ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

DOTTORATO DI RICERCA IN

DATA SCIENCE AND COMPUTATION

Ciclo 36

Settore Concorsuale: 09/H1 - SISTEMI DI ELABORAZIONE DELLE INFORMAZIONI

Settore Scientifico Disciplinare: ING-INF/05 - SISTEMI DI ELABORAZIONE DELLE INFORMAZIONI

SCALING PERFORMANCE AT THE END OF MOORE'S LAW: A PROGRAMMER'S PERSPECTIVE

Presentata da: Federico Ficarelli

Coordinatore Dottorato

Daniele Bonacorsi

Supervisore

Luca Benini

Co-supervisore

Andrea Bartolini

Esame finale anno 2025

Contents

List of Figures

List of Tables

Acknowledgments

I wish to thank Prof. Luca Benini and Prof. Andrea Bartolini for their endless patience and support.
I thank Prof. Biagio Cosenza and Prof. Tobias Grosser for their feedbacks, suggestions and willingness to review this thesis.
Thanks to all friends at The University of Edimburgh and Cambridge University, especially Sasha Lopoukhine, Chris Vasiladiotis and Anton Lydike.
Thanks to all colleagues at CINECA for bearing with me, especially Dr. Chiara Latini.
Thanks to Prof. Giuseppe Tagliavini for his support and guidance, and thanks to all the rest of very fine folks at the Energy-Efficient Embedded Systems Laboratory (a.k.a. *The Dungeon*) at Università di Bologna.
Thanks to Simone Manoni for his help in deciphering weird wave forms from weird processors.
Thanks to Panagiota Dimopoulou and Prof. Daniele Bonacorsi for their invaluable help in navigating stormy waters.

This is dedicated to my family, newcomers included.

Abstract

Computer architectures face a fundamental shift as Moore's law and Dennard scaling reach their technological limits. This evolution has sparked a *Cambrian explosion* of specialized hardware designs trying to keep scaling systems sustainable: *computing is now a power-bound challenge*. While applications still struggle to scale on current exascale systems, future high-performance computing (HPC) systems must integrate an increasingly diverse spectrum of host processors and accelerators, while software stacks must adapt to heterogeneous and application-specific platforms. The need for domain-specific features in hardware designs is driving the advent of the RISC-V architecture: by enabling seamless extension of its widely supported ISA, it could be the answer to the evolutionary challenges that computing systems are facing. While already deployed in industry, the adoption of RISC-V in HPC is still uncharted territory: its role in the future of HPC brings additional challenges for large-scale system-integration and software stacks. This thesis focuses on three ideas. Embarrassingly parallel, task-based workloads must explore throughput-optimized GPU kernel designs to unlock extreme-scale drug discovery campaigns on current TOP500 accelerated systems. On the other hand, HPC systems must overcome design and integration challenges to prepare for future post-exascale clusters that will be increasingly diverse and application-specific, where RISC-V could be an answer. At the same time, the hardware/software interface must adapt. While higher levels of the software stack are adopting novel programming languages and paradigms, target-specific components of the compilation stack must evolve to make domain-specific code generation sustainable for future computing systems. The first part of this thesis involves implementing and scaling drug discovery simulations on multiple heterogeneous, GPU-accelerated TOP500 systems, focusing on efficient acceleration of task-based workloads that scale to trillions of concurrent tasks. The second part centers on designing, building, and evaluating *Monte Cimone*, the world's first RISC-V HPC production cluster, including a comprehensive experimental evaluation and full-scale benchmarks. The third part focuses on the endeavor of developing an MLIR-based compiler backend for Snitch, a novel, energy-efficient RISC-V streaming accelerator for machine learning. The presented work enabled the largest drug discovery simulation for SARS-CoV-2 research ever performed,

demonstrating the practical impact of efficient GPU acceleration techniques needed on present-day accelerated supercomputers. Moreover, the successful deployment of the *Monte Cimone* cluster prototype has proven the production readiness of RISC-V for HPC, paving the way for future RISC-V supercomputers. On the software side, this thesis covers the collective work that extended the multi-level, progressive lowering approach to the compiler backend, enabling efficient micro-kernel code generation for application-specific RISC-V accelerators.

Preface

Computer architectures are currently facing the problem of delivering steadily growing amounts of compute power at ever-increasing efficiency levels to face the new challenges posed by the *end of Moore's law* [1, 2, 3, 4]: this goal is crucial to keep operating large-scale supercomputers [5, 6, 7, 8] feasible. The slowdown of Dennard scaling [9] and the emergence of the effects related to dark silicon [10, 11] add further complexity to the scenario. The profound effects of this disruption are becoming progressively more apparent, to the point that delays in the deployment of TOP500 [12] exascale systems are now a reality to be dealt with [13]. At the same time, high-performance computing (HPC) and low-power devices' goals and constraints are becoming increasingly overlapped under the pressure of the machine learning (ML) market driver: *computing at scale has become a power-bound problem* [14]. Computing architectures historically relying on multi-core, superscalar, single instruction, multiple data (SIMD) CPUs, are now augmented with a diverse spectrum of application-specific accelerators to achieve higher efficiency levels at the cost of decreased generality. The computing market is experiencing a *Cambrian explosion* of vertical hardware designs spanning from tensor processors [15] to in-memory accelerators [16], from massively-parallel specialized architectures [17, 18, 19] to graphics processing units (GPUs) augmented with application-specific cores [20]. This growing diversity of domain-specific accelerators spans all scales, from mobile devices to data centers. Managing this diversity constitutes a global challenge across the HPC stack and is particularly relevant in the field of ML.

Albeit being focused on running linear algebra kernels at scale, accelerator architectures are radically diverse in their underlying design principles, a diversity that directly impacts the software ecosystem. From the point of view of traditional, large-scale scientific applications, developed and optimized to scale up to entire TOP500 [12] clusters, this diversity is posing challenges. Despite their rapid growth in HPC installations, GPU accelerators still face adoption barriers due to many scientific workloads not being designed to exploit accelerated computing capabilities [21]. HPC sites surveys reveal how most GPU-accelerated jobs tend to have low utilization of available accelerator resources [22]: adapting algorithms and porting large code bases is still an ongoing effort. Moreover, task-based, embarrassingly-

parallel workloads are at the core of mainstream scientific applications like molecular docking for drug discovery [23, 24], astronomical image processing [25], Monte Carlo methods [26] and genomic sequence alignment [27]. They pose unique challenges as dealing with multi-level load balancing, task distribution and design of throughput-oriented GPU kernels for such workloads is an open research question:

> How task-based, embarrassingly-parallel scientific workloads like virtual screening can efficiently scale up to entire pre-exascale, state-of-the-art GPU-accelerated HPC systems?

From the point of view of artificial intelligence (AI) applications, despite growing investments, the *accelerator diversity challenge* usually translates into the need for efficient implementations of a wide variety of specialized kernels for each hardware platform: while deep neural network (DNN) frameworks try their best to leverage existing vendor libraries, large corpora of hand-optimized, vendor-specific operators written by experts are becoming a reality to be dealt with.

Compilers are increasingly being adopted as a possible response to these challenges as they can unlock both efficient utilization of existing kernel libraries or direct generation of high-performance accelerator-specific code. This approach fuels a diverse spectrum of compilation techniques: from polyhedral (i.e. Tensor Comprehensions [28]) to loop synthesis (i.e. Halide [29], TVM [30] and PlaidML [31]), from tensor-based intermediate representations (IRs) (i.e. XLA [32] and Glow [33]) to tile-based approaches (i.e. Triton [34]). IRs are the languages used by compilers to internally represent, analyze and transform a program. Traditional IRs based on a single, uniform level of abstraction proved themselves to be solid foundations for compilers that focused on relatively low-level frontend languages (i.e., C, C++) [35, 36]. Since programs are now being written in high-level languages where tensor algebra is a first-class citizen [37, 38], compilation stacks are transitioning from the traditional low-level, fixed IRs to expressive, flexible representations. While a definitive answer to the question of *"what a post-Moore software stack would look like?"* remains unknown, all novel compiler construction approaches share the use of *multi-level IRs*.

Multi-level IRs allow constructs at multiple levels of abstraction (e.g., both an immutable tensor and a memory address) to coexist in the same program. This single feature alone enables *progressive lowering*, allowing the compiler to preserve semantic information until the optimal transformation can be applied, such that a non-reversible expense of semantic information brings the most value to the lowering result. Avoiding any *semantic loss* during the lowering process is proving itself as an effective approach [39, 40, 33].

While being investigated and implemented in production compilers in the past [41, 42], the static single assignment (SSA), multi-level IR concept is now further expanded and successfully brought into production. A notable example is MLIR (*Multi-Level IR*) [39], a new compiler infrastructure built on top of a flexible and extensible IR that recently became the foundation for several novel deep learning (DL) and domain-specific language (DSL) toolchains. By aiming at making abstractions and transformations modular and interoperable, MLIR is proving itself as a key tool to tackle the challenges of the *post-Moore* compute era.

Along with compilers and software stacks, the challenge of extreme specialization is affecting the landscape of hardware architectures. In order to satisfy the demand for efficient linear algebra computations coming from the HPC and ML markets, industry-standard instruction set architectures (ISAs) are getting extended with domain-specific features [43, 44, 45]. While already well established, dominant ISAs (i.e., x86, ARM) are proving themselves not flexible enough for the scaling challenges posed by the ML market: being closed for modification, they can be neither extended nor adapted, forcing third parties and research institutions into looking for alternatives. The need for a more flexible solution for computing innovation is proven by the rise of RISC-V, an open, modular, extensible, and royalty-free ISA. Being designed from scratch to be naturally extensible without breaking the rich existing software ecosystem, RISC-V has become the platform of choice for architecture research [46, 47, 48, 49] and the market enabler for an increasing set of vendors who are exploring novel concepts in accelerator and processor design [50, 51, 52, 53, 54, 55]. Albeit rapidly growing, the extreme flexibility of an open-ended ISA poses unique challenges both to software implementors and system integrators. While RISC-V is believed to have a bright future in the data center [56], its extreme flexibility poses new questions still to be addressed:

> How an extreme-scale HPC system, based on a heterogeneous offering of highly specialized accelerators, can be sustainable?

> If RISC-V can be an answer to the challenge of extreme hardware specialization, how future post-exascale systems based on RISC-V will look like?

On the software side, this increased flexibility is stressing traditional compiler designs to their limits. With its flexible ISA intended to be extended, RISC-V becomes a modular compilation target forcing compiler backends to deal with a virtually unbounded number of optional, cross-interacting ISA extensions and features. This issue raises new questions that have yet to be addressed:

Can a multi-level IR represent RISC-V domain specific extensions for novel linear algebra accelerators?

Can the multi-level approach to compiler construction enable SSA compiler backends to generate high-performance kernels leveraging custom, application-specific hardware features?

This thesis covers collective works that tried to answer the questions posed by our *post-Moore* era. A description of each chapter, along with references to relevant publications and my contributions to each work are detailed in the remainder of this preface.

The Present of Sustainable HPC: GPU-Accelerated Systems. Chapter 1 considers today's TOP500 HPC systems that have become heterogeneous, GPU-accelerated systems. While the execution of large-scale, tightly-coupled workloads is a well-understood problem [57, 58], many significant HPC applications like *drug discovery* still rely on less explored task-based, data-parallel workloads. This chapter presents how algorithms that need to scale up to trillions of embarrassingly-parallel tasks can be efficiently executed on pre-exascale HPC systems accelerated with GPUs [59]. Considering a real-world, production use case stemming from *virtual screening*, the work explores different approaches to efficiently map such workloads on GPU accelerators and how they must be treated differently from common tightly-coupled workloads, and how warp-synchronous kernels and load balancing are involved. The experimental evaluation reports results obtained by integrating such approaches in a production-level HPC application that, by scaling up to two full pre-exascale systems, is able to screen trillions of potential drug molecules and helped identify prospect candidates for being active against SARS-CoV-2 virus replication. This experiment remains the largest drug discovery simulation to date [24, 60].

Contributions to Chapter 1. The work presented in this chapter stems from a long-standing partnership between CINECA and Dompé Farmaceutici S.p.A. for the scientific and technological development of the LiGen [61] HPC drug discovery platform. This collaborative effort resulted in several publications.

In particular, Vitali et al. [59] is a collaboration between CINECA, Dompé Farmaceutici S.p.A., the research group led by Prof. Gianluca Palermo at Politecnico di Milano and NVIDIA. It covers a novel approach to embarrassingly-parallel, task-based workloads on NVIDIA GPUs that resulted in the first port of the LiGen platform to GPU-accelerated HPC systems. My contributions to this work concern the exploration of both the batch and latency workload execution strategies, development and optimization of GPU kernels relative to the estimation of protein-compound binding energy (*chemical scoring*) according to both strategies, benchmarking, performance

modeling and micro-architectural profiling, statistical analysis of results, and overall development of the LiGen platform.

Gadioli et al. [24] presents a description of the overall drug discovery platform and how the *one-trillion-docking experiment* against SARS-CoV-2 protein targets was run after the EXSCALATE4CoV European project called for an *urgent computing* action during the global pandemics. This work brings the advancements presented in Vitali et al. [59] at scale in an HPC production setting, where my contribution concerned the planning and execution of the experiment, data pre- and post-processing and statistical analysis.

Moreover, in Emerson et al. [60] I present my contributions on securing and managing the HPC resources used during the *one-trillion-docking experiment*.

Finally, Vistoli et al. [62] presents the polypharmacology results from the EXSCALATE4CoV and the MEDIATE initiative, calling for a collaborative, open-access drug discovery effort against pandemic emergencies. My contributions to this work concern polypharmacology analysis, the *big data* post-processing pipeline to compute statistical descriptors and the overall analysis of data produced during the *one-trillion-docking experiment* [24].

The Future of Sustainable HPC: RISC-V. Chapter 2 presents *Monte Cimone* [63], the world's first HPC production cluster based on the RISC-V architecture. *Monte Cimone* was designed and built with the purpose of *priming the pipe* and exploring the challenges of integrating a multi-node RISC-V HPC cluster. Being composed of a small number of nodes, it doesn't aim at achieving strong floating point cluster-wise performance. On the other hand, to be able to explore application readiness, system integration, deployment and energy efficiency, the cluster must capable of providing a complete HPC production stack including interconnect, storage and power monitoring. For this goal, HPC scientific applications must be taken into account: a platform's readiness is evaluated by widespread scientific community applications that are large, complex, and highly optimized code bases that must efficiently scale up to the full system. The results of the hardware/software integration efforts demonstrate a remarkable level of software and hardware readiness and maturity, showing that the first generation of RISC-V HPC machines may not be so far in the future.

Contributions to Chapter 2. The work presented in this chapter stems from a collaborative effort between both the research groups led by Prof. Luca Benini and Prof. Andrea Bartolini at Università di Bologna, CINECA and E4 Computer Engineering. My contributions concern the overall system's design, the bring-up of the HPC *user space*, evaluation of the RISC-V software stack including compiler toolchains, MPI implementations, linear algebra libraries, and both the methodology and execution of full-system benchmarks of scientific applications like quantumESPRESSO [64].

This work resulted in several publications, in particular Ficarelli et al. [65,

66] and Bartolini et al. [63], all covering the collaborative effort needed to design, deploy, and evaluate a RISC-V HPC cluster from-scratch.

Multi-level SSA Compilers for RISC-V Accelerators. In Chapter 3 I present the work done by the research group led by Prof. Tobias Grosser at the University of Edimburgh (now at Cambridge University) and my contributions while visiting. The work investigates how modern compiler approaches, based on multi-level SSA IRs, can be leveraged to build a kernel compiler for novel streaming architectures that, by being extremely energy-efficient, can become an answer to the future of large-scale systems. The concept of *progressive lowering* is applied to the compiler backend to investigate how traditional code generation tasks can be efficiently performed on a multi-level SSA IR like MLIR. The selected compilation target is Snitch [48], a RISC-V accelerator architecture designed by the research group led by Prof. Luca Benini at ETH Zurich. Its design goal is to pursue extreme compute energy efficiency by means of novel features like stream-semantics registers to reach perfect, software-programmed prefetching, and floating point hardware loops to elide control flow. The resulting publication by Lopoukhine et al. [67] presents how a new tensor kernel compiler, completely based on MLIR [39] for both optimization and code generation, can overcome compilation challenges by progressively lowering high-level linear algebra programs to leverage Snitch features. Evaluation of the generated code is supported by both the definition of a theoretical peak-performance model for the target micro-architecture and a comprehensive, reproducible simulation environment based on precise execution traces. The reference benchmark set is designed to take into account meaningful linear algebra micro-kernels that are both difficult to lower efficiently on Snitch (i.e. mixing reduction and element-wise operations) and widespread in both HPC and DL applications. Results from an extensive experimental campaign prove that this approach can lower high-level linear algebra code down to kernels capable of surpassing performance attained by hand-written, expert-optimized kernels.

Contributions to Chapter 3. My contributions to the effort presented in Lopoukhine et al. [67] concern several aspects of the MLIR dialects, transformations, and the experimental evaluation of the resulting kernel compiler.

The `snitch` dialect encodes the target-specific operations needed to set up and operate the Snitch data movers. The `memref_stream.generic` operation extends the `linalg.generic` operation from upstream MLIR: it augments the information carried by the operation (affine memory access expressions and data dependencies) with additional information needed by the compilation pipeline to generate optimized RISC-V assembly code exploiting both streaming semantics and hardware loops.

Due to architectural constraints, when dealing with floating-point values of precision lower than double, instructions in stream-accelerated regions have to operate on vectors instead of scalars. Thus, from the point of view of a compiler backend, elements smaller than double precision must be con-

sidered *illegal* in code sections where Snitch-specific features are leveraged to accelerate computations. For this reason, the legalization pass transforming `memref_stream.generic` operations also performs mandatory packed-SIMD vectorization.

Another area of contribution is the experimental evaluation, testing, benchmarking, and performance modeling. Building a kernel compiler requires continuous feedback both in terms of correctness and performance of the generated code, so the adopted approach introduced a deterministic test and benchmarking harness that is then used in a continuous integration infrastructure to provide feedback on every change introduced in the code generation pipeline. The Snitch Verilog source repository provides utilities to decode simulation traces: post-processing the cycle-accurate execution traces allowed to compute all the performance metrics needed to support our benchmarking model without any hardware support (like an actual *performance monitoring unit*). A kernel templating system supports the parameter space exploration needed to guide both the implementation and tuning of optimization passes. While not directly implemented in MLIR but instead as an external tool, our kernel generator works similarly to the `mlir-gen` generator by Golin et al. [68] in that it instantiates variants of kernel programs based on parametric input tensor shapes and, at the same time, provides test input data along with reference results to ensure correctness.

