifications on top of the baseline architecture pipeline.

the baseline, the issue and dispatch stages are temporarily closely connected, since the OC stage only lasts a few cycles. Thus, the instruction issue sequence is directly linked with the instruction dispatch sequence. In LOOG, issue and dispatch are temporally far away since in the OC stage, instructions also resolve data dependencies, which takes more cycles. Consequently, the instruction issue and dispatch sequences diverge broadly, making LOOG insensitive to the warp scheduling policy.

**Register renaming.** In LOOG, the CUs function in addition as reservation stations. When an instruction allocates a CU, it first reads the RAT once for each source operand. The RAT is indexed by the operand ID and warp ID. Every RAT entry contains a CU ID field. A special null value is used to indicate that the latest value of the register is in the RF. Otherwise, the register has been renamed and the RAT entry contains the ID of the CU that holds the instruction that will produce the value of the register. This ID value is copied in the allocated CU and the result broadcast bus is monitored to match the CU ID and capture the value of the register. Also, the RAT is updated so that the instruction's destination register now points to the allocated CU ID. In Sec. 10.1.2, we introduce the RRS module and provide deeper insight into LOOG's register renaming scheme.

**Load-store re-ordering.** LOOG implements a non-speculative memory re-ordering scheme. We opted for a non-speculative scheme to avoid the necessary recovery mechanism and its associated cost. Memory operations might have address dependencies in addition to register dependencies. To track address dependencies, memory instructions allocate an entry in the Load Queue (LQ) and Store Queue (SQ). Two separate queues are used since loads and stores require different handling. More specifically, before re-ordering any memory operation,

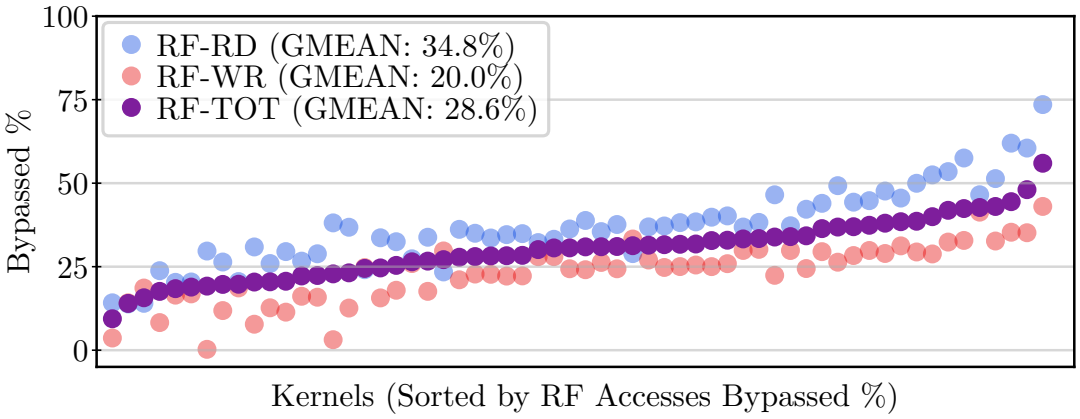



FIGURE 9.2: Percentage of Register File Read (RF-RD), Write (RF-WR) and Total (RF-TOT) accesses that are bypassed due to the addition of the result broadcast mechanism of LOOG.

the value of the address operand needs to be compared against the addresses of all earlier issued store instructions. If a match is detected, the memory operation cannot be reordered. In addition, if a store has not yet resolved its target address, it is effectively blocking all subsequent memory operations, due to the potential address conflict. Finally, to satisfy the GPU memory consistency model, no memory instructions are reordered ahead of memory barrier operations. While in an extreme scenario, every memory instruction can generate up to 32 memory address requests, in practice across our whole benchmark set, we saw a median of 1.67 requests per memory operation.

**Result broadcast.** Upon exiting the EXU, an instruction has calculated the value of the destination register and is ready to update the RF. The warp and destination register ID is used to index the RAT and compare its content with the CU ID of the just retired instruction. If it matches, a write request is sent to the arbitrator and the RAT entry valid bit is cleared, meaning that the latest value of the register can be found in the RF. Otherwise, the register has been renamed since the instruction was issued, thus neither the RF nor the RAT need to be updated. Notice how LOOG hides some RF writes compared to the baseline, alleviating the pressure to the RF banks. More specifically, as it can be seen in Fig. 9.2 across all 60 kernels comprising our evaluation set, in terms of geometric mean 34.8% of all reads bypass the RF, 20.0% of all writes bypass the RF, and in total the RF is accessed 28.6% less often. The result bus has been extended to carry the CU ID, and comparative logic has been added to the CUs to be able to capture the broadcasted result. Only after writeback the CU holding the retired instruction is freed.

TABLE 9.1: Area and Power overhead breakdown of LOOG with the same sizing as a traditional in-order GPU, modelled after the NVIDIA GeForceGTX1080Ti [88].

| **Unit** | **Capacity** | **ΔPower** | **ΔArea** |
|---|---|---|---|
| Decoder | 4 instr/cycle | 0.163 % | 0.033 % |
| I-Buffer | 4 entries/warp | 0.075 % | 0.009 % |
| Issue Scheduler | single, RR | 0.980 % | 0.101 % |
| RAT/ Scoreboard | 16-port RAT | 3.351 % | 0.185 % |
| RRS | 64 entries | 0.138 % | 0.039 % |
| Operand Collector | 32 CUs | 0.019 % | 0.008 % |
| Load/ Store Queue | 16 entries | 0.088 % | 0.087 % |
| Result Bus | – | 3.104 % | 0.042 % |
| **Core Total** | A: 30.58 $mm^2$, P: 5.37 W | **7.921** % | **0.506** % |
| **GPU Total (28 cores)** | A: 873.05 $mm^2$, P: 156.70 W | **7.582** % | **0.496** % |

## 9.2 LOOG Area and Power Modelling

We model LOOG u-arch functionality by introducing and implementing all the aforementioned mechanisms in GPGPU-Sim [110]. GPUWattch [134], a framework heavily based on McPAT [135] and CACTI [136], was used to model power and area for both LOOG and the baseline GPU architectures. The additional RRS, RAT and LSQ structures were modelled as SRAM arrays. Furthermore, since some GPU specific structures, i.e. the SIMT stack and the Scoreboard, were missing from the original version of GPUWattch, we implemented them. All the modifications affected the core pipeline; peripherals such as the Network-On-Chip, the memory controllers and the L2 memory were left intact. Table 9.1 lists all the modified components, their capacity as well as their overhead w.r.t. the baseline GPU platform, modelled after the NVIDIA GeForce GTX1080Ti [88]. Since the most resource-hungry structures like the RF, the EXUs, and the main memory were unaffected, the overall power overhead of LOOG is 7.58% and the area overhead is limited to 0.5% per platform.

The most heavily modified part of the core execution is the Operand Collect stage, where the register renaming operation takes place. According to the baseline model on which LOOG is based, the Operand Collect stage is pipelined in four sub-stages:

1. Allocate Collector Units: At first, the instructions that issued on the previous cycle look for and allocate free collector units.

2. Allocate Reads: For every source operand, the a read request will be forwarded to the Register File arbitrator.

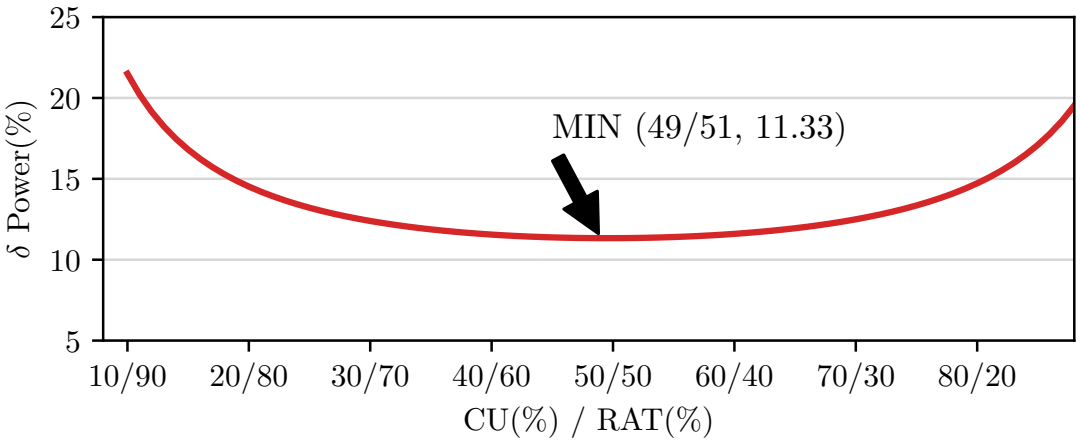



FIGURE 9.3: Adjusting the latency of the RAT and the CUs, so that both structures can be accessed within one cycle, and the associated power overhead.

3. Process banks: Then, the Register File banks are accessed to read the requested operand values. The arbitrator is used to maximize the throughput of the RF banks. A crossbar connects the Collector units with the Register File banks.

4. Finally, the Collector units that have read all source operands can dispatch to the execution units.

Consequently, four cycles is the minimum latency an instruction spends in the Operand Collect stage. In LOOG, the Register Alias Table (RAT) is also accessed when a Collector Unit is allocated, to find which source operand values should be read from the Register File, and which will have to be captured from the result bus.

Accessing the RAT can be modeled as an additional step of the Operand collect stage, or by using low latency SRAM components, so that both the RAT access and the CU allocation can take place in the same cycle. The RAT is more than two orders of magnitude smaller than the RF, since it holds for every register only a tag of length $\log_2 \#RRS$, i.e. 6 bits for a 64-entry RRS, while the RF holds the entire value of the register which is $32 \times 32 = 1024$ bits wide. As a result, the RAT overhead is minimal compared to the RF. Therefore, we preferred to decrease the latency of the RAT and the CU allocation so that both operations can take place in a single cycle. In turn, this increases the power consumption of these structures.

In Table 9.3, we vary the portion of the cycle allocated to accessing the RAT and allocating the CU in order to discover the most power efficient configuration. We observe that when allocating 49% of the cycle for accessing the RAT and 51% for accessing the CUs, the power overhead takes the minimum value which

TABLE 9.2: The cycles in which instructions complete executing the corresponding pipeline stages in the baseline and LOOG architecture, for a simple execution paradigm.

| Baseline Pipeline Stages Timing | | | | | | | |
|---|---|---|---|---|---|---|---|
| Instructions | | F | D | I | OC | EX | WB |
| I1 ld.global.u32 | r1,[r2] | 0 | 1 | 2 | 4 | 4+L1 | 5+L1 |
| I2 add.u32 | r1,r1,1 | 0 | 1 | 6+L1 | 8+L1 | 9+L1 | 10+L1 |
| I3 ld.global.u32 | r3,[r4] | 11+L1 | 12+L1 | 13+L1 | 15+L1 | 15+L1+L3 | 16+L1+L3 |
| I4 sub.u32 | r3,r3,1 | 11+L1 | 12+L1 | 17+L1+L3 | 19+L1+L3 | 20+L1+L3 | 21+L1+L3 |
| Total cycles: 21+L1+L3 | | L1 \| L3 | | H\|H: **23** | M\|H: **122** | H\|M: **122** | M\|M: **221** |
| LOOG Pipeline Stages Timing | | | | | | | |
| I1 ld.global.u32 | r1,[r2] | 0 | 1 | 2 | 4 | 4+L1 | 5+L1 |
| I2 add.u32 | r1,r1,1 | 0 | 1 | 2 | 5+L1 | 6+L1 | 7+L1 |
| I3 ld.global.u32 | r3,[r4] | 0 | 1 | 3 | 5 | 5+L3 | 6+L3 |
| I4 sub.u32 | r3,r3,1 | 0 | 1 | 3 | 6+L3 | 7+L3 | 8+L3 |
| Cycles: max(7+L1,8+L3) | | L1 \| L3 | | H\|H: **9** | M\|H: **107** | H\|M: **108** | M\|M: **108** |
| H: Hit, M: Miss, Lk: lk Latency | | **Delta(%)** | | **60.9** | **12.3** | **11.5** | **51.1** |

is 11.33%. In the remainder of this thesis, we assume this modeling of the Operand Collect stage. As mentioned above, an alternative approach could be to separate in two cycles the RAT and CU accesses.

## 9.3 LOOG Execution Paradigm

In order to provide some more insights on the performance improvements provided by LOOG compared to traditional in-order GPU architectures, we focus our discussion on a small code sample, i.e. few lines of assembly operations. We have made some assumptions through this example that allow us to focus on the instruction-level parallelism exploited by LOOG and demonstrate its potential within few lines of assembly operations. Let the sample program composed of four instructions (see Table 9.2): two global memory 32-bit load operations, each followed by an arithmetic operation that is data-dependent on the previous load. We assume single-cycle access latency on a L1 cache hit and 100-cycle latency on a miss. Furthermore, we suppose that we have enough resources to avoid structural hazards (e.g. no available CUs or pipeline registers) during this short exemplary execution. Finally, to amplify and focus the attention to the Out-Of-Order capability of LOOG, we consider only a single warp is active at the moment.

The rows of Table 9.2 contain the cycles the corresponding instruction leaves each of the six pipeline stages, for the baseline, at the top, and LOOG, at the bottom. In the baseline, two instructions are fetched and decoded per cycle. In LOOG, a four instruction wide I-Window is used, meaning also that up to four instructions can be fetched and decoded per cycle. Instructions I2 and I4 depend on I1 and I3, respectively, and cannot execute until after I1 and I3 have completed. Since there is no data dependency between I1 and I3, in LOOG, they can dispatch almost simultaneously, effectively exploiting instruction level parallelism and overlapping the two memory access requests. In the baseline, the two requests cannot be overlapped and need to run sequentially. Note that the writeback happens Out-Of-Order in LOOG. There is no need to enforce in-order writeback since LOOG is not executing speculatively, and precise exception and interrupt support is not typically a requirement for GPUs [137], as it is for CPUs.

The total number of cycles needed for this code snippet, according to the outcome of the memory accesses, in the baseline and LOOG is also shown in Table 9.2. In the all hit and all miss scenarios, LOOG requires less than half cycles in total compared to the baseline. In the mixed, miss/hit and hit/miss, scenarios LOOG requires $\approx$ 14% less cycles than the baseline. Thus, even in this small example, it is evident that in situations with reduced TLP, the ILP exploited by LOOG can lead to significant performance gains.

# Chapter 10

# Design Trade-Offs & Fine Tuning

## 10.1 Key Design Points & Trade-offs

### 10.1.1 Boosting the Instruction-Level Parallelism Potential with Memory Re-Ordering.

The initial version of LOOG was only capable of re-ordering Arithmetic and Logic Unit (ALU), and Special Function Unit (SFU) operations. When a memory operation, both load and store was detected, it would not be issued to the Operand Collect stage, and in addition it would block the associated warp from issuing any further instructions, since for functional correctness, the issue sequence must be kept in-program order. To achieve this, the main additional structures needed were the RAT, the modified CUs to serve as reservation stations, and the addition of a result broadcast bus. However, we soon noticed that not being able to re-order memory instructions was a major limiting factor for LOOG and its ability to exploit ILP.

To mitigate this issue, we enabled load and store re-ordering. To do that, load and store operations need to allocate an entry in the Load Queue and Store Queue respectively. Then, in-addition to potential register dependencies, that are resolved with the existing re-naming and result broadcast scheme, the address dependencies are also monitored. Since LOOG is a non-speculative OOO execution scheme, to avoid the implementation of a costly recovery mechanism, if a store instruction has not yet resolved its target address, all subsequent memory operations of all warps are blocked. This needs to happen, since if the target address is the same as this of another load operation, the store had to execute first. If a store has resolved its target address, this address is then compared against all earlier issued load and store operations. If no match is detected, these load and store operations can be re-ordered and execute ahead of the earlier issued store.

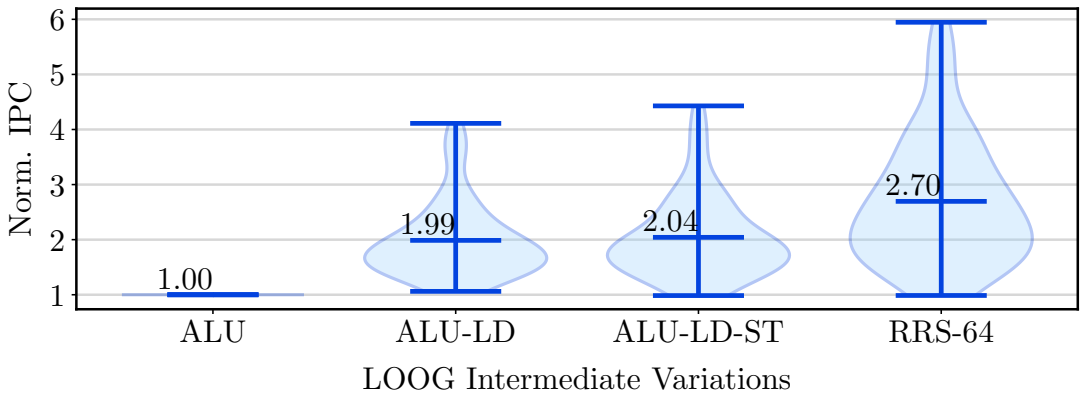



FIGURE 10.1: IPC comparison of LOOG's intermediate variations. All results are normalized to LOOG with only ALU and SFU instruction re-ordering.

