Università degli Studi di Salerno

Dipartimento di Informatica

Dottorato di Ricerca in Informatica – XXXVIII Ciclo

Tesi di Dottorato/Ph.D. Thesis

Abstractions for Approximate and Energy-Efficient Computing on Modern SIMD/SIMT Architectures

Lorenzo Carpentieri

Supervisor: **Prof. Biagio Cosenza**

Ph.D. Program Director: **Prof. Andrea De Lucia**

AA 2024/2025

Curriculum Internet of Things and Smart Technologies

Università degli Studi di Salerno

Dipartimento di Informatica

Dottorato di Ricerca in Informatica

Curriculum Internet of Things and Smart Technologies

XXXVIII Ciclo

TESI DI DOTTORATO / PH.D. THESIS

Abstractions for Approximate and Energy-Efficient Computing on Modern SIMD/SIMT Architectures

LORENZO CARPENTIERI

SUPERVISOR: **PROF. BIAGIO COSENZA**

PHD PROGRAM DIRECTOR: **PROF. ANDREA DE LUCIA**

A.A 2024/2025

In loving memory of my grandmother, Mafalda.

ACKNOWLEDGMENTS

First and foremost, I extend my deepest gratitude to my supervisor, Prof. Biagio Cosenza, for his enduring support, patience, and understanding. His mentorship was not only professional but also personal, guiding me through my PhD journey with the kindness of a friend. My sincere thanks to Prof. Sohan Lal from TU Hamburg for his invaluable assistance, both academically and personally, during my time with his group. His support greatly enriched my visiting period in Hamburg. I appreciate Prof. Andrea De Lucia, our PhD coordinator, for his excellent organization throughout these years. A big thank you to all professors and members of the ISIS Lab who made every day in the lab a special experience. I am fortunate to have worked alongside my colleagues and friends at the University of Salerno. Their collaboration, help, and kindness have been a cornerstone of my academic life. I would also like to thank all my friends in Cava De' Tirreni, who made this PhD journey far less stressful and reminded me that there is a world beyond research deadlines and academic responsibilities. Your friendship has been a constant source of balance and joy. A very special thank you goes to my girlfriend, Paola. I must admit that, for her, I have probably been a bit of a "scam." We met at a time when I had no imminent conference deadlines, and shortly after we got together, she found herself trapped in a life of conference deadlines, projects, career decisions, and constant uncertainty about the future. For this reason, I want to thank you for your patience, but above all for your strength, for always trying to pull me out, even if just for a moment, from a life made only of HPC and papers. Most importantly, I owe my deepest gratitude to my mother, Trofimena, and my father, Antonio. I will never be able to thank you enough for everything you have done for me, not only during these past years, but throughout my entire life. You have both been the light in the darkness, the ones I have always known want the best for me. My mother has always encouraged me to chase my dreams and aim high, and it is also thanks to you that I decided to start

a PhD and a career in research. At the same time, I must thank my father, whose pragmatism taught me that, at a certain point, one must also return with both feet on the ground. From you, I learned the importance of balance between ambition and reality. I would also like to thank my brother, Federico. We have always shared a wonderful relationship, and even if we do not speak often, I always know that I can count on him. That certainty has been invaluable. Each of you has played an integral part in my journey, and for that, I am forever grateful.

ABSTRACT

The limits imposed by power, memory, and instruction-level parallelism walls have driven the computing industry toward heterogeneous architectures, which combine general-purpose cores with specialized accelerators such as GPUs. These architectures promise higher performance and energy efficiency, but they also introduce significant programming challenges, since efficiently exploiting diverse hardware requires developers to navigate complex vendor-specific APIs and parallel programming paradigms. At the same time, emerging workloads, ranging from AI inference to large scale scientific simulations, demand approximate and energy-efficient computing techniques to meet both performance and sustainability goals. In response, a rich ecosystem of programming models has emerged, spanning low-level, hardware-specific APIs that expose fine-grained control, to high-level abstractions that aim to improve developer productivity while maintaining efficient hardware utilization across heterogeneous systems. While prior research has evaluated low-level programming models, there is a lack of detailed analysis of high-level programming models and their ability to achieve competitive performance compared to low-level APIs, as well as a lack of domain-specific abstractions that expose approximate and energy-efficient computing techniques to developers. This thesis addresses these gaps across the programming-model landscape through three main contributions: the analysis of high- and low-level abstractions for SIMT architectures, the design of domain-specific abstractions and novel techniques for approximate and energy-efficient computing, and the exploration of compiler approaches for automatically generating vectorized code on SIMD architectures. First, we provide a comprehensive evaluation of programming models, assessing how high-level constructs map efficiently to GPUs and approach the performance of native low-level APIs. Second, we extend high-level programming models with domain-specific abstractions and techniques for approximate and energy-efficient computing, enabling developers to

exploit these approaches without specialized hardware knowledge. Lastly, we investigate the autovectorization capabilities of modern compilers for RISC-V vector architectures, providing insights into compiler effectiveness, vectorization coverage, and performance optimization opportunities.

Overall, this thesis contributes to improve the performance of high-level programming abstractions for SIMD and SIMT architectures, as well as the design of new programming-model abstractions and techniques for approximate and energy-efficient computing on heterogeneous architectures.

Contents

List of Figures

List of Tables

LISTINGS

ACRONYMS

DVFS	Dynamic Voltage and Frequency Scaling
GPU	Graphics Processing Unit
GPUs	Graphics Processing Units
ISA	Instruction Set Architecture
GPGPU	General Purpose GPU
MSRs	Model-Specific Registers
CPUs	Central Processing Units
TPUs	Tensor Processing Units
NPUs	Neural Processing Units
SVE	Scalable Vector Extension
AVX	Advanced Vector Extension
RVV	RISC-V Vector
ILP	Instruction Level Parallelism
SISD	Single Instruction Single Data
SIMT	Single Instruction Multiple Thread
SIMD	Single Instruction Multiple Data
MISD	Multiple Instruction Single Data
MIMD	Multiple Instruction Multiple Data
SMs	Streaming Multiprocessors
CUs	Compute Units

1 INTRODUCTION

Over the past decades, the landscape of computing architectures has evolved dramatically. The end of Dennard scaling [46] and the slowdown of Moore's law [169] have driven the emergence of increasingly heterogeneous systems that integrate a wide range of computational units. These include traditional multicore Central Processing Units (CPUs) enhanced with Single Instruction Multiple Data (SIMD) extensions, Graphics Processing Units (GPUs) featuring Single Instruction Multiple Thread (SIMT) execution, and a growing number of domain-specific accelerators, such as Tensor Processing Units (TPUs) and Neural Processing Units (NPUs). While such architectures deliver high performance computing, they also incur greater energy consumption and pose significant challenges for performance portability and optimization. Efficiently exploiting heterogeneous hardware requires appropriate programming abstractions that bridge the gap between low-level architectural details and high-level productivity. Figure 1.1 illustrates the spectrum of programming models across modern heterogeneous hardware, organized by abstraction levels.

At the lowest level, programming models expose fine-grained hardware control. For SIMD CPUs, intrinsic APIs (e.g., Advanced Vector Extension (AVX) for Intel [76], Scalable Vector Extension (SVE) for ARM [9], RISC-V Vector (RVV) for RISC-V CPUs [146]) let developers manually express data-level parallelism in order to fully exploit the vector capabilities of the target architecture. For SIMT hardware, vendor-specific interfaces such as CUDA [128], HIP [4], and Level Zero [79] provide a direct way to program, respectively, NVIDIA, AMD and Intel GPUs. Similarly, domain-specific accelerators, such as Google TPUs, rely on custom APIs to exploit their specialized hardware efficiently.

While these low-level programming models enable maximum performance on the target architecture, they significantly reduce productivity and portability. Programs written with low-level

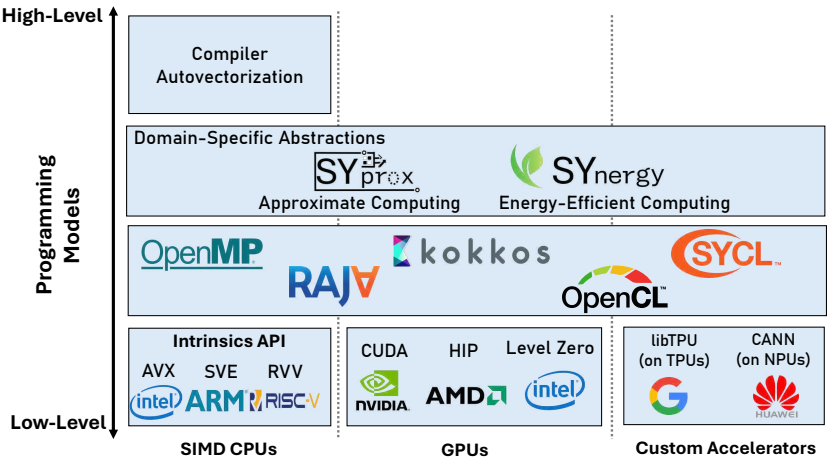


Figure 1.1: Overview of programming model abstractions.

interfaces are tightly coupled to a particular architecture and require extensive re-engineering when ported to new platforms. As a result, there is growing demand for high-level programming models such as Kokkos [185], OpenMP [131], and SYCL [65] that enable performance portability across heterogeneous architectures. However, high-level abstractions remain general-purpose, overlooking domain-specific features and optimization opportunities. As computing systems evolve toward energy-constrained and accuracy-flexible paradigms, there is a growing need to extend these programming models with domain-specific abstractions that explicitly expose aspects such as energy efficiency and approximation. At the top of the abstraction hierarchy, compilers play a key role in automatic optimization. Through autovectorization, they detect and exploit data-level parallelism, generating SIMD code without programmer intervention. This fully automatic approach represents the highest level of abstraction, where the compiler handles the optimizations. However, its effectiveness varies across architectures and compilers, particularly for emerging systems such as RVV processors.

This thesis spans the full abstraction stack shown in Figure 1.1, covering the analysis of low- and high-level programming models, the design of domain-specific abstractions for approximate and energy-efficient computing, as well as the evaluation of compiler autovectorization in modern SIMD architectures.

1.1 TOWARDS MODERN SIMD AND SIMT ARCHITECTURES

For several decades, improvements in processor performance were driven primarily by increases in clock frequency and advances in Instruction Level Parallelism (ILP), while memory accesses rarely limited performance, since computation was typically slower than loads and stores. During this period, the major assumption was that single processor performance would continue to double approximately every 18 months, allowing developers to rely on faster hardware for performance gains rather than explicit parallelization. Around the early 2000s, however, this trend began to break down as processors encountered three fundamental barriers: the *power wall*, the *memory wall*, and the *ILP wall* [10]. Increasing clock frequencies became even more difficult due to thermal and power density limits (*power wall*), while memory latency increasingly constrained processor utilization (*memory wall*) [191]. At the same time, the potential for extracting additional instruction-level parallelism was approaching its limits due to inherent data and control dependencies (*ILP Wall*). Together, these limitations, often referred to as the *Brick wall* of uniprocessor performance, marked the end of automatic performance scaling. Figure 1.2 illustrates this evolution of microprocessors over the past decades. While the number of transistors per chip continued to grow in line with Moore's Law, the gains from higher single-core frequencies plateaued, and around the mid-2000s, manufacturers began integrating multiple cores on a single chip, initiating a shift toward parallel computing. As single-core performance improvements slowed down, hardware designers increasingly relied on parallelism to continue scaling performance. This transition led to the widespread adoption of SIMD and SIMT parallel architectures [176]. While these architectures provide substantial performance improvements, they require programmers to expose parallelism explicitly, and carefully manage data movement, memory hierarchies, and execution models. Consequently, the effective utilization of SIMD and SIMT hardware depends not only on the underlying architecture but also on the availability of abstractions that allow developers to express parallelism and leverage hardware-specific optimizations. With performance scaling now relying on parallelism, program-

ming model abstractions have become a critical interface between complex hardware and productive software development.

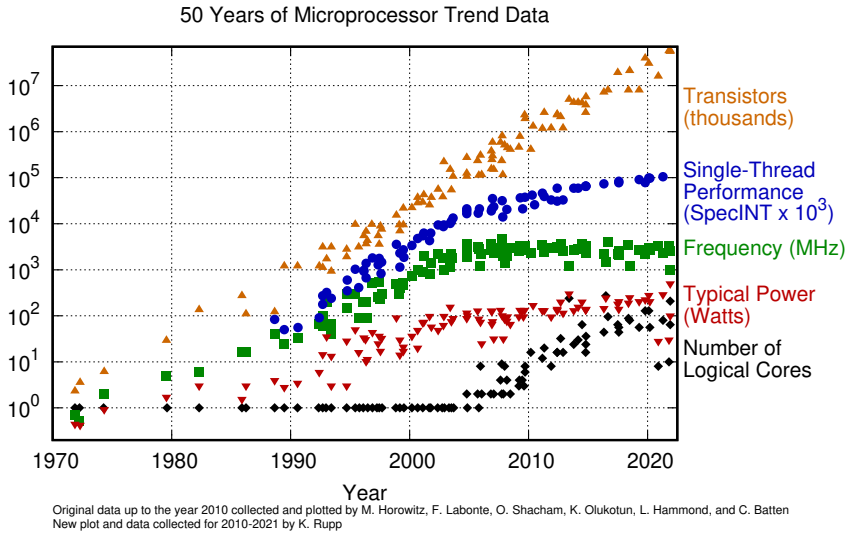


Figure 1.2: The evolution of microprocessors [158].

1.2 PROGRAMMING MODELS FOR MODERN SIMD AND SIMT ARCHITECTURES

Programming models define how developers express computations, manage parallelism, and interact with hardware resources. They are essential for bridging the gap between software and increasingly complex heterogeneous hardware, where CPUs, GPUs, and domain-specific accelerators coexist. In Figure 1.3, we represent the landscape of programming models using the metaphor of a ski touring route across a mountain. On the ascent, we find low-level programming models. These models offer direct and detailed control over the hardware, allowing expert developers to extract maximum performance at the cost of greater complexity. On top of the mountain, domain-specific languages represent the maximum level of control that we can achieve for particular application domains and accelerators. However, as with the uphill phase of a ski tour, progress requires significant effort and

expertise. Developers must have deep knowledge of the target architecture, carefully tune their code, and accept limited portability across platforms in exchange for peak performance. Once we reach the top of the mountain, the descent represents high-level programming models that make development smoother and more accessible. Frameworks such as OpenMP, Kokkos and SYCL abstract away the complexities of low-level programming models. Developers can write portable code that targets multiple architectures without manually tuning every detail. Finally, at the base of the slope, compiler-driven techniques, such as autovectorization and automatic parallelization, represent the highest level of abstraction, where the compiler automatically identifies and exploits parallelism without explicit programmer intervention.

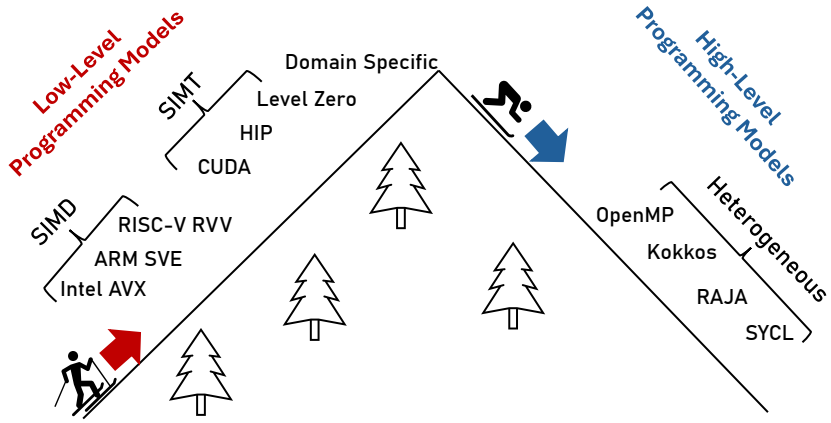


Figure 1.3: Programming models for SIMD and SIMT architectures.

1.3 CHALLENGES FOR HIGH LEVEL PROGRAMMING MODELS

High-level programming models promise portability and productivity across increasingly heterogeneous architectures such as CPUs, GPUs, and custom accelerators. These programming models enable developers to express parallelism at a higher level of abstraction, freeing them from the complexities of hardware-specific programming. However, achieving functional portability (i.e. the ability to compile and execute code on multiple platforms) is only part of the challenge. The final goal is performance

portability [135, 136] that is the ability to obtain near-optimal performance across heterogeneous architectures without the need for architecture-specific code modifications or extensive manual tuning. Although high-level programming models have been extensively studied for performance portability [38, 139], a significant gap between portability and achievable performance persists. High-level abstractions often hide architectural details that are critical for optimization, limiting the ability of developers and compilers to fully exploit hardware capabilities.

1.4 THE NEED FOR DOMAIN-SPECIFIC ABSTRACTIONS

Back to our metaphor for ski tourism, high-level programming models allow developers to reach the top of the mountain with an helicopter, without the effort of climbing. They provide performance-portable interfaces that simplify development across heterogeneous architectures. However, these models are inherently general-purpose and do not expose specialized features. Emerging paradigms such as approximate computing [123, 171] and energy-efficient computing [50] require programming abstractions that can express domain-specific constraints, such as acceptable accuracy loss or energy consumption, while maintaining the productivity of high-level programming. Extending current programming models with these capabilities is essential to fully exploit the trade-offs between performance, accuracy, and energy efficiency in heterogeneous systems.

Approximate Computing

Approximate computing is a computational paradigm that allows controlled inaccuracies to improve performance and optimize resource utilization while maintaining acceptable results. Many applications, such as image and video processing, machine learning, and artificial intelligence, can tolerate small errors, allowing to reduce computation and memory demands while preserving an acceptable output quality. However, the practical application of approximate computing remains highly challenging. Approximation introduces a multi-objective optimization problem, requiring a careful balance between accuracy, performance, and energy

efficiency. Since these objectives are often in conflict, developers must navigate a Pareto frontier of trade-offs rather than aiming for a single optimal solution. Furthermore, implementing approximation techniques from scratch and applying them to a program demands deep expertise, careful tuning, and extensive validation to avoid unacceptable errors, making the process error-prone, time-consuming, and largely inaccessible to most developers. These challenges highlight the need for abstractions that enable approximation capabilities in high-level programming models.

Energy-efficient Computing

Energy efficiency has become a primary design objective in modern high-performance and exascale computing [113]. The end of Dennard scaling, rising electricity costs, and increasingly strict power constraints have made it essential to optimize performance within fixed energy budgets. One of the most effective mechanisms for energy optimization is Dynamic Voltage and Frequency Scaling (DVFS), which adjusts the processor frequency and voltage to balance power consumption and performance. While vendor-specific interfaces provide access to these capabilities, there is still no portable and unified way to control energy behavior across CPUs, GPUs, and custom accelerators. As a result, developers find it difficult to integrate energy optimization into heterogeneous applications. In this context, the integration of portable energy-efficient abstractions and optimizations into high-level programming models represents an important step toward advancing heterogeneous energy-efficient computing.

1.5 RESEARCH QUESTIONS

Building on the challenges and motivations presented in sections 1.3 and 1.4, this thesis is organized around a set of research questions that are examined in detail throughout the work. These research questions span the entire stack of programming model abstractions presented in Figure 1.1: from an analysis of low-level and high-level programming models for SIMT hardware (RQ1), to novel domain-specific extensions for approximate (RQ2) and energy-efficient computing (RQ3), to fully automatic compiler-

driven optimizations for emerging SIMD architectures (RQ4). Each question is addressed in one or more chapters, and finally, Chapter 8 answers these questions, summarizing the key findings and insights of the study.

Research Question 1:

Research Question 1
How do high-level abstractions in SYCL perform against native low-level APIs on SIMT architectures?

The rapid evolution of heterogeneous computing architectures has made it increasingly challenging to achieve high performance across diverse hardware. SYCL provides a high-level, single-source C++ programming model that allows applications to run on a heterogeneous hardware. However, achieving performance comparable to low-level APIs such as CUDA is still a challenge. Low-level APIs offer direct control over hardware resources, whereas SYCL relies on abstraction layers that do not always map efficiently to every target architecture. In particular, on SIMT architectures, the efficient mapping of features such as parallel reductions and atomic operations often requires careful utilization of architecture-specific instructions. This motivates the first research question of the thesis, which focuses on understanding the limitations of SYCL 2020 features on modern SIMT architectures and its efficiency compared to low-level abstractions.

Research Question 2:

Research Question 2
How can we design domain-specific and high-level abstractions for approximate computing?

Approximate computing has become increasingly important in applications such as machine learning, image processing and scientific simulations, where small and controlled accuracy losses can lead to significant improvements in performance and energy efficiency. However, modern high-level programming mod-

els, while improving performance and portability across heterogeneous platforms, remain general-purpose and lack domain-specific abstractions for approximate computing. As a result, developers must implement approximation techniques from scratch and tailor them to each target architecture, limiting adoption and hindering systematic exploration of performance–accuracy trade-offs. This gap motivates our second research question on approximate computing.

Research Question 3:

Research Question 3
How can we design domain-specific and high-level abstractions for energy-efficient computing?

Energy efficiency has become a critical concern in modern heterogeneous computing systems, where power and thermal constraints increasingly limit scalability and operational costs. However, current high-level programming models do not provide abstractions for implementing energy-efficient computing techniques, forcing developers to rely on vendor-specific interfaces and ad-hoc solutions that are neither portable nor scalable. To address this challenge, Research Question 3 is decomposed into three complementary sub-questions:

3.1. How can we design a portable, high-level interface that integrates energy profiling and frequency-scaling capabilities across heterogeneous architectures?

Currently, each hardware vendor provides its own proprietary API for profiling energy consumption, scaling frequency, and accessing other energy-related functionality. This fragmentation makes it difficult to develop portable energy-aware applications.

3.2. How can we improve existing frequency-scaling approaches?

DVFS is one of the most widely used software techniques for improving energy efficiency. Existing approaches, however, are typically limited to per-kernel or per-application frequency adjustments, which are not able to fully exploit opportunities for energy savings.

3.3. How can we improve the prediction of the optimal frequency configurations of state-of-the-art models?

Predicting the optimal frequency configuration is essential for automatic frequency tuning. However, state-of-the-art models often require offline profiling, execution data, or are limited to CPUs, ignoring SIMT architectures such as GPUs. Improving these models to accurately predict optimal frequency configurations without executing the application would enable portable energy-aware programming in heterogeneous systems.

Together, these sub-questions address the design, methodology, and predictive modeling aspects required to implement high level abstractions for energy-efficient computing.

Research Question 4:

Research Question 4
How well do compilers support autovectorization on modern RISC-V SIMD architectures?

Modern workloads increasingly rely on data-level parallelism, making SIMD architectures a key mechanism for efficiently accelerating computations. RISC-V, as an open-source Instruction Set Architecture (ISA), enables developers and hardware designers to customize SIMD units freely. Unlike traditional vector instructions, such as Intel AVX and ARM Neon, which rely on fixed-width registers, the RVV extension introduces variable-length vector registers, providing greater flexibility and scalability across different hardware platforms. Programming SIMD architectures often requires a deep understanding of the underlying hardware and of the code patterns that can be effectively vectorized. While intrinsic APIs give developers precise control over vector instructions, they also require manual implementation and careful tuning, which reduces portability and increases complexity. To mitigate these challenges, autovectorization provides much higher-level of abstraction by enabling compilers to detect vectorizable patterns and automatically generate vector instructions, without requiring developers to manage low-level architectural details. However, automatically generating efficient vector instructions remains a challenge for compilers due to RVV's vector-length-agnostic paradigm, the diversity of hardware capabilities, and the inherent difficulty of detecting vectorizable patterns

and mapping them to appropriate vector instructions. Furthermore, the RISC-V software ecosystem is still in its early stages, and compiler support for autovectorization remains limited and rapidly evolving. These considerations motivate our last research question.

1.6 MAJOR CONTRIBUTIONS

In this thesis, we target the four research questions identified in the design of programming model abstractions and present our effort to address each in one or more chapters. In this section, we present our major contributions related to each research question.

-*Analysis of SYCL 2020 high-level abstractions on SIMT architectures:* In Chapter 3, we address RQ1 by focusing on SYCL, one of the emerging high-level programming models for heterogeneous architectures. We examine the high-level semantics introduced in the SYCL 2020 standard [65] and evaluate their effectiveness on SIMT architectures from AMD, Intel, and NVIDIA. To support this study, we present SYCL-Bench 2020, a benchmark suite consisting of 9 templated kernels covering 44 configurations and six computational patterns, specifically designed to assess key SYCL 2020 capabilities. Our objective is to provide a comprehensive understanding of the potential and limitations of these high-level abstractions, offering guidance for the development of efficient and portable heterogeneous applications. To this end, we investigate features such as reductions and atomic operations to show how effectively they are translated to the underlying low-level hardware APIs across different GPU vendors. By analyzing both the source code and observed performance differences between SYCL implementations and backends on each platform, we quantify the extent to which these abstractions can approach the efficiency of native vendor-specific implementations. Our results demonstrate performance variability between SYCL implementations, and show that some backends generate suboptimal low-level instructions for the target architecture, thereby limiting achievable hardware performance. These findings highlight both the strengths and limitations of SYCL's high-level abstrac-

tions, clarifying when they can deliver near-native performance and when architecture-specific tuning remains necessary. The resulting insights provide actionable guidance for developers and compiler designers, helping shape the evolution of a performant and portable programming model.

-Domain-specific Abstractions For Approximate Computing:
In Chapters 4, we tackle RQ2 by introducing a new domain-specific abstractions that extend SYCL with support for approximate computing. In this part of the thesis we present *SYprox*, a SYCL-based interface that enables programmers to easily implement heterogeneous approximate computing applications. In SYprox, we implement state-of-the-art approximation techniques, such as perforation, mixed precision, and signal reconstruction, and introduce a novel perforation strategy that exploit the host–device execution model of modern heterogeneous systems to improve performance. We further show how multiple approximation techniques can be combined to explore new Pareto-optimal configurations, achieving better performance–accuracy trade-offs compared to applying each technique individually. Our experiments conducted on GPUs from different vendors show that SYprox is portable and that its approximation techniques consistently outperform state-of-the-art frameworks [114, 134].

-Domain-specific Abstractions For Energy-efficient Computing:
Chapter 5 introduces high-level abstractions for energy-efficient computing and addresses two subquestions of RQ3.
RQ3.1 is tackled through the design of *SYnergy*, a portable energy interface based on SYCL that enables energy profiling and frequency scaling across a wide range of heterogeneous hardware. Unlike traditional approaches that apply frequency scaling at the application level, SYnergy provides fine-grained per-kernel energy control, allowing each kernel to be tuned independently according to its performance and energy characteristics.
RQ3.2 is addressed through the development of a phase-based frequency-scaling methodology. Our analysis shows that changing GPU frequencies introduces non-negligible overheads that can undermine the performance and energy benefits of DVFS. To mitigate this, we propose a phase-based approach that analyzes

both the task DAG and runtime behavior to identify execution phases in which frequency change can be applied with minimal impact. By grouping tasks into phases, the methodology significantly reduces the number of frequency transitions, limiting overhead while enabling energy-efficient execution. We further extend this approach to multi-GPU and multi-node programs by enriching the task DAG with MPI communication information, effectively overlapping DVFS overhead with communication in distributed environments. Our study demonstrates that this methodology overcomes the limitations of state-of-the-art approaches based on coarse- and fine-grained frequency scaling, enabling scalable energy savings on heterogeneous and distributed HPC systems.

Finally, in Chapter 6, we address RQ3.3, focusing on general purpose and domain-specific energy modeling for DVFS prediction. We start by defining a new general purpose energy model that predicts, for each kernel, the optimal frequency based on static code features extracted at compile time. This model enables per-kernel optimizations that balance energy savings and performance according to user defined energy target metrics, such as minimum energy, maximum performance, or energy delay product. We further enhance the general purpose model by considering a methodology for building energy models based on domain specific features. In many real-world applications, energy consumption and performance are influenced not only by static code characteristics but also by problem parameters, workload structure, and input-dependent behavior. These factors can significantly affect the compute/memory balance of a kernel, and including them allows the model to better capture the actual behavior of each application. We validate our methodology on two representative scientific applications with substantial societal and scientific impact: LiGen, a GPU-accelerated drug discovery platform, and Cronos, a magnetohydrodynamics solver widely used in astrophysical simulations. By tailoring the energy model to the internal structure and workload characteristics of these applications, we achieve substantially improved prediction accuracy over the general-purpose approach.

-*Analysis of autovectorization support in RISC-V SIMD architectures:* In Chapter 7, we address RQ4 by examining the highest level of the programming model hierarchy (Figure 1.1), focusing on the autovectorization capabilities of GCC and LLVM compilers on modern RISC-V Vector architectures. The goal of this study is to identify the strengths and limitations of current compiler support and to provide insight into the role of compiler optimizations for emerging RVV hardware. Our analysis shows how effectively compilers generate vectorized code for different loop patterns and how specific RISC-V Vector features influence performance.

1.7 LIST OF PUBLICATIONS

Conference Proceedings

1. Lorenzo Carpentieri, Biagio Cosenza: *Towards a SYCL API for Approximate Computing*, in Proceedings of the 2023 International Workshop on OpenCL (pp. 1-2). DOI: https://doi.org/10.1145/3585341.3585374.

2. Lorenzo Carpentieri, Biagio Cosenza: SYprox: Combining Host and Device Perforation with Mixed Precision Approximation on Heterogeneous Architectures, in Proceedings of the 39th ACM International Conference on Supercomputing (pp. 1-12). DOI: https://doi.org/10.1145/3721145.3725741.

3. Lorenzo Carpentieri, Antonio De Caro, Majid Salimi Beni, Kaijie Fan, Biagio Cosenza: *Phase-Based Frequency Scaling for Energy-Efficient Heterogeneous Computing*, in 2025 IEEE International Parallel and Distributed Processing Symposium (IPDPS) (pp. 824-836). IEEE. DOI: https://doi.org/10.1109/IPDPS64566.2025.00078.

4. Lorenzo Carpentieri, Mohammad VazirPanah, Biagio Cosenza: *A Performance Analysis of Autovectorization on RVV RISC-V Boards*, in 2025 33rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP) (pp. 129-136). IEEE. DOI: https://doi.org/10.1109/PDP66500.2025.00026.

5. Kaijie Fan, Marco D'Antonio, Lorenzo Carpentieri, Biagio Cosenza, Federico Ficarelli, Daniele Cesarini: *SYnergy: Fine-grained Energy-Efficient Heterogeneous Computing for Scalable Energy Saving*, in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (pp. 1-13). DOI: https://doi.org/10.1145/3581784.3607055.

   In this paper, I contributed to the development of the SYnergy interface and the implementation of the applications used for the experiments.

6. Luigi Crisci, Lorenzo Carpentieri, Peter Thoman, Aksel Alpay, Vincent Heuveline, Biagio Cosenza: *SYCL-Bench 2020: Benchmarking SYCL 2020 on AMD, Intel, and NVIDIA GPUs*, in Proceedings of the 12th International Workshop on OpenCL and SYCL (pp. 1-12). DOI: https://doi.org/10.1145/3648115.3648120.

   In this paper, I contributed to the analysis and development of the reduction, group algorithm, and atomic operations benchmarks in SYCL.

7. Gianmarco Accordi, Davide Gadioli, Gianluca Palermo, Luigi Crisci, Lorenzo Carpentieri, Biagio Cosenza, Andrea Rosario Beccari: *Unlocking performance portability on LUMI-G supercomputer: A virtual screening case study*, in Proceedings of the 12th International Workshop on OpenCL and SYCL (pp. 1-4). DOI: https://doi.org/10.1145/3648115.3648125.

In this paper, I contributed to the porting of LiGen application from CUDA to SYCL.

Journal Publications

1. Luigi Crisci, Lorenzo Carpentieri, Biagio Cosenza, Gianmarco Accordi, Davide Gadioli, Emanuele Vitali, Gianluca Palermo, Andrea Rosario Beccari: *Enabling performance portability on the LiGen drug discovery pipeline*, in *Future Generation Computer Systems*, 2024, 158, 44-59. DOI: `https://doi.org/10.1016/j.future.2024.03.045`.

   In this article, I contributed to the improvement of LiGen performance portability by using SYCL features such as sub-group, group algorithm and reduction.

Workshop Proceedings

1. Lorenzo Carpentieri, Marco D'Antonio, Kaijie Fan, Luigi Crisci, Biagio Cosenza, Federico Ficarelli, Daniele Cesarini, Gianmarco Accordi, Davide Gadioli, Gianluca Palermo, Peter Thoman, Philip Salzmann, Philipp Gschwandtner, Markus Wippler, Filippo Marchetti, Daniele Gregori, Andrea Rosario Beccari: *Domain-Specific Energy Modeling for Drug Discovery and Magnetohydrodynamics Applications*, in Proceedings of the SC'23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis (pp. 1790-1800). DOI: `https://doi.org/10.1145/3624062.3624261`.

2. Lorenzo Carpentieri, Biagio Cosenza: *Energy Saving By Approximate Computing*, in IPDPS PhD Forum 2025 Welcome and Abstracts, 2025 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). DOI: `https://doi.org/10.1109/IPDPSW66978.2025.00227`.

1.8 THESIS ORGANIZATION

In this chapter, we discussed the ideas introduced in this thesis, raised four main research questions to be addressed, and mentioned our major contributions. Chapter 2 provides the necessary background and theoretical concepts by introducing SIMD/SIMT architectures, high-level and low-level programming models, approximate and energy-efficient computing. Chapter 3 analyzes high-level abstractions of the SYCL programming model. Chapter 4 introduces abstractions for approximate computing, and Chapters 5 and 6 present abstractions and optimizations for energy-efficient computing. Chapter 7 investigates autovectoriza-

tion on RISC-V Vector architectures. Finally, Chapter 8 provides a summary of the ideas and contributions of the thesis, recalls the research questions and answers them, and provides some future directions for each of the chapters.

2 BACKGROUND

In this chapter, the necessary background information is provided to facilitate a better understanding of the ideas and outcomes of this dissertation. First, we introduce the Flynn taxonomy with a focus on SIMD and SIMT architectures, as well as the corresponding low-level and high-level programming models used to develop efficient code for these systems. Next, we provide an overview of the RISC-V ISA and the RVV vector extension. Finally, we examine the most important approximate and energy-efficient computing techniques at the software level.

2.1 FLYNN'S TAXONOMY

The field of computer architecture has seen rapid advancements since the inception of computing machines, evolving from early sequential processors to the highly parallel and heterogeneous systems used today. To describe and compare these architectural trends, researchers often rely on Flynn's taxonomy [176], a classification that organizes architectures according to the number of instruction streams and data streams they process concurrently. This model distinguishes four fundamental classes each capturing a different form of parallelism and computational behavior. Figure 2.1 illustrates these four categories and provides a visual representation of how processing elements coordinate instructions and data within each paradigm.

In a Single Instruction Single Data (SISD) architecture, a single control unit executes one instruction at a time on a single data element. This represents the traditional sequential execution model characteristic of early uniprocessors and still underpins the scalar cores present in modern CPUs. Although optimizations such as pipelining or multiple functional units may be used to enhance throughput, SISD systems fundamentally operate on a single instruction stream applied to a single stream of data. The SIMD model extends this execution style by applying the same

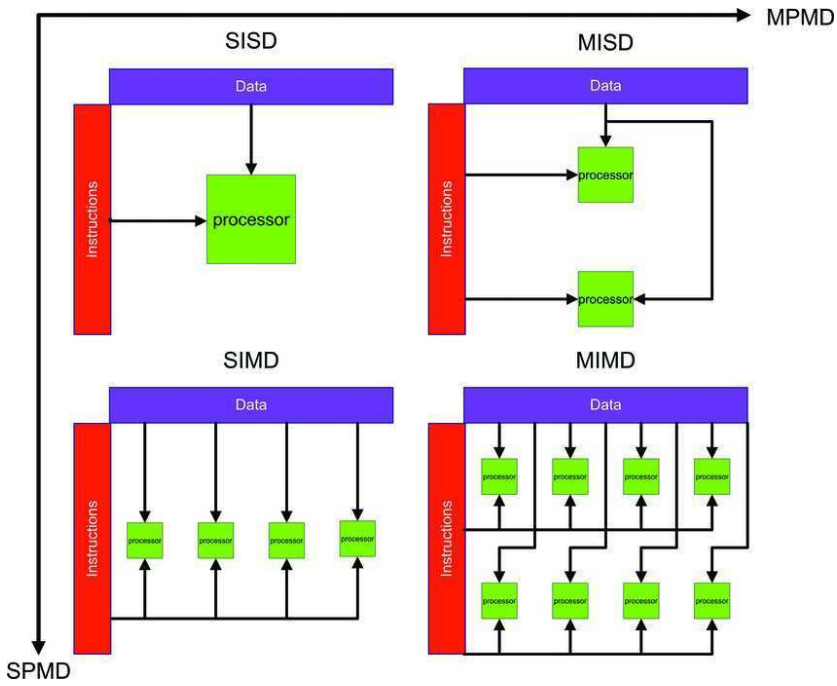


Figure 2.1: Classification of computer architectures.

instruction simultaneously across multiple data elements. This approach is highly effective when workloads exhibit substantial data-level parallelism, such as in image and signal processing, scientific simulations, and numerical kernels. SIMD architectures typically implement wide vector registers that enable operations to be broadcast over large sets of data, drastically improving throughput for regular, uniform computations. A Multiple Instruction Single Data (MISD) architecture, in contrast, allows several instruction streams to operate on the same data stream. While this category is largely theoretical and seldom realized in general-purpose systems, certain accelerators exhibit characteristics reminiscent of MISD, where multiple computations are applied in a pipeline to the same input data. The Multiple Instruction Multiple Data (MIMD) paradigm encompasses architectures in which multiple processors execute different instructions on different data simultaneously. This is the dominant model for contemporary parallel computing and includes multicore CPUs, distributed-memory clusters, and large-scale supercomputers.

MIMD architectures support heterogeneous workloads and provide flexibility through task-level parallelism, with individual processing units operating independently except for explicit synchronization or communication. Together, these four categories provide a conceptual foundation for understanding the parallel execution models employed in modern systems.

SIMD Architectures

Modern CPUs increasingly rely on SIMD execution units to exploit data-level parallelism and improve throughput of different applications. SIMD capabilities are now ubiquitous across mainstream processor families, including Intel x86 architectures equipped with AVX extensions, ARM processors implementing SVE, and the emerging RISC-V ISA, which has introduced a flexible vector extension [54]. Although the details of these ISA differ, they share a common execution principle: vector instructions operate on wide registers containing multiple data elements, allowing the processor to apply the same operation across an entire array in a single cycle. To support this capability, SIMD processors are equipped with special hardware, which allows it to execute an instruction which operates on multiple data elements. They are equipped with vector registers which can hold more than 1 element each. Traditional systems such as Intel's AVX-128/AVX-256/AVX-512, employ fixed-width vector registers (128, 256, or 512 bits), while ARM's SVE and the RISC-V Vector Extension adopt a vector-length-agnostic model, where the actual register width is determined at implementation time rather than encoded in the ISA. In this thesis we focus on the study of the vector extension for modern RISC-V Vector processors. Also, SIMD processors are equipped with vector instructions which perform the same operation on all elements in a vector register. For example, a `VLOAD` (vector load) can load multiple elements from RAM into a vector register, a `VADD` (vector add) can perform addition on all elements, and so on. Thus single instruction acts on multiple data elements. This reduces the total number of instructions executed, reducing the overhead of an instruction cycle. In SIMD architectures, computation is driven by vector

registers, whose width determines the number of elements that can be processed simultaneously.

SIMT Architectures

Modern computing systems often combine multiple paradigms from Flynn's taxonomy to achieve high performance across diverse workloads. One example is the SIMT execution model, which underpins modern GPUs. In SIMT, a single central control unit broadcasts instructions to multiple processing units (PUs), allowing them to execute operations concurrently on independent data. Each PU has its own registers and local memory, but unlike multi-core (MIMD) systems, instruction sequencing is determined entirely by the central unit. In Flynn's taxonomy, SIMT can be regarded as a variation of SIMD. The key difference between SIMT and SIMD lanes is that each of the PUs in the SIMT Array have their own local memory, and may have a completely different Stack Pointer (and thus perform computations on completely different data sets), whereas the ALUs in SIMD lanes know nothing about memory per se, and have no register file. Modern GPUs leverage the SIMT model to execute thousands of threads in lockstep, exposing a thread-based programming model while fundamentally operating as a predicated SIMD system. This enables high-throughput parallelism for workloads such as image processing, scientific simulations, and deep learning, while MIMD-like features allow flexible scheduling for heterogeneous or control-flow divergent tasks.

General Purpose GPU

A GPU is a specialized electronic circuit originally designed to manipulate and alter memory to accelerate the creation of images in a frame buffer intended for output to a display device.In order to manipulate images, the GPU performs a set of operations called the graphics pipeline. Over time, GPU became more flexible and programmable, enhancing their capabilities. This allowed graphics programmers to create more interesting visual effects and realistic scenes with advanced lighting and shadowing techniques. Other developers also began to exploit

the power of GPUs for general purpose computation. This is the born of General Purpose GPU (GPGPU). In early GPGPU programming in order to perform general-purpose computation it was necessary to map the problem into the graphics pipeline representing, for example, arrays as texture, kernels as shaders and computing as drawing. In modern GPGPU to simplify this process different programming model have been developed to fully exploit the GPU architecture for general purpose computing. The Graphics Processing Unit (GPU) provides much higher instruction throughput and memory bandwidth than the CPU within a similar price and power envelope. Many applications leverage these higher capabilities to run faster on the GPU than on the CPU. This difference in capabilities between the GPU and the CPU exists because they are designed with different goals in mind. While the CPU is designed to excel at executing a sequence of operations, called a thread, as fast as possible and can execute a few tens of these threads in parallel, the GPU is designed to excel at executing thousands of them in parallel (amortizing the slower single-thread performance to achieve greater throughput). The GPU is specialized for highly parallel computations and therefore designed such that more transistors are devoted to data processing rather than data caching and flow control. The schematic Figure 2.2 shows an example distribution of chip resources for a CPU versus a GPU. In general, an application has a mix of parallel parts and sequential parts, so systems are designed with a mix of GPUs and CPUs in order to maximize overall performance. Applications with a high degree of parallelism can exploit the massively parallel nature of the GPU to achieve higher performance than on the CPU.

GPU ARCHITECTURE Figure 2.3 illustrates the general organization of a modern GPU. GPU architectures are organized around highly parallel compute blocks, referred to as Streaming Multiprocessors (SMs) in NVIDIA terminology and Compute Units (CUs) in AMD terminology. These units contain the processing elements responsible for executing threads. SMs and CUs are grouped into larger processing clusters that typically share an L2 cache, enabling efficient data access and coordination across the GPU.

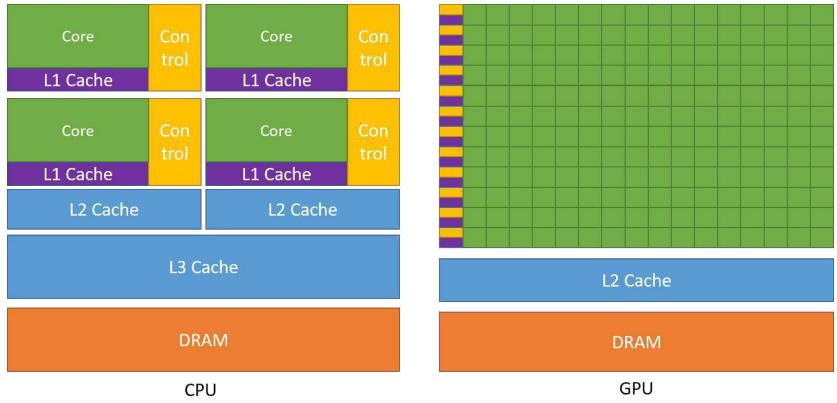


Figure 2.2: CPUs vs GPUs high-level architecture

Figure 2.4 presents the internal structure of an NVIDIA A100 SM. At the top of the hierarchy, an L1 instruction cache supplies instructions to four independent processing blocks, each equipped with its own L0 instruction cache for low-latency access. The SM also integrates an L1 data cache and a shared memory region, which together support fast data transfers and efficient communication among threads. The computational capability of the SM is provided by four symmetric processing blocks. Each block contains warp schedulers and dispatch units capable of issuing instructions to 32 threads per cycle. The blocks include several types of execution units: general-purpose ALUs for integer and floating-point operations, tensor cores for matrix computations, load/store units for memory operations, and special-function units for complex mathematical operations. Each block is also equipped with a dedicated register file for storing operands and intermediate results.

EXECUTION MODEL A GPU is programmed through *kernels*, which are functions that are executed in parallel by multiple threads. Each thread executes the same kernel and uses coordinates to determine which data element it should process.

When a kernel is launched, threads are organized into a grid, or range, which represents the complete parallel workload. This grid is further divided into thread blocks, or work-groups, which share resources like shared memory and can synchronize their

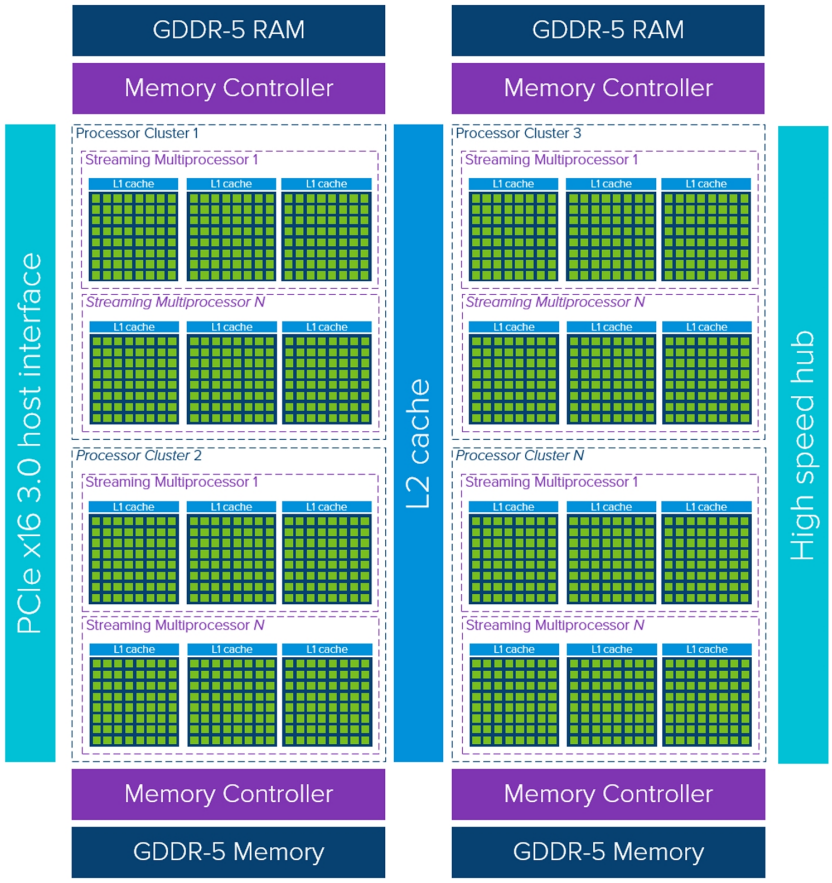


Figure 2.3: A typical GPU architecture [188].

execution. Each thread block contains multiple threads, which are the smallest unit of computation. In GPUs, threads are not scheduled independently. Instead, they are grouped together to minimize scheduling overhead. Depending on the architecture, they have different names, e.g. warp on NVIDIA, wavefront on AMD, sub-group on Intel, and different sizes, e.g. 32 threads on NVIDIA and 64 threads on AMD. This three-level hierarchy (thread $\rightarrow$ block $\rightarrow$ grid) maps naturally to parallel problems: threads might process individual data elements, blocks might handle data subsets, and the grid encompasses the entire computation. Some newer architectures introduce an additional hierarchy level, called a thread group, which is a collection of threads that can synchronize and communicate between SM/CU within

Figure 2.4: The structure of an NVIDIA A100 SM

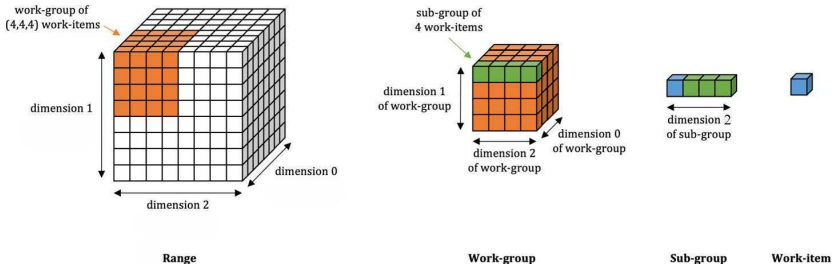


Figure 2.5: Thread hierarchy example for a 3D kernel in SYCL. Threads are organized into thread blocks, which are grouped into a grid. Note that each programming model organizes dimensions differently.

a Processing Cluster [32]. Figure 2.5 shows an example of thread hierarchy for a 3D kernel.

MEMORY HIERARCHY At the highest level, GPU memory systems are structured as a hierarchy of distinct tiers, each offering different trade-offs among latency, bandwidth, capacity, and

scope. Figure 2.6 illustrates the memory organization of a typical modern GPU.