Chapter 1

The Present of Sustainable HPC: GPU-Accelerated Systems

Since its dawn, the technological evolution of the semiconductor industry has been driven by two laws: *Moore's law* and *Dennard scaling*. While presenting several economic and societal corollaries, the technological formulation of *Moore's law* [69] stated that *the number of components per integrated circuit doubles every year*, an empirical observation that, while adjusted [70] since its former inception in 1965, has been generally accepted as a *self-fulfilling prophecy*. *Dennard scaling* [71] on the other hand, focuses on the relation between integration scale and power and roughly states that, *as transistors get smaller, their power density stays constant, so that the power use stays in proportion with area*. The combination of these two laws has driven the semiconductor industry's evolution since the 1970s.

During mid-2000s, the semiconductor market started experiencing *slow-downs* of the laws discussed above, to the point that the end of both Dennard scaling [72] and Moore's law [1, 2, 3, 4] became major discussion topics among professionals and researchers and, subsequently, concrete factors to be taken into account. A direct consequence of this technological disruption is that energy efficiency has become paramount for HPC and large-scale systems overall sustainability [73, 74]. Moreover, it is now widely recognized that *computing is power bound* [14], and some research claims that computing energy efficiency must obtain improvements by up to ten orders of magnitude to be able to solve major scientific problems [75].

A historically significant gauge for technological trends in computing architectures is the HPC market. Given its extreme scaling needs, all components of a computing system must be sustainable to be able to deploy and maintain an HPC system successfully in operation. As soon as the first effects of the slowdown of the *semiconductor laws* became apparent, the HPC

market was immediately affected. Figure 1.1 shows how technological dis-

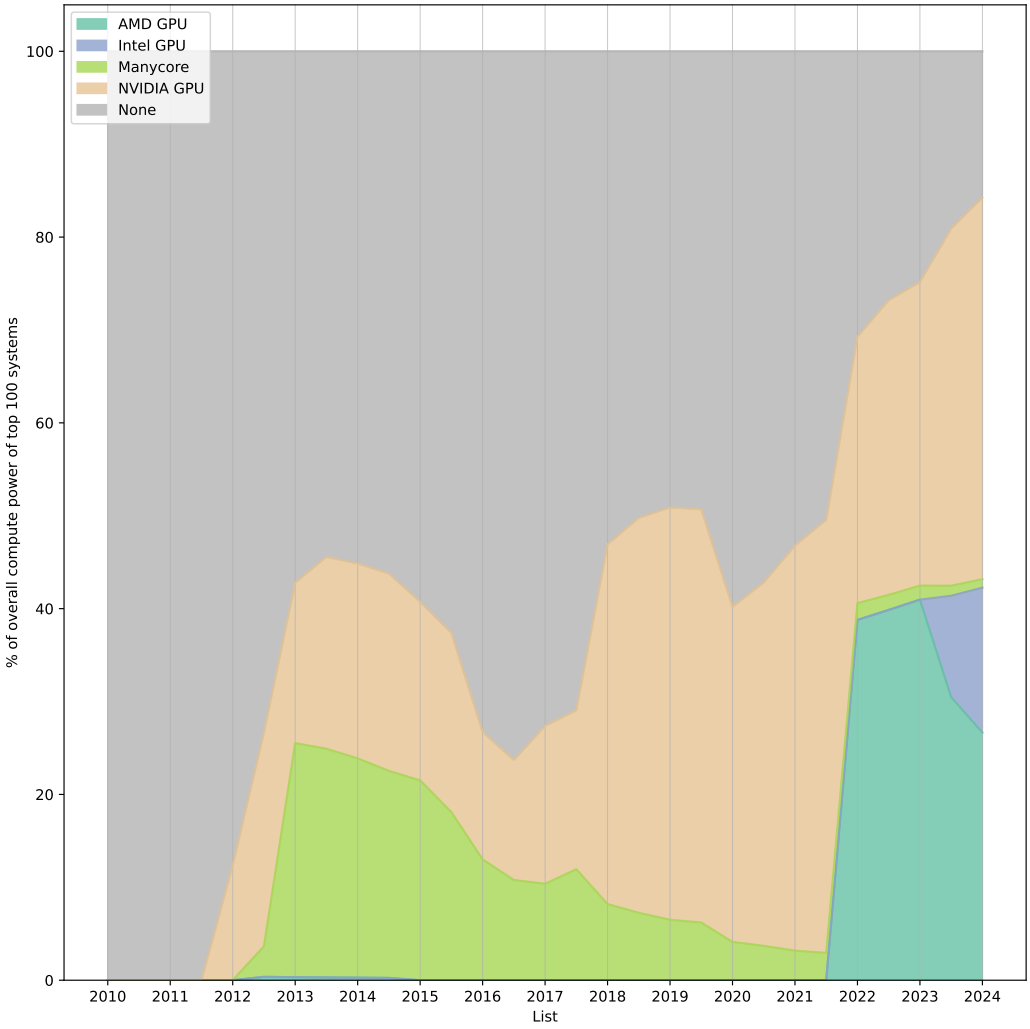


Figure 1.1: Evolution of the share of compute power per accelerator across the first 100 positions of TOP500 supercomputer rankings since 2010.

ruptions affected the largest supercomputer installations across the globe: both the *multicore era* and the rise of accelerated computing are visible in the historical evolution of the TOP500 ranking. Starting from 2010s accelerators have become a key factor for computing efficiency and essential enablers for systems scale-out [76]. As shown by historical data, after the *multicore era*, TOP500 systems began integrating GPU accelerators at an increasing rate to keep up with computing demands: the evolution of large-scale systems has been driven by the ability of GPUs to offer energy-efficient massive parallelism.

At the time of their introduction in the late 1990s, GPUs were specialized, fixed-function accelerators for graphics applications like video games or scientific datasets visualizations [77]. The adoption of *programmable unified shaders* by NVIDIA in 2006 and, more importantly, the introduction of CUDA [78] as the programming model for general-purpose computing on graphics processing units (GPGPU), allowed GPUs to be gradually adopted in HPC to accelerate an increasingly diverse spectrum of scientific workloads. Lastly, the introduction of double precision floating-point arithmetic in 2010 (with the NVIDIA Fermi architecture) led to Titan, the supercomputer operated by Oak Ridge National Laboratories and powered by NVIDIA GPUs, being ranked as the world's most powerful supercomputer in 2012. Since its inception, the concept of GPGPU has been applied to all scientific computing fields traditionally reliant on HPC resources to solve increasingly hard problems, and the importance of the single instruction, multiple thread (SIMT) programming model introduced by CUDA is widely recognized as the key enabling factor to leverage the new massively parallel architecture [79].

Starting from this accelerator-centric perspective, HPC applications must provide efficient GPU-enabled implementations of their key algorithms. Large-scale, tightly coupled (or *synchronous*) workloads are mapped to GPUs by dividing the problem into smaller parallel tasks that can be executed simultaneously [80]. Examples of such workloads vary from linear algebra on large tensors to solving differential systems on large domains [81]. In this kind of workload the challenge shifts to parallel communication strategies instead of computation, and load imbalance between tasks is usually small and limited to boundaries. Massively parallel, throughput-oriented GPUs are the ideal platform [82].

Heterogeneous task-based workloads, on the other hand, present challenges when mapped to GPUs due to potential load imbalance between threads within warps and between warps in the same kernel. Previous work on this class of algorithms focuses on loosely-coupled workloads, where data dependencies between tasks (usually modeled as a directed acyclic graph) are the major limiting factor for execution parallelism. Current approaches focus on efficient scheduling of the task graph by means of runtime support [83, 84, 85, 86, 87] either via task stealing on work queues [88] or even sophisticated thread coalescing based on their execution path [89].

When dealing with heterogeneous *embarrassingly parallel* workloads, on the other hand, runtime approaches become less effective due to the workload becoming heavily *throughput-oriented*: individual task's execution time is less important than overall system efficiency, and optimal occupancy of the available hardware resources becomes paramount. This chapter presents a novel approach to heterogeneous, embarrassingly parallel workloads based on warp-synchronous [90] GPU kernels and applied to extreme-scale drug discovery on pre-exascale HPC systems.

1.1 Use Case: Exascale Drug Discovery

In recent years, high-throughput virtual screening has been widely applied in the early stages of the industrial computer-aided drug discovery process. Indeed, this helped find novel drugs [91, 92, 93]. Several steps are required to perform a virtual screening campaign [94]; however, the focus is on the molecular docking step.

Many software tools have been created for this goal, both open [95, 96] and closed [97, 98] source. There are two main approaches to the molecular docking step: the first is deterministic, while the second favors a random-based approach. Random-based approaches use well-known techniques to create different poses of a *ligand* and measure their interactions with the protein *docking sites*. Examples of these are MolDock [99] and Gold [100] where genetic algorithms are used, or Glide [97] and MCDock [101] where the technique used is Monte Carlo simulation. However, this approach has a significant drawback since its results may only be partially reproducible. This drawback may be a blocking issue for some pharmaceutical companies that refuse to start the expensive in-vitro and in-vivo phases without a reproducible result. For this reason, a deterministic approach is often a strict requirement. Examples of deterministic approaches are BIGGER [102], DOCK [95], LiGen [61], and Flexx [98]. These approaches use deterministic algorithms that can modify the shape of the ligands by leveraging their torsional bonds. Many molecular docking applications were born as single workstation applications; however, given the amount of complex elaboration that has to be performed, they quickly evolved into HPC applications. Surveys [103] highlighted that different techniques are currently being studied to improve the capabilities of widespread software tools and scale them to HPC machines. The prominent approaches adopt either scaling via MPI [104] or manually distributing data via intermediate files via ad-hoc solutions [105]. Given the general availability of GPU accelerators, though, molecular docking applications are in the process of being ported to exploit these co-processors [106, 107, 108, 109, 110, 23]. Early examples are MedusaDock [106], achieving a $1.54\times$ overall speedup on time-to-solution, and GeauxDock [109] achieving a $3.5\times$ speedup. More recent examples obtaining much larger advantages are PIPER [107] ($17\times$), AutoDock-Vina [23] ($50\times$) and PLANT [108] ($60\times$). The latest GPU porting of AutoDock [111, 112, 113] has been optimized for running on the Summit supercomputer [114] to support COVID-19-related research. A new Autodock development was recently released: Uni-Dock [115]. Uni-Dock increases the accuracy compared to the Autodock and VINA GPU versions, making the execution ten times faster thanks to batching. Uni-Dock tries to use heuristics, based on the type of architecture, to execute a batch of inputs likely to fill the entire memory of the GPU. Other approaches have improved performance by using dedicated hardware for matrix computation. For example, Autodock

Algorithm 1: LiGen virtual screening algorithm

```
Data: max_num_ligands
Input: ligand_library, target
Output: top_candidates
candidates ← ∅;
foreach ligand ← ligand_library do
    candidates ← candidates ∪ dock(ligand, target);
end
return top_n(candidates, max_num_ligands)
```

has been accelerated using NVIDIA tensor cores [116]. Using this approach, they have achieved a $4\times$ to $7\times$ speedup in reduction operations, with an overall reduction of 27 % in docking time. LiGen is an HPC application that implements an integrated workflow for extreme-scale pharmaceutical virtual screening; it has been designed from scratch to scale-out on TOP500 systems and distributes the workload across computing nodes via MPI [117]. It is both an industrial application, being currently used in production by Dompé Farmaceutici S.p.A, and a notable example of embarrassingly-parallel, throughput-oriented workload in HPC. LiGen has been used to perform the largest virtual screening campaign ever run ($>$ 70 billion ligands and 12 viral proteins) during the COVID-19 pandemic outbreak [24] (see Section 1.7). In this chapter, we focus on the efficient GPU porting of LiGen by describing and analyzing two different parallelization approaches considering the peculiarities of the target workload and GPU devices.

1.2 The Virtual Screening Application

LiGen [61] is a molecular docking application designed to run on HPC systems and adapted for extreme-scale virtual screening campaigns [24]. In this chapter, we focus on the docking kernel because it is the most demanding regarding hardware requirements and computation effort, accounting for 90 % of the execution time.

Algorithm 1 presents the pseudo-code for virtual screening of an input ligand library against a target docking site. The output lists the ligands with the highest interaction strength with the target. The procedure is straightforward; we must dock each ligand from the input library to estimate its interaction strength using a scoring function. When we have more than one docking site, repeating the entire procedure for another target is possible, generating a different set of best candidates. Domain experts will then combine the data to select a global set of candidates to test *in-vitro* (or further *in-silico*). Thus, we can focus on the single-target scenario without losing generality.

Algorithm 2: LiGen dock algorithm

```
   Data: num_restart
   Input: ligand, target
   Output: best_pose
 1 poses ← ∅;
 2 for i ← 0 to num_restart do
 3 |   pose ← init_pose(ligand, i);
 4 |   pose ← align(pose, target);
 5 |   pose ← optimize(pose, target);
 6 |   pose.validity ← is_valid(pose, target);
 7 |   if pose.validity then
 8 |   |   pose.score ← score(pose.atoms, target);
 9 |   else
10 |   |   pose.score ← −∞;
11 |   end
12 |   poses ← poses ∪ {pose};
13 end
14 return max_score(poses)
```

Algorithm 2 describes in more detail all the steps LiGen uses to dock a ligand inside a target. The overall algorithm is a gradient descent with multiple restarts [118]. At each restart, we generate an initial pose (line 3) by rotating the ligand's rotamers using a heuristic that maximizes the distance among the iterations in the conformation space of the molecule. The gradient descent procedure is composed of two operations. The first considers the molecule to be a rigid body to align with the docking site (line 4). In contrast, the second uses the internal molecule flexibility to optimize its shape for the target and performs a local minimization (line 5). We use a geometric score to define the gradient that drives the docking. To select the most suitable pose, we need to re-score the poses using a scoring function that considers physical and chemical properties (line 8). To avoid useless computation, we do not compute the scores of molecules (line 10) that clash internally or with the protein (lines 6,7). Finally, among all the ligand's poses, we are interested only in the one that yields the highest score (line 14). From the algorithm description, we can notice how a pose evaluation is independent of the others. We can generate many ligands by simulating known chemical reactions, making the virtual screening problem embarrassingly parallel. LiGen uses these properties to distribute the input ligand library across different nodes [24] and to offload the computation to GPUs. In this chapter, we explore two strategies to implement the algorithm in CUDA that use drastically different design choices to leverage hardware parallelism.

1.3 Architecture of GPU Accelerators

While CPUs are *latency*-optimized processors, GPUs are *throughput*-optimized processors. On NVIDIA architectures, the underlying design pillar that embodies this principle is *reducing non-compute silicon* as much as possible, resulting in large arithmetic logic units (ALUs) and simplified, shared execution control units. A GPU accelerator is subdivided into multiple streaming multiprocessors (SMs), and each SM is again subdivided into $4\times$ streaming multiprocessor sub-partitions (SMSPs): the SMSP is the actual execution unit.

In CUDA, *threads* are arranged hierarchically, whereby they are grouped into *blocks* and then launched in *grids* [119]. These *threads* and *blocks* may be grouped in a three-dimensional structure, allowing for efficient organization and execution of instructions.

The most immediate consequence of the simplified execution control is *lockstep execution* and the associated SIMT architecture. SIMT architecture is similar to SIMD in that one instruction controls multiple processing elements. The main difference is that SIMD exposes its width to software, while SIMT handles the execution and branching of individual threads. Unlike SIMD, SIMT allows programmers to write both independent, scalar thread code and data-parallel code for synchronized threads. NVIDIA GPUs group threads of execution in units of 32 threads [119] called *warps*: each thread in the same warp executes the same instruction a the same time. The CUDA programming model supports conditional branches, though: when threads of the same warp take different paths from the same conditional branch, the effects of *threads divergence* become apparent as a non-negligible performance loss. It is worth noting that allowing divergent threads on a SIMT architecture is a collaborative effort between the compiler and the GPU hardware. For each conditional branch on the control flow graph (CFG) of the input kernel, the CUDA compiler generates a *diverge* instruction; at the same time, it identifies the CFG node that *post-dominates* the branch and generates a *reconverge* instruction. Those instructions manipulate a *divergence stack* maintained by the GPU hardware: a *diverge* instruction pushes information about the threads that must be active while executing the predicated control flow; a *reconverge* pops from the same stack, closing the current top-level conditional branch. If both paths of a conditional branch have active threads, both must be executed *one at a time*: although SIMT behavior can generally be ignored for correctness, avoiding thread divergence within warps can significantly improve performance.

As with threads, GPU memory is also hierarchically organized. There are 3 different memory levels in an NVIDIA accelerator:

- Global memory: the slowest (2 TiB/s of bandwidth on A100) and largest memory (up to 80 GB on A100) available on the accelerator. It is shared

by all threads running in the kernel grid.

- Shared memory: a small, low-latency, software-managed, high-bandwidth memory shared among the threads of a block, and accessible by all of them.

- Registers: the fastest memory space available to threads. Each SM has 65536 32-bit registers that are partitioned among the threads of the resident blocks. Along with the amount of shared memory, the number of registers required per thread determines how many blocks can concurrently run on an SM (also known as *occupancy*).

It is paramount that all levels of the memory hierarchy are used optimally: global memory must be accessed following specific patterns only (i.e., regularly strided), and shared memory is usually leveraged as a software-managed cache. Due to the complete absence of speculative features (e.g., prefetchers) and the sheer amount of load/store executed by a large number of threads, spatial and temporal locality of memory accesses are paramount on a GPU.