Being able to re-order both compute, i.e. ALU and SFU, and memory operations, LOOG is a complete, non-speculative OOO scheme. Figure 10.1 shows the IPC comparison of LOOG's intermediate variations for the 60 general-purpose kernels in our benchmark set. All the results are normalized to the first LOOG variation, which the one capable of re-ordering only compute operations. The ability to re-order load operations provides 2× improvement in execution efficiency. Then, an additional 5% was gained by allowing the out-of-order execution of both load and store operations. Finally, as will be shown later in more detail, the inclusion of the Register Renaming Stack (RRS), enables earlier de-allocation of the collector units, which boosts performance by an additional 66%.

### 10.1.2 Operand Collect Stage Congestion

In traditional GPU architectures, the CUs are used to hold instructions until their source operands are read from the banked register file. Typically, this takes a few cycles. At best, a CU can read three source operands in a single cycle. Then, as soon the instruction dispatches to the execution stage, the CU is de-allocated and can accept a new instruction.

In LOOG, however, the CUs are additionally used as reservation stations, for instructions to resolve RAW dependencies. A long-latency memory operation dependency can take hundreds of cycles before it is resolved. Furthermore, the CUs cannot be freed when the instruction dispatches to the execution stage. For correct program execution, the CUs can only be freed after the writeback stage. To better illustrate this, let us assume that an original instruction I0 of warp W0 is issued to the CU0. The CU ID will be written in the instructions' destination register field in the RAT and subsequent dependent instructions will

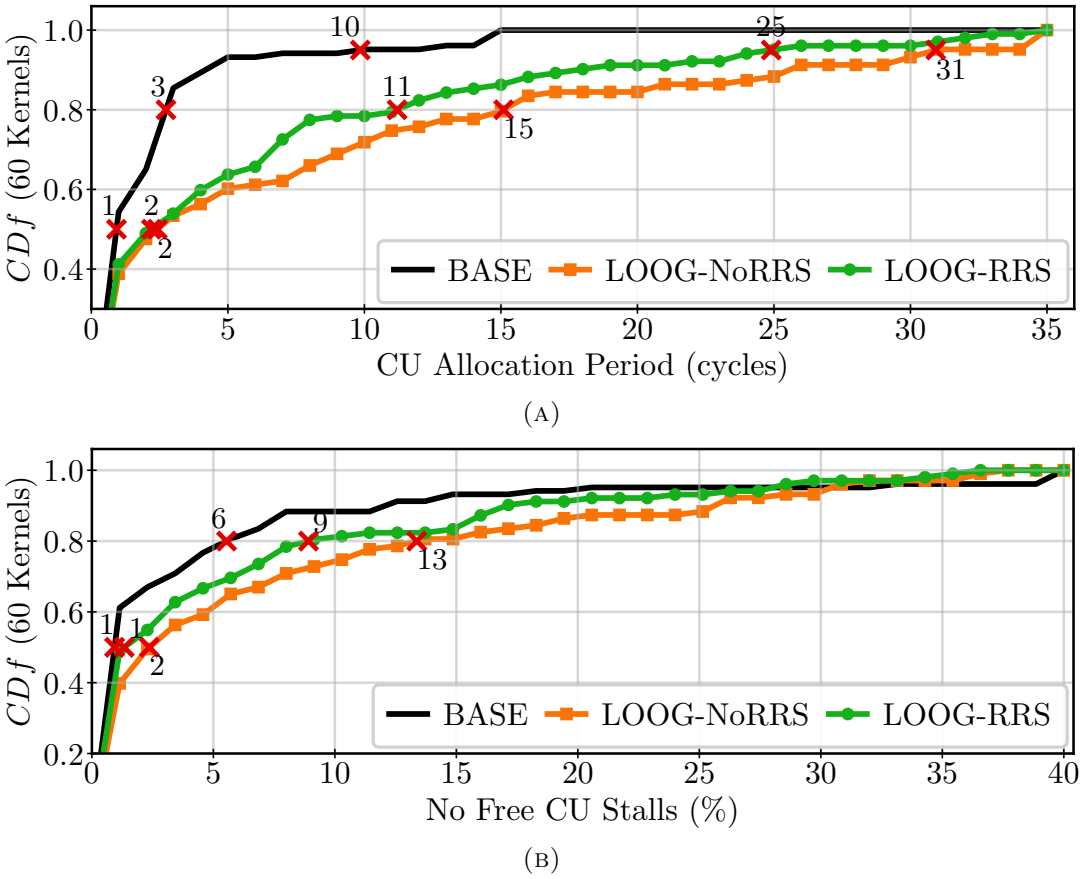



FIGURE 10.2: Cumulative Distribution Function of 60 GPU kernels of the CU allocation period (top), and pipeline stalls due to no available CU (bottom). The lines correspond to the baseline, LOOG without RRS, and LOOG with RRS architectures.

monitor the result bus to match this ID and capture the value of the broadcasted register. After a few cycles I0 dispatches and begins executing. If the CU0 was freed at this moment, a following instruction, say I1 of warp W0, could allocate it. If I1 would finish executing before I0, I1 would broadcast its result together with the CU ID 0 and instructions dependent to I0 would incorrectly capture the output of I1, breaching program correctness. Therefore, in LOOG the CUs are freed after the writeback stage; much later than in the baseline.

In Fig. 10.2a we can see the Cumulative Density Function (CDF) of the CU allocation interval in the baseline (*BASE*) and in LOOG (*LOOG-v0*). In the baseline, 50% of the instructions spend a single cycle in the CU, 80% of the instructions spend less than three cycles and 95% of the instructions less than ten cycles. In LOOG-v0, the allocation period is on average longer; two cycles for 50% of the instructions, less than 15 cycles for 80% and less than 31 cycles for 95% of operations. This longer allocation period translates directly to more stalls due to no available CUs. Indeed, the Pearson's correlation coefficient between the CU stalls and the CU allocation period is 91%.

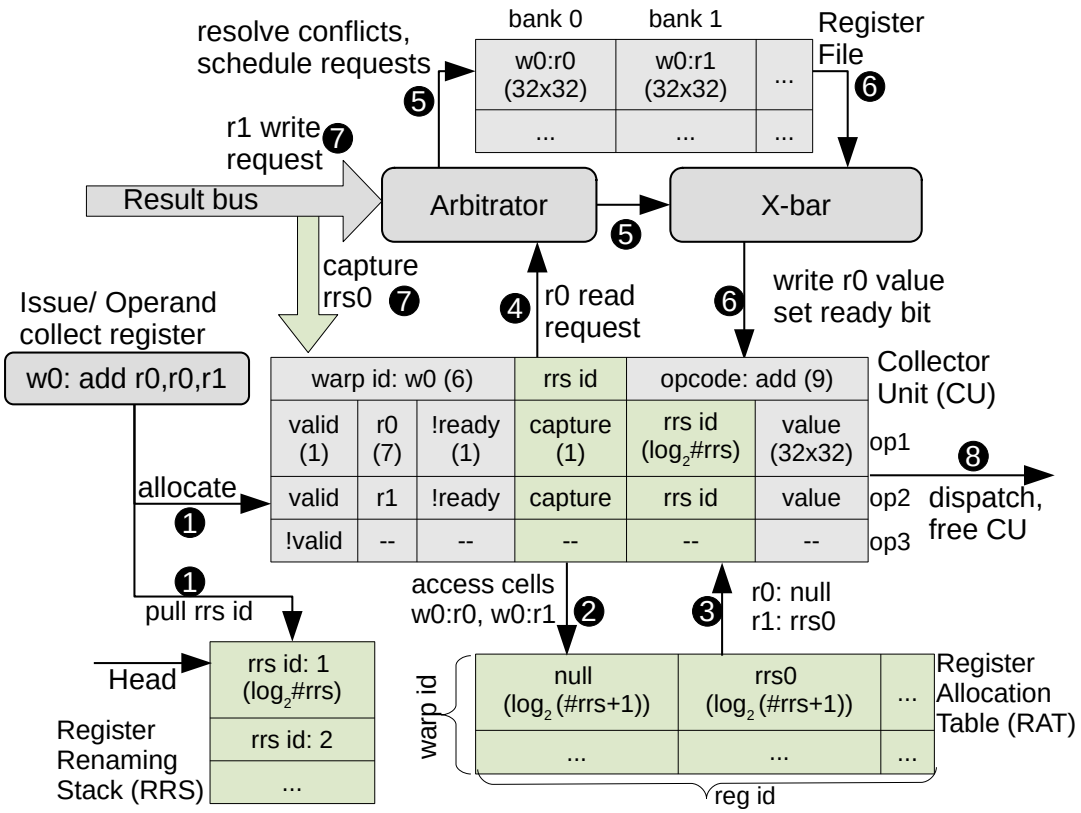



FIGURE 10.3: Register renaming in LOOG with RRS. The circled numbers indicate the sequence of steps involved in the operand collect stage.

A simple solution to this issue would be to increase the number of CUs. However, as we will see in detail in Sec. 10.2.2, CUs come with a significant area cost. To alleviate this problem in a cost-effective way, we introduced the Register Renaming Stack (RRS). The RRS is a structure that holds a list of unique IDs to be used in the RAT instead of the CU ID. When an instruction issues, it allocates a free CU and pulls a unique ID from the RRS. This ID is written in the RAT and subsequent dependent instructions will use it to capture the result from the result bus. This way, instructions can release the CU after dispatch, and hold only the RRS ID until writeback. The size of the RRS is $\#RRS \times \log_2(\#RRS)$, e.g. only 384 bits for a 64-entry RRS.

Leveraging the RRS, we allow the CUs to be freed earlier, decompressing the CU congestion. In Fig. 10.2a, we can see that the introduction of the RRS (*LOOG-RRS*) has decreased the average CU allocation period compared to LOOG without RRS (*LOOG-v0*), by approximately 20%, which resulted in a similar decrease in pipeline stalls due to no available CU, as it can be seen in Fig. 10.2b.

Fig. 10.3 depicts the sequence of operations that take place during the operand collect stage, in order for a typical instruction with two source operands

and one destination operand to become ready for dispatch to the EXUs. The numbers in parenthesis inside the OC structures correspond to the sizes of the data fields in bits. At first, there has to be a free CU and a free entry in the RRS(❶). The `rrsid` pointed by the head RRS pointer is copied to the `rrsid` field in the header of the CU. This unique ID is the new name of the destination register (`r0`). Then the RAT table, which is indexed by the warp ID and register ID, is accessed to read the values of the two source operands, `w0:r0` and `w0:r1`, and update the value of the destination operand `w0:r0`, to point to `rrs1` (❷). A special null value means that the latest value of this register is stored in the RF (`r0`) (❸). Otherwise, there is a RAW dependency, and the RAT value contains the ID of the renamed register (`rrs0`). The capture bit is set, and the result bus is monitored to match the tag field `rrs0` and capture the register value (❼). For `r0`, a read request is sent to the arbitrator (❹). The arbitrator, after resolving potential bank conflicts (❺), schedules the `w0:r0` read request. A copy of the value of the `r0` is stored in the firs operand row of the CU and the corresponding ready bit is set (❻). When the value of the renamed register `rrs0` is broadcasted, the CU captures it and sets the ready bit of the second operand (`r1`) (❼). The third source operand (`op3`) is not used by this instruction. Since both source operands have been read, the instruction is ready to dispatch and de-allocate the CU (❽).

### 10.1.3 To Predict or Not to Predict?

The branch predictor is a fundamental component of the front-end pipeline of OOO processors, crucial for maintaining a flow of instructions in the presence of conditional branches [138]. In alignment with traditional architectures, LOOG does not include a branch prediction mechanism.

Typical GPU workloads do not involve complex instruction flow, thus the performance degradation from control hazards is minimal. Indeed in Fig. 10.4, we can see the control stalls, normalized to the total execution cycles, for each of the 60 kernels in our test suite. On average 5.45% cycles are stalled due to control hazards and, therefore, the potential speedup of a branch prediction scheme is limited.

In addition, the size of a GPU branch predictor would be immense. In CPUs, a branch can be either Taken (T) or Not-Taken (NT). This binary information can be captured in a single bit. In GPUs, the threads of a warp execute in lockstep and might not evaluate uniformly a conditional branch. In fact, for an SIMT width of 32, there are $2^{32}$ possible branch outcomes, and 32 bits are needed to store this information. Using McPAT [135], we modelled a basic branch predictor structure adapted to the purposes of a GPU platform.

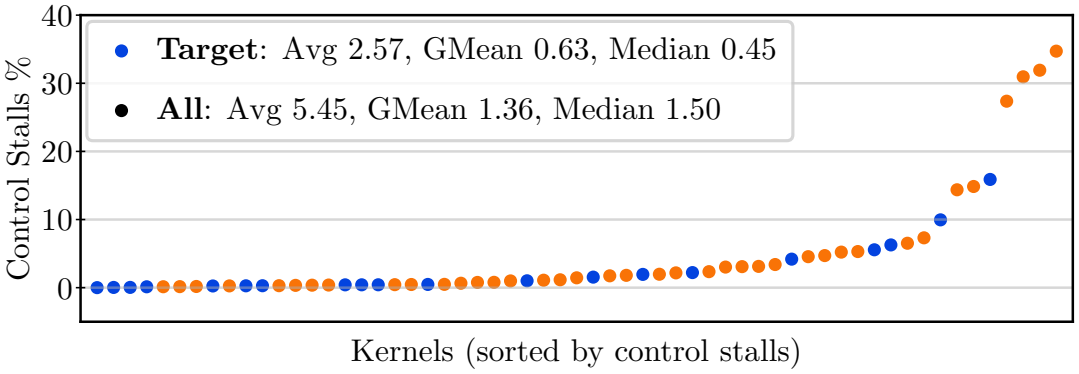



FIGURE 10.4: Percentage of cycles stalled due to control stalls. Kernels sorted according to control stalls percentage.

Specifically, we consider a branch predictor unit that includes branch target buffer (BTB) and can predict up to four targets per cycle, since our baseline platform can issue up to four instructions per cycle. Given that typical GPU kernels are shorter and contain less control flow compared to CPU functions, we scaled down the number of entries found in CPU predictors by a factor of four to eight. The measured area overhead ranges between 4.7% and 8.35%. Note that the structures needed to implement a recovery mechanism in case of mis-speculation are not included in this estimation, thus in reality, the measured overhead will be even greater. This makes such a predictor extremely demanding in terms of area and power.

### 10.1.4 Obeying Operand Dependencies

Obeying the RAW, WAR and WAW operand dependencies is essential for ensuring program correctness. In LOOG, the RAW dependencies are resolved in the CUs. Dependent instructions wait until the latest value of their read operands is broadcasted in the result bus, from where they can capture it. Both WAR and WAW hazards, which arise from name dependencies are eliminated with register renaming [139]. By renaming the destination registers of all instructions, using the RRS ID, we ensure that out-of-order writes will not affect instructions that depend on an earlier operand value.

In typical GPUs, a different approach is followed. A scoreboard is used to track and satisfy both RAW and WAW dependencies. When an instruction issues, it sets the dependency bit of the output register in the scoreboard and following instructions cannot issue if their source or destination registers have their dependency bit set in the scoreboard. In reality, GPU platforms complement the scoreboard with a mechanism to safeguard against WAR hazards [140]. However, a known default of GPGPU-Sim [110], the u-arch simulator used for

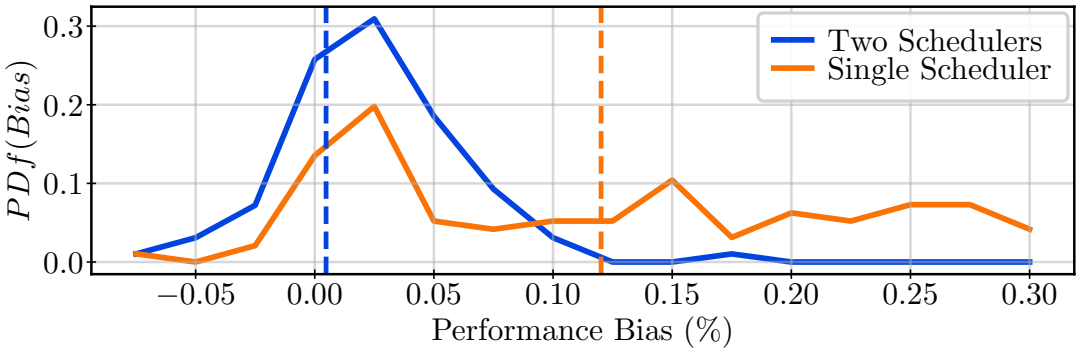



FIGURE 10.5: Performance bias towards the baseline due to neglecting WAR hazards.

modeling the baseline architecture and LOOG, is that it incorrectly ignores WAR hazards [140].