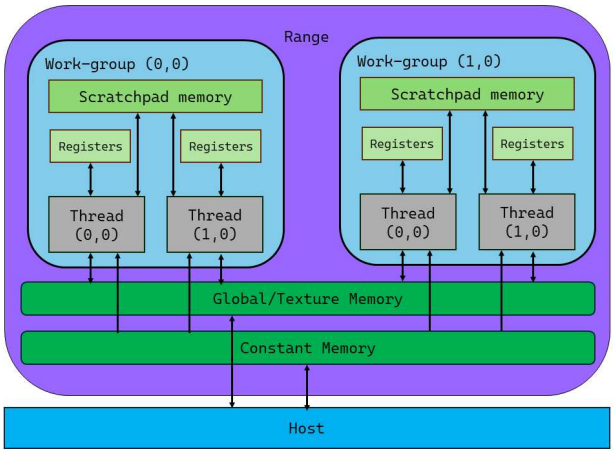


Figure 2.6: Memory hierarchy of a GPU.

The largest and slowest tier is global memory, commonly implemented using HBM or GDDR. It provides high capacity and bandwidth but also the highest latency. Global memory is accessible to all Streaming Multiprocessors (SMs) or Compute Units (CUs) and typically stores large datasets used by the application. Sitting below global memory, the L2 cache is shared across all SMs/CUs. It reduces traffic to global memory and lowers effective latency for data reused across different processing units. Within each SM/CU, the L1 cache provides fast access to frequently used data and instructions. On many architectures, the L1 data cache is unified with shared memory, a low-latency, programmer-managed scratchpad used for inter-thread communication within a thread block or workgroup. Proper use of shared memory can dramatically reduce global memory accesses and improve performance. At the lowest level, register files offer the fastest storage but have very limited capacity. Registers are private to each thread, allocated at compile time, and hold the thread's immediate operands and intermediate results. Register pressure can cause register spilling, where the compiler places excess data into local memory, a per-thread memory region that physically resides in global memory and therefore suffers the same high latency. GPUs also implement specialized memory

spaces to support common access patterns. Constant/Texture memory provides cached read-only storage optimized for broadcast across threads, while texture memory is tuned for 2D/3D spatial locality. These memories accelerate workloads involving lookup tables, images, and other structured datasets.

2.2 RISC-V ISA

RISC-V is an open-standard, modular ISA developed at the University of California, Berkeley in the early 2010s [197]. RISC-V represents a royalty-free alternative to commercial ISAs such as x86 and ARM, and has gained widespread adoption in both academia and industry. Its open nature, combined with a freely available development toolchain and open-source hardware designs, has positioned RISC-V as a major contender for general-purpose and specialized processors. The RISC-V ISA follows traditional Reduced Instruction Set Computer (RISC) principles: it provides a minimal set of frequently used instructions, is highly regular and consistent in instruction encoding, and achieves complex functionality by combining simple instructions. The base 32-bit integer instruction set, RV32I, contains only 40 instructions with two source operands and one destination operand, and supports six different instruction formats. Despite its simplicity, RISC-V is Turing complete, meaning it can implement any computational algorithm. RISC-V is modular and extensible, allowing developers to build processors with custom capabilities by adding optional extensions. Common extensions include the integer multiplication/division extension (M), atomic instructions (A), single- and double-precision floating point (F and D), and, importantly for this work, the vector extension (V) for data-parallel operations.

RISC-V Vector Extension (RVV)

The RVV extension adds a set of instructions and vector registers to support data-parallel computation. Unlike traditional SIMD extensions such as Intel AVX, RVV implements a vector-length agnostic (`VLA`) model, allowing programs to execute efficiently across implementations with varying vector register sizes (`VLEN`). This flexibility enables software portability across

different RVV-compliant processors without modification. RVV defines 32 vector registers, each `VLEN` bits wide. The number of elements operated on in a vector instruction is controlled by the vector length (vl) and the element width (`SEW`, Selected Element Width). The RVV standard also define a group multiplier (`LMUL`) which determines the number of registers forming a single vector operand. For instance, $\texttt{LMUL} = 4$ groups four consecutive registers together for vector operations, while fractional `LMUL` values constrain vl to a portion of a single register. The effective maximum vector length (`VLMAX`) is computed as $\texttt{VLMAX} = \texttt{LMUL} \times \frac{\texttt{VLEN}}{\texttt{SEW}}$. RVV supports masking and tail policies to provide fine-grained control over operations. Mask registers allow selective updating of vector elements: elements with the mask off can either remain unchanged (undisturbed) or take arbitrary values (agnostic). Similarly, tail policies define how elements beyond vl are handled, enabling operations on arbitrary-sized vectors without affecting remaining elements. Programs written with RVV intrinsics or compiler via autovectorization can exploit these mechanisms to efficiently map data-parallel computations to the underlying hardware.

2.3 PROGRAMMING MODELS

Programming models provide the abstraction layer that allows developers to express parallel computation in a form that can effectively exploit the architectural features of modern hardware. On SIMD architectures, programming models determine how data-parallel operations are expressed so that they can be mapped with vector instructions. On SIMT-based GPUs, programming models define how many threads are grouped together, how they share and access memory, and how their execution is coordinated. The programming model also provides mechanisms for threads to communicate, synchronize, and manage dependencies within these groups. Across both paradigms, programming models range from low-level interfaces, which expose hardware details and give developers fine-grained control, to high-level abstractions, which also consdier portability, maintainability, and ease of use. The following subsections introduce these two cate-

gories and describe the most relevant approaches for SIMD and SIMT programming.

Low-Level Programming Models

Low-level programming models expose the execution hardware directly, allowing developers to tailor their implementations to a specific processor architecture. On SIMD-capable CPUs, this often involves writing code using intrinsics, low-level functions that map directly to specific vector instructions. Intrinsics provide fine-grained control over the vector units and allow programmers to exploit ISA-specific features such as masked operations, gather/scatter instructions, and custom data rearrangements. Each vendor provides its own vector extension, Intel with AVX [76], ARM with SVE [9], and RISC-V with the Vector Extension [146], offering intrinsics to operate with vector instructions. As a result, programmers must implement separate code paths using the appropriate intrinsics API for each target hardware to fully exploit its capabilities. Listing 2.1 shows an example of vector addition written with AVX intrinsics. In Line 5-7 we initialize the array that will be used for the addition. Line 10-11 load the data in the vector. Finally, in line 11 the intrinsics `_mm512_add_ps` execute the vector addition.

```
1  #include <immintrin.h>
2  #include <stdio.h>
3
4  int main() {
5      float a[16] = {...};
6      float b[16] = {...};
7      float c[16];
8
9      __m512 vec_a = _mm512_load_ps(a);
10     __m512 vec_b = _mm512_load_ps(b);
11     __m512 vec_c = _mm512_add_ps(vec_a, vec_b);
12     _mm512_store_ps(c, vec_c);
13     // Print results...
14     return 0;
15 }
```

Listing 2.1: AVX512 code snippet.

While intrinsics can deliver optimal performance, this approach reduces portability and is generally reserved for performance-critical kernels Similarly for SIMT architectures, each major vendor provides its own low-level programming interface designed to expose the full capabilities of its hardware. NVIDIA GPUs are programmed through CUDA, which offers explicit control over thread hierarchies, memory spaces, and synchronization. AMD provides HIP, a programming model derived from CUDA that allows both native execution on AMD GPUs and source translation from CUDA code. Intel offers Level Zero, a low-level API that allows developers to control command queues, memory transfers, and kernel execution on Intel GPUs.

These vendor-specific interfaces enable developers to exploit fine-grained architectural features, but they also lock implementations to a particular hardware family and require significant porting effort when targeting multiple GPU architectures.

High-Level Programming Models

High-level programming models aim to provide performance portability by abstracting away many of the architectural details that low-level approaches require. For SIMD architectures, the primary high-level mechanism is autovectorization, where the compiler analyzes loops and transforms suitable regions into vector instructions automatically. Modern compilers such as LLVM/Clang and GCC include advanced vectorization passes that recognize vectorizable loop patterns and generate efficient SIMD code. The main advantage of autovectorization requires no explicit programmer intervention, although its effectiveness is ultimately limited by the compiler's ability to understand and transform the code. In the remainder of this thesis, we focus specifically on the Autovectorization is attractive because it preserves portability and does not require any effort from the programmer, but its effectiveness is limited by the compiler's ability to detect loop patterns that can be vectorized. For GPU programming, high-level models provide abstractions that remove the need to write vendor-specific CUDA [128], HIP [4], or Level Zero [79] code. OpenMP offloading [131] extends the OpenMP standard with directives that allow code regions to be

executed on accelerators, enabling heterogeneous execution with minimal modifications to existing CPU code. Frameworks such as OpenCL [64], Kokkos [44] and RAJA [15] further raise the level of abstraction by offering performance-portable parallel programming interfaces that decouple application logic from hardware backends. These models allow the same codebase to target CPUs and GPUs from different vendors by selecting the appropriate execution backend at compilation time. Among programming models for heterogeneous architectures, the emerging SYCL standard provides a portable and unified approach for programming CPU, GPUs, and custom accelerators. In this thesis, SYCL will be used as the high-level programming model for building our approximate and energy efficient computing abstractions. .

The SYCL Programming Model

SYCL [65] is a royalty-free, cross-platform abstraction layer that enables code for heterogeneous processors to be written in a "single-source" style using completely standard C++. The SYCL standard is developed by the Khronos Group , a consortium of industry-leading companies that work together to create open standards for graphics, parallel computing, and vision processing. Initially designed as a modern interface to OpenCL, since revision 3 SYCL officially supports third party backends, making it more flexible and portable across different hardware platforms. SYCL builds on the concepts of modern C++, providing a higher-level abstraction for parallel programming that is easier to program compared to proprietary solutions. The SYCL platform represents a valid alternative to CUDA and HIP programming model, offering several advantages. Unlike CUDA and HIP, which are a proprietary language used only for NVIDIA and AMD hardware respectively, SYCL allows developers to support various devices, including CPUs, GPUs, and custom accelerators.

2.3.0.1 *SYCL Compilers*

In contrast with proprietary language like CUDA or HIP, SYCL is an open standard, and therefore anyone is free to implement it.

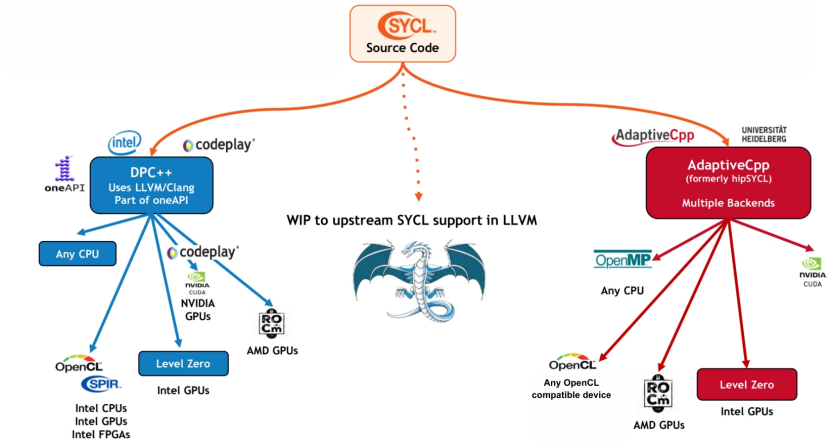


Figure 2.7: The SYCL current major compilers [37].

While SYCL also supports library-only implementations, to fully exploit the potential of the standard, a compiler is needed.

Figure 2.7 shows the current major SYCL compilers. DPC++ is Intel's implementation of SYCL, which is part of the oneAPI initiative. It's clang-based and exposes and it primarily targets Intel GPUs and CPUs via OpenCL and LevelZero. Additionally, it includes also av NVIDIA backend (developed by Codeplay) for NVIDIA GPUs and a HIP backend for AMD GPUs. AdaptiveCpp is a lightweight SYCL implementation developed by the University of Heidelberg. Formerly known as hipSYCL, it was the first SYCL implementation to support third party backend [6]. It exposes a CUDA and HIP backend for NVIDIA and AMD GPUs respectively, an OpenMP backend for CPUs and an OpenCL backend which can leverage the SPIR-V format to target any OpenCL device. Additionally, AdaptiveCpp features a unique, Single-Source Single-Compilation (SSSC) mode [7], which allows the same source code to be compiled for multiple backends in a single compilation step, leveraging a powerful JIT compiler to generate optimized code for each target device at runtime.

In addition to production-grade implementations, the SYCL flexibility lead to the development of several extensions and novel backends, such as distributed computing [162], Vulkan API [180], and debugging [181]. Figure 2.8 shows a comprehensive set of the current major SYCL extensions.

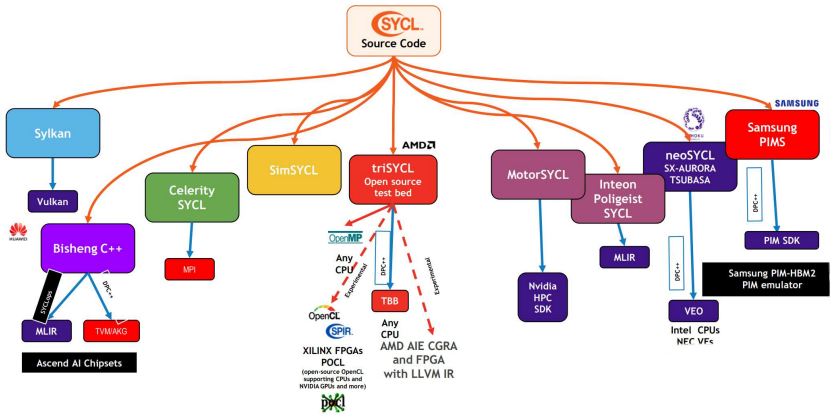


Figure 2.8: The SYCL extension ecosystem [37].

2.3.0.2 *A SYCL example program*

Listing 2.2 shows a simple SYCL application that performs a vector addition on a device. A SYCL program is conceptually structured into three scopes: the *application scope*, which contains all host-side logic; the *command group scope*, which describes a unit of work to be executed on the device; and the *kernel scope*, which defines the device function to be compiled and run on the accelerator. The application begins by including the SYCL headers (line 1) and allocating the host-side arrays in `main` (lines 44–47). The vector addition is implemented inside the `simple_vadd` function, which first constructs a device selector (line 19). Unlike CUDA, which is bound to NVIDIA GPUs, SYCL supports heterogeneous devices, and the `default_selector` chooses a device according to backend-specific heuristics. A `queue` is then created (line 21), encapsulating the device information required for executing kernels. To make data available to the device, the input and output arrays are wrapped into SYCL `buffers` (lines 24–28). These buffers allow the SYCL runtime to manage data transfers automatically, ensuring that the kernel receives the data in the correct memory space. Work is submitted to the device by providing a command group (line 30). This command group defines `accessors` (lines 31–33), which describe how each `buffer` will be accessed by the device, including the access modes (e.g. read-only, write-only etc.). The SYCL runtime uses this information to deter-

mine dependencies between kernels submitted to a device and move data as needed. The kernel itself is expressed as a lambda function (line 35). Each instance of this lambda is executed by a different work-item, identified by its `id<1>` parameter. The kernel simply computes the sum of the two input arrays and stores the result in the output accessor. The call to `parallel_for` (line 39) launches the kernel across a one-dimensional index space of size `N`. The template parameter `SimpleVadd<T>` provides a unique name for the kernel, which is required by the SYCL compilation model when using lambda-based kernels. One important aspect of SYCL's execution model is that command submission is asynchronous: `deviceQueue.submit` returns immediately while the device computation proceeds in the background. To guarantee that results are available on the host before checking them, the program relies on SYCL's RAII semantics: once the buffers go out of scope at the end of `simple_vadd`, the runtime ensures that all associated device work has completed and the data has been copied back. The correctness of the computation is then verified in `main` (lines 51–57), confirming that the device produced the expected output. This example illustrates the essential structure of a SYCL application: host-side data preparation, creation of buffers, construction of a command group that binds accessors and kernels, and the asynchronous execution of data-parallel kernels on a selected hardware. Although simple, this code captures the core concepts used in more complex SYCL applications throughout this thesis.

```
#include <sycl/sycl.hpp>

using namespace std;

template <typename T, size_t N>
void simple_vadd(array<T, N>& VA, array<T, N>& VB, array<T, N>& VC)
     {
  sycl::default_selector device_selector;
  sycl::queue deviceQueue{device_selector};
  sycl::range<1> numOfItems{N};
  // Buffers
  sycl::buffer<T, 1> bufferA(VA.data(), numOfItems);
  sycl::buffer<T, 1> bufferB(VB.data(), numOfItems);
  sycl::buffer<T, 1> bufferC(VC.data(), numOfItems);

  deviceQueue.submit([&](sycl::handler& cgh) {
    sycl::accessor A(bufferA, cgh, sycl::access_mode::read);
    sycl::accessor B(bufferB, cgh, sycl::access_mode::read);
    sycl::accessor C(bufferC, cgh, sycl::access_mode::write);
    // Kernel function
    auto kern = [=](sycl::id<1> wiID) {
      accessorC[wiID] = accessorA[wiID] + accessorB[wiID];
    };
    // Launch kernel using parallel_for
    cgh.parallel_for<class SimpleVadd<T>>(numOfItems, kern);
  });
}

int main() {
  const size_t array_size = 4;
  array<int, array_size> A = {{1, 2, 3, 4}};
  array<int, array_size> B = {{1, 2, 3, 4}};
  array<int, array_size> C;
  simple_vadd(A, B, C);
  for (unsigned int i = 0; i < array_size; i++) {
    // Results check ...
  }
  return 0;
}
```

Listing 2.2: SYCL code snippet.

2.4 APPROXIMATE COMPUTING

Approximate computing is a computing paradigm that deliberately trades a small reduction in output accuracy for gains in performance or energy efficiency. Rather than always generating accurate answers, approximation computing approaches aim to produce "approximately correct" outcomes within acceptable ranges. It is important to note that approximate computing is not suitable for all types of applications. It is most effective in scenarios where the trade-offs between accuracy and resource efficiency are well understood and can be managed effectively. In critical applications, such as medical or aerospace systems, where high precision is paramount, approximate computing is generally not suitable. However, in many consumer and commercial applications, it offers a valuable approach to optimizing resource usage while achieving acceptable results. This approach is particularly valuable in domains where exact precision is unnecessary, such as multimedia processing, machine learning, or certain scientific simulations.

PERFORMANCE–ENERGY–ERROR TRADE-OFFS The key challenge in approximate computing comes from its inherently multi-objective nature. When introducing approximation, the goal is not only to maximize performance or minimize energy consumption, but also to keep the induced error within tolerable limits. Figure 2.9 shows the multi-objective nature of the approximate computing problem. As the problem to solve is non-linear we can not select a single solution that is better than the other one in all the dimensions (Error, Performance, and Energy) but we can find a set of points for which no other point is better with respect to all the objectives, called Pareto-set or Pareto-frontier [40]. To reduce the complexity of the problem we can focus on two dimensions. Figure 2.10 (a) shows the trade-off between speedup (x-axis) and error (y-axis) for different approximate techniques applied to the same application. In this setting, the most attractive points lie toward the bottom right, representing high speedup and low error. Similarly, Figure 2.10 (b) plots energy consumption against error, where the best solutions appear in the bottom left, since lower energy usage is preferred. In both

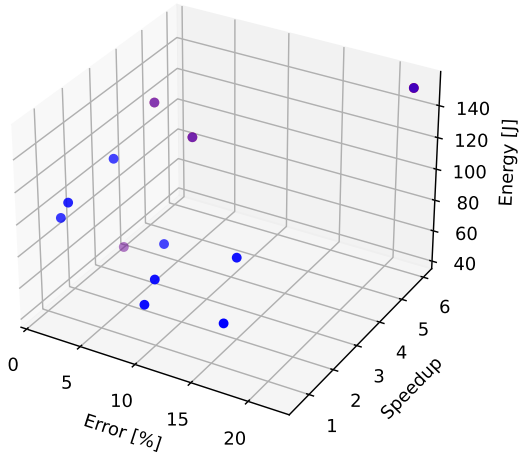
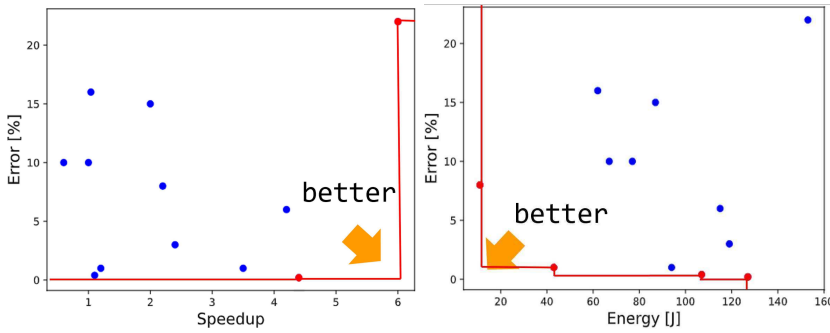


Figure 2.9: Appriximate Computing: 3D multi-objective problem

cases, we consider the error introduced by the approximation to avoid solutions with a too-large error. The red line in Figure 2.10 highlights the Pareto-optimal solutions. All the points that are not in the Pareto-frontier are not interesting as we can select one of the points in the Pareto-set that is better in both dimensions.



(a) Speedup–Error (b) Energy–Error

Figure 2.10: Approximate computing as a multi-objective optimization problem with a Pareto frontier.

APPROXIMATE COMPUTING TECHNIQUES Approximate computing techniques can be introduced either as hardware features during the design of a computing platform or implemented at the software level [123]. In this thesis, we focus on software-level approximate computing techniques and their integration into the SYCL programming model. To provide context, it is useful to discuss the most relevant software techniques commonly employed in approximate computing.

One of the most widely used methods is *perforation*, a general-purpose technique that reduces output accuracy by intentionally skipping certain computations, instructions, data, or loop iterations. By omitting selected operations or data, perforation decreases the computational resources required to produce a result, potentially improving performance and energy efficiency.

Another key approach is *reduced precision computation*, where variables and data structures are represented with fewer bits. Choosing the appropriate numerical representation involves a trade-off between accuracy and resource cost. To mitigate the risk of significant accuracy loss, *mixed precision* strategies are often applied, where critical sections of a program use full precision while less sensitive parts employ lower precision formats.

Memoization involves the storage of function results along with their input parameters. When the same function is called with previously encountered inputs, the cached result can be reused instead of recomputing it. In the context of approximate computing, exact matches are not required; a similarity function can be employed to identify inputs that are close enough, allowing an approximate result to be returned based on previously computed values.

Finally, *relaxed synchronization* is a technique applied in parallel applications where synchronization represents a bottleneck. Synchronization is used especially to ensure that threads reach various points in their execution in a predictable manner or to ensure that all threads see consistent values when shared variables are updated. By relaxing these constraints, threads can execute more independently, trading strict correctness for increased throughput and reduced latency.

These software-level techniques illustrate a range of strategies for trading accuracy for performance and energy efficiency. In

the following chapters, we discuss how mixed precision and perforation methods can be implemented within SYCL programs to exploit the capabilities of SIMT architectures.

2.5 ENERGY-EFFICIENT COMPUTING

Energy efficiency has emerged as one of the top ten research challenges in HPC [167]. Among software-level approaches to improve energy efficiency, DVFS is particularly promising, as it dynamically adjusts processor voltage and clock frequency to achieve a balance between performance and power consumption.

Dynamic Voltage and Frequency Scaling

DVFS is a widely-used power management technique in modern processors which dynamically adjusts the supply voltage and operating frequency of a computing device based on workload demands. By modulating voltage and frequency, DVFS reduces power consumption during periods of low utilization while scaling up performance when high computational throughput is required. This adaptability allows significant energy savings while maintaining acceptable performance, making DVFS a cornerstone for energy-efficient computing in HPC systems. The trade-off is that reducing voltage and frequency improves energy efficiency but also reduces performance. The theoretical foundation of DVFS lies in the behavior of complementary metal-oxide-semiconductor (CMOS) circuits, where total power consumption is composed of dynamic and static components [200]. Dynamic power, which depends on switching activity, capacitance load, supply voltage, and clock frequency, can be expressed as:

$$P_{\text{dynamic}} = \alpha C V^2 f$$

where α is the average switching activity, C is the total capacitance, V is the supply voltage, and f is the operating frequency. Static power, or leakage power, arises from leakage currents in active circuits and is independent of the clock frequency:

$$P_{\text{static}} = V_{cc} \cdot I_{cc}$$

where V_{cc} is the supply voltage and I_{cc} is the leakage current. In CPU and GPUs, voltage and frequency are typically coupled through discrete voltage-frequency pairs defined by the processor. Adjusting one parameter inherently affects the other, making DVS and DFS effectively interchangeable. Hence, DVFS is often considered to have the same meaning as Dynamic Frequency Scaling (DFS) and Dynamic Voltage Scaling (DVS). DVFS is implemented and exposed differently across computing devices. Each hardware vendor provides low-level interfaces that expose on-chip sensors and power-management controls, enabling software to read and modify power and energy parameters. On CPUs, examples include Intel's RAPL interface [36], IBM's EnergyScale [154], and AMD's Application Power Management (APM) [2], all of which allow monitoring of power states and, depending on the architecture, modification of power limits or frequency settings. On GPUs, similar functionality is available through vendor-specific tools such as NVIDIA's Management Library (NVML) [126], AMD's ROCm System Management Interface (ROCm SMI)[1], and Intel's Level Zero API [79], which provide access to device power consumption, clock frequencies, and voltage controls. Together, these interfaces allow software to dynamically manage energy usage across CPUs and GPUs by balancing performance and power consumption in real time. In the following sections, we discuss the application of DVFS techniques on NVIDIA, AMD and Intel GPUs as well as CPUs, providing the foundation for energy optimization strategies explored in Chapters 5 and 6.

2.5.0.1 *DVFS on CPUs*

DVFS on CPUs is primarily exposed through the RAPL interface. First introduced with the Sandy Bridge architecture, RAPL [36] provides hardware support for monitoring and controlling power across different components of a processor. RAPL organizes the processor into several power domains, each corresponding to a physically hardware subsystem such as the entire package, the CPU cores, or the DRAM attached to the memory controller. Figure 2.11 shows the different domains.

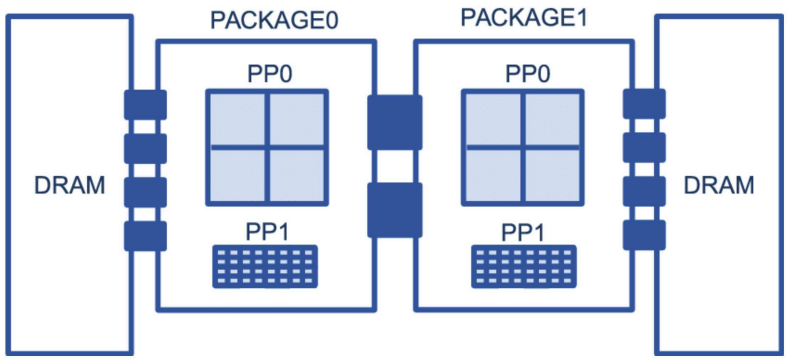


Figure 2.11: RAPL power domains

For each domain, RAPL exposes cumulative energy consumption, configurable power limits, time windows for enforcing these limits, and static information such as minimum and maximum supported power levels. The principal RAPL domains include:

- *Package (PKG):* measures the total energy consumed by the entire processor socket, including CPU cores and uncore components;
- *Power Plane 0 (PP0):* accounts for the energy usage of all CPU cores on the socket;
- *Power Plane 1 (PP1):* represents the energy consumed by uncore devices such as integrated GPUs (primarily present in desktop processors);
- *DRAM:* reports the energy consumption of the memory subsystem attached to the integrated memory controller.

More recent Intel architectures, such as Skylake, introduce an additional system-level domain called *PSys*, designed to account for platform-wide energy consumption when the primary source of power draw lies outside the CPU or GPU. Energy measurements in RAPL are retrieved through Model-Specific Registers (MSRs). Each RAPL domain maintains a 32-bit counter that accumulates energy consumption since system boot, updated roughly every millisecond. These counters store energy values in processor-defined units. On Linux systems, MSRs can be accessed directly using the `msr` kernel module. After detecting

the CPU model and reading the energy unit, domain-specific energy values can be obtained by reading the corresponding MSR. For example, `MSR_PKG_ENERGY_STATUS` reports the accumulated energy for the package domain. In addition to direct MSR access, several higher-level interfaces provide convenient access to RAPL such as Power Capping framework [144] or PAPI [198]. Beyond energy monitoring, RAPL also enables *power capping*, i.e., enforcing an upper bound on the average power that a domain may consume over a configurable time interval. Power capping indirectly affects DVFS. When a cap is active, the processor may reduce its operating frequency to remain within the specified power envelope. However, RAPL does not provide explicit control over frequency. To set a specific CPU frequency, it is necessary to rely on other mechanisms such as the Linux `cpufreq` [58] interface or direct manipulation of MSRs.

2.5.0.2 *DVFS on GPUs*

GPU FREQUENCY DOMAIN GPUs expose core and memory frequency clock domains that influence both performance and power consumption. In earlier desktop-oriented devices, such as NVIDIA Maxwell and Pascal GPUs (e.g., Titan X) two domains were typically configurable: the *core frequency*, governing the clock rate of SMs, and the *memory frequency*, determining the operating speed of the DRAM subsystem. The former dictates the computational throughput of arithmetic and control units, while the latter directly affects the available memory bandwidth. Figure 2.12 conceptually illustrates these domains, highlighting the separation between compute (SM) and memory (DRAM) subsystems on NVIDIA GPUs. However, the degree of control over these domains has changed substantially in recent data center GPUs. While early data center GPUs allowed independent scaling of both frequencies, new accelerators such as NVIDIA Volta, Ampere, and Hopper, AMD's Instinct series, and Intel's Max GPUs generally restrict DVFS to the compute domain only while memory frequency is fixed by the vendor. As a consequence, DVFS on modern HPC GPUs focuses exclusively on adjusting the core frequency, while memory operations run at a constant,

hardware-defined clock.

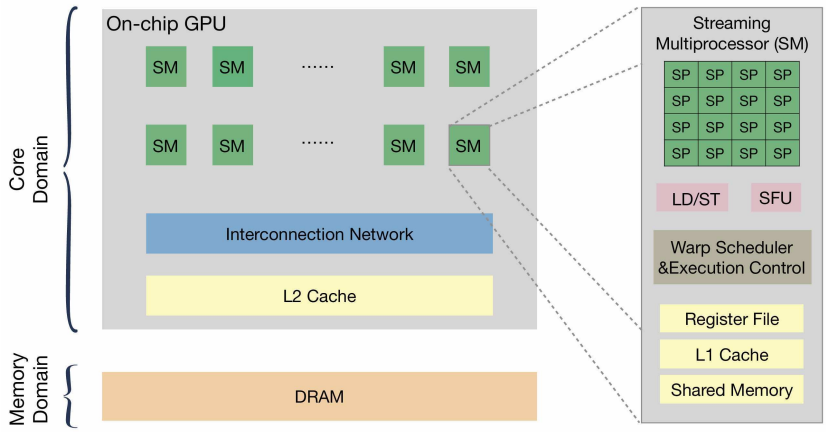


Figure 2.12: Simplified structure of NVIDIA GPUs frequency domains

POWER MANAGEMENT LIBRARY Vendor-specific system management interfaces provide access to DVFS functionality. NVIDIA exposes power, frequency, utilization, and energy counters through NVML. AMD provides similar capabilities through the ROCm SMI interface, while Intel offers power and frequency control for accelerators through the Level Zero API. These interfaces make it possible to retrieve data such as instantaneous power draw, cumulative energy, and currently active frequencies, and, depending on architectural constraints, to modify frequency. GPU power, energy, and frequency management are exposed through vendor-specific interfaces, each providing dedicated functions for different operations. For NVIDIA GPUs, the NVIDIA Management Library (NVML) provides `nvmlDeviceGetPowerUsage()` to read the instantaneous power consumption of the GPU. If the GPU supports hardware energy counters, total energy consumption can be obtained with `nvmlDeviceGetTotalEnergyConsumption()`. To modify the operating frequency of the GPU cores, NVML exposes `nvmlDeviceSetApplicationsClocks()`, which allows applications to set preferred core and memory clocks, though on modern HPC GPUs only the core clock can typically be changed. AMD GPUs provide similar functionality through the ROCm System Management Interface (ROCm SMI). The function

`rsmi_dev_power_ave_get()` returns the averaged power draw of the device over a sampling window, while `rsmi_dev_energy_count_get()` gives access to cumulative energy consumption on devices that support energy counters. Frequency control is handled via `rsmi_dev_od_freq_range_set()`, which allows adjusting the allowed range of core frequencies; as with NVIDIA devices, memory frequency is generally fixed on modern datacenter GPUs. Intel GPUs expose power and frequency telemetry through the Level Zero API. The function `zesGetPowerProps()` reports the available power domains and associated characteristics, while `zeDeviceGetPowerEnergy()` provides cumulative energy consumption for the selected domain. Core frequency can be modified within allowed limits using `zesFrequencySetRange()`, whereas memory frequency remains fixed on current architectures.

3

HIGH-LEVEL ABSTRACTIONS IN SYCL

HPC systems are increasingly heterogeneous, integrating multiple processing technologies such as general-purpose processors (CPUs), graphics processing units (GPUs), and domain-specific accelerators to meet the growing performance and functionality demands. While this diversity enables significant computational throughput, it also complicates programming and optimization. Exploiting the full potential of each hardware platform traditionally requires deep knowledge of the underlying architecture and specialized parallel programming models. Historically, this has led to the proliferation of hardware-specific programming languages, such as CUDA for NVIDIA GPUs or HIP for AMD GPUs, which, while highly efficient, significantly undermine code portability across different architectures. Ideally, developers aim for a single source code capable of running efficiently on multiple platforms. One approach is to maintain multiple architecture-specific implementations, optimizing each individually. Although this yields high performance, it is unsustainable in terms of development effort and long-term maintainability. To address this challenge, high-level programming models have emerged, offering performance portability [127, 137]: the ability to write a single codebase that can execute efficiently across a range of heterogeneous devices.

In this chapter, we focus on SYCL, a C++ single-source programming model designed for heterogeneous computing. SYCL provides abstractions that simplify the management of devices and memory while enabling developers to exploit parallelism across CPUs, GPUs, and other accelerators. By using SYCL, developers can achieve a balance between high-level productivity and low-level performance, potentially reducing the need for multiple hardware-specific implementations. However, the true value of high-level abstractions lies in their ability to generate code that approaches the efficiency of native low-level APIs. Evaluating how effectively these abstractions map to hardware and exploit

architecture-specific features is crucial to understanding their practical performance and the extent to which they can replace specialized, low-level programming models.

This chapter presents a detailed micro-benchmarking study to assess the performance and portability of SYCL 2020 high-level abstractions. We extend the old SYCL-Bench[97] suite, introducing nine new benchmarks covering 44 configurations and six code patterns to evaluate key SYCL 2020 features, including unified shared memory, reduction kernels, specialization constants, group algorithms, in-order queues, and atomic operations. Although SYCL-Bench integrates six key SYCL 2020 features, in this chapter we focus specifically on high-level abstractions designed to simplify common parallel computing operations, such as reduction algorithms and atomic operations. By focusing on specifically designed micro-benchmarks, this chapter examines the bottom two levels of the abstraction stack introduced in Chapter 1, Figure 1.1: low-level and hardware-specific programming models (i.e. CUDA, HIP and LevelZero) and high-level, performance-portable abstractions (i.e. SYCL). We analyze how reduction and atomic operations are implemented in SYCL and how effectively different SYCL implementations and backends leverage architecture-specific properties to achieve high performance across different SIMT architectures (AMD, Intel and NVIDIA). To the best of our knowledge, this represents the first benchmark suite specifically tailored for SYCL 2020 that focuses on high-level abstractions, systematically evaluating how they are implemented across different SYCL backends and implementations, how they map to low-level instructions, and how effectively they enable performance and portability across diverse SIMT architectures.

3.1 RELATED WORKS ON PERFORMANCE PORTABILITY

As high-level programming models continue to evolve, it becomes increasingly important to develop dedicated benchmark suites that enable consistent and reproducible evaluation of how well these abstractions translate into efficient execution on different hardware architectures. In particular, assessing whether high-level constructs can achieve performance comparable to na-

tive low-level APIs is essential for understanding their practical viability in heterogeneous computing. Benchmark suites provide a systematic methodology for evaluating performance, resource utilization, and the effectiveness of different programming approaches. A number of benchmark suites have been developed to study specific architectural or computational characteristics. Bienia et al. [16] introduced PARSEC, a suite of C/C++ applications designed to support characterization and development of Chip-Multiprocessors (CMPs). Kulkarni et al. [94] proposed the Lonestar suite to investigate parallelism and locality patterns in sparse graph workloads. Additionally, with the rise of heterogeneous programming models such as OpenCL and SYCL, several studies have benchmarked their behavior and portability across architectures [27, 39, 82, 84].

More recently, SYCL has emerged as a promising high-level programming model for heterogeneous systems, resulting in the introduction of several SYCL-focused benchmarking efforts. Some works have evaluated SYCL implementations in terms of compile-time characteristics [182], performance portability [120, 150, 182], and maturity over multiple compiler and backend revisions [107]. Other studies explored SYCL optimizations for specific target architectures [83, 149], often focusing on performance tuning rather than semantic evaluation. There has also been work examining particular SYCL 2020 features. Alpay et al. [5] studied interoperability mechanisms through *host tasks*. Ashbaugh et al. [12] investigated the evolution of the SYCL memory model, including the introduction of *sycl::atomic_ref*. Joubrel et al. [85] compared USM and Accessors using a PCI-bound microbenchmark, analyzing performance and programmability trade-offs.

This chapter focuses on the performance behavior of high-level abstractions such as reductions and atomic operations, showing how different SYCL implementations and backends map them to hardware on AMD, NVIDIA, and Intel GPUs, and evaluating how closely their generated code can approach the efficiency of native low-level APIs.

3.2 MOTIVATION

SYCL is rapidly evolving to address the demands of modern heterogeneous programming, yet the performance impact of its newest SYCL 2020 features remains unexplored. To motivate the analysis presented in the following subsections, we first outline the evolution from SYCL 1.2.1 to SYCL 2020 and review the current state of its implementations.

From SYCL 1.2.1 to SYCL 2020

SYCL was first proposed in March 2014 by the Khronos Group as a high-level programming model for OpenCL. SYCL shared much of its definitions with OpenCL, inheriting the execution model, runtime feature set, and device capabilities of the underlying model, while providing C++ usability and flexibility alongside an easier, single-source programming style. A novel working group, the SYCL Working Group, was created within the OpenCL Working Group as a sub-project. However, developers found that the bond with OpenCL was too strict, as the SYCL specification was shown to be well-suited for third-party custom backends [6]. SYCL 2020 was ratified in February 2021 and constitutes a major milestone for the SYCL ecosystem. With the novel specification, the binding with OpenCL drops, allowing for novel third-party acceleration API backends, e.g. CUDA, ROCm, LevelZero, etc. As a result, the SYCL working group was split from the OpenCL one, becoming a standalone entity in the Khronos group. Moreover, the SYCL 2020 release incorporates over 40 new features to enhance flexibility, performance, and productivity. Those additions encompass:

- Unified Shared Memory (USM), a low-level, pointer-based memory API
- Built-in parallel reduction support
- Support for native API interoperability
- Work group and subgroup common algorithm library
- SYCL atomic alignment to the C++ standard

- Runtime queries for fine-grained device selection

These enhancements position SYCL 2020 as a strong candidate for performance and portable heterogeneous programming, but their real impact must be evaluated empirically.

SYCL 2020 Implementations

SYCL currently includes two major implementations: OneAPI DPC++ and AdaptiveCpp. OneAPI DPC++ [11], developed by Intel and based on LLVM, supports OpenCL devices (including CPUs and FPGAs) and provides CUDA, HIP, and Level Zero backends for NVIDIA, AMD, and Intel GPUs. AdaptiveCpp [6], developed at Heidelberg University, offers OpenMP and OpenCL backends for CPUs and supports NVIDIA, AMD, and Intel GPUs through native backends. Its Single-Source Compiler Pass (SSCP) [7] significantly reduces compilation times by performing a single unified compilation and JIT-compiling device code at runtime. Although neither implementation is yet fully SYCL 2020 compliant, both support the majority of key features introduced in the new specification. Additional implementations, such as NeoSYCL [86], triSYCL [63], and Sylkan [180], demonstrate SYCL's growing ecosystem, though they primarily target research or specialized accelerators. Despite this progress, the practical performance portability of the new SYCL 2020 features remains insufficiently explored. Existing studies largely focus on implementation maturity, limited benchmarks, or specific hardware targets, leaving open the question of how these features behave across architectures and real applications. Motivated by this gap, this chapter provides a comprehensive evaluation of the new SYCL 2020 abstractions. We analyze the performance portability achieved by the two major SYCL implementations using both micro-benchmarks explicitly designed to isolate SYCL core features and a real-world industrial application for drug discovery. This combined approach allows us to assess not only feature-level performance but also the practical impact of SYCL's abstractions in a realistic HPC scenario.

3.3 SYCL 2020 FEATURES

In this section, we describe the high-level semantics introduced by SYCL for implementing parallel reduction and atomic operations on SIMT architectures.

Reduction Kernel

Reductions are a common pattern widely adopted in parallel applications to combine elements into a single output. This aggregation is achieved through the application of a specified associative and commutative operation. Relying on optimized implementation of the reduction pattern is crucial for enhancing overall application performance. SYCL 2020 introduced the support for built-in reduction operations that allow writing reduction kernels by leveraging the novel `sycl::reducer` class and the `sycl::reduction` function. Listing 3.1 shows a code snippet implementing a reduction using the reducer class. Lines 2–5 initialize the input buffer on the host, while lines 6–9 define the output buffer that stores the final results of the reduction.

In line 13, an accessor is defined to access the input data on the target device. Lines 15–16 construct two reducer objects: one implementing a sum reduction using `sycl::plus<>()`, and another implementing a maximum reduction with `sycl::maximum<>()`. The `sycl::reducer` objects created by the `sycl::reduction` function in lines 15–16 are implementation-defined and therefore must be constructed using the `auto` keyword. Each reducer encapsulates a reduction variable that exposes a standardized interface defining the operations allowed on that variable. While the exact underlying type depends on the SYCL implementation, the functions and operators used to manipulate the reduction variable are defined by the SYCL standard and guaranteed to be available across all implementations. The reducer provides a `sycl::combine()` function that merges the contribution of a single work-item with the reduction variable using the selected operator, such as addition or multiplication. Finally, in lines 17–19, the `parallel_for` receives the reducer objects as parameters, and the reduction process is automatically handled by the

SYCL runtime, abstracting away low-level architecture-specific details that would otherwise need manual tuning.

```
buffer<int> valuesBuf{1024};
{
  host_accessor a{valuesBuf};
  std::iota(a.begin(), a.end(), 0);
}
int sumResult = 0;
buffer<int> sumBuf{&sumResult, 1};
int maxResult = 0;
buffer<int> maxBuf{&maxResult, 1};

myQueue.submit([&](handler& cgh) {
  // Input values to reductions are standard accessors
  auto inputValues = valuesBuf.get_access<access_mode::read>(cgh);

  auto sumReduction = reduction(sumBuf, cgh, plus<>());
  auto maxReduction = reduction(maxBuf, cgh, maximum<>());
  cgh.parallel_for(range<1>{1024}, sumReduction, maxReduction,
    [=](id<1> idx, auto& sum, auto& max) {
        sum += inputValues[idx];
        max.combine(inputValues[idx]);
    });
});
```

Listing 3.1: Reduction variable in SYCL

The SYCL built-in kernel reduction comes with several advantages in performance, portability, code readability, and maintenance. Manually implementing kernel reductions on each target architecture can be difficult and time-consuming since it necessitates a thorough understanding of each target architecture together with a precise tuning phase. In this way, SYCL hides low-level implementation details, allowing the user to focus on the core application. Furthermore, SYCL 2020 reduction kernels usually require fewer lines of code compared to manual implementations, reducing codebase sizes and improving readability. We included a novel benchmark in SYCL-Bench 2020 for SYCL kernel reductions. The primary objective is to assess the performance enhancements brought about by this new feature compared with manually optimized reductions [70]. We employ `sycl::range parallel_for` instead of `sycl::nd_range parallel_for` for writing the kernel reduction: as it enforces less

strict requirements on thread scheduling, SYCL implementations are free to apply additional optimizations compared to `nd_range` ones. In this way, we plan to better evaluate the kernel reduction implementation quality. Additionally, to explore possible optimization the benchmark is parameterized by a coarsening factor, determining the number of elements combined by each thread.

Group Algorithms

SYCL 2020 introduced group algorithms a set of functions that provide support for operations involving groups of work items, such as group barriers, and collective operations (shift, permute, reduction, etc...). All group algorithm functions take a group as the first argument that defines which group of work items executes the specified function (sub-group or work group). Group algorithm functions abstract away low-level hardware details by encapsulating complex operations into a single, well-defined function optimized for the target architecture. In contrast, a manually implemented function may not be as well tuned to the underlying hardware, potentially resulting in suboptimal performance. Group algorithm functions are designed to be device-agnostic, allowing them to adapt to different devices. Furthermore, group algorithm functions are often more concise and readable than manually implemented alternatives. To evaluate the performance achieved by the group algorithms function we used the `sycl::reduce_over_group` collective function as a case study. The reason behind this choice is straightforward: as reduction performance strongly depends on the target device, they represent a perfect study case to evaluate SYCL implementation quality. Listing 3.2 illustrates the use of group-level reduction operations in SYCL. Line 1-6 initialize the input and output buffer. In line 12-20 two forms of group reduction are implemented. First, the `joint_reduce` function (line 15) performs a reduction over a contiguous range of input values explicitly specified through iterators `first` and `last`. This operation assigns each work-item a portion of the iteration space, and the work-group collaboratively computes the sum of all elements in the range using the `plus<>` operator. The final reduction result is stored in `outputValues[0]` (line 16). The second reduction uses

`reduce_over_group` (line 17), which operates directly on values held by individual work-items. Each work-item contributes a single value, here, `inputValues[it.get_global_linear_id()]`. The work-group then performs a reduction across all such values, producing a `partial_sum`, which is written to `outputValues[1]`.

```
1  buffer<int> inputBuf{1024};
2  buffer<int> outputBuf{2};
3  {
4    host_accessor a{inputBuf};
5    std::iota(a.begin(), a.end(), 0);
6  }
7
8  myQueue.submit([&](handler& cgh) {
9    accessor inputValues{inputBuf, cgh, read_only};
10   accessor outputValues{outputBuf, cgh, write_only, no_init};
11
12   cgh.parallel_for(nd_range<1>(...), [=](nd_item<1> it) {
13     int* first = inputValues.get_pointer();
14     int* last = first + 1024;
15     int sum = joint_reduce(it.get_group(), first, last, plus<>());
16     outputValues[0] = sum;
17     int partial_sum = reduce_over_group(
18         it.get_group(), inputValues[it.get_global_linear_id()],
       plus<>());
19     outputValues[1] = partial_sum;
20   });
21 });
22
23 host_accessor a{outputBuf};
24 assert(a[0] == 523776 && a[1] == 120);
```

Listing 3.2: Group reduction in SYCL

This example demonstrates how SYCL provides group-level collective operations that enable efficient parallel reductions at a granularity between local (work-item) and device-wide execution. By using these high-level primitives, developers can express parallel reduction patterns concisely while relying on the runtime and backend to map them efficiently to the underlying hardware. In SYCL-Bench, we designed a benchmark explicitly targeting the performance of group algorithms by implementing a partial reduction over the elements associated with each work-item within the same work-group using `sycl::reduce_over_group`. The resulting values produced by each work-group are then merged

into a single final result through a hierarchical tree-reduction strategy.

Atomics

SYCL 2020 deprecates the `sycl::atomic` class in favor of the `sycl::atomic_ref` to be aligned with the C++ atomic model. The SYCL `atomic_ref` class adds three template parameters to the standard C++ atomic reference: `sycl::memory_order`, specifying the memory synchronization order of the atomic operation; `sycl::memory_scope`, defining the work items and devices to which the memory ordering constraints of the atomic operation are applied; and `sycl::access::address_space`, indicating the address space of the object referenced by the `atomic_ref` class. Since the previous release of SYCL-Bench does not include a use case for atomic operations, we developed a new benchmark to evaluate the performance of atomic operations for the different SYCL implementations and hardware. The main purpose of the benchmark is to investigate how different SYCL implementations map atomic operations for each target architecture, as well as atomic hardware support on different vendors' GPUs. The benchmark implements a sum reduction (Listing 3.3) using only the `atomic_ref` class (line 23) and the `atomic_fetch_add` functions (line 29). For all the benchmarks we used `sycl::memory_order::relaxed` as it is the only one that SYCL specification guarantees to be supported on all devices [159] (line 24). As `sycl::address_space` and `sycl::memory_scope` we consider the general case where the object referenced by the atomic_ref is in the `sycl::global_space` (line 26) and the ordering constraint applies only to work-items executing on the same device as the calling work item (`sycl::memory_scope::device`) (line 25). The benchmark is templated on the datatype on which the atomic operation is performed.

```
1 buffer<T> inputBuf{N};
2 buffer<T> outputBuf{1};
3 {
4   // Initialize input and output buffer on host
5   host_accessor in_acc{inputBuf};
6   std::iota(in_acc.begin(), in_acc.end(), 1);
7   host_accessor out_acc{outputBuf};
```

```
  out_acc[0] = 0;
}
  myQueue.submit([&](handler &cgh) {
    accessor in_acc{inputBuf, cgh, read_only};
    accessor out_acc{outputBuf, cgh, write_only};

    range<1> globalRange{N};
    range<1> localRange{256};
    nd_range<1> ndrange{globalRange, localRange};

    cgh.parallel_for(ndrange, [=](nd_item<1> it) {
      const auto gid = it.get_global_id(0);
      atomic_ref<T,
                 memory_order::relaxed,
                 memory_scope::device,
                 address_space::global_space>
                 atm(out_acc[0]);

      atm.fetch_add(in_acc[gid]);
    });
  });
```

Listing 3.3: Atomic operations in SYCL

3.4 BENCHMARKING SYCL 2020 FEATURES

In the this section we present the experimental evaluation conducted using the benchmark described in Section 3.3.