1.4 Latency-Optimized Kernels for Task-Based Workloads

The first implementation we will analyze is the *latency implementation*. This approach aims to keep a synchronous interface, where a single *ligand* is docked on the GPU in every host call to the dock function. This approach is the same as the previous implementation of LiGen [120, 24] and allows us to focus only on the acceleration of the kernels without having to modify the whole application structure, thus purely following Algorithm 1 for screening a ligand library. On the GPU, we distribute the operation that we have to perform as much as possible, trying to make each kernel as parallel (and fast) as possible (Figure 1.2). This approach is the most straightforward and traditional one, and it is the same followed by most of the GPU porting for molecular docking (e.g., AutoDock-GPU [111, 112]). Figure 1.3 provides an overview of the approach regarding the main parallelism exploited and execution phases. The idea is to execute all the iterations for the loop at line 3 of Algorithm 2 in parallel. To reach this goal, we must perform each step of the computation on all the poses as shown in Figure 1.3. The CUDA grid for each kernel launch is set to have enough execution resources to cover the different ligand poses. We implemented all the steps using at least one kernel to increase the exposed parallelism. In this way, we can execute in parallel the internal loops required to carry out the computation. In particular, for the *init_pose* step, each CUDA thread updates the position of a single atom. For the *align* step, we use two kernels. The first one evaluates all the rigid transformations for all the poses in parallel, where each CUDA

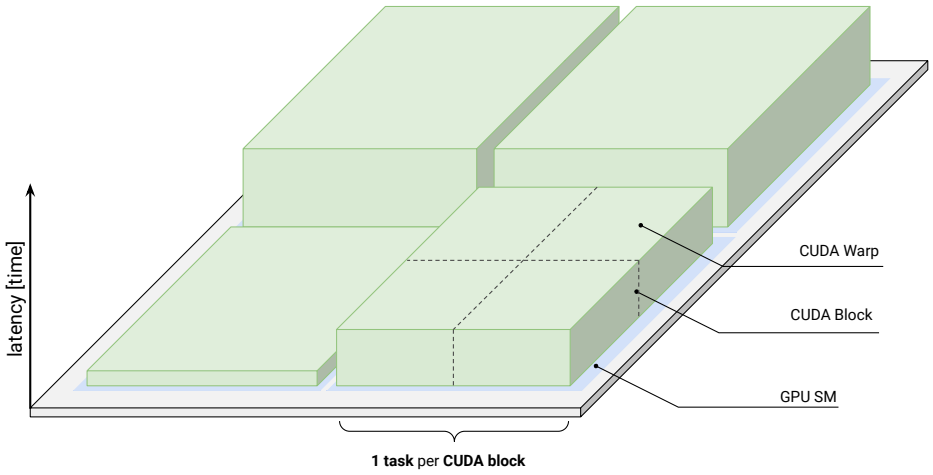


Figure 1.2: Task-to-resources mapping for the latency-oriented strategy on the GPU. Execution time (x axis) is kept optimal by allocating the largest possible resources to each task. Multiple warps process a task, and each task consumes all the resources available on a GPU streaming multiprocessor (SM).

thread updates the displacement atoms and computes the gradient value. The second kernel performs a reduction to find the optimal alignment for the ligand and updates the displacement of the atoms accordingly. In the *optimize* step, we need to evaluate each rotamer sequentially to preserve the ligand geometry. We use two kernels to evaluate a single rotamer using an approach similar to the *align* step. Besides rotating and computing the gradient value, the main difference is that each CUDA thread needs to check if the rotamer's angle leads to an internal clash. For the *is_valid* step, we use two different kernels to check whether there is a clash with the protein or an internal one. In both cases, the distance between each atom pair has to be calculated. An *early exit* is performed when we detect a bump between atoms, a condition that would lead to an invalid pose.

By computing poses using the parallelism at the grid level, these kernels aim at minimizing the execution time of each task, hence the *latency* name. As depicted in Figure 1.2, each task is assigned the largest possible amount of execution resources to run in the shortest time possible, usually a whole SM.

To maximize the GPU utilization, we rely on a multi-threaded approach to instantiate several different kernels on different CUDA streams. Every *ligand* will be tied to a host thread controlling an asynchronous execution queue (CUDA stream) and a dedicated, pre-allocated GPU global memory area. This first optimization saves memory operations since this memory space is not linked to the docking of a single ligand but is tied to the application's lifetime. The drawback of this approach is that we need to allocate

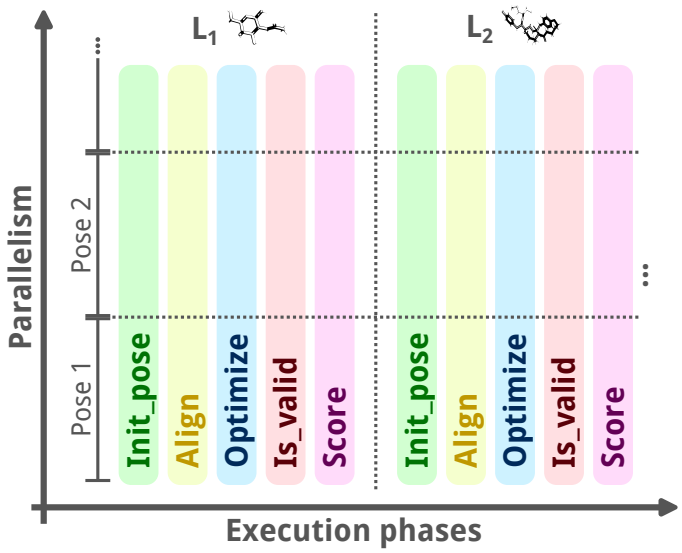


Figure 1.3: Logical mapping on how the latency approach hinges on GPU parallelism to accelerate the execution time. Each step is implemented using at least one dedicated kernel.

the *worst case* space, which must be known at compile time. However, this is not a critical issue since, for each virtual screening campaign, the maximum size of a molecular graph is known and an important chemical parameter taken into account by the pharmaceutical scientists. Moreover, some data structures (such as the one that represents the target pocket) can be shared among all the threads using the same GPU: this is possible since they are read-only data structures.

The access to the pocket does not follow a coalesced pattern; instead, it is given by the atom's cartesian coordinates and, for this reason, has a random pattern. Random accesses are a costly operation in GPU since they prevent efficient transactions to/from the global memory. However, GPU texture caches allow organizing data in 2D or 3D spaces and are optimized for semantic data locality: locality becomes n-dimensional in space. Rotations and translations in three-dimensional space will always interact with local atoms since chemistry limits the size of the rigid fragment. For this reason, we use the texture cache to store the protein pocket values. Moreover, to be able to perform coalesced memory accesses on multi-dimensional arrays from different blocks, we extensively use CUDA *pitched arrays* [119] to store temporary tensors that are needed across kernels: by respecting alignment constraints, they guarantee optimal memory layout.

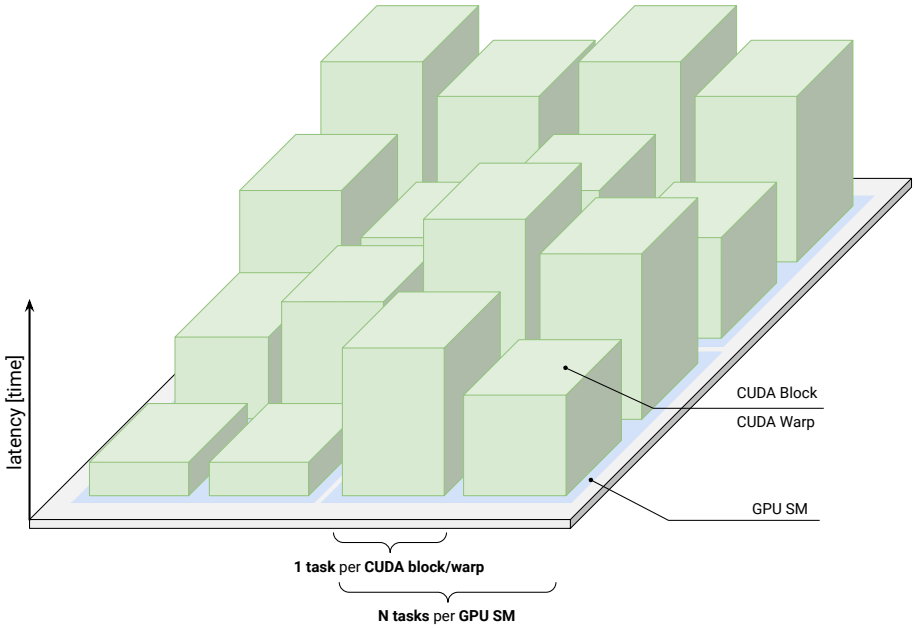


Figure 1.4: Task-to-resources mapping for the throughput-oriented strategy on the GPU. Execution time (x axis) is worse than with the latency-oriented approach, but each task is granted a fixed, minimum amount of resources. A single warp processes a task, and multiple tasks can be fit into the resource pool of a single GPU streaming multiprocessor (SM).

1.5 Throughput-Optimized Kernels for Task-Based Workloads

The second version of the application is the *batched* (or *throughput-oriented*) implementation. This implementation follows a different paradigm from the previous approach: instead of using the whole GPU to process a single ligand at a time, we pack it with as many ligands as possible that are processed in parallel (using fewer resources per ligand) as depicted in Figure 1.4. This approach follows a paradigm similar to the NAS [121, 122] benchmark suite, where its throughput efficiency is leveraged to measure peak floating point performance of a GPU. Adopting this approach is possible since the amount of data per ligand is limited (up to 20 kB input and 1 MB output), and thus we can upload on the GPU a much greater number.

When optimizing for throughput instead of latency, the time to process a single ligand (t_{batch}) will be greater than the time required by the previous implementation ($t_{latency}$); however, many more ligands will be processed in parallel during the overall kernel execution time (t_{batch}). As long as the size of the batch of ligands processed in parallel is greater than $t_{batch}/t_{latency}$, this implementation is expected to deliver higher throughput than the *latency*

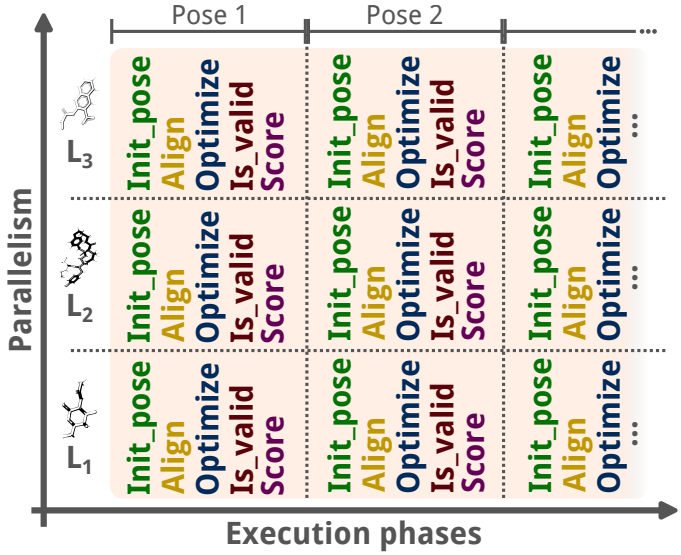


Figure 1.5: Logical mapping on how the batched approach hinges on GPU parallelism to accelerate the execution time. All the steps are implemented in a single kernel.

one since it reduces to the minimum the number of synchronization points.

When we focus on the kernel design, we must take a completely different approach. The main idea is to parallelize the loop at line 2 of Algorithm 1 and to implement all the steps in Algorithm 2 sequentially in the same kernel.

The starting point is to adopt a *warp-synchronous* [90] approach to kernel writing. The underlying idea is to use each warp as an autonomous execution unit, each one processing whole tasks, without having the task itself spread across multiple warps in the same block. This way, the size of a block becomes irrelevant as long as the number of threads per block is a multiple of the warp size (32 threads). After an extensive search of the parameter space, we found the optimal configuration to be 1 warp per block on both NVIDIA V100 and A100 GPUs. A CUDA thread may process more than one atom when the molecule has more than 32 atoms and conversely some remainder threads are inactive if the molecule has less than 32 atoms. We launch the kernel over an amount of ligands enough to cover the GPU parallelism. Figure 1.5 provides a schematic view of the logical mapping. It is important to note that using a single warp to compute a ligand implies that all reductions become warp-wide synchronous operations: we perform those with a mix of CUDA warp intrinsics and CUDA *cooperative groups* [119]. In this implementation, external parallelism (CUDA grid) is addressed by docking multiple ligands simultaneously, while thread-level parallelism by distributing the set of operations on the atoms of a single ligand over a single warp.

To make this approach effective, we need to address some critical as-

pects. The obvious one is that we need a large number of ligands to fully utilize the GPU. This is not a concern since, as mentioned in Section 1.1, the virtual screening task we are targeting considers a large chemical space (up to millions or billions of candidate molecules). The second one is that, since we are processing batches of ligands concurrently, the overall kernel time will be dictated by the slowest warp of the grid, i.e., the warp assigned to the ligand that requires more operations. Since our kernels' latency is data-dependent, we need to balance the size of the ligands that are collected in a single batch. It is also important to efficiently use registers and shared memory, two precious and scarce resources on NVIDIA GPUs. To make the *throughput* kernels run as fast as possible, temporal locality (i.e., atom coordinates and fragments indices) is leveraged in registers and shared memory. Since this requires defining the resources used by the kernels at compile time, balancing the sizes of the ligands in the batches allows for maximizing the usage of those statically allocated resources. The optimal grid shape is obtained by a parameter search campaign on any new GPU architecture the application is deployed to. To manage load imbalance, we cluster incoming ligands in 5 different batches according to their number of atoms, obtained by analyzing statistical distributions of production data sets: $(0, 32]$, $(32, 64]$, $(64, 96]$, $(96, 128]$ and $(128, 160]$. A beneficial side-effect of statically known batch sizes is the ability to compile multiple versions of the same kernel, each instantiated on the constant upper bound of each range: this allows for precise resource (i.e., registers and shared memory) management and optimization. Thus, the size of each batch is set equal to the maximum number of warps that can be concurrently active on all SMs on that specific kernel launch. We determined this number by using Equation 1.1.

$$l = b \times SM \times \frac{t}{ws} \tag{1.1}$$

In Equation 1.1, we compute the size of each batch l, where SM is the number of SMs available on the GPU, ws is the warp size, and b is the number of blocks that can run on the same SM for any given kernel. Section 1.6 uses an NVIDIA A100 GPU to validate the approach. However, since we compute the number of ligands l for each bucket using a query to the CUDA runtime, the proposed methodology is agnostic about the target architecture. Indeed, we efficiently deployed LiGen on systems also equipped with V100 and H100 NVIDIA cards. The proposed methodology was able to adapt the number of ligands accordingly.

Moreover, since the *optimize* step needs to process the ligand's rotamers sequentially, we can introduce a strong imbalance if we bundle in the same batch ligands with a different number of rotamers. For this reason, we also need to cluster the *ligands* by their number of fragments. We decided to group them by four (i.e., ligands with 0-3 fragments clustered in one batch, ligands with 4-7 fragments in another). This strategy is a trade-off between

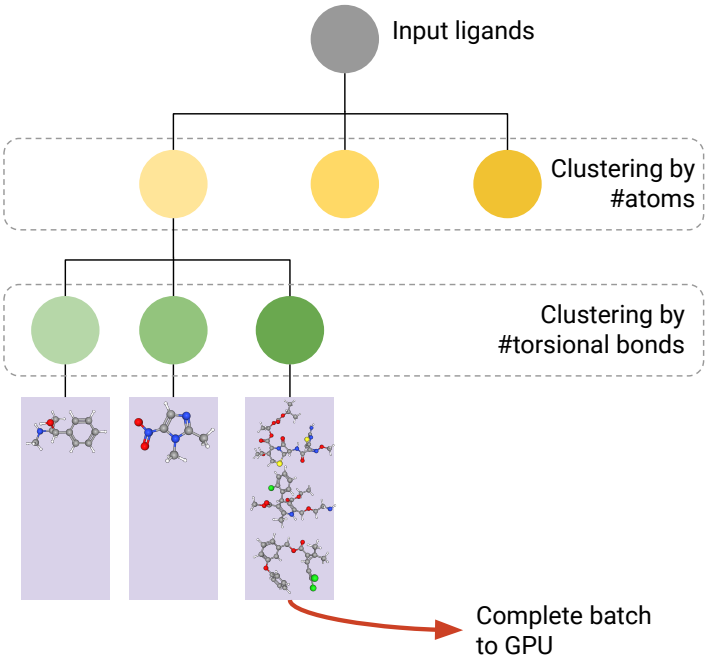


Figure 1.6: Graphical representation of the batch formation process. Incoming ligands are clustered according to their characteristics: first by the number of atoms, then by the number of torsional bonds. When a bucket is full, i.e., when the amount of ligands is enough to allocate all GPU resources, the batch is executed.

maximizing ligands similarity and avoiding a combinatorial explosion of the number of batches. Considering all the resulting ligand clusters, we have a matrix of buckets where we collect ligands with similar features. This clustering process is depicted in Figure 1.6.

1.6 Experiments

This section compares the two implementations in terms of application throughput (i.e., *time-to-solution*) on different datasets. Given that the target molecular docking application has a highly data-dependent throughput, we performed four types of analyses by changing the characteristics and size of ligand libraries.

The first one takes considers uniform datasets (*preprocessed datasets*), where the input ligands have been clustered ahead of time according to their characteristics in terms of the number of atoms and fragments. The goal of this analysis is to highlight the different performance trends free of noise introduced by the different sizes and flexibility of the target molecules.

The second analysis focuses on the application's throughput scaling according to the dataset's size (see Paragraph 1.6.2). In this case, we want to know whether each strategy is always optimal or, as expected, it depends on the size of the dataset. In this second circumstance, we are particularly interested in finding the dataset size that triggers the optimality change. This analysis is performed on preprocessed and real-world datasets, where the molecule size and flexibility are unknown *a priori*.

The third analysis refers to the performance of both implementations on real-world public datasets from the MEDIATE [62] initiative. The datasets are characterized by large compounds and varying chemical characteristics: for this reason is considered representative of an actual virtual screening campaign.

Finally, in Paragraph 1.6.2 and Paragraph 1.6.4, we report an in-depth analysis of the workload using the *instruction roofline methodology*. This analysis has been performed to better understand the different resource utilization for the two implementations.

The benchmarking machine is equipped with 2 × AMD Epyc 7282 CPUs and one NVIDIA A100 GPU attached via PCIe.

1.6.1 Preprocessed Datasets

The first set of experiments aims at showing the throughput of the two implementations when we are running at the best of the application capabilities (i.e., we recorded the average throughput, which is the total number of items processed since the application launch divided by the total execution time of the application, when its value reaches the steady state). We have docked several uniform datasets of 50K ligands, each with fixed characteristics in heavy atoms and fragments. The range is between 20 and 50 heavy atoms and 1 and 20 fragments. In this context, we define every non-hydrogen atom part of the molecule as a heavy atom. We need to point out that for the batched implementation, having the same number of heavy atoms does not mean that all the *ligands* belong to the same batch since LiGen groups them according to the total number of atoms, which also includes the hydrogens. The ranges of heavy atoms and molecule fragments are chosen considering the ones available in commercial databases. Figure 1.7 and Figure 1.8 report the throughput reached by the two implementations for each uniform dataset from two different perspectives. Figure 1.7 plots the varying throughput according to the change in the number of fragments (x-axis) while considering a fixed number of heavy atoms: this feature heavily impacts the throughput. The two implementations show similar behavior, going from a high throughput value with 1 fragment to slowing down more as the number of fragments increases. However, if we look at the y-axis, we can notice that the batched implementation is, on average, three times faster than the latency one.