The following, uncommon but nevertheless possible scenario can lead to a WAR violation. The front-end scheduler selects every cycle a warp to issue up to two consecutive instructions given that they are no structural, control or data hazards. Let us assume that there is a WAR dependency between the two instructions, meaning that the second instruction is writing a register that the first instruction is reading. After issuing, both instructions allocate a CU. There, they schedule a read request for their input operands. The arbitrator resolves bank conflicts and selects which CUs will receive data from the RF. It is possible that while the first instruction is delayed due to bank conflicts, the second instruction dispatches, executes and writes back the destination register. Then, the first instruction will read the value of the register that the second instruction just modified, violating the program's correctness.

A straightforward way to resolve this pathogenic scenario and ensure correct functional execution is to only allow one instruction per warp to be in the back-end pipeline at a time, instead of two originally. Neglecting WAR hazards results in a performance bias, i.e. an overestimation of the real IPC, w.r.t. the more conservative but functionally correct approach described above. In Fig. 10.5, we measured for every kernel in our test suite, the performance bias of the baseline for not explicitly handling WAR hazards, with two different configurations: i) using a single or ii) two warp schedulers. Every scheduler can select one warp per cycle to issue, thus two schedulers have twice as much issue capacity as a single scheduler. The y-axis of Fig. 10.5 shows the Probability Density Function (PDF) of the performance bias (x-axis). A higher PDF value translates to a higher probability that a kernel will experience the corresponding performance bias. With a single scheduler, we see a considerable performance bias of 12%, on average. With two schedulers, the performance bias is limited to less than 1%. Thus, to minimize the IPC overestimation, we use two warp schedulers

Table 10.1: Collection of benchmark applications.

| | | | | |
|---|---|---|---|---|
| Streamcluster[108] | Histo-Par [109] | Hotspot [108] | Pathfinder[108] | SPMV [109] |
| ParticleFilter [108] | MergeSort [84] | SRAD-v1[108] | VecAdd [84] | MRI-Q [109] |
| RayTrace [110] | Reduction [84] | Gaussian [108] | Transpose [84] | AES [110] |
| Sorting Nets [84] | SRAD-v2 [108] | DWT2D [108] | MUMmer [110] | NN-Rod[108] |
| MonteCarlo [84] | Convolve [84] | KMeans [108] | Histo-SDK [84] | QRNG [84] |
| SumAbsDiff [109] | BackProp [108] | Sobol [84] | Binomial [84] | BFS [108] |
| FastWalsh [84] | Laplace3D[110] | NN-ISP [110] | ScalarProd [84] | |

Table 10.2: Baseline GPU platform specifications.

| | | | |
|---|---|---|---|
| Clock | 1481MHz | SP/SFU/MEM Units | 4/1/1 |
| Technology | 23nm | RF Size | 256KB |
| Shader Cores | 28 | SP/SFU/MEM CUs | 20/4/8 |
| SIMT Width | 32 | L1/Shared Cache | 64/96KB |
| Warps/ Shader | 64 | Tex./Const. Mem | 48/12KB |
| Schedulers/ Shader | 2 | Inst. Issue/Warp | 2 |

in the baseline platform, each able to issue up to two consecutive instructions from a selected warp per cycle. Similarly, in LOOG, we use a single scheduler that can pick up to two warps and issue up to two instructions from each per cycle. This gives a slight ($< 1\%$) performance bias towards the baseline for neglecting WAR hazards. We need to clarify that LOOG satisfies all operand hazards ensuring correct functional execution.

## 10.2 LOOG Right-sizing

In this section, we examine LOOG's sensitivity to key micro-architectural parameters, in an attempt to analyze its characteristics and rightsize its configuration. LOOG was evaluated under a wide collection of 60 kernels, covering among others the domains of scientific computing, artificial intelligence, graph theory and image processing. The applications were selected from CUDA SDK [84], Parboil [109], Rodinia [108, 141] and the GPGPU-Sim benchmark suite [110]. From the complete collection of workloads, 20 kernels with the most frequent stalling time and lowest warp occupancy were selected as our target evaluation set, since they can benefit the most from an enhanced ILP run-time. These kernels originate from 15 applications, which are listed in Table 10.1.

The baseline GPU architecture as well as our extension were developed and evaluated with the latest version of GPGPU-Sim [110], a broadly used, cycle-accurate GPU simulator. The evaluation platform is modeled after the Pascal

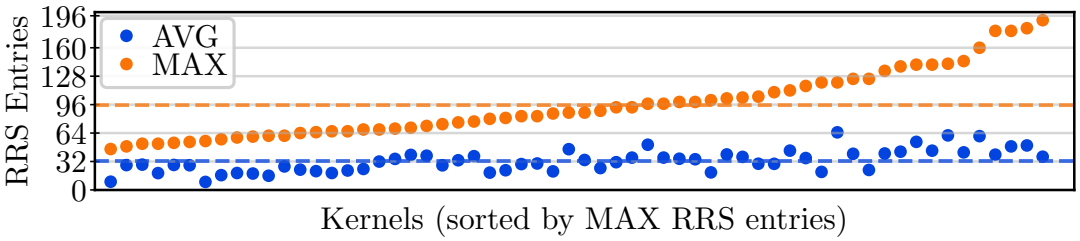



FIGURE 10.6: Average (AVG) and maximum (MAX) number of RRS entries allocated per cycle.

GeForce GTX1080Ti [89, 88]. Its specifications are listed in Table 10.2. All experimental workloads were run with the PTXPlus (SASS) simulation mode enabled. Therefore, compiler assisted optimizations such as instruction reordering are integrated in the simulated instruction flow.

### 10.2.1 Register Renaming Stack Size

Extending the CUs to additionally function as reservation stations applies more pressure on the operand collect stage. As discussed in Sec. 10.1.2 and shown Fig. 10.2, in LOOG the CUs are on average allocated by much longer periods compared to the baseline. This translates to a significant amount of CU stalls. The RRS is a structure designed to alleviate this congestion. Without the RRS, instructions needed to allocate the CUs from the issue to the writeback stage. Using the RRS, instructions can de-allocate the CUs after they dispatch to the execution units, reducing effectively the CU allocation period. In Fig. 10.6, we can see the average and maximum number of RRS entries used by our benchmark set. No application required more than 196 RRS entries while on average, no application allocated more than 64 RRS entries at once.

Furthermore, in Fig. 10.7, the sensitivity to the RRS size is demonstrated. The y-axis values are normalized to the IPC of LOOG with an RRS of size 32. As shown, using a larger RRS of 64 entries brings a 10% gain in performance. The area overhead is minimal; 0.04% and 0.06% when using 64 and 256 entries, respectively. Similarly, the power overhead is only 0.02% and 0.3%, respectively. Since no performance gain from 64 to 256 RRS entries is extracted, we settle the RRS size to 64 in the remaining experiments.

### 10.2.2 Operand Collector Size

The addition of the RRS structure reduced the congestion caused in the operand collect stage. Instructions can de-allocate the CUs right after dispatch. Nevertheless, as depicted in Fig. 10.2, the allocation period of LOOG, even with the RRS, is longer than that of the baseline. This longer allocation period causes

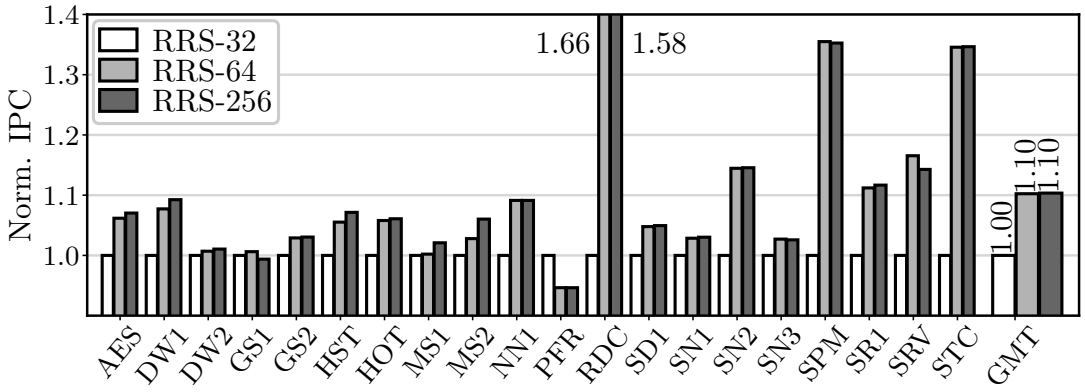



FIGURE 10.7: IPC of LOOG-RRS (with 64 and 256 entries) normalized to LOOG without RRS (LOOG-v0).

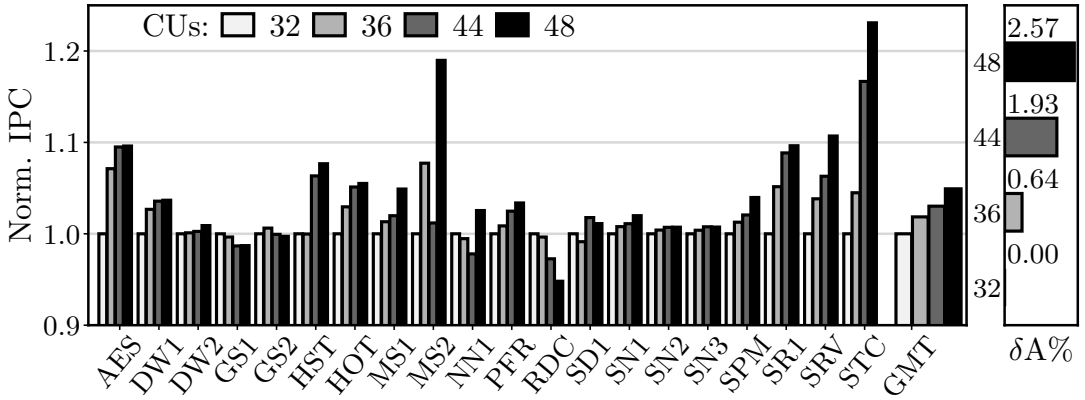



FIGURE 10.8: Sensitivity to the number of CUs and area overhead normalized to the 32 CU configuration.

more stalls due to no available CUs, meaning that LOOG is sensitive to the operand collector size. A larger number of CUs allows more instructions to issue and increases the probability that some of them will quickly resolve their data dependencies and dispatch to the EXUs.

In Fig. 10.8, we quantify the sensitivity of LOOG to the number of CUs. The smaller 32 CU configuration is identical to that of the baseline. We also try larger configurations with 36, 44 and 48 CUs in total. The y-axis values are normalized to the smallest, 32 CU configuration. Increasing the number of CUs from 32 to 48, brings a 5% gain in performance and a 2.57% additional area cost. The additional power overhead in the largest configuration is limited to less than 1%. We note that the larger contribution to the area cost is not due to the hardware needed by the CUs, but due to the x-bar between the RF and the CUs that has to accommodate more outputs. More specifically, the x-bar contributes 91.1% to the additional area cost of the larger CU configurations, and the increased CU count contributes the remaining 8.9%.

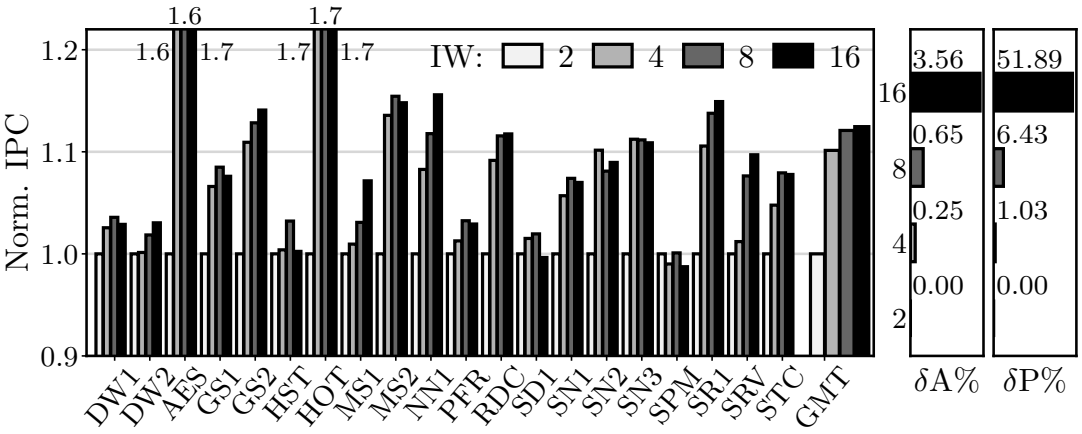



FIGURE 10.9: I-Window width sensitivity, and area and power overhead normalized to the IW-2 configuration.

### 10.2.3 Instruction Window Width

In LOOG, the I-Window width defines the number of instructions per warp that will be considered for issuing by the front-end warp scheduler. Except from that, it also defines the number of instructions that can be fetched and decoded per cycle. The I-Window affects directly the number of instructions that will be candidates for OOO execution. While modern CPUs use deep I-Windows of more than 128 slots [142], in LOOG, to minimize the area overhead, the I-Window width takes values from two to 16 instructions.

Fig. 10.9 shows the effect of the I-Window to LOOG's performance for four I-Window sizes – two, four, eight and 16. The values on the y-axis are normalized to the smallest I-Window size. With an I-Window width of four instructions we extract a speedup of 10% with an area and power overhead of 0.25% and 1.03%, respectively. With a larger I-Window of eight instructions, we can improve the performance by an additional 2% with a small power and area cost. However, the performance gain of a 16-wide I-Window is limited and the power overhead is 65% more than the baseline, making it impractical. In a few cases, like the HST kernel, we can see a small IPC degradation with a deeper I-Window. This happens when the issued instructions form a chain of long-latency dependencies, thus blocking other, possibly independent instructions from allocating the CUs. Furthermore, to simplify the design complexity and minimize LOOG's overhead, a warp is considered for fetching new instructions only when there are no more valid instructions in the I-Buffer. Thus, a deep I-Window will be fully exploited after a fresh instruction fetch. As instructions are issued and executed, the I-Window will be shrinking in depth, limiting the ILP opportunities.

In LOOG, the priority to issue is given to the next in-order instruction of every warp. Thus, in regular workloads with sufficient data parallelism, LOOG

is mostly dispatching instructions in-order. Only in irregular cases, where the TLP and the warp occupancy are not sufficient, LOOG will begin dispatching instructions OOO to improve utilization.

# Chapter 11

# Experimental Evaluation

## 11.1 LOOG vs. Warp Scheduling Policies & TLP Throttling

In this section, we evaluate the run time performance of LOOG compared to conventional GPU architectures. Specifically, for the baseline, we consider three different warp schedulers: GTO [97], LRR and TLS [98], since traditional GPU workloads are sensitive to the warp issue scheduling policy. Each of the baseline configurations is better suited for different sets of applications. LOOG uses the exact same configuration of CUs as the baseline and a 4-slot I-Window.

In order to evaluate LOOG's efficiency against TLP throttling mechanisms, we implemented the Best Static Wavefront Limiting (BSWL) technique [97]. BSWL is a static TLP throttling technique in which for every application, all the possible CTA limits are scanned and the one that performed the best is used. BSWL targets cache-sensitive applications, but has little to no effect on cache insensitive workloads. BSWL has been shown to outperform both the CCWS [97] and Dyncta [99] TLP throttling strategies.

We focus our analysis on the 20 target kernels with the most frequent stalls and least mean occupancy which were identified in Sec. 7 and Table 10.1. Fig. 11.1 shows LOOG's IPC normalized to the four baseline configurations described above. Among the baseline workloads, the greatest performance variation is observed in the SR1, NN1 and MS2 benchmarks, with a 29.8%, 22.1% and 17.6% spread, respectively. The average performance variation per kernel, when varying the warp scheduler is 8.65%. Despite the considerable kernel-by-kernel variation, when averaging the performance of all kernels, we observe a limited variation of about 1% among the baseline schedulers. Since the target kernels were not chosen based on their access locality or cache footprint, they do not coincide with GTO's, LRR's or TLS's target workloads.

LOOG outperforms all three baseline scheduler by 22% to 23%. With the

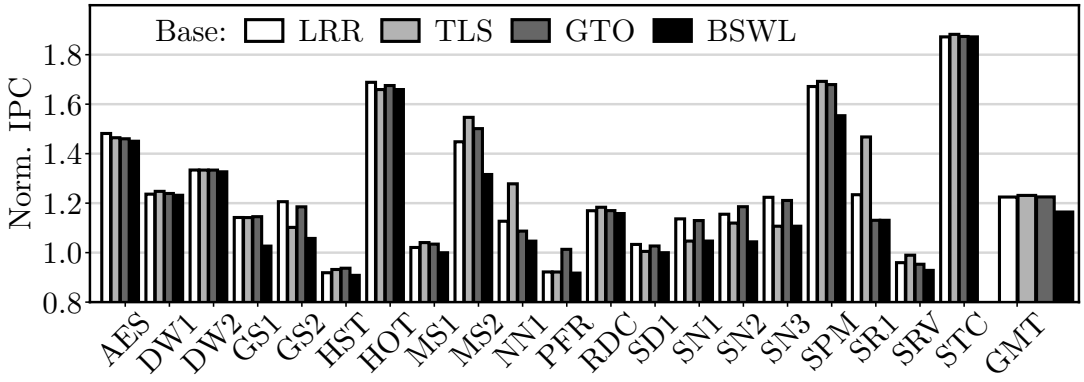



FIGURE 11.1: LOOG IPC normalized to the baseline GPU platform using three different warp scheduling policies (LRR, TLS, GTO), and the BSWL technique.

exception of the HST and PFR kernels, LOOG performs better than the baseline in every kernel. As expected, the BSWL technique that relies on oracle knowledge to discover the best performing CTA limit per kernel, is the fastest among the baseline policies. Nevertheless, LOOG is faster by 16% on average compared to BSWL.