3.4.1 *Experimental Setup*

We present the results obtained on three GPUs from three principal GPU vendors, i.e. NVIDIA Tesla V100S, AMD MI100, and Intel Max 1100.

The NVIDIA node is equipped with an Intel Xeon Gold 5218 CPU, operating at 2.30GHz and featuring 64 cores. Additionally, it incorporates an NVIDIA Tesla V100S connected via PCI Express. The GPU comprises 80 Streaming Multiprocessors, totaling 5120 cores running at a frequency of 1.245GHz, supplemented by 620 Tensor Cores. The Tesla V100S achieves 8.2 TFLOP/s FP64, 16.4 TFLOP/s with FP32, 32.2 TFLOP/s with FP16, and 130 TFLOP/s utilizing FP16 Tensor cores. The GPU has 32GB

HBM2 memory, attaining a high bandwidth of up to 1132 GB/s. It includes 128KB L1 Cache per Streaming Multiprocessor (SM), 6MB L2 Cache, and 256KB registers per SM.

The AMD node is configured with an AMD EPYC 7313 CPU clocked at 3.7GHz, featuring 16 x 2 cores. Additionally, it incorporates an AMD MI100 GPU connected via PCI Express. The GPU is comprised of 120 compute units, for a total of 7680 cores, each running at a frequency of 1.0 GHz. The MI100 GPU delivers up to 11.5 TFLOP/s FP64, 23.1 TFLOP/s FP32, and an 184.6 TFLOP/s using FP16. With 32 GB of HBM2 memory, the GPU achieves a high bandwidth of up to 1129 GB/s. Further specifications include a 16KB L1 Cache per Compute Unit, an 8MB L2 Cache, and 256KB registers.

The Intel node was provided by the CINECA consortium , it is equipped with an Intel(R) Xeon(R) Platinum 8480+ and an Intel Max 1100, connected via PCI Express. It has 56 Xe cores, for a total of 7168 cores. The max clock is 1550 MHz, achieving 22.3 TFLOP/s with FP32 and FP64 [1] It is equipped with 48 GB of HBM2E memory, with a maximum bandwidth of 1228.8 GB/s, alongside 28MB L1 cache and 108 MB L2 cache.
For the SYCL implementations, we chose Intel DPC++ (git commit sha f43cd7b) and AdaptiveCpp (git commit sha eeebfd4) which are the two main SYCL implementations at the time of writing. Additional software stack is Clang 17.0.1 (for building AdaptiveCpp), CUDA 12.1 (driver 535.129.03), ROCm 5.5.0 (driver 505.302.01), and LevelZero driver version 170.007.42. We run each experiment 10 times and take the median as the reference value.

3.4.2 *Reduction Kernel*

Figure 3.1 shows the results for the *reduction kernel* benchmark compared with the *local memory reduction* benchmark on four data types (int32, int64, fp32, fp64) and 150,000,000 elements.

The AdaptiveCpp results for the Intel Max 1100 are excluded as the feature is not currently supported.

1 Unofficial data as Intel has not released official performance statistics yet.

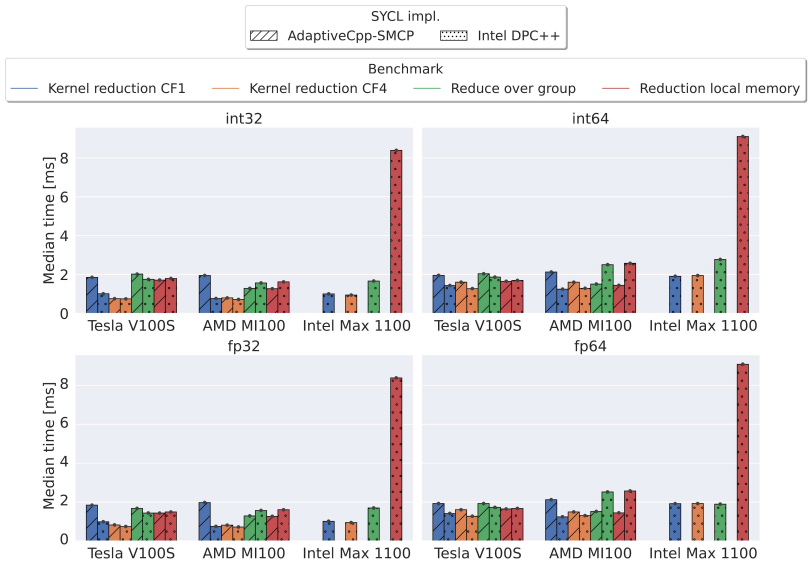


Figure 3.1: SYCL 2020 kernel reductions with coarsening factor 1 and 4 compared to *reduce_over_group* and local memory reductions

In evaluating the *kernel reduction* benchmark with a coarsening factor 1, the Intel DPC++ implementation demonstrates approximately a 2x speedup compared to the AdaptiveCpp implementation across all hardware platforms. The speedup of DPC++ over AdaptiveCpp resides in thread coarsening. AdaptiveCpp implements the kernel reduction with a tree reduction approach, launching multiple kernels until a single element is produced. Furthermore, thread coarsening optimization is left to the user since selecting the optimal coarsening factor requires parameter-tuning strategies. Differently, the DPC++ kernel reduction leverages fast atomic and the reduce_over_group function. In more detail, when using the `sycl::range parallel_for`, DPC++ adjusts the kernel grid size to spread the computation on all the available device compute units. If the number of threads specified by the user cannot be scheduled concurrently on the device resources, DPC++ applies thread coarsening in order to perform the reduction in a single-step kernel execution. The use of thread coarsening under the hood clarifies the speedup of DPC++ over AdaptiveCpp. In fact, by increasing the coarsening factor to 4, AdaptiveCpp achieves a 2x speedup compared to the version without thread coarsening showing the same performance as

DPC++. In contrast, for the DPC++ implementation adjusting the coarsening factor does not notably impact performance.

We compare the built-in reduction with a manually implemented one using local memory, inspired by [70]. On the AMD and NVIDIA GPUs the *kernel reduction* benchmark with Intel DPC++ achieves $\sim$ 2x times speedup compared to the manually implemented local memory reduction. In contrast, for AdaptiveCpp without considering the thread coarsening optimization, the two benchmarks have similar performance. The major performance improvement of kernel reduction over reduction with local memory is registered on Intel Max 1100 where we have $\sim$ 8 times speedup for all data types.

Overall the built-in SYCL reduction for both implementations can be considered a performing and easy-to-write alternative to the manually implemented reduction. However, the use of SYCL kernel reductions should not exclude user-driven optimizations such as thread coarsening. Instead, users can focus on application-specific optimizations while ignoring the reduction implementation details.

3.4.3 *Group Algorithms*

Figure 3.1 shows the performance comparison between the *reduce_over_group*, *local memory reduction* and *kernel reduction* benchmarks on four data types (int32, int64, fp32, fp64) and 150.000.000 elements. The AdaptiveCpp results for the Intel Max 1100 are excluded due to the absence of support for the `reduce_over_group` feature in its implementation. On the Tesla V100S, both implementations achieve similar performance for all the data types. On AMD MI100 both implementations have the same behavior for 32-bit data, while AdaptiveCpp with 64-bit data achieves 2 times speedup compared to DCP++. The two implementations have different ways to implement `sycl::reduce_over_group`. AdaptiveCpp implements the `reduce_over_group` in two steps. First, it applies reduction on each sub-group leveraging shuffle operation mapped to sub-group primitives depending on the target architectures. Then, the results of the sub-group reductions are loaded into local memory and the final output is computed using a local memory tree reduction. Differently, the DPC++ imple-

mentation maps reduce_over_group over SPIRV intrinsics, which are lowered to a sub-group shuffle implementation on every platform.

On Tesla V100S and AMD MI100, the `reduce_over_group` achieves similar performance to the manually implemented reduction, while on Intel Max 1100 the DPC++ shows 4 times speedup compared to the manually implemented reduction. Comparing the two possibilities of implementing a reduction using SYCL features we can notice that for AdaptiveCpp the kernel reduction with coarsening factor 1 and the *reduce_over_group* benchmark show the same performance for all the hardware and data types. Differently, in DPC++ the *reduce_over_group* benchmark is 2x slower compared to the kernel reduction, even if it is implemented using `sycl::reduce_over_group` under the hood. The reason behind this lies in thread coarsening, which is automatically applied in the case of kernel reduction with `sycl::range`. Group algorithms are available only in `parallel_for` launched with `nd_range`, therefore no automatic thread coarsening can be applied.

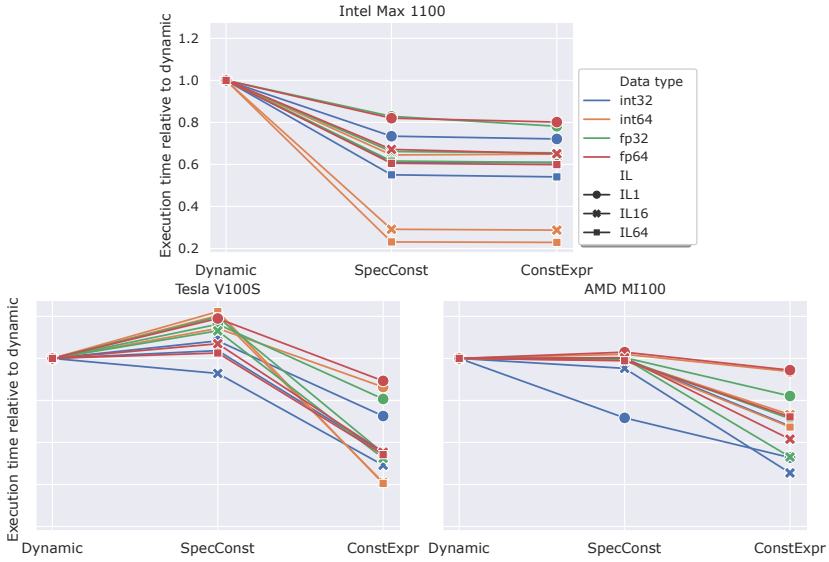


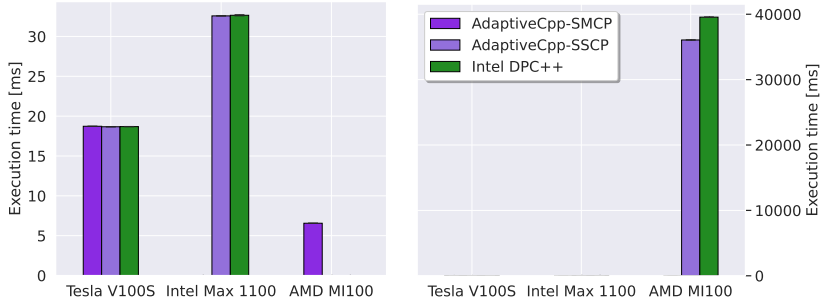
Figure 3.2: Specialization constants performance on NVIDIA V100S, AMD MI100, and Intel Max 1100 with DPC++

3.4.4 *Atomics*

The *Atomic* benchmark has been executed on 4 data types (int32, int64, fp32, fp64) with 9.400.000 elements. For AdaptiveCpp, we used both the SMCP (single-source, multiple compiler pass) and the generic SSCP (single-source, single compiler pass) [7] compilation flows as they diverge in the atomic implementation strategy.

For the AMD MI100, floating point atomic operations are simulated using a CAS loop. Support for built-in floating-point is rather incomplete: the functions have no return value, e.g., `fetch_add` return `void` instead of the previous value, and it's limited to 32-bit floating-point. The HIP toolchain exposes the *-munsafe-fp-atomics* parameter to enable fast atomic operations: it includes a compiler pass that checks if the return value of the atomic operations is used[2]. If not, issue the fast built-in, otherwise, it falls back to the CAS loop.

Figures 3.3 (a) and (b) show the execution times achieved by the *atomic* benchmark on the Tesla V100S, Intel Max 1100, and AMD MI100 GPUs using 32-bit floating-point operations.



(a) No CAS loop AMD MI100 (b) With CAS loop AMD MI100

Figure 3.3: Atomic operations performance with 32-bit floating-point.

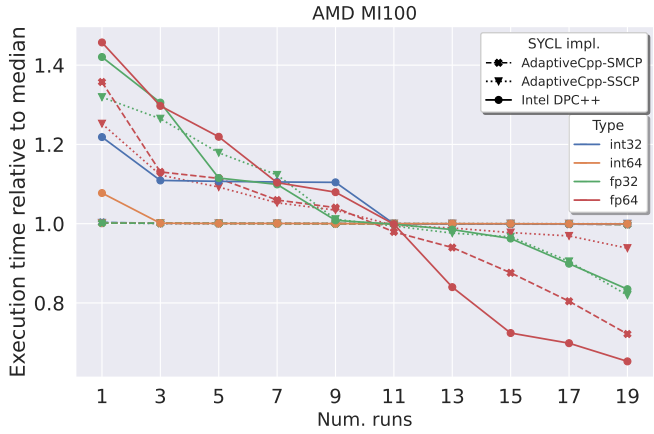
Both SYCL implementations exhibit similar performance on NVIDIA Tesla V100S and Intel Max 1100. Differently, on AMD a MI100 tomic operations for AdaptiveCpp-SSCP and DPC++ are ∼6000 times slower compared to AdaptiveCpp-SMCP (Figure 3.3a and 3.3b). Upon inspecting the AMD intermediate represen-

2 https://github.com/ROCm-Developer-Tools/hipamd/issues/19

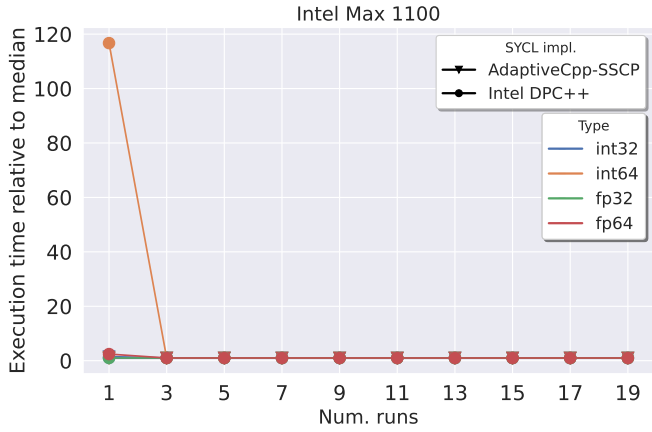
tation, we observed that for DPC++ and AdaptiveCpp-SSCP the *-munsafe-fp-atomics* option is disregarded leading the compiler to fallback on the CAS loop implementation, which drastically affects the overall performance. By analyzing the source code, we found that DPC++ skips the flag and directly handles the fast atomic built-in generation by manually checking the target architecture. On the AMD MI100, no builtin is generated as it lacks the required return value. Therefore, fast atomics are not available on the MI100 with DPC++. On the other hand, AdaptiveCpp-SSCP does not use the default clang HIP toolchain, therefore the flag cannot be interpreted. Conversely, using the unsafe option with AdaptiveCpp-SMCP the compiler correctly generates the *global_atomic_add_f32* builtin, leading to a $\sim$ 6000 times speedup compared to DPC++ and AdaptiveCpp-SSCP. The results for the 64-bit floating-point, int 32, and int 64 are omitted since they show trends similar to the one observed for fp32 atomics. Both SYCL implementations demonstrate comparable performance on all the hardware. However, the AMD MI100 achieves a 6000x slowdown on fp64 compared to other hardware due to the lack of fast atomic support.

On AMD and Intel platforms, we observed some variability in the execution time depending on the run number. Figure 3.4 illustrates the execution time of each run normalized to the median runtime across 20 runs of the benchmark on AMD MI100 and Intel Max 1100.

Looking at the AMD MI100 results for DPC++ and AdaptiveCpp-SSCP, we notice that the execution times on fp32 and fp64 curiously decrease for each run. Instead, AdaptiveCpp-SMCP time decreases only with fp64. Differently, the execution times on int32 and int64 are consistent for AdaptiveCpp-SMCP and DPC++. We guess that this behavior is related to the CAS loop, which is generated in all the results that show this variability. On Intel Max 1100, DPC++ exhibits a high overhead for the first run on int64 and fp64, due to the JIT compilation process. However, while also AdaptiveCpp-SSCP relies on JIT compilation, it does not show the same overhead, probably due to some additional optimizations happening in the DPC++ toolchain.



(a) AMD MI100



(b) Intel Max 1100

Figure 3.4: Atomic operations overhead.

3.5 SUMMARY AND DISCUSSION

In this chapter, we investigated the performance characteristics of SYCL 2020 with a particular focus on high-level abstractions and how they map onto different GPU architectures and compiler backends. We introduced SYCL-Bench 2020, the first benchmark suite specifically designed to evaluate SYCL 2020 features, including reduction kernels, group algorithms, and atomic operations. Through targeted micro-benchmarks executed on multiple GPUs from NVIDIA, AMD, and Intel, and using two major SYCL im-

plementations (oneAPI DPC++ and AdaptiveCpp), we observed that SYCL's high-level abstractions can deliver competitive performance and, in several cases, performance approaching native low-level APIs. Our results showed that reductions, group algorithms, and atomic operations can be translated efficiently into hardware-specific constructs, achieving close-to-native execution where compilers and backends are sufficiently optimized. At the same time, we observed significant variance in the SYCL compiler maturity and backend behavior. DPC++ provided the most complete and functionally correct implementation of all tested SYCL 2020 features, but performance and optimization quality differed substantially across its CUDA, HIP, and Level Zero backends. Conversely, AdaptiveCpp offered more uniform backend support but lacked support for some advanced SYCL 2020 constructs in its newer compilation stack.

Overall, our study demonstrates that SYCL 2020's high-level abstractions provide a promising foundation for cross-architecture performance portability. While comparable performance with native low-level APIs remains compiler-dependent, our results confirm that SYCL's high-level abstractions can be effectively mapped onto diverse GPU architectures and that continued compiler backend refinement will further close the remaining performance gap.

4

APPROXIMATE COMPUTING ON HETEROGENEOUS ARCHITECTURES

Faced with the end of Dennard scaling and the constraints of traditional Moore's Law-driven scaling, researchers and engineers explored other methods to preserve and improve computational performance. This demand for efficiency, along with the idea that not all applications require absolute precision, gave rise to the notion of Approximate Computing. Approximate computing is an emerging paradigm that aims to exploit the inherent error tolerance of many applications, particularly in domains such as image processing and machine learning. Taking advantage of this property, applications can trade off accuracy for significant gains in performance and power consumption. The use of approximate computation techniques presents several challenges [14, 123]. By trying to minimize error and maximize other metrics, typically performance, we are actually formulating a multi-objective problem. Since approximation accuracy and performance are not correlated, this leads to a multi-objective problem without a single optimal solution, but rather a set of Pareto optimal dominant solutions [40]. As a consequence, exploring and understanding the approximation space becomes challenging for developers, especially as the number of available techniques grows. When multiple approximation techniques, such as data perforation, mixed precision, and reconstruction, are considered together, the space of possible configurations expands dramatically, offering potential for unprecedented improvements but also increasing complexity. Most existing frameworks treat these techniques in isolation, and only a few support even partially composed approaches. Furthermore state-of-the-art approximate computing frameworks are tied to specific compiler and architectures, limiting the portability and applicability on heterogeneous systems. Contemporary systems combine CPUs and GPUs, and domain-specific accelerators, each exposing different execution models, memory hierarchies, and optimization opportunities. Applying

approximate computing consistently across such heterogeneous platforms remains difficult without appropriate high-level abstractions. Programmers need mechanisms that let them apply and explore approximation techniques without having to implement them from scratch or deal with architecture-specific details. At the same time, these mechanisms must remain expressive enough to capture rich combinations of approximations and general enough to support different vendors and hardware backends.

In this chapter, we address these challenges by introducing a new abstraction layer for approximate computing built upon the SYCL programming model. We propose SYprox, a new approximate computing interface based on SYCL that allows programmers to easily implement heterogeneous approximated applications with state-of-the-art approximation techniques such as perforation, mixed precision, and signal reconstruction. We define a new host perforation technique that fully exploit the host-device execution model of modern CPU-GPU systems, and for the first time, we provide a way to combine different techniques across heterogeneous architectures. To evaluate the effectiveness and portability of the proposed abstraction, we conduct an extensive experimental study on eight benchmarks and 100 datasets, using GPUs from three major vendors: AMD, Intel, and NVIDIA. We compare SYprox both to standalone techniques from prior work and to state-of-the-art frameworks such as HPAC [53, 134] and the method proposed by Maier et al. [114, 116].

4.1 RELATED WORK ON APPROXIMATE COMPUTING

Over the years, several researchers have explored approximate computing through custom hardware components or innovative software approaches. In this chapter, we focus exclusively on software-based techniques, and this section provides an overview of the most relevant approximate computing methods and frameworks developed to date.

Mixed Precision leverages different levels of numerical precision within the same program to reduce computational workload, memory usage, and energy demands. Instead of uniformly using high-precision types (e.g., double precision), mixed-precision

methods selectively apply lower-precision representations (e.g., single, half, or fixed-point) in computations that can tolerate reduced accuracy. Tools such as Precimonious [157] perform automated program analysis to determine which variables or operations can safely be reduced in precision while maintaining a user-defined accuracy threshold. CRAFT [98] uses binary-level instrumentation to convert double-precision instructions to single precision in existing programs. TAFFO [28] extends LLVM compiler passes to assist programmers in precision tuning, relying on metadata and annotations to guide the transformation of variables and operations to lower precision. Mixed precision is particularly effective for numerical and GPU-accelerated workloads, providing significant performance gains while introducing minimal accuracy loss.

Loop Perforation reduces execution time by skipping selected iterations within computational loops. Instead of executing all iterations, loops are perforated according to patterns such as fixed-step (modulo), random, or truncation strategies. Sidiroglou et al. [153, 174] introduced a criticality-based approach to identify loops that can tolerate perforation, generating tunable sets of loops and skip factors that maximize performance for a given accuracy bound.

Data Perforation extends the perforation concept from loops to the data itself, selectively skipping subsets of input or output data rather than iterations in the control flow. Lou et al. [111] applied data perforation in image processing pipelines, showing that skipping portions of image pixels can reduce computation time while maintaining visually acceptable output. They complemented perforation with output reconstruction techniques based on interpolation to recover skipped computations and minimize the resulting error. Figurnov et al. [52] applied similar ideas to Convolutional Neural Networks (CNNs), reducing the number of processed feature map elements in convolutional layers, which lowers computational cost and memory access while maintaining network accuracy without retraining.

Memoization leverages the reuse of previously computed results to reduce redundant computations and improve performance and energy efficiency. Mishra et al. [121] introduced iACT, a hardware-software toolkit that applies memoization alongside

precision reduction. In iACT, programmers annotate functions for approximation, and the run-time framework maintains a table of previously computed inputs and outputs. When a new function call receives similar inputs, the system reuses the stored result instead of recomputing it, reducing computational cost and energy consumption. Tziantzioulis et al. [187] proposed a complementary approach that exploits *temporal locality* rather than explicit similarity functions. In this technique, the output of a function is approximated based on recently computed values, avoiding the overhead of similarity checks for every new call.

4.2 MOTIVATION

To support the adoption of approximation techniques, the community has proposed a wide range of frameworks that operate at different levels of the software stack. However, these solutions vary widely in scope, generality, and usability, leaving programmers without a unified and portable way to explore approximations, especially on modern heterogeneous architectures. In the following section , we discuss the main categories of existing approaches and highlight the gap that motivates the work presented in this chapter.

MANUAL APPROACH Approximate computing techniques can be applied directly by the programmer without any assistance from compilers or specialized programming models. Maier et al. [115, 116] define a local memory-aware approximation approach based on loop perforation that approximates the input data of GPU applications by reducing the amount of data loaded from global memory and reconstructing a high-accuracy approximation with local reconstruction techniques.
Limitation: While manual techniques enable fine-grained control, they require deep expertise, significant engineering effort, and are difficult to port across architectures. These limitations highlight the need for abstractions that make the approximation accessible without the complexity of manual implementations.
Our contribution: With respect to manual approaches, our work proposes a library-based framework that automatically applies

approximations, eliminating the need for manual implementation by programmers.

COMPILERS Although manual approaches method can yield significant performance gains, it places a heavy burden on developers and is error-prone due to the lack of automated tools. Several works have focused on integrating approximate computing techniques directly into compilers. These compilers can automatically identify opportunities for approximation and apply transformations that improve performance, such as reducing computational precision [28, 35, 95, 98, 157, 166], loop perforation [13, 73, 105, 111, 122, 163, 165], or relaxing memory consistency [23, 102, 151]. Paraprox [163] is a software solution applicable to commodity hardware that automatically discovers and approximates common patterns such as map, scatter/Gather, reduction, and others. For each pattern, Paraprox provides an approximation strategy with one or more knobs that can be used to navigate the performance-accuracy trade-off. Lou et al. [111] presented a new prototype language and compiler that applies loop perforation and output reconstruction only to the image processing pipeline. Laguna et al. [95] propose GPUMixer a tool to tune the data precision of applications on GPUs. Although common mixed-precision approaches change the precision of variable declarations, a fine-grained approach is to express the precision of each floating-point operation in the program. GPUMixer provides a practical method to select the computations to be performed on FP32 or FP64 precision so that user-defined accuracy constraints are maintained and performance is significantly improved. By automating the application of approximations, compiler-based approaches reduce the complexity of adopting approximate computing techniques.

Limitation: With fully automatic approaches, programmers are required to trust the system, with no way to intervene when opportunities are overlooked or invariants are compromised.

Our contribution: Our work abstracts compiler-level approximations to the library level, without requiring novel programming model prototypes and specialized compilers. Furthermore, our approach applies to general-purpose applications, enabling approximation beyond a specific domain.

LANGUAGE EXTENSIONS Programming models have also been extended to support approximate computing by providing abstractions and frameworks that allow developers to specify approximate behaviors more easily [53, 189, 190] . These models often provide APIs, annotations, or directives that enable programmers to define where and how the approximation should occur. This simplifies the development process by integrating approximate computing into higher-level programming constructs, offering a more structured and less error-prone approach than manual or compiler methods. Parasyris et al. [134] propose HPAC, a framework that extends the OpenMP programming model in order to provide approximate computing techniques. This approach incorporates pragma-based annotations composable with standard OpenMP annotation, code transformations, and, analysis to enable developers to identify approximation opportunities in their applications. HPAC facilitates the integration of loop perforation and memoization, offering a mechanism to investigate the accuracy-performance trade-offs for a given application. This system suggests potential approximations that can enhance performance while maintaining the accuracy levels defined by the user. The framework has been extended to enable approximate computing techniques also on GPUs [53] by extending the OpenMP annotation for GPU offloading. Despite their benefits, directive-based approaches based on OpenMP remain constrained by the compilers that interpret them.

Limitation: While these approaches improve usability, their directive-based design inherently relies on specific compiler implementations. As a result, their effectiveness and portability depend on compiler support, which can vary widely across platforms and architectures, limiting their applicability in heterogeneous environments. This limitation motivates the need for a library based approximation support that does not depend on specialized compiler implementations.

Our contribution: With respect to the state of the art, we propose SYprox the first header-only library in SYCL that implements data perforation, reconstruction, and mixed precision allowing programmers to define configurable and heterogeneous approximated application.

Table 4.1 compares the capabilities of several related works and SYprox. By comparison, our proposed solution introduces an innovative library-based method for approximate computing in heterogeneous architectures, implementing host/device perforation, reconstruction, and mixed precision.

Table 4.1: Comparison against state of the art

	Approach	*Perforation*	*Reconstruction*	*Mixed Precision*
Maier et al. [114]	manual	device	input	✗
Paraprox [164]	compiler	loop	output	✗
HPAC [53]	compiler	loop	✗	✗
GPUMixer [95]	compiler	✗	✗	✓
Lou et al. [111]	compiler	image*	output	✗
SYprox[24]	library	host, device	input, output	✓

*Lou et al. apply device perforation only to images.

OVERVIEW SYprox is a header-only library designed to extend the SYCL programming model with advanced approximate computing techniques, including perforation, reconstruction, and mixed precision. This extension enables developers to integrate approximations into their applications with minimal code modifications. Figure 4.1 illustrates a SYCL code enriched with approximate computing features and shows the combination of applied approximations. SYprox provides an easy-to-use and highly customizable interface. Programmers can implement approximations using built-in parameterizable schemes (prow, pcol) for perforation (pbuffer, paccessor) and reconstruction (input and output), develop custom schemes tailored to specific applications, or mix different data types for mixed precision computing. SYprox introduces a novel technique called *host perforation*, which selectively perforates data on the host side before transferring them to the device, thus optimizing both computation and communication. In Figure 4.1 the host data are perforated and reduced to the `half` data type before data transfer. The data on the device are processed in a perforated shape using *device perforation* (`paccessor`).

The perforated elements are then reconstructed according to the selected schema (`nn_out`) and sent back to the host. The ability of SYprox to combine different approximations increases the range of feasible configurations by generating new trade-offs between performance and accuracy (Section 4.5). The flexibility of the SYprox interface not only facilitates the adoption of approximate computing techniques across heterogeneous architectures but also expands the approximation domain by enabling new Pareto-optimal configurations.

SYprox API

```
range<2> r = range<2>{N,N};
pbuffer<half,2,prow<half,2>> inBuf(a,r);
pbuffer<float,2,prow<float,1,nn_out>> outBuf(b,r);
prange<> gl{N,N}, ws{32, 32};
q.submit([&](handler &h){
 paccessor<float,2,prow<float>> in_acc{in,h,read};
 paccessor<float,2> out_acc{out,h};
 h.parallel_for(gl,[&](id<2> id){
     out_acc[id] = perf_acc[id] * 2;
  });
}
```

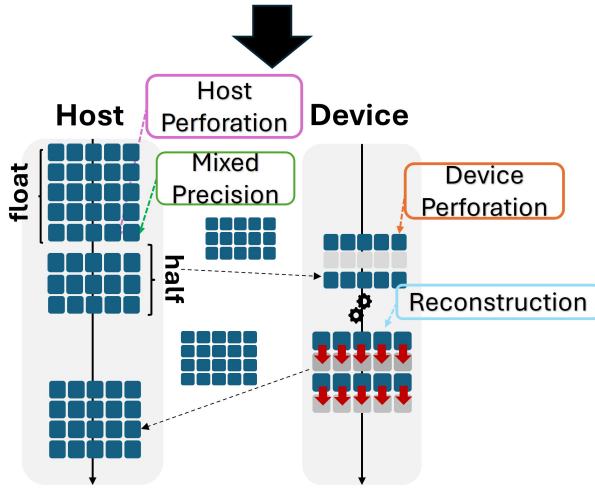


Figure 4.1: Overview of SYprox approximation

4.3 SYPROX PROGRAMMING INTERFACE

SYCL Programming Model

```
1 queue q(gpu_selector_v);
2 buffer<int, 2> outBuf(out,range<2>(N,N));
3 q.submit([&](handler& h) {
4  accessor outAc(outBuf,h);
5  h.parallel_for(r,[=](id<2> i) {
6   outAc[i] += outAc[i] * 2;
7  });
8 });
```

Listing 4.1: SYCL accurate code

The SYprox library is based on SYCL a single-source C++ programming model designed to improve the performance and portability of applications running on heterogeneous architectures, such as CPUs, GPUs, and other accelerators.
Listing 4.1 shows a simple SYCL code. The `parallel_for` (line 5) define a kernel code to be executed on a device such as a GPU. The device on which the kernel code is executed is represented by a `queue` (line 1). SYCL offers two methods for handling data transfers between the host and the device: a pointer-based strategy called Unified Shared Memory (USM) and the `buffer/accessor`. Our work leverages `buffer` and `accessor` but can be easily extended to USM. Buffers (lines 2) abstract memory management and represent a range of memory that can be used on either the host or the device. Accessors (lines 5) are used to specify how to access the data within a buffer (e.g read or write). Differently from USM, with `buffer` and `accessor` the SYCL runtime automatically manages data movements between the host and the device. The next sections describe in detail how SYprox extends the `buffer` and `accessor` of SYCL to enable approximate computing features.

Data Perforation

The SYprox library implements two types of data perforation: the *device perforation* defined by Maier et al. [114] and a novel approach called *host perforation*.

4.3.0.1 *Device Perforation*

```
1 // buffers ...
2 q.submit([&](handler &h){
3  paccessor<float,2,prow<float,2>> inAcc{inBuf,h};
4  paccessor<float,2,prow<float,2>> outAcc{outBuf,h};
5  h.parallel_for(prange<>{N,N},[&](id<2> id){
6     outAcc[id[0]][id[1]] = inAcc[id[0]][id[1]]*2;
7  });
8 }
```

Listing 4.2: SYprox code with device perforation

The *device perforation* is implemented as an extension of the SYCL `accessor` class. The `paccessor` adds a new template parameters to the SYCL `accessor` which specify how the data are accessed on the device. The class implements an on-line perforated access to the data by overloading the subscript operator ([]) so that, while all original data remain in memory, only specific parts are accessed according to the selected perforation schema (Section 4.3.0.3). Using a row schema and skip factor of x the access to `a[i][j]` is translated into `a[i*x][j]`. Listing 4.2 shows a SYprox code with device perforation. Lines 3-4 define two bidimensional perforated accessors using a row schema with a skip factor of two. In line 6 according to the row schema, the data will be accessed in a perforated shape. The access to the element at the index `{id[0],id[1]}` translates in `{id[0]*2, id[1]}`. The `prange` semantic (line 5) defines a set of deduction rules to automatically infer the `range` of the kernel according to the perforation schema used.

4.3.0.2 *Host Perforation*

```
1 range<2> r{N,N}
2 pbuffer<float,2,prow<float,2>> inBuf(in,r);
3 buffer<float,2> outBuf(out,r);
4 q.submit([&](handler &h){
5  // accessors...
6  h.parallel_for(prange<>{N, N},[&](id<2> id){
7   outAcc[{id[0]*2,id[1]}]=inAcc[id]*2;
8  });
9 }
```

Listing 4.3: SYprox code with host perforation

SYprox implements *host perforation* by defining a `pbuffer`, which extends the SYCL buffer. The `pbuffer` introduces new template parameters into the SYCL buffer to specify the type of perforation scheme to be used (Section 4.3.0.3). The `pbuffer` intercepts the SYCL `buffer` constructor, applying the perforation to the data before the invocation of the `buffer` constructor. This process involves iterating over the elements passed to the `pbuffer`, perforating data based on the approximation strategy defined by the `ApproxSchema` class (e.g. `prow`, `pcol`). Listing 4.3 shows a SYprox code with *host perforation* and a row scheme (line 2).

4.3.0.3 *Perforation and Reconstruction Schemes*

SYprox `pbuffer` and `paccessor` class perforate and reconstruct data according to a perforation and reconstruction schema defined by a specialization of the SYprox `ApproxSchema` class. SYprox defines for the `pbuffer` and the `paccessor` three built-in schemes configurable by the type of data used and the number of data to skip (`skip_factor`).

Strided scheme skips a fixed number of data points in a consistent stride. For example, with a skip factor of 2, every other data point is skipped.

Row scheme applies to bi-dimensional data. Skip entire rows of data according to the defined skip factor.

Col is similar to the row scheme but operates column-wise.

For each schema, SYprox also provides input and output reconstruction strategies of two types: *nearest neighbor* where perforated elements are reconstructed using the neighbor element;

lerp where perforated elements are reconstructed with a linear interpolation of two or more elements. Both reconstructions are implemented in an optimized way leveraging the SYCL sub-group and group algorithms. The SYprox `ApproxSchema` class is designed with flexibility in mind to serve as a base class for customizing data perforation methods according to the specific use case. Programmers can define any kind of static approximation schema by implementing the method defined by the `ApproxSchema` class.

```
1 range<2> r = range<2>{N,N};
2 pbuffer<float,2,prow<float,2,nn_out>> inBuf(in,r);
3 q.submit([&](handler &h){
4  paccessor<float,2> inAcc{inBuf,h,read_write};
5  paccessor<float,2> outAcc{outBuf,h,read_write};
6  h.parallel_for(prange<>{N,N},[&](id<2> id){
7   outAcc[id] = inAcc[id] * 2;
8  });
9 }
```

Listing 4.4: SYprox code with host perforation and output reconstruction

Listing 4.4 shows a SYprox code that combines host perforation with a row perforation schema and output reconstruction with the nearest-neighbor reconstruction schema (line 2).

Mixed Precision

```
1 std::vector<float> in;
2 pbuffer<half,2> inBuf(in,range<2>{N,N});
3 buffer<float,2> outBuf(out,range<2>{N,N});
4 q.submit([&](handler &h){
5  // accessors...
6  h.parallel_for(prange<>{N, N},[&](id<2> id){
7   outAcc[id] = inAcc[id]*2;
8 });
9 }
```

Listing 4.5: SYprox code with mixed precision

Listing 4.5 shows a mixed precision computation with `float` and `half` data type with SYprox. The `float` data in the `in` vector are converted in `half` precision during buffer construction through

the `sycl::reinterpret` function. Then the kernel performs a mixed precision computation on the `float` and `half` data. The library supports all the lower precision formats defined in the SYCL standard and also other formats such as `bfloat16`, which are implemented as experimental extensions in DPC++ [77].

4.4 HOST PERFORATION

This section presents *host perforation* a novel data perforation technique implemented in SYprox. Figure 4.2 shows a comparison between the traditional *device perforation* and the *host perforation* approach.

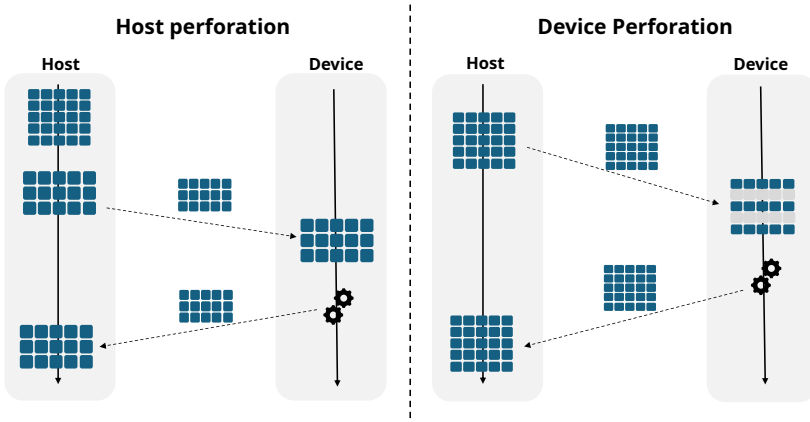


Figure 4.2: Host and device perforation approach.

Host perforation performs data perforation on the host before sending the data to the device. In contrast, device perforation requires transferring all data from the host to the device, where they are then accessed in a perforated shape according to a pattern defined by the SYprox approximation schema (Section 4.3.0.3). The *host perforation* offers two distinct advantages over the device perforation. Firstly, it significantly reduces the amount of data transfer needed between the host and the device. This reduction in data transfer can lead to improved performance in applications where data movement is a bottleneck. Secondly, host perforation provides a better cache utilization, since eliminates data access issues due to the data layout. By perforating the data on the host, accesses to the device data can be performed continuously,

maximizing cache utilization (Section 4.6.3). However, in *host perforation*, once the data have been perforated in the host, they cannot be used on the device. This limitation may affect scenarios where devices require direct access to perforated data for further processing or computations. Moreover, host perforation is only beneficial when the time required to perforate the data on the host is less than the time needed to transfer the full dataset to the device. In our experiments (Section 4.6), we tested data sizes up to the maximum available and did not observe cases where device perforation was more efficient. However, this may not hold for systems with higher host-device memory bandwidth or hosts with lower compute capabilities.

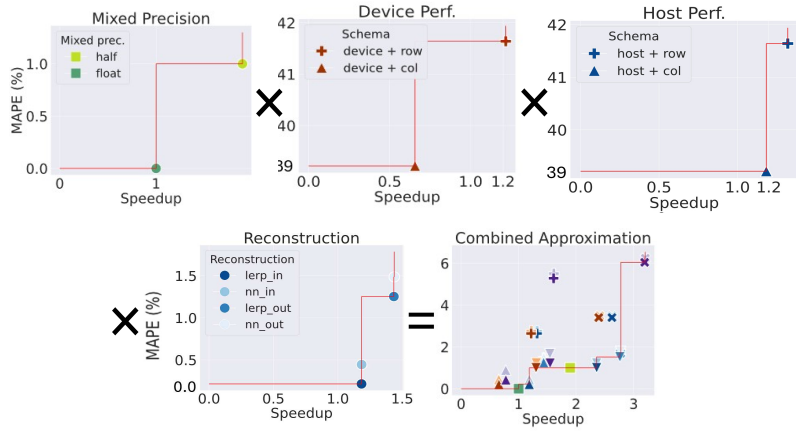


Figure 4.3: Individual and combined approximation

4.5 COMBINED APPROXIMATION

This section demonstrates how the integration of various approximation techniques can lead to improvements in both the accuracy and efficiency of applications. Figure 4.3 illustrates the speedup and error for the individual approximations and how they can be combined to generate new performance and accuracy trade-offs using a blur filter application as an example. To perforate the data, we applied a skip factor of two, reducing the number of rows and columns by half for row and column schema, respectively. The red line represents the Pareto frontier: a set of optimal solutions where no solution can be improved

without degrading another objective. Here, the Pareto front helps to identify trade-offs between performance and error.

In the following sections, we provide insight into the behavior of individual approximation techniques and their combination.

Individual Approximation

SYprox implements five approximation techniques: Mixed Precision, Device and Host Perforation, and Reconstruction. Each approximation can be represented as a set of different configurations.

Mixed Precision. For mixed precision, we used floating point data as a baseline and half precision as a lower precision data type ($Mp = \{ \textit{floating}, \textit{half} \}$). With the *half* configuration we mix floating-point and half data types. Unlike data perforation, which skips data processing, mixed precision results in only a 1% error, as it only processes the entire dataset with reduced precision.

Device Perforation. The possible configurations with device perforation depend on the number of schemes implemented. In our experiment, we used the row and column schemes with a skip factor of two, resulting in two configurations ($Dp = \{$row, col$\}$). Data perforation without reconstruction leads to an error of approximately 50% since for skip factor of two half of the data are perforated. Furthermore, the column schema exhibits a significant performance slowdown due to increased cache misses caused by the data layout.

Host Perforation. The host approach shares the same number of configurations as the device perforation. Both approaches result in the same error, since they operate on the same data. However, the host approach achieves a higher speedup due to reduced data transfer and optimized data layout for the column schema.

Reconstruction. SYprox provides two types of reconstruction schemes nearest-neighbor (`nn`) and linear interpolation (`lerp`) both implemented with input and output reconstructions, resulting in four different configurations ($R = \{$`nn_in`, `nn_out`, `lerp_in`, `lerp_out`$\}$). Nearest-neighbor reconstruction is faster but less accurate since the reconstruction only uses one of the computed elements to approximate the perforated data , while linear interpolation provides higher accuracy with only a minor

impact on speedup due to the time spent in the interpolation. Input reconstruction involves reconstructing data before computation, resulting in less error but lower performance. Output reconstruction, on the other hand, reconstructs data after computation, which usually leads to higher error but also higher speedup.

Approximation by Combining Techniques

The SYprox interface provides a way to combine individual approximations. When combining these techniques, the number of available configurations is equal to the cartesian product of each individual approximation: $|Mp| \times |Dp| \times |Hp| \times |R|$, resulting in an approximation space composed of 32 points. Combining different approximation techniques allows us to expand the approximation domain and explore new performance-accuracy trade-offs. In fact, Figure 4.3 (Combined Approximation) shows that we can achieve a 3x speedup with a maximum error of 6%. However, not all combinations yield efficient results. For example, any combination that applies *device perforation* with a column schema typically shows lower performance due to cache misses related to the type of data layout. Composing different approximations can also increase the error. However, signal reconstruction techniques can help mitigate the error by generating new Pareto-optimal solutions. Notice that the number of available configuration can also be higher since we can have more data types (e.g. `bfloat16`) or for perforation schemes `skip_factors` higher then two.

4.6 EXPERIMENTAL EVALUATION

In this section, we present an analysis to assess the efficacy of approximate computing techniques comparing SYprox with state-of-the-art approaches [53, 114, 134].

4.6.1 *Experimental Setup*

4.6.1.1 *Benchmark Description*

We conducted the experimental evaluation on eight benchmarks described in Table 4.2.

Table 4.2: Applications used for experimental evaluation

Benchmark	Domain	Size	Kernels' LoC
median	Medical imaging	3072^2	45
sobel	Edge detection	3072^2	40
blur	Image blurring	3072^2	21
tv	Edge detection	3072^2	18
gaussian	Image blurring	3072^2	34
hotspot	Phisical simulation	4096^2	150
lavaMD	Molecular dynamics	128^3	219
leukocytes*	Medical imaging	219x640	281

*leukocytes consists of 3 kernels that process several frames.

The benchmarks were selected to have a direct comparison of the proposed host perforation approach with the device perforation method described by Maier et al. [114]. We implemented the applications in SYCL using the SYprox library to apply our approximation. To ensure a direct comparison with the HPAC framework [53, 134], all benchmarks were ported from SYCL to OpenMP with GPU offloading and then integrated into the framework using specific HPAC directives for approximation.

The image processing benchmarks are executed on a dataset composed of 100 images of 3072^2 size to analyze the error variation of each approximation on different inputs. The input data sets are taken from the USC-SIPI Image Database [199]. For `lavaMD` and `hotspot` we randomly generated 100 input files. The speedup is calculated using the accurate application as a baseline,

while the error uses the mean absolute percentage error, defined as MAPE $= \frac{100\%}{n} \sum_{i=1}^{n} \left| \frac{I_i - \hat{I}_i}{I_i} \right|$, where I_i and $\hat{I}_i$ are the accurate and approximated data, respectively.

4.6.1.2 *Parameter Description*

In all experiments, we executed host and device perforations with row and column schemes using a skip factor of 2, while for mixed precision, we adopted floating and half-data types. The half-configuration mix floating point and half-data types. For the reconstruction step, we applied the input (`in`) or output (`out`) reconstruction with the nearest neighbor (`nn`) or linear interpolation (`lerp`). Benchmarks implemented with local memory use a size defined by the block size, which in our case is fixed to 32x32.

4.6.1.3 *Software and Hardware Configuration*

For the experimental evaluation of our approach, we rely on Intel DPC++ [78] SYCL compiler and the one provided by HPAC [53] for OpenMP. All SYCL and OpenMP codes have been compiled using the *-O3* flag. We performed our experiments on three separate nodes, each equipped with: an AMD EPYC 7313 CPU and AMD MI100 GPU; an Intel Xeon Platinum 8480 CPU and an Intel Max 1100 GPU; an Intel Xeon Gold 5218 and NVIDIA V100S.

4.6.2 *Error Analysis*

Figure 4.4 shows the MAPE for each benchmark executed with host perforation and input/output reconstruction on 100 different data sets. For `lavamd`, `hotspot`, and `leukocytes`, we only show the results for the column schema as they have been implemented using one-dimensional buffers and a one-dimensional perforation scheme that skips every other element, effectively corresponding to a column schema in two dimensions. The error analysis is not affected by the type of perforation applied (host or device), as both techniques perforate the same elements, leading to identical errors. For this reason, our analysis focuses only on *host perforation*. The results highlight that the amount of error

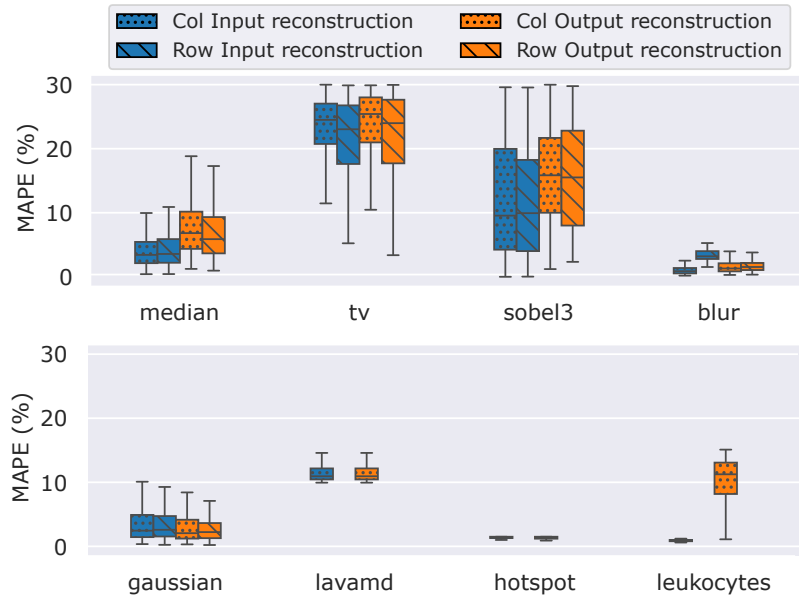


Figure 4.4: Error of different approximate strategies.

depends on type of computation performed by the application, input data, perforation schema applied (row, col) and the type of reconstruction (input, output).

Computation. The `tv` and `sobel` benchmarks exhibit an error of up to 30% compared to the 1-15% error range of the other applications. This variation is due to the type of computation performed by each application. Applications based on average and median calculations (`gaussian`, `blur`, and `median`) are less affected by data perforation compared to the `sobel` and `tv` filters.