In Figure 1.8, we plot the variation in the throughput at the change of the

number of atoms (plotted on the x-axis) while keeping the number of fragments constant. We can notice that, in this case, the behavior is different. When we change the number of atoms with a constant number of fragments, the latency implementation has a negligible throughput degradation, while the batched implementation has a more significant throughput loss. However, since it starts from a higher throughput, it still performs better than the latency implementation, in the worst case, by $1.37\times$.

To conclude this analysis, we can see in Figure 1.9 the heatmap of the speedup obtained by the batched implementation compared to the latency implementation, with several datasets of 50K ligands. As we can see, the batched implementation is always better than the latency one, given this dataset dimension on a single GPU. However, we can notice that the amount of speedup changes according to the characteristics of the *ligands*: the batched implementation behaves dramatically better with fewer atoms and a higher number of fragments.

1.6.2 Scaling Analysis

With this experiment, we aim to find the minimal dataset size to reach throughput optimality with both implementations and are interested in seeing how the ligand composition affects this size. We use two different datasets with homogeneous and heterogeneous ligands for this analysis. We considered a set of molecules with 35 heavy atoms and 12 fragments within the first dataset. This dataset has been selected based on the average values for ligand size and flexibility from the ligands considered in the previous section. The second dataset includes a heterogeneous mix of ligands from all previously considered libraries.

Figure 1.10 shows the growth of the throughput (y-axis) at the varying of the dataset size (x-axis). As we can see, the latency implementation outperforms the batched implementation with small datasets. This happens because the batched implementation waits until the batch size is reached and distributes the computation on different CUDA warps. If the dataset is too small and does not reach the size of the batch, we will underutilize the GPU, which explains why in these circumstances, the latency implementation performs better. However, after a certain threshold, we can see that the batched implementation overtakes the latency implementation (with almost exponential growth) until it reaches its saturation point (with a total speedup of around $3.5\times$). This behavior is observed in both the homogeneous dataset (purple and yellow lines) and the heterogeneous one (blue and red lines). The only difference between the two is when the batched implementation overtakes the latency one, and this happens for the homogeneous dataset one order of magnitude earlier. We can also notice that the growth phase of the batching application when using a mixed dataset ends at almost 10^6 ligands. This means that to get the maximum out of this

implementation, we need to dock a large dataset with at least 10^6 ligands for each GPU involved in the computation. On the other hand, for the homogenous dataset, 20K ligands are enough to reach a steady state. Finally, it is interesting to notice the fluctuations of the throughput in the yellow line (homogeneous batched implementation). As mentioned, the batches are created according to the total number of atoms, including hydrogens, and some *ligands* are processed in different batches. When many buckets are processed, even if they are not completed (i.e., small batches), the resources are not well used, resulting in a performance loss. The higher the number of ligands, the lower the probability of falling in this situation, which can be noticed by the fluctuation reduction while increasing the dataset size. The higher the number of ligands, the lower the probability of falling into this situation, which can be noticed by the reduced fluctuation at an increased the dataset size.

1.6.3 Real World Datasets

Finally, we want to evaluate the performance of the two implementations on real-world datasets. These datasets come from the MEDIATE [62] initiative and contain libraries including ligands from different categories: commercial compounds, natural products, drugs, and peptides.

The *Commercial* category represents compounds already available on the market [123]. This set is clustered in three libraries where molecules are selected according to their molecular weight (MW). The first contains ligands with a molecular weight lower than 330 (MW330-), the second set has ligands with a molecular weight between 330 and 500 (MW330-500), and the last one contains all the ligands with a molecular weight higher than 500 (MW500+). The *Drugs* category contains known drugs, including the set of safe-in-man drugs, commercialized or under active development in clinical phases. The *Natural* category contains two sets of molecules: *Foods* and *Natural Products* taken from the *FooDB* [124] online database. Finally, *Peptides* were generated by mixing in a combinatorial way all 20 natural amino acids. They are collected in three files according to the number of amino acids that compose the peptide. In particular, 2AA contains dipeptides (peptides formed by two amino acids), 3AA contains tripeptides, and 4AA contains tetrapeptides. All peptides have been constructed with an extended structure and optimized with MOPAC [125]. They have been protected with acetylation of the N-terminal and the addition of amide in the C-terminal. Since they are built by a combination of the 20 amino acids found in nature, the total amount of peptides is quite low and not evenly distributed. To better contextualize the different sets concerning the analysis done in the previous subsections, a characterization of them in terms of the size of the ligand library, number of heavy atoms, and rotatable bonds, has been performed (Table 1.1).

Figure 1.11 reports the observed throughput for the two approaches on

the MEDIATE datasets. We can notice that the batched version strongly outperforms the latency implementation on the largest files (the *Commercial* compounds with the different molecular weight). This is expected since we are considering 5 million molecules, which heavily exceeds what we have found to be the cross-over point (Paragraph 1.6.2). However, the remaining files are smaller. There are, in particular, two datasets (*Drugs* and *Peptides_2AA*), where the batched version is unable to reach its optimal performances and a throughput good enough to be better than the latency implementation. The first dataset has 14K ligands, which should be enough for the scaling analysis to exceed the latency implementation's performance at least. However, it cannot reach a good throughput because it is heavily unbalanced. Thus, at run time, it forces the execution of several almost-empty batches, which is detrimental to the overall execution. On the other hand, the *Peptides_2AA* is a minimal dataset, and even if it is quite uniform, it still does not have enough data to outperform the latency implementation. In all the remaining libraries, the batched implementation performs closely or better than the latency but cannot reach a steady state.

1.6.4 Micro-Architectural Profiling

The previous analysis shows that the batched implementation has a slow start but a better overall throughput. Now, we want to analyze the two implementations more in-depth to find the reason behind this result, given that the batched implementation's performance improvement goes beyond the reduction in the grid-level synchronization. To reach this goal, we will characterize both workloads in terms of execution profiles, applying the *instruction roofline methodology* [126], using GIPS (Giga Instructions Per Second) to assess and measure performance on an input dataset constructed to be representative of different molecule categories from real-world datasets [62].

We now consider the dimensions that affect the computational complexity of the workload, namely the number of atoms and rotatable bonds. We

Table 1.1: MEDIATE dataset characterization. For each library, its size and the average values (± standard deviations) for the number of heavy atoms and rotatable bonds is reported.

Library	**Size**	**#Heavy Atoms**	**#Rot. Bonds**
Comm. MW330-	1.9M	18.06 ± 4.05	3.65 ± 1.79
Comm. MW330-500	2.8M	28.12 ± 3.70	5.71 ± 2.11
Comm. MW500+	250K	38.46 ± 4.83	8.35 ± 3.52
Drugs	8.8K	29.04 ± 12.89	6.87 ± 5.66
Foods	65.5K	51.06 ± 18.88	37.91 ± 20.45
Natural Products	263.5K	30.94 ± 13.03	6.35 ± 6.10
Peptides 2AA	400	20.07 ± 3.33	7.60 ± 1.77
Peptides 3AA	8K	29.05 ± 4.07	11.40 ± 2.16
Peptides 4AA	160K	37.40 ± 4.71	15.20 ± 2.51

cannot analyze all possible combinations of atoms and fragments. Therefore, we analyze the application performance with three clusters of molecules. The characteristics of the clusters have been chosen in an attempt to highlight different levels of complexity. A sample molecule was randomly selected from the test dataset for each cluster and then duplicated. The duplicate of the molecule in each cluster coincides with the suggested batch size, as described in Section 1.5. A uniform input dataset allows for homogeneous execution paths across all warps involved in a single kernel grid, especially for the batched implementation where each warp handles different input ligands. The results of this analysis would be the same if, instead of an artificial dataset composed of a duplicated molecule, we used a dataset composed of different ligands referring to the same batch. The test molecule clusters have been defined as:

- *Small*: $(0, 64]$ atoms, 1 rotatable bond, batch of 1920 molecules;
- *Medium*: $(64, 96]$ atoms, 12 rotatable bonds, batch of 1600 molecules;
- *Large*: $(96, 160]$ atoms, 20 rotatable bonds, batch of 960 molecules.

Since the goal is to understand why the steady state throughput varies across the two approaches, we focus our analysis on each implementation's CUDA bottleneck kernel. For the *latency* version, this is the kernel that performs the ligand fragment optimization (lines 10-16 in Algorithm 2, accounting for 92 % of the overall docking pipeline's runtime).

Resources Allocation

We first analyze the static resource allocation to understand the consequences of different design principles between the two approaches.

In Figure 1.12, the maximum amount of ligands allocated on a single SM is shown. While the *latency* version dedicates all the resources within an SM to a single ligand, the *batch* version allocates multiple ligands to a single warp, allowing for multiple concurrently running ligands in a single SM. In the latter implementation, the registers per thread are the limiting factor for the ligands allocation to an SM. Therefore, the number of ligands assigned to an SM decreases with the increment of their complexity.

This has two consequences: on the one hand, the latency implementation has a more consistent behavior that does not depend on the ligand size; on the other hand, the batched implementation is strongly influenced by the data size. It has an optimal behavior with small ligands and degrades, increasing the size of the ligand.

Moreover, we can see that the batched implementation can process more ligands per SM, which allows it to reduce the overheads when launching the kernels since it will have fewer kernels to launch. Indeed, while in the latency

implementation, we have to launch at least one kernel per ligand, we process between 960 and 1920 ligands with a single kernel in the batched one.

Roofline Analysis

In this section, we compare the use of computing resources for the two implementations to understand if this could explain the performance differences. In this analysis, we are more interested in the differences between the two implementations than in their absolute values. We present a comparison between different roofline plots [126] produced by measuring the execution behavior of both approaches via the NVIDIA NSight [127] profiler for both instruction performance (Figure 1.13) and shared memory utilization (Figure 1.14).

In particular, in Figure 1.13a and Figure 1.13b, we report the instruction issued roofline. These rooflines are obtained by considering all kinds of warp-level instructions issued. From these two graphs, both application implementations are not memory-bound. Moreover, we use the GPU appropriately since we are close to the roof. If we look at their behavior regarding the size of the different molecules, we can notice a difference in the two implementations. On the one hand, in the *batch* implementation, the amount of GIPS decreases with larger molecules; on the other hand, in the *latency* implementation, the GIPS value increases with larger molecules. This is expected due to respective scaling design choices: on the latency version, we improve the number of instructions because the efficiency of the kernel is constant; thus, with bigger molecules, the amount of data to feed the GPU increases. On the other hand, in the batched implementation, we use more registers to store bigger ligands, which results in fewer active threads per SM and decreases the number of instructions issued.

These two plots also provide insight into cache reuse: the horizontal distance between points of the same molecule class represents the cache's ability to satisfy a request. The larger the distance between two points, the more data can be reused in the highest-level memory (i.e., the distance between L1 and L2 caches represents the ability of the L1 cache to serve the read request).

The *latency* implementation (Figure 1.13a) shows regular cache reuse across molecule classes, and we can notice that the reuse of the L2 cache increases with the size of the ligands; we can see from the image that the distance between the red symbols (L2) and purple symbols (HBM) are greater when comparing squares (Small ligands) with circles (Large ligands). On the other hand, the *batched* implementation (Figure 1.13b) has a high L1 reuse for *Small*, but the L1 arithmetic and instruction intensities are $\sim 10\times$ lower than L2 and HBM values. However, larger molecule classes begin to rely heavily on L2 cache: this can be seen by the fact that the HBM arithmetic and instruction intensities are $\sim 100\times$ higher than L1 and L2 values.

This also strengthens the idea that the batched implementation has better behavior with small molecules but degrades with the data size growth.

The second set of images reports the *shared memory roofline* (Figure 1.14a and Figure 1.14b). They are obtained by measuring both warp-level load/store instructions issued and shared memory transactions performed. The x-axis indicates how efficient the kernel is in terms of shared memory access within the interval between no bank conflict and 32-way bank conflicts. It is the ratio between the number of shared load and store instructions issued by warps and the actual number of shared memory transactions. For example, in case of no conflict, we can accommodate the load/store operations of all warp threads in one shared memory transaction; on the contrary, we need to serialize all of them. The y-axis represents the number of shared memory load/store instructions per second. Both implementations show little to no impact due to shared memory bank conflicts and, thus, an efficient access pattern.

From this analysis, the two implementations look similar, with the batched one showing a slightly better utilization of the GPU for small ligands. At the same time, the latency kernel is more efficient for large ligands. However, this analysis is unable to explain the speedups that we have found from the experiments done up to this point.

Instruction Mix

In this section, we want to investigate the execution profiles of the two implementations to gain more insight into them. The results of this analysis are reported in Figure 1.15.

Figure 1.15a reports the *occupancy*, defined as the ratio between sustained and peak percentage of active warps per SM (measured by relevant performance counters[1]). Occupancy is one of the factors that can be used to improve performance. However, there are others since it is possible to reach optimal performances by decreasing the occupancy and having more registers per thread [128]. For this reason, we are not interested in the absolute value in this graph, but we are looking at the comparison between the two implementations. Both implementations show a comparable degree of SM occupancy. We can notice that while for *batch*, it decreases with an increasing molecule complexity (more registers used), for *latency*, the behavior is uniform. This analysis does not provide insight into the difference in throughput but helps explain why the advantage of using the batched implementation decreases with larger molecules.

Figure 1.15b reports the *efficiency*, defined as the degree of thread predication across all the instructions executed in a single SMSP [119] (measured

[1] `sm__maximum_warps_per_active_cycle_pct` and `sm__warps_active.avg.pct_of_peak_sustained_active` respectively

by relevant performance counters[2]). Both implementations show high execution efficiency and thus low degrees of thread predication. Slightly lower efficiency in *batch* is ascribed to molecule sizes not being a multiple of the warp size. This plot demonstrates how both implementations are efficient in the use of resources.

Finally, Figure 1.15c reports the *instruction mix*, defined as the percentage of instructions executed in a single SMSP grouped by instruction type:

- `fp`: floating point instructions (any precision, including scalar, FMA, and tensor),
- `int`: integer instructions (any integer data type),
- `mem`: memory operations (load/stores),
- `cf`: control flow operations,
- `comm`: inter-thread communication and synchronization,
- `misc`: everything else including bit-wise operations and casts

Table 1.2 lists the `ncu` metrics used to measure instructions and data operations on the GPU cores. Translation from the *legacy* `nvprof` metrics used in [126] and their `ncu` counterparts used in this chapter has been carried out according to [127]. There are two interesting pieces of information in this figure. The first one is that the largest part of the operation done is integer arithmetic. This is expected since they comprehend index calculations, and the *Score* function used to select the best pose is a sum over integer values. Moreover, if we look at the latency implementation, it has a large (20 % to 40 %) of `comm` instructions that almost completely disappear in the batched implementation. These `comm` instructions are mostly due to the design of the latency kernel, requiring synchronization between multiple warps to perform block-wide reductions. As mentioned, we need to process all the fragments sequentially in the pose optimization phase. To analyze the impact on the performance of the *check_ bump* kernels, we have run both implementations without the early exit from this loop and report the result in Figure 1.16. The advantage in terms of speedup in using the batched approach has been reduced a lot and reached a maximum value of 2× only for very small ligands. This is expected since the previous analysis shows that the batched approach is more efficient for small molecules. On other molecule dimensions (i.e., larger in terms of atoms and fragments) the speedup is slightly above 1, including a slight slowdown for the bottom left corner. This analysis confirms that the management of the early exit condition is the tie-breaker between the two implementations since the batched version can be used without introducing much synchronization overhead.

[2] `smsp__thread_inst_executed.sum` for thread-level instructions, `smsp__inst_executed.sum` for warp-level instructions

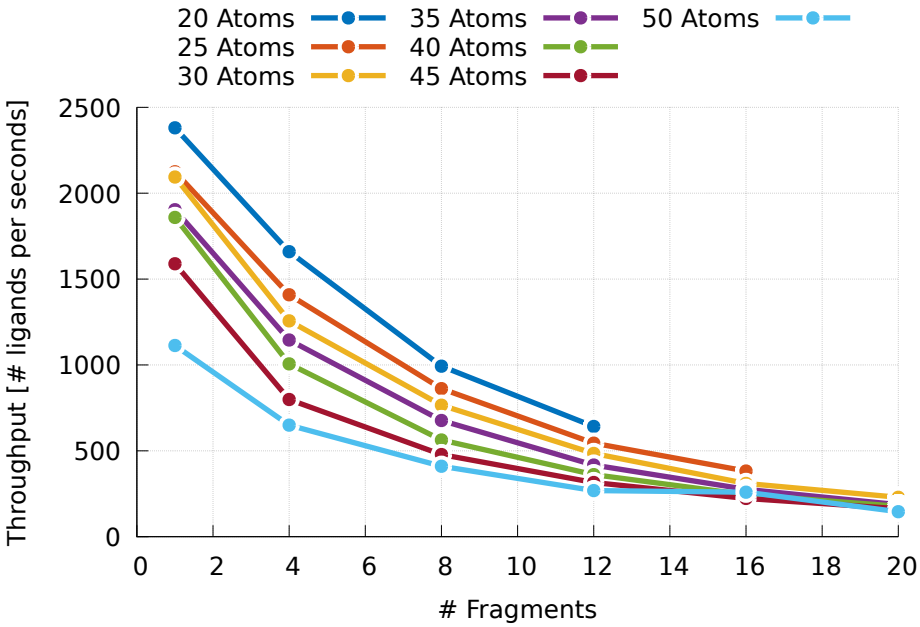
The *latency* implementation demonstrates consistent performance across molecule classes regarding performance, occupancy, and instruction throughput. This is due to its design principle of scaling computing resources based on the complexity of the input ligand.

On the other hand, the *batch* implementation uses a fixed amount of computing resources allocated to a batch of input ligands and deals with the increasing molecules' complexity by increasing the amount of work a single warp must carry out. Moreover, this second implementation has its best behavior with small molecules, and its performances have a slight degradation when increasing the data size because fewer compute resources are used since we need more registers for the data, thus decreasing the number of active threads.