LOOG's slightly lower IPC in the PFR and HST kernels, by 7% and 8% respectively, is due to the altered scheduling sequence. More specifically, in traditional GPUs, when instructions issue to the OC stage, they have already passed the scoreboard check, meaning they are free of dependencies. In LOOG, there is no dependency-checking mechanism before issuing; instructions wait in the CUs to resolve RAW dependencies. In some cases, a chain of data dependent instructions on a long-latency memory request is formed, leaving no space for other warps to issue potentially dependency-free instructions. Indeed, in the HST and PFR kernels, we saw an average CU allocation period two times longer than the average, indicating increased congestion in the OC stage.

## 11.2 LOOG vs. Dual-Operation Mode Mechanisms

Warped-P [113] and HAWS [114] are alternative GPU architectures that enable OOO execution, similarly to LOOG. HAWS suggests a modified compiler infrastructure combined with a hint-assisted warp scheduler to implement OOO execution, and Warped-P is a purely hardware-based mechanism. HAWS targets the AMD architectures [85] and Warped-P targets NVIDIA architectures. Despite few implementation differences, both techniques are based on the same concept; in the shadow of long-latency memory stalls the warp scheduler enters a

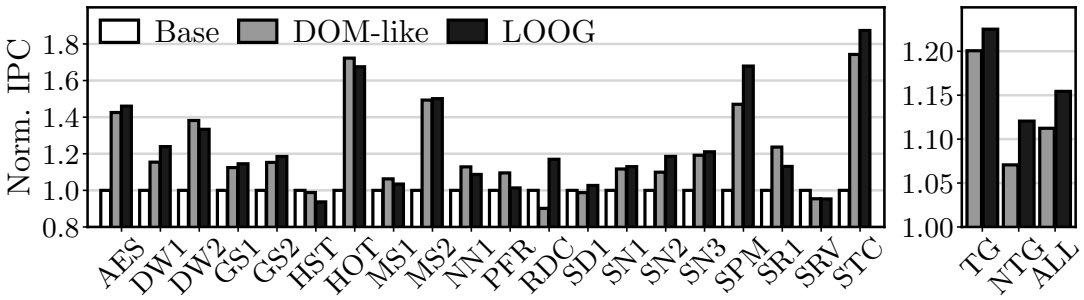
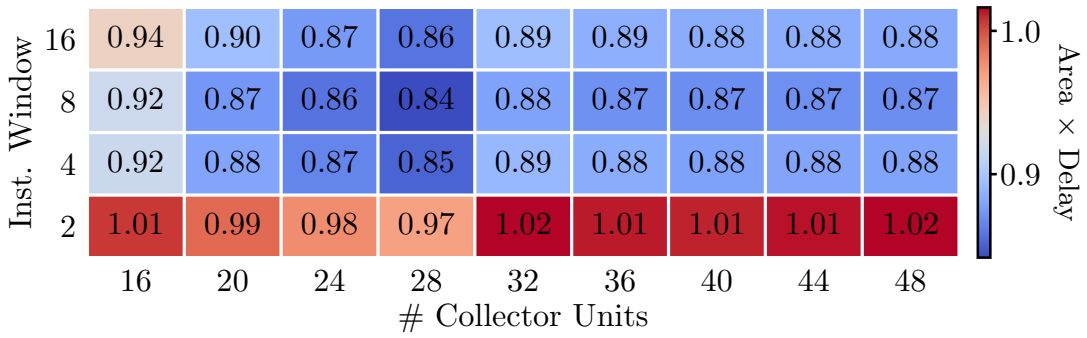



FIGURE 11.2: Comparison of LOOG against a Dual-Operation Mode [113, 114] analogous mechanism.

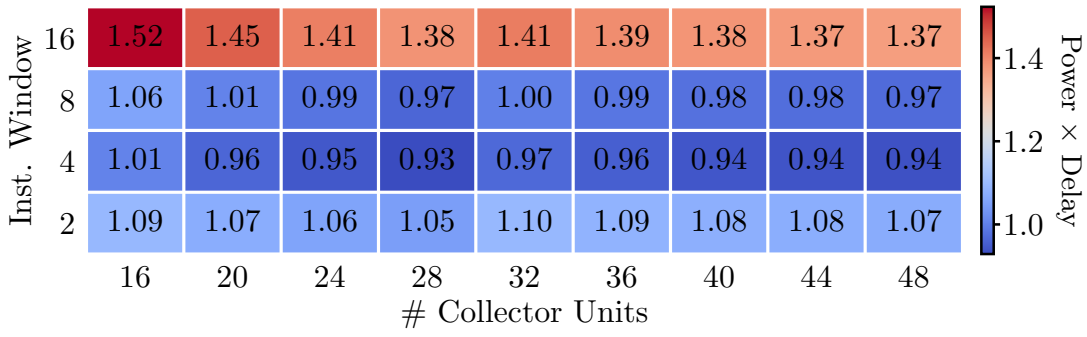
special pre-execution mode, and continues to fetch, decode and execute instruction in an OOO fashion to keep the back-end pipeline busy. When the stalled warps resume normal execution, pre-calculated ALU operations are converted to memory load operations from a renaming unit. Due to their similarities, we will refer to these techniques as Dual-Operation Mode (DOM).

In order to evaluate LOOG against DOM, we extended GPGPU-Sim [110] with a DOM-like mechanism. Similarly to the original DOM techniques, DOM-like allows for load but not store re-ordering. Furthermore, DOM mechanisms are capable of considering instructions further ahead in the instruction stream than LOOG. To simulate this behavior, in DOM-like we i) increase the number of collector units from 32 to 64 and we use an I-Window of size 16, instead of four. Fig. 11.2 shows the performance comparison between LOOG and the examined DOM-like GPU. The y-axis shows the IPC of both techniques, normalized to that of the baseline, conventional GPU architecture. On the right, we have included the geometric means of the target (TG), non-target (NTG) and all (ALL) kernels for DOM-like and LOOG. As we can see, the performance values of both techniques are very close, but nevertheless, LOOG offers on average an additional 3% run time gain among the target kernels, 5% among the non-target kernels and 4% across all 60 general-purpose kernels.

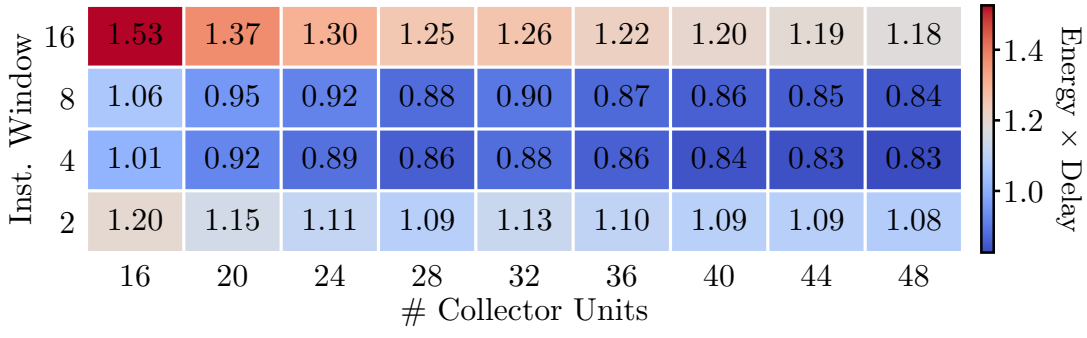
DOM does not pre-execute store operations and, load operations do not modify the architectural state either, but only warm-up the L1 cache. Issuing a speculative load operation in the shadow of a long-latency memory stall can aggravate the memory congestion and deteriorate performance. Additionally, there is no guarantee that the prefetched data will not be over-written before the warp resumes normal execution and issues the actual load operation. On the other hand, LOOG offers true load-store re-ordering, in the absence of address dependencies. DOM mechanisms are designed to mitigate a single pathogenic scenario: stalls due to long-latency memory operations. LOOG is an inherent OOO, non-speculative architecture, able to improve performance under a wider range of pathogenic scenarios, e.g. long-delay transcendental operations,




(A) Area Delay Product.




(B) Power Delay Product.




(C) Energy Delay Product.

FIGURE 11.3: PDP and EDP of various LOOG configurations, normalized to the baseline, in-order GPU platform.

excessive control and structural stalls, and deficient thread level parallelism.

## 11.3 Evaluating LOOG's Area, Power, and Energy Efficiency

In Sections 10.2.2 and 10.2.3, we showed that LOOG is sensitive to the number of CUs and I-Window width. Larger configurations offer better performance but come with greater power and area budgets. The Area-Delay-Product (ADP), Power-Delay Product (PDP), and Energy-Delay Product (EDP) metrics are used as a figure of merit of the area and energy efficiency of hardware extensions. Figure 11.3 shows the ADP, PDP, and EDP of LOOG with various I-Window

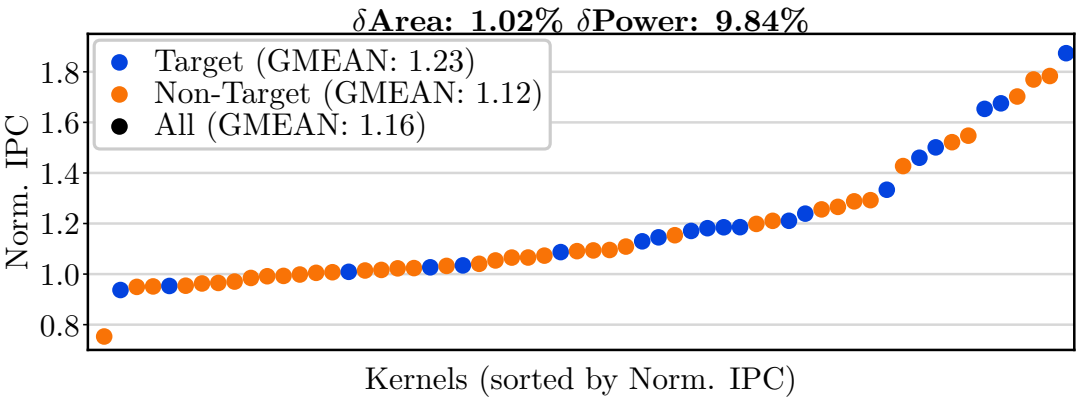



FIGURE 11.4: LOOG IPC normalized to the baseline. LOOG demonstrates its potential as a general-purpose u-arch.

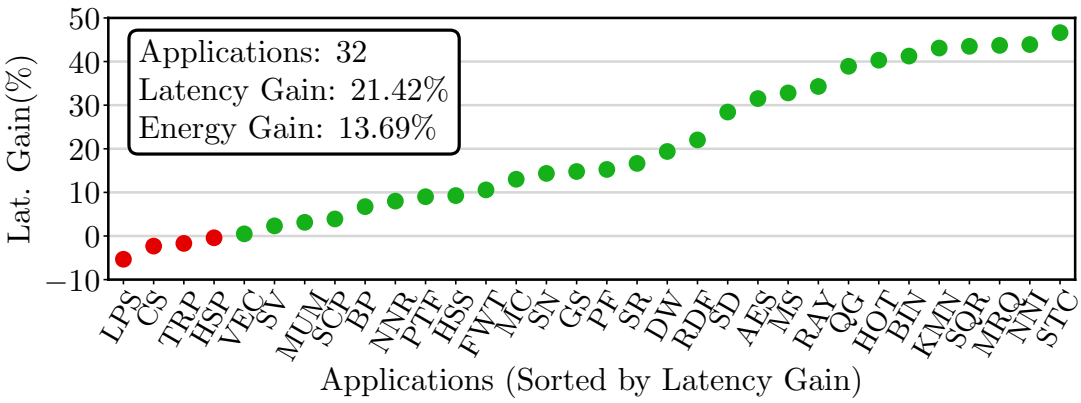



FIGURE 11.5: LOOG latency gain percent across 32 applications.

and CU sizes, normalized to the baseline, in-order GPU platform, as described in Table 10.2. Lower values correspond to more efficient configurations. The best LOOG configuration achieves 16%, 7%, and 17% gains in terms of ADP, PDP and EDP respectively, w.r.t. the baseline architecture. Considering all three metrics, the best compromise is achieved when using an I-Window of four instructions with 28 CUs. However, if higher performance is required, the configuration that contains a 4-slot I-Window and 44 CUs is the most preferable and EDP efficient design point.

## 11.4 LOOG: Promising u-Arch Alternative

In the previous section, LOOG demonstrated its superior performance and energy efficiency against a set of 20, low TLP target kernels. In principle, enabling more aggressive ILP to complement the existing TLP can be beneficial for every kernel. Indeed, in this section, we show that LOOG's efficiency is also evident for general-purpose kernels. Fig. 11.4 shows the normalized IPC of all 60 kernels,

coming from a wide range of application domains and benchmark suites [108, 109, 84, 110]. We consider a LOOG configuration with a 4-slot I-Window, 32 CUs and 64 RRS entries. The area overhead of our lightweight extension is only 1.02% and the additional power cost is 9.84%. Interestingly, LOOG is 23% and 12% faster than the baseline on average in the sets of target and non-target kernels, respectively. When averaging across all kernels, LOOG offers 16% performance gain. *This is a strong indication that a light-weight OOO execution scheme like LOOG forms a very promising alternative to traditional GPU architectures.*

We further evaluate LOOG's efficiency regarding end-to-end application acceleration, i.e. for real workloads deployed in GPUs combining one or more kernels. Fig. 11.5 reports the gain in latency, of LOOG compared to the baseline GPU, for the 32 applications found in the considered benchmark suites. As depicted, only four applications show a minor slowdown of 1-5%, while for the rest 28 applications, the latency gain reaches up to 50%. On average, LOOG saves 21.42% of GPU cycles when considering entire GPU-accelerated applications. The gain in cycles is accompanied with a 13.69% gain in energy, across all 32 GPU accelerated applications.

## 11.5 In-Order GPU and LOOG Co-Exploration Analysis

Having discussed how LOOG [143, 144], a non-conventional GPU architecture, is implemented and fine-tuned, this section demonstrates an extensive exploration of the design space of traditional GPUs and LOOG architectures simultaneously. To evaluate the efficiency of an OoO LOOG architecture when combined with both older and newer GPU generations, we explore three L2 Cache (L2C) sizes, and three execution unit (EXU) counts, i.e. 9 baseline configurations. When scaling the EXUs, both the Scalar Processors (SPs) and the Special Function Units (SFUs) are scaled. Then, the LOOG OoO extension is applied on top of these 9 configurations, and for each of them, four I-Window sizes and four CU sizes are considered, i.e. 16 LOOG configurations in total. Therefore, the exploration space is composed of $9 \times 16 = 144$ LOOG, and 9 baseline instances. All 60 kernels of our benchmark collection are executed on each of these distinct instances.

Figure 11.6 shows the average normalized latency for all baseline and OoO architecture configurations. Values below one correspond to a latency gain compared to the baseline. As shown, the smallest OoO configurations with 2-slot I-Window and 32 CUS, match the performance of in-order GPUs. More

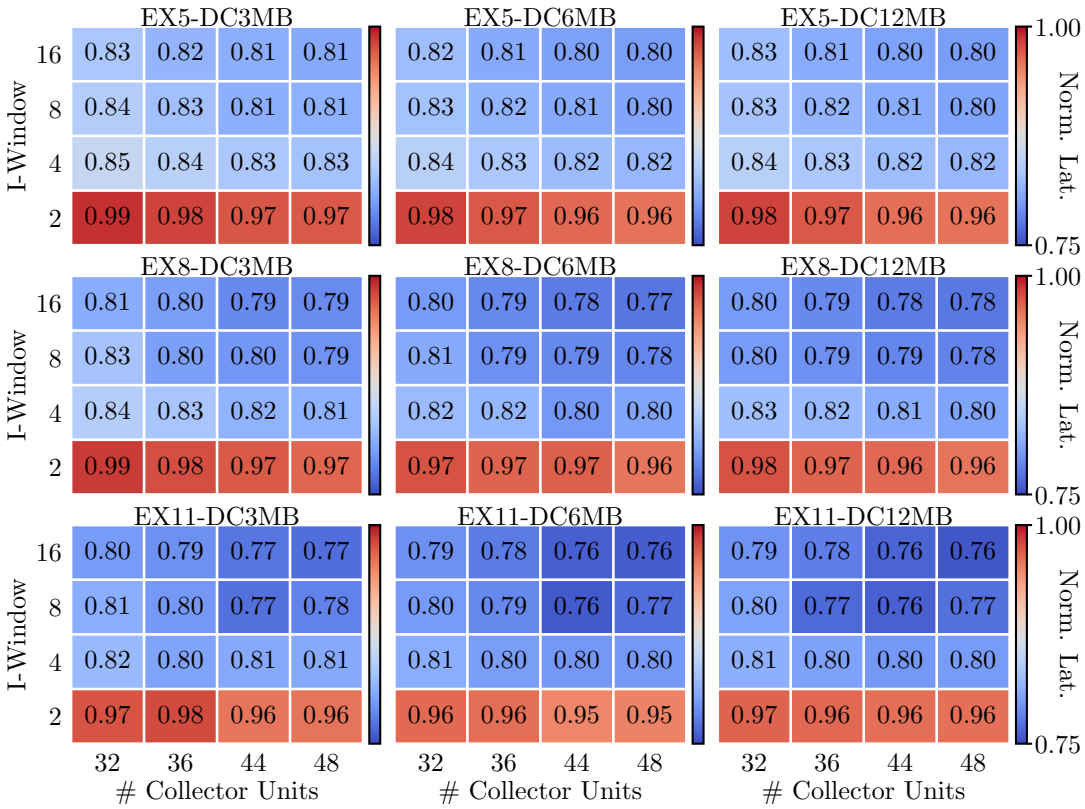



FIGURE 11.6: Normalized latency of various baseline and OoO GPU configurations. Lower values correspond to more efficient configurations.

aggressive configurations, with 48 CUs and 16-slot I-Window, achieve up-to 25% latency gain. As anticipated from Sec. 10.2, there is a higher sensitivity to the I-Window than to the CUs. However, I-Windows of size 16 are impractical due to the large power overhead of more than 50%. In all in-order GPU configurations, both those closer to the Pascal [89] architecture, with 5 EXUs and 3MB L2C, and those closer to more modern GPU architectures, a speedup of 14-20% is observed with the minimal overhead 36 CU, 4-slot I-Window configuration. Generally, the latency gain increases with the number of EXUs, which shows that the OoO extension exploits the extra resources more efficiently than in-order GPUs. Focusing on the most powerful baseline GPU configuration, the one that contains 11 EXUs and 12 MB L2C, we observe that with an I-Window of size four and 36 CUs, a 20% latency gain is achieved on average. Overall, we conclude that the benefits coming from an OoO extension can be seen even with minimal additional resources in terms of I-Window and CUs, and are relevant to a wide range of in-order GPU architectures.