Input Data. The error introduced by data perforation is also correlated with the type of input. Applying perforation and reconstruction on inputs with higher data similarity produces a lower error. Looking at the column schema results for `leukocytes` and `guassian` the MAPE is in the range 1-15% while for `blur` 3-5%. The variation in error is much clearer on the `sobel` and `tv` benchmark, where MAPE varies in the range 5-30%.

Perforation Schemes. The error is also affected by the type of schema used. As an example, the blur application shows an error between 3-5% for the row schema and 1-2% for the column schema.

Reconstruction. The input reconstruction usually achieves a lower error compared to the output reconstruction. With the input approach, the perforated data are reconstructed prior to computation. Consequently, the computation uses the same number of data as the accurate application, resulting in a lower error. For all applications, the input reconstruction has a lower or similar error compared to the output reconstruction. For instance, in the `leukocytes` and `sobel` application, output reconstruction leads to an error of 15% and 30%, while the input reconstruction keeps it below 5% and 20%.

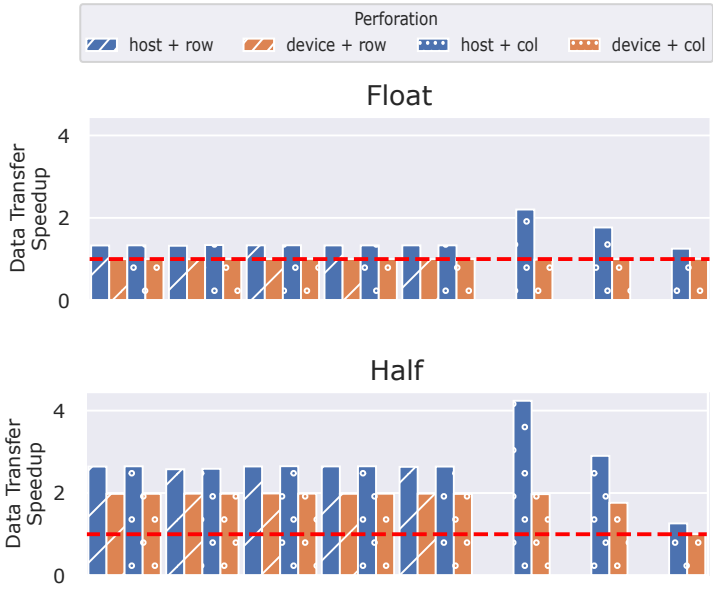
To reduce the number of configuration in each plot, we only show results using a `skip_factor` of two. However, we conducted additional experiments to explore how the variation of the `skip_factor` affects both performance and error. As the `skip_factor` increases, performance improves approximately linearly-typically, a skip factor of X yields an X-fold speedup. In contrast, the error does not increase linearly, as it is influenced by multiple factors discussed above.

4.6.3 *Host vs Device Perforation*

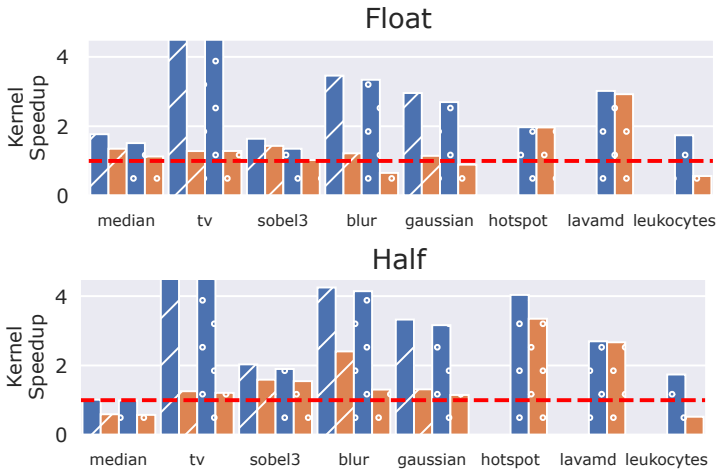
Here, we focus on a performance comparison of the *host perforation* implemented in SYprox with the *device perforation* defined by Maier et al. [114]. Figure 4.5 illustrates the performance improvement in data transfer and kernel computation for each application compared to accurate execution (red dashed line) for both host (blue) and device (orange) perforation. The dashed and dotted hatch represent the row and column perforation schemes, respectively.

Host perforation consistently outperforms device perforation in terms of data transfer speedup, as shown in Figure 4.5a. This is because host perforation reduces data transferred to the device, achieving a speedup of 1.3x to 2x for floating-point data and 2.5x to 4.2x for half-precision data. In contrast, device perforation offers no such improvement, as all data are sent to the device.

Looking at the kernel speedup results in Figure 4.5b, host perforation outperforms device perforation in all applications, with the exception of `lavamd` and `hotspot`, where both methods achieve the same speedup. Host perforation speedups range from



(a) Data transfer speedup.



(b) Kernel speedup.

Figure 4.5: Data transfer 4.5a and kernel 4.5b speedup of all applications for host/device perforation and float/half precision. The red line represents the baseline defined as the accurate execution.

1.7x to 4.2x, while device perforation range from 0.5x to 3.5x. The primary reason for the lower performance of device perforation is related to increased cache misses, which significantly impact its efficiency, in particular for the column schema. Profiling the applications with *NVIDIA Nsight Compute* we can notice an L2 cache hit rate of 76% for *host perforation* against the 49% of the device approach. In device perforation, the perforated data are still in memory, while accesses are performed in a perforated shape. Therefore, with a column schema, more than half of the data loaded into the cache are never used during computation, increasing the number of cache misses. In contrast, with *host perforation*, we achieve comparable performance for both schemes, since the data are reorganized in memory to avoid the load of unused data in the cache. For the column schema, this issue is highlighted in `blur`, `leukocytes`, and `gaussian`, which achieve speedups of 0.5x, 0.6x, and 0.75x, respectively, resulting in a slowdown compared to the accurate version, while host perforation reaches a speedup of up to 2x. For the row schema, `gaussian`, `tv`, and `blur` show a similar behaviour since each kernel thread processes a 6x6 filter, leading to the same access problem of the column schema. In contrast, for `sobel` and `median`, which use a smaller 3x3 filter, device perforation achieves similar performance compared to host perforation due to the lower number of cache misses. The slowdown related to cache misses is mitigated using the `half` data types, since with reduced precision, we can store more data in the cache, resulting in a lower number of cache misses.

4.6.4 *SYprox vs HPAC*

In this section, we conducted a multi-objective evaluation to analyze the trade-offs between speedup and error in the SYprox and HPAC frameworks. All SYprox benchmarks were ported from SYCL to OpenMP with GPU offloading and subsequently integrated into the HPAC framework. We tested HPAC applications with two loop perforation approaches: *small* skips one iteration for every n iterations; *large* executes one iteration for every n iterations. The `hotspot` results with HPAC are unavailable because using the perforation pragma defined by HPAC causes the application to run indefinitely. Figure 4.6 illustrates multi-

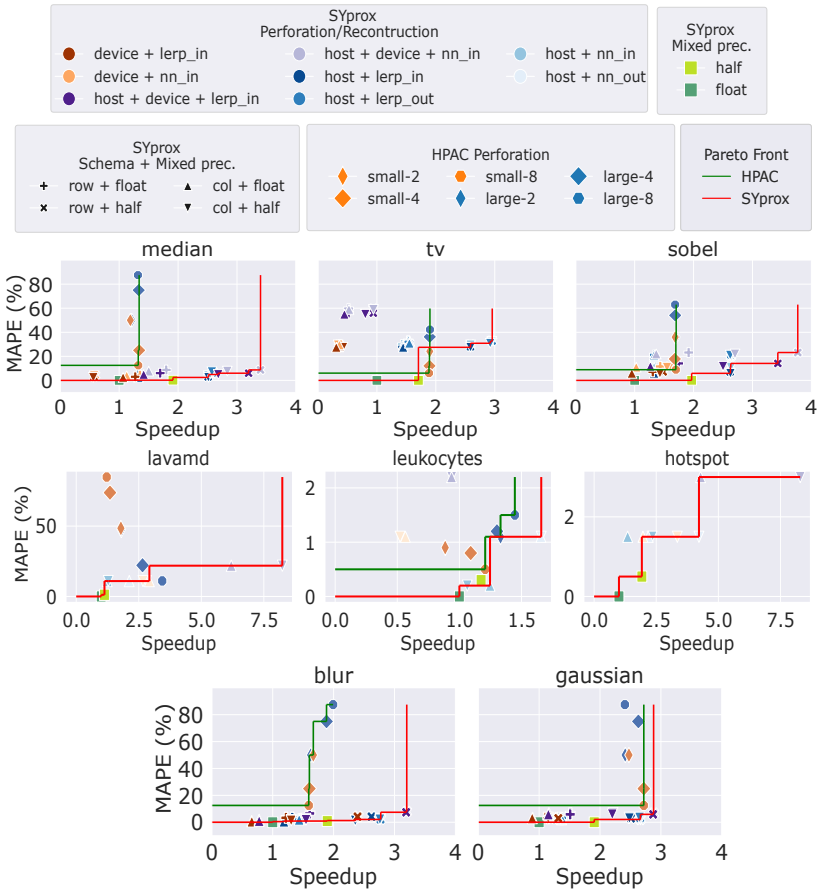


Figure 4.6: SYprox vs HPAC. The colors represent perforation and reconstruction techniques. Markers define the combination of schemes and data types. The green and red lines represent the HPAC and SYprox Pareto frontier.

objective plots for the 8 applications comparing the SYprox and HPAC frameworks. The x-axis represents the speedup, while the y-axis the Mean Absolute Percentage Error (MAPE). Points located in the bottom right corner of the plot are preferable as they indicate better performance with a lower error. For SYprox, the color of markers with different shades of red and blue represent the device and host perforation approaches, respectively, while the violet point represents the combination of host and device perforation. The different shades of a color indicate different reconstruction methods: nearest-neighbor or linear interpolation

with input or output reconstruction (`nn_in`, `nn_out`, `lerp_in`, `lerp_out`). Markers are utilized to differentiate between combinations of perforation schemes (row, column) and data types (float, half). Within the HPAC setup, the blue markers denote configurations that employ loop perforation of type *'large'*, while the orange ones employ loop perforation of type *'small'*. The different markers in the HPAC framework correspond to different skip factors. The red and green lines represent the Pareto frontier of SYprox (P_S) and HPAC (P_H), respectively.

Multi-objective Analyses. For all benchmarks, the HPAC configurations show a higher error range compared to SYprox, due to the lack of reconstruction techniques. For the `gaussian` benchmark, it achieves up to 3x speedup at the cost of the 80% error, while for the other benchmarks, the speedup is between 1.5x and 2x with an error in the range of 10-60%. On the other hand, SYprox has multiple configurations involving mixed precision and perforation/reconstruction with different schemes. This diversity allows for a wide range of trade-offs, potentially making it more flexible in tuning performance versus accuracy. Most the SYprox configurations achieve an error less than 20% while achieving a nearly 4x speedup for all benchmarks. Furthermore, except for the `tv` benchmark, the Pareto frontier of SYprox always dominates the one of HPAC. This implies that for any given range of error or performance, SYprox offers configurations that are at least as good as, and often better than, those offered by HPAC.

Hypervolume Analyses. Table 4.3 shows different metrics to compare the configuration of HPAC and SYprox. $|P_H|$ and $|P_S|$ represent the number of points in the Pareto set of HPAC and SYprox, respectively. For all benchmarks, we can notice $|P_S| >= |P_H|$ meaning that the SYprox framework generates more Pareto optimal solution compared to HPAC. In multi-objective optimization, the hypervolume metric [210] (HV) is used to evaluate the performance of a Pareto front by calculating the volume of the space in the objective domain that is dominated by the Pareto front, up to a reference point. A larger hypervolume indicates better performance, as it means that the Pareto front spans a larger region of the objective space, representing more optimal trade-offs between the objectives. In our case, we are interested in the coverage difference between two sets: the Pareto set P calculated

considering the configuration of both SYprox and HPAC; and the Pareto set calculated only considering the SYprox configurations P_S. Therefore, we use the binary hypervolume metric [172], which is defined as $D(P, P_S) = HV(P) - HV(P_S)$, where $HV(P)$ and $HV(P_S)$ represents the hypervolume of the P and P_S Pareto frontier. In our experiment, the reference point for each benchmark has been selected according to the analysis provided by Ishibuchi et al.[80] in order to have accurate hypervolume results. Looking at the hypervolume values the SYprox Pareto front always covers a larger region of the objective space compared to HPAC. This observation indicates that the SYprox approximations are distributed along the speedup and error axes in a way that encompasses wider levels of performance and accuracy. The coverage difference ($D(P, P_H)$) shows that for all the applications the Pareto frontier of SYprox dominates the one of HPAC. The only exception is the `tv` and `lavamd` benchmarks, where the coverage difference is non-zero because there are two and one points, respectively, in P_H that have no counterparts in the SYprox Pareto front that outperform them in both dimensions.

Table 4.3: Evaluation of HPAC and SYprox Pareto fronts

Benchmark	$\|P_H\|$	$\|P_S\|$	HV_H	HV_S	$D(P, P_H)$
median	3	6	0.1	5.9	0
sobel	2	5	0	18.7	0
blur	3	7	3.3	5.7	0
tv	3	5	1.7	14.6	1.2
gaussian	3	5	2.2	3.8	0
leukocytes	3	3	1.4	1.8	0
lavaMD	1	5	3.05	6.8	0.07
hotspot	-	4	-	8.1	-

4.6.5 *Approximation Space Evaluation*

Figure 4.7 shows the speedup and error of SYprox in comparison to the Maier et al. approach, highlighting several key advantages of our method.

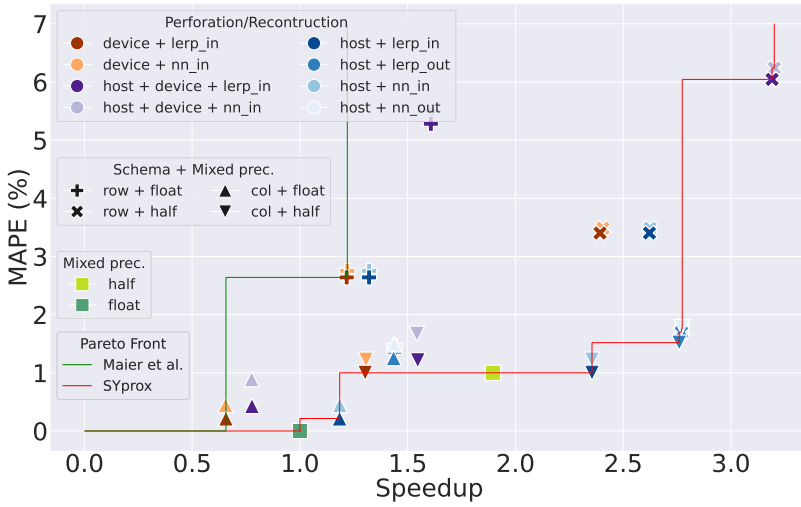


Figure 4.7: Domain space of the approximate computing techniques for Maier et al. and SYprox approach. Different colors represent combination of perforation and reconstruction. Markers distinguish the combination of perforation schemes and data types.

Enhanced Coverage of the Objective Space. One of the primary benefits of SYprox lies in its ability to generate a substantially larger number of configurations compared to the approach proposed by Maier et al. [114]. This advantage is illustrated in Figure 4.7, where the SYprox configurations span a wider section of the objective space. This expanded coverage allows an exploration of a wider spectrum of speedup and error trade-offs, thereby facilitating more precise optimization tailored to diverse application requirements. The SYprox configurations extend across both axes, indicating a versatile approach capable of addressing various performance and error needs. Additionally, the approximation domain can be further broadened by adjusting the skip factor parameter, thereby generating new potential Pareto-optimal solutions.

Discovery of New Pareto Optimal Solutions. By combining different approximation techniques, SYprox not only expands the configuration space, but also identifies new Pareto optimal solutions that were previously unattainable with existing methods. SYprox's advantages are evident when considering device perforation with the column schema. In this case, the column schema can cause cache misses due to the data layout, leading to a performance slowdown of up to two times. By applying host perforation instead of device perforation, we mitigate the cache misses generated by the column data layout. This adjustment allows the configuration with host perforation and the column schema to perform comparably to the row schema and even become a Pareto optimal solution. This demonstrates how SYprox can overcome specific performance bottlenecks and optimize configurations that were previously suboptimal. This capability is reflected in the red Pareto front associated with SYprox, which dominates the green Pareto front of Maier et al., showing that our approach consistently outperforms the prior state-of-the-art.

Combining Approximation. SYprox demonstrates significant performance improvements by combining various approximation techniques. Combining mixed precision with the other approximation often generates new configurations that are Pareto optimal, since reducing precision in most cases introduces a small error with up to 2x speedup. Furthermore, by combining host and device perforation with mixed precision, we achieve up to 3.5x speedup with only a 7% error. This combination is particularly effective because it takes advantage of the strengths of all the approximations.

4.6.6 *Performance and Accuracy Portability*

Figure 4.8 demonstrates the portability of our approach across different hardware architectures, including AMD MI100, Intel Max 1100 and NVIDIA V100S GPUs. A key observation is that the error remains consistent across all three hardware platforms. This consistency highlights that the approximate computing techniques implemented in SYprox are predominantly data-dependent rather than hardware-dependent. The only notable exception is half-precision, which introduces slight variations due

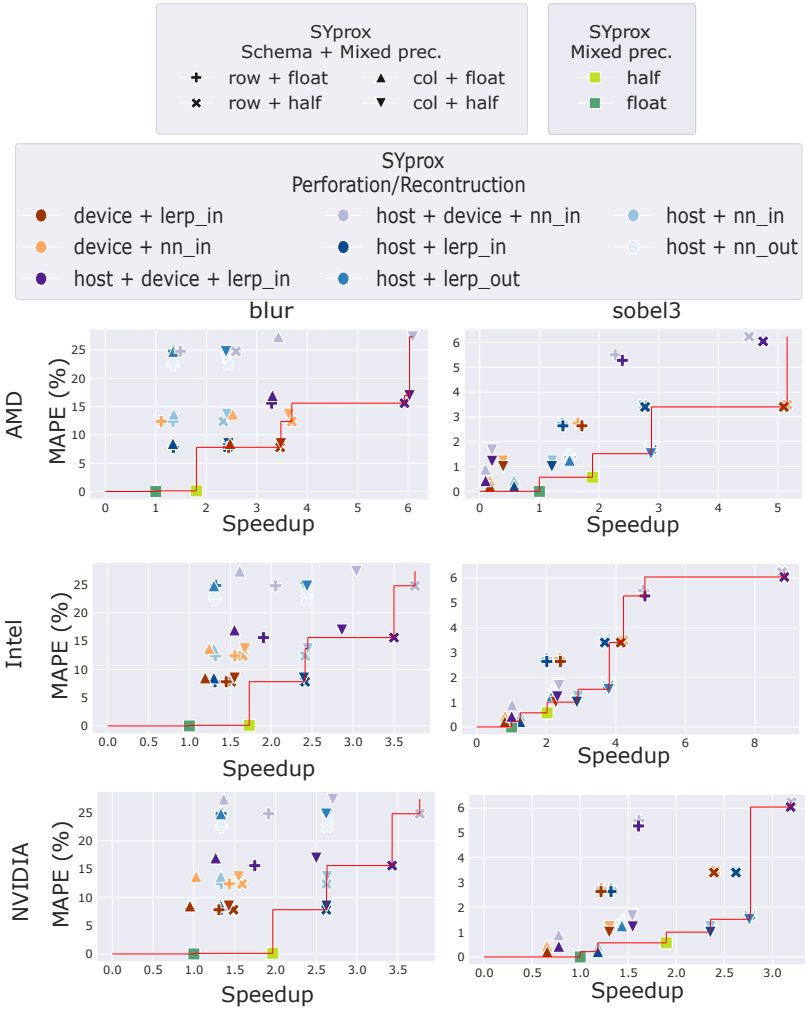


Figure 4.8: Performance evaluation of SYprox on AMD, Intel and NVIDIA hardware. The color represents different perforation and reconstruction techniques. Markers are used to distinguish the combination of perforation schemes and data types.

to differences in hardware precision handling. However, these variations are minimal and do not significantly affect the overall error profile. When analyzing performance, we notice some variability between hardware platforms. All platforms achieve speedups of more than 3x, with some hardware yielding even better performance using the same approximation.

4.7 SUMMARY AND DISCUSSION

This chapter introduced SYprox, a novel framework designed to enable approximate computing on heterogeneous architectures through a portable and expressive SYCL-based abstraction. The contributions of this work span several complementary directions. We enhanced the SYCL programming model by extending buffers and accessors with approximation-aware capabilities, enabling developers to incorporate approximation techniques directly within heterogeneous applications without relying on custom compiler support. We also presented a new host-side data perforation strategy that exploits the host–device execution model to significantly reduce both kernel execution time and data-transfer overhead. In addition, SYprox provides a unified environment in which multiple approximation techniques, data perforation, reconstruction, and mixed precision, can be composed, thereby expanding the approximation design space and enabling the discovery of new Pareto-optimal configurations. We have experimentally assessed the SYprox methodology on AMD MI100, Intel Max 1100, and NVIDIA V100, comparing it with state-of-the-art frameworks. The results highlighted the advantages of host perforation, which consistently outperforms device perforation by reducing both kernel computation and data transfer times. Moreover, the ability of SYprox to combine multiple approximation techniques provides a rich set of Pareto optimal solutions that outperform prior methods, such as those proposed by Maier et al. and the HPAC framework. SYprox not only discovers more Pareto optimal solutions, but also expands the coverage of the objective space, offering greater flexibility for tuning trade-offs between speedup and error. Finally, our approach demonstrated robust performance and accuracy across different GPUs, validating the SYprox performance and accuracy portability.

5

ENERGY-EFFICIENT COMPUTING WITH FREQUENCY SCALING

Energy efficiency has become a central concern in modern computing, spanning domains from cloud and edge computing to big data analytics, in-memory processing, and large-scale HPC systems [75, 179, 186, 208]. In particular, exascale computing platforms face significant energy challenges, as large-scale heterogeneous systems consume substantial power, resulting in higher electricity costs, cooling requirements, and carbon emissions [132, 178, 195]. These issues, combined with the diminishing efficiency gains of Moore's Law, have made energy-efficient computing a critical research objective [72]. Dynamic Voltage and Frequency Scaling (DVFS) has emerged as a key technique for improving energy efficiency in modern hardware. By adjusting core and memory frequencies, DVFS can reduce energy consumption while maintaining acceptable performance [81, 92, 155]. However, exploiting DVFS in today's heterogeneous architectures presents two main challenges. First, existing frequency-scaling mechanisms are often vendor-specific, such as Intel's RAPL, NVIDIA's NVML, and AMD's ROCm SMI, making it difficult to achieve portable solutions across CPUs, GPUs, and specialized accelerators. Second, the energy-optimal frequency can vary across different computational kernels; applying the same frequency uniformly across an entire application may miss significant energy-saving opportunities [47, 66, 93]. Existing energy management frameworks, such as GEOPM [43] and EAR [30], typically operate at coarse granularity, applying energy policies at the job or application level. While simpler to implement, these approaches often fail to capture the fine-grained energy characteristics of individual kernels, limiting achievable energy savings. On the other hand, fine-grained approaches, which adapt frequencies per kernel, can improve energy efficiency but may incur overhead due to the latency of frequency changes, especially in multi-GPU and multi-node environments [48].

In this chapter, we first present SYnergy, a SYCL-based framework for portable energy profiling and frequency scaling on heterogeneous systems. SYnergy abstracts low-level energy management APIs, to implement energy profiling and DVFS at different levels of granularity (i.e. application and kernel), without requiring programmers to handle the complexities of specific hardware architectures. Building on SYnergy, we also propose a phase-based frequency scaling, a novel energy optimization methodology that applies frequency scaling to distributed heterogeneous systems. The proposed methodology overcomes the limitations of existing coarse- and fine- grained solutions by proposing an adaptive phase-based approach. Given a *Direct Acyclic Graph* (DAG) representation of the computation, along with profiling information, our approach automatically identifies phases and applies the optimal frequency for each phase. Furthermore, by encoding MPI information in the DAG, our method can further hide the frequency-change latency through communication overlap in multi-GPU and multi-node applications. Our methodology is fundamentally abstracted from the underlying hardware, and we demonstrate its portability to GPU systems from various vendors (i.e., AMD, Intel, NVIDIA). The methodology is evaluated on a range of benchmarks and real-world applications, highlighting both energy reduction and scalability across modern GPU platforms from multiple vendors. Through the combination of SYnergy and phase-based frequency scaling, this chapter demonstrates how high-level programming model abstractions for energy-efficient computing can enable portable energy optimization on heterogeneous systems.

5.1 RELATED WORK ON FREQUENCY SCALING

Systems composed of heterogeneous hardware such as CPUs and GPUs are now broadly used in exascale computing, they bring high computing capability but also consume significant power and energy. In fact, power and energy consumption are considered primary issues in large-scale HPC [161, 168]. Many researchers have proposed approaches to tackle this issue [61, 129, 152].

DVFS-based technique DVFS is one of the widely used techniques to enable energy efficiency in HPC. Numerous studies have investigated different DVFS-based mechanisms from different perspectives. For example, some researchers focused on analyzing the impact of DVFS on multi-objective optimization using machine learning [47, 88]. Some researchers focused on combining frequency scaling with other energy-efficient techniques, such as power capping [69]. Furthermore, DVFS techniques can be implemented on different hardware and heterogeneous processors [26, 31, 48, 88, 93, 124]. Countdown [26] is a runtime library on CPUs, which can automatically reduce power consumption by adding DVFS capabilities into standard MPI libraries. Countdown supports both fine and coarse granularity MPI to inject power management calls. *However, existing DVFS-based methods do not consider fine-grained frequency scaling and MPI applications on CPU-GPU heterogeneous architecture.* **Phase-aware DVFS technique** From a granularity perspective, many researchers started to investigate frequency scaling based on the phases rather than the fine- or coarse-grained approaches [18, 89, 112, 117, 145, 170]. Among them, Qiu et al. [145] presented a three-phase DVFS algorithm that achieves higher energy saving by clustering task slacks via task graph unrolling. Booth et al. [18] proposed a phase-based voltage and frequency scaling that chooses the phases according to the Segments of code with unique performance and power attributes using Hidden Markov Models. *However, existing phase-aware methods are not implemented on modern heterogeneous architecture.*

DVFS in MPI applications Energy efficiency with the DVFS-based techniques has also been applied to distributed memory context [26, 55, 103, 108]. Rountree et al. [156] used linear programming to resolve the NP-complete problem of when to change the frequency in an MPI program. Zamani et al. [205] aimed to apply DVFS to a specific application on multiple GPUs. They used the algorithmic knowledge from the application to predict slack times and do frequency scaling on both CPU and GPUs. Endrei et al. [45] performed frequency scaling and proposed a statistical model to find the best tradeoff between performance and energy efficiency in MPI applications. *However, none of the*

related work tried to hide the overhead of frequency scaling, and none of them provided a generic approach applicable to any type of application.

5.2 TOWARDS PORTABLE ABSTRACTION FOR ENERGY-EFFICIENT COMPUTING

IMPACT OF FREQUENCY SCALING Nowadays, most GPU vendors provide an energy management interface to enable power and energy profiling and core frequency scaling. Examples include AMD's ROCm SMI [1], Intel's Level Zero [79], and NVIDIA's NVML [126]. Frequency scaling, as an optimization strategy, can significantly reduce the energy consumption of a task. While some hardware supports changing both core and memory frequency, most modern data center GPUs only allow changing the core frequency. Generally, energy efficiency comes at the cost of performance, creating a multi-objective problem where we can investigate different trade-offs.

Figure 5.1 shows the energy-speedup tradeoff for two kernels on three different GPUs. *Default configuration* shows the default GPU frequency set by the manufacturer. For NVIDIA GPUs, this corresponds to the default core frequency with AutoBoost disabled. For AMD, we determined this value by comparing the *performance level automatic* with the manually configured performance level. For Intel, the default frequency is derived by evaluating and comparing different min-max core frequency ranges. *Min EDP* is the frequency that minimizes the *Energy Delay Product* (EDP) [99], while *Min Energy* minimizes the energy consumption. *Max Perf* is the frequency that maximizes performance. The *Pareto front* represents a set of optimal solutions where no solution can be improved without compromising another objective. In this context, the Pareto front helps to identify trade-offs between performance and energy consumption. The energy characterization presents the multi-objective energy-performance distribution of different core-frequency settings, using the default frequency as a baseline. We used two kernels from the SYCL-Bench suite [34], `matrix_mul` (size: 5000×5000) and `mersenne_twister` (size: 524288), and executed both kernels across all supported frequencies on three different GPUs (AMD MI100, Intel Max 1100, and NVIDIA V100S). For all three GPUs, it is clear that scaling the

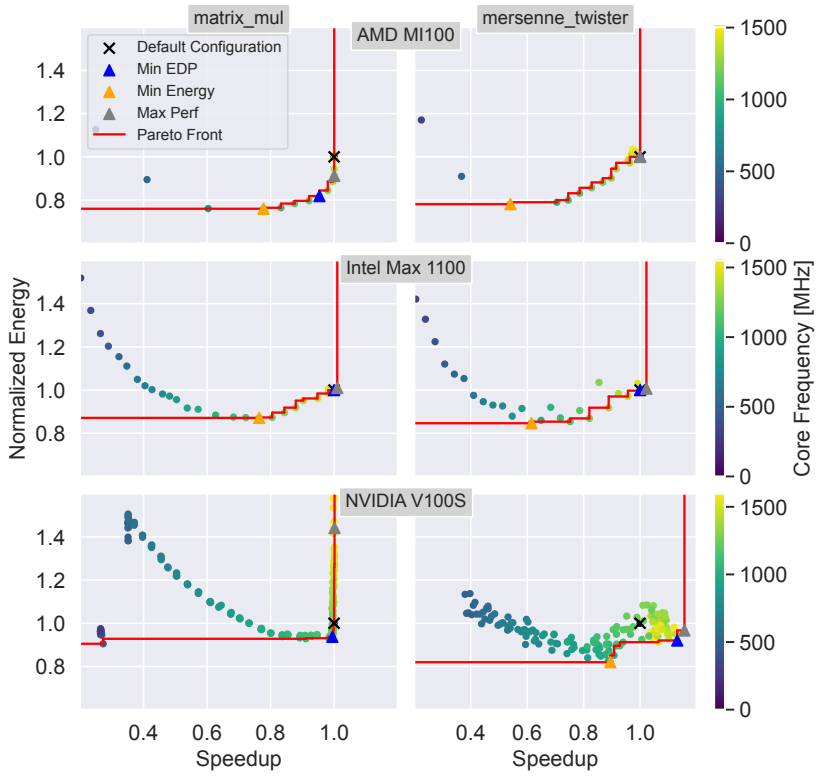


Figure 5.1: Multi-objective characterization of `matrix_mul` (left) and `mersenne_twister` (right) benchmarks.

frequency can lead to improvements in at least one of the objectives: energy and performance. For instance, in the case of the NVIDIA V100, with Min EDP as the energy target, `matrix_mul` achieves an 8 % energy savings with a performance loss of about 1 %, while `mersenne_twister` achieves a 10 % energy saving and a 15 % performance gain. The relationship between time and energy depends on whether a kernel is memory or compute bound. Compute-bound kernels benefit more from higher core frequencies compared to memory-bound kernels.

COARSE-GRAINED FREQUENCY SCALING Current approaches for frequency scaling are coarse-grained [93, 203, 207] meaning that they optimize the energy or performance of the application by setting a single frequency for the whole program. Although this method can be easily implemented by job schedulers and

is effective for single-kernel applications, it becomes inefficient for more complex applications with multiple kernel executions. In fact, each kernel may have different energy characterizations, which require a more fine-grained energy tuning, such as setting a distinct frequency for each individual kernel as shown in Figure 5.1.

VENDOR-SPECIFIC LIBRARIES FOR DVFS Although modern GPUs provide energy monitoring and frequency scaling capabilities, their practical use across heterogeneous systems is limited by the reliance on vendor-specific interfaces. Each hardware vendor provides its own low-level API for energy management, such as AMD's ROCm SMI [1], Intel's Level Zero [79], and NVIDIA's NVML [126]. While these APIs expose detailed control over device behavior, they require programmers to write architecture-dependent code, limiting portability and making the integration of energy-aware optimizations more difficult. At the same time, widely adopted programming models such as SYCL, OpenMP, and Kokkos focus primarily on performance portability and do not expose interfaces for power and energy management. As a result, developers who wish to incorporate DVFS or energy profiling into heterogeneous applications must manually interact with device-specific libraries, breaking abstraction layers and significantly increasing development complexity. This lack of portable, high-level support makes it difficult to design or deploy energy-efficient techniques that operate consistently across different architectures.

In this chapter, we address the gap of coarse-grained approaches and vendor-specific libraries by extending the SYCL programming model with SYnergy, a portable and unified interface for energy monitoring and fine-grained frequency scaling on heterogeneous architectures.

5.3 SYNERGY: A PORTABLE INTERFACE FOR ENERGY-EFFICIENT COMPUTING

The SYnergy energy-aware SYCL API addresses the lack of a unified and portable mechanism for accessing energy data and controlling DVFS across heterogeneous GPU architectures. In-

spired by the Celerity SYCL extension [162, 183], the API provides a common interface for energy profiling and frequency scaling, allowing SYCL applications to operate across different vendors without relying on low-level, device-specific libraries.

Distributed as a lightweight, header-only library, SYnergy integrates seamlessly into existing C++ build environments. In this work, we use the SYnergy bindings for NVIDIA, AMD and Intel GPUs, which internally map to NVML [126], ROCm SMI [1] and Level Zero [79], respectively. This abstraction enables portable, vendor-agnostic energy management within SYCL applications.

Profiling Interface

The main entry point for using the SYnergy interface is the `synergy::queue` class, which extends the standard SYCL queue with energy capabilities. The SYnergy API provides coarse-grained and fine-grained energy profiling capabilities that allow measuring, respectively, the energy consumption of the whole device and the energy consumption of each kernel executed in a SYCL application.

Listing 5.1 shows how the API can be used to query the energy consumed by a kernel, i.e., a `parallel_for` and a device using `kernel_energy_consumption` and `device_energy_consumption` functions, respectively. Since device computation is asynchronous, we wait for the kernel to finish before querying energy consumption.

```
synergy::queue q{gpu_selector_v};
buffer<float, 1> x_buf{x};
buffer<float, 1> y_buf{y};
buffer<float, 1> z_buf{z};
event e = q.submit([&](handler& h) {
  accessor<float, 1, read> x_acc{x_buf, h};
  accessor<float, 1, read> y_acc{y_buf, h};
  accessor<float, 1, write> z_acc{z_buf, h};
  float a{alpha};
  h.parallel_for(range<1>{n}, [=](id<1> id) {
    z_acc[id] = a * x_acc[id] + y_acc[id];
  });
});
e.wait_and_throw();
double kernel_energy = q.kernel_energy_consumption(e);
```

```
16 double device_energy = q.device_energy_consumption();
```

Listing 5.1: Energy profiling with the SYnergy API

With the coarse-grained approach, the device energy consumption is measured by sampling the instantaneous power in a time window that begins when the SYnergy queue is built and ends when it is destroyed. This feature is useful whereas an application is made up of a mix of large and small kernels, the latter of which are not easy to profile because of the short execution time as explained in Section 5.3. In order to enable fine-grained energy profiling the SYnergy API leverages the SYCL event features that allow us to query the execution status of a kernel (i.e. submitted, running, complete). Using an asynchronous thread to poll the kernel status we sample the power of a kernel until it is complete. In this way, we measure only the kernel energy consumption rather than the entire device.

Frequency Scaling Interface

A SYnergy queue can be constructed as a conventional SYCL queue or by manually defining the memory and core frequencies configuration for the kernels that will be submitted to the queue. Listing 5.2 illustrates a SYnergy queue construction with a memory frequency of 1215 MHz and a core frequency of 210 MHz.

```
1 synergy::queue q{1215, 210, gpu_selector_v};
2 ... // setup buffers
3 event e = q.submit([&](handler& h) {
4   ... // setup accessors
5   h.parallel_for(n, kernel);
6 });
```

Listing 5.2: SYnergy queue with target frequencies

For fine-grained frequency scaling, each kernel submitted to the queue can be executed with a specified frequency configuration, which is set just before the kernel starts.

Finally, all of these approaches can be mixed, allowing for multiple queues with different target configurations, with the

ability to specify a target for each kernel submission, as shown in Listing 5.3.

```
synergy::queue low_freq{877, 810, gpu_selector_v};
synergy::queue default_freq{gpu_selector_v};
... // setup buffers
low_freq.submit([&](handler& h) {
  ... // setup accessors
  h.parallel_for(n, kernel1);
});
default_freq.submit(877, 1530, [&](handler& h) {
  ... // setup accessors
  h.parallel_for(m, kernel2);
});
```

Listing 5.3: Queues and kernels with different targets

Limitations

Since SYCL has no way to execute instructions just before a kernel starts executing on the device, SYnergy implements frequency scaling in the command group. However, as the kernel execution is asynchronous, and the corresponding kernel may not have been executed yet; therefore, we wait for the completion of the task. The energy profiling and frequency scaling are also affected by issues related to the underlying vendor-specific libraries. Accurate fine-grained energy profiling is limited by the fact that the kernel execution must be long enough in order to produce meaningful results, due to the maximum sampling frequency supported by the hardware, which needs around 15 ms long sampling intervals [21].

5.4 TOWARDS PHASE-BASED FREQUENCY SCALING

FREQUENCY SCALING OVERHEAD Although fine-grained frequency scaling can effectively improve energy and performance, we noticed that frequency change imposes an overhead, which can influence the effectiveness of energy-tuning granularity. To show this, we designed a benchmark called `fsbench1` consisting of two kernel types with different energy characteristics. The first kernel is a `matrix_mul` with 1024×1024 elements and an opti-

mal frequency of 1110 MHz, and the second is a `median_filter` with a 2048×2048 input size and an optimal frequency of 645 MHz. We first run the `matrix_mul` in a loop and then run the `median_filter` in another loop right after the first loop. The optimal frequency is selected based on the *Min Energy* target in this case. We implement frequency scaling through two methods: coarse-grained and fine-grained, utilizing ROCm for AMD, LevelZero for Intel, and NVML for NVIDIA.

Figure 5.2 shows the time taken for each approach when we repeat the `fsbench1` benchmark with a loop count of 512 (512 invocations for the first kernel, followed by 512 for the second kernel). The coarse-grained approach sets a frequency once at the beginning of the program, whereas the fine-grained approach sets the frequency 1024 times, once before each kernel invocation.

If we do not consider the overhead of frequency changing (the hatched area), the fine-grained technique will outperform the coarse-grained method for all three GPUs, as it sets an optimal frequency for each kernel. However, considering the high overhead of changing frequency (the hatched area), the fine-grained approach becomes even slower for Intel and NVIDIA compared to the coarse-grained method. For NVIDIA, which experiences a higher overhead for frequency changes, each frequency change with NVML takes almost 0.6 ms, making the fine-grained approach around 47 % slower than coarse-grained, overall.

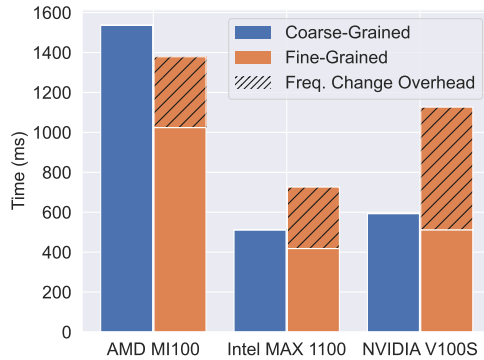


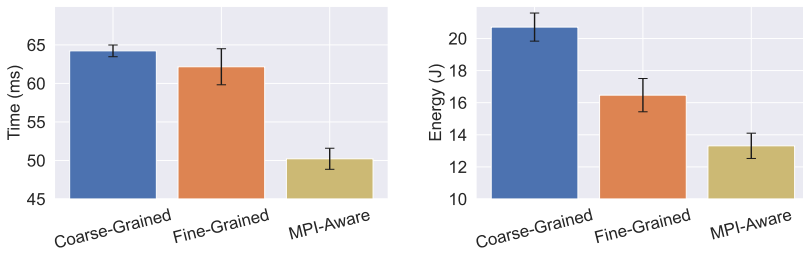
Figure 5.2: Frequency scaling overhead for coarse- and fine-grained approaches on `fsbench1`.

As we showed, changing the frequency with vendor-provided tools incurs an overhead in the range of a fraction of a millisecond.

This is a problem: first, if we have an application consisting of many lightweight kernels, each with a runtime of microseconds or a fraction of a millisecond. In this case, the fine-grained method would have a very high overhead, and this overhead may become higher than the benefit we get from frequency scaling. Second, when multiple consecutive kernels in the application require the same optimal frequency, changing the frequency for each individual kernel is inefficient.

PHASE-BASED MPI-AWARE APPROACH Considering the challenges of coarse- and fine-grained frequency tuning methods, there is a need for a third approach that addresses their shortcomings. This work proposes a **phased-based** approach that aims to minimize the frequency changes in fine-grained approaches while maintaining their energy efficiency.

The first key insight is that we can *group tasks with similar energy characteristics* into a single phase. After identifying the application phases, we will set an optimized frequency for each phase. The second key insight is that it is possible to *hide the frequency change latency by overlapping it with communication*, similar to typical computation-communication overlap optimization. This strategy is effective in multi-GPU and multi-node applications, where a significant amount of time is spent communicating between GPUs and nodes. We have experimentally evaluated the optimization potential of overlapping frequency changes while the GPUs are communicating.



Execution time Energy consumption

Figure 5.3: Execution time and energy consumption of the coarse-grained, fine-grained, and MPI-aware approach for `fsbench2` on 4 Intel Max 1100 GPUs.

To assess this, we designed a simple benchmark called `fsbench2` consisting of 2 kernels (with different energy characteristics) and 2 MPI collective operations. We call a kernel after each MPI collective operation in the following order: communication 1 (`MPI_Bcast`), kernel 1 (`geometric_mean`), communication 2 (`MPI_Reduce`), kernel 2 (`vector_addition`). Intuitively, we will have 2 phases, assigning one phase to each kernel. Using the techniques described later in Section 5.5.2, we overlap the frequency change overhead with MPI communications. Figure 5.3 shows the time taken and energy consumption for the coarse-grained, fine-grained (which is equivalent to a phase-based technique with only 2 phases), and the MPI-overlapped (which is phase-based with frequency change overhead overlapped with MPI communications) across 4 GPUs. The results indicate that such overlapping can significantly enhance performance, reducing the time and energy consumption of the fine-grained approach by 20 % and 19 %, respectively, compared to the same method without overlapping.

5.5 PHASE DETECTION METHODOLOGY

In this section, we describe in detail the phase-based frequency scaling methodology for single and multi- GPU systems.

5.5.1 *Phase Detection on Single Device*

The benefits of employing a fine-grained frequency scaling approach may be constrained by the frequency change overhead (see Fig. 5.2). Our methodology for a single device, shown in Figure 5.4, introduces a phase detection algorithm that aims to minimize the number of frequency changes while preserving the energy efficiency of the fine-grained approach. This approach begins with the input of a SYCL code targeting, e.g., a single GPU. ❶ In the first step, after conducting a one-shot profiling and predicting the energy using the model, we generate a *Directed Acyclic Graph* (DAG) that represents task (kernel execution) dependencies enriched with their time and energy characteristics. ❷ In the second step, the phase detection algorithm is applied to the DAG to detect the phases. This is achieved by identifying

groups of tasks in the application that require a similar frequency. A frequency is then set for all tasks within the identified phase.

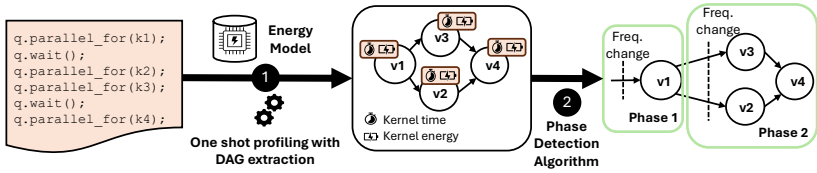


Figure 5.4: Single-device energy-aware DAG modeling.

In the rest of this section, we present the theoretical framework used to identify the energy phases in single device scenario: the energy-aware DAG model, the algorithm based on dynamic programming, and the cost function.

5.5.1.1 *Energy-aware DAG Model*

We model the execution of a parallel program using a computational DAG, where tasks are kernels executed on the device, represented by graph vertices, and inter-task dependencies are captured by graph edges. The DAG is extended with energy and runtime information extracted from the one-shot profiling step. Formally, we define an *Energy-annotated DAG*, which is 4-tuple $C_e = (V, E, t, e)$ of finite sets, such that: V is the set of vertices representing the tasks of the profiled application; E is the set of edges defining data dependencies between tasks; t and e are the time and energy functions defined as $t : V \times F \rightarrow R$ and $e : V \times F \rightarrow R$, where F is the set of available frequencies.

5.5.1.2 *Cost Function*

The phase detection algorithm leverages the cost function $Cost(i, j)$ to compute the cost of having a phase with tasks $(vi, \ldots, vj)$. The cost associated with a phase considers four factors: the number of times r_k that the task v_k is repeated (in one or more nested loops); the frequency change overhead ovh_e; the execution time of each task $t(v_k, f_{opt})$; and determining the optimal frequency f_{opt} for the entire phase by selecting the frequency that minimizes overall

energy consumption across all included tasks. Our methodology provides a cost function that encapsulates all these aspects:

$$Cost(i,j) = ovh_e + \sum_{k=i}^{j} r_k \cdot t(v_k, f_{\text{opt}}) \tag{5.1}$$

Traditionally, loops in the source code are unrolled in the task DAG. This approach significantly impacts the complexity of algorithms operating on the DAG, leading to increased computational overhead and memory usage. Real-world applications may contain several tasks repeated multiple times inside loops. For instance, CloverLeaf, the real-world application used in this chapter, encountered over 7000 kernel invocations. To address this challenge, instead of expanding loops in the task DAG, we opt to incorporate loop information as metadata attached to the nodes during the profiling step. For this, we count the number of kernel invocations for each kernel in the profiling phase. This strategy allows us to retain loop semantics without inflating the size of the DAG, providing a more efficient solution for managing loop-heavy scenarios. In order to handle loop information, for a task v_i, we define $L_i = \{\text{loops } \ell : v_i \text{ occurs in } \ell\}$. The repetition factor $reps_i(\ell)$ of a task v_i in a loop $\ell \in L_i$ is the number of times the task v_i is executed during the loop ℓ. The number of times that the task v_k is repeated is defined as $r_k = \prod_{\ell \in L_k} reps_k(\ell)$.

The overhead ovh_e considers two aspects: first, the overhead associated with a single frequency change (either time or energy, depending on the target metric), defined as ϵ, and second, the number of times the frequency change must be executed if it occurs within a loop ($reps_i$). Formally, the overhead of frequency changes for a phase composed of the tasks $(v_i, \ldots, v_j)$ is defined as follows:

$$ovh_e = \prod_{\ell \in L_i} reps_i(\ell) \cdot \epsilon \tag{5.2}$$

The optimal frequency f_{opt} that minimizes the energy consumption of the tasks $(v_i, \ldots, v_j)$ is computed as follows:

$$f_{opt} = \underset{f_t \in \{f_i, \ldots, f_j\}}{\arg\min} \sum_{k=i}^{j} (e(v_k, f_t) - e(v_k, f_k)) \cdot r_k \tag{5.3}$$

where $\{f_i, \ldots, f_j\}$ are the optimal frequencies that minimize the energy, respectively, for tasks $(v_i, \ldots, v_j)$. The optimal frequency selection for tasks $(v_i, \ldots, v_j)$ can be generalized to support other target metrics such as Max Perf or Min EDP.

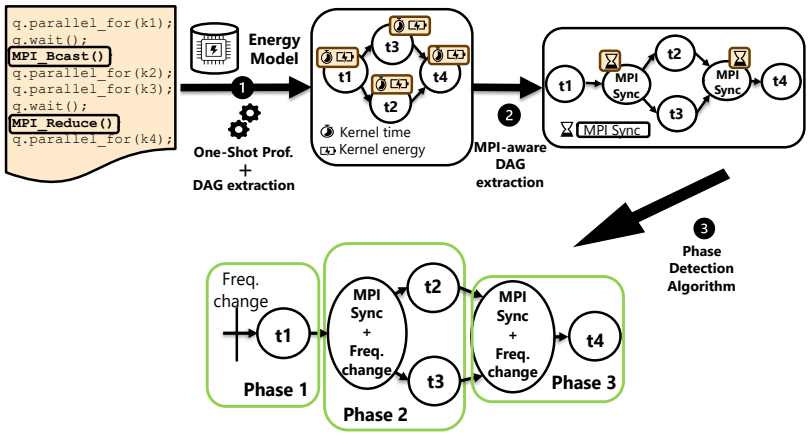


Figure 5.5: MPI-aware phase detection algorithm.

5.5.2 *Phase Detection on Multi-nodes*

Thus far, we have demonstrated how to detect phases for single-device applications. However, in multi-GPU and multi-node applications, communication also needs to be taken into account. MPI communications can introduce additional data dependencies to the DAG. On the other hand, we have the opportunity to hide the frequency change overhead within the communication time.