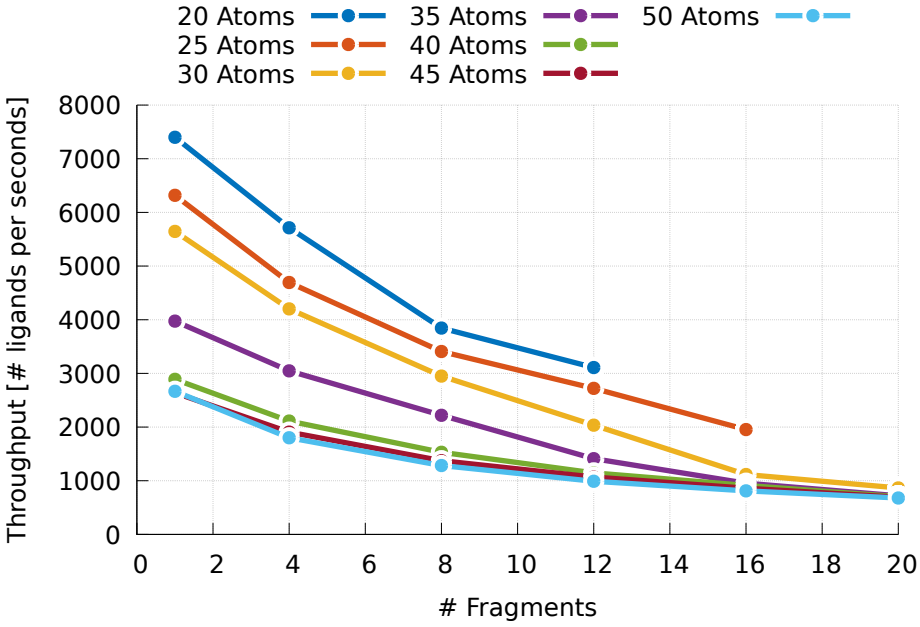
We have seen that a batched method provides a significant benefit, primarily because processing a ligand with a warp can eliminate most synchronization issues among warps in the same SM. This is fundamental in the *check_bump* function because it allows the exploitation of the early exit condition without introducing too much overhead.

Derived Metric	**Metrics (ncu)**
Timing	sm__cycles_elapsed.avg sm__cycles_elapsed.avg.per_second
FLOP	sm__sass_thread_inst_executed_op_dfma_pred_on.sum sm__sass_thread_inst_executed_op_dmul_pred_on.sum sm__sass_thread_inst_executed_op_dadd_pred_on.sum sm__sass_thread_inst_executed_op_ffma_pred_on.sum sm__sass_thread_inst_executed_op_fmul_pred_on.sum sm__sass_thread_inst_executed_op_fadd_pred_on.sum sm__sass_thread_inst_executed_op_hfma_pred_on.sum sm__sass_thread_inst_executed_op_hmul_pred_on.sum sm__sass_thread_inst_executed_op_hadd_pred_on.sum sm__inst_executed_pipe_tensor.sum
Thread Instructions	smsp__thread_inst_executed.sum /32
L1 Global Transactions	l1tex__t_sectors_pipe_lsu_mem_global_op_ld.sum l1tex__t_sectors_pipe_lsu_mem_global_op_st.sum
L1 Shared Transactions	l1tex__data_pipe_lsu_wavefronts_mem_shared_op_ld.sum l1tex__data_pipe_lsu_wavefronts_mem_shared_op_st.sum
L2 Transactions	lts__t_sectors_op_read.sum lts__t_sectors_op_atom.sum lts__t_sectors_op_red.sum lts__t_sectors_op_write.sum
DRAM Transactions	dram__sectors_read.sum dram__sectors_write.sum
Warp Instructions	smsp__inst_executed.sum
Warp global load/store	smsp__inst_executed_op_global_ld.sum smsp__inst_executed_op_global_st.sum
Warp shared load/store	smsp__inst_executed_op_shared_ld.sum smsp__inst_executed_op_shared_st.sum
DRAM bytes	dram__bytes.sum
L2 bytes	lts__t_bytes.sum
L1 bytes	l1tex__t_bytes.sum
Inst. mix: integer	sm__sass_thread_inst_executed_op_integer_pred_on.sum
Inst. mix: control flow	sm__sass_thread_inst_executed_op_control_pred_on.sum
Inst. mix: thread comm.	sm__sass_thread_inst_executed_op_inter_thread_communication_pred_on.sum
Inst. mix: memory	sm__sass_thread_inst_executed_op_memory_pred_on.sum
Inst. mix: misc.	sm__sass_thread_inst_executed_op_bit_pred_on.sum sm__sass_thread_inst_executed_op_conversion_pred_on.sum sm__sass_thread_inst_executed_op_misc_pred_on.sum

Table 1.2: Metrics for the Instruction Roofline Model and Instruction Mix analysis.

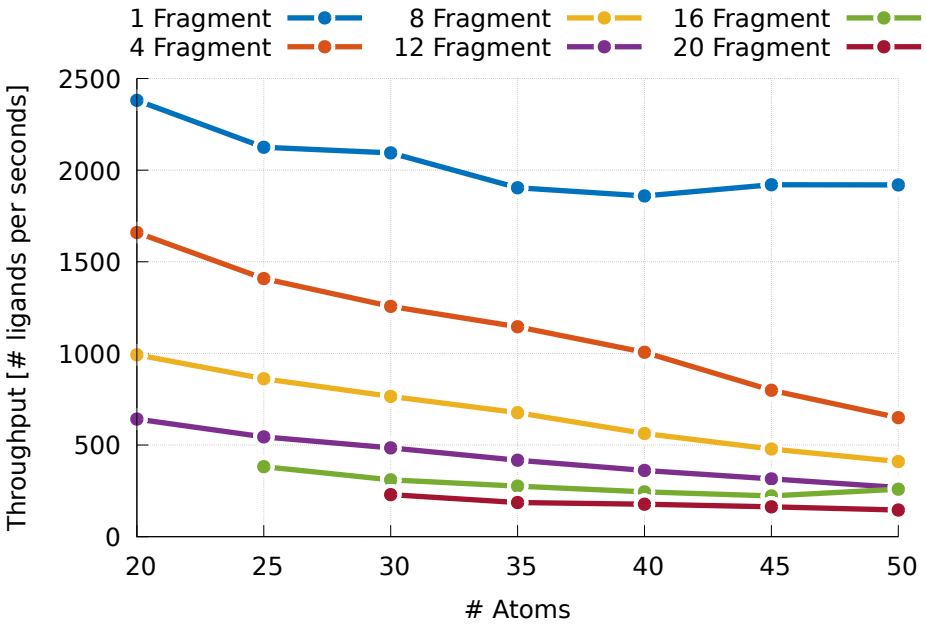


(a) Latency

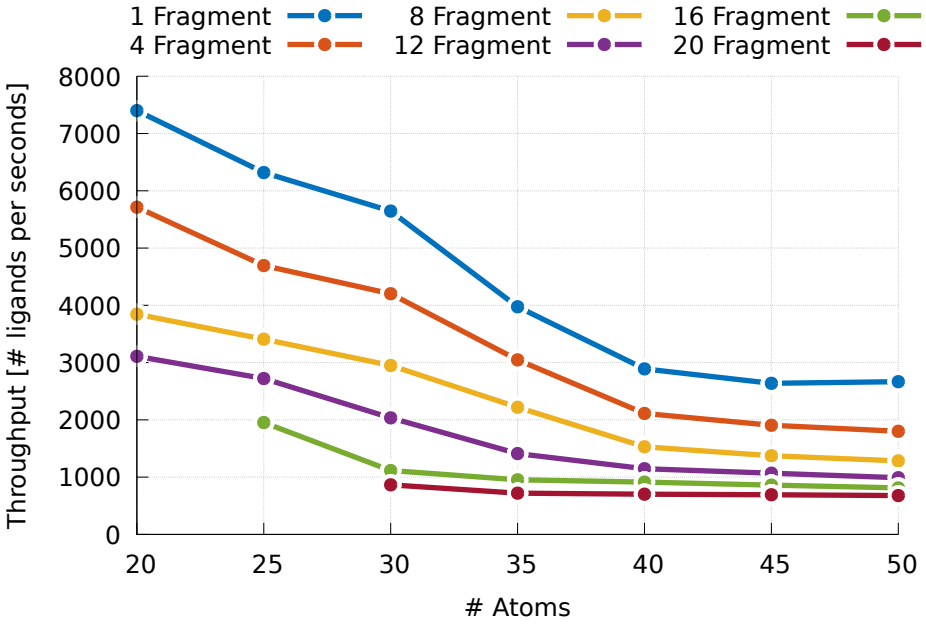


(b) Batched

Figure 1.7: Throughput of the two implementations with the different datasets, organized by the number of atoms and increasing the number of fragments on the x-axis.



(a) Latency



(b) Batched

Figure 1.8: Throughput of the two implementations with the different datasets, organized by the number of fragments and increasing the number of atoms on the x-axis.

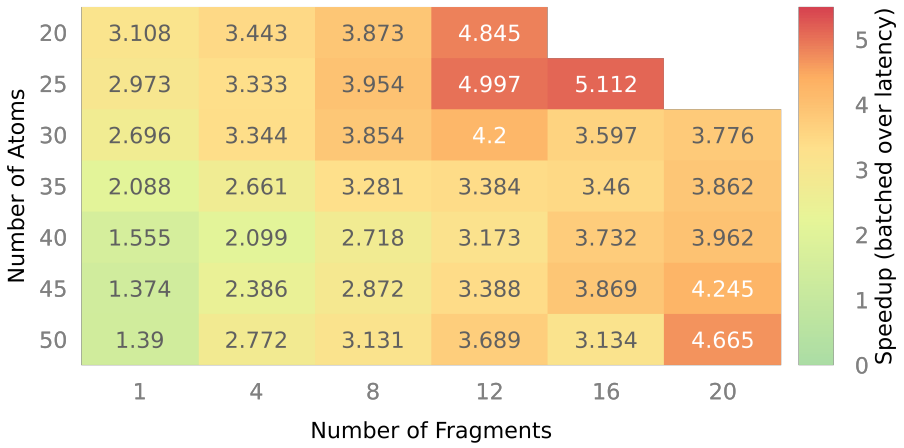


Figure 1.9: Speedup heatmap of batched versus latency for different homogeneous datasets of 50K ligands with the same characteristics.

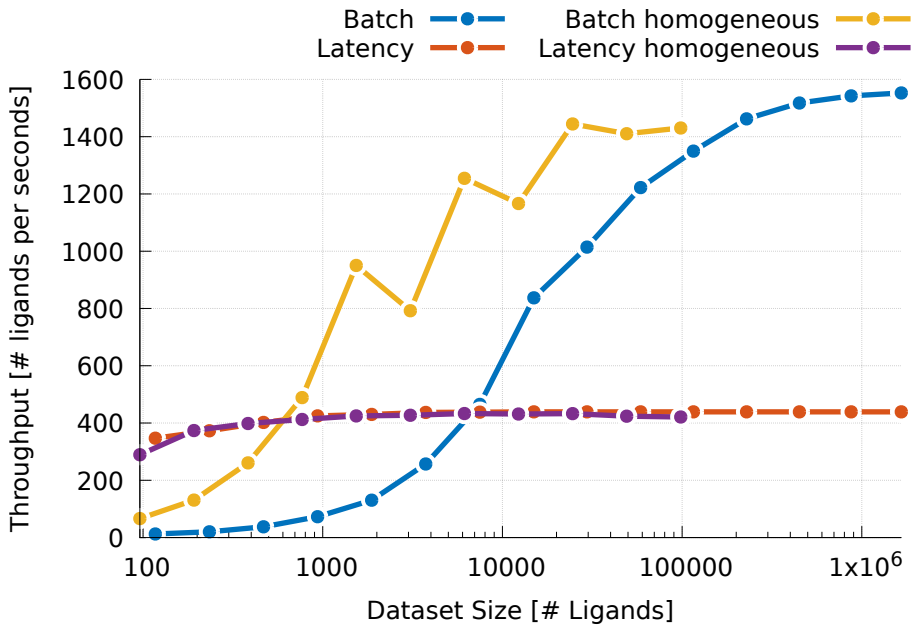


Figure 1.10: Single GPU throughput behaviour with varying input dataset size for all the presented approaches. While the latency-optimized kernel dominates the throughput-optimized one on small datasets, after the break-even point at around 1000 ligands the latter scales up to a sustained 1600 ligands per second.

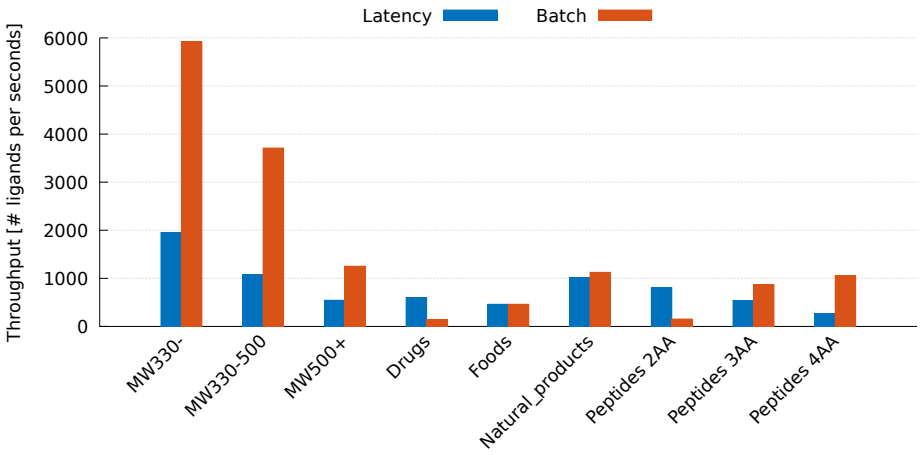


Figure 1.11: Throughput comparison on the Mediate dataset.

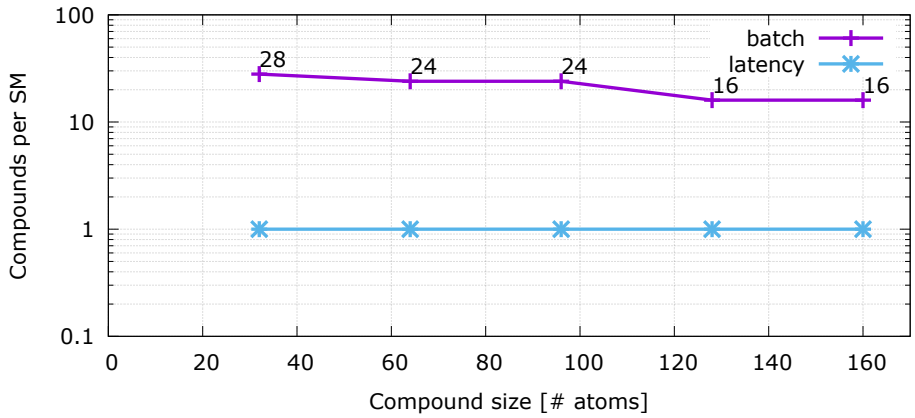
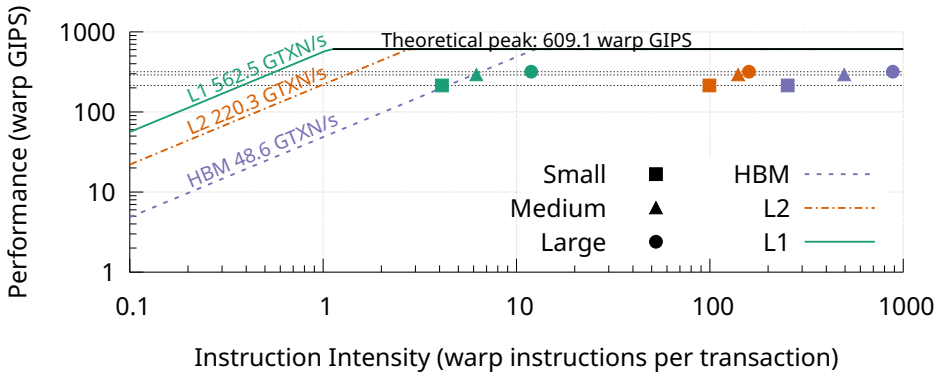
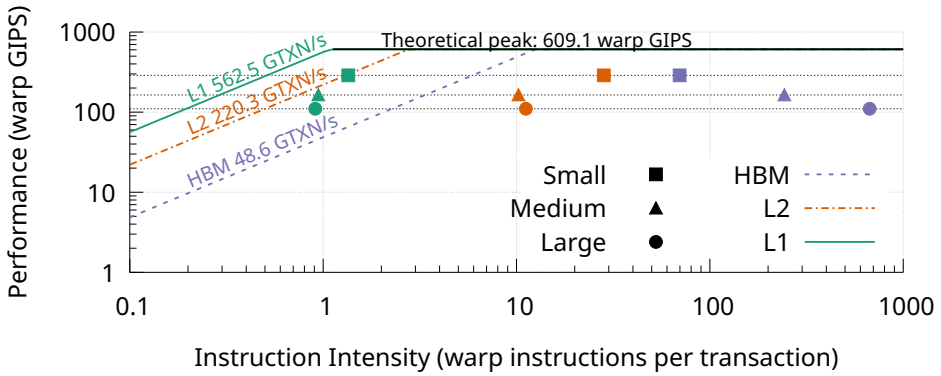


Figure 1.12: Ligand allocation per GPU streaming multiprocessor (SM).

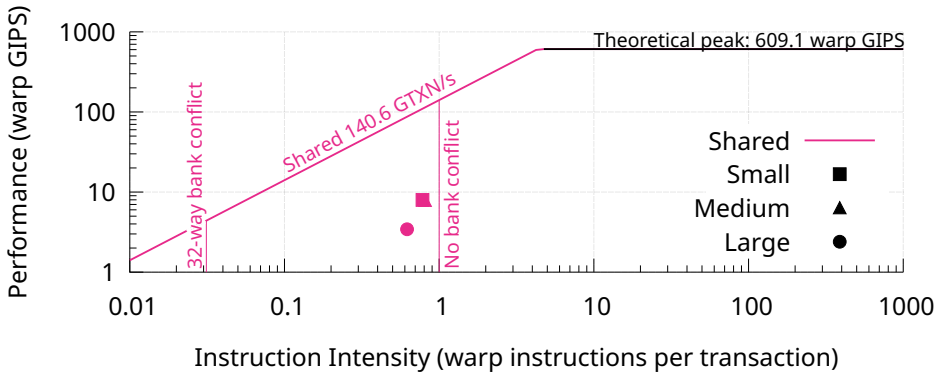


(a) Instruction roofline for *latency*

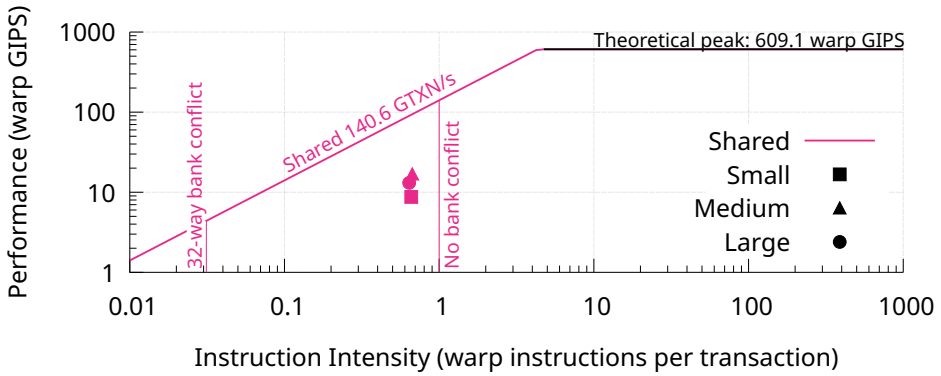


(b) Instruction roofline for *batch*

Figure 1.13: Roofline analysis comparison between *latency* (top) and *batch* (bottom) on instruction performance.