Fig. 11.7 demonstrates the normalized power-delay product, or energy, consumed on average to run the benchmarking kernels. It is evident that, using very large I-Windows results in a overheads in terms of energy-efficiency. Similarly,

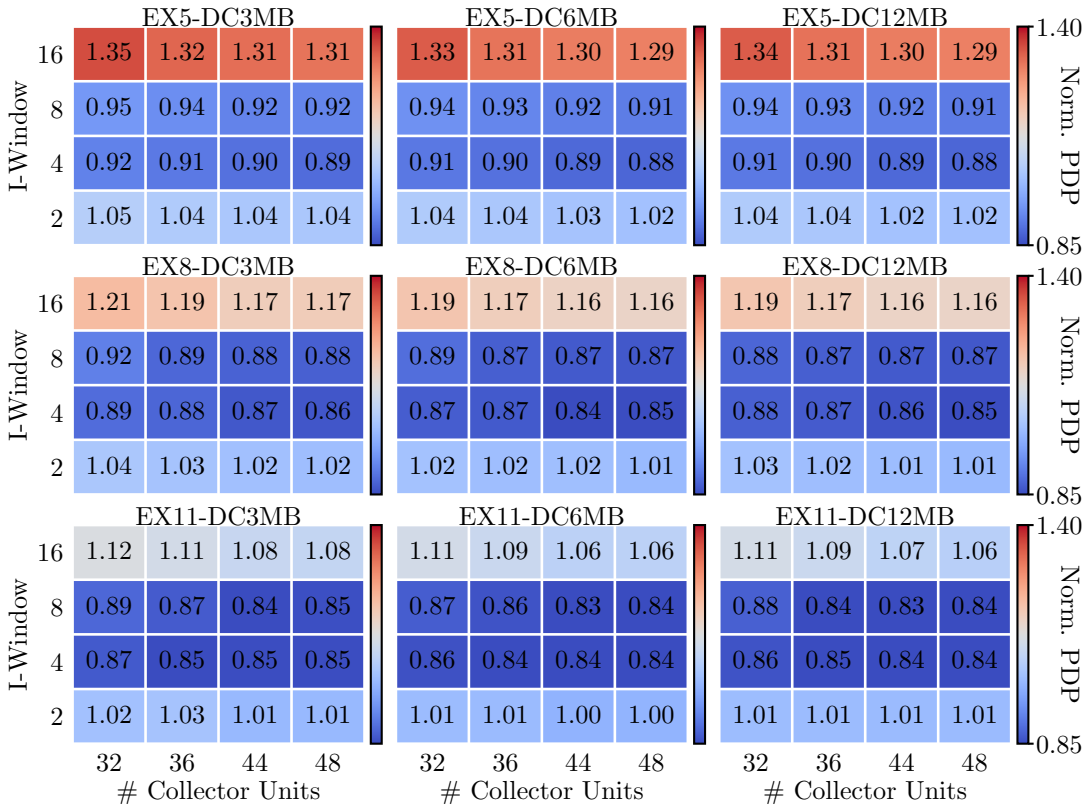



FIGURE 11.7: Normalized power-delay product of various baseline and OoO GPU configurations. Lower values correspond to more efficient configurations.

too small I-Windows do not manage to capture ILP opportunities effectively and perform worse that the in-order baseline. With an I-window of 4, a reduction in energy in the order of 11-17% is extracted. The energy gain of the LOOG OoO extension becomes more evident when combined with more powerful GPU architectures. Two factors contribute to this observation, i) the OoO extension exploits more efficiently the additional EXUs, and ii) the power overhead is reduced since it is independent of the EXUs and the data cache size. More specifically, with 36 CUs and 4-slot I-Window, the OoO extension provides on average 15% improved energy efficiency compared to traditional GPU architectures. The area and power overhead of this configuration is limited to 0.92% and 5.74%, respectively.

A more detailed evaluation of the optimal instance of LOOG (*OPT*), against high-end in-order GPUs and state-of-art OOO GPU schemes, which is DOM [113, 114] is provided in Fig. 11.8. Modern GPU generations [90, 107] contain large L2 cache capacity and a plethora of execution units (EXUs). The most powerful and future-looking baseline configuration simulated in Fig. 11.6 is the one using 11 EXUs and 12MB L2C. The DOM mechanism was summarized in Sec. 11.2. We note that our DOM-like mechanism provides an upper-bound estimate to

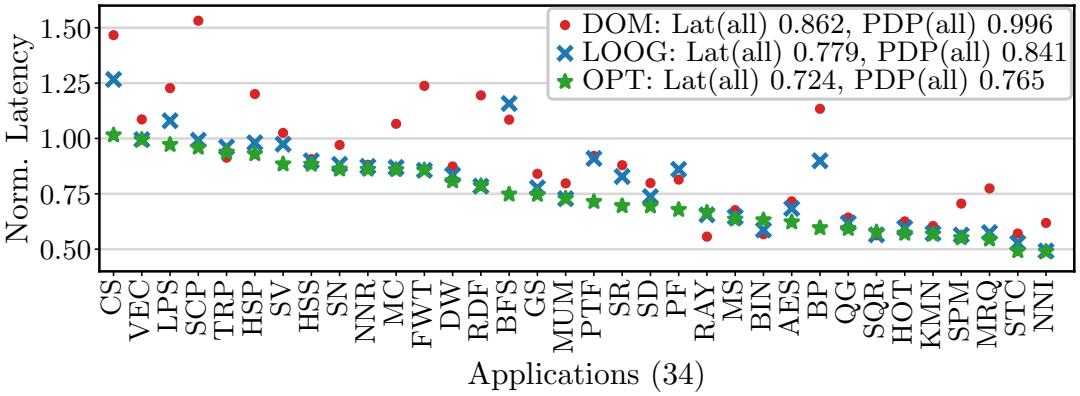



FIGURE 11.8: End-to-end latency comparison of 34 applications for the DOM, LOOG, and LOOG-OPT mechanisms. The latency is normalized to the baseline GPU platform. Lower values correspond to greater latency gains.

the actual DOM mechanisms.

In Fig. 11.8, the performance of entire applications is examined, considering the time spent in GPU kernels. On average each application contains two kernels. In the x-axis of Fig. 11.8, the short-names of the applications listed in Table 10.1 are shown. The y-axis shows the latency normalized to that of the baseline in-order GPU with 11 EXUs and 12MB of L2C. Apart from *DOM*, we see the performance of the optimized LOOG (*OPT*), and the performance of the non-optimized LOOG (*LOOG*). Values below one indicate gain in latency w.r.t. the baseline and lower values correspond to greater gains in latency.

OPT and LOOG demonstrate similar behavior, since are both instances of the same architecture, with OPT clearly outperforming the non-optimized LOOG variation. DOM on the contrary exhibits a larger spread in latency. On average, *OPT* is 13.8% faster than DOM. This showcases the impact of re-ordering both load and store operations. DOM does not pre-execute store operations or any operations ahead of store instructions. LOOG on the other hand can re-order all types of operations, including loads and stores, in the absence of address dependencies. Only in three cases, which are the Transpose (TRP), RayTracing (RAY) and Binomial (BIN) applications, DOM achieves greater gain in latency compared to the optimized LOOG instance, by 2.4%, 10% and 6.5% respectively. This is the result of the deeper I-Window exploited by DOM in applications with fewer load-store operations.

OPT provides on average a narrow, but statistically robust, 5.5% additional gain in latency compared to the non-optimized LOOG. OPT is slightly slower, by 1.5%, than the baseline in only one case, the Convolution (CS) application. This is related to the fact that the OoO dispatching mechanism alters the original in-order instruction scheduling sequence. In rare cases, inter-dependent

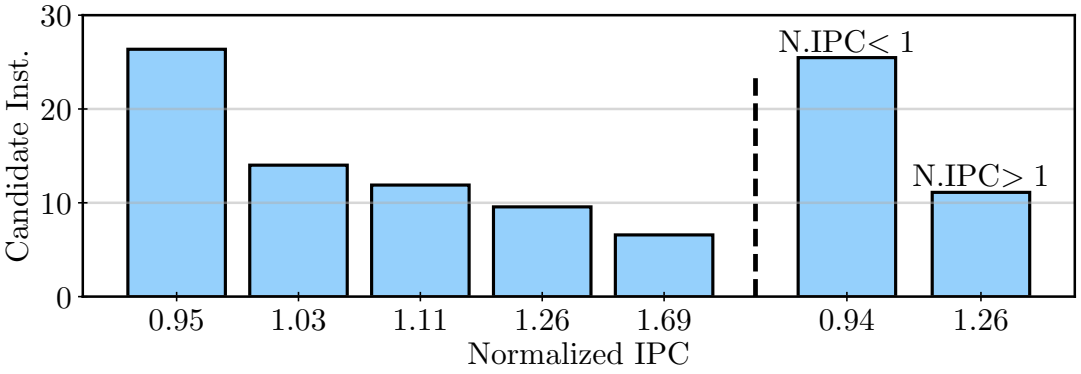



FIGURE 11.9: Average candidate instructions for issue per cycle, grouped by normalized IPC.

instructions allocate the CUs, and block other instructions from issuing. In this scenario, a scheduling mechanism that only issues instructions after checking that there are no data-dependencies, as the scoreboard in the baseline architecture, can result in better utilization of the execution pipeline. Apart from CS, all the remaining applications exhibit up-to 2× improved run time. The geometric mean across all 34 applications shows a 27.6% latency gain.

Interestingly, despite the increase in power shown in Table 9.1, the optimized LOOG-based OoO GPU achieves a geometric mean energy reduction of 23.5%. The non-optimized LOOG provides 15.9% gain in energy, while with DOM [113] the energy consumption is not improved. It is evident that the power overhead of the incorporated OoO mechanisms is completely outbalanced by the achieved performance gains while the fine-tuned resource allocation of the proposed OoO GPU solution allows for an extra 7.6% gain in geo-mean energy efficiency. We conclude that LOOG-based, OoO GPU architectures have great potential in accelerating general-purpose workloads, providing significant gains in terms of energy and performance.

## 11.6 Instruction Level Parallelism Analysis

The exploitable Instruction-Level Parallelism is a characteristic that is inherent to the workload. It represents the number of independent instructions that can be executed in parallel given enough resources. Fig. 11.9 shows for the baseline architecture, the average number of independent instructions per cycle, divided into groups according to the normalized IPC. A pattern is clearly depicted; applications with lower degree of exploitable ILP profit more from LOOG. The ILP is typically limited either because the warp occupancy of the kernel is low to begin with, or because warps cannot be considered for issue due to control, structural or data hazards. Both clauses of this conclusion are in agreement with

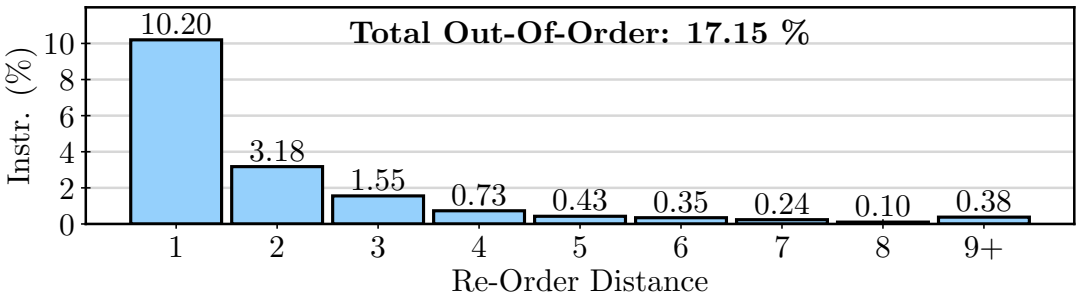



FIGURE 11.10: Percentage of instructions that dispatched OOO and their distance from the instruction in program order.

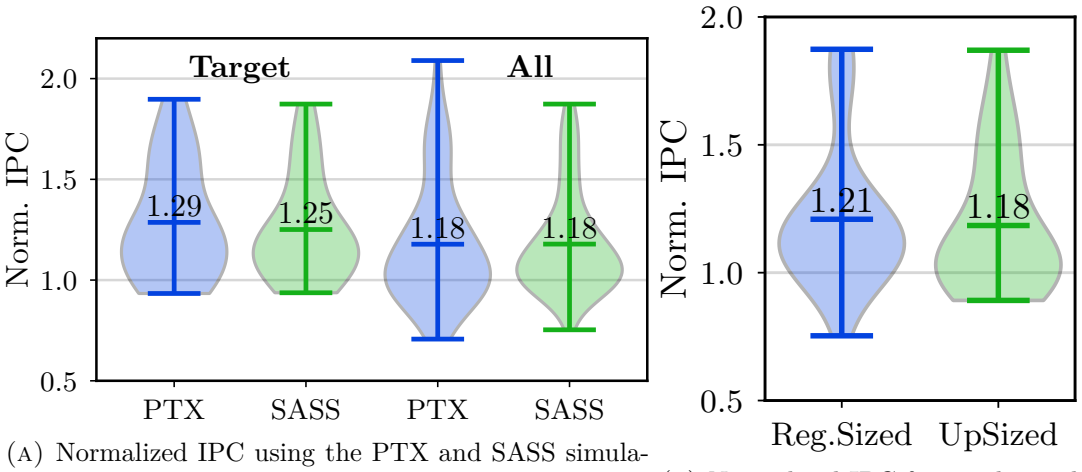
our original hypothesis. Firstly, kernels with low occupancy fail to efficiently hide the latency of memory and other long-lasting operations under a TLP only scheme. Secondly, LOOG can alleviate the stalling time of kernels with frequent structural, control or data hazards by considering a larger window of candidate instructions and thus managing to find hazard-free instructions to issue and execute.

The LOOG front-end warp scheduler prioritizes for issuing the first in-order instruction of every warp, before looking deeper into the I-Window. The main motivation for this is that in "regular", data-parallel kernels, LOOG will execute instructions very similarly to the baseline, maintaining equivalent performance. In Fig. 11.10, we can see the average re-order distance of all 60 kernels in our benchmark suite. On average, 17.15% of all instructions execute out of the program order. Most of the re-ordered instructions (10.2%) only surpass one instruction ahead in program order and less than 1% of the instructions are re-ordered more than five places ahead. Note that, for this analysis, an I-Window of 16 instructions was used to maximize the instruction re-ordering possibilities.

## 11.7 Compiler-based Re-Ordering and LOOG

GPGPU-Sim [110] provides two simulation modes: PTX and PTXPlus (SASS). The Parallel Thread Execution (PTX) [124] is an intermediate, hardware-independent, instruction set used by NVIDIA for portability and stability. The PTX ISA is assembled into SASS, the native GPU ISA, before executing. Compiler optimizations such as static instruction re-ordering and register allocation are only integrated in the SASS binary and not in the PTX pseudo-assembly. Thus, the PTX simulation mode is generally less accurate than SASS.

The SASS simulation mode was used in all our experiments, therefore LOOG has already demonstrated compelling performance gains when combined with static code optimizations. In Fig. 11.11a we evaluate the effect of compiler optimizations on LOOG's performance, examining PTX vs. SASS binaries.




(A) Normalized IPC using the PTX and SASS simulation mode.