Figure 5.5 shows the phase detection algorithm adapted for MPI applications. ❶ Similar to single-device applications, the first step is to generate a task DAG called C_e, enriched with profiled information and energy predictions. ❷ The C_e is further augmented with MPI-related information, specifically MPI synchronization nodes with annotated runtimes to generate a C_{mpi} graph. ❸ The phase detection algorithm is then applied to the DAG to detect phases. The algorithm remains the same as that used for single-device applications but operates on a C_{mpi} DAG rather than a C_e DAG.

The rest of the section describes how the C_e DAG and the cost function, designed for the single-device context, are extended to handle multi-node/device scenarios.

5.5.2.1 *MPI-aware DAG Modeling*

By incorporating *MPI_Sync* nodes, the energy-aware DAG is enhanced as an energy- and MPI-aware DAG, considering the communication times in multi-device systems. With MPI blocking functions, the next scheduled task should wait for data from other devices to proceed. The integration of *MPI_Sync* nodes represents these waiting times, which can be used by the phase detection algorithm to perform MPI communication simultaneously with the frequency change. Formally, we extend our methodology with an MPI-aware DAG defined as a 4-tuple $C_{mpi} = (V', E', t', e')$, of finite sets such that: V' is the set of vertices representing the tasks of the profiled application with the *MPI_Sync* point; E' is the set of edges defining data dependencies between tasks, including the dependencies introduced by MPI communications; t' and e' are, respectively, the time and energy functions defined in the same way as C_e DAG. For t', the C_{mpi} DAG encapsulates the time spent in the new *MPI_Sync* node, while for e', the energy for the *MPI_Sync* node is always 0 since we are only interested in the communication time that can be exploited to hide the frequency change.

5.5.2.2 *MPI-aware Overhead*

The cost function defined for C_e is extended to C_{mpi} in order to consider, in the overhead, the time saved by hiding the frequency change overhead during MPI blocking communications. Formally, Eq. 5.2 is extended as follows:

$$ovh_{mpi} = \prod_{\ell \in L_i} reps_i(\ell) \cdot (\epsilon - MPI\_Sync) \tag{5.4}$$

where the *MPI_Sync* value for *MPI_Sync* node represents the time or energy spent in MPI blocking communication, which is 0 for the nodes representing the tasks. When $MPI\_Sync > \epsilon$, the overhead (ovh_{mpi}) is considered 0 since the cost of frequency changes can be fully overlapped with the communication process.

5.5.2.3 *MPI Collective Communications*

In MPI blocking communications, all the involved processes wait until the operation they are performing is completed be-

fore allowing the program to proceed. Synchronization occurs internally within the blocking calls. Therefore, it is impossible to overlap these communications with the calls to frequency change APIs. In this case, we need to change these blocking calls to non-blocking and then overlap them with the frequency change function calls. Listing 5.4 illustrates an example of such a case in which we replace the `MPI_Bcast` with `MPI_Ibcast` and a corresponding `MPI_Wait` in Listing 5.5.

```
1 MPI_Bcast();
2 Freq_change();
3
4 q.parallel_for(kernel);
```

Listing 5.4: Frequency change without overhead hiding.

```
1 MPI_Ibcast();
2 Freq_change();
3 MPI_Wait();
4 q.parallel_for(kernel);
```

Listing 5.5: Frequency change with overhead hiding.

5.5.2.4 *Stencil Communications*

Stencil communications are common patterns in many MPI applications, where each process communicates with its neighbor processes in different dimensions. Here, we demonstrate that our method for overlapping communication and frequency change also applies to stencils. Listings 5.6 and 5.7 provide an example of a one-dimensional stencil in which we overlap communication with the call to change the device frequency.

```
1 MPI_Sendrecv();
2
3 Freq_change();
4
5 q.parallel_for(kernel);
```

Listing 5.6: Freq. change example in stencils.

```
1 MPI_Irecv();
2 MPI_Isend();
3 Freq_change();
4 MPI_Waitall();
5 q.parallel_for(kernel);
```

Listing 5.7: Hiding freq. change overhead in stencils.

The proposed overlapping technique can be applied to all MPI communication operations, whether they are blocking or non-blocking. If the communications are already non-blocking, taking into account the data dependency between the communication and the following kernel, we can overlap the frequency modification time with the data transfer. The only case where the current method is not applicable is when the non-blocking MPI

communications overlap with kernel execution. In this case, since the kernel is already executing during the MPI communication, we cannot modify the frequency of the GPU as it impacts the currently running kernel.

5.5.3 *Phase Detection Algorithms*

In order to have a comparison and choose the most efficient algorithm for phase detection, we tested three different algorithms: *Dynamic programming* (Dp), *Greedy* (Gr), and *Clustering* (Cl). The first step in common between all the three algorithms is to apply a topological ordering on C_e, producing a sequence of tasks $T = (v_0, \ldots, v_{|T|-1})$, where $|T| = |V|$. In this work, we adopt the `in_order` queue property defined by SYCL; that is, the kernels execute in the same order in which they are submitted to the queue.

Dynamic programming (Dp). This approach is commonly used in optimization problems, where the aim is to find the best solution among many possible ones. Given T the topological order of C_e, the dynamic programming solution can be expressed using the following recurrence relation:

$$opt(i,j,k) = \begin{cases} Cost(i,j) & \text{if } i{=}j \vee k{=}0 \\ \min \begin{cases} \min\limits_{i \le l < j} \{opt(i,l,k-1) + opt(l+1,j,k-1)\} \\ Cost(i,j) \end{cases} & \text{otherwise} \end{cases} \tag{5.5}$$

where $opt(i,j,k)$ represents the cost of the optimal partitioning for tasks $(v_i, \ldots, v_j)$ with at most $2^k - 1$ splits, and $Cost(i,j)$ defines the cost of having $(v_i, \ldots, v_j)$ in the same phase. In iteration k, the algorithm evaluates whether to proceed with further subdivisions based on the optimal partitions found in iteration $k-1$. The terms $opt(i,l,k-1)$ and $opt(l+1,j,k-1)$ reflect the costs of splitting the interval $[i,j]$ at position l, where each subproblem has been solved in iteration $k-1$.

The time complexity of Dp is $O(|T|^4)$, as it requires four nested loops on the number of tasks ($|T|$). However, as the number of tasks grows, a time complexity of $O(|T|^4)$ can limit the applicability of the phase-based approach in practice.

Greedy (Gr). The greedy approach (*Gr*) reduces the time complexity to $O(|T|^2)$ while providing solutions that are not necessarily optimal. The greedy algorithm examines each task in the list, one by one, and decides whether to add it to the current phase or the next phase. This decision is based on the cost function $Cost(i, j)$, as defined in Section 5.5.1.2. The algorithm can be implemented with a loop iterating over tasks in T, where in each iteration, the cost function is applied with time complexity of $O(|T|)$, resulting in an overall time complexity of $O(|T|^2)$. The greedy approach constructs phases by processing tasks in the order they appear in the list. Although this approach allows for on-the-fly phase detection, it can limit the potential to select optimal phases.

Clustering (Cl). The clustering approach offers greater flexibility by allowing tasks to be grouped in a more efficient way. The key idea is to prioritize the phases that provide the highest potential for energy savings and then determine, using the cost function $Cost(i, j)$, whether it is better to merge that phase with the next one. Initially, each task composes one phase (or cluster), creating a set of clusters that can be combined. At each iteration, the algorithm chooses the cluster that yields the maximum energy savings. Once a promising cluster is identified, the cost function helps decide whether combining it with the adjacent cluster is beneficial. If combining is advantageous, the clusters are merged; otherwise, the cluster is moved to the final list of phases. This process continues until no further clusters can be merged. The algorithm iterates over all tasks, applying the selection and cost function at every step, resulting in the same computational complexity as the Greedy algorithm. By prioritizing energy savings rather than following a strict task order, the clustering approach has the potential to detect more efficient phases.

Section 5.6.2 provides an in-depth comparison of *Gr* and *Cl* approaches against the optimal approach (Dp) on a wide range of multi-kernel applications with different characteristics.

5.5.4 *Implementing Phase-based Frequency Scaling*

The theoretical framework is translated into a concrete implementation where energy profiling is tailored to device-specific

APIs, tasks are extracted from SYCL kernel executions, and MPI communications are specifically handled.

5.5.4.1 *Profiling Time and Energy*

The first step in identifying the application phases is to analyze and profile the application and extract the required features. The profiling information comprises timing data for kernels, the number of times each kernel is invoked, MPI communications, and energy characterization of the kernels. For MPI profiling, we recorded the time spent on each collective or point-to-point operation. The time and energy profiling of each kernel is performed using the SYnergy API and model [48] along with the SYCL events.

Our methodology involves one-shot profiling at the default device frequency, which gathers all the required information. By utilizing the model, which requires only the LLVM IR of code as input, we can automatically predict time and energy values for all supported frequencies for each device, thereby eliminating the need to execute the application multiple times. In our experiments, the profiling process involved running each application five times at the default frequency to gather reliable time and energy metrics. For all applications, including both single- and multi-GPU applications, this step took approximately 7 minutes.

Although the phase-based approach has been implemented using SYCL, it is decoupled from the programming model, energy prediction model, and energy/time profiler used. In fact, energy profiling can be done with other available libraries, and the model can be replaced with others that predict time and energy values [8, 45, 66, 194] or can be entirely replaced by a profiling step that runs the application using all available frequencies on each device.

5.5.4.2 *Task DAG Creation with SYCL*

We implemented our DAG approach on top of SYCL. By default, a SYCL queue executes kernel functions based on dependency information. A SYCL program specifies the data needed to execute a particular kernel, including access modes and memory

types. The SYCL runtime ensures that kernels are executed in an order that ensures correctness by building a DAG of tasks at runtime. Generally, a DAG does not specify the order of execution for tasks but only establishes partial ordering constraints represented as edges. However, SYCL queues may operate in an in-order manner, where the schedule follows the same order of submissions to the queue. This limitation simplifies the analysis and is also used by related work [48].

To efficiently generate the task DAG of the applications, we used the oneAPI graph SYCL extension `sycl_ext_oneapi_graph` [33], which decouples command submission from command execution, allowing us to extract task order and dependencies during one-shot profiling. The profiling and modeling data also include the optimal frequency for each kernel, timings, loop information, and MPI communication times. The outcome of this step is a task DAG that represents the data dependencies between kernels and includes additional metadata to be used for phase detection.

5.5.4.3 *Phase Detection*

After creating the task DAG, we execute the phase detection algorithm described in Section 5.5.1. This algorithm takes the DAG as input and detects the phases, specifying where we need to set the frequency. To further clarify the phase detection process, Table 5.1 shows the simplified structure of some of the single-GPU applications used in our experiments. As shown, each application consists of multiple kernels, some of which are iterated inside the loops. On the right-hand side of the table, the *Min Energy* indicates the frequency selected for each kernel according to the *Min Energy* target, while *Runtime* represents the ratio of that kernel's runtime to the total runtime of all kernels, considering that each kernel may be called multiple times in the loops. The blue horizontal lines separate the phases detected by our algorithm, and the *Phase* value is the frequency chosen for all the kernels in that phase. When specifying the phases, our phase-detection algorithm follows three primary objectives determined within its cost function (Equation 5.1). First, according to the cost function, the algorithm groups the kernels that have the *Min Energy* within a specific close range together and assigns

Table 5.1: Single-GPU applications profile and detected phases on NVIDIA V100S.

	Code structure	Phase (MHz)	Min Energy Freq. (MHz)	Runtime (%)
ace	`for n in num_iters:`			
	`calculateForce()`	P1 1080	982	21.05 %
	`allenCahn()`		1080	55.52 %
	`boundCondPhi()`		772	1.11 %
	`thermalEquation()`	P2 202	202	16.06 %
	`boundCondU()`		630	1.11 %
	`swapGrid()`		532	5.15 %
aop	`generatePaths()`	P1 960	172	0.34 %
	`prepareSvd()`		960	96.09 %
	`for n in num_iters:`			
	`partialBeta()`	P2 202	202	2.38 %
	`finalBeta()`		150	0.05 %
	`updateCashflow()`		217	1.09 %
	`partialSums()`	P3 157	157	0.04 %
	`finalSum()`		682	0.01 %
mnist	`for n in num_iters:` `for i in train:`			
	`fwPass()`	P1 240	240	32.77 %
	`err()`		225	2.30 %
	`bwPass()`	P2 520	520	50.37 %
	`labeling()`		240	6.37 %
	`for i in test:`	P3 225		
	`fwPass()`		202	8.19 %

the frequency of the kernel with the longest runtime to all kernels in that group: In ace: `calculateForce()`, `allenCahn()`, and `boundCondPhi()` are grouped as Phase1 (P1), and they got the frequency of 1080 MHz which is related to `allenCahn()`, having the highest runtime percentage among the others. Second, based on the cost function, it aims to group the kernels invoked within

a loop into the same phase to prevent multiple frequency changes during the loop execution. In `aop`: `partialBeta()`, `finalBeta()`, and `updateCashflow()` are in a loop consecutively, and despite having different energy requirements, they are grouped into one phase. Third, it tries to unify phases with shorter runtimes with either the preceding or subsequent phase, which has a longer execution time. In `mnist`: kernels `bwPass()` and `labeling()` are grouped in one phase despite having different *Min Energy* frequencies. Given that the first kernel takes 50.37 % of the time and the second only 6.37 %, after assessing the cost of frequency change in the cost function, the algorithm maintains the frequency for the second kernel without alteration.

Our phase detection algorithm follows the same logic when considering MPI communications. By assessing the cost of frequency change and considering the MPI communication time, it enables the overlap of communication with the overhead of changing the frequency as in Section 5.5.2.

5.5.4.4 *Frequency Scaling with the SYnergy API*

After detecting the phases, we need to set the desired frequency for each phase. For this purpose, we used the SYnergy API [48], described in Section 5.3.

5.6 EXPERIMENTAL EVALUATION

In this section, we validate the accuracy of the algorithms defined in section 5.5.3, and we present the results of the experiments on single and multi-GPU/node. In all experiments, the **fine-grained** and **coarse-grained** approaches refer to two of the state-of-the-art methods, [48] and [196], respectively. The energy target for all experiments is *Min Energy*.

5.6.1 *Experimental Setup*

We used 4 different machines: two single-node single-GPU systems with NVIDIA and AMD GPUs, a single-node system featuring 4 Intel GPUs from the Intel® Tiber™ AI Cloud platform, and a 4-node cluster equipped with 4 NVIDIA GPUs per node.

For the single-GPU experiments, we used five benchmarks from the HeCBench SYCL benchmark suite [84]: `ace`, `aop`, `srad`, `metropolis`, and `mnist`, and evaluated them with the largest available input sizes. Multi-GPU experiments include two real-world applications: CloverLeaf [71], a compressible Euler equations solver on a Cartesian grid, and miniWeather [130], which is from the domain of weather-like flows simulation. The applications are fed with the maximum possible input sizes for each configuration (weak scaling). We utilized the PowerCap interface [144] for host energy measurements.

5.6.2 *Phase-detection Algorithms Evaluation*

This section evaluates the *Gr* and *Cl* algorithms in comparison to the optimal solution Dp. The algorithms are evaluated by comparing the energy savings achieved by their solutions. We compare the energy saved by *Gr* and *Cl* to the energy savings of the optimal solution (Dp).

In order to have diverse scenarios and evaluate applications with diverse characteristics, we selected twelve single-kernel benchmarks from SYCL-Bench [34], each with different characteristics and energy requirements. By combining these kernels in various ways, we developed several multi-kernel test applications (with up to 1000 kernel calls), each exhibiting distinct characteristics, allowing us to test the algorithms in a comprehensive manner. Considering that phase selection algorithms must consider the kernels' runtime, repetitions of kernels in loops, and optimal frequency, we classified our test applications into three distinct classes ($R1$, $R2$, $R3$) to encompass a broad range of scenarios observed in real-world applications.

$R1$: Multi-kernel applications where kernels have similar runtimes but different energy requirements with optimal frequencies ranging from 135 MHz to 1597 MHz. In this context, we ensure that no single kernel dominates in execution time to provide insight into how well the algorithms perform in such balanced scenarios.

$R2$: In contrast to the first class, some kernels have significantly longer runtimes than others. This setup tests how effectively the

algorithms handle unbalanced workloads, where certain kernels disproportionately affect overall energy consumption.

*R*3: The third class consists of test applications containing different numbers of kernels repeated within loops. These loops introduce complexity by adding repeated kernel invocations, potentially impacting the application's energy characteristics and runtime. This class assesses the ability of algorithms to optimize energy savings in scenarios where the application features more complex control flows and loop patterns.

In addition to these three classes, we define *E*1 and *E*2 as the edge border cases. *E*1 consists of tasks where the optimal frequency of task v_i differs from v_{i+1}, covering a scenario with varying energy behaviors in consecutive tasks. Unlike for *E*2, there is a drastic frequency change every four or eight tasks.

Table 5.2 summarizes the accuracy results of *Gr* and *Cl* in percentage compared to the *Dp* approach, which is the optimal solution, when minimizing *MIN_ENERGY* or *MIN_EDP*. The reported numbers are the average accuracy across all tested applications in each class. For both algorithms, in the *R*1-*R*3 test cases, we save 90 % to 100 % of the energy saved by the optimal algorithm. The clustering method consistently achieves greater energy savings than the Greedy approach, suggesting that an online Greedy solution that identifies phases while scheduling tasks may overlook certain optimization opportunities. Examining the edge cases, we observe that both algorithms achieve 83 % of the energy saved by the optimal solution in the E_1 scenario. In contrast, with E_2 increasing the similarity between consecutive tasks, the energy savings come closer to the optimal solution.

In the rest of the chapter, we employ the Clustering phase detection algorithm, which offers high accuracy near the optimal solution and has lower time complexity compared to the Dynamic Programming algorithm.

5.6.3 *Single-GPU Analysis*

PERFORMANCE Figure 5.6 illustrates the performance of the fine-grained and phase-based approaches compared to the coarse-grained approach for single-GPU benchmarks. For each bench-

Table 5.2: Phase detection algorithms accuracy in percentage.

	Greedy (G_r)		**Clustering (C_l)**	
App. Class	*MIN_ENERGY*	*MIN_EDP*	*MIN_ENERGY*	*MIN_EDP*
R1	95.1	100	96.5	100
R2	92.0	100	93.2	100
R3	97.3	100	98	100
E1	83	100	83	100
E2	94.1	100	99	100

mark, we fed the maximum possible input that each GPU could handle.

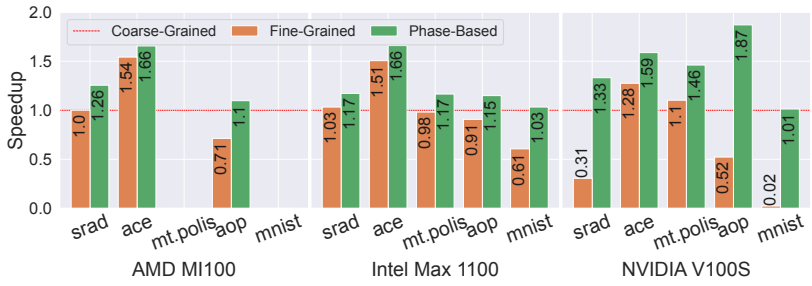


Figure 5.6: Single-GPU benchmarks performance (higher is better).

In this figure, the phase-based method consistently outperforms the other two methods across all three GPUs, proving that the proposed phase-based approach effectively reduces the overhead associated with frequency changes in the fine-grained approach. The largest gap between fine-grained and phase-based performance across various GPUs is observed in the NVIDIA GPU, where the phase-based method enhances the performance of fine-grained for `srad`, `aop`, and `mnist` by 4.3 ×, 3.6 ×, and 50.5 ×, while for Intel GPU, these are 1.13 ×, 1.26 ×, and 1.68 ×, respectively. This is mainly related to the inefficiency of the fine-grained approach for applications comprising several lightweight kernels invoked multiple times within loops, such as `mnist`. In this case, the gain of changing the frequency for each kernel is not enough to overcome the overhead. This results in a significant slowdown of the fine-grained method, particularly on NVIDIA

systems, mainly due to the high overhead of NVML compared to ROCm and Level Zero when changing the frequency (as illustrated in Figure 5.2). Also, the runtimes of each kernel differ across various GPUs. We have fed different input sizes to each GPU based on its available memory, resulting in varying speedup ratios when comparing the performance of different GPUs. Notably, we were not able to run `metropolis` and `mnist` on AMD due to the high memory requirements of these benchmarks.

HOST AND DEVICE ENERGY Figure 5.7 shows the corresponding energy consumption of the three approaches on both the device and host while running benchmarks on various GPUs. The host energy refers to the energy consumed by the CPU for the entire application.

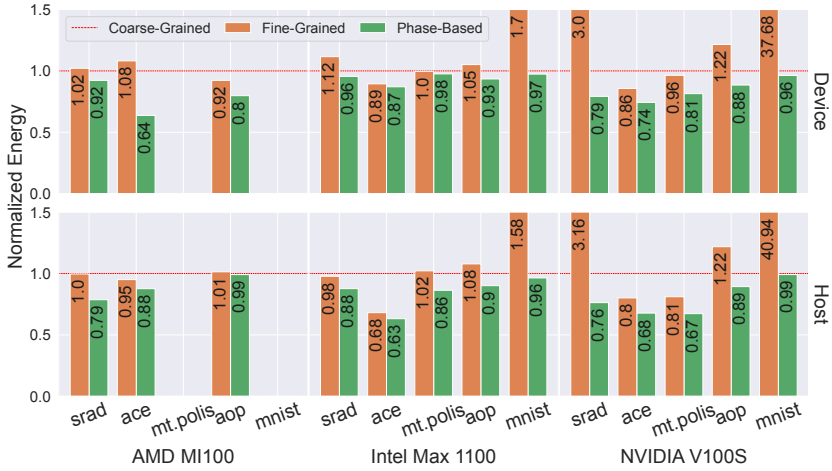


Figure 5.7: Normalized energy consumption on the device (above) and host (below) for single-GPU benchmarks (lower is better).

Similar to the performance in Figure 5.6, here in Figure 5.7, a similar consistent trend is observed: on the device and host, for all three GPUs, the phase-based approach exhibits lower energy consumption compared to both coarse-grained and fine-grained methods across all benchmarks. In particular, for the device energy of `aop` and `mnist`, the fine-grained approach consumes 1.12 × and 1.75 × more energy on Intel and 1.39 × and 39.25 × more energy on NVIDIA compared to the phase-based method. This increase is mainly attributed to the high number of GPU

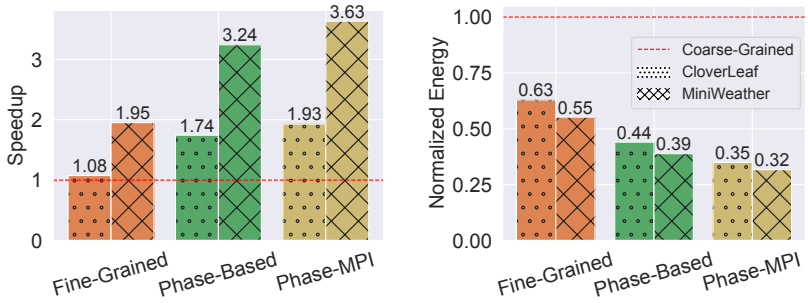
frequency changes occurring in these two benchmarks using the fine-grained approach. Regarding host energy, the decline in energy consumption of these benchmarks using the phase-based technique is linked to the reduction in their runtime. Our approach shortens the overall execution time of the benchmarks, leading to a decrease in both device and host (CPU) energy consumption.

5.6.4 *Real-world MPI Applications*

In this section, we compare our phase-based MPI-aware approach with both coarse- and fine-grained methods on multiple GPUs using two real-world MPI+SYCL applications: CloverLeaf and miniWeather.

SINGLE-NODE Figure 5.8 shows the speedup and normalized energy (relative to the coarse-grained approach) for these two applications on a single node equipped with four Intel GPUs (*Mach. C*). The *Phase-Based* represents the phase-based approach without considering the MPI communications, while the *Phase-MPI* represents the phase-based approach with MPI communications overlapping with the frequency scaling overhead. In Figure 5.8 (a), the phase-based approach consistently outperforms both the coarse- and fine-grained methods for the two applications. For CloverLeaf and miniWeather, it shows better performance 1.61× and 1.66× than the fine-grained approach, respectively. In addition, phase-MPI further improves the performance of the phase-based by 1.10× and 1.12× for CloverLeaf and miniWeather, respectively. Overall, the phase-MPI method improves the performance by 1.79× and 1.86× for CloverLeaf and miniWeather compared to the state-of-the-art fine-grained method.

In Figure 5.8 (b), the phase-based approach is more energy efficient than coarse- and fine-grained methods for the two applications. It achieves an energy savings of 30 % relative to the fine-grained method for both CloverLeaf and miniWeather. In phase-MPI, there is an additional improvement: compared to the phase-based method, phase-MPI reduces energy consumption by 21 % for CloverLeaf and 19 % for miniWeather. Overall, phase-MPI reduces the energy consumption of the state-of-the-



(a) Performance (higher is better) (b) Energy (lower is better)

Figure 5.8: The normalized performance and energy consumption of CloverLeaf and miniWeather on 4 Intel Max 1100 GPUs.

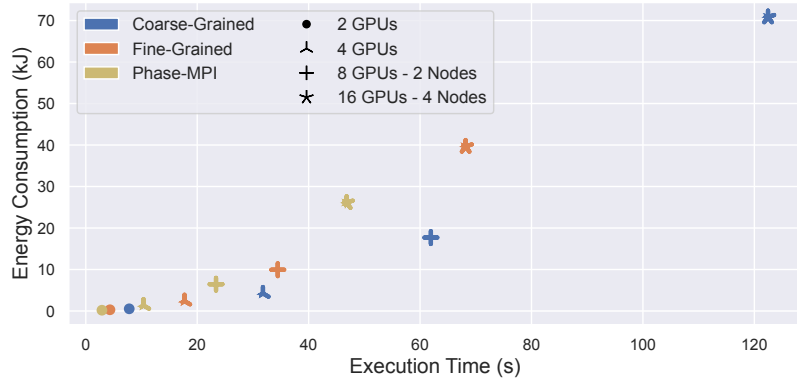


Figure 5.9: Energy scalability of miniWeather on 2-, 4-, 8- and 16 GPUs using different frequency scaling methods.

art fine-grained approach by 45 % and 43 % for CloverLeaf and miniWeather, respectively. As shown, our proposed phase-based method improves both energy efficiency and performance in real-world applications, each consisting of multiple kernels with varying energy requirements and execution times. Notably, miniWeather has 12 different kernel types, with 1940 kernel calls in total, and CloverLeaf consists of 37 different kernels with 7854 kernel invocations in total.

MULTI-NODES Figure 5.9 shows the energy scaling of miniWeather up to 16 NVIDIA A30 GPUs using a weak scaling approach. Both the performance and energy scaling results align

with our previous findings in this study. At any number of GPUs, the coarse-grained is always slower than the fine-grained, and they are both slower than our MPI-aware phase-based approach. This holds also true for energy efficiency, with our phase-based MPI-aware method proving to be more energy efficient than the other two approaches. For example, with 16 GPUs, our method saves energy by 35% and achieves a 1.45× higher performance compared to the state-of-the-art fine-grained method.

5.7 SUMMARY AND DISCUSSION

In this chapter, we introduced SYnergy, a SYCL-based library that provides a portable interface for energy profiling and dynamic voltage and frequency scaling (DVFS) across heterogeneous architectures. SYnergy abstracts vendor-specific energy management APIs, enabling programmers to implement energy-efficient computing at both the application and kernel granularity without dealing with low-level hardware complexities. Furthermore, we highlighted the overhead of frequency scaling across different GPUs and proposed a phase-based frequency scaling technique for heterogeneous systems that minimizes this overhead while preserving the energy efficiency and performance of fine-grained frequency scaling approaches. Our method identifies energy phases using the task DAG enriched with profiling data. We developed a phase-detection algorithm that identifies these phases and then sets an optimal frequency for each phase. We proposed a novel approach for MPI programs to further hide frequency change overhead by overlapping it with MPI communications. Our approach reduces overhead, increasing energy efficiency and performance compared to state-of-the-art methods. It improves real-world application performance by 1.45 × and saves 35 % of energy on 16 GPUs compared to the state-of-the-art fine-grained method.

6

DOMAIN-SPECIFIC ENERGY MODELING FOR FREQUENCY SCALING

DVFS provides an effective mechanism for trading performance and energy; however, its effectiveness depends on selecting a frequency configuration that optimize the energy-performance trade-off of a target application. The previous chapter demonstrated that DVFS and phase-based frequency selection can significantly improve energy efficiency in heterogeneous systems. However, these techniques inherently rely on the availability of accurate models capable of predicting how a given kernel will behave under different frequency settings. This chapter addresses the complementary challenge of predicting the optimal frequency for a kernel before execution, enabling proactive energy–performance optimizations. We first introduce a general-purpose energy model [48] designed to predict the optimal frequency of a kernel before execution. This model relies on static features extracted during compilation, such as the number of global and local memory operations, arithmetic instructions, and complex operations, to characterize the computational and memory behavior of a kernel. By feeding these features into machine-learning model we estimate the frequency setting that best satisfies different optimization goals, such as minimizing energy, maximizing performance, or reducing energy–delay product (EDP). Because it requires no application-specific profiling or annotations, this general-purpose model offers broad applicability across diverse GPU kernels. However, our analysis shows that the energy behavior is not fully captured by static computational features alone [25]. In many real-world applications, energy consumption and performance are strongly influenced by problem parameters, workload structure, and input-dependent behavior. These factors can significantly change the compute/memory balance of a kernel, leading to substantial prediction gaps when relying solely on general-purpose models. To address this challenge, we extend our methodology to develop domain-specific

energy models, tailored to individual applications. By incorporating features that reflect the internal structure of the algorithms and the characteristics of their workloads, we achieve significantly higher predictive accuracy. We validate this approach using two representative scientific applications with substantial societal and scientific impact: LiGen, a GPU-accelerated drug discovery platform, and Cronos, a magnetohydrodynamics solver widely used in astrophysical simulations.

6.1 RELATED WORK ON ENERGY MODELING

The energy sustainability of modern HPC systems is a critical concern from both an economic and an ecological standpoint. For this reason, many efforts in the scientific community are directed toward the development of tools that can contribute to reducing the environmental footprint of HPC systems. In particular, in recent years, many studies on modeling the energy consumption of scientific applications running on large-scale clusters have been published, enabling researchers to identify new strategies to reduce their energy consumption [29, 59, 66, 67, 109, 201]. Among them, Lopes *et al.* [109] proposed a model that relies on extensive GPU micro-benchmarking using a cache-aware roofline model. Wu *et al.* [201] studied the performance and energy models of an AMD GPU by using K-means clustering. Guerreiro *et al.* [66] made more improvements: they not only presented the approach of gathering performance events by micro-benchmarks in detail but also predicted how the GPU voltage scales. Such methods are often based on power limitation or frequency modulation of computing systems. In terms of power limitation, Remesh *et al.* [147] modeled the impact of dynamic power capping schemes in progress for a set of online HPC applications. Hao *et al.* [69] combined the powercap with uncore frequency scaling and proposed a machine learning modeling to predict the Pareto-optimal powercap configurations for achieving trade-offs among performance and energy consumption. In terms of frequency modulation, Ge *et al.* [60] applied fine-grained GPU core frequency and coarse-grained GPU memory frequency on a Kepler K20c GPU, but only analyzed three compute-intensive benchmarks to study the impact of GPU DVFS.

However, additional challenges and opportunities arise when these tools must be integrated into existing systems using workload managers [68, 184, 206, 207]. Furthermore, most of the above energy models are proposed for general purposes, which may lead to lower prediction accuracy of energy consumption targeting specific applications. Our proposal, starting from those challenges, specifically targets drug discovery and magnetohydrodynamics applications and outperforms the general-purpose energy model.

Other works propose tools to dynamically modulate frequency or adjust the job's power cap by analyzing features of the application at run-time [31, 43]. These tools lack an easily portable approach, either because designed for specific hardware or because they require a custom implementation in order to support different kinds of architectures. Differently, our approach provides a model-based and architecture-independent solution that only requires the range of frequency configuration to work on any architecture. Also, this solution can be easily integrated into other existing toolchains to drive the frequency selection of Pareto-optimal solutions.

DRUG DISCOVERY IN HPC How to dock a ligand inside a protein and how to estimate their interaction strength are well-known problems in the literature that we need to solve for different purposes [19, 192]. Indeed, we can find a wide range of software that can dock and score, covering the accuracy-performance spectrum [17, 125, 133, 204]. In this work, we focus on virtual screening, the initial stage of drug discovery that suggests which molecules to test *in-vitro* and *in-vivo*. Recent studies demonstrated how the introduction of this stage increases the success probability of the drug discovery pipeline [62]. Since this problem is embarrassingly parallel and the chemical space is huge, the size of the chemical library that we need to virtual screen is constrained only by the available computation effort. Recently, the largest campaigns of virtual screening performed billions [62] and a trillion [57] of protein-ligand evaluations against SARS-CoV-2. The latter used LiGen to carry out the dock and score computation, using almost all the nodes of two supercomputers (HPC5 at ENI and MARCONI100 at CINECA).

CRONOS CODE FOR ASTROPHYSICAL MAGNETOHYDRODYNAMICS The use of numerical methods is common in the field of astrophysics, to complement real behaviors observed in experiments with simulations. There is a wide range of applications developed for solving problems in hydrodynamics or magnetohydrodynamics, such as Racoon [42], Ramses [56], Nirvana [209], Amrvac [74, 87], Athena [177], Pluto [119], WENO–WOMBAT [41], each developed with a particular problem focus. The Cronos code [90, 91], instead, was developed so that it could easily adapt to the various problems investigated in the field of astrophysical modeling. In addition, the code also allows the solver to be used for other conservation laws that can be provided by the user. Recently, the Cronos code related to the solver has been ported to the SYCL [65] and Celerity [183] programming models to run on a single node and a distributed memory cluster, respectively.

6.2 MOTIVATION

In this section, we present an overview of how frequency scaling works on modern GPUs, and the key insights and challenges in applying frequency scaling to different applications and workloads. We present two speedup-energy characterization results performed on two applications running on an NVIDIA V100 GPU.

Saving Energy by Frequency Scaling

DVFS allows for exploring new opportunities in optimizing applications' energy consumption. While DVFS is able to significantly reduce the energy consumption of a task, this typically comes at the cost of performance; therefore, the problem is a multi-objective one, where we can explore different tradeoffs. Previous work shows that this trade-off is not trivial, and good energy savings can be achieved at the cost of a negligible loss in performance [47]. As the search space of frequency configurations can be very large, the typical approach is to highlight the configurations that have a dominant energy-performance trade-off. These "optimal" frequencies can be obtained by computing the Pareto-optimal solutions. These solutions represent the boundary of the

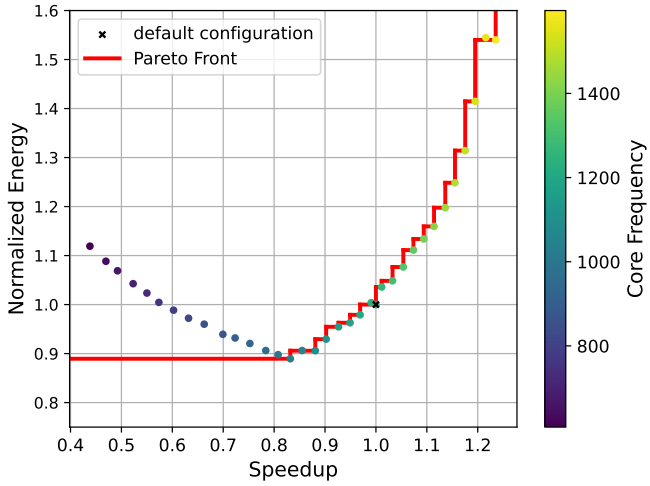
set of all possible outcomes where no improvement can be made in one objective without sacrificing the improvement of at least one other objective. Particularly, in our case the Pareto frontier would depict all the combinations of speedup and normalized energy where no other combination can achieve higher speedup without increasing the energy or reducing the energy without decreasing the speedup.

Heterogeneous systems are often programmed by using heterogeneous programming models such as OpenCL or SYCL, which provide code portability to different platforms. DVFS, however, is made available through different, vendor-specific, libraries such as NVML for NVIDIA GPUs or ROCm SMI for AMD GPUs. As our analysis is carried out on two SYCL-based applications, in this work we used the SYnergy API [48, 160], which interfaces with vendor-specific libraries from NVIDIA, AMD, and Intel GPUs and allows for portable energy profiling and frequency scaling.

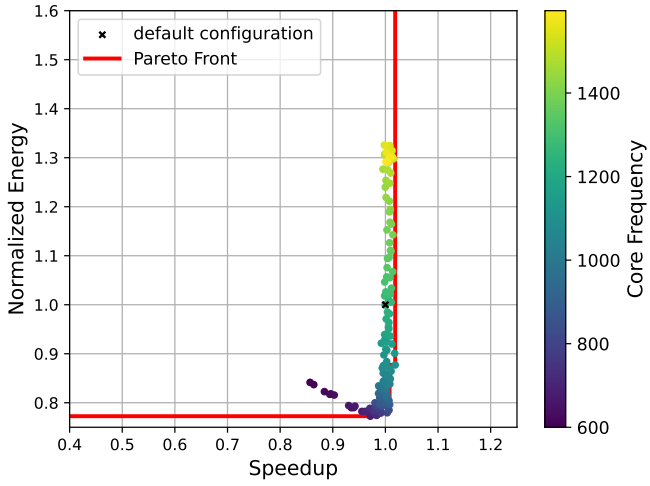
Energy Saving depends on the Application

Frequency scaling can have a very different impact on energy depending on the type of application. For compute-bound applications, we can have performance improvement at the cost of higher energy consumption by increasing the core frequency. On the other hand, memory-bound applications may benefit from core down-scaling to reduce energy consumption with small performance degradation. Figure 6.1 shows how different core frequency configurations can affect the speedup and energy of two real-world applications LiGen (Figure 6.1a) and Cronos (Figure 6.1b). The baselines used for computing the speedup and normalized energy are the time and energy of the application executed with the default core frequency. The percentage improvement or loss in speedup and normalized energy due to frequency scaling are computed with respect to the default frequency configuration.

In LiGen the core frequency can be raised to produce speedup improvements of up to 25%. While, decreasing the core frequency leads to smaller energy savings, up to 10%, at the expense of a 15% loss in performance. For the Cronos application, by decreas-



(a) LiGen



(b) Cronos

Figure 6.1: LiGen and Cronos multi-objective characterization.

ing the core frequency, we can achieve energy savings up to 22% with a speedup loss close to 0%. While raising the core frequency increases the energy consumption by 30% without providing any significant improvement in performance. Due to the multi-objective nature of the problem, there is no single frequency that achieves at the same time the best performance and energy consumption. However, using the Pareto-set we can explore the

solutions that can provide different trade-offs between speedup and energy. Through this analysis, it is possible to choose a Pareto-optimal frequency configuration according to the energy constraints of the specific use case.

Energy Saving depends on the Workload

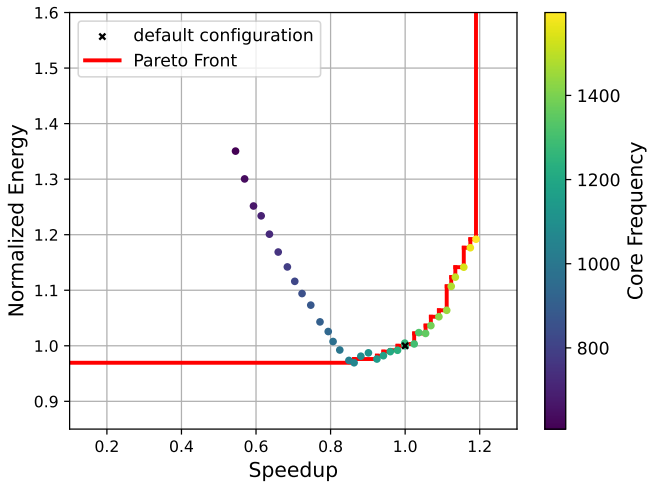
Exploring the trade-off between energy and speedup can be even more challenging as the energy behavior of the same application can be affected by the workload size.

Figure 6.2 shows speedup and normalized energy consumption for the LiGen application with an input size of 2 ligands *x* 89 atoms *x* 8 fragments and 10000 ligands *x* 89 atoms *x* 20 fragments. The red line highlights the Pareto-optimal solutions.

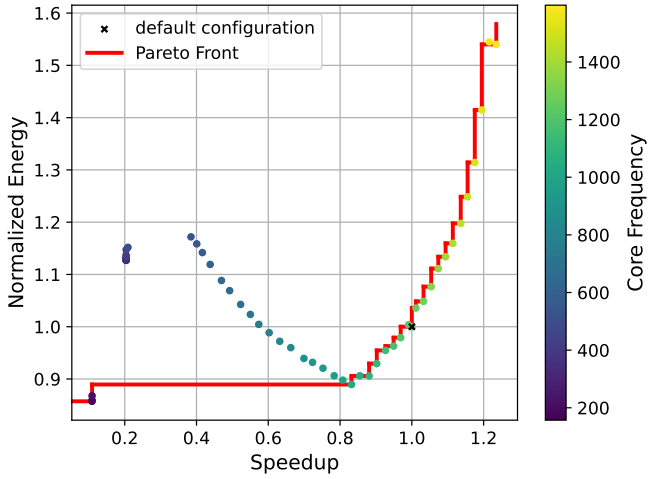
In Figure 6.2a, we notice a speedup up to 20% by increasing the core frequency while consuming 20% more energy. Differently, decreasing the core frequency does not provide any energy savings. For large input size Figure 6.2b, the energy behavior is the opposite. By decreasing the core frequency, we can notice energy saving is up to 10% with a speedup loss of 10%, while the same performance improvement in Figure 6.2a comes with a 60% increase in energy consumption.

Figure 6.3 shows the Pareto-optimal solutions for the Cronos application with an input size of 20*x*8*x*8 and 160*x*64*x*64. For a small input size (Figure 6.3a) we notice a speedup up to 3% with a 10% growth in energy consumption by increasing the core frequency. Differently, decreasing the core frequency does not provide any significant energy savings. For larger input sizes (Figure 6.3b), decreasing the core frequency allows for significant energy saving up to 20%, while only losing 1% speedup. On the other hand, we have no speedup improvement, but an increase in energy consumption of up to 30% by increasing the core frequency.

Based on these observations, we build a more accurate domain-specific multi-objective model that seeks to automatically predict Pareto-optimal frequency configurations of a specific application based on the input characteristics.



(a) Small input size

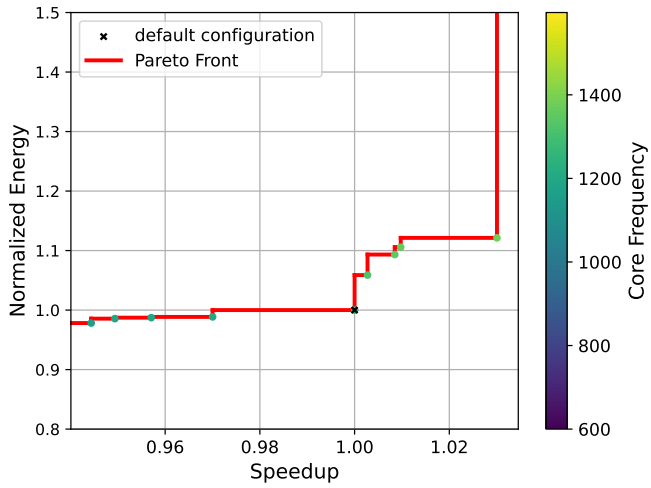


(b) Large input size

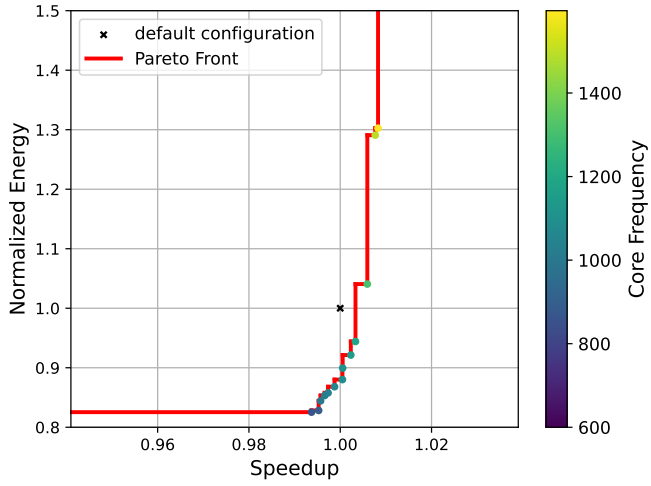
Figure 6.2: LiGen multi-objective characterization with Pareto-optimal solutions on input sizes of 2 ligands × 89 atoms × 8 fragments (a) and 10000 ligands × 89 atoms × 20 fragments (b).

6.3 ENERGY CHARACTERIZATION OF REAL WORLD APPLICATIONS

In this section, we provide an energy characterization of a magnetohydrodynamics code (Cronos) and a drug discovery framework (LiGen) on a set of hardware composed of NVIDIA V100 and



(a) Small input size



(b) Large input size

Figure 6.3: Cronos multi-objective characterization with Pareto-optimal solution on input size 20x8x8 (a) and 160x64x64 (b).

AMD MI100 GPUs. As defined in Section 6.2 the percentage improvement (or loss) in speedup and normalized energy are computed with respect to the default frequency configuration.

6.3.1 *Magnetohydrodynamics*

Cronos [90, 91, 183] is a magnetohydrodynamics (MHD) code developed for the solution of plasma-dynamical problems in astrophysics and space science. Algorithm 1 shows the structure of the Cronos code. The pseudocode outlines a simplified version of the basic structure of the Cronos application. The main computation happens in the function *computeChanges*, which computes the changes that occur for every element of the grid, as well as the CFL (Courant-Friedrichs-Lewy) value for every grid cell. This function represents a 13-point stencil application, where all cells can be computed in parallel. Due to the finite volume scheme involved, they need access to their neighborhood of 2 cells in each direction, i.e. 4 cells in each dimension. The next step in the algorithm is a reduction using the max operation on the CFL values of each cell. This step is parallelized using parallel reductions. After that the *integrateTime* function applies the previously computed changes to the grid, updating the state of every cell. This function is parallelized for every cell in the grid. The last step of the main computational loop is to update the boundary of the grid. This function only touches the outermost surfaces of the entire grid in parallel, rather than every cell like in the previous steps. In every timestep, the *timeDelta* is adjusted using the maximum *cfl* value which then advances the simulation by one timestep. The whole simulation runs until a preconfigured *endTime* is reached.

In this chapter, we explore how frequency scaling can affect the trade-off between energy and speedup as the input size increases.

6.3.1.1 *Energy scalability on grid dimensions*

Figure 6.4 and 6.5 show the speedup and normalized energy of the Cronos application executed on NVIDIA V100 and AMD MI100 GPUs with a small ($10x4x4$) and large ($160x64x64$) grid sizes. The red line highlights the Pareto-optimal solutions. On NVIDIA V100 GPU (Figure 6.4a and 6.4b) for both small and large grids, increasing the core frequency leads to higher energy

Algorithm 1 Cronos algorithm

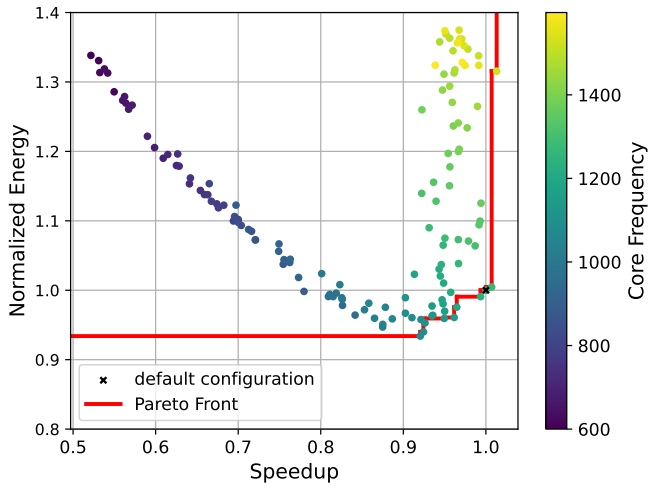
```
1: grid[SIZE_Z][SIZE_Y][SIZE_X]
2: grid ← initialise()
3: grid ← applyBoundary(grid)
4: while currentTime ≤ endTime do
5:     for substep = 0 to 2 do
6:         changeBuf[SIZE_Z][SIZE_Y][SIZE_X]
7:         cflBuf[SIZE_Z][SIZE_Y][SIZE_X]
8:         cflBuf, changeBuf ← computeChanges(grid)
9:         cfl ← reduce(cfl, cflBuf, max)
10:        grid ← integrateTime(grid, changeBuf, substep)
11:        grid ← applyBoundary(grid)
12:    end for
13:    timeDelta ← adjustTimestepDelta(timeDelta, cfl)
14:    currentTime ← currentTime + timeDelta
15: end while
```

consumption, up to 40%, with respect to the default configuration, without any improvement in performance. Energy-wise, as the grid size increases we can have a higher chance of energy saving with a loss of speedup close to 0% by lowering the core frequency.