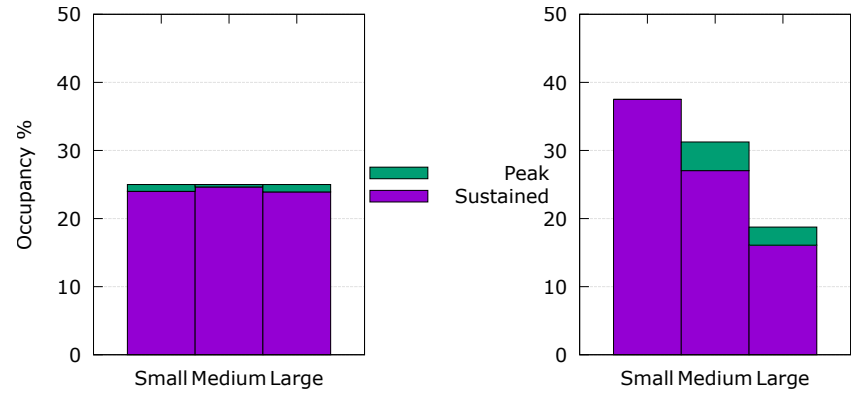


(a) Shared memory access pattern roofline for *latency*

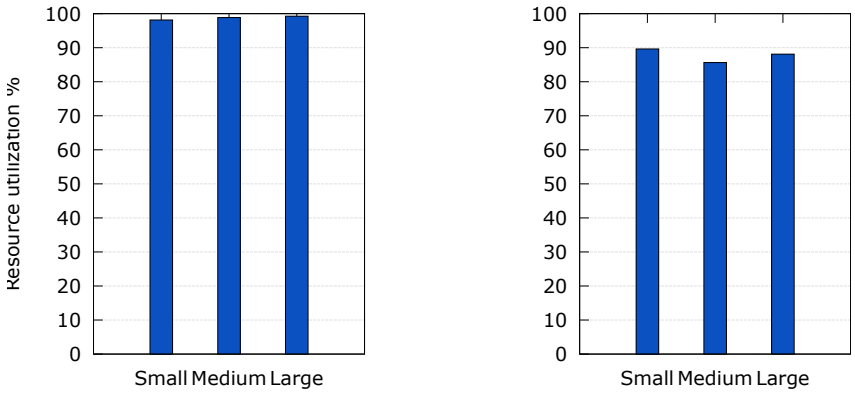


(b) Shared memory access pattern roofline for *batch*

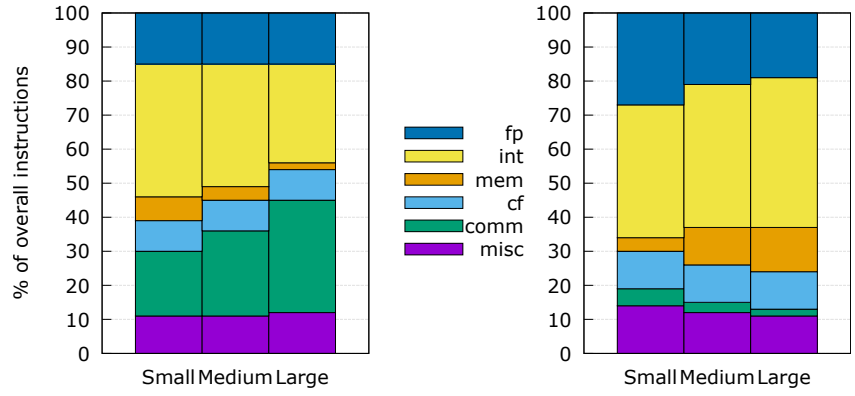
Figure 1.14: Roofline analysis comparison between *latency* (top) and *batch* (bottom) on shared memory access pattern.



(a) SM occupancy



(b) Efficiency (thread predication)



(c) Instruction mix

Figure 1.15: Comparison between *latency* (left) and *batch* (right) on (a) peak and sustained active warps, (b) efficiency (or thread predication) and (c) instruction mix.

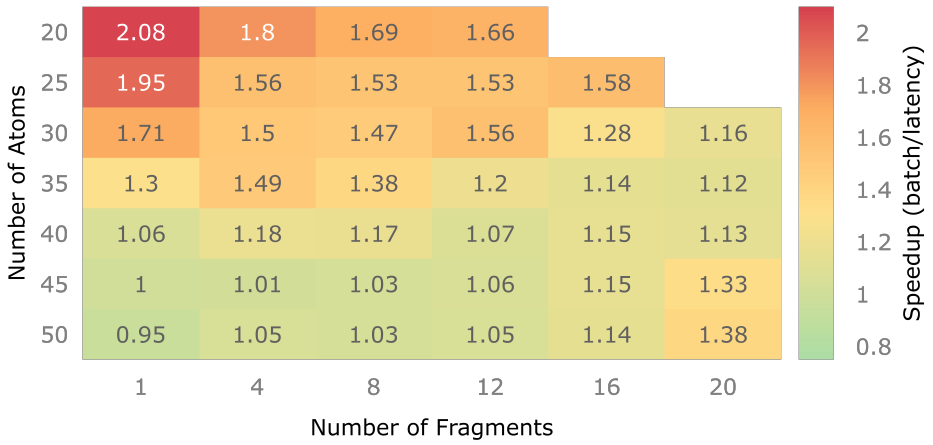


Figure 1.16: Speedup heatmap of the batched version against the latency one for the different homogeneous datasets without the early exit from the *check_bump* function. Both throughputs are taken with large enough datasets.

1.7 Urgent Computing Against COVID-19

The social and economic impact of the COVID-19 pandemic demanded the reduction of the time required to find a therapeutic cure.

During the first wave of the global pandemic in 2020 [129], the European Union called for urgent action to leverage the continent's HPC resources aiming at finding possible active drugs in the shortest time possible. The Exscalate4CoV initiative[3] gathered 18 institutions and resulted in 1 clinical trial (repurposing of Raloxifene [130]) and $>$ 400 compounds found to be active in experimental assays [131]. On the HPC side, the joint effort of the collaboration between CINECA, the research group led by Prof. Gianluca Palermo at Politecnico di Milano and NVIDIA optimized and deployed the Exscalate platform on two HPC machines with a combined throughput peak of 81 PFLOP/s to rank a chemical library of more than 70 billion small molecules against 15 binding-sites of 12 viral proteins of SARS-CoV-2. This required a deep re-design of the Exscalate molecular docking platform to benefit from heterogeneous computation nodes and avoid scaling issues. The extreme-scale virtual screening simulation, known as *one-trillion-docking experiment*[4], is still to date the largest *in-silico* drug discovery simulation ever performed. It lasted 60 hours and improved on the previous largest experiment [114] by 50 $\times$ the number of screened compounds and 7.5 $\times$ the number of protein targets. The knowledge generated by this experiment, in terms of top-ranked molecules for each protein pocket (binding sites), is available via the open-access MEDIATE[5] [62] initiative, in an effort to foster a collaborative environment in case of future pandemics.

In the context of *urgent computing*, where the time required to find a therapeutic cure should be as short as possible, the Exscalate platform has been re-designed with the goal of virtual screening as many ligands as possible in a feasible time budget, i.e. hours, for billions of ligands instead of months necessary before. To maximize the throughput of the docking platform, the target are TOP500 European HPC systems: other than sheer accelerator throughput (covered in the first part of this chapter), pre-exascale scale-out in production poses known challenges [132, 133], from data management to file system layout optimizations, from MPI topology tuning to fault management and recovery.

1.7.1 Related Work

The state-of-the-art of high-performance molecular docking on GPU accelerators has been explored in Section 1.1.

[3] https://www.exscalate4cov.eu/
[4] https://1trilliondock.exscalate4cov.eu/
[5] https://mediate.exscalate4cov.eu

However, the focus on *urgent computing* requires a holistic point of view to take into account all aspects needed to large-scale production runs. AMIDE [134] focuses on inverse docking, where ligands are evaluated in a large number of proteins. They propose to divide each protein into twelve overlapping sub-grids to use as independent pockets. To orchestrate the computation, they use custom scripts and SLURM [135] job arrays. METADOCK [136] focuses on blind docking, where the docking phase is not restrained in a specific pocket, but it can be docked in the whole protein's surface. They propose to use a combined OpenMP/CUDA approach to leverage NVIDIA accelerators.

In this regard, AutoDock [96] is the most relevant work since it has been ported to CUDA [111] and deployed on the Summit supercomputer [137] where they docked over one billion molecules on two SARS-CoV-2 proteins in less than two days [114]. They hinge on the Summit's NVMe local storage to dock batches of ligands in the target pocket and to store the intermediate results. In particular, AutoDock uses OpenMP to implement a threaded-based pipeline, where each thread reads ligands from the file, launches the CUDA kernels, waits for their completion, and it writes back the results. Since most docking algorithms use a fast but approximated scoring function to drive the estimation of the 3D pose of a ligand, it is common to re-score the most promising ones with a more accurate scoring function. They use a custom CUDA version of RFScore-VS [138] to perform such task and BlazingSQL [139] for computing statistics and selecting the top-scoring ligands. To orchestrate the workflow, they use FireWorks [140] from an external cluster to ensure a consistent state in the presence of faults in the compute nodes.

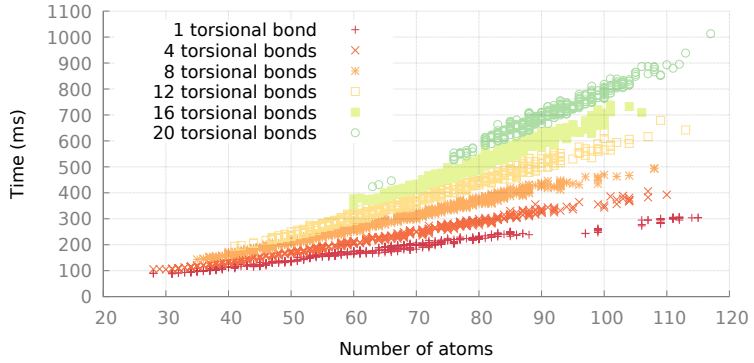
1.7.2 High-throughput Docking Workflow

A monolithic application to dock and re-score the ligands has been developed, using MPI [141] to scale out, *C++11* threads to scale up and CUDA kernels to accelerate the compute-intensive sections, exploiting the NVIDIA V100 GPUs available on the target HPC clusters. The proposed solution can reach a high throughput without relying on the node's local storage, which is unavailable in the target HPC systems. The innovation introduced in this work can be categorized into three main contributions:

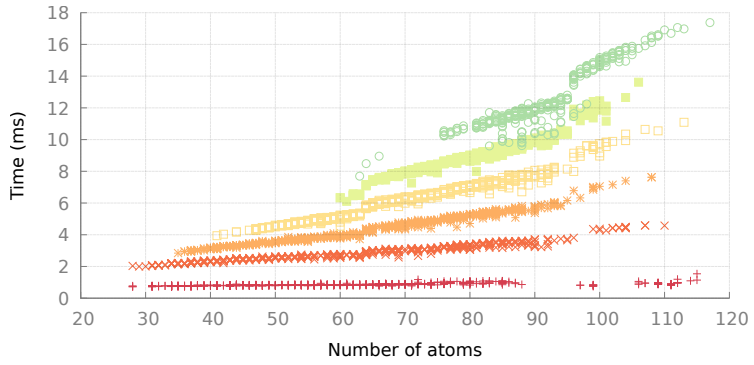
1. at the *algorithm* level, with the CUDA porting and optimization of the docking and scoring phases.

2. at the *application* level, with the complete rework of the application and the creation of the high-throughput molecular docking application.

3. at the *workflow* level, with the creation of the Exscalate workflow that allows us to handle the operation easier and more resiliently.



(a) *C++* implementation on CPU



(b) CUDA implementation on GPU

Figure 1.17: Time required to dock and score a ligand by varying the number of atoms and torsional bonds. The C++ implementation use a single core IBM 8335-GTG 2.6 GHz. The CUDA implementation use a single NVIDIA V100.

The final output of the algorithm is an estimation of the bond strength between a given ligand and the binding site of the target protein. In the virtual screening context, reducing the problem's complexity is common by using heuristics and empirical rules instead of performing a molecular dynamic simulation [142]. One implication of this choice is that the numeric score of a ligand is strongly correlated by the given 3D displacement of its atoms, which is not trivial to compute due to the high number of degrees of freedom involved in the operation. In addition to the six degrees of freedom derived by rotating and translating a rigid object in 3D space, the ligand's flexibility has to be considered. A subset of the ligand's bonds, named *torsional bonds* [143], partition the ligand's atoms in two disjoint sets that can rotate along the bond's axis, changing the ligand's shape. A small molecule can have tens of torsional bonds.

The algorithm developed in Exscalate to dock and score a ligand comprises four steps. The first step is ligand pre-processing, which flattens the ligand by rotating the torsional bonds to maximize the sum of the internal distances between all the molecule atoms. This computation is protein-independent. The second step docks the ligand inside the binding site of the target protein by using a greedy optimization algorithm with multiple restarts. The scoring function used to drive the docking considers only geometrical steric effects. While the ligand's flexibility is taken into account, the pocket is considered a rigid body [144]. 256 different initial poses have been evaluated for each ligand in the experiment. The third step sorts the generated poses to select only a few to re-score using the LiGen chemical scoring function [145] in the fourth step. In particular, the generated poses are clustered using a root mean square deviation of atomic positions (RMSD) of 3 Å as the threshold to deem two poses as similar. Then all the poses are sorted according to their score. Only the top 30 poses are scored for each ligand. The score of the ligand is the score of the best pose found.

The only information required to dock and score a ligand in the target binding site is its description. Thus, the virtual screening process is an embarrassingly parallel problem. However, it is paramount to design how the data can be read from the storage, transferred to the accelerator, and written back to the storage. Indeed, another innovation introduced with this work is the high-throughput docking application, which aims to address all the issues that are not related to the docking and scoring kernels but are required for the experiment's success, such as data management, resource organization, and multi-node scaling.

The workflow application implements an an asynchronous pipeline that uses MPI point-to-point operations for data exchange. A single process (MPI rank) is executed per node. Each process manages all the resources on the node and spawns a local asynchronous pipeline where each stage is a dedicated worker thread.

The first stage is the *reader*, which reads from the actual file that represents the chemical library a chunk of data that it enqueues in the *splitter*'s queue. The splitter stage inspects each chunk to split all ligand descriptions. Then it enqueues each ligand description in the *docker*'s queue. In the experiment, a ligand is described using a custom binary format derived from the TRIPOS Mol2 format. The *docker* stage dequeues a ligand description, it constructs the related data structures, performs the dock and score steps described in Section 1.1, and it enqueues the ligand's score in the *writer*'s queue. The writer stage dequeues the ligand score and accumulates the related output in an internal buffer, which is the ligand's SMILES [146] representation and its score value. The writer stage initiates the writing procedure when the accumulation buffer is full.

The *docker* stage is the only one that can be composed of several threads that operate on the same queues to enable work-stealing. Moreover, it is pos-

sible to use different algorithm implementations, such as CUDA and *C++*, to leverage the node's heterogeneity. Any *docker* thread is referred to here as *worker*. All the workers that use the CUDA implementation are named *CUDA workers*, while the ones that use the *C++* implementation are named *CPP workers*. Even if a single CUDA worker is tied to a single GPU, it is possible to have multiple CUDA workers tied to the same GPU.

The target binding site is considered constant during the elaboration. Therefore, each process will fetch the related information once at the beginning of the execution. Each algorithm implementation can store the pocket data structures in the most appropriate memory location during its initialization. In particular, the *C++* implementation uses constant static memory, while the CUDA implementation uses texture memory.

I/O operations from virtually all the nodes of an HPC system rely on MPI collective I/O facilities. Since the computation pipeline is the same for all the processes, only one MPI process pipeline is depicted here. Regarding read operations, each MPI process sequentially reads its section of the input file according to its MPI rank. Collective I/O operations for write operations are used, too. Indeed, the user can configure the number of processes that issue I/O operations to reduce the pressure on the file system. All the writing operations are parallel and sequential. Indeed, as apparent from Figure 1.18, the I/O does not represent a bottleneck, and the scaling of the high-throughput molecular docking application is very close to the optimal theoretical scaling.

As can be seen from the strong scaling experiment, storing all the ligands to be docked in a single file and deploying the application on the whole machine is possible. However, this approach has several drawbacks.

The main concern is fault resiliency. The default action to respond to a fault in an MPI communicator, for example, after a node failure, is to terminate all the processes [141], which can lead to a significant waste of computational resources without careful management of application checkpoints: this is a known issue [147, 148, 149].

Another concern is load balancing. Figure 1.17 shows how docking and scoring a large and complex ligand required much more time than a small ligand. Therefore, there is a significant imbalance between the MPI processes if all the ligands with many atoms and torsional bonds are close.

These issues were addressed with a pre-processing phase on the chemical library to have a relatively small number of jobs that can run in parallel using a plain job array to coordinate the execution, such as the one provided by SLURM [135] or PBS [150]. The job array is in charge of controlling the execution of these jobs, and custom reactive tools help it identify failing jobs, re-run them, and exclude failed nodes. To achieve this goal, the amount of ligands (70 billion) is divided into ~3400 smaller sets. For every set, a job running on a subset of 32 nodes is created.