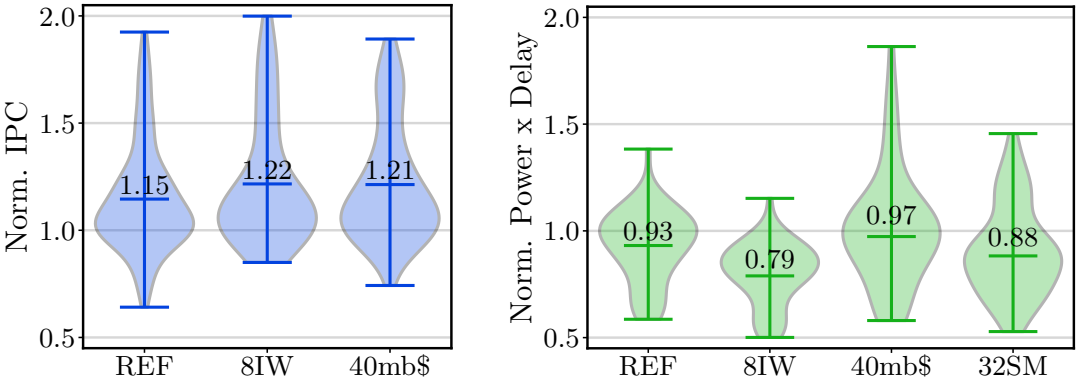
(B) Normalized IPC for regular and up-sized inputs.

FIGURE 11.11: The proposed OOO execution model is relevant in the presence of compiler-based optimizations, and up-scaled workloads with additional TLP.

The y-axis shows the IPC of LOOG using PTX and SASS, normalized to the corresponding baseline. The violin plots correspond to the *target* kernels with PTX and SASS, as well as the full set of *all* kernels with PTX and SASS. For the 20 target kernels, we see that on average, the static code optimizations withdraw only 4% of LOOG's speedup. Across all 60 kernels, we observe similar average speedup. As expected, the all-SASS distribution has lower best-case IPC than the all-PTX, given that static instruction re-ordering optimizations are only integrated in the SASS binary, thus limiting, but not canceling, the dynamic reordering opportunities. In summary, the dynamic ILP exploited by LOOG provides compelling performance gains, even when combined with compile-time optimizations.

An interesting argument related to LOOG-like architectures has to do with the workload input sizes. Increasing an application's input set provides enough work to support more active warps and exploit higher degree of TLP. Given that LOOG's target workloads fail to efficiently exploit TLP, should we assume that only smaller scale workloads benefit from LOOG?

In Sec. 11.4, we established that although LOOG has a specific workload target set, any application can profit from ILP as a complement to the existing TLP model. To further stress LOOG's sensitivity to increased TLP kernels, we evaluate LOOG on larger workloads. Specifically, we upscaled the input sizes of 12 applications (27 kernels) from the Rodinia [108] suite, by a factor of five, resulting in 2.85x more CTAs issued per kernel invocation. Figure 11.11b shows the normalized IPC distribution of LOOG over the baseline for the regular and up-sized datasets. By comparing the two violin plots, we observe a minor effect




(A) Impact of modern micro-architectural advances in LOOG's execution efficiency.

(B) LOOG's power efficiency compared to simpler, alternative base configurations.

FIGURE 11.12: Performance and power efficiency of alternative LOOG and baseline configurations. All 60 kernel have been considered.

on the latency gain of LOOG, i.e. from a geometric mean of 21% to 18%. In conclusion, the speedup of LOOG is not diminished in up-scaled, higher TLP workloads w.r.t. conventional GPU architectures.

## 11.8 Compatibility with Recent U-Arch Advances

The wide applicability of GPU accelerators drive rapid advances in GPU micro-architectures. Modern NVIDIA GPU generations have introduced novel features, not present in older generations. The baseline LOOG architecture is based on the NVIDIA Pascal [89] generation, however it can be easily adapted and implemented on top of newer GPU architectures. In this section, we study how two features introduced in the NVIDIA Volta [90] and Ampere [107] architectures impact the efficacy of the proposed LOOG execution model. The Volta architecture increased the instruction issue capacity to eight instructions per cycle, using four warp schedulers, each capable of scheduling up-to two instructions of a warp per cycle, while the Ampere architecture extended the L2 cache capacity to 40MB. Our baseline Pascal architecture uses a 3MB L2 cache and has an issue width of four instructions per cycle.

To evaluate the impact of these two new features on LOOG, we modify both LOOG and baseline to use 40MB L2 cache, or 8 instructions issue width. In Fig. 11.12a, the violin plots from left to right correspond to the IPC of LOOG normalized to the baseline, when both systems use the original configuration

(*REF*), 40MB L2 cache (*40mb$*), or 8 instructions issue width (*8IW*) respectively. All 60 general-purpose kernels are included in the violin plots. Higher values correspond to greater speedups. As shown, the average normalized IPC of the two new configurations is higher (1.22 and 1.21, respectively) than this of the reference (1.15), meaning that the efficacy of LOOG is not diminished, but on the contrary, LOOG seems to exploit more efficiently the additional cache capacity and issue width. This suggests that the LOOG execution model is compatible with recent micro-architectural advances.

## 11.9 Alternative ISO-Power Configurations

In Sec. 11.4 we demonstrated how the proposed OOO execution model provides 16% performance gain on average across a wide range of general-purpose kernels. However, this comes with an increased power budget of approximately 10%. In this section, we study whether greater power efficiency can be extracted by alternative GPU configurations with more typical micro-architectural enhancements than LOOG. More specifically, we consider three alternative base configurations: a) using an issue width of eight instructions per cycle (*8IW*), b) using a 40MB L2 cache (*40mb$*), and c) using 32 SMs (*32SM*).

In Fig. 11.12b we can see the Power-Delay Product (PDP) of the most power efficient LOOG configuration as defined in Sec. 11.3, normalized to the three alternative baseline configurations, as well as the default baseline (*REF*) architecture with four instructions issue width, 3MB L2 cache and 28 cores. All 60 general-purpose kernels are included in the violin plots. Lower values correspond to less power efficient baseline configurations. The average of all violin plots being less than one means that LOOG is more power efficient than all the alternative, simpler base configurations. On average, the *32SM*, and *8IW* configurations are less efficient in terms of power than the reference baseline meaning that the IPC gain is out-weighted by the added power overhead. The *40mb$* configuration with an average normalized PDP of 0.97 shows that increasing the L2 cache from 3MB to 40MB provides a power efficiency 3% lower than the LOOG extension. However, when looking at the area overhead, LOOG induces only 1.3% area increase, while increasing the L2 capacity to 40MB results in 5.4% area increase. Having an issue capacity of eight instructions per cycle, or 32SMs comes with an area overhead of 6.5% and 14% respectively. In summary, LOOG is more power- and area-efficient than alternative, simpler micro-architectural configurations.

# Chapter 12

# Conclusions

## 12.1 Conclusions

Living in the era beyond Moore's law and Dennard's scaling, the path to achieve top performance is today more than ever via software customization, exploitation of hardware accelerator systems, and the design of domain-specific architectures. In the scope of this thesis, we approach high-performance computing from two different points of view: meticulous software customization with HPC strategies, and hardware specialization for improved performance and energy efficiency.

The first part of this thesis brought cutting-edge, High Performance Computing strategies to the field of beam longitudinal dynamics, by completely refactoring the *BLonD* simulator. Special attention was given to the user-friendliness, since scientific simulators like *BLonD* are intended to be used by non-computer scientists who value productivity-oriented programming languages, such as Python. *BLonD++*, focused on vertical scalability and intra-node efficiency, and demonstrated 18× reduced latency compared to *BLonD*. This can be especially helpful for conducting design space exploration studies, that are often composed of a large number of medium-sized simulation workloads. Then, to accommodate high-precision studies that require running a small number of large-scale simulations, we designed a hybrid, MPI-over-OpenMP code called *HBLonD*. *HBLonD* integrates an automatic load-balancing system to assist poorly balanced workloads. Furthermore, the implementation of two physics-motivated approximate computing methods optimized the network traffic and minimized the communication and synchronization overhead, without noticeable deterioration of the simulation accuracy. When run on 32 nodes, *HBLonD* demonstrated 43-56× speedup across three real-world test-cases. Finally, to anticipate the ever-growing demand for larger input workloads, longer and more simulation studies, we combined MPI with CUDA to build a GPU-accelerated, distributed simulator, called *CuBLonD*. *CuBLonD* was built on top of *HBLonD*, therefore inheriting all its advanced capabilities such as

dynamic load-balancing and approximate computing techniques. On 32 GPU platforms, *CuBLonD* showed an up to two orders of magnitude reduction in execution time w.r.t. a 20-core *BLonD++* CPU instance.

The second part of this thesis moved towards micro-architectural customization to achieve higher performance and energy efficiency in modern applications. More specifically, it focuses on GPUs, the most widespread platform for general-purpose workload acceleration. The diverse nature of modern GPU accelerated workloads implies the existence of kernels with various characteristics. Among them, we observed a significant class of kernels, which fail to maintain enough active warps to hide the latency of memory operations relying solely on thread-level parallelism. As a result, they suffer from frequent stalling and poor resource utilization. These irregular kernels perform sub-optimally under the original, Thread-Level Parallel model. To remedy this shortcoming, this thesis suggested LOOG, a dynamic, hardware-only, general-purpose, light-weight Out-Of-Order architecture for GPUs. LOOG exploits Instruction-Level Parallelism to enhance the existing TLP model and improve the efficiency of underperforming kernels. The area and power overhead of LOOG is minimized by re-purposing existing hardware components and leaving intact the most resource-hungry structures. A thorough exploration of the design space of both traditional GPU architectures and LOOG proved that LOOG can be applied on top of a wide range of GPU models, both low- and high-end. LOOG outperforms conventional architectures, delivering an average speedup of 27.6%, and reducing energy consumption by 22.4%, with an area overhead of only 1.02%.

In conclusion, this thesis proposes alternative approaches to achieve high performance in terms of execution time and energy-efficiency. Firstly, we explored meticulous software customization strategies to take advantage of existing multi-processors and hardware accelerators, while providing an easy-to-use interface to the user base. The dramatic reduction in execution time in the field of beam longitudinal dynamics enables scientists to simulate scenarios that combine more complex physics phenomena with finer resolution and larger number of simulated particles. These complex, accurate and fast simulations have proven essential in the field of beam dynamics to overcome current technological limitations, plan the upcoming upgrades of particle accelerators, and design future machines that will help science advance further. For instance, in the Super Proton Synchrotron (SPS), the speedup brought by *HBLonD* enabled users to simulate more realistic operational scenarios and discover multi-bunched effects, impossible to observe with a reduced number of bunches. These results lead to important hardware design modifications [42, 43], which would otherwise not

have been realised. Secondly, we focused on micro-architectural specializations of GPU platforms to exploit Instruction-Level parallelism to complement the existing Thread-Level parallel model and boost the execution and energy efficiency of modern workloads. By demonstrating the potential of LOOG to accelerate a wide collection of applications originating from multiple computing domains, we conclude that LOOG is a promising alternative GPU micro-architecture that can expand the applicability of future GPU platforms even further.

## 12.2 Future Research Directions

Both parts presented in this thesis can be further advanced or extended towards new research directions. Regarding the *BLonD* code, our main future development ideas focus on the following topics:

- Develop a distributed implementation of the input-output operations and particle distribution generation phase. This will enable the simulation of much larger input workloads, in the order of tens or hundreds of billion macro-particles.

- Investing in a more intelligent dynamic load-balancing scheme. Specifically, a load imbalance predictor mechanism can be implemented so that the DLB scheme will be able to pro-actively resolve load imbalances instead of reactively.

- Fine tune multi-node heterogeneous simulations. Based on the existing capabilities of the Dynamic Load-Balancer, different types of hardware can take part in the calculation of a *BLonD* simulation, e.g. CPU multi-processors and GPU accelerators. The advanced automatic load-balancing scheme will ensure a fair distribution of the workload according to the processing power of each platform. However, the parallelization scheme can be further enhanced by assigning specific tasks to specific types of hardware, based on the characteristics of the available hardware and tasks. For example, the FFT operations can be solely handled by GPU accelerators, while other more control-intensive operations can be assigned entirely to the CPU multi-processors.

The studies for novel, LOOG-based, Out-Of-Order GPU architectures can be also extended towards the following directions:

- Simulation robustness. By transitioning LOOG to Accel-Sim's [145] upgraded timing model, and trace-based simulation mode, it is possible to

gain access to more real-world workloads and modern GPU platforms with higher modeling accuracy.

- Domain-specific LOOG variations. The simulation framework and the LOOG architecture are highly configurable. Instances of the LOOG architecture that target specific application domains can be designed.

- Micro-architectural optimization for performance and energy gains. Some LOOG components, such as the instruction fetch and decode units can be modified to operate in a pipeline fashion, to reduce the power overhead for a potential increase in execution cycles.

- Run-time aware re-configuration. We have observed that certain workloads are not well-fitted for Out-Of-Order execution, while other workloads profit from more aggressive Out-Of-Order execution and Instruction-Level parallelism. By monitoring dynamic characteristics of the application, it is possible to detect such classes of applications and re-configure the underlying architecture ranging from simple in-order to aggressive Out-of-Order execution.

# Appendix A

# List of Publications and Notable Contributions

**Journal articles:**

- 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

- Konstantinos Iliakis, Sotirios Xydis, and Dimitrios Soudris. "Repurposing GPU Microarchitectures with Light-Weight Out-Of-Order Execution". In: *IEEE Transactions on Parallel and Distributed Systems* 33.2 (2021), pp. 388–402

Submitted for publication:

- H. Timko and S. Albright and T. Argyropoulos and K. Iliakis and I. Karpov and A. Lasheen and D. Quartullo and J. Repond and P. Tsapatsaris and L. Medina and J. E. Müller and M. Schwarz. "Beam Longitudinal Dynamics Simulation Suite BLonD". in: *Physical Review Accelerators and Beams (to be published)* (2021)

- Iliakis, Konstantinos and Helga, Timko and Xydis, Sotirios and Tsapatsaris, Panagiotis and Soudris, Dimitrios. "Enabling Large Scale Simulations For Particle Accelerators". In: *IEEE Transactions on Parallel and Distributed Systems* 0.0 (2022), pp. 1–1

**Conference proceedings:**

- Konstantinos Iliakis and Helga Timko and Sotirios Xydis and Dimitrios Soudris. "BLonD++: performance analysis and optimizations for enabling complex, accurate and fast beam dynamics studies". In: *Proceedings of the 18th International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation, Pythagorion, Greece, July*

*15-19, 2018*. ACM, 2018, pp. 123–130. DOI: 10.1145/3229631.3229640. URL: https://doi.org/10.1145/3229631.3229640

- Iliakis, Konstantinos and Timko, Helga and Xydis, Sotirios and Soudris, Dimitrios. "Scale-out Beam Longitudinal Dynamics Simulations". In: *Proceedings of the 17th ACM International Conference on Computing Frontiers*. CF '20. Catania, Sicily, Italy: Association for Computing Machinery, 2020, 29–38. ISBN: 9781450379564. DOI: 10.1145/3387902.3392616. URL: https://doi.org/10.1145/3387902.3392616

- Repond, Joël and Iliakis, Konstantinos and Schwarz, Markus and Shaposhnikova, Elena. "Simulations of Longitudinal Beam Stabilisation in the CERN SPS With BLonD". in: *Proceedings ICAP2018: Key West, FL, USA*. 2018, TUPAF06

- Schwarz, Markus and Iliakis, Konstantinos and Lasheen, Alexandre and Papotti, Giulia and Repond, Joël and Shaposhnikova, Elena and Timko, Helga. "Flat-Bottom Instabilities in the CERN SPS". in: *10th Int. Partile Accelerator Conf.(IPAC'19), Melbourne, Australia*. JACOW. 2019, pp. 3224–3227

Submitted for publication:

- Iliakis, Konstantinos and Xydis, Sotirios and Soudris, Dimitrios. "Towards Out-Of-Order GPUs: Micro-architectural Design and Exploration". In: *Proceedings of the 19th ACM International Conference on Computing Frontiers*. CF '22. Turin, Piedmont, Italy: Association for Computing Machinery, 2022

**Invited talks & Others**

- PhD Forum, DATE 2020, Efficient Scale-Up and Scale-Out of Beam Longitudinal Dynamics Simulations, March 9-13, 2020, ALP-EXPO, Grenoble, France.

- Intel Extreme Performance Users Group (IXPUG), Hybrid-BLonD: Efficient Scale-out of Beam Longitudinal Dynamics Simulations, September 24-27, 2019, Globe of Science and Innovation at CERN, Geneva, Switzerland.

- CERN Beam Department Seminars, High Performance Computing Cookbook: The BLonD recipe, May 29, 2020, CERN, Geneva, Switzerland.

Publications, not related to main PhD thesis topic:

- Konstantinos Iliakis, Sotirios Xydis, and Dimitrios Soudris. "Decoupled MapReduce for Shared-Memory Multi-Core Architectures". In: *IEEE Computer Architecture Letters* 17.2 (2018), pp. 143–146. DOI: 10.1109/LCA.2018.2827929

- Konstantinos Iliakis, Sotirios Xydis, and Dimitrios Soudris. "Resource-Aware MapReduce Runtime for Multi/Many-core Architectures". In: *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2020, pp. 897–902. DOI: 10.23919/DATE48585.2020.9116281

- Iliakis, Konstantinos and Koliogeorgi, Konstantina and Litke, Antonios and Varvarigou, Theodora and Soudris, Dimitrios. "GPU Accelerated Blockchain over Key-Value Database Transactions". In: *IET Blockchain* 1 (2022), p. 30

# Appendix B

# Source Code

## B.1 BLonD Source code repository

The source code of the BLonD code can be found at:
https://github.com/blond-admin/BLonD.