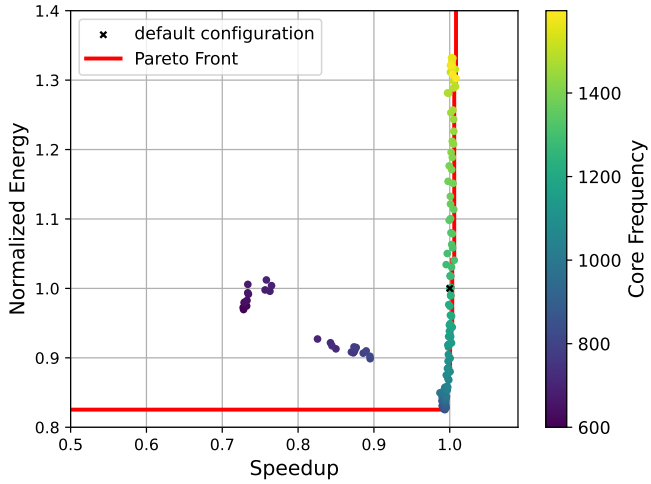
Figure 6.5a and 6.5b show the results on the AMD MI100 GPU. AMD GPUs do not have a default frequency, but instead use a performance level for dynamic frequency change. Normally the performance level is set to "automatic", which is the configuration that we consider as the default behavior of the GPU. We can see that the default setting is very close to the higher achievable speedup, but energy consumption can be reduced by lowering the frequency. In particular, for small grid sizes, the achievable energy saving is around 35% with a speedup loss of about 10%. On the other hand, larger grid sizes have lower energy savings, with a difference of about 5%, while the speedup loss remains the same as the small input size.

6.3.2 *Drug Discovery*

LiGen is the molecular docking engine part of the EXSCALATE [57] virtual screening platforms. It aims at finding the best drug



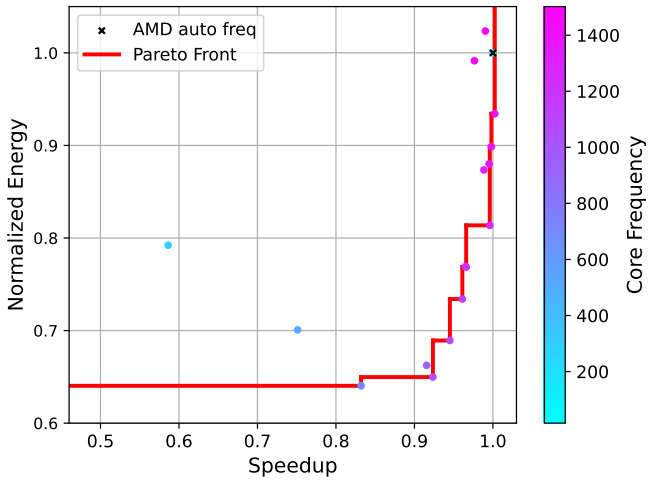
(a) 10x4x4 grid size



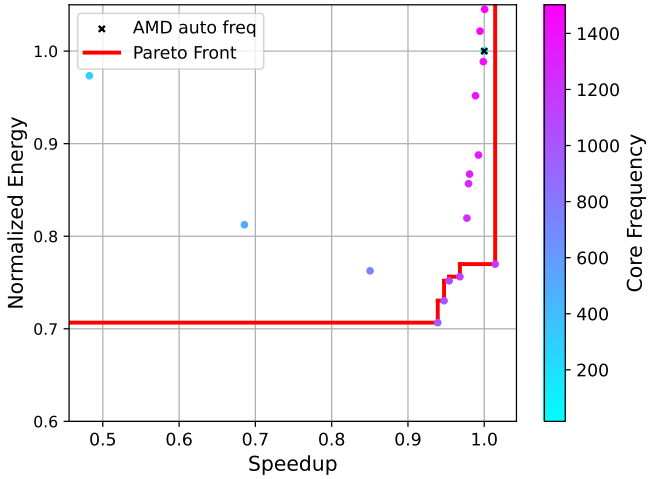
(b) 160x64x64 grid size

Figure 6.4: Cronos multi-objective characterization on NVIDIA V100 with small (10x4x4) and large (160x64x64) grid size.

candidates to test *in-vitro* and *in-vivo* in a drug discovery process. In this context, we have a target protein representing the disease and a very large chemical library of ligands (small molecules) representing the possible drugs. The platform's goal is to rank the chemical library according to the ligand-protein interaction strength to identify the best candidates to forward to the next



(a) 10x4x4 grid size



(b) 160x64x64 grid size

Figure 6.5: Cronos multi-objective characterization on AMD MI100 with small (10x4x4) and large (160x64x64) grid size.

stages of drug discovery. Algorithm 2 shows the two main tasks that allow to compute the interaction strength. The first one is named *dock*, and it aims at estimating the 3D displacement of the ligand's atoms when it interacts with the protein, i.e., dock the ligand inside the protein (lines 2-12). The second task is named *score*, and it aims at computing the interaction strength of the ligand-protein pair (lines 13-18). LiGen is the application that

carries out these tasks. It provides the computation kernel for different devices with the C++*17*, CUDA, and SYCL implementations.

All the ligand-protein evaluations are independent. Thus, the problem is embarrassing parallel. Moreover, the protein is a constant for each virtual screening campaign. Thus, we can analyze the algorithm complexity using features of the ligands. From the asymptotic complexity analysis [57, 193], the complexity of the single ligand-protein evaluation depends on the ligand's number of atoms and the number of rotamers. The latter is a subset of the ligand's bonds that LiGen can use to change its geometric shape without altering physical and chemical properties. In particular, each rotamer splits the ligand's atoms into two disjoint sets that can rotate independently along the rotamer axis. In this document, we refer to each set as a ligand *fragment*.

In this section, we provide a multi-objective analysis highlighting how frequency scaling can affect LiGen's energy consumption and performance when given different numbers and types of ligands as inputs.

6.3.2.1 *Energy scalability on atoms and fragments*

For this analysis, we use raw values instead of normalized values to avoid an overlap of the energy curves and have a better visualization of the data as the input size increases. Figure 6.6 shows the energy and time behavior of the LiGen application running on NVIDIA V100 GPU with 100000 ligands. Each ligand is composed of a small (Figure 6.6a) and a large number of atoms (Figure 6.6b) while varying the number of fragments. As the number of fragments increases, both the energy consumption and time increase. In particular, this behavior is more evident with a large number of atoms.

Figure 6.7 shows the results using the same experiments performed in Figure 6.6 on an AMD MI100 GPU.
As with the NVIDIA V100 GPU, increasing the number of fragments produces an increase in energy consumption and time. Additionally, on AMD MI100 we can observe greater energy consumption variations with a large atom size, as the number of

Algorithm 2 LiGen virtual screening algorithm

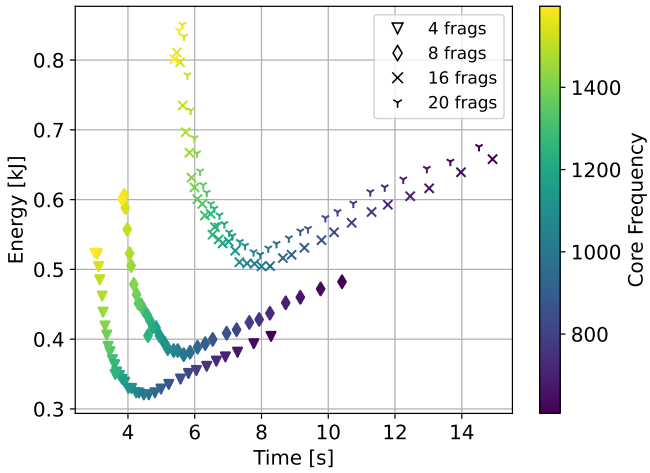
```
1: scores ← ∅
2: poses ← ∅
3: for i = 0 to num_restart do
4:     pose ← initialize_pose(ligand, i)
5:     pose ← align(pose, target)
6:     for n = 0 to num_iterations do
7:         for each fragment in pose.fragments do
8:             pose ← optimize(pose, fragment, target)
9:         end for
10:    end for
11:    pose ← evaluate(pose, target)
12:    poses ← poses ∪ {pose}
13: end for
14: poses ← clip(sort(poses), max_num_poses)
15: for each pose in poses do
16:    score ← compute_score(pose, target)
17:    scores ← scores ∪ {score}
18: end for
19: return max(scores)
```

fragments increases. Compared with the NVIDIA V100 results, both energy and time are higher on AMD MI100.

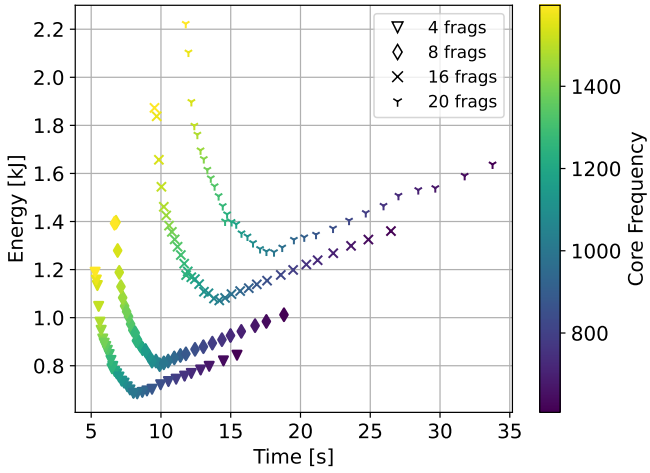
In Figure 6.8 we explore the energy and time behavior of LiGen with a fixed number of fragments while increasing the number of atoms in the ligands (Figure 6.8a and Figure 6.8b). By only increasing the number of atoms, we can notice more variability in energy consumption with respect to Figure 6.6. The same experiments with a fixed fragment size have been reproduced on the AMD MI100 GPU, showing a similar behavior that can be seen in Figure 6.9.

6.3.2.2 *Energy scalability on number of ligands*

All the computation performed during the docking and scoring phases of LiGen can be easily parallelized by allowing each kernel on the GPU to compute several ligands simultaneously. By scaling the number of ligands computed in a kernel, we may



(a) 31 atoms

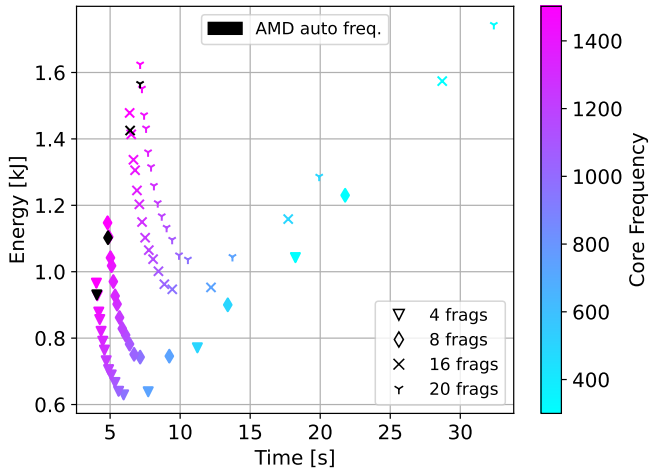


(b) 89 atoms

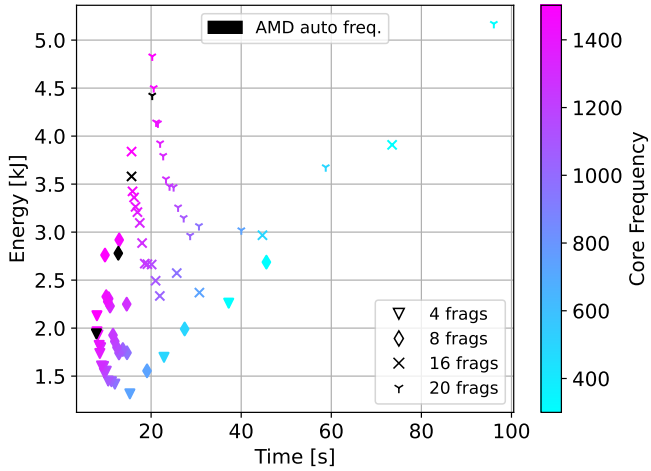
Figure 6.6: LiGen multi-objective characterization on NVIDIA V100 scaling the number of fragments with a fixed number of atoms.

have a different utilization of the GPU resources that may affect the energy behavior of the whole application.

Figure 6.10 shows the speedup and normalized energy with the Pareto-optimal solutions of the LiGen application on a set of small and large ligands. On NVIDIA V100 (Figure 6.10a and Figure 6.10b) we may have a speedup up to 22% with an en-



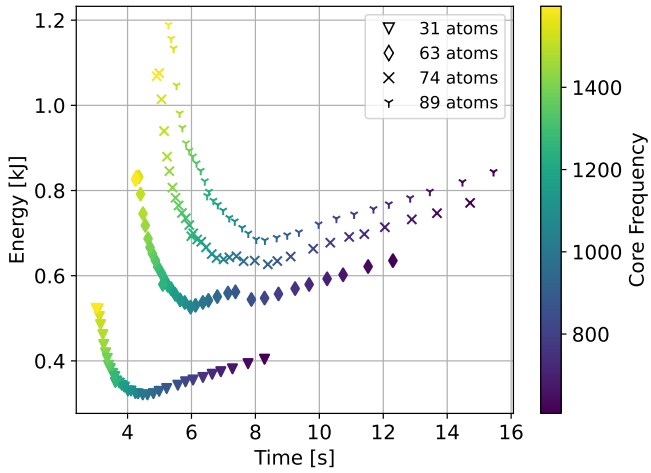
(a) 31 atoms



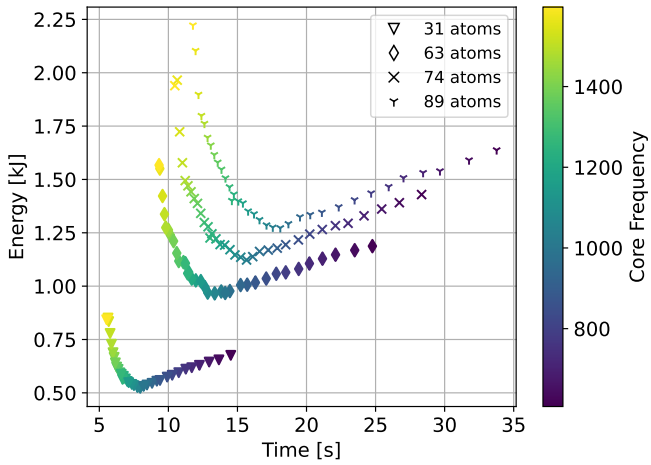
(b) 89 atoms

Figure 6.7: LiGen multi-objective characterization on AMD MI100 scaling the number of fragments (4, 8, 16, 20) with a fixed atom size.

ergy consumption increase of 60% by using a large input size. Differently, on small input sizes, the achieved speedup is lower, up to 15%, while the energy consumed is 30% compared to the 60% on large input sizes. Furthermore, on small input, we have more chance of saving energy. Looking at the Pareto-optimal solutions on the red line we can select some frequency config-



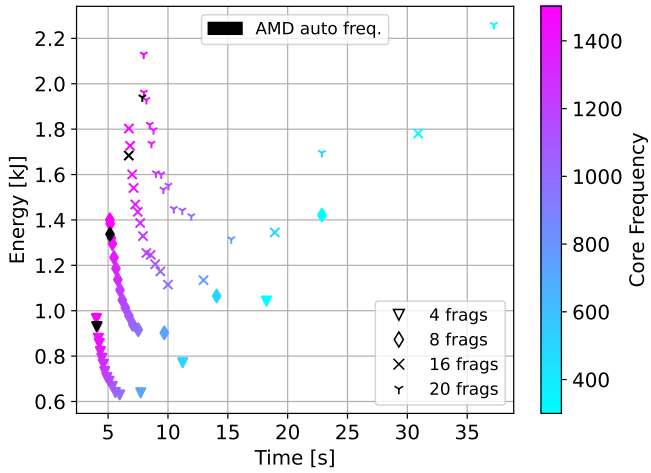
(a) 4 fragments



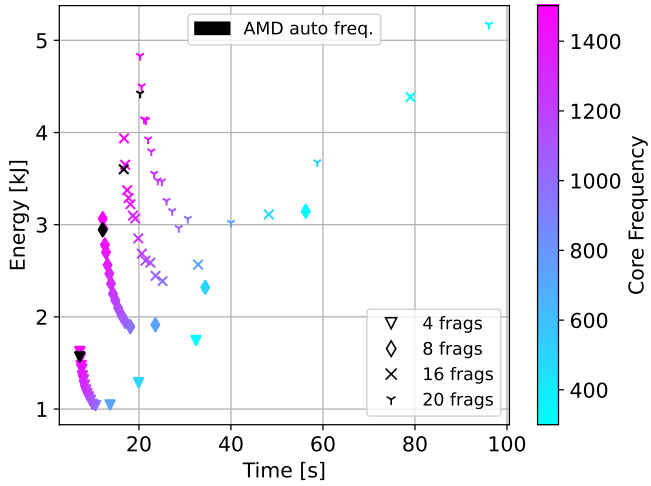
(b) 20 fragments

Figure 6.8: LiGen multi-objective characterization on NVIDIA V100 scaling the number of atoms (31, 63, 74, 89) with a fixed fragment size.

urations that provide a 10% of energy saving with a speedup loss of 5%. For the AMD MI100 GPU (Figure 6.10c and Figure 6.10d), the normalized energy and speedup are computed with respect to the frequency automatically selected by the GPU. This frequency always performs better on both small and large inputs. The Pareto-optimal solutions highlight an energy behavior simi-



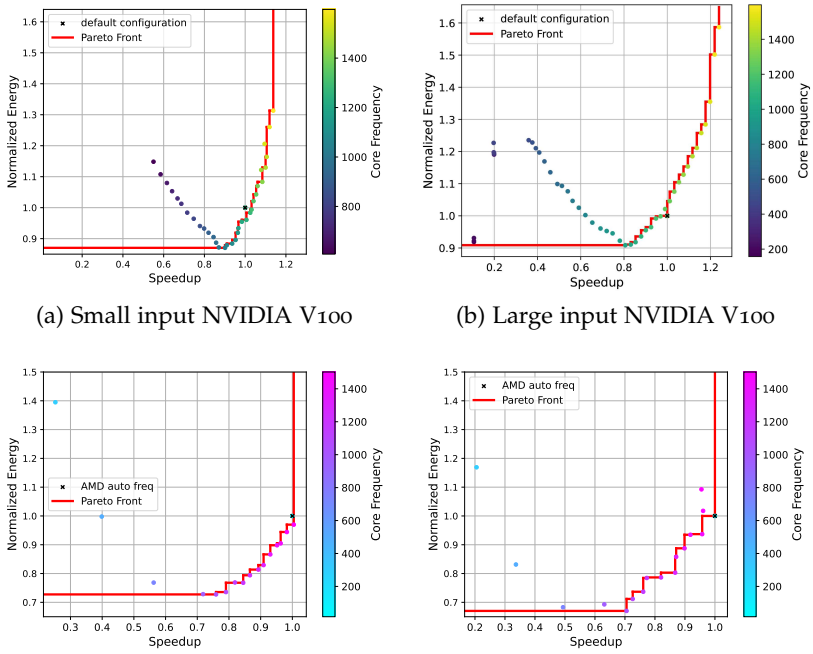
(a) 4 fragments



(b) 20 fragments

Figure 6.9: LiGen multi-objective characterization on AMD MI100 scaling the number of atoms (31, 63, 74, 89) with a fixed fragment size.

lar to the results on NVIDIA V100. With small input sizes, we can select Pareto-optimal frequency configurations that achieve 20% energy saving with a speedup loss of 10%. While for large inputs the same energy saving comes at the cost of higher speedup loss.



(a) Small input NVIDIA V100

(b) Large input NVIDIA V100

(c) Small input AMD MI100

(d) Large input AMD MI100

Figure 6.10: LiGen multi-objective characterization on NVIDIA V100 and AMD MI100 with small (256 ligands x 31 atoms x 4 frag.) and large (10000 ligands x 89 atoms x 20 frag.) input size.

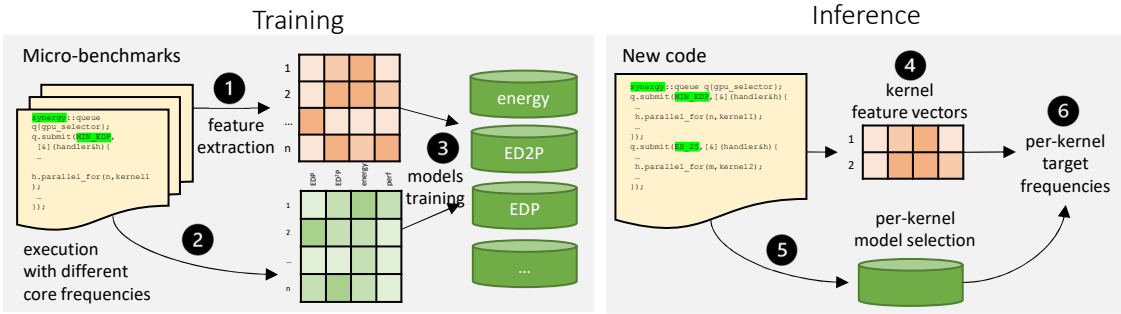


Figure 6.11: General-purpose machine learning based energy models

6.4 GENERAL PURPOSE ENERGY MODELING

To optimize the energy efficiency brought by GPU DVFS [48] we proposed a machine-learning model that select the optimal

frequency of the GPU according to an energy target metric such as minimum energy (`MIN_ENERGY`), maximum performance (`MAX_PERF`) or energy delay product (`MIN_EDP`). The model is based on typical two-phase modeling with supervised learning: training phase and prediction phase. Figure 6.11 shows both phases. In the training phase, we first build 106 carefully-designed micro-benchmarks and extract a set of static features of each micro-benchmark ❶. The features description is provided in Table 6.1.

Table 6.1: Static code features.

Feature	Description
k_{int_add}	integer additions and subtractions
k_{int_mul}	integer multiplications
k_{int_div}	integer divisions
k_{int_bw}	integer bitwise operations
k_{float_add}	floating point additions and subtractions
k_{float_mul}	floating point multiplications
k_{float_div}	floating point divisions
k_{sf}	special functions
k_{gl_access}	global memory accesses
k_{loc_access}	local memory accesses

Successively, each micro-benchmark is executed with various frequency configurations ❷ to obtain energy measurements. Those measurements, together with frequency configurations and static features, are used to train the normalized energy consumption model ❸. All the micro-benchmarks are built to stress one or more features that characterize the device's energy consumption and they can be used to train a general-purpose model that works on different applications. In the prediction phase, static code features are extracted from a new input code ❹. Those features, ❺ combined with frequency configurations and the previously trained models, are used to predict the normalized energy consumption of a new code. Once we have this value we can use it to predict the optimal frequency according to the enrgy target specified ❻. The approach, takes advantage of static code features to predict the normalized energy consumption of a

new code without executing it. However, the static code features have more weight on computing ability, which leads to a higher prediction accuracy of compute-bound applications and lower prediction accuracy of memory-bound applications.

6.5 DOMAIN-SPECIFIC ENERGY MODELING

As demonstrated in Section 6.3 the energy behavior of real-world applications may depend on the input characteristics. State-of-the-art general-purpose models only consider static features without taking into account how the application can be affected by different workloads and input types. To improve the accuracy of general-purpose models we provide two domain-specific energy models that rely on the input characteristics of a program to predict the speedup and normalized energy for each device core frequencies.

The methodology used for modeling is based on supervised learning, in which each model is built during the training phase using the general purpose features described in Table 6.11 extendend with the domain-specific one. Then, the prediction phase takes unseen features as input and generates execution time and energy consumption, which are used to compute speedup and normalized energy for each frequency configuration of the target hardware.

6.5.0.1 *Features selection*

The domain-specific nature of the models implies that there is not a said set of features for every application, but rather, for each application different features must be chosen through an in-depth time and energy analysis. In our case, the Cronos and LiGen models are built on top of the energy characterization analysis in Section 6.3, which provides interesting insights to select the features that best represent the energy characterization of the target application.

MAGNETOHYDRODYNAMICS The characterization in Figure 6.4 and 6.5 shows that the Cronos application exhibits different behaviors when considering varying grid sizes. For this reason,

we model the behavior of the application through the use of the grid size on the x,y, and z-axis.

DRUG DISCOVERY As expected, the energy consumed by the LiGen application increases as the number of ligands increases (Figure 6.10). Furthermore, Figure 6.6 and 6.8 show that the LiGen energy and time are strictly related to the structure of the ligands, e.g., the number of fragments and atoms in each ligand. To fully capture all these aspects, our domain-specific models use the number of ligands, fragments, and atoms as features.

The final features for the two applications are summarized in Table 6.2.

Table 6.2: Domain-specific model features.

Application	Features
Cronos	$f_{grid\_x}, f_{grid\_y}, f_{grid\_z}$
LiGen	$f_{ligands}, f_{fragments}, f_{atoms}$

6.5.0.2 *Training phase*

The workflow of the training phase is illustrated in Figure 6.12. For each application, the outputs of this phase are two models: one for execution time and one for energy consumption. In order to build the training set, the models require both the input features $\vec{f}$ and the ground truth values for the execution time (t) and energy consumption (e) of the application. Furthermore, as we must capture the behavior of the application (1) with different frequencies, each input must be executed for each (or a part) of the frequency configuration (c) of the target hardware (2). Once the training dataset $D = \{s : s = (\vec{f}, c, t, e)\}$ is built (3), we apply different machine learning algorithms to build the two models $T(\vec{f}, c)$ for execution time (4) and $E(\vec{f}, c)$ for energy consumption (5).

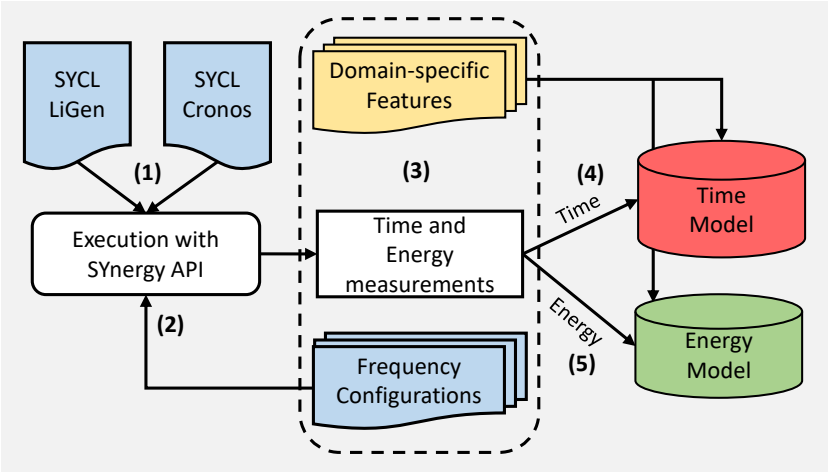


Figure 6.12: Domain-specific model training phase

6.5.0.3 *Prediction phase*

The workflow of the prediction phase is presented in Figure 6.13. During the prediction phase, given a new features vector $\vec{f}'$ (1) and a frequency configuration c' (2), the models are used to predict $\hat{t} = T(\vec{f}', c')$ and $\hat{e} = E(\vec{f}', c')$. Once the energy and time predictions for all frequency configurations are available, the predicted speedup (3) and normalized energy (4) can be computed, taking as a baseline the predicted values for the default frequency configuration. Finally, speedup and normalized energy are used to select the Pareto-optimal solutions.

6.6 EXPERIMENTAL EVALUATION

In this section, we present the experimental evaluation of the proposed models, along with a comparison with the state-of-the-art model that is based on static code features.

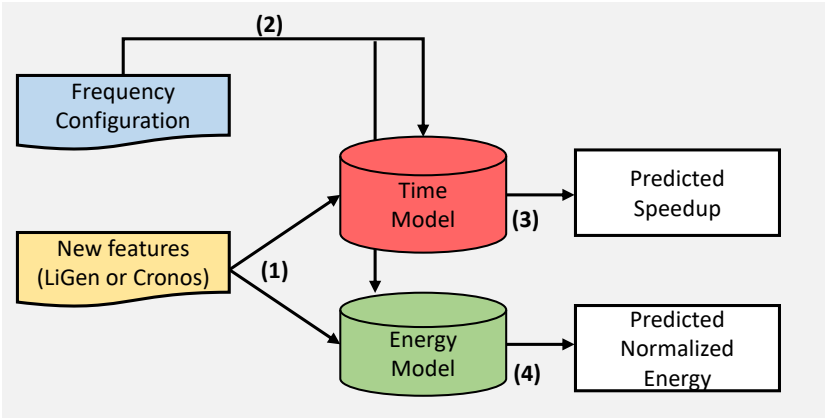


Figure 6.13: Domain-specific model prediction phase

6.6.1 *Experimental Setup*

The models' training dataset is obtained by launching the application on a system using Ubuntu 22.04, an Intel Xeon Gold 5218 CPU, and an NVIDIA V100 GPU with 32 GB of High Bandwidth Memory (HBM2). The NVIDIA driver version and CUDA version are respectively 530.30.02 and 12.1. The V100 GPU supports one memory frequency (1107 MHz) and 196 core frequencies from 135 MHz to 1597 MHz. The execution time is profiled through the standard C++ library, while the energy consumption is profiled through the SYnergy API. Each experiment is repeated five times to reduce the impact of any outliers. We launched the two applications, Cronos and LiGen, with many different inputs. For Cronos, we use five grid configurations, starting from a $10x4x4$ grid up to a $160x64x64$ grid. LiGen experiments are executed on different numbers of ligands that have varying numbers of fragments and atoms. Each experiment can be represented by a tuple $(l, a, f) \in \{2, 16, 1024, 4096, 10000\} \times \{31, 63, 71, 89\} \times \{4, 8, 16, 20\}$, where l is the number of ligands, a is the number of atoms and f is the number of fragments. To validate our approach the applications were executed with all the frequencies of the V100 GPU and then we highlighted the point predicted by the general-purpose model and domain-specific model.

6.6.2 *Domain-specific versus General-purpose Models Accuracy*

The general-purpose models are built using a set of well-defined micro-benchmarks that stress different architectural components of the target hardware [47, 48, 66]. Then, the prediction phase for general-purpose models uses the static code features of the application to predict the energy and execution time values.

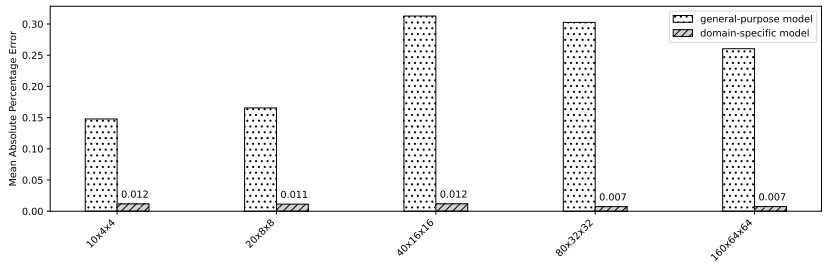
Differently, the presented domain-specific models are trained on a specific application and are meant to be used for that same application. Thus, we validate these models by using leave-one-out cross-validation over the domain-specific features dataset as defined in Table 6.2. Specifically, for each different input feature $\vec{f}$ we build a set D_v for validation and a set D_t for training, defined as follows:

$$D_v = \{s \in D : s \text{ has input features } \vec{f}\}$$
$$D_t = D \setminus D_v$$

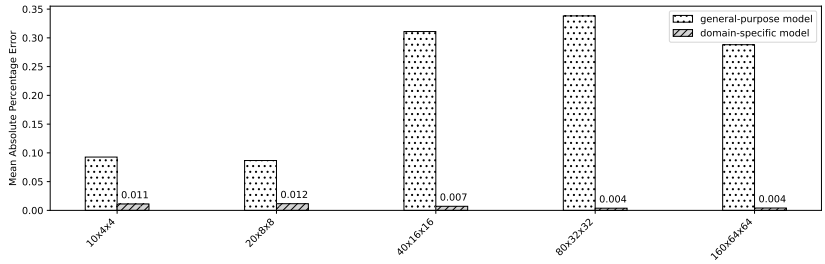
where $s = (\vec{f}, c, t, e)$ as defined in Section 6.5.0.2.

6.6.2.1 *Accuracy of Speedup and Normalized Energy predictions*

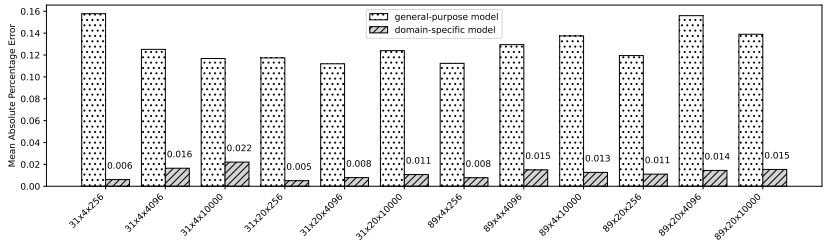
Our approach is built on top of two models: one is to predict speedup and the other is to predict normalized energy. Through the *scikit-learn* Python library, we perform the training phase of these models using different kinds of regression algorithms (Linear, Lasso, SVR_RBF, Random Forest) and finally select the algorithm that achieves the highest accuracy for each model. Random Forest achieves the maximum accuracy for both normalized energy and speedup models. We performed a hyperparameter tuning of the Random Forest regression algorithm through a grid search method. The grid space has been defined by three parameters: the maximum depth of the tree (max_depth); the number of trees in the forest (n_estimators); the number of features to consider when looking for the best split (max_features). As a result, the default parameter performs better for both the speedup and energy models. The accuracy comparison is based on the analysis of the mean absolute percentage error (MAPE) of the models. For each application, we first compute the absolute



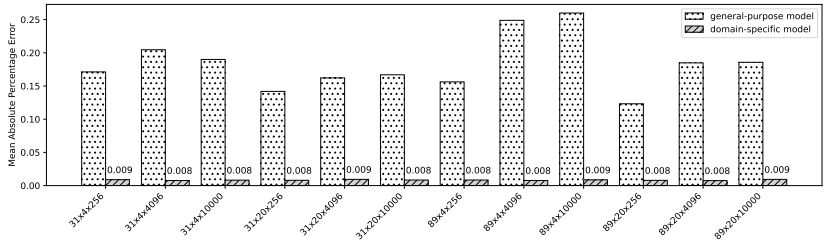
(a) Cronos speedup prediction error



(b) Cronos normalized energy prediction error



(c) LiGen speedup prediction error



(d) LiGen normalized energy prediction error

Figure 6.14: Comparison of prediction accuracy for Cronos and LiGen using the general-purpose and domain-specific models.

percentage error between the predicted values by the two models and the true values obtained through running the application

with each frequency configuration. Then, we measure the prediction accuracy as the mean of the absolute percentage error over all the frequency configurations. Figure 6.14 shows the speedup and normalized energy prediction accuracy of the two models expressed as MAPE for both applications Cronos and LiGen on different input workloads. The domain-specific models achieve a lower error for both applications on every input and are at least 10 times more accurate than the general-purpose model.

6.6.2.2 *Accuracy of Predicted Pareto Set*

The final purpose of the models is to predict Pareto-optimal frequency configurations that allow for energy saving or a boost in performance. In the accuracy analysis, the general-purpose and domain-specific models are both used to predict the execution time and the energy consumption for each frequency configuration. Then to obtain the Pareto-optimal frequency configurations, we

1. compute the predicted speedup and normalized energy, using the predicted values of the default frequency configuration as baseline;
2. compute the predicted Pareto-optimal solutions;
3. create the Pareto-optimal frequency configuration set by selecting the frequency configurations associated with each predicted Pareto-optimal solution.

This process produces two sets of predicted Pareto-optimal frequency configurations, one for the general-purpose models and one for the domain-specific ones, that are comparable with the actual Pareto-optimal frequency configuration.

To assess the models' accuracy we use the speedup and normalized energy obtained running the applications with the predicted Pareto-optimal frequency configurations. In fact, these are the real values that would be obtained if the applications were executed with the predicted Pareto-optimal frequencies. Analyzing the accuracy of the predicted Pareto-optimal solutions is not trivial as the actual points obtained by the predicted Pareto-optimal frequency configurations are not for sure Pareto-optimal.

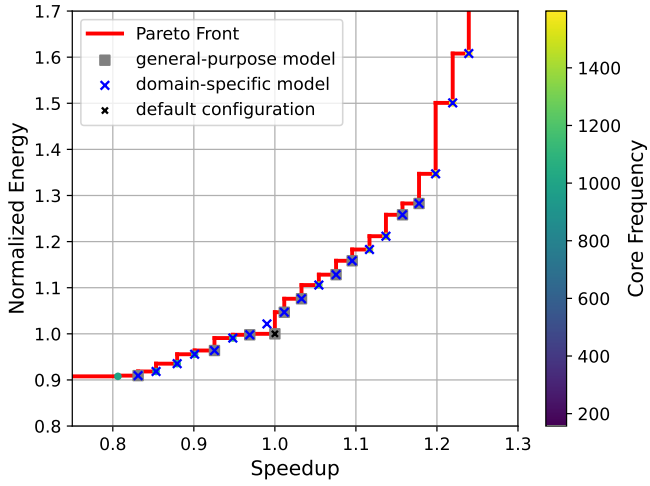
In general, a better Pareto approximation is a set of solutions that, in terms of speedup and normalized energy, is the closest to the real Pareto-optimal one. Moreover, the number of predicted Pareto-optimal frequency configurations that exactly match the true Pareto-optimal frequency configurations can be considered as a metric when comparing accuracy between models.

Figure 6.15 shows the predicted Pareto-optimal solutions for the LiGen and Cronos on the same input of Figure 6.10b and 6.4b. The red line represents the true Pareto-optimal configurations, while the gray squares and the blue crosses represent respectively the Pareto-optimal solutions predicted by the general-purpose and the domain-specific models. The predicted frequency configurations of the general-purpose model for LiGen fully match the true Pareto set, while the domain-specific model predicts a configuration that is not in the true Pareto set. On the other hand, the domain-specific model predicts more Pareto-optimal points than the general-purpose model. This is a positive trait of the model, as it allows us to explore deeply the trade-off between speedup and normalized energy. For example, in Figure 6.15a only the domain-specific model is able to predict the highest-performing frequencies achieving around 23% of speedup compared to the 18% speedup obtained by using the general-purpose model prediction. For the Cronos application, the general-purpose and the domain-specific models have more difficulty in predicting the speedup as the values are very close to each other. Therefore, differently from LiGen, both models predict fewer frequency configurations that exactly match the true Pareto-optimal configurations. With respect to energy, the domain-specific model is more precise in the prediction of frequency configurations that correspond to normalized energy values in line with the true Pareto-optimal solutions.

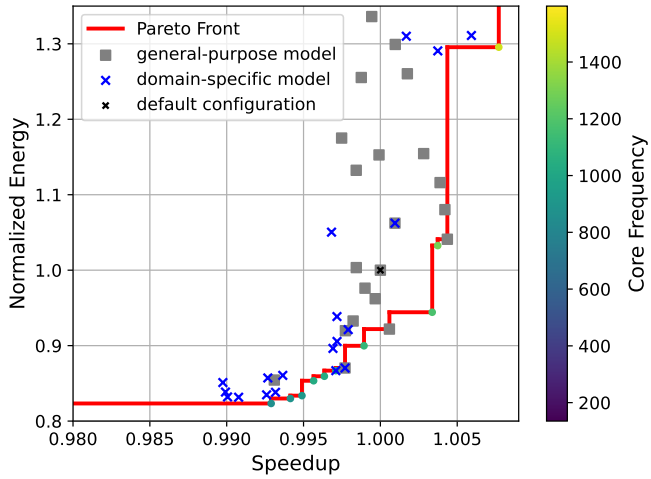
6.7 SUMMARY AND DISCUSSION

In this chapter, we presented a comprehensive methodology for DVFS prediction on GPUs, spanning from general-purpose energy modeling to domain-specific approaches.

We first introduced a general-purpose machine learning model capable of predicting the optimal GPU frequency for different en-



(a) LiGen
($10000x89x20$)



(b) Cronos
($160x64x64$)

Figure 6.15: LiGen and Cronos Pareto-optimal solution predicted by the general-purpose and domain-specific models compared with the true Pareto-set (red line).

ergy targets, such as minimum energy, maximum performance, and minimum energy-delay product. The model leverages a two-phase approach: a training phase, where micro-benchmarks are executed across multiple frequency configurations to extract

static code features and corresponding energy measurements, and a prediction phase, where features of new kernels are used to estimate normalized energy and select the optimal frequency without executing the code. However, in real-world scenarios, we observed that the input size and application-specific parameters can significantly affect the energy behavior, reducing the prediction accuracy of general-purpose models that rely only on static code features. To overcome the limitations of general-purpose models, we proposed domain-specific energy models tailored to the characteristics of each target application, and we validated the model by using two real-world applications in the field of drug discovery (i.e LiGen) and magnetohydrodynamics (i.e. Cronos). By incorporating input-dependent features such as grid dimensions for Cronos or the number of ligands, atoms, and fragments for LiGen, these models capture the behavior of real-world workloads under different frequency configurations. The training phase builds separate models for execution time and energy consumption using supervised learning, while the prediction phase estimates speedup and normalized energy for all frequency configurations, enabling the identification of Pareto-optimal solutions.

Our experimental evaluation on NVIDIA V100 and AMD MI100 demonstrates the effectiveness of domain-specific models. We showed that domain-specific models achieve significantly higher accuracy in predicting both speedup and normalized energy compared to general-purpose models. In particular, domain-specific models are able to predict additional Pareto-optimal frequency configurations that are not captured by general-purpose models, allowing for a more comprehensive exploration of the trade-off between performance and energy efficiency. Overall, this chapter demonstrates that combining static architectural features with application- and input-specific characteristics allows highly accurate energy modeling and efficient frequency selection.

7

PROGRAMMING RISC-V SIMD ARCHITECTURES

RISC-V is an open-standard Instruction Set Architecture (ISA) [197] that has gained significant attention in recent years due to its modular, extensible, and open-source design. RISC-V is built on the principles of Reduced Instruction Set Computing (RISC), offering a streamlined and flexible base architecture that can be extended with a variety of optional features. This flexibility has made RISC-V a popular choice in academia, research, and industry, driving innovations across a wide spectrum of computing applications, from embedded systems to HPC. In modern workloads, data-level parallelism plays a crucial role in achieving high performance, and SIMD units have become a fundamental mechanism for accelerating computations efficiently. To address these needs within its modular design, RISC-V introduces the RISC-V Vector Extension (RVV), a flexible and scalable approach to SIMD execution. Unlike fixed-width SIMD extensions such as Intel AVX or ARM NEON, RVV allows hardware vendors to choose vector widths freely and enables software to remain portable across heterogeneous RISC-V devices. The combination of an open, royalty-free ISA and portability across different hardware makes RVV a promising basis for the development of modern SIMD architectures. Efficiently exploiting RVV's SIMD capabilities requires appropriate programming abstractions that allow developers to express parallelism without being tied to low-level hardware details. By relying on compiler autovectorization, developers can automatically generate SIMD code and leverage vector units effectively. However, the flexibility of RVV also places significant demands on compiler technology. Building on this motivation, in this chapter, we provide a detailed analysis of different vectorization patterns across both RVV 1.0 and 0.7 implementations on real SIMD hardware, examining the autovectorization capabilities of LLVM, LLVM-EPI, and GCC compilers. We extend this investigation to real-world scientific applications, evaluating the autovectorization capabilities of the

same compilers on the AllWinner D1 and BananaPi-F3 boards to understand how these tools perform beyond microbenchmarks and in practical HPC workloads. Finally, we study the impact of key parameters introduced by the RISC-V Vector Extension, such as the distinction between Vector Length Agnostic (VLA) and Vector Length Specific (VLS) programming styles and the selection of LMUL, showing how these choices influence compiler decision and affect performance on modern RISC-V SIMD architectures.

7.1 FROM SIMT TO SIMD ABSTRACTIONS

Chapters 3-6 examined abstraction mechanisms for SIMT architectures, demonstrating how high-level programming models enable developers to achieve performance portability while relying on the runtime and compiler toolchain to efficiently map these abstractions onto GPU hardware for approximate and energy-efficient computing. This chapter extends that discussion to SIMD architectures, focusing on RISC-V processors implementing the RISC-V Vector Extension. Unlike SIMT systems, where parallelism is exposed through programming-model constructs, SIMD architectures have traditionally required developers to rely on architecture-specific intrinsics or assembly code to achieve high performance, thereby limiting portability and long-term maintainability. Modern compiler technology enables a different abstraction pathway: autovectorization. Instead of explicitly programming vector instructions, developers write scalar or structured data-parallel code, and the compiler automatically detects vectorizable patterns and generates the corresponding vector instructions. In this sense, autovectorization acts as a high-level abstraction layer for SIMD architectures. It allows programmers to express algorithms in a hardware-agnostic manner while delegating the vectorization to the compiler. The abstraction boundary moves from the programming model to the compilation process, where the compiler translates hardware-agnostic code into optimized SIMD instructions. The RISC-V Vector Extension further reinforces this abstraction-oriented perspective. Its vector-length-agnostic design decouples software from a fixed vector width, enabling the same binary or source code to scale across imple-

mentations with different hardware vector lengths. However, the effectiveness of this portability critically depends on compiler maturity and its ability to detect and transform vectorizable patterns. Although this chapter primarily evaluates execution time and vectorization coverage, improved vectorization has direct implications for energy-to-solution. By reducing dynamic instruction count and shortening execution time, effective autovectorization lowers the total energy consumed by an application. Even if instantaneous power increases when vector units are heavily utilized, the reduction in time-to-solution proportionally decreases overall energy consumption. Furthermore, improved vectorization often enhances data locality and reduces memory traffic, thereby decreasing energy consumption in the memory subsystem, which represents a significant fraction of total system energy in modern architectures. In this respect, compiler-driven SIMD abstraction complements the energy-efficient techniques explored in the previous chapters. While those chapters introduced explicit mechanisms such as frequency scaling and domain-specific energy modeling to adapt runtime behavior, autovectorization improves baseline efficiency by reducing computational overhead and time-to-solution without requiring explicit runtime intervention. Taken together, the SIMT abstractions analyzed in earlier chapters and the SIMD compiler automation studied here provide a unified view of abstraction-enabled performance portability and sustainability across heterogeneous systems. As RVV-based processors emerge as scalable and open alternatives for high-performance and energy-constrained computing environments, the effectiveness of compiler-driven vectorization becomes a key enabler for portable and sustainable SIMD execution.

7.2 RELATED WORKS ON RISC-V SIMD ARCHITECTURES

Vectorization plays a critical role in enhancing the performance of applications across various fields, including HPC, machine learning, and multimedia processing. Compiler support for autovectorization [3, 20, 49, 140–142] is essential for realizing these performance gains. With the increasing adoption of `VLA` ISAs such as RISC-V RVV, recent studies focus on optimizing compiler support to fully leverage RISC-V vectorization across various

hardware platforms [110, 138, 143]. Adit and Sampson [3] used the gem5 simulation to assess LLVM autovectorization performance for RISC-V RVV 1.0, examining both `VLA` and `VLS` configurations. Their study highlighted LLVM's limitations in handling dynamic vector lengths and shuffle patterns, comparing RVV's performance to fixed-length ISAs like AVX-512. Lin et al. [106] address the portability challenges in `VLA` programming by developing methods to adapt Arm SVE intrinsics to RVV. Ramírez et al. [148] extended gem5 to support RVV instruction, allowing customizable configurations for vector registers, memory ports, and lanes. They developed a benchmark suite of diverse HPC and embedded applications. Their findings show performance gains with vectorization, though compiler limitations impact complex data structures, highlighting the need for further compiler optimizations for RISC-V vector architectures. Lai et al. [96] presents enhancements to LLVM's autovectorization capabilities for linear recurrence programs on the RISC-V Vector RVV, focusing on scan operations to address loop-carried dependencies. Lee et al. [101] evaluated autovectorization support of T-Head GCC-8.4 compiler compatible with RVV 0.7.1 on Allwinner D1 hardware. To facilitate a comparison between the T-Head GCC-8.4 and the LLVM compiler—which lacks support for RVV 0.7—they modified the assembly code generated by LLVM (originally compatible with RVV 1.0) to be compatible with RVV 0.7[100]. Their study demonstrated not only vectorization speedups for HPC kernels, but also revealed limitations due to incomplete support in both tooling and hardware, including the lack of 64-bit element support and sensitivity to loop structures. Their findings emphasize the early development stage of the RVV ecosystem and the need for improved compatibility with present standards like RVV v1.0. Lin et al. [104] assess the performance of RVV on computer vision algorithms using intrinsic functions on the Xuantie C906 RISC-V. Their evaluation specifically examines the impact of using different integer `LMUL` settings showing 2.98x performance speedup against the OpenCV implementation. Shih et al. [173] developed a predictor within LLVM that automatically selects the optimal `LMUL` value to minimize register pressure.

In this chapter we provide a comprehensive analysis of compiler autovectorization capabilities for RISC-V RVV 0.7 and 1.0 across

different compiler versions and real hardware, showing the performance impact of `VLA` and `VLS` on the `LMUL` parameter tuning.

7.3 BENCHMARK METHODOLOGY

To thoroughly assess the autovectorization capabilities of compilers, we adopted a dual approach that includes both synthetic benchmarks (TSVC) and real-world applications, ensuring a comprehensive evaluation across a wide range of vectorization patterns and practical workloads. In the following, we describe these approaches in detail.