Figure 1.19 depicts the overall Exscalate workflow. Two kinds of input

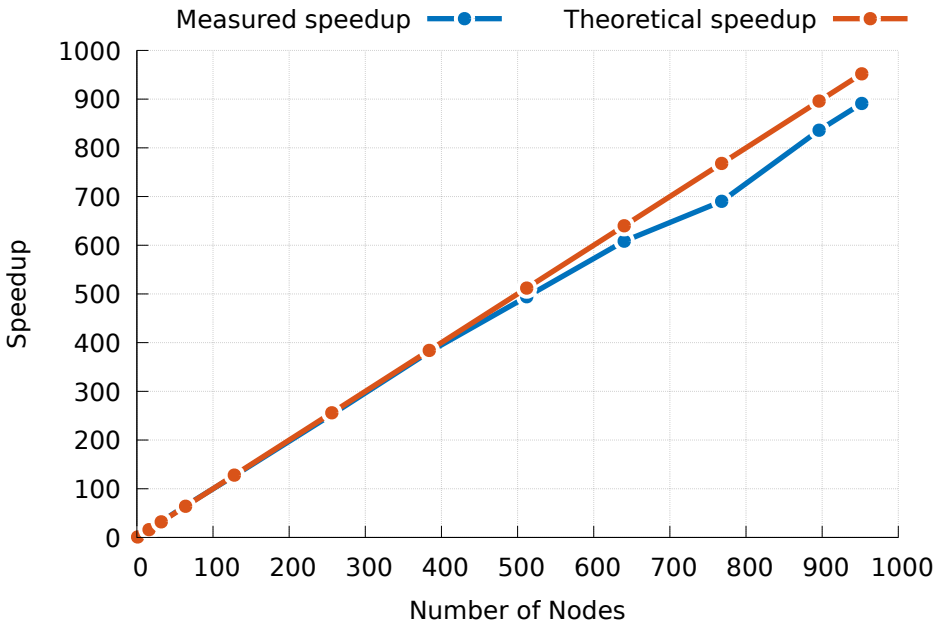


Figure 1.18: Strong scaling experiment of the high-throughput molecular docking on the whole Marconi100 supercomputer.

data are required for a virtual screening campaign: viral protein binding sites and a library of candidate compounds. The identification of protein binding sites is a complex, compute-intensive endeavor that is outside of the scope of this work. In the context of the EXSCALATE4CoV effort, a dedicated team provided the protein structures and described the methodology in Gervasoni et al. [151]. On the other hand, the candidate compound library is provided in the SMILES format [146].

The first pre-processing step is to obtain the 3-dimensional atom conformation from each a-dimensional input SMILES string. This is carried out by the LiGen toolchain itself using the MMFF94 force field [152]. The resulting structures are then minimized via gradient optimization to obtain minimal energy 3-dimensional coordinates for each atom.

The next step in the ligand pre-processing is to broadly classify them in buckets according to their expected execution time, reducing the imbalance during the computation as much as possible. As shown in Figure 1.17, the number of torsional bonds and atoms seem good predictors. However, extracting these properties from from the SMILES representation is not trivial. For this reason, a model is trained, to predict the execution time given properties that are more accessible at this point of the workflow: the number of heavy atoms, rings, and chains. Interactions between them is also considered. To predict the ligand's execution time a decision tree model is adopted, with a maximum depth of 16.

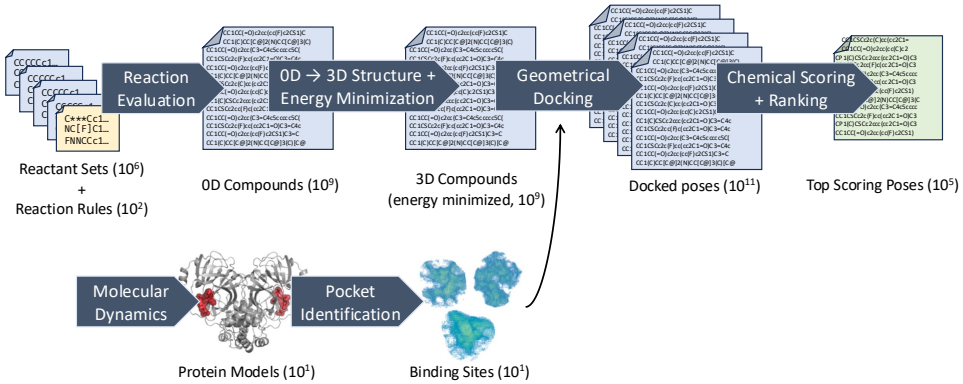


Figure 1.19: Exscalate workflow, from the input (ligand's chemical library and the protein models) on the left to the outcome (most promising set of molecules) on the right.

Figure 1.20 shows the experimental campaign used to train a decision tree regressor [153] written in Python. Figure 1.20a shows the measured execution time of a dataset with 21 million of ligands with a different number of atoms and torsional bonds. The 80 % of the data are used to train the model, while the remaining data are used to compute the prediction error reported in Figure 1.20b. The model has a negligible mean error ($-0.000\,88$ ms), with a standard deviation of ± 3.81 ms.

Even if the average error is close to zero, the standard deviation suggests that there is an error when predicting the docking time of a given ligand. In the experiment, The ligands are clustered in buckets of 10 ms to account for this variability. Since this pre-process aims at avoiding computation imbalance,only the average behavior is taken into account.

The last pre-processing step can be performed after the ligands classification according to their complexity. For each ligand, the hydrogen atoms are added, the initial displacement of its atoms in the 3D space is generated, and the molecule is unfolded(Paragraph 1.7.2). This elaboration is required once, and it can be re-reused in all the virtual screening campaigns.

Finally, the virtual screening campaign can be performed, once the target binding sites and the ligand binaries are available. The docking application is launched on all the ligand files, one pocket at a time.

The output of the virtual screening is the ranking of the chemical library against each docking site. Domain experts selected ligands that strongly interact with multiple docking sites or proteins, allowing them to re-create the 3D displacement of the ligand's atoms on demand. For this reason, only the topological representation of the molecule using the SMILES notation can be stored.

1.7.3 Urgent Computing Setup

This section reports the experimental setup for the virtual screening experiment in which 70 billion ligands were evaluated against 15 binding sites of 12 viral proteins of SARS-CoV-2.

Target Dataset

The ligands evaluated in the experiment are part of the EXSCALATE library owned by Dompé Farmaceutici S.p.A., built starting from a database of millions of available commercial reagents that were combined using a set of robust synthetic reactions in order to obtain a tangible chemical space, meaning that this is truly achievable in one reaction step. The list of target proteins used in the experiments has been reported in Table 1.3, with the corresponding PDB code. The crystal structures of the main functional units of the SARS-CoV-2 proteome were obtained from the Protein Data Bank [154]. Homology models of the proteins for which the crystal structure is unavailable were generated and used.

Table 1.3: The 3D targets used in the molecular docking experiments. A target might have different pockets.

Protein	PDB code
3CL protease (NSP5)	6LU7
N-protein	6VYO
NSP3	6W02
NSP6	De novo model
NSP9	6W4B
NSP12	7BV2
NSP13	6XEZ
NSP14	Homology Model
NSP15	6W01
NSP16	6W4H
PL protease	6W9C
Spike-ACE2	6M0J

Hardware Resources

The Exscalate platform was deployed on Marconi100 [155] at CINECA and HPC5 [156] at ENI S.p.A., aggregating around 81 PFLOP/s of compute capability (respectively 29.3 PFLOP/s and 51.7 PFLOP/s). At the time of the experiment, they were the Europe's two largest HPC systems. A Marconi100 node is equipped with 32 IBM POWER9 AC922 cores (128 hardware

threads) and 4 NVIDIA V100 GPUs attached via NVLINK 2.0. The computation node of HPC5 is very similar since it also uses 4 NVIDIA V100 GPUs, but it relies on Intel Xeon Gold 6252 24C as the host processor (24 cores and 48 hardware threads) and it uses NVLINK only for the direct GPU-to-GPU interconnection. The CPU-to-GPU connection uses PCIe. The experiment has been run using a reservation of 800 out of 970 Marconi100 nodes and 1500 out of 1820 HPC5 nodes for 60 hours on each machine.

Software Environment

For both production systems, all software components were built on top of the same software stack: upstream GCC 9.3, CUDA toolkit 11.0, and upstream MPICH 3.4.1.

The main difference between the two systems was in the job scheduler: SLURM on Marconi100 and PBS on HPC5. On the former, the single 32-nodes jobs were managed in multiple job arrays, each one covering the whole set of docking targets, while on the latter, an ENI internal, proprietary workflow management tool was used to schedule single jobs and deal with transient node faults.

For the post-processing phase, a custom Dask pipeline dealing with statistical descriptors and threshold selection was developed and ran on an environment deployed using upstream conda-forge (Dask 2.21.0 on Python 3.8.3). The same Python environment was also used for the pre-processing phase, where a custom regression model was trained, serialized and deployed using scikit-learn 0.22.1.

Performance Measurements

Since a single MPI process is run on each node, the node's throughput can be measured using standard C++ timing facilities. Each time the throughput has to be calculated, the number of ligands the application has elaborated is divided by the elapsed time. This information is logged in the application output during the evolution of the elaboration. To compute the average node's throughput, the average value among the application final throughput logs is calculated.

To measure the machine throughput, the total number of ligands is divided by the wall time of the computation, i.e., the time required to complete the job array. In this way, the measure includes all the overheads related to the execution. Since a pocket elaboration lasts for hours, the accuracy of the measure is compatible with the method used.

1.7.4 Evaluating the storage requirements

When scaling an experiment to the scale of a trillion docking operations, the data to be read and written must be carefully evaluated, paying attention to

formats. To perform the virtual screening, information is needed about the binding sites of the target proteins and the chemical library of ligands to be analyzed. The former is not an issue since it requires total storage of 29 MB and the information needed is read once the application starts.

Domain experts use the SMILES format to represent a ligand. The chemical library evaluated in the experiment encoded in the SMILES format requires a total of 3.3 TB. However, the docking application requires a richer molecule description, as detailed in Paragraph 1.7.2. The most widely used format to store the required information is the TRIPOS Mol2, encoded in ASCII and focuses on readability rather than efficiency. For this reason, a custom binary format that stores only the information required by the docking application is adopted, such as the atom's position, type, and bonds. By comparing the size of the same molecules, the Mol2 format requires 5 × to 6 × more space concerning the binary format. Nonetheless, the whole binary chemical library for the experiment requires 59 TB of storage.

Storing all the docked poses is unfeasible, as targeting 15 binding sites and re-scoring 30 alternative poses per input ligand would require 26 PB of storage. For this reason, only the SMILES representation of the molecule and its best score as a scalar value for each docking site is retained. The docked posed can be re-generated on demand if needed, since the docking algorithm is deterministic. The final output size is 69 TB.

On average, the docking application requires a relatively small I/O bandwidth: 1.68 GB/s for reading and 0.12 GB/s for writing on the Marconi100 machine, while 2.53 GB/s for reading and 0.18 GB/s for writing on the HPC5 machine. Despite this, the I/O configuration must be carefully tuned (Paragraph 1.7.2) to avoid scaling issues on large systems due to the sheer number of MPI tasks performing I/O operations [157].

1.7.5 Intra-node Scaling

The availability of multiple dock and score algorithm implementations grants access to heterogeneous resources. Figure 1.17 shows the time elapsed by the CPP and CUDA workers to perform the docking operation with different ligand characteristics. It can be noticed that the CUDA implementation has, on average, a 65 × speedup concerning the CPU version. Therefore using only the CUDA implementation is the most efficient solution. However, the relation between the number of CUDA and CPP workers (Paragraph 1.7.2) and the application throughput is not trivial. Table 1.4 shows the application throughput in terms of docked ligands per second by varying the number of CUDA and CPP workers when the application is deployed on a Marconi100 node equipped with 32 × IBM POWER9 AC922 cores (128 × hardware threads) and 4 × NVIDIA V100 GPU. The application binds each CUDA worker to a single GPU in a round-robin fashion. For example, when using 24 × CUDA workers, 6 × threads feed data and retrieve the results for

Table 1.4: The throughput reached per node and per machine for each binding site evaluated in the experiment. The NSP13ortho binding site has been partially computed on both machines.

Binding site	Thr (ligands/s/node)	Thr (ligands/s)	HPC machine
PLPRO	2496	1996800	M100
SPIKEACE	2498	1998400	M100
NS12thumb	2499	1999200	M100
NS13palm	2486	1988800	M100
3CL	2427	1941600	M100
NSP13allo	2498	1998400	M100
Nprot	2010	3015000	HPC5
NSP16	1980	2970000	HPC5
NSP3	1969	2953500	HPC5
NSP6	1985	2977500	HPC5
NSP12ortho	2001	3001500	HPC5
NSP14	1965	2947500	HPC5
NSP9	1996	2994000	HPC5
NSP15	1990	2985000	HPC5
NSP13ortho	2454/1987	1963200/2980500	M100/HPC5

each GPU in the node. It can be noticed how the application reaches peak performance for a high number of CUDA workers. Moreover, overall performance decreases when the number of CPP workers is increased to match the number of hardware threads. This behavior implies that, in our case study, it is better to use CPUs to support accelerators and I/O operations rather than contribute to the elaboration itself. Furthermore, to benefit most from a GPU, using a single CUDA worker is not enough. This is the expected result, since the CUDA worker needs to parse the ligand description and initialize the related data structures before launching any CUDA kernel. Thus, those overheads can be hidden and GPU fully utilized using more CUDA workers. To perform this analysis, the *Commercial Compound MW<330* dataset from the MEDIATE database has been chosen (already detailed in Paragraph 1.6.3).

1.7.6 HPC System Scale-out

For this experiment, the binding sites are evaluated sequentially. With this configuration, a job array of ~3400 jobs for every binding site is used, where each job is composed of 32 × MPI processes that last for about 5 minutes and targets a single binding site.

Table 1.4 reports for each binding site the average throughput of a node and the whole machine. On average, a single node's throughput is $2.4k$

ligands per second on Marconi100 and $2k$ ligands per second on HPC5. Both supercomputer nodes are equipped with $4\times$ NVIDIA V100 GPUs. Since most of the application's throughput comes from CUDA-accelerated kernels, the performance difference between the two nodes is unexpected. However, there is a big difference in the architectures of the Marconi100 node and the HPC5 node, that is how the GPUs and CPUs are connected: Marconi100 has NVLINK, while HPC5 relies on standard PCIe. In our case study, NVLINK is better at transferring the ligand inputs.

By taking into account the throughput measured on a single Marconi100 node while running live in production, it can be noticed that values are similar to the results obtained while fine-tuning the application (i.e., the number of CPU and GPU workers). The Exscalate platform exploited all the available resources, reaching a combined throughput of $5M$ ligands per second on both supercomputers.

Finally, Figure 1.21 shows the execution track of the job arrays on two different proteins running on the two supercomputers. Also, in this case, the difference in performance is visible, which is not due to the target protein but mainly to the different node architecture. Despite the performance being stable across the workload, the difference in throughput between the jobs is due to the average complexity (i.e., number of atoms and rotatable bonds) of the ligands included in the sub-reactions associated with the jobs. This can also be noticed by the similar profile of the plots on the two different machines. These results show how the input data strongly influences the throughput.

1.7.7 Data Pre/Post-processing

The main challenge of the experiment is to generate the chemical knowledge of the virtual screening. However, it requires a pre-processing phase to prepare the ligands: this phase is described in detail in Paragraph 1.7.2 and it must be performed only once as the same pre-processed chemical space is evaluated against all docking sites.

The experiment's output is a list of output files, each ranking the ligands according to the interaction strength with the target protein. Even if the output can be used as-is, for the sake of convenience a preliminary post-processing step is performed to join all the scores for the same ligand across all docking sites thus obtaining a single global table for the whole experiment.

The actual post-processing phase involves several steps aimed at obtaining statistical descriptors for the full score's distributions (mean, median, standard deviation, several percentiles); these descriptors are then used to extract the best-scoring compounds for each docking site to form the final released dataset. The computation has been carried out using a Dask distributed pipeline on the Marconi100 system. To identify the best prospect molecules taken into account, for each docking site, all the compounds scored

Table 1.5: Time required to complete the experiment's phases.

Phase	Time	Resources
Pre-processing	5 days	100 M100 nodes (no GPUs)
Dock & Score	60 hours	800 M100 + 1500 HPC5 nodes
Post-processing	5 days	19 M100 nodes (no GPUs)

higher than 3 standard deviations from the distribution's mean. The resulting data set, containing more than 570 million top-scoring compounds, is freely available[6]. Complete chemical analysis of the result dataset is presente in [62].

Table 1.5 summarizes the computational resources involved in each phase.

1.8 Conclusion

This chapter tackles the problem of virtual screening a large set of molecules, a representative problem for embarrassingly-parallel, task-based workloads on HPC systems. As discussed, this type of workload requires a radically different approach to GPU acceleration compared to the one used for tightly coupled (or *synchronous*) workloads, which are characteristic of classical HPC scientific applications. The problem is usually addressed by performing molecular docking of the candidate molecules in the protein pocket, which often requires large-scale computer simulations.

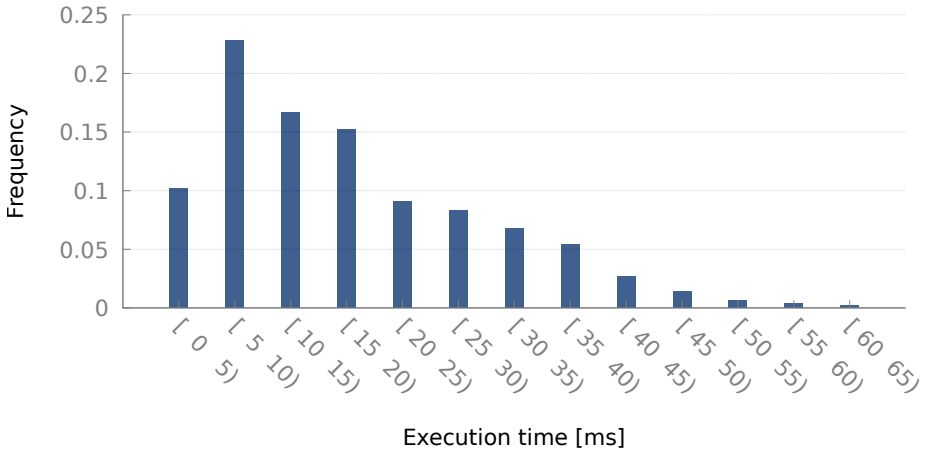
Concerning GPU efficient acceleration, the classical *latency*-optimized approach that spreads the computation of a single task (a ligand-protein pair in our case) across the device to minimize single task latency leads to satisfactory performance at the task level, but severely hinders overall time-to-solution. A *throughput*-optimized approach, on the other hand, even if it usually leads to a single-task latency increase, is extremely beneficial for GPU occupancy, leading to $5\times$ better throughput. It has been shown how, when dealing with embarrassingly-parallel, task-based workloads on GPU accelerators, the absolute execution latency of a single task is not a meaningful indicator of efficient acceleration, while increased GPU occupancy yields large improvements in overall kernel throughput in terms of completed tasks per second, even at the cost of worse task-level latency.