### B.1.1 License

BLonD is distributed with GNU General Public License v3.0. The complete license text can be found at:
https://github.com/blond-admin/BLonD/blob/master/LICENSE.txt

## B.2 LOOG Source code repository

The source code of the LOOG code can be found at:
https://github.com/kiliakis/gpgpu-sim.

### B.2.1 License

LOOG is distributed with BSD 2-Clause "Simplified" License. The complete license text can be found at:
https://github.com/kiliakis/gpgpu-sim/blob/master/LICENSE.txt

# References


[1] Probir K Bondyopadhyay. "Moore's law governs the silicon revolution". In: *Proceedings of the IEEE* 86.1 (1998), pp. 78–81.

[2] *Microprocessor Trend Data*. 2021. URL: https://github.com/karlrupp/microprocessor-trend-data (visited on 11/30/2021).

[3] Joel S Emer and Douglas W Clark. "A Characterization of Processor Performance in the VAX-11/780". In: *ACM SIGARCH Computer Architecture News* 12.3 (1984), pp. 301–310.

[4] John L Henning. "SPEC CPU2000: Measuring CPU performance in the new millennium". In: *Computer* 33.7 (2000), pp. 28–35.

[5] John L Henning. "SPEC CPU2006 benchmark descriptions". In: *ACM SIGARCH Computer Architecture News* 34.4 (2006), pp. 1–17.

[6] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. "SPEC CPU2017: Next-generation compute benchmark". In: *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. 2018, pp. 41–42.

[7] John L Hennessy and David A Patterson. "Computer architecture: a quantitative approach". In: 6th. Elsevier, 2017. Chap. 4, pp. 310–336.

[8] *Dennard Scaling*. 2021. URL: https://en.wikipedia.org/wiki/Dennard_scaling (visited on 11/30/2021).

[9] Gene M Amdahl. "Validity of the single processor approach to achieving large scale computing capabilities". In: *Proceedings of the April 18-20, 1967, spring joint computer conference*. ACM. 1967, 483–z485.

[10] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. "Numba: A llvm-based python jit compiler". In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. 2015, pp. 1–6.

[11] Mark Turner, David Budgen, and Pearl Brereton. "Turning software into a service". In: *Computer* 36.10 (2003), pp. 38–44.

[12] Leonardo Dagum and Ramesh Menon. "OpenMP: an industry standard API for shared-memory programming". In: *IEEE computational science and engineering* 5.1 (1998), pp. 46–55.

[13] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov, Ahmed Fasih, AD Sarma, D Nanongkai, G Pandurangan, P Tetali, et al. "PyCUDA: GPU run-time code generation for high performance computing". In: *Arxiv preprint arXiv* 911 (2009).

[14] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. "Pytorch: An imperative style, high-performance deep learning library". In: *arXiv preprint arXiv:1912.01703* (2019).

[15] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. "Tensorflow: A system for large-scale machine learning". In: *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*. 2016, pp. 265–283.

[16] Mark James Abraham, Teemu Murtola, Roland Schulz, Szilárd Páll, Jeremy C Smith, Berk Hess, and Erik Lindahl. "GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers". In: *SoftwareX* 1 (2015), pp. 19–25.

[17] Tayler H Hetherington, Mike O'Connor, and Tor M Aamodt. "Memcachedgpu: Scaling-up scale-out key-value stores". In: *Proceedings of the Sixth ACM Symposium on Cloud Computing*. 2015, pp. 43–57.

[18] Kai Zhang, Kaibo Wang, Yuan Yuan, Lei Guo, Rubao Lee, and Xiaodong Zhang. "Mega-kv: A case for gpus to maximize the throughput of in-memory key-value stores". In: *Proceedings of the VLDB Endowment* 8.11 (2015), pp. 1226–1237.

[19] François Englert and Robert Brout. "Broken symmetry and the mass of gauge vector mesons". In: *Physical Review Letters* 13.9 (1964), p. 321.

[20] R Brout and François Englert. "Spontaneous symmetry breaking in gauge theories: A Historical survey". In: *arXiv preprint hep-th/9802142* (1998).

[21] Peter W Higgs. "Broken symmetries and the masses of gauge bosons". In: *Physical Review Letters* 13.16 (1964), p. 508.

[22] G. S. Guralnik, C. R. Hagen, and T. W. B. Kibble. "Global Conservation Laws and Massless Particles". In: *Phys. Rev. Lett.* 13 (20 Nov. 1964), pp. 585–587. DOI: 10.1103/PhysRevLett.13.585. URL: https://link.aps.org/doi/10.1103/PhysRevLett.13.585.

[23] Gerald S Guralnik. "The history of the Guralnik, Hagen and Kibble development of the theory of spontaneous symmetry breaking and gauge particles". In: *International Journal of Modern Physics A* 24.14 (2009), pp. 2601–2627.

[24] Georges Aad, JM Butterworth, J Thion, U Bratzler, PN Ratoff, RB Nickerson, JM Seixas, I Grabowska-Bold, F Meisel, S Lokwitz, et al. "The ATLAS experiment at the CERN large hadron collider". In: *Jinst* 3 (2008), S08003.

[25] CMS Collaboration et al. *The CMS experiment at the CERN LHC*. 2008.

[26] *European Strategy for Particle Physics*. 2021. URL: https://europeanstrategy.cern/european-strategy-for-particle-physics (visited on 09/03/2021).

[27] Richard Keith Ellis et al. *Physics Briefing Book: Input for the European Strategy for Particle Physics Update 2020*. Tech. rep. arXiv:1910.11775. 254 p. Geneva, Oct. 2019. URL: http://cds.cern.ch/record/2691414.

[28] H Damerau, A Funken, R Garoby, S Gilardoni, B Goddard, K Hanke, A Lombardi, D Manglunki, M Meddahi, B Mikulec, G Rumolo, E Shaposhnikova, M Vretenar, and J Coupard. *LHC Injectors Upgrade, Technical Design Report, Vol. I: Protons*. Tech. rep. CERN-ACC-2014-0337. CERN, Dec. 2014. URL: http://cds.cern.ch/record/1976692.

[29] Apollinari G., Bejar Alonso I., Bruning O., Fessia P., Lamont M., Rossi L., and Tavian L. *High-Luminosity Large Hadron Collider (HL-LHC): Technical Design Report V. 0.1*. CERN Yellow Reports: Monographs. Geneva: CERN, 2017. URL: http://cds.cern.ch/record/2284929.

[30] A. Ball, M. Benedikt, L. Bottura, O. Dominguez, F. Gianotti, B. Goddard, P. Lebrun, M. Mangano, D. Schulte, E. Shaposhnikova, R. Tomas, and F. Zimmermann. "The Future Circular Collider study". In: *CERN Courier* 54.3 (Apr. 2014), pp. 16–18. URL: http://cds.cern.ch/record/2064538.

[31] FCC collaboration et al. "FCC Physics Opportunities: Future Circular Collider Conceptual Design Report Volume 1". In: *European Physical Journal C* 79.6 (2019), p. 474.

[32] Frank Zimmermann, M Benedikt, M Capeans Garrido, F Cerutti, B Goddard, J Gutleber, JM Jimenez, M Mangano, V Mertens, JA Osborne, et al. *Future circular collider*. Tech. rep. CERN-ACC-2018-0059, 2018.

[33] John Ellis and Ian Wilson. "New physics with the compact linear collider". In: *Nature* 409.6818 (2001), pp. 431–435.

[34] R. Tomás. "Overview of the Compact Linear Collider". In: *Phys. Rev. ST Accel. Beams* 13 (1 Jan. 2010), p. 014801. DOI: 10.1103/PhysRevSTAB.13.014801. URL: https://link.aps.org/doi/10.1103/PhysRevSTAB.13.014801.

[35] Simon Albright, Konstantinos Iliakis, Alexandre Lasheen, Danilo Quartullo, Joel Repond, and Helga Timko. *CERN Beam Longitudinal Dynamics code BLonD*. 2014. URL: https://blond.web.cern.ch/ (visited on 03/02/2018).

[36] H. Timko and S. Albright and T. Argyropoulos and K. Iliakis and I. Karpov and A. Lasheen and D. Quartullo and J. Repond and P. Tsapatsaris and L. Medina and J. E. Müller and M. Schwarz. "Beam Longitudinal Dynamics Simulation Suite BLonD". In: *Physical Review Accelerators and Beams (to be published)* (2021).

[37] Elena Shaposhnikova, Joël Repond, Helga Timko, Theodoros Argyropoulos, Thomas Bohl, and Alexandre Lasheen. "Identification and Reduction of the CERN SPS Impedance". In: *Proceedings of the 57th ICFA Advanced Beam Dynamics Workshop on High-Intensity, High Brightness and High Power Hadron Beams, HB2016*. 2016.

[38] Vincenzo Forte, Elena Benedetto, Alessandra Lombardi, and Danilo Quartullo. "Longitudinal Injection Schemes For the CERN PS Booster at 160 MeV Including Space Charge Effects". In: *Proceedings, 6th International Particle Accelerator Conference (IPAC 2015): Richmond, Virginia, USA, May 3-8, 2015*. 2015, MOPJE042. DOI: 10.18429/JACoW-IPAC2015-MOPJE042. URL: http://accelconf.web.cern.ch/AccelConf/IPAC2015/papers/mopje042.pdf.

[39] Danilo Quartullo, Simon Albright, Elena Shaposhnikova, et al. "Studies of Longitudinal Beam Stability in CERN PS Booster After Upgrade". In: *8th Int. Particle Accelerator Conf.(IPAC'17), Copenhagen, Denmark, 14â 19 May, 2017*. JACOW, Geneva, Switzerland. 2017, pp. 4469–4472.

[40] Danilo Quartullo, Elena Shaposhnikova, and Helga Timko. "Controlled longitudinal emittance blow-up using band-limited phase noise in CERN PSB". In: *Journal of Physics: Conference Series*. Vol. 874. 1. IOP Publishing. 2017, p. 012066.

[41] Alexandre Lasheen, Edgaras Radvilas, Elena Shaposhnikova, Toon Roggen, Thomas Bohl, and Steven Hancock. "Single bunch longitudinal instability in the CERN SPS". In: *7th International Particle Accelerator Conference, Busan, Korea, 8 - 13 May 2016*. 2016.

[42] Repond, Joël and Iliakis, Konstantinos and Schwarz, Markus and Shaposhnikova, Elena. "Simulations of Longitudinal Beam Stabilisation in the CERN SPS With BLonD". In: *Proceedings ICAP2018: Key West, FL, USA*. 2018, TUPAF06.

[43] Schwarz, Markus and Iliakis, Konstantinos and Lasheen, Alexandre and Papotti, Giulia and Repond, Joël and Shaposhnikova, Elena and Timko, Helga. "Flat-Bottom Instabilities in the CERN SPS". In: *10th Int. Partile Accelerator Conf.(IPAC'19), Melbourne, Australia*. JACOW. 2019, pp. 3224–3227.

[44] E Métral, T Argyropoulos, H Bartosik, N Biancacci, Xavier Buffat, JF Esteban Muller, W Herr, G Iadarola, A Lasheen, K Li, et al. "Beam instabilities in hadron synchrotrons". In: *IEEE Transactions on Nuclear Science* 63.2 (2016), pp. 1001–1050.

[45] Helga Timko, Danilo Quartullo, Alexandre Lasheen, and Juan Esteban Müller. "Benchmarking the beam longitudinal dynamics code BLonD". In: *Proceedings of the 7th International Particle Accelerator Conference (IPAC 2016): Busan, Korea*. 2016.

[46] Christopher J Hughes. "Single-instruction multiple-data execution". In: *Synthesis Lectures on Computer Architecture* 10.1 (2015), pp. 1–121.

[47] Ahmad Yasin. "A top-down method for performance analysis and counters architecture". In: *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*. IEEE. 2014, pp. 35–44.

[48] Samuel Williams, Andrew Waterman, and David Patterson. "Roofline: an insightful visual performance model for multicore architectures". In: *Communications of the ACM* 52.4 (2009), pp. 65–76.

[49] Aleksandar Ilic, Frederico Pratas, and Leonel Sousa. "Cache-aware roofline model: Upgrading the loft". In: *IEEE Computer Architecture Letters* 13.1 (2013), pp. 21–24.

[50] S. Y. Lee. *Accelerator physics*. Third. Singapore: World Scientific, 2012. URL: https://cds.cern.ch/record/1425444.

[51] Charles K. Birdsall and A. Bruce Langdon. *Plasma physics via computer simulation*. New York: Taylor and Francis, 2005. ISBN: 0750310251 9780750310253.

[52] JA MacLachlan. *ESME: Longitudinal Phase Space Particle Tracking-Program Documentation*. Tech. rep. TM-1274. Fermi National Accelerator Lab., Batavia, IL (United States), 1984.

[53] JA MacLachlan. *Particle tracking in E-$\phi$ space for synchrotron design and diagnosis*. Tech. rep. Fermi National Accelerator Lab., Batavia, IL (United States), 1992.

[54] Andrei Shishlo, S Cousineau, V Danilov, J Galambos, S Henderson, J Holmes, and M Plum. "The ORBIT simulation code: benchmarking and applications". In: *Proceedings of ICAP*. 2006.

[55] Andrei Shishlo, Sarah Cousineau, Jeffrey Holmes, and Timofey Gorlov. "The particle accelerator simulation code PyORBIT". In: *Procedia Computer Science* 51 (2015), pp. 1272–1281.

[56] Michael Borland. *Elegant: A flexible SDDS-compliant code for accelerator simulation*. Tech. rep. Argonne National Lab., IL (US), 2000.

[57] Yusong Wang and Michael Borland. "PELEGANT: A parallel accelerator simulation code for electron generation and tracking". In: *AIP Conference Proceedings*. Vol. 877. AIP. 2006, pp. 241–247.

[58] Konstantinos Iliakis and Helga Timko and Sotirios Xydis and Dimitrios Soudris. "BLonD++: performance analysis and optimizations for enabling complex, accurate and fast beam dynamics studies". In: *Proceedings of the 18th International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation, Pythagorion, Greece, July 15-19, 2018*. ACM, 2018, pp. 123–130. DOI: 10.1145/3229631.3229640. URL: https://doi.org/10.1145/3229631.3229640.

[59] Per Hammarlund, Alberto J Martinez, Atiq A Bajwa, David L Hill, Erik Hallnor, Hong Jiang, Martin Dixon, Michael Derr, Mikal Hunsaker, Rajesh Kumar, et al. "Haswell: The fourth-generation intel core processor". In: *IEEE Micro* 34.2 (2014), pp. 6–20.

[60] Eric Jones, Travis Oliphant, Pearu Peterson, et al. *SciPy: Open source scientific tools for Python*. [Online; accessed 2018-28-05]. 2001–. URL: http://www.scipy.org/.

[61] Stéfan van der Walt, S Chris Colbert, and Gael Varoquaux. "The NumPy array: a structure for efficient numerical computation". In: *Computing in Science & Engineering* 13.2 (2011), pp. 22–30.

[62] Boris Schäling. *The boost C++ libraries*. Boris Schäling, 2011.

[63] *Intel® VTune™ Amplifier 2017*. 2017. URL: https://software.intel.com/en-us/intel-vtune-amplifier-xe (visited on 01/19/2017).

[64] Arnaldo Carvalho De Melo. "The new linux'perf'tools". In: *Slides from Linux Kongress*. Vol. 18. 2010, pp. 1–42.

[65] *ITT API Open Source*. 2018. URL: https://software.intel.com/en-us/articles/intel-itt-api-open-source (visited on 03/04/2018).

[66] Nicolai M Josuttis. *The C++ standard library: a tutorial and reference*. Addison-Wesley, 2012.

[67] Danilo Piparo, Vincenzo Innocente, and Thomas Hauth. "Speeding up HEP experiment software with a library of fast and auto-vectorisable mathematical functions". In: *Journal of Physics: Conference Series* (June 2014).

[68] *Intel Math Kernel Library*. 2018. URL: https://software.intel.com/en-us/mkl (visited on 03/04/2018).

[69] Matteo Frigo and Steven G Johnson. "FFTW: An adaptive software architecture for the FFT". In: *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*. Vol. 3. IEEE. 1998, pp. 1381–1384.

[70] William Gropp. "MPICH2: A new start for MPI implementations". In: *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer. 2002, pp. 7–7.

[71] Helga Timko, Elena Shaposhnikova, Philippe Baudrenghien, and Themis Mastoridis. "Studies on Controlled RF Noise for the LHC". In: *Proceedings of the 54th ICFA Advanced Beam Dynamics Workshop on High-Intensity, High Brightness and High Power Hadron Beams, East-Lansing, USA, 10 - 14 Nov 2014*. 2014.