Category	**#Loops**	**Set 1**	**Set 2**	**Set 3**	**Set 4**	**Set 5**
Control Flow	22	10	8	4	15	10
Control Loops	13	12	0	1	11	12
Crossing Thresholds	8	2	1	5	2	3
Equivalencing	5	3	1	1	5	5
Expansion	12	8	0	4	6	7
Global Data Flow	10	7	0	3	7	8
Indirect Addressing	7	5	0	2	4	7
Induction Variables	9	2	3	4	6	6
Linear Dependence	14	9	4	1	8	9
Loop Rerolling	4	0	4	0	3	2
Loop Restructuring	9	3	3	3	4	1
Node Splitting	6	2	0	4	1	2
Packing	3	1	0	2	0	0
Recurrences	3	1	0	2	0	0
Reductions	15	7	2	6	2	4
Searching	2	0	1	1	1	0
Statement Reordering	3	0	0	3	0	0
Symbolics	6	4	0	2	5	5
Sum	**151**	**76**	**27**	**48**	**80**	**81**

Table 7.1: Number of loops in Set 1: (`GCC-14<LLVM-19`), Set 2: (`GCC-14>LLVM-19`), Set 3: (`GCC-14=LLVM-19`), Set 4 and 5: vectorized loops by GCC-14 and LLVM-19 compilers respectively.

Test Suite for Vectorizing Compilers (TSVC)

The Test Suite for Vectorizing Compilers (TSVC) builds on earlier work that used 100 Fortran loops to evaluate the effectiveness of autovectorizing compilers [22]. Additionally, TSVC2 includes 151 loops implemented in C/C++ categorized into 18 distinct groups[175] (Table 7.1).

Real Applications

In addition to the synthetic benchmarks categorized in the TSVC loops, we evaluated seven real-world applications across diverse domains, as summarized in Table 7.2. These benchmarks were selected for their relevance in different computational fields and their ability to highlight vectorization patterns. Each application belongs to a domain and demonstrates specific computational models and data-level parallelism (DLP) patterns, providing a comprehensive overview of vectorization challenges and opportunities on RISC-V systems.

In addition, these applications cover a range of algorithmic models, such as dynamic programming, structured grids, and linear algebra, offering a diverse set of workloads for performance evaluation. The selected benchmarks also vary in their DLP patterns, including regular, irregular, and mixed patterns most of which are derived from RISCV-Benchmark-Suite [148]. Table 7.2 provides an overview of the selected applications, highlighting their domains, algorithmic models, DLP patterns, and size of benchmark. These characteristics illustrate the diversity of workloads used to evaluate vectorization potential on RISC-V RVV systems.

7.4 EXPERIMENTAL EVALUATION

This section presents a performance analysis of autovectorization on RVV RISC-V boards for the GCC and LLVM compilers.

Table 7.2: Overview of applications and their characteristics

Application	Application Domain	Algorithmical Model	DLP Pattern	Size of Benchmark
Blackscholes	Financial Analysis	Dense Linear Algebra	Regular	16K entries
Jacobi-2D	Engineering	Dense Linear Algebra	Regular	256 x 256 grid, 100 iterations
Needleman-Wunsch	Bioinformatics/Genomics	Dynamic Programming	Irregular	Sequence length 2^{20}
Particlefilter	Medical Imaging	Structured Grids	Mix	128 x 128 x 16 grid, 4096 particles
Pathfinder	Grid Traversal	Dynamic Programming	Regular	2048 width, 256 iterations
Streamcluster	Data Mining	Dense Linear Algebra	Mix	8192 points, 128 dimensions
Axpy	Vector Operations	Dense Linear Algebra	Regular	Vector size 2^{20}

7.4.1 *Experimental setup*

7.4.1.1 *Hardware*

In this study, we evaluated two RISC-V boards with support for the vector extension. The first board, the Allwinner D1, features a single-core XuanTie C906 processor running at 1.0 GHz, with 32 KB of instruction cache and 32 KB of data cache, and 1GB of DDR3 memory. This board implements the RV64GCV ISA with RVV 0.7 support and provides a 128-bit vector width. The second board, the BananaPi-F3, incorporates a more powerful SpacemiT K1 processor and supports the RV64GCVB ISA with RVV 1.0. The processor is organized into two clusters each with 4-core unit delivering 2.0 TOPS for AI workloads with 32 KB L1 per core, 512 KB L2, and 512 KB TCM. The board has 4 GB of LPDDR4 memory and supports a vector width of 256/128 bits across the two clusters. These setups enable the analysis of autovectorization on two RISC-V boards with different RVV versions.

7.4.1.2 *Compilers*

Table 7.3 provides a comprehensive overview of the compilers and compiler flags utilized in our experiments. The RVV 0.7 specification is not supported by the mainstream GCC and LLVM compilers. However, on the *Allwinner D1* board, vector instructions compatible with RVV 0.7 can be generated using the XuanTie GCC [202] or the LLVM-EPI compilers [51]. XuanTie GCC, a customized fork of the GNU compiler created by T-Head,

Table 7.3: Compiler Flags for Vectorization Enabled, Categorized by RVV Version

RVV Version	Compiler	Vectorization Enabled
RVV 0.7	**GCC-8.4 / GCC-10.2**	`-O3 -ftree-vectorize -march=rv64gcv0p7`
	LLVM-EPI	`-O3 -fvectorize -mepi -menable-experimental-extensions -march=rv64gcv0p7`
RVV 1.0	**GCC-13 / GCC-14**	`-O3 -ftree-vectorize -march=rv64gcv1p0`
	LLVM-16 / LLVM-19	`-O3 -fvectorize -march=rv64gcv1p0`
	LLVM-EPI	`-O3 -fvectorize -mepi -menable-experimental-extensions -march=rv64gcv1p0`
	GCC-14-VLS	`-O3 -ftree-vectorize -mrvv-vector-bits=zvl -march=rv64gcv_zvl256b`
	LLVM-19-VLS	`-O3 -fvectorize -mrvv-vector-bits=zvl -march=rv64gcv_zvl256b`

is specifically designed for T-Head processors and supports the RVV 0.7 specification. The LLVM-EPI compiler also enables the generation of vector instructions for RVV 0.7. However, the compiler is built by default to work with `SEW` up to 64, which is not compatible with the *Allwinner D1* hardware. For experiments related to the updated RVV 1.0 standard, we employed the newer versions of GCC (14.2), LLVM (19.1.1) and LLVM-EPI (v. 2024-09-28), which fully support RVV 1.0.

7.4.1.3 *Benchmarks setup*

In order to evaluate the autovectorization capabilities of modern compilers, we followed a comprehensive analysis based on TSVC loops categories as we mentioned in Section 7.3. To aggregate the results of each loop into categories, we followed the same formula specified by Siso et al. [175]. We compiled TSVC loops

using different compilers with and without autovectorization (Table 7.3). The scalar code is compiled only using `-O3` and `-fno-vectorize` and `-fno-tree-vectorize` for GCC and LLVM respectively. For each loop t in TSVC we computed the *median time* of 5 runs for both the auto-vectorized and non-vectorized code. Then we calculated the speedup of the auto-vectorized version ($\overline{E}_t^{\text{vec}}$) using as a baseline the non-vectorized code (scalar) $\overline{E}_t^{\text{scalar}}$. For computing the speedup of codes autovectorized by a specific compiler, we considered the scalar version generated by the same compiler. The speedup of the loop t is defined as $\eta = \frac{\overline{E}_t^{\text{scalar}}}{\overline{E}_t^{\text{vec}}}$. As we mentioned all 151 loops were divided into 18 distinct categories. Therefore, for each category c, we aggregated the speedup of loops using the *geometric mean* $(\prod_{i=1}^{n} \eta_i)^{\frac{1}{n}}$, where n is the number of loops in each category c (Figure. 7.1 and 7.3). Finally, we computed the overall *geometric mean* for each compiler in the 18 categories to have a comprehensive comparison of the autovectorization support of each compiler (Figure. 7.7). Likewise, for real-word application we followed the same approach by computing the speedup of the autovectorized code using as baseline the scalar one. The input size of each benchmark is defined in Table 7.2.

The following sections explore the compiler's autovectorization capabilities for RISC-V RVV 1.0 and 0.7, assessed using TSVC and real-world benchmarks.

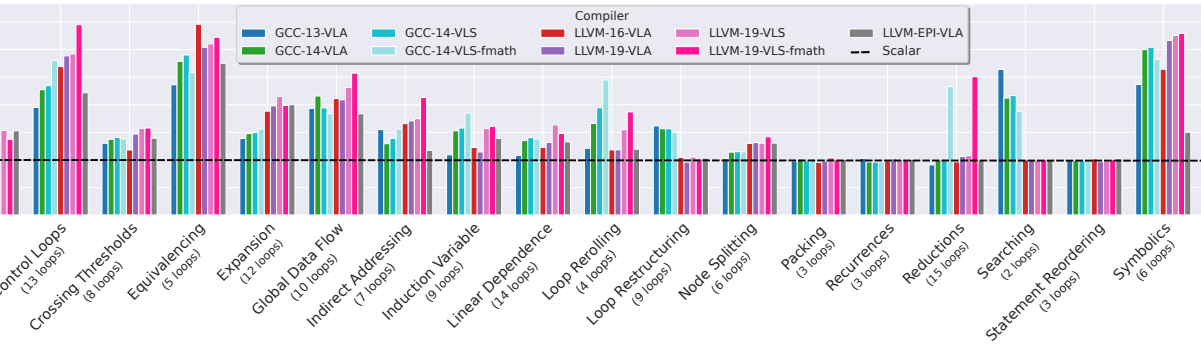


Figure 7.1: Geometric mean of speedup achieved through autovectorization across different loop categories and compilers using RVV 1.0.

7.4.2 *Measuring Vectorization Performance of TSVC Loops*

7.4.2.1 *TSVC on RISC-V RVV 1.0*

Figure 7.1 illustrates the autovectorization capabilities of the compilers listed in Table 7.3 for each category of the TSVC benchmark for RVV 1.0. With regard to Figure 7.1, the principal findings can be summarized as follows.

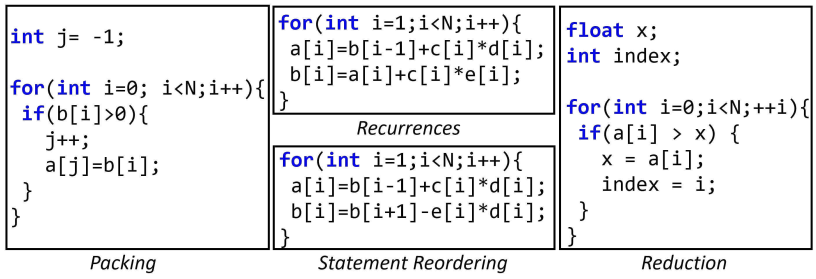


Figure 7.2: TSVC patterns that are not vectorized by any compiler

Code not vectorized by any compiler: For loop patterns such as `Packing`, `Recurrences`, `Reductions`, and `Statement Reordering` (as shown in Figure 7.2), neither GCC nor LLVM could effectively vectorize the code on RISC-V.

In the `Packing` pattern, the compiler is unable to vectorize the code because the conditional statement introduces uncertainty about the control flow and the data access pattern associated with the variable `j`.

For the `Recurrences` pattern, vectorization is hindered by loop-carried dependencies: the value of `a[i]` depends on `b[i-1]` from the previous iteration, and `b[i]` relies on the value of `a[i]` calculated within the same iteration. These dependencies enforce sequential execution, making vectorization impossible.

The `Statement Reordering` pattern introduces additional write-after-read dependencies into the `Recurrences` pattern, complicating the compiler's ability to reorder computations to eliminate read-after-write and write-after-read dependencies, further obstructing vectorization.

Lastly, in the `Reduction` pattern without using `-fast-math` option most of the loops are not vectorized due to non associativity of floating point operations. Out of 15 reduction patterns, only

two are vectorized by GCC 14, and four by LLVM 19 (Table 7.1). The vectorized code might generate instructions where elements are aggregated in a different order than in the scalar version. This reordering can yield different numerical results, discouraging the compiler from vectorizing the code to maintain accuracy. Enabling `fast-math` optimizations, both GCC and LLVM compilers are able to vectorize loops in the Reduction category, leading to a speedup of up to 3x. The fast-math optimizations also enhances other categories, particularly those involving reduction operations, by increasing the overall speedups.

To further explore this limitation, we also evaluated these patterns on x86 architecture and observed the same behavior. This implies that these loop categories inherently represent a challenge for autovectorization, posing optimization difficulties across both x86 and RISC-V architectures.

Codes vectorized only by GCC: GCC demonstrates a notable speedup in patterns such as `Loop Restructuring` and `Searching`, indicating its advantage in handling these types of loops compared to LLVM.

In `Loop Restructuring` 5 out of 9 loops are not vectorized by the GCC and LLVM 16 and 19 compilers (Table 7.1). GCC is able to vectorize the other 4 loops, while LLVM 16 and 19 vectorize only one loop. LLVM-EPI fails to vectorize any loops.

```
for(int i=1;i<N;i++)
{
 a[i]+=b[i]*c[i];
 e[i]=e[i-1]*e[i-1];
 a[i]-=b[i]*c[i];
}
```

Listing 7.1: TSVC loop s222 (Loop Restructuring)

```
for(int i=0;i<N;++i){
 for(int j=1;j<N;j++){
  a[j][i]=
    a[j-1][i]+b[j][i];
 }
}
```

Listing 7.2: TSVC loop s231 (Loop Restructuring)

Listings 7.1 and 7.2 show the TSVC `s222` and `s231` loops that are not vectorized by LLVM. For `s222`, the LLVM compiler is unable to vectorize the code due to loop-carried dependencies in the middle part of the loop (line 4), while the GCC compiler separates the loop into two, enabling automatic vectorization.

In loop `s231` LLVM is not able to apply loop interchange due to the data dependencies between iterations (line 4) missing the opportunity to vectorize the code.

Listing 7.3 shows the loop `s331` from `Searching` pattern. LLVM is not able to vectorize the code while GCC achieves 4.5x speedup. The GCC compiler uses mask operation to vectorize the code, while LLVM is unable to deduce that `j` retains only the value for the last negative element. To help LLVM vectorize, all the indexes for which `a[i]<0` can be stored in a vector as defined in the commented code and then use the last element of the `indices` vector.

```
// std::vector<int> indices(N);
for(int i = 0; i < N; i++){
   if(a[i] < 0){
    // indices.push_back(i);
    j = i;
   }
  }
```

Listing 7.3: TSVC loop s331 (Searching)

Code vectorized only by LLVM: In the `Expansion` category, LLVM demonstrates a clear performance advantage over GCC, particularly in loop `s255` (Listing 7.4).

```
 for(int i = 0; i < N; i++){
  a[i] = (b[i] + x + y) * 0.333;
  y = x;
  x= b [i];
 }
```

Listing 7.4: TSVC loop s255 (Expansion)

In this loop, variables x and y introduce interdependency between consecutive iterations, meaning that x and y cannot be computed in parallel throughout the loop. Due to this loop-carried data dependence, GCC is unable to vectorize `s255`, whereas LLVM overcomes this limitation, resulting in a 7x speedup. Compared to GCC, LLVM identifies the dependencies introduced by

x and y and separates the scalar portion of the code from the vectorizable part, allowing automatic vectorization. Furthermore, LLVM demonstrates better performance in the Expansion category, especially in loops s252 and s253, achieving speedups ranging from 4x to 6x. In contrast, GCC only manages speedups of 1.5x to 2.5x due to a different selection of the LMUL parameter. While LLVM-EPI shows a slowdown compared to LLVM 16 and 19 for the Expansion category, on loops s256 and s257, it performs more efficiently by avoiding unnecessary vectorization.

VLA vs VLS: For both compilers, specifying the vector length at compile time generates a more optimized code for VLS, compared to VLA. In VLA since the vector length is variable at runtime, the compiler has to generate more generic code to handle different possible lengths. This introduces overhead, as the compiler cannot optimize for a fixed length and may introduce run-time checks or masking to handle different vector lengths efficiently. Furthermore, knowing the vector length can affect the LMUL parameter tuning leading to drastic differences in performance (Section 7.4.3).

7.4.2.2 *TSVC on RISC-V RVV 0.7*

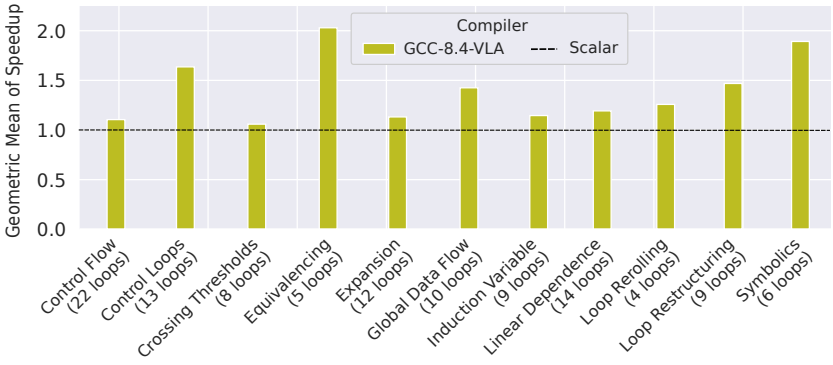


Figure 7.3: Geometric mean of speedup achieved through auto-vectorization across different loop categories and compilers using RISC-V RVV 0.7

Figure 7.3 shows the TSVC categories that achieve some speedup on RISC-V RVV 0.7. We removed the GCC 10.2 compiler since it is unable to vectorize any of the loops. Additionally, the missing categories are not vectorized by GCC 8.4 either. The support

for autovectorization for the RVV 0.7 extension is significantly poorer compared to the newer version 1.0.

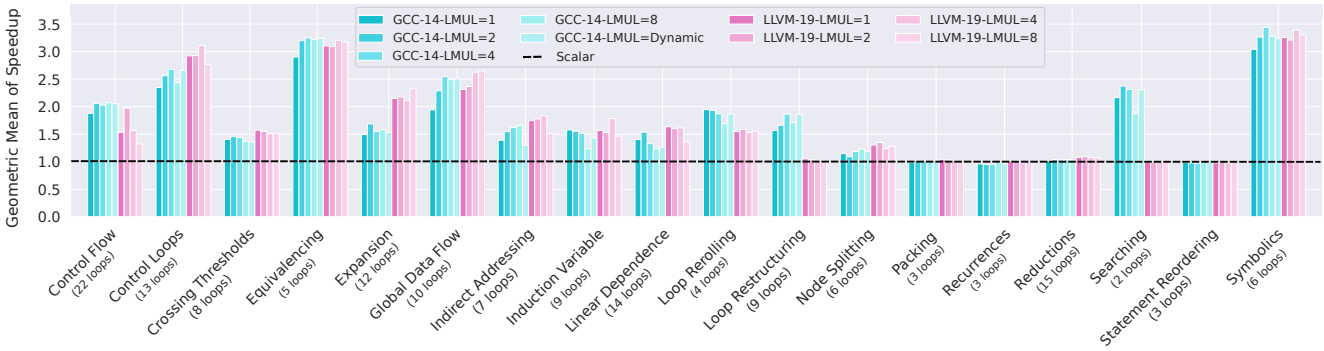


Figure 7.4: Geometric mean of speedup achieved through autovectorization across TSVC loop categories, comparing performance for GCC-14 and LLVM-19 with various LMUL configurations with RVV 1.0

7.4.3 *Improving performance with LMUL*

To further enhance flexibility and performance, RVV defines the `LMUL` parameter which specifies the size of a vector register group, allowing single vector instructions to operate on operands that span multiple registers.

RVV 0.7 supports integer `LMUL` values of 1, 2, 4, or 8, forming vector register groups consisting of 1, 2, 4, or 8 registers, respectively. Additionally, RVV 1.0 introduces support for fractional `LMUL` values: 1/2, 1/4, and 1/8. These fractional values reduce the effective size of the vector register, allowing more vector registers to be used simultaneously, which can be beneficial in scenarios where register pressure is high, and smaller data sets are being processed.

Figure 7.4 shows the *geometric mean* of the speedup for each TSVC category as we increase the `LMUL` parameter from 1 to 8 for GCC-14 and LLVM-19. For both compilers, we can set the suggested `LMUL` as option using `-mrvv-max-lmul` and `-riscv-v-register-bit-width-lmul` The GCC compiler benefits from higher `LMUL` values in 10 out of 18 TSVC categories while in LLVM only 5 categories show improvement. With higher `LMUL` values,

each instruction can process a larger subset of data, reducing the number of instructions needed to handle a full data set. This leads to a more compact code for data-parallel sections. Furthermore, during loop strip mining, a higher `LMUL` reduces the number of iterations required, as more elements can be processed in each iteration, resulting in performance improvements. For `Induction Variables`, `Loop Rerolling` and `Node Splitting`, GCC does not benefit from increased `LMUL` due to high register pressure introduced by the use of `LMUL`. In fact, by increasing `LMUL` from 1 to 8 we also decrease the number of available registers from 32 to 4. Therefore, having fewer registers available can result in memory spilling as it becomes challenging to keep all variables in registers simultaneously. The same occurs in LLVM for the `Indirect Addressing` and `Linear Dependencies` categories. Other categories remain unaffected by `LMUL`, as they are not vectorized by any compiler.

Since fractional `LMUL` cannot be specified as a compiler option, we developed a small benchmark using vector intrinsics to demonstrate the effectiveness of fractional `LMUL`. The benchmark consists of a loop that runs a vector addition on 8-bit integers followed by add, sub, and multiply operations on `int64` array. Without fractional `LMUL` the compiler, in order to work on the same number of elements, needs to use `LMUL=1` for the first operation and `LMUL=8` for the others. However, using `LMUL=8`, only 4 registers are available. This compromises the performance due to high register pressure introduced by 64-bit operations.

Figure 7.5 shows the speedup of three versions of the benchmark: the `no-fraction LMUL` version sets the `LMUL` to 1 and 8; the `fractional LMUL` sets the `LMUL` parameter to 1/8 and 1, the `autovec` version, which automatically select the `LMUL` size. `Fractional LMUL` achieves a 2.39x speedup compared to the 2.14x speedup of the code that cannot take advantage of fractional `LMUL`. Furthermore, leveraging autovectorization, the compiler uses an `LMUL` of 1/4 and 2, resulting in even higher performance compared to the code implemented with intrinsics.

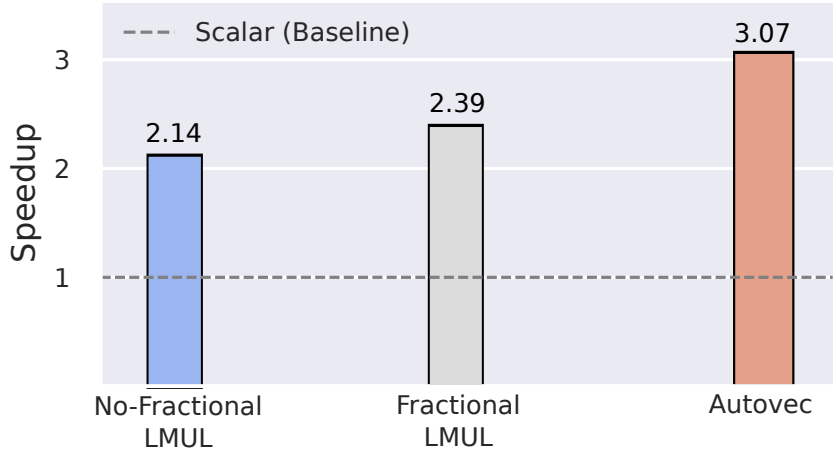


Figure 7.5: Speedup of code with and without fractional LMUL

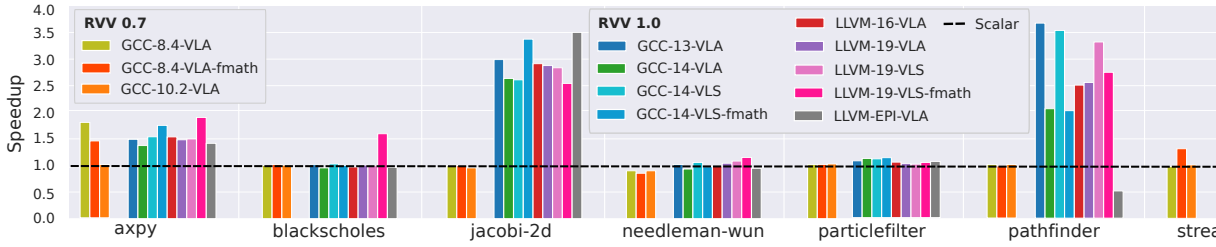


Figure 7.6: Speedup achieved through autovectorization across real applications and compilers with using RVV 0.7 and 1.0

7.4.4 *Measuring Vectorization Performance of Real Applications*

To complement the evaluation, we also measured the performance of real-world applications with autovectorization enabled, as defined in Table 7.3.

Figure 7.6 shows the speedup achieved through autovectorization across six real applications and all compilers using RVV 0.7 and 1.0.

With RVV 0.7 only the simplest code `axpy`, achieves a 1.7x speedup due to vectorization, while for all other applications vectorization is not applied. The improvement achieved in `streamcluster` is not related to vectorization but only to the use of `-ffast-math`. These results validate the limited support for autovectorization found in RVV 0.7.

Looking at the RVV 1.0 results, for `needleman-wunsch` and `particelfilter` the compilers cannot autovectorize critical sec-

tions due to memory dependencies that are considered unsafe for vectorization, resulting in minimal speedup over scalar code. Although `blackscholes` is embarrassingly parallel, compilers are not able to vectorize the code due to difficulties in vectorizing math functions. LLVM-EPI achieves a speedup of 3.5x in `jacobi-2d` compared to the speedup of 3x in the other compilers, while for `pathfinder` experiences a slowdown, reducing performance to half of the scalar version. These results are justified by the different settings of the `LMUL` parameter between compilers that can drastically impact performance. In `jacobi-2d` using `LMUL=1`, LLVM-EPI is able to achieve higher performance, while in `pathfinder` selecting an `LMUL=1/2` the LLVM-EPI compiler results in a slowdown. With `VLS` configuration `pathfinder` achieves up to 3.5x speedup for both LLVM 19 and GCC 14 compared to 2.5x speedup of the `VLA` configuration. Specifying the vector length allows the compiler to select a better `LMUL` value, improving overall performance.

With `-ffast-math` enabled and the `VLS` configuration, LLVM 19 allows the vectorization of `blackscholes` application resulting in speedup up to 1.6x. The compiler unrolls math function to process them in scalar mode and then switches back to vector computation.

7.4.5 *Comparative Discussion Across Compilers*

Figure 7.7 shows the aggregated *geometric mean* of speedup across all TSVC categories for each compiler. The compiler support for autovectorization is notably limited in RVV 0.7 compared to RVV 1.0. With GCC 10.2 code is not vectorized while with GCC 8.4, vectorization is applied only on simple patterns. In contrast, in RVV 1.0, the support for vectorization has significantly improved showing an overall speedup of up to 1.60x. Although the number of loops vectorized by GCC-14-VLS and LLVM-19-VLS are the same, the LLVM-19-VLS achieves better performance on 76 loops over 151 (Table 7.1). These results are mostly related to a better selection of `LMUL` parameter. LLVM 19 is more influenced by vector length knowledge than GCC 14. In fact, LLVM 19 shows a consistent improvement with `VLS` across all categories. It achieves a 1.66x speedup compared to the 1.51x of `VLA`. Meanwhile, GCC

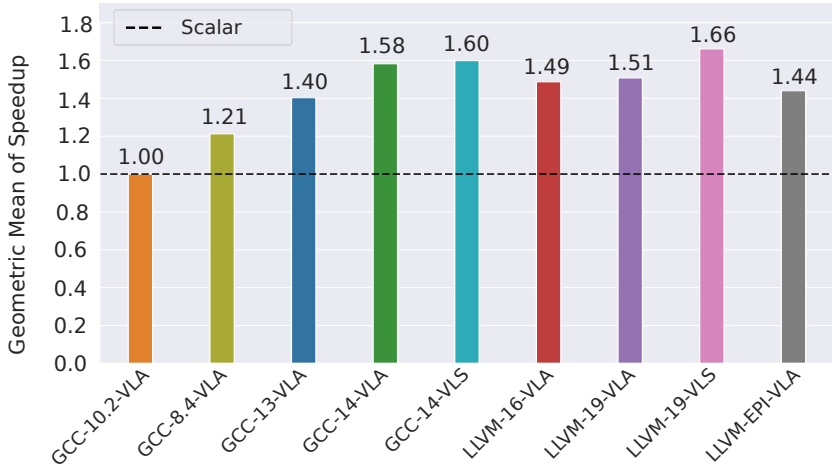


Figure 7.7: Geometric mean of speedup across all categories TSVC loops and compilers

14 shows similar performance between `VLS` and `VLA`. GCC 14 shows a noticeable performance improvement over GCC 13. This suggests that the GCC 14 version has undergone optimizations that enhance performance. LLVM-EPI performance aligns closely with that of LLVM-19 in `VLA` mode.

In summary, LLVM 19 with `VLS` configuration achieves the highest performance among all compilers.

7.5 SUMMARY AND DISCUSSION

In this chapter, we evaluated the automatic vectorization capabilities of modern RISC-V compiler toolchains, comparing RVV 0.7 and RVV 1.0. Our analysis shows that RVV 1.0 compilers generate vectorized instructions more effectively than RVV 0.7, improving the coverage of diverse loop patterns without requiring manual intervention. We also highlighted the impact of parameters specific to RISC-V, such as `LMUL`, demonstrating that selecting appropriate values can reduce register pressure and increase performance. Overall, this study demonstrates the effectiveness of autovectorization in RISC-V compilers, showing how modern compilers bridge the gap between high-level abstractions and efficient low-level SIMD execution.

8

CONCLUSION, AND FUTURE WORK

8.1 SUMMARY AND CONCLUSION

In this thesis, we moved through three complementary directions, starting from the analysis of low- and high-level programming models and progressing toward the design of high-level and domain-specific abstractions for approximate and energy-efficient computing. In Chapter 1 of this thesis, we provided the introductory information, explained the reasons and motivations behind our work, and raised four main research questions. In Chapter 2, we presented the background information related to our study. Chapter 3 investigated how high-level abstractions in SYCL 2020 perform on different SIMT architectures and across different implementations of SYCL. To this end, we developed a feature-oriented benchmarking suite designed to test high-level abstractions defined in SYCL 2020. This suite extended SYCL-Bench with nine new templated benchmarks, six code patterns, and 44 configurations. The benchmarks targeted key SYCL 2020 features such as reduction kernels, group algorithms and atomic operations. We executed these benchmarks on NVIDIA, AMD, and Intel GPUs using AdaptiveCpp and oneAPI DPC++, providing a cross-architecture and cross-compiler evaluation of SYCL's performance characteristics. Through this analysis, we quantified how effectively SYCL's high-level abstractions are mapped to low-level instructions, identifying both their strengths and the current limitations of existing compiler and backend implementations. In particular, our results illustrate the inherent difficulties compiler toolchains face in generating efficient code across heterogeneous architectures, where differences in execution models, subgroup organization, memory hierarchies, and atomic semantics require backend-specific optimizations that are not yet uniformly implemented. Chapter 4 focused on the design and evaluation of domain-specific and high-level abstractions for approximate computing on SIMT architectures. In this chapter, we addressed the

challenges posed by multi-objective optimization in approximate computing, where accuracy, performance, and energy efficiency often conflict, leading to a complex exploration of the approximation space. We highlighted the limitations of existing frameworks, which are typically tied to specific compilers or architectures, and emphasized the need for portable and expressive abstractions that enable developers to apply multiple approximation techniques across diverse hardware platforms. To tackle these challenges, we introduced SYprox, a SYCL-based abstraction layer for approximate computing. SYprox enabled programmers to implement heterogeneous approximate applications using state-of-the-art techniques such as perforation, mixed precision, and signal reconstruction. In SYprox, we designed a novel host perforation technique to fully exploit the CPU–GPU execution model, and for the first time, we provided a mechanism to combine multiple approximation techniques across heterogeneous devices. We validated the effectiveness and portability of SYprox through an extensive experimental evaluation on eight benchmarks and 100 datasets, targeting GPUs from AMD, Intel, and NVIDIA. The evaluation demonstrated that the approximation techniques implemented in SYprox generate better Pareto-Frontiers compared to standalone methods and state-of-the-art frameworks such as HPAC [53] and prior manual approaches developed by Maier et al. [114]. Furthermore, the study confirmed that high-level abstractions can significantly simplify the adoption of approximate computing in heterogeneous systems while maintaining portability and enabling efficient exploration of complex approximation spaces.

Chapter 5 addressed the increasing demand for energy-efficient computing in large-scale heterogeneous systems. In this work, we contributed to the advancement of portable abstractions for energy profiling and frequency scaling, as well as to the development of a novel methodology for applying frequency control efficiently at scale. We introduced SYnergy, a portable SYCL-based interface that enables energy profiling and frequency scaling across heterogeneous hardware platforms. By abstracting vendor-specific APIs, SYnergy provides per-kernel energy control through a unified interface, allowing developers to optimize energy behavior without interacting directly with vendor-specific

libraries such as RAPL, NVML, ROCm SMI, and Level Zero. Through experimental analysis, we demonstrated the limitations of existing frequency scaling methodologies. Coarse-grained approaches, where frequency is changed one time at the start of the application, fail to capture the distinct energy behavior of individual kernels, leaving significant optimization opportunities unexplored. Conversely, fine-grained per-kernel scaling can be hindered by the overhead introduced by frequent frequency changes. To address these challenges, we proposed a phase-based frequency scaling methodology that operates at an intermediate level of granularity. Using a DAG representation of the computation, we defined an algorithm that identifies sequences of consecutive tasks (i.e. a phases) optimized with similar optimal frequencies, reducing transition overhead while enabling scalable energy optimization. We further extended this methodology by embedding MPI communication metadata into the DAG, allowing frequency change latency to be hidden through communication and frequency change overlap. This enables scalable energy savings even across multi-GPU and multi-node configurations.

We evaluated SYnergy and the phase-based methodology on hardware from AMD, Intel, and NVIDIA, and at scale on cluster configurations with up to 64 GPUs. Our results showed that SYnergy enables portable energy optimization across heterogeneous GPU vendors, validating its practicality as an energy-aware programming model. Moreover, phase-based scaling significantly outperforms both coarse-grained and fine-grained approaches, enabling energy reductions while minimizing performance loss in large-scale systems.

Chapter 6 tackled the complementary challenge of predicting optimal GPU frequencies for energy-efficient execution, extending the DVFS-based optimization capabilities introduced in Chapter 5 with general-purpose and domain-specific energy models. We introduced a general-purpose energy model [48] capable of predicting an optimal frequency for a given kernel using static compile-time features. Based on these static characteristics, the machine learning model can estimate the frequency that best satisfies a selected optimization target, whether minimizing energy consumption, maximizing performance, or optimizing energy–delay product, without requiring any application-specific

profiling or user annotations. However, our analysis demonstrated that the energy behavior is not fully captured by static kernel features alone [25]. In many scientific and industrial applications, the energy characterization of a kernel depends on domain-specific characteristics. These properties can significantly affect the compute/memory balance, leading to inaccuracies in predictions made by domain-agnostic models. To overcome this limitation, we developed domain-specific energy models that extend the general-purpose model with input-dependent and application-level features. By tailoring the feature space to the application domain, we significantly improved predictive accuracy and frequency selection quality. We validated this approach using two representative GPU-accelerated scientific applications: LiGen, a molecular-screening platform for drug discovery, and Cronos, a magnetohydrodynamics simulation used in astrophysics. Across both domains, the domain-specific models consistently outperformed the general-purpose model, demonstrating an improved selection of energy-optimal frequency configurations.

Chapter 7 explored the highest level of the abstraction stack of programming models by analyzing the autovectorization capabilities of modern compilers in generating efficient vector instructions. In particular, we focused on the emerging RISC-V Instruction Set Architecture and its RISC-V Vector Extension (RVV), which introduces the SIMD execution model. In this chapter, we evaluated the compiler autovectorization step for RVV, examining GCC, LLVM, and LLVM-EPI, and assessing their ability to automatically map scalar code into optimized vector instructions. We compared RVV 0.7 and RVV 1.0 implementations on synthetic loop pattern from the TSVC benchmark [118] and real hardware and investigated how compiler maturity, internal heuristics, and target-specific backend support influence the quality of autovectorized code. Our results demonstrated that autovectorization support for RVV 0.7 is significantly inferior to RVV 1.0, and that even RVV 1.0 compilers still fail to vectorize certain patterns efficiently due to suboptimal handling of mask operations and math functions. Furthermore, we analyzed the impact of key RVV programming styles, vector-length-agnostic (VLA) vs. vector-length-specific (VLS), as well as the role of LMUL (vector register grouping), illustrating how these choices affect compiler

decisions and ultimately determine SIMD performance on modern RISC-V systems.

Overall, the contributions of this thesis lie in the development of methodologies and tools that address key challenges across the programming model abstraction stack, from low-level performance and portability analysis to high-level domain-specific abstractions for approximate and energy-efficient computing, and compiler autovectorization. These contributions provide practical solutions that improve performance, portability, and energy efficiency on heterogeneous and large-scale computing systems.

8.2 ANSWERS TO RESEARCH QUESTIONS

In this section, we provide answers to the research questions that were introduced at the beginning of this thesis in Section 1.5.

Research Question 1:

Research Question 1
How do high-level abstractions in SYCL perform against native low-level APIs on SIMT architectures?

Answer: We implemented SYCL-bench a benchmark suite specifically designed for measuring the performance of high-level abstractions in SYCL 2020 on different SIMT architectures. We conducted an in-depth analysis of different SYCL implementations and compiler backends, evaluating reduction and atomic operations on SIMT architectures from multiple vendors, including NVIDIA, AMD, and Intel data-center GPUs. This analysis provided insights into how SYCL maps high-level abstractions to low-level APIs, the varying levels of compiler maturity across platforms, and how high-level programming abstractions do not always translate efficiently into optimized code on every architecture. Our results show that the performance of high-level SYCL abstractions depends on compiler maturity, optimization strategies such as thread coarsening, and the mapping of SYCL to low-level intrinsics. For example, DPC++ kernel reductions achieve up to 8× speedup compared to manually implemented reductions and AdaptiveCPP reductions on Intel GPUs, thanks to

optimized local memory usage and automatic thread coarsening. Differently, atomic operations on AMD hardware are mapped to a naive CAS loop for both SYCL implementations, resulting in up to 6000× slower performance compared to using hardware-supported atomic instructions. As a consequence, high-level abstractions do not uniformly compile into optimal low-level code, and achieving portable performance still requires understanding implementation behaviors and occasionally applying explicit user-level optimizations.

Research Question 2:

Research Question 2
How can we design domain-specific and high-level abstractions for approximate computing?

Answer: We developed SYprox a new approximate computing interface based on SYCL that allows programmers to easily implement heterogeneous approximated applications with state-of-the-art approximation techniques such as perforation, mixed precision, and signal reconstruction. We expanded the set of approximation techniques by introducing a new perforation approach called *host perforation*, which applies perforation on the host data and only sends/receives perforated data to/from the device. Furthermore, we defined a way to combine data perforation and reconstruction techniques (on the host, device, or both) with mixed precision to support an unprecedented number of approximations. Our results demonstrate that approximate programs implemented using SYprox techniques, individually or in combination, outperform state-of-the-art frameworks such as HPAC [53] and the approach by Maier et al. [114]. Host perforation achieves up to 2× speedup in data transfer relative to the accurate execution, compared to Maier et al.'s device perforation, while kernel speedups range from 1.7× to 4.2×, outperforming device perforation (0.5–3.5×). By combining host/device perforation with mixed precision and input/output reconstruction, SYprox configurations consistently dominate HPAC's Pareto-frontier, achieving up to 4× speedup with <20% error, compared to HPAC's 2× speedup at 80% error, all measured relative to

accurate execution. Furthermore, we showed that SYprox enables portable approximate computing across different SIMT architectures, providing consistent performance and accuracy on GPUs from AMD, Intel, and NVIDIA for eight benchmarks.

Research Question 3:

Research Question 3
How can we design domain-specific and high-level abstraction for energy-efficient computing?

Answer: To address the challenge of designing domain-specific, high-level abstractions for energy-efficient computing, we focused on three subquestions.

3.1. How can we design a portable, high-level interface that integrates energy profiling and frequency-scaling capabilities across heterogeneous architectures?
We developed SYnergy, a SYCL-based library that provided a portable, high-level abstraction for energy profiling and frequency scaling across heterogeneous hardware. SYnergy abstract vendor-specific energy management APIs, enabling developers to measure per-kernel and per-app energy consumption and adjust device frequency at runtime without handling low-level hardware details. By supporting CPUs, GPUs, and other accelerators through a unified interface, SYnergy enabled energy-efficient computing across diverse platforms.

3.2. How can we improve existing frequency scaling approaches?
We demonstrated the limitations of existing frequency scaling methodologies. Coarse-grained approaches, where frequency was changed once at the start of the application, failed to capture the distinct energy behavior of individual kernels, leaving significant optimization opportunities unexplored. Fine-grained per-kernel scaling, on the other hand, was often hindered by the overhead of frequent frequency changes. To address these challenges, we proposed a phase-based frequency scaling methodology operating at an intermediate level of granularity. Using a DAG representation of the computation, we developed an algorithm to identify energy phases and apply frequency changes strategically, reducing

transition overhead while enabling scalable energy optimization. We further extended this methodology to distributed heterogeneous systems by integrating MPI communication metadata into the DAG, allowing frequency change latency to be hidden by overlapping communication and frequency change.

We performed a comprehensive experimental evaluation on real benchmarks and applications across AMD, Intel, and NVIDIA GPUs, demonstrating both the portability and energy efficiency of our phase-based methodology. Our results showed up to 37% energy savings and 1.87× speedup on a single GPU, and up to 68% energy savings and 3.63× speedup on multi-GPU applications running on up to 64 GPUs.

3.3. How can we improve the prediction of the optimal frequency configurations of state-of-the-art models?
We designed a general-purpose energy prediction model based on static compile-time kernel features, which enabled selecting optimal frequency configurations for each kernel according to an energy target metric without requiring application-specific profiling. We then extended this approach by developing domain-specific models that incorporated application- and workload-dependent characteristics, significantly improving prediction accuracy and enabling more effective performance-energy optimization. We validated the domain-specific methodology on two representative scientific GPU workloads, LiGen for molecular docking and Cronos for magnetohydrodynamics, demonstrating that for both applications, domain-specific models achieve 10× lower error compared to the general-purpose energy model.

Research Question 4:

Research Question 4
How well do compilers support autovectorization on modern RISC-V SIMD architectures?

Answer: We explored the autovectorization capabilities of LLVM, LLVM-EPI, and GCC compilers across various versions on real RISC-V RVV hardware. This analysis focuses on how these compilers automatically generate vectorized code compliant with

RVV 1.0 and 0.7 specifications. We evaluated the autovectorization performance across six real-world applications on the AllWinner D1 and BananaPi-F3 RISC-V RVV boards. We highlighted the performance impact of using vector length agnostic (VLA) and vector length specific (VLS) programming and optimized the vector Length Multiplier (LMUL) settings, measuring performance using the geometric mean of speedups across all vectorization patterns compared to the scalar code. Furthermore, we showed how knowledge of the vector length can influence LMUL selection. Our results showed that GCC and LLVM compilers provide poor autovectorization capabilities for RVV 0.7 by only showing 1.05× speedup over scalar code. In contrast, RVV 1.0 significantly improves vectorization coverage and performance: GCC 14 achieves 1.51× speedup with VLA and 1.52× with VLS, while LLVM 19 achieves 1.51× with VLA and up to 1.66× with VLS across different loop categories. LLVM-EPI reaches 1.50× with VLA, performing slightly below LLVM 19. Proper tuning of the LMUL parameter is critical, as higher LMUL values increase throughput for data-parallel operations, and fractional LMUL improves register utilization in workloads with high register pressure, resulting in up to 2.39× speedup in microbenchmarks. Overall, modern compilers can effectively exploit RVV 1.0, but RVV 0.7 remains largely limited in practical autovectorization.

8.3 FUTURE WORK

Several areas have been identified for further investigation and enhancement in future work, covering performance portability, approximate and energy efficient computing and vectorization.

Future work on high-level abstractions:

Future work in this area will focus on further enhancing the performance and portability of the SYCL programming model. In particular, we plan to investigate techniques to improve the quality of code generation by SYCL compilers, aiming to reduce the performance gap between SYCL-generated code and highly optimized CUDA or HIP implementations. This includes exploring compiler optimizations, code transformations, and runtime strategies that can better exploit architecture-specific features while maintaining portability across heterogeneous systems.

Additionally, we aim to extend the evaluation of performance portability to RISC-V hardware, embedded systems, and new SYCL features, providing deeper insights into the evolution of performance portability.

Future work on approximate computing:

Future work in approximate computing will focus on implementing new optimized approximations and autotuing for approximate parameter configurations. For new apporximation, we plan to implement a portable memoization technique for GPUs that leverages both global and shared memory efficiently. This implementation will then be extended with an NVIDIA-specific feature named Thread Block Clustering (TBC) which introduces an additional level in the memory hierarchy and enable fine-grained control over memory placement, improving performance. Approximate computing techniques inherently introduce additional parameters, including which techniques to apply and how to combine them, resulting in a highly complex optimization space. To address this challenge, we plan to develop a general-purpose autotuning framework capable of navigating this space, selecting the most effective combinations of approximation techniques and their corresponding parameter configurations. By integrating state-of-the-art autotuning strategies, this framework will enable efficient exploration of approximation configurations maximizing performance and accuracy across heterogeneous architectures.

Future work on energy-efficient computing:

Future work in energy-efficient computing will extend this research toward carbon-aware optimization strategies. While frequency configurations that minimize energy consumption often reduce carbon emissions, this correlation is not guaranteed, as carbon intensity varies with renewable energy availability, datacenter topology, and time-of-day factors. We plan to develop frequency-scaling policies that explicitly target carbon minimization by incorporating real-time carbon-intensity signals into the decision process. Additionally, we aim to integrate the SYnergy framework into container-based orchestration environments such as Kubernetes, enabling energy- and carbon-aware execution directly in cloud and hybrid HPC-cloud settings. In this direction, we envision the development of carbon-elastic execution mechanisms, where computational workloads can be migrated

dynamically across geographical regions or datacenter nodes based on current carbon intensity values. Such approach would allow the system to transparently relocate jobs to locations with lower carbon footprint, thereby minimizing environmental impact while preserving performance and scalability.

Future work on RISC-V Vector Extension:

Future work on the RISC-V Vector Extension will explore improved compiler and programming model support for RVV, including the analysis of RVV-targeting code generation through SYCL. A key focus will be the optimization of core kernels such as matrix multiplication, developing improved heuristics for LMUL selection and vectorization patterns to better exploit vector resources on modern RISC-V CPUs. Furthermore, we plan to integrate optimized matrix multiplication implementations into established AI and HPC frameworks, such as oneDNN, thereby enabling AI workloads to fully leverage efficient low-level vector execution on RVV-enabled platforms.

BIBLIOGRAPHY

[1] AMD. *ROCm System Management Interface*. 2023. URL: https://github.com/RadeonOpenCompute/rocm_smi_lib (visited on 03/06/2023).

[2] AMD. *AMD EPYC™ CPU Power Management White Paper*. Tech. rep. AMD, 2024. URL: https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/amd-epyc-8004-and-9004-series-cpu-power-management-white-paper.pdf.

[3] Neil Adit and Adrian Sampson. "Performance Left on the Table: An Evaluation of Compiler Autovectorization for RISC-V." In: *IEEE Micro* 42.5 (2022), pp. 41–48. DOI: 10.1109/MM.2022.3184867.

[4] Advanced Micro Devices, Inc. *HIP Programming Guide*. Heterogeneous-compute Interface for Portability (HIP) documentation, part of the ROCm platform. AMD. 2024. URL: https://rocmdocs.amd.com.

[5] Aksel Alpay, Thomas Applencourt, Gordon Brown, Ronan Keryell, and Gregory Lueck. "Using Interoperability Mode in SYCL 2020." In: *International Workshop on OpenCL*. IWOCL'22. Bristol, United Kingdom, United Kingdom: Association for Computing Machinery, 2022. ISBN: 9781450396585. DOI: 10.1145/3529538.3529997. URL: https://doi.org/10.1145/3529538.3529997.

[6] Aksel Alpay and Vincent Heuveline. "SYCL beyond OpenCL: The architecture, current state and future direction of hipSYCL." In: *Proceedings of the International Workshop on OpenCL*. 2020, pp. 1–1.

[7] Aksel Alpay and Vincent Heuveline. "One Pass to Bind Them: The First Single-Pass SYCL Compiler with Unified Code Representation Across Backends." In: *Proceedings of the 2023 International Workshop on OpenCL*. IWOCL '23. Cambridge, United Kingdom: Association for Computing Machinery, 2023. ISBN: 9798400707452. DOI: 10.

1145/3585341.3585351. URL: https://doi.org/10.1145/3585341.3585351.

[8] Francesco Antici, Andrea Bartolini, Zeynep Kiziltan, Ozalp Babaoglu, and Yuetsu Kodama. "Mcbound: An online framework to characterize and classify memorya/compute-bound hpc jobs." In: *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2024, pp. 1–15.

[9] Arm Ltd. *Arm Architecture Reference Manual Supplement: The Scalable Vector Extension (SVE)*. Official reference for the Armv8-A Scalable Vector Extension. Arm. 2021. URL: https://developer.arm.com/documentation.

[10] Krste Asanovic, Ras Bodik, Bryan Catanzaro, Joseph Gebis, Parry Husbands, Kurt Keutzer, David Patterson, William Plishker, John Shalf, and Samuel Webb Williams. "The Landscape of Parallel Computing Research: A view from Berkeley." In: (2006).