Section 1.7 covers how, in the context of urgent computing, extreme-scale virtual screening campaigns can lead to drug prospects for viral diseases in times of pandemics. By leveraging the GPU kernels discussed in this chapter, more than 70 billion of ligands against 15 binding sites of 12 viral proteins of SARS-CoV-2 were screened. The workflow scaled over two full HPC systems, Marconi100 at Cineca and HPC5 at ENI S.p.A., at the time

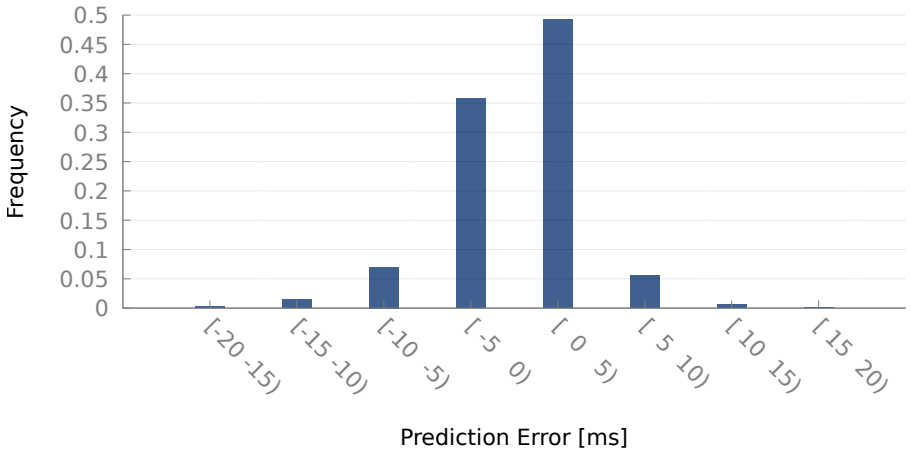
[6] https://mediate.exscalate4cov.eu

of the experiment the two most powerful supercomputers in Europe, to run a *one-trillion-docking experiment*[7] in 60 continuous hours of production. This is, to date, the largest virtual screening experiment ever attempted.

[7] https://1trilliondock.exscalate4cov.eu/

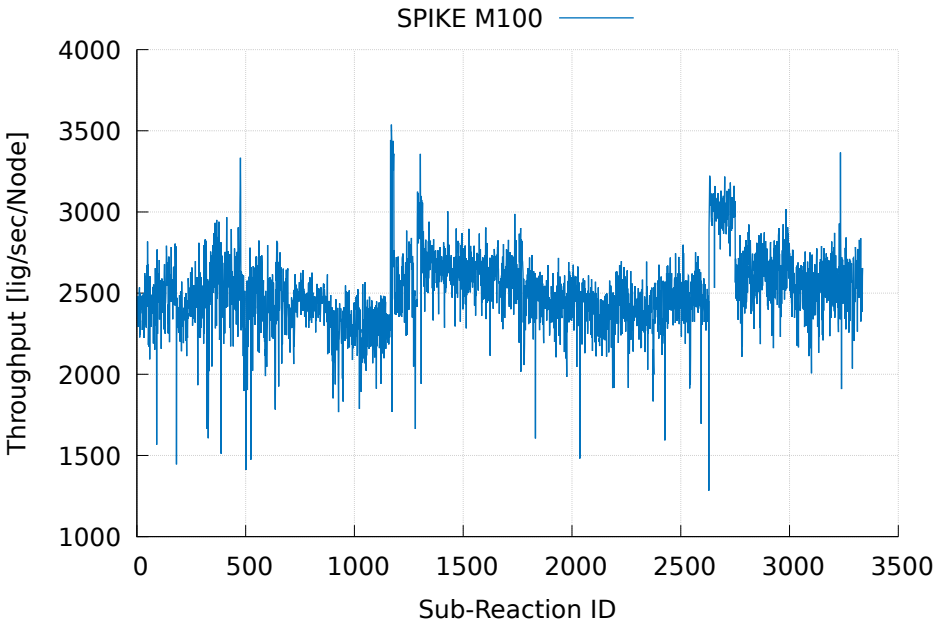


(a) Measured docking time

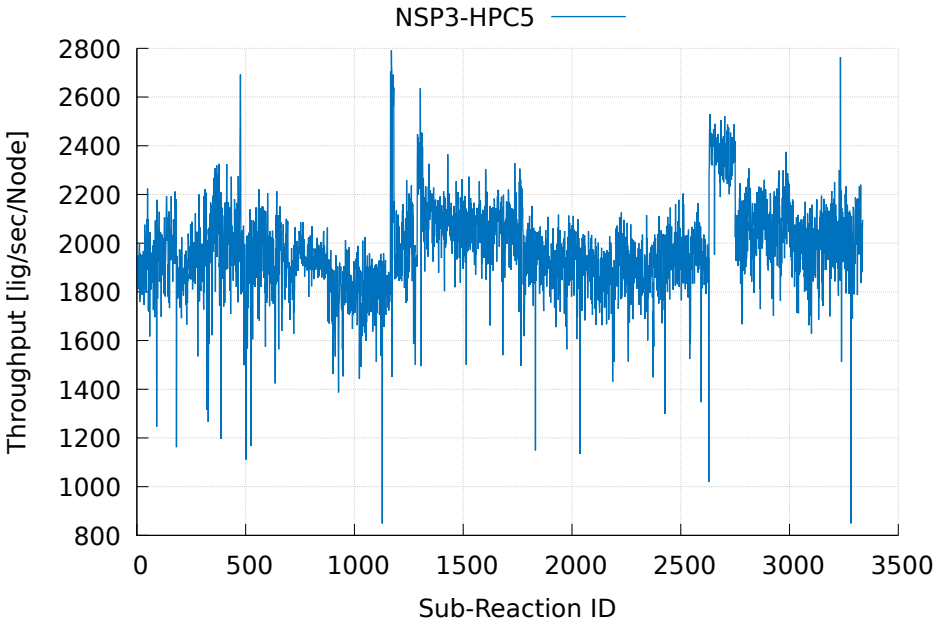


(b) Prediction error

Figure 1.20: Frequency distribution of the measured docking time, using the CUDA implementation, and its prediction error. Values with a frequency lower than 0.001 are discarded for conciseness purposes.



(a) SpikeACE on Marconi100



(b) NSP3 on HPC5

Figure 1.21: Execution track of two entire job arrays targeting two different protein pockets on the two different supercomputers.

Chapter 2

The Future of Sustainable HPC: RISC-V

From the perspective of ISAs, few established players have dominated the market in the last 20 years. With minor exceptions [158], the same players driven both the *multicore era* and the following *manycore era* with widespread solutions like the Intel Xeon Phi [159]. This trend is reflected in the TOP500: the largest supercomputer systems across the globe have been powered by a small set of processor architectures.

Since the late 1990s, the most significant evolution in the ISA space has been the universal adoption of SIMD instructions. Starting with Intel MMX in 1997, the introduction of vector capabilities into established architectures has been a steady trend that evolved with conservative, incremental improvements concerning lane data types, vector register size, and complex instructions. This trend of conservative evolution of industry-standard ISAs ended with the advent of ML. The exploding demand for computing power driven by the ML market and the technological hurdles posed by the *end of Moore's law* are driving the integration of an increasing set of application-specific instructions into industry-standard ISA: the common goal is to optimize performance and energy efficiency for DL workloads. New number formats became paramount for DL training and inference to the point that all major ISAs introduced new instructions to handle non-IEEE floating point operands like *bfloat16*. In the same way, mixed precision instructions have been introduced by all major processor vendors, especially expanding reductions to keep error accumulation under control when dealing with small-precision operands. Moreover, matrix-vector and matrix-matrix instructions backed by multi-dimensional SIMD registers have been adopted by an increasing number of established ISAs (e.g., Intel Advanced Matrix Extensions [160], ARM Scalable Matrix Extensions [45], PowerPC Matrix Engine [43]).

At the same time, the innovation rate in the space of accelerator archi-

tectures is even more extreme, with a proper *Cambrian explosion* of vertical hardware designs with features that must be exposed either by extending an existing ISA or by designing a brand new one: while the former approach brings licensing and software ecosystem issues that must be addressed, the latter requires the *bringup* of full software and hardware integration stacks.

The need for a more flexible solution for computing innovation is proven by the rise of RISC-V, an open, modular, extensible, and royalty-free ISA. Being designed from scratch to be naturally extensible without breaking the existing software ecosystem, RISC-V has become the platform of choice for architecture research [46, 47, 48, 49] and the market enabler for an increasing set of vendors who are exploring novel concepts in accelerator and processor design [50, 51, 52, 53, 54, 55].

Albeit rapidly growing, the extreme flexibility of an open-ended architecture is posing unique challenges to both software implementors and system integrators: they have to deal with a combinatorial explosion in possible ISA choices. From the perspective of operating systems and scientific libraries, there is no concrete RISC-V target since each processor can provide a custom set of standard and custom extensions. The issue is much more severe concerning compilers that have to consider a growing set of optional ISA extensions, often interacting in multiple ways. The outcomes are so severe that pragmatic details like *long and unwieldy ISA strings* [161] became real concerns. This issue is recognized by RISC-V International itself, which recently tried to impose some structure with the introduction of *architecture profiles* [162, 161], predefined sets of extensions that should be fully implemented by a platform. The processor market is already experiencing concrete challenges like actual products being shipped with different versions of the same extension, where a mix of pre-ratified and ratified implementations of must coexist in the same system. A particularly severe instance of this issue involves the RISC-V vector extension (RVV), where available products are sporting both pre-ratification [163, 164] and ratified versions.

While RISC-V brings key advantages to the prospects of computing architectures, it poses new challenges like extreme ISA fragmentation and fast evolution, issues that are yet to be solved by software and system integration stacks. For these reasons, the question of whether RISC-V will be sustainable in a post-exascale HPC system is still unanswered. In an attempt to answer this question, a collaborative effort between both the research groups led by Prof. Luca Benini and Prof. Andrea Bartolini at Università di Bologna, CINECA and E4 Computer Engineering, designed, built, and deployed *Monte Cimone* [65, 66, 63], the world's first fully functional RISC-V HPC prototype. The goal of *Monte Cimone* is to assess the readiness of the whole software-hardware integration stack, from interconnects to performance monitoring infrastructures, from scientific libraries to compilers. Since heterogeneous systems are obiquitous in TOP500 (see Chapter 1), issues related to accelerated systems must be taken into account. For this

reason, the *Monte Cimone* testbed has been augmented with an array of RISC-V accelerators, from the ML streaming accelerator Occamy [165, 166, 19] to the heterogeneous EPAC [55] released by the European Processor Initiative [167].

2.1 The *Monte Cimone* Experimental System

Both academia and industry are aggressively pursuing architectural innovation to develop HPC systems to mitigate the efficiency limitations of traditional architectures. ISAs have to evolve rapidly to sustain this domain-driven architectural evolution, and the advent of the RISC-V open, royalty-free, and extensible ISA has been a major step toward accelerating innovation in this area. An additional advantage of RISC-V concerning the dominant proprietary ISAs (x86 and ARM) is that it is maintained by a global non-for-profit foundation with members across the world, ensuring a high degree of neutrality concerning geopolitical tensions and their technology downfalls.

Currently, high-performance 64-bit (`RV64`) RISC-V processors and accelerator chips are being designed, promising prototypes are demonstrated in numerous publications [2], and products are announced at a fast cadence [168, 169]. Thus, it is reasonable to expect that high-performance chips based on RISC-V will be available as production silicon within the next couple of years. However, building a HPC system requires significantly more than just high-performance chips. Many think that the RISC-V software stack and system platform are extremely immature, and will need several additional years of development effort before full applications could be run, benchmarked and optimized on a RISC-V-based HPC system. The goal of this work is to dispel this overly conservative notion.

In this chapter, we present *Monte Cimone*, the first physical prototype and testbed of a complete RISC-V (`RV64`) compute cluster, integrating not only all the key hardware elements besides processors, namely main memory, non-volatile storage and interconnect but also a complete software environment for HPC, as well as a full-featured system monitoring infrastructure. Further, we demonstrate that it is possible to run real-life HPC applications on *Monte Cimone* today. Even though achieving strong floating point performance will be possible only with upcoming high-performance chips, we achieved the following milestones:

- we designed and set up the first RISC-V-based cluster containing eight computing nodes enclosed in four computing blades. Each computing node is based on the U740 SoC from SiFive and integrates four U74 `RV64GCB` application cores, running up to 1.2 GHz and 16 GB of DDR4, 1 TB node-local NVME storage, and PCIe expansion cards. The cluster is connected to a login node and master node running the job scheduler, network file system, and system management software;

- we ported and assessed the maturity of a HPC software stack composed of (i) SLURM job scheduler, NFS filesystem, LDAP server, Spack package manager, (ii) compilers toolchains, scientific and communication libraries, (iii) a set of HPC benchmarks, and applications, (iv) the ExaMon datacenter automation and monitoring framework. The full software stack usually adopted for a production TOP500 machine is available and functional;

- we measured the efficiency of the HPL benchmark and STREAM benchmark with the toolchain and libraries installed by Spack. We compared the attained results against the one obtained for other RISC-based TOP500 supercomputers (namely, Fugaku and Alps). We build the HPL and STREAM benchmark following the same approach for the *Monte Cimone* cluster on two state-of-the-art computing nodes, namely the Marconi100 [155] node based on a `ppc64le` IBM Power9 CPU, and the Armida [170] node based on an `armv8a` Marvell ThunderX2 CPU. We compared the attained floating-point unit (FPU) utilization from both HPC systems as a metric of efficiency versus *Monte Cimone* while keeping the same benchmarking boundary conditions (e.g., vanilla, unoptimized libraries, and software stack deployed via a popular package manager). Results show that upstream HPL achieved 46.5 % utilization on *Monte Cimone*, the Marconi100 and Armida compute nodes achieved 59.7 % and 65.79 % of their peak respectively. The *Monte Cimone* node achieves slightly lower FPU utilization but in the range with state of the art. When running an unoptimized STREAM benchmark, *Monte Cimone* obtained just the 15.5 % of the peak bandwidth, while Marconi100 and Armida obtained an efficiency of 48.2 % and 63.21 % respectively, pointing to significant margins for improvement in application and software stack tuning to the hardware target;

- we extended the ExaMon monitoring framework [171] to monitor the *Monte Cimone* cluster. We characterized the power consumption of various applications executed on *Monte Cimone*. We reported a power consumption of 4.81 W in idle, composed of 64 % of core power, 13 % related to DDR and 23 % of related to PCI subsystem. During CPU intensive benchmark runs on the SiFive Freedom U740 SoC we reported a power consumption of 5.935 W, composed of 69 % of core power, 14 % related to DDR and 18 % related to PCI subsystem. By profiling the power consumption of the core complex during the boot process, we measured a 0.981 W of leakage-only power (32 % of the Idle power) and measured 0.514 W of power consumed by the operating system during idle (17 % of the Idle power) and a remaining 1.577 W of dynamic and clock tree power, accounting for the 51 % of the core idle power. In addition to providing a detailed analysis of power consumption,

ExaMon enabled us to detect and mitigate thermal design issues in the early cluster physical design;

- we extended the SiFive partition with an accelerated partition based on a high-performance, SIMD-capable host CPU (the XuanTie C920) augmented with the Occamy RISC-V accelerator connected via PCIe. This experiment provides the world's first fully RISC-V heterogeneous cluster and a valuable development vehicle for system software, scientific libraries, and compilers.

2.2 State-of-the-art

The most recent successful effort to introduce a new ISA to HPC has involved the ARM ISA. Bringing the ARM ISA and software ecosystem to HPC maturity has required almost a decade and several funding rounds: the Mont-Blanc European project series started in 2011, leading to the first ARM-based HPC cluster deployed in 2015 [172], based on a SoCs developed for the embedded computing market. Notably, Fugaku [173] the fastest supercomputer in the TOP500 list published in June 2020, is based on an ARM ISA, and achieves more than 400 PFLOP/s. Further, high-performance ARM-based SIMD processors are being adopted in servers and data centers worldwide. We observe that it took approximately a decade for ARM to become a strong player in these highly competitive markets, even though x86 is still the dominant architecture in HPC and cloud.

The RISC-V ISA was conceived just a decade ago, thus, clearly, its market penetration is much smaller than the incumbent ARM and x86 ISAs. Today, only a few 64-bit RISC-V (`RV64G` ISA) SoCs are available commercially, and none is in volume production for HPC or performance servers. Nevertheless, several high-performance RISC-V processors have been announced for general-purpose and accelerated computing markets [174, 175, 176]. In addition, a few research prototypes have been presented in recent literature that demonstrate on silicon the technical feasibility and competitiveness of high-performance RISC-V computing engines [177, 50, 178]. Furthermore, the European Processor Initiative (EPI) launched in 2019 is funding a major research thrust to develop RISC-V-based accelerators for HPC [167]. One of the many outputs of the EPI project is the EPAC [55] accelerator, based on a capable RISC-V out-of-order (OoO) vector engine.

Among the `RV64G` chips available in low volumes on the market, for our cluster we chose the SiFive Freedom U740 SoC, featuring a 64-bit dual-issue, superscalar RISC-V U7 core complex configured with four U74 cores and one S7 core, an integrated high speed DDR4 memory controller and PCIe Gen3 channels and standard peripherals. The availability of a main memory interface with reasonable performance and a PCIe root complex for connecting fast storage, peripherals, and accelerators, makes this SoC a good

basis for exploring the deployment of RISC-V processors in a scalable cluster and working on the software stack. Still, the performance and number of cores in the SoC is insufficient to achieve performance comparable to mature ARM and x86 cores.

The maturity of the software ecosystem around RISC-V has been growing at a very fast rate. A reasonably complete snapshot of major software packages available for RISC-V is maintained by RISC-V International [179]. While the list is not complete due