[72] *Minimize Cash Flow Among a Given Set*. 2022. URL: https://www.geeksforgeeks.org/minimize-cash-flow-among-given-set-friends-borrowed-money/ (visited on 04/02/2022).

[73] Ankireddy Nalamalpu, Nasser Kurd, Anant Deval, Chris Mozak, Jonathan Douglas, Ashish Khanna, Fabrice Paillet, Gerhard Schrom, and Boyd Phelps. "Broadwell: A family of IA 14nm processors". In: *2015 Symposium on VLSI Circuits (VLSI Circuits)*. IEEE. 2015, pp. C314–C315.

[74] Gregory F Pfister. "An introduction to the infiniband architecture". In: *High Performance Mass Storage and Parallel I/O* 42 (2001), pp. 617–632.

[75] Wei Huang, Gopalakrishnan Santhanaraman, H-W Jin, Qi Gao, and Dhabaleswar K Panda. "Design of high performance MVAPICH2: MPI2 over InfiniBand". In: *Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06)*. Vol. 1. IEEE. 2006, pp. 43–48.

[76] Edgar Gabriel, Graham E Fagg, George Bosilca, Thara Angskun, Jack J Dongarra, Jeffrey M Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, et al. "Open MPI: Goals, concept, and design of a next generation MPI implementation". In: *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer. 2004, pp. 97–104.

[77] Rolf Rabenseifner, Georg Hager, and Gabriele Jost. "Hybrid MPI/ OpenMP parallel programming on clusters of multi-core SMP nodes". In: *2009 17th Euromicro international conference on parallel, distributed and network based processing*. IEEE. 2009, pp. 427–436.

[78] Christoph Lameter et al. "NUMA (Non-Uniform Memory Access): An Overview." In: *Acm queue* 11.7 (2013), p. 40.

[79] Yinan Li, Ippokratis Pandis, Rene Mueller, Vijayshankar Raman, and Guy M Lohman. "NUMA-aware algorithms: the case of data shuffling." In: *CIDR*. 2013.

[80] Sergey Blagodurov, Alexandra Fedorova, Sergey Zhuravlev, and Ali Kamali. "A case for NUMA-aware contention management on multicore systems". In: *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE. 2010, pp. 557–558.

[81] David B Kirk and W Hwu Wen-Mei. *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann, 2016.

[82] John E Stone, David Gohara, and Guochun Shi. "OpenCL: A parallel programming standard for heterogeneous computing systems". In: *Computing in science & engineering* 12.3 (2010), p. 66.

[83] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. "cudnn: Efficient primitives for deep learning". In: *arXiv preprint arXiv:1410.0759* (2014).

[84] *CUDA C/C++ SDK Code Samples*. 2011. URL: https://docs.nvidia.com/cuda/cuda-samples/index.html.

[85] *Southern Islands Series Instruction Set Architecture*. Dec. 2012. URL: https://developer.amd.com/wordpress/media/2012/12/AMD_Southern_Islands_Instruction_Set_Architecture.pdf.

[86] Martin Burtscher, Rupesh Nasre, and Keshav Pingali. "A quantitative study of irregular programs on GPUs". In: *2012 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE. 2012, pp. 141–151.

[87] Jee Ho Ryoo, Saddam J Quirem, Michael Lebeane, Reena Panda, Shuang Song, and Lizy K John. "GPGPU benchmark suites: How well do they sample the performance spectrum?" In: *2015 44th International Conference on Parallel Processing*. IEEE. 2015, pp. 320–329.

[88] NVIDIA Corporation. *NVIDIA GeForce GTX 1080*. 2016. URL: https://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_1080_Whitepaper_FINAL.pdf.

[89] NVIDIA Corporation. *GP100 Pascal Whitepaper*. 2016. URL: https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf.

[90] NVIDIA Corporation. *NVIDIA Tesla V100 GPU Architecture*. 2018. URL: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[91] Sparsh Mittal. "A survey of techniques for architecting and managing GPU register file". In: *IEEE Transactions on Parallel and Distributed Systems* 28.1 (2016), pp. 16–28.

[92] Farzad Khorasani, Hodjat Asghari Esfeden, Amin Farmahini-Farahani, Nuwan Jayasena, and Vivek Sarkar. "Regmutex: Inter-warp gpu register time-sharing". In: *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2018, pp. 816–828.

[93] John Kloosterman, Jonathan Beaumont, D. Anoushe Jamshidi, Jonathan Bailey, Trevor Mudge, and Scott Mahlke. "Regless: Just-in-Time Operand Staging for GPUs". In: *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. New York, NY, USA: Association for Computing Machinery, 2017, 151–164. URL: https://doi.org/10.1145/3123939.3123974.

[94] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P Scarpazza. "Dissecting the nvidia volta gpu architecture via microbenchmarking". In: *arXiv preprint arXiv:1804.06826* (2018).

[95] Zhe Jia, Marco Maggioni, Jeffrey Smith, and Daniele Paolo Scarpazza. "Dissecting the NVidia Turing T4 GPU via Microbenchmarking". In: *arXiv preprint arXiv:1903.07486* (2019).

[96] Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. "Demystifying GPU microarchitecture through microbenchmarking". In: *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*. IEEE. 2010, pp. 235–246.

[97] Timothy G Rogers, Mike O'Connor, and Tor M Aamodt. "Cache-Conscious Wavefront Scheduling". In: *MICRO*. IEEE. 2012, pp. 72–83.

[98] Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N Patt. "Improving GPU performance via large warps and two-level warp scheduling". In: *MICRO*. ACM. 2011.

[99] Onur Kayıran, Adwait Jog, Mahmut Taylan Kandemir, and Chita Ranjan Das. "Neither more nor less: optimizing thread-level parallelism for GPGPUs". In: *PACT*. IEEE. 2013, pp. 157–166.

[100] Adwait Jog, Onur Kayiran, Nachiappan Chidambaram Nachiappan, Asit K Mishra, Mahmut T Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R Das. "OWL: cooperative thread array aware scheduling techniques for improving GPGPU performance". In: *SIGPLAN*. ACM. 2013, pp. 395–406.

[101] Andrew Lavin and Scott Gray. "Fast algorithms for convolutional neural networks". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2016, pp. 4013–4021.

[102] Karen Simonyan and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition". In: *arXiv preprint 1409.1556* (2014).

[103] Hai Jiang, Yi Chen, Zhi Qiao, Tien-Hsiung Weng, and Kuan-Ching Li. "Scaling up MapReduce-based big data processing on multi-GPU systems". In: *Cluster Computing* 18.1 (2015), pp. 369–383.

[104] Akhil Arunkumar, Evgeny Bolotin, David Nellans, and Carole-Jean Wu. "Understanding the future of energy efficiency in multi-module GPUs". In: *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2019, pp. 519–532.

[105] Ugljesa Milic, Oreste Villa, Evgeny Bolotin, Akhil Arunkumar, Eiman Ebrahimi, Aamer Jaleel, Alex Ramirez, and David Nellans. "Beyond the socket: NUMA-aware GPUs". In: *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. 2017, pp. 123–135.

[106] Akhil Arunkumar, Evgeny Bolotin, Benjamin Cho, Ugljesa Milic, Eiman Ebrahimi, Oreste Villa, Aamer Jaleel, Carole-Jean Wu, and David Nellans. "MCM-GPU: Multi-chip-module GPUs for continued performance scalability". In: *ACM SIGARCH Computer Architecture News* 45.2 (2017), pp. 320–332.

[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] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. "Rodinia: A benchmark suite for heterogeneous computing". In: *IISWC*. IEEE. 2009, pp. 44–54.

[109] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. "Parboil: A revised benchmark suite for scientific and commercial throughput computing". In: *Center for Reliable and High-Performance Computing* (2012).

[110] Ali Bakhoda, George L Yuan, Wilson WL Fung, Henry Wong, and Tor M Aamodt. "Analyzing CUDA workloads using a detailed GPU simulator". In: *ISPASS*. IEEE. 2009, pp. 163–174.

[111] Myung Kuk Yoon, Keunsoo Kim, Sangpil Lee, Won Woo Ro, and Murali Annavaram. "Virtual thread: Maximizing thread-level parallelism beyond GPU scheduling limit". In: *ISCA*. IEEE. 2016, pp. 609–621.

[112] Dong Li, Minsoo Rhu, Daniel R Johnson, Mike O'Connor, Mattan Erez, Doug Burger, Donald S Fussell, and Stephen W Redder. "Priority-based cache allocation in throughput processors". In: *HPCA*. IEEE. 2015, pp. 89–100.

[113] Keunsoo Kim, Sangpil Lee, Myung Kuk Yoon, Gunjae Koo, Won Woo Ro, and Murali Annavaram. "Warped-preexecution: A GPU pre-execution approach for improving latency hiding". In: *HPCA*. IEEE. 2016, pp. 163–175.

[114] Xun Gong, Xiang Gong, Leiming Yu, and David Kaeli. "HAWS: Accelerating GPU Wavefront Execution through Selective Out-of-order Execution". In: *ACM TACO* 16.2 (2019), p. 15.

[115] Mohammad Abdel-Majeed and Murali Annavaram. "Warped register file: A power efficient register file for GPGPUs". In: *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2013, pp. 412–423.

[116] Mark Gebhart, Daniel R Johnson, David Tarjan, Stephen W Keckler, William J Dally, Erik Lindholm, and Kevin Skadron. "Energy-efficient mechanisms for managing thread context in throughput processors". In: *2011 38th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2011, pp. 235–246.

[117] Xiang Gong, Zhongliang Chen, Amir Kavyan Ziabari, Rafael Ubal, and David Kaeli. "TwinKernels: An Execution Model to Improve GPU Hardware Scheduling at Compile Time". In: *CGO*. IEEE Press, 2017, pp. 39–49.

[118] Thomas Bradley. "Hyper-Q example". In: *NVidia Corporation. Whitepaper v1. 0* (2012).

[119] Ankit Sethia, D Anoushe Jamshidi, and Scott Mahlke. "Mascar: Speeding up GPU warps by reducing memory pitstops". In: *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2015, pp. 174–185.

[120] Jayvant Anantpur, Nagendra Gulur Dwarakanath, Shivaram Kalyanakrishnan, Shalabh Bhatnagar, and R Govindarajan. "RLWS: A Reinforcement Learning based GPU Warp Scheduler". In: *arXiv preprint 1712.04303* (2017).

[121] Minseok Lee, Seokwoo Song, Joosik Moon, John Kim, Woong Seo, Yeongon Cho, and Soojung Ryu. "Improving GPGPU resource utilization through alternative thread block scheduling". In: *HPCA*. IEEE. 2014, pp. 260–271.

[122] Saumay Dublish, Vijay Nagarajan, and Nigel Topham. "Poise: Balancing Thread-Level Parallelism and Memory System Performance in GPUs using Machine Learning". In: *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2019, pp. 492–505.

[123] Tor M Aamodt, Wilson Wai Lun Fung, and Timothy G Rogers. "General-purpose graphics processor architectures". In: *Synthesis Lectures on Computer Architecture* 13.2 (2018), pp. 1–140.

[124] NVIDIA Corporation. *PTX: Parallel thread execution ISA version 5.0*. 2017. URL: https://docs.nvidia.com/pdf/ptx_isa_5.0.pdf.

[125] Wilson WL Fung, Ivan Sham, George Yuan, and Tor M Aamodt. "Dynamic warp formation: Efficient MIMD control flow on SIMD graphics hardware". In: *ACM Transactions on Architecture and Code Optimization (TACO)* 6.2 (2009), p. 7.

[126] 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.

[127] Yunho Oh, Myung Kuk Yoon, William J Song, and Won Woo Ro. "FineReg: fine-grained register file management for augmenting GPU throughput". In: *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2018, pp. 364–376.

[128] Ahmed ElTantawy, Jessica Wenjie Ma, Mike O'Connor, and Tor M Aamodt. "A scalable multi-path microarchitecture for efficient GPU control flow". In: *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2014, pp. 248–259.

[129] James E Thornton. "The CDC 6600 project". In: *Annals of the History of Computing* 2.4 (1980), pp. 338–348.

[130] Gregory Diamos, Benjamin Ashbaugh, Subramaniam Maiyuran, Andrew Kerr, Haicheng Wu, and Sudhakar Yalamanchili. "SIMD re-convergence at thread frontiers". In: *2011 44th Annual IEEE ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2011, pp. 477–488.

[131] Brett W Coon, John Erik Lindholm, Gary Tarolli, Svetoslav D Tzvetkov, John R Nickolls, and Ming Y Siu. *Register file allocation*. US Patent 7,634,621. Dec. 2009.

[132] John Erik Lindholm, Ming Y Siu, Simon S Moy, Samuel Liu, and John R Nickolls. *Simulating multiported memories using lower port count memories*. US Patent 7,339,592. Mar. 2008.

[133] Samuel Liu, John Erik Lindholm, Ming Y Siu, Brett W Coon, and Stuart F Oberman. *Operand collector architecture*. US Patent 7,834,881. Nov. 2010.

[134] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M Aamodt, and Vijay Janapa Reddi. "GPUWattch: enabling energy optimizations in GPGPUs". In: *SIGARCH*. ACM. 2013.

[135] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi. "McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures". In: *MICRO*. ACM. 2009, pp. 469–480.

[136] Shyamkumar Thoziyoor, Jung Ho Ahn, Matteo Monchiero, Jay B Brockman, and Norman P Jouppi. "A comprehensive memory modeling tool and its application to the design and analysis of future memory hierarchies". In: *ACM SIGARCH Computer Architecture News* 36.3 (2008), pp. 51–62.

[137] Jaikrishnan Menon, Marc De Kruijf, and Karthikeyan Sankaralingam. "iGPU: exception support and speculative execution on GPUs". In: *ACM SIGARCH Computer Architecture News*. Vol. 40. IEEE Computer Society. 2012, pp. 72–83.

[138] James E Smith. "A study of branch prediction strategies". In: *25 years of the international symposia on Computer architecture (selected papers)*. 1998, pp. 202–215.

[139] Dezso Sima. "The design space of register renaming techniques". In: *IEEE micro* 20.5 (2000), pp. 70–83.

[140] Michael Mishkin, Nam Sung Kim, and Mikko Lipasti. "Write-after-Read Hazard Prevention in GPGPUsim". In: *Workshop on Deplicating, Deconstructing, and Debunking (WDDD)*. June 2016.

[141] Shuai Che, Jeremy W Sheaffer, Michael Boyer, Lukasz G Szafaryn, Liang Wang, and Kevin Skadron. "A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads". In: *International Symposium on Workload Characterization (IISWC'10)*. IEEE. 2010, pp. 1–11.

[142] Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N Patt. "Runahead execution: An alternative to very large instruction windows for out-of-order processors". In: *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.* IEEE. 2003, pp. 129–140.

[143] 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.

[144] Konstantinos Iliakis, Sotirios Xydis, and Dimitrios Soudris. "Repurposing GPU Microarchitectures with Light-Weight Out-Of-Order Execution". In: *IEEE Transactions on Parallel and Distributed Systems* 33.2 (2021), pp. 388–402.

[145] Mahmoud Khairy, Zhesheng Shen, Tor M Aamodt, and Timothy G Rogers. "Accel-Sim: An Extensible Simulation Frameworkfor Validated GPU Modeling". In: *2020 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2020.

[146] Iliakis, Konstantinos and Helga, Timko and Xydis, Sotirios and Tsapatsaris, Panagiotis and Soudris, Dimitrios. "Enabling Large Scale Simulations For Particle Accelerators". In: *IEEE Transactions on Parallel and Distributed Systems* 0.0 (2022), pp. 1–1.

[147] Iliakis, Konstantinos and Timko, Helga and Xydis, Sotirios and Soudris, Dimitrios. "Scale-out Beam Longitudinal Dynamics Simulations". In: *Proceedings of the 17th ACM International Conference on Computing Frontiers*. CF '20. Catania, Sicily, Italy: Association for Computing Machinery, 2020, 29–38. ISBN: 9781450379564. DOI: 10.1145/3387902.3392616. URL: https://doi.org/10.1145/3387902.3392616.

[148] Iliakis, Konstantinos and Xydis, Sotirios and Soudris, Dimitrios. "Towards Out-Of-Order GPUs: Micro-architectural Design and Exploration". In: *Proceedings of the 19th ACM International Conference on Computing Frontiers*. CF '22. Turin, Piedmont, Italy: Association for Computing Machinery, 2022.

[149] Konstantinos Iliakis, Sotirios Xydis, and Dimitrios Soudris. "Decoupled MapReduce for Shared-Memory Multi-Core Architectures". In: *IEEE Computer Architecture Letters* 17.2 (2018), pp. 143–146. DOI: 10.1109/LCA.2018.2827929.

[150] Konstantinos Iliakis, Sotirios Xydis, and Dimitrios Soudris. "Resource-Aware MapReduce Runtime for Multi/Many-core Architectures". In: *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2020, pp. 897–902. DOI: 10.23919/DATE48585.2020.9116281.

[151] Iliakis, Konstantinos and Koliogeorgi, Konstantina and Litke, Antonios and Varvarigou, Theodora and Soudris, Dimitrios. "GPU Accelerated Blockchain over Key-Value Database Transactions". In: *IET Blockchain* 1 (2022), p. 30.