[11] Ben Ashbaugh, Alexey Bader, James Brodman, Jeff Hammond, Michael Kinsner, John Pennycook, Roland Schulz, and Jason Sewall. "Data Parallel C++ Enhancing SYCL through Extensions for Productivity and Performance." In: *Proceedings of the International Workshop on OpenCL*. 2020, pp. 1–2.

[12] Ben Ashbaugh, James C Brodman, Michael Kinsner, Gregory Lueck, John Pennycook, and Roland Schulz. "Toward a Better Defined SYCL Memory Consistency Model." In: *International Workshop on OpenCL*. IWOCL'21. Munich, Germany: Association for Computing Machinery, 2021. ISBN: 9781450390330. DOI: 10.1145/3456669.3456696. URL: https://doi.org/10.1145/3456669.3456696.

[13] Woongki Baek and Trishul M Chilimbi. "Green: A framework for supporting energy-conscious programming using controlled approximation." In: *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2010.

[14] Hrishav Bakul Barua and Kartick Chandra Mondal. "Approximate computing: A survey of recent trends—bringing greenness to computing and communication." In: *Journal of The Institution of Engineers (India): Series B* (2019).

[15] David A Beckingsale, Jason Burmark, Rich Hornung, Holger Jones, William Killian, Adam J Kunen, Olga Pearce, Peter Robinson, Brian S Ryujin, and Thomas RW Scogland. "RAJA: Portable performance for large-scale scientific applications." In: *2019 ieee/acm international workshop on performance, portability and productivity in hpc (p3hpc)*. IEEE. 2019, pp. 71–81.

[16] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. "The PARSEC benchmark suite: Characterization and architectural implications." In: *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. 2008, pp. 72–81.

[17] Jacek Biesiada, Aleksey Porollo, Prakash Velayutham, Michal Kouril, and Jaroslaw Meller. "Survey of public domain software for docking simulations and virtual screening." In: *Human Genomics* (2011). ISSN: 14797364.

[18] Joshua Dennis Booth, Jagadish Kotra, Hui Zhao, Mahmut Kandemir, and Padma Raghavan. "Phase detection with hidden Markov models for DVFS on many-core processors." In: *2015 IEEE 35th International Conference on Distributed Computing Systems*. IEEE. 2015, pp. 185–195.

[19] Alexander V. Boukhanovsky, Valeria V. Krzhizhanovskaya, and Marian Bubak. "Urgent computing for decision support in critical situations." In: *Future Generation Computer Systems* (2018).

[20] Nick Brown, Maurice Jamieson, Joseph Lee, and Paul Wang. "Is RISC-V ready for HPC prime-time: Evaluating the 64-core Sophon SG2042 RISC-V CPU." In: *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*. SC-W '23. Denver, CO, USA: Association for Computing Machinery, 2023, 1566–1574. ISBN: 9798400707858. DOI: 10.1145/3624062.3624234. URL: https://doi.org/10.1145/3624062.3624234.

[21] Martin Burtscher, Ivan Zecena, and Ziliang Zong. "Measuring GPU Power with the K20 Built-in Sensor." In: *Proceedings of Workshop on General Purpose Processing Using GPUs*. 2014.

[22] D. Callahan, J. Dongarra, and D. Levine. "Vectorizing compilers: a test suite and results." In: *Supercomputing '88:Proceedings of the 1988 ACM/IEEE Conference on Supercomputing, Vol. I*. 1988, pp. 98–105. DOI: 10.1109/SUPERC.1988.44642.

[23] Simone Campanoni, Glenn Holloway, Gu-Yeon Wei, and David Brooks. "HELIX-UP: Relaxing program semantics to unleash parallelization." In: *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2015.

[24] Lorenzo Carpentieri and Biagio Cosenza. "SYprox: Combining Host and Device Perforation with Mixed Precision Approximation on Heterogeneous Architectures." In: *Proceedings of the 39th ACM International Conference on Supercomputing*. 2025, pp. 1–12.

[25] Lorenzo Carpentieri, Marco D'Antonio, Kaijie Fan, Luigi Crisci, Biagio Cosenza, Federico Ficarelli, Daniele Cesarini, Gianmarco Accordi, Davide Gadioli, Gianluca Palermo, et al. "Domain-specific energy modeling for drug discovery and magnetohydrodynamics applications." In: *Proceedings of the SC'23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis*. 2023, pp. 1790–1800.

[26] Daniele Cesarini, Andrea Bartolini, Pietro Bonfà, Carlo Cavazzoni, and Luca Benini. "Countdown: a run-time library for performance-neutral energy saving in MPI applications." In: *IEEE Transactions on Computers* 70.5 (2020), pp. 682–695.

[27] 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: *2009 IEEE international symposium on workload characterization (IISWC)*. Ieee. 2009, pp. 44–54.

[28] Stefano Cherubin, Daniele Cattaneo, Michele Chiari, Antonio Di Bello, and Giovanni Agosta. "TAFFO: Tuning Assistant for Floating to Fixed Point Optimization." In: *IEEE Embedded Systems Letters* (2020).

[29] Jee W. Choi and Richard W. Vuduc. "Analyzing the Energy Efficiency of the Fast Multipole Method Using a DVFS-Aware Energy Model." In: *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2016.

[30] Julita Corbalán, Lluis Alonso, Jordi Aneas, and Luigi Brochard. "Energy Optimization and Analysis with EAR." In: *IEEE International Conference on Cluster Computing, CLUSTER 2020, Kobe, Japan, September 14-17, 2020*. IEEE, 2020, pp. 464–472. DOI: 10.1109/CLUSTER49012.2020.00067. URL: https://doi.org/10.1109/CLUSTER49012.2020.00067.

[31] Julita Corbalan, Oriol Vidal, Lluis Alonso, and Jordi Aneas. "Explicit uncore frequency scaling for energy optimisation policies with EAR in Intel architectures." In: *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE. 2021, pp. 572–581.

[32] NVIDIA Corporation. *NVIDIA Grace Hopper 100 architecture*. https://resources.nvidia.com/en-us-tensor-core. 2024.

[33] Ewan Crawford, Pablo Reble, Ben Tracy, and Julian Miller. "Towards Deferred Execution of a SYCL Command Graph." In: *Proceedings of the 2023 International Workshop on OpenCL*. 2023, pp. 1–2.

[34] Luigi Crisci, Lorenzo Carpentieri, Peter Thoman, Aksel Alpay, Vincent Heuveline, and Biagio Cosenza. "SYCL-Bench 2020: Benchmarking SYCL 2020 on AMD, Intel, and NVIDIA GPUs." In: (2024).

[35] Eva Darulova and Viktor Kuncak. "Towards a compiler for reals." In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* (2017).

[36] Harold David, Eitan Gorbatov, Ulf R. Hanebutte, Rahul Khanna, and Christian Le. "RAPL: Memory power estimation and capping." In: *Proceedings of the 2010 ACM/IEEE International Symposium on Low-Power Electronics and Design*. IEEE. 2010, pp. 189–194.

[37] Tom Deakin. *SYCL State of the Union IWOCL24*. `https://www.iwocl.org/wp-content/uploads/iwocl-2024-sycl-sou.pdf`.

[38] Tom Deakin, Simon McIntosh-Smith, James Price, Andrei Poenaru, Patrick Atkinson, Codrin Popa, and Justin Salmon. "Performance portability across diverse computer architectures." In: *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE. 2019, pp. 1–13.

[39] Tom Deakin, James Price, Matt Martineau, and Simon McIntosh-Smith. "Evaluating attainable memory bandwidth of parallel programming models via BabelStream." In: *International Journal of Computational Science and Engineering* 17.3 (2018), pp. 247–262.

[40] Kalyanmoy Deb, Karthik Sindhya, and Jussi Hakanen. "Multi-objective optimization." In: *Decision sciences*. 2016.

[41] JMF Donnert, H Jang, P Mendygral, Gianfranco Brunetti, D Ryu, and TW Jones. "WENO–WOMBAT: Scalable Fifth-order Constrained-transport Magnetohydrodynamics for Astrophysical Applications." In: *The Astrophysical Journal Supplement Series* (2019).

[42] Jürgen Dreher and Rainer Grauer. "Racoon: A parallel mesh-adaptive framework for hyperbolic conservation laws." In: *Parallel Computing* (2005).

[43] Jonathan Eastep, Steve Sylvester, Christopher Cantalupo, Brad Geltz, Federico Ardanaz, Asma Al-Rawi, Kelly Livingston, Fuat Keceli, Matthias Maiterth, and Siddhartha Jana. "Global extensible open power manager: A vehicle for HPC community collaboration on co-designed energy management solutions." In: *High Performance Computing: 32nd International Conference, ISC High Performance*. Springer. 2017, pp. 394–412.

[44] H Carter Edwards, Christian R Trott, and Daniel Sunderland. "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns." In: *Journal of parallel and distributed computing* 74.12 (2014), pp. 3202–3216.

[45] Mark Endrei, Chao Jin, Minh Ngoc Dinh, David Abramson, Heidi Poxon, Luiz DeRose, and Bronis R de Supinski. "Energy efficiency modeling of parallel applications." In: *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2018, pp. 212–224.

[46] Hadi Esmaeilzadeh, Emily Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. "Dark Silicon and the End of Multicore Scaling." In: (2011), pp. 365–376. DOI: 10.1145/2000064.2000108.

[47] Kaijie Fan, Biagio Cosenza, and Ben H. H. Juurlink. "Predictable GPUs Frequency Scaling for Energy and Performance." In: *Proceedings of the 48th International Conference on Parallel Processing, ICPP, Kyoto, Japan, August 05-08*. 2019, 52:1–52:10.

[48] Kaijie Fan, Marco D'Antonio, Lorenzo Carpentieri, Biagio Cosenza, Federico Ficarelli, and Daniele Cesarini. "SYnergy: Fine-grained Energy-Efficient Heterogeneous Computing for Scalable Energy Saving." In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis SC'23*. 2023. DOI: 10.1145/3581784.3607055. URL: https://doi.org/10.1145/3581784.3607055.

[49] Jing Ge Feng, Ye Ping He, and Qiu Ming Tao. "Evaluation of compilers' capability of automatic vectorization based on source code analysis." In: *Scientific Programming* 2021.1 (2021), p. 3264624.

[50] Xixhou Feng, Rong Ge, and Kirk W Cameron. "Power and energy profiling of scientific applications on distributed systems." In: *19th IEEE International Parallel and Distributed Processing Symposium*. IEEE. 2005, 10–pp.

[51] R. Ferrer. *LLVM-EPI Repository*. `https://repo.hca.bsc.es/gitlab/rferrer/llvm-epi`. Accessed: [Insert Date Here]. 2024.

[52] Mikhail Figurnov, Aizhan Ibraimova, Dmitry P Vetrov, and Pushmeet Kohli. "PerforatedCNNs: Acceleration through elimination of redundant convolutions." In: *Advances in neural information processing systems* (2016).

[53] Zane Fink, Konstantinos Parasyris, Giorgis Georgakoudis, and Harshitha Menon. "HPAC-Offload: Accelerating HPC Applications with Portable Approximate Computing on the GPU." In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2023.

[54] The RISC-V Foundation. *RISC-V Vector Extension*. `https://github.com/riscv/riscv-v-spec`. 2021.

[55] Vincent W Freeh, Feng Pan, Nandini Kappiah, David K Lowenthal, and Robert Springer. "Exploring the energy-time tradeoff in MPI programs on a power-scalable cluster." In: *19th IEEE International Parallel and Distributed Processing Symposium*. IEEE. 2005, 10–pp.

[56] Sébastien Fromang, Patrick Hennebelle, and Romain Teyssier. "A high order Godunov scheme with constrained transport and adaptive mesh refinement for astrophysical magnetohydrodynamics." In: *Astronomy & Astrophysics* (2006).

[57] Davide Gadioli, Emanuele Vitali, Federico Ficarelli, Chiara Latini, Candida Manelfi, Carmine Talarico, Cristina Silvano, Carlo Cavazzoni, Gianluca Palermo, and Andrea Rosario Beccari. "EXSCALATE: An extreme-scale virtual screening platform for drug discovery targeting polypharmacology to fight SARS-CoV-2." In: *IEEE Transactions on Emerging Topics in Computing* 11.1 (2022), pp. 170–181.

[58] Adriano M Garcia, Matheus Serpa, Dalvan Griebler, Claudio Schepke, Luiz GL Fernandes, and Philippe OA Navaux. "The impact of CPU frequency scaling on power consumption of computing infrastructures." In: *International Conference on Computational Science and Its Applications*. Springer. 2020, pp. 142–157.

[59] Rong Ge, Xizhou Feng, and Kirk W Cameron. "Modeling and evaluating energy-performance efficiency of parallel processing on multicore based power aware systems." In: *2009 ieee international symposium on parallel & distributed processing*. 2009.

[60] Rong Ge, Ryan Vogt, Jahangir Majumder, Arif Alam, Martin Burtscher, and Ziliang Zong. "Effects of Dynamic Voltage and Frequency Scaling on a K20 GPU." In: *2013 42nd International Conference on Parallel Processing*. 2013.

[61] Neha Gholkar, Frank Mueller, and Barry Rountree. "Power tuning HPC jobs on power-constrained systems." In: *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*. 2016, pp. 179–191.

[62] Jens Glaser et al. "High-throughput virtual laboratory for drug discovery using massive datasets." In: *The International Journal of High Performance Computing Applications* (2021).

[63] Andrew Gozillon, Ronan Keryell, Lin-Ya Yu, Gauthier Harnisch, and Paul Keir. "triSYCL for Xilinx FPGA." In: *The 2020 International Conference on High Performance Computing and Simulation. IEEE*. 2020.

[64] Khronos OpenCL Working Group. *OpenCL 3.0 API Specification*. Tech. rep. Khronos Group, 2021.

[65] Khronos SYCL Working Group. *SYCL 2020 Specification, revision 9*. Tech. rep. Khronos Group, 2024.

[66] Joao Guerreiro, Aleksandar Ilic, Nuno Roma, and Pedro Tomas. "GPGPU Power Modelling for Multi-Domain Voltage-Frequency Scaling." In: *24th IEEE International Symposium on High-Performance Computing Architecture, HPCA*. 2018.

[67] João Guerreiro, Aleksandar Ilic, Nuno Roma, and Pedro Tomás. "GPU Static Modeling Using PTX and Deep Structured Learning." In: *IEEE Access* (2019).

[68] Tomoaki Hamano, Toshio Endo, and Satoshi Matsuoka. "Power-aware dynamic task scheduling for heterogeneous accelerated clusters." In: *2009 IEEE International Symposium on Parallel & Distributed Processing*. 2009.

[69] Meng Hao, Weizhe Zhang, Yiming Wang, Gangzhao Lu, Farui Wang, and Athanasios V. Vasilakos. "Fine-Grained Powercap Allocation for Power-Constrained Systems Based on Multi-Objective Machine Learning." In: *IEEE Trans. Parallel Distributed Syst.* (2021).

[70] Mark Harris et al. "Optimizing parallel reduction in CUDA." In: *NVIDIA developer technology* 2.4 (2007), pp. 1–39.

[71] JA Herdman, WP Gaudin, Simon McIntosh-Smith, Michael Boulton, David A Beckingsale, Andrew C Mallinson, and Stephen A Jarvis. "Accelerating hydrocodes with OpenACC, OpenCL and CUDA." In: *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. IEEE. 2012, pp. 465–471.

[72] Michael A. Heroux, Lois McInnes, Xiaoye Sherry Li, James Ahrens, Todd Munson, Kathryn Mohror, Terece Turton, Jeffrey Vetter, and Rajeev Thakur. *ECP Software Technology Capability Assessment Report*. Tech. rep. June 2022. DOI: 10.2172/1888898. URL: https://www.osti.gov/biblio/1888898.

[73] Henry Hoffmann, Sasa Misailovic, Stelios Sidiroglou, Anant Agarwal, and Martin Rinard. "Using code perforation to improve performance, reduce energy consumption, and respond to failures." In: (2009).

[74] Bart van der Holst, Rony Keppens, and Zakaria Meliani. "A multidimensional grid-adaptive relativistic magnetofluid code." In: *Computer Physics Communications* (2008).

[75] Shadi Ibrahim, Tien-Dat Phan, Alexandra Carpen-Amarie, Houssem-Eddine Chihoub, Diana Moise, and Gabriel Antoniu. "Governing energy consumption in Hadoop through CPU frequency scaling: An analysis." In: *Future Generation Computer Systems* 54 (2016), pp. 219–232.

[76] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Includes AVX, AVX2, and AVX-512 SIMD instruction set extensions. Intel. 2023. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[77] Intel Corporation. *SYCL EXT ONEAPI Bfloat16 Math Functions*. Accessed: 2024-09-12. 2024. URL: https://github.com/intel/llvm/blob/sycl/sycl/doc/extensions/experimental/sycl_ext_oneapi_bfloat16_math_functions.asciidoc.

[78] Intel. *oneAPI Data Parallel C++ compiler*. 2022. URL: https://github.com/intel/llvm/releases/tag/2022-09.

[79] Intel. *Intel Level Zero API Specification*. 2023. URL: https://spec.oneapi.io/level-zero/latest/index.html (visited on 03/06/2023).

[80] Hisao Ishibuchi, Ryo Imada, Yu Setoguchi, and Yusuke Nojima. "Reference point specification in hypervolume calculation for fair comparison and efficient search." In: *Proceedings of the genetic and evolutionary computation conference*. 2017.

[81] Shailendra Jain et al. "A 280mV-to-1.2V wide-operating-range IA-32 processor in 32nm CMOS." In: *IEEE International Solid-State Circuits Conference, ISSCC*. 2012, pp. 66–68.

[82] Zheming Jin. *The rodinia benchmark suite in SYCL*. Tech. rep. Argonne National Lab.(ANL), Argonne, IL (United States). Argonne Leadership ..., 2020.

[83] Zheming Jin and Jeffrey S Vetter. "Understanding performance portability of bioinformatics applications in sycl on an nvidia gpu." In: *2022 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*. IEEE. 2022, pp. 2190–2195.

[84] Zheming Jin and Jeffrey S Vetter. "A Benchmark Suite for Improving Performance Portability of the SYCL Programming Model." In: *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE. 2023, pp. 325–327.

[85] S Joube, H Grasland, D Chamont, and E Brunet. "Comparing SYCL data transfer strategies for tracking use cases." In: *Journal of Physics: Conference Series* 2438.1 (2023), p. 012018. DOI: 10.1088/1742-6596/2438/1/012018. URL:

https://dx.doi.org/10.1088/1742-6596/2438/1/012018.

[86] Yinan Ke, Mulya Agung, and Hiroyuki Takizawa. "NeoSYCL: A SYCL Implementation for SX-Aurora TSUBASA." In: *The International Conference on High Performance Computing in Asia-Pacific Region*. HPC Asia 2021. Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, 50–57. ISBN: 9781450388429. DOI: 10.1145/3432261.3432268. URL: https://doi.org/10.1145/3432261.3432268.

[87] Rony Keppens, Zakaria Meliani, Allard Jan van Marle, Peter Delmont, Alkis Vlasis, and Bart van der Holst. "Parallel, grid-adaptive approaches for relativistic hydro and magnetohydrodynamics." In: *Journal of Computational Physics* (2012).

[88] Hamidreza Khaleghzadeh, Muhammad Fahad, Arsalan Shahid, Ravi Reddy Manumachu, and Alexey Lastovetsky. "Bi-objective optimization of data-parallel applications on heterogeneous HPC platforms for performance and energy through workload distribution." In: *IEEE Transactions on Parallel and Distributed Systems* 32.3 (2020), pp. 543–560.

[89] Jungsoo Kim, Sungjoo Yoo, and Chong-Min Kyung. "Program phase-aware dynamic voltage scaling under variable computational workload and memory stall environment." In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30.1 (2010), pp. 110–123.

[90] Ralf Kissmann, David Huber, and Philipp Gschwandtner. "High-Resolution Simulations of LS 5039." In: *A&A (Forthcoming)* (2023).

[91] Ralf Kissmann, Jens Kleimann, Barbara L. Krebl, and Tobias Wiengarten. "The CRONOS code for astrophysical magnetohydrodynamics." In: *The Astrophysical Journal Supplement Series* (2018).

[92] Masaaki Kondo, Hiroshi Sasaki, and Hiroshi Nakamura. "Improving fairness, throughput and energy-efficiency on a chip multiprocessor through DVFS." In: *ACM SIGARCH Computer Architecture News* 35.1 (2007), pp. 31–38.

[93] Karlo Kraljic, Daniel Kerger, and Martin Schulz. "Energy Efficient Frequency Scaling on GPUs in Heterogeneous HPC Systems." In: *International Conference on Architecture of Computing Systems*. Springer. 2022, pp. 3–16.

[94] Milind Kulkarni, Martin Burtscher, Calin Casçaval, and Keshav Pingali. "Lonestar: A suite of parallel irregular programs." In: *2009 IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE. 2009, pp. 65–76.

[95] Ignacio Laguna, Paul C Wood, Ranvijay Singh, and Saurabh Bagchi. "Gpumixer: Performance-driven floating-point tuning for gpu scientific applications." In: *High Performance Computing: 34th International Conference, ISC High Performance 2019, Frankfurt/Main, Germany, June 16–20, 2019, Proceedings 34*. 2019.

[96] Hung-Ming Lai, Jenq-Kuen Lee, and Yuan-Shin Hwang. "Enhancing LLVM Optimizations for Linear Recurrence Programs on RVV." In: *Proceedings of the 52nd International Conference on Parallel Processing Workshops*. ICPP Workshops '23. Salt Lake City, UT, USA: Association for Computing Machinery, 2023, 79–87. ISBN: 9798400708428. DOI: 10.1145/3605731.3605904. URL: https://doi.org/10.1145/3605731.3605904.

[97] Sohan Lal, Aksel Alpay, Philip Salzmann, Biagio Cosenza, Alexander Hirsch, Nicolai Stawinoga, Peter Thoman, Thomas Fahringer, and Vincent Heuveline. "SYCL-Bench: A Versatile Cross-Platform Benchmark Suite for Heterogeneous Computing." In: *Euro-Par 2020: Parallel Processing*. 2020.

[98] Michael O. Lam, Jeffrey K. Hollingsworth, Bronis R. de Supinski, and Matthew P. Legendre. "Automatically Adapting Programs for Mixed-Precision Floating-Point Computation." In: *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*. 2013.

[99] James H Laros III, Kevin Pedretti, Suzanne M Kelly, Wei Shu, Kurt Ferreira, John Van Dyke, Courtenay Vaughan, James H Laros III, Kevin Pedretti, Suzanne M Kelly, et al.

"Energy delay product." In: *Energy-Efficient High Performance Computing: Measurement and Tuning* (2013), pp. 51–55.

[100] Joseph K. L. Lee, Maurice Jamieson, and Nick Brown. "Backporting RISC-V Vector Assembly." In: *High Performance Computing*. Ed. by Amanda Bienz, Michèle Weiland, Marc Baboulin, and Carola Kruse. Cham: Springer Nature Switzerland, 2023, pp. 433–443. ISBN: 978-3-031-40843-4.

[101] Joseph K. L. Lee, Maurice Jamieson, Nick Brown, and Ricardo Jesus. "Test-Driving RISC-V Vector Hardware for HPC." In: *High Performance Computing*. Ed. by Amanda Bienz, Michèle Weiland, Marc Baboulin, and Carola Kruse. Cham: Springer Nature Switzerland, 2023, pp. 419–432. ISBN: 978-3-031-40843-4.

[102] Kooktae Lee and Raktim Bhattacharya. "On the relaxed synchronization for massively parallel numerical algorithms." In: *2016 American Control Conference (ACC)*. 2016.

[103] Dong Li, Dimitrios S Nikolopoulos, Kirk Cameron, Bronis R de Supinski, and Martin Schulz. "Power-aware MPI task aggregation prediction for high-end computing systems." In: *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE. 2010, pp. 1–12.

[104] Ruo-Shi Li, Ping Peng, Zhi-Yuan Shao, Hai Jin, and Ran Zheng. "Evaluating RISC-V Vector Instruction Set Architecture Extension with Computer Vision Workloads." In: *Journal of Computer Science and Technology* 38.4 (2023), pp. 807–820. DOI: 10.1007/s11390-023-1266-6. URL: https://www.sciopen.com/article/10.1007/s11390-023-1266-6.

[105] Shikai Li, Sunghyun Park, and Scott Mahlke. "Sculptor: Flexible approximation with selective dynamic loop perforation." In: *Proceedings of the 2018 International Conference on Supercomputing*. 2018.

[106] Jhih-Kuan Lin, Yu-Lun Yang, Hung-Ming Lai, and Jenq-Kuen Lee. "Rewriting and Optimizing Vector Length Agnostic Intrinsics from Arm SVE to RVV." In: *Workshop Proceedings of the 53rd International Conference on Parallel*

Processing. ICPP Workshops '24. Gotland, Sweden: Association for Computing Machinery, 2024, 38–47. ISBN: 9798400718021. DOI: 10.1145/3677333.3678151. URL: https://doi.org/10.1145/3677333.3678151.

[107] Wei-Chen Lin, Tom Deakin, and Simon McIntosh-Smith. "On Measuring the Maturity of SYCL Implementations by Tracking Historical Performance Improvements." In: *International Workshop on OpenCL*. IWOCL'21. Munich, Germany: Association for Computing Machinery, 2021. ISBN: 9781450390330. DOI: 10.1145/3456669.3456701. URL: https://doi.org/10.1145/3456669.3456701.

[108] Charles Lively, Xingfu Wu, Valerie Taylor, Shirley Moore, Hung-Ching Chang, Chun-Yi Su, and Kirk Cameron. "Power-aware predictive models of hybrid (MPI/OpenMP) scientific applications on multicore systems." In: *Computer Science-Research and Development* 27 (2012), pp. 245–253.

[109] André Lopes, Frederico Pratas, Leonel Sousa, and Aleksandar Ilic. "Exploring GPU performance, power and energy-efficiency bounds with Cache-aware Roofline Modeling." In: *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2017.

[110] Alexandre Lopoukhine, Federico Ficarelli, Christos Vasiladiotis, Anton Lydike, Josse Van Delm, Alban Dutilleul, Luca Benini, Marian Verhelst, and Tobias Grosser. "A Multi-Level Compiler Backend for Accelerated Micro-Kernels Targeting RISC-V ISA Extensions." In: *Proceedings of the 2025 Conference on Compiler and Generator Optimization (CGO)*. 2025.

[111] Liming Lou, Paul Nguyen, Jason Lawrence, and Connelly Barnes. "Image perforation: Automatically accelerating image pipelines by intelligently skipping samples." In: *ACM Transactions on Graphics (TOG)* (2016).

[112] Erika Susana Alcorta Lozano and Andreas Gerstlauer. "Learning-based phase-aware multi-core CPU workload forecasting." In: *ACM transactions on design automation of electronic systems* 28.2 (2022), pp. 1–27.

[113] Robert Lucas, James Ang, Keren Bergman, Shekhar Borkar, William Carlson, Laura Carrington, George Chiu, Robert Colwell, William Dally, Jack Dongarra, et al. "Top ten exascale research challenges." In: *DOE ASCAC subcommittee report* (2014), pp. 1–86.

[114] Daniel Maier, Biagio Cosenza, and Ben Juurlink. "Local memory-aware kernel perforation." In: *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. 2018.

[115] Daniel Maier and Ben Juurlink. "Model-Based Loop Perforation." In: *European Conference on Parallel Processing*. 2021.

[116] Daniel Maier, Nadjib Mammeri, Biagio Cosenza, and Ben Juurlink. "Approximating memory-bound applications on mobile GPUs." In: *2019 International Conference on High Performance Computing & Simulation (HPCS)*. 2019.

[117] Konrad Malkowski, Padma Raghavan, Mahmut Kandemir, and Mary Jane Irwin. "Phase-aware adaptive hardware selection for power-efficient scientific computations." In: *Proceedings of the 2007 international symposium on Low power electronics and design*. 2007, pp. 403–406.

[118] Simon McIntosh-Smith. *TSVC 2 Loops*. `https://github.com/UoB-HPC/TSVC_2`.

[119] Andrea Mignone, C Zanni, Petros Tzeferacos, B Van Straalen, P Colella, and G Bodo. "The PLUTO code for adaptive mesh computations in astrophysical fluid dynamics." In: *The Astrophysical Journal Supplement Series* (2011).

[120] Nenad Mijić and Davor Davidović. "Benchmark DPC++ code and performance portability on heterogeneous architectures." In: *2023 46th MIPRO ICT and Electronics Convention (MIPRO)*. 2023, pp. 331–337. DOI: `10.23919/MIPRO57284.2023.10159832`.

[121] Asit K Mishra, Rajkishore Barik, and Somnath Paul. "iACT: A software-hardware framework for understanding the scope of approximate computing." In: *Workshop on Approximate Computing Across the System Stack (WACAS)*. 2014.

[122] Subrata Mitra, Manish K Gupta, Sasa Misailovic, and Saurabh Bagchi. "Phase-aware optimization in approximate computing." In: *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2017.

[123] Sparsh Mittal. "A survey of techniques for approximate computing." In: *ACM Computing Surveys (CSUR)* (2016).

[124] Mohammad Alaul Haque Monil, Mehmet E Belviranli, Seyong Lee, Jeffrey S Vetter, and Allen D Malony. "Mephesto: Modeling energy-performance in heterogeneous SoCs and their trade-offs." In: *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*. 2020, pp. 413–425.

[125] Natarajan Arul Murugan, Artur Podobas, Davide Gadioli, Emanuele Vitali, Gianluca Palermo, and Stefano Markidis. "A Review on Parallel Virtual Screening Softwares for High-Performance Computers." In: *Pharmaceuticals* (2022).

[126] *NVML API reference guide :: GPU deployment and management documentation*. `https://docs.nvidia.com/deploy/nvml-api/index.html`. (Accessed on 03/21/2024).

[127] JR Neely. *DOE centers of excellence performance portability meeting*. Tech. rep. Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2016.

[128] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. "Scalable Parallel Programming with CUDA: Is CUDA the Parallel Programming Model that Application Developers Have Been Waiting for?" In: *Queue* 6.2 (2008), pp. 40–53.

[129] Md SQ Zulkar Nine, Tevfik Kosar, Muhammed Fatih Bulut, and Jinho Hwang. "GreenNFV: Energy-Efficient Network Function Virtualization with Service Level Agreement Constraints." In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2023, pp. 1–12.

[130] Matthew R. Norman. *miniWeather*. [Computer Software] `https://doi.org/10.11578/dc.20201001.88`. 2020. DOI: `10.11578/dc.20201001.88`.

[131] OpenMP Architecture Review Board. *OpenMP Application Programming Interface Version 5.2*. https://www.openmp.org/specifications/. Accessed: YYYY-MM-DD. 2021.

[132] Anne-Cecile Orgerie, Marcos Dias de Assuncao, and Laurent Lefevre. "A survey on techniques for improving the energy efficiency of large-scale distributed systems." In: *ACM Computing Surveys (CSUR)* 46.4 (2014), pp. 1–31.

[133] Nataraj S. Pagadala, Khajamohiddin Syed, and Jack Tuszynski. "Software for molecular dockingAMD ROCm profiler: a review." In: *Biophysical Reviews* (2017).

[134] Konstantinos Parasyris, Giorgis Georgakoudis, Harshitha Menon, James Diffenderfer, Ignacio Laguna, Daniel Osei-Kuffuor, and Markus Schordan. "HPAC: Evaluating Approximate Computing Techniques on HPC OpenMP Applications." In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2021.

[135] S John Pennycook, Jason D Sewall, Douglas W Jacobsen, Tom Deakin, and Simon McIntosh-Smith. "Navigating performance, portability, and productivity." In: *Computing in Science & Engineering* 23.5 (2021), pp. 28–38.

[136] Simon J Pennycook, Simon D Hammond, Steven A Wright, JA Herdman, Ian Miller, and Stephen A Jarvis. "An investigation of the performance portability of OpenCL." In: *Journal of Parallel and Distributed Computing* 73.11 (2013), pp. 1439–1450.

[137] Simon J Pennycook, Jason D Sewall, and Victor W Lee. "A metric for performance portability." In: *arXiv preprint arXiv:1611.07409* (2016).

[138] Matteo Perotti, Samuel Riedel, Matheus Cavalcante, and Luca Benini. "Spatz: Clustering Compact RISC-V-Based Vector Units to Maximize Computing Efficiency." In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2025).

[139] Philipp Pindl. "Performance Portability for HPC Applications through the RAJA abstraction layer." In: (2022).

[140] Angela Pohl, Biagio Cosenza, and Ben Juurlink. "Portable cost modeling for auto-vectorizers." In: *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE. 2019, pp. 359–369.

[141] Angela Pohl, Biagio Cosenza, and Ben Juurlink. "Vectorization cost modeling for NEON, AVX and SVE." In: *Performance Evaluation* 140 (2020), p. 102106.

[142] Angela Pohl, Biagio Cosenza, Mauricio Alvarez Mesa, Chi Ching Chi, and Ben Juurlink. "An evaluation of current SIMD programming models for C++." In: *Proceedings of the 3rd Workshop on Programming Models for SIMD/Vector Processing*. 2016, pp. 1–8.

[143] Angela Pohl, Mirko Greese, Biagio Cosenza, and Ben Juurlink. "A performance analysis of vector length agnostic code." In: *2019 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE. 2019, pp. 159–164.

[144] *Power Capping Framework — The Linux Kernel documentation*. `https://www.kernel.org/doc/html/next/power/powercap/powercap.html`. (Accessed on 03/28/2024).

[145] Meikang Qiu, Zhong Ming, Jiayin Li, Shaobo Liu, Bin Wang, and Zhonghai Lu. "Three-phase time-aware energy minimization with DVFS and unrolling for chip multiprocessors." In: *Journal of Systems Architecture* 58.10 (2012), pp. 439–445.

[146] RISC-V International. *The RISC-V Vector Extension, Version 1.0*. Specification for RVV (RISC-V Vector Extension). RISC-V International. 2021. URL: `https://riscv.org/technical/specifications`.

[147] Srinivasan Ramesh, Swann Perarnau, Sridutt Bhalachandra, Allen D. Malony, and Peter H. Beckman. "Understanding the Impact of Dynamic Power Capping on Application Progress." In: *2019 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2019, Rio de Janeiro, Brazil, May 20-24, 2019*. 2019.

[148] Cristóbal Ramírez, César Alejandro Hernández, Oscar Palomar, Osman Unsal, Marco Antonio Ramírez, and Adrián Cristal. "A RISC-V Simulator and Benchmark Suite for Designing and Evaluating Vector Architectures." In: *ACM Trans. Archit. Code Optim.* 17.4 (2020). ISSN: 1544-3566. DOI: 10.1145/3422667. URL: https://doi.org/10.1145/3422667.

[149] Goutham Kalikrishna Reddy Kuncham, Rahul Vaidya, and Mahesh Barve. "Performance Study of GPU applications using SYCL and CUDA on Tesla V100 GPU." In: *2021 IEEE High Performance Extreme Computing Conference (HPEC)*. 2021, pp. 1–7. DOI: 10.1109/HPEC49654.2021.9622813.

[150] Istvan Z Reguly. "Evaluating the performance portability of SYCL across CPUs and GPUs on bandwidth-bound applications." In: *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*. 2023, pp. 1038–1047.

[151] Lakshminarayanan Renganarayana, Vijayalakshmi Srinivasan, Ravi Nair, and Daniel Prener. "Programming with relaxed synchronization." In: *Proceedings of the 2012 ACM workshop on Relaxing synchronization for multicore and manycore scalability*. 2012.

[152] Haris Ribic and Yu David Liu. "Aequitas: Coordinated energy management across parallel applications." In: *Proceedings of the 2016 International Conference on Supercomputing*. 2016, pp. 1–12.

[153] Martin Rinard, Henry Hoffmann, Sasa Misailovic, and Stelios Sidiroglou. "Patterns and Statistical Analysis for Understanding Reduced Resource Computing." In: *ACM Sigplan Notices* (2010).

[154] Todd J. Rosedahl. *POWER9 EnergyScale – Configuration and Management*. IBM Community Blog. 2020. URL: https://community.ibm.com/community/user/power/blogs/todd-j-rosedahl1/2020/06/17/power9-energyscale-configuration-and-management.

[155] Barry Rountree, Dong H Ahn, Bronis R De Supinski, David K Lowenthal, and Martin Schulz. "Beyond DVFS: A first look at performance under a hardware-enforced power bound." In: *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*. IEEE. 2012, pp. 947–953.

[156] Barry Rountree, David K Lowenthal, Shelby Funk, Vincent W Freeh, Bronis R De Supinski, and Martin Schulz. "Bounding energy consumption in large-scale MPI programs." In: *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*. 2007, pp. 1–9.

[157] Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H. Bailey, Costin Iancu, and David Hough. "Precimonious: Tuning Assistant for Floating-Point Precision." In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 2013.

[158] Karl Rupp. *50 Years of Microprocessor Trend Data*. https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/. 2018.

[159] *SYCL Specification*. https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html. 2023.

[160] *SYnergy API*. 2023. URL: https://github.com/unisa-hpc/SYnergy (visited on 03/06/2023).

[161] Issa Saba, Eishi Arima, Dai Liu, and Martin Schulz. "Orchestrated Co-scheduling, Resource Partitioning, and Power Capping on CPU-GPU Heterogeneous Systems via Machine Learning." In: *Architecture of Computing Systems - 35th International Conference, ARCS 2022, Heilbronn, Germany, September 13-15, 2022, Proceedings*. Ed. by Martin Schulz, Carsten Trinitis, Nikela Papadopoulou, and Thilo Pionteck. 2022.

[162] Philip Salzmann, Fabian Knorr, Peter Thoman, Philipp Gschwandtner, Biagio Cosenza, and Thomas Fahringer. "An Asynchronous Dataflow-Driven Execution Model For Distributed Accelerator Computing." In: *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Inter-*

net Computing (CCGrid). 2023, pp. 82–93. DOI: 10.1109/CCGrid57682.2023.00018.

[163] Mehrzad Samadi, Davoud Anoushe Jamshidi, Janghaeng Lee, and Scott Mahlke. "Paraprox: Pattern-based approximation for data parallel applications." In: *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*. 2014.

[164] Mehrzad Samadi, Davoud Anoushe Jamshidi, Janghaeng Lee, and Scott Mahlke. "Paraprox: Pattern-based approximation for data parallel applications." In: *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*. 2014.

[165] Mehrzad Samadi, Janghaeng Lee, D Anoushe Jamshidi, Amir Hormati, and Scott Mahlke. "Sage: Self-tuning approximation for graphics engines." In: *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. 2013.

[166] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. "EnerJ: Approximate Data Types for Safe and General Low-Power Computation." In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2011.

[167] Nora Sánchez Gassen, Oskar Penje, and Elin Slätmo. *Global goals for local priorities: The 2030 Agenda at local level*. Nordregio, 2018.

[168] Osman Sarood, Akhil Langer, Abhishek Gupta, and Laxmikant V. Kalé. "Maximizing Throughput of Overprovisioned HPC Data Centers Under a Strict Power Budget." In: *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2014, 2014*, pp. 807–818.

[169] Robert R Schaller. "Moore's law: past, present and future." In: *IEEE spectrum* 34.6 (2002), pp. 52–59.

[170] Julian Scheipl, Amir Raoofy, Michael Ott, and Josef Weidendorfer. "Phase-aware System-Side Sampling for HPC." In: *Proceedings of the 20th ACM International Conference on Computing Frontiers*. 2023, pp. 220–221.

[171] Lukas Sekanina. "Introduction to approximate computing: Embedded tutorial." In: *2016 IEEE 19th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*. 2016.

[172] Ke Shang, Hisao Ishibuchi, Linjun He, and Lie Meng Pang. "A survey on the hypervolume indicator in evolutionary multiobjective optimization." In: *IEEE Transactions on Evolutionary Computation* (2020).

[173] Meng-Shiuan Shih, Hung-Ming Lai, Chao-Lin Lee, Chung-Kai Chen, and Jenq-Kuen Lee. "Register-Pressure Aware Predicator for Length Multiplier of RVV." In: *Workshop Proceedings of the 51st International Conference on Parallel Processing*. ICPP Workshops '22. Bordeaux, France: Association for Computing Machinery, 2023. ISBN: 9781450394451. DOI: 10.1145/3547276.3548513. URL: https://doi.org/10.1145/3547276.3548513.

[174] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. "Managing performance vs. accuracy trade-offs with loop perforation." In: *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 2011.

[175] Sergi Siso, Wes Armour, and Jeyarajan Thiyagalingam. "Evaluating Auto-Vectorizing Compilers through Objective Withdrawal of Useful Information." In: *ACM Trans. Archit. Code Optim.* 16.4 (Oct. 2019). ISSN: 1544-3566. DOI: 10.1145/3356842. URL: https://doi.org/10.1145/3356842.

[176] David B Skillicorn. "A taxonomy for computer architectures." In: *Computer* 21.11 (1988), pp. 46–57.

[177] M Aaron Skinner and Eve C Ostriker. "The Athena astrophysical magnetohydrodynamics code in cylindrical geometry." In: *The Astrophysical Journal Supplement Series* (2010).

[178] Abel Souza, Noman Bashir, Jorge Murillo, Walid Hanafy, Qianlin Liang, David Irwin, and Prashant Shenoy. "Ecovisor: A virtual energy system for carbon-efficient applications." In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 2023, pp. 252–265.

[179] Yacine Taleb, Shadi Ibrahim, Gabriel Antoniu, and Toni Cortes. "Characterizing performance and energy-efficiency of the ramcloud storage system." In: *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2017, pp. 1488–1498.

[180] Peter Thoman, Daniel Gogl, and Thomas Fahringer. "Sylkan: Towards a Vulkan Compute Target Platform for SYCL." In: IWOCL'21. Munich, Germany: Association for Computing Machinery, 2021. ISBN: 9781450390330. DOI: 10.1145/3456669.3456683. URL: https://doi.org/10.1145/3456669.3456683.

[181] Peter Thoman, Fabian Knorr, and Luigi Crisci. "SimSYCL: A SYCL Implementation Targeting Development, Debugging, Simulation and Conformance." In: *Proceedings of the 12th International Workshop on OpenCL and SYCL*. IWOCL '24. Chicago, IL, USA: Association for Computing Machinery, 2024. ISBN: 9798400717901. DOI: 10.1145/3648115.3648136. URL: https://doi.org/10.1145/3648115.3648136.

[182] Peter Thoman, Facundo Molina Heredia, and Thomas Fahringer. "On the Compilation Performance of Current SYCL Implementations." In: *International Workshop on OpenCL*. IWOCL'22. Bristol, United Kingdom, United Kingdom: Association for Computing Machinery, 2022. ISBN: 9781450396585. DOI: 10.1145/3529538.3529548. URL: https://doi.org/10.1145/3529538.3529548.

[183] Peter Thoman, Philip Salzmann, Biagio Cosenza, and Thomas Fahringer. "Celerity: High-level C++ for Accelerator Clusters." In: *Euro-Par 2019: Parallel Processing: 25th International Conference on Parallel and Distributed Computing, Göttingen, Germany, August 26–30, 2019, Proceedings 25*. 2019.

[184] Ananta Tiwari, Michael Laurenzano, Joshua Peraza, Laura Carrington, and Allan Snavely. "Green Queue: Customized Large-Scale Clock Frequency Scaling." In: *2012 Second International Conference on Cloud and Green Computing*. 2012.

[185] Christian R Trott, Damien Lebrun-Grandié, Daniel Arndt, Jan Ciesko, Vinh Dang, Nathan Ellingwood, Rahulkumar Gayatri, Evan Harvey, Daisy S Hollman, Dan Ibanez, et al. "Kokkos 3: Programming model extensions for the exascale era." In: *IEEE Transactions on Parallel and Distributed Systems* 33.4 (2021), pp. 805–817.

[186] Alessandro Tundo, Marco Mobilio, Shashikant Ilager, Ivona Brandić, Ezio Bartocci, and Leonardo Mariani. "An energy-aware approach to design self-adaptive AI-based applications on the edge." In: *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2023, pp. 281–293.

[187] Georgios Tziantzioulis, Nikos Hardavellas, and Simone Campanoni. "Temporal approximate function memoization." In: *IEEE Micro* (2018).

[188] VMWare. *Exploring the GPU Architecture*. `https://www.vmware.com/docs/exploring-the-gpu-architecture`. 2024.

[189] Vassilis Vassiliadis, Charalampos Chalios, Konstantinos Parasyris, Christos D Antonopoulos, Spyros Lalis, Nikolaos Bellas, Hans Vandierendonck, and Dimitrios S Nikolopoulos. "Exploiting significance of computations for energy-constrained approximate computing." In: *International Journal of Parallel Programming* (2016).

[190] Vassilis Vassiliadis, Konstantinos Parasyris, Charalambos Chalios, Christos D Antonopoulos, Spyros Lalis, Nikolaos Bellas, Hans Vandierendonck, and Dimitrios S Nikolopoulos. "A programming model and runtime system for significance-aware energy-efficient computing." In: *ACM SIGPLAN Notices* (2015).

[191] Jeffrey S. Vetter et al. "Extreme Heterogeneity 2018 - Productive Computational Science in the Era of Extreme Heterogeneity: Report for DOE ASCR Workshop on Extreme

Heterogeneity." In: (Dec. 2018). DOI: 10.2172/1473756. URL: https://www.osti.gov/biblio/1473756.

[192] Giulio Vistoli et al. "MEDIATE - Molecular DockIng at homE: Turning collaborative simulations into therapeutic solutions." In: *Expert Opinion on Drug Discovery* (2023).

[193] Emanuele Vitali, Federico Ficarelli, Mauro Bisson, Davide Gadioli, Gianmarco Accordi, Massimiliano Fatica, Andrea R. Beccari, and Gianluca Palermo. "GPU-optimized approaches to molecular docking-based virtual screening in drug discovery: A comparative analysis." In: *Journal of Parallel and Distributed Computing* 186 (2024), p. 104819. ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2023.104819.

[194] Matthew Walker, Sascha Bischoff, Stephan Diestelhorst, Geoff Merrett, and Bashir Al-Hashimi. "Hardware-validated CPU performance and energy modelling." In: *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE. 2018, pp. 44–53.

[195] Cong Wang, Michael Zink, and David Irwin. "Energy-agile design for parallel HPC applications." In: *Sustainable Computing: Informatics and Systems* 19 (2018), pp. 123–134.

[196] Qiang Wang, Xinxin Mei, Hai Liu, Yiu-Wing Leung, Zongpeng Li, and Xiaowen Chu. "Energy-aware non-preemptive task scheduling with deadline constraint in DVFS-enabled heterogeneous clusters." In: *IEEE Transactions on Parallel and Distributed Systems* 33.12 (2022), pp. 4083–4099.

[197] Andrew Waterman, Yunsup Lee, David A Patterson, and Krste Asanovic. "The RISC-V instruction set manual, volume I: User-level ISA, version 2.0." In: *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-54* (2014), p. 4.

[198] Vincent M Weaver, Matt Johnson, Kiran Kasichayanula, James Ralph, Piotr Luszczek, Dan Terpstra, and Shirley Moore. "Measuring energy and power with PAPI." In: *2012 41st international conference on parallel processing workshops*. IEEE. 2012, pp. 262–268.

[199] Allan G. Weber. *The USC-SIPI Image Database*. `http://sipi.usc.edu/database/database.php`. Accessed: August 2018. 2006.

[200] Mark Weiser, Brent Welch, Alan Demers, and Scott Shenker. "Scheduling for reduced CPU energy." In: *Mobile Computing*. Springer, 1994, pp. 449–471.

[201] Gene Wu, Joseph L. Greathouse, Alexander Lyashevsky, Nuwan Jayasena, and Derek Chiou. "GPGPU performance and power estimation using machine learning." In: *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 2015.

[202] XUANTIE-RV. *xuantie-gnu-toolchain*. `https://github.com/XUANTIE-RV/xuantie-gnu-toolchain`. Accessed: 2024-10-30.

[203] Andy B Yoo, Morris A Jette, and Mark Grondona. "SLURM: Simple Linux utility for resource management." In: *Workshop on job scheduling strategies for parallel processing*. Springer. 2003, pp. 44–60.

[204] Elizabeth Yuriev, Jessica Holien, and Paul A. Ramsland. "Improvements, trends, and new ideas in molecular docking: 2012-2013 in review." In: *Journal of Molecular Recognition* 10 (2015).

[205] Hadi Zamani, Laxmi Bhuyan, Jieyang Chen, and Zizhong Chen. "GreenMD: Energy-efficient Matrix Decomposition on Heterogeneous Multi-GPU Systems." In: *ACM Transactions on Parallel Computing* 10.2 (2023), pp. 1–23.

[206] Huazhe Zhang and Henry Hoffmann. "Performance & Energy Tradeoffs for Dependent Distributed Applications Under System-Wide Power Caps." In: *Proceedings of the 47th International Conference on Parallel Processing*. 2018.

[207] Huazhe Zhang and Henry Hoffmann. "PoDD: Power-Capping Dependent Distributed Applications." In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2019.

[208] Amelie Chi Zhou, Tien-Dat Phan, Shadi Ibrahim, and Bingsheng He. "Energy-efficient speculative execution using advanced reservation for heterogeneous clusters." In: *Proceedings of the 47th International Conference on Parallel Processing*. 2018, pp. 1–10.

[209] Udo Ziegler. "The NIRVANA code: Parallel computational MHD with adaptive mesh refinement." In: *Computer Physics Communications* (2008).

[210] Eckart Zitzler, Lothar Thiele, Marco Laumanns, Carlos M Fonseca, and Viviane Grunert Da Fonseca. "Performance assessment of multiobjective optimizers: An analysis and review." In: *IEEE Transactions on evolutionary computation* (2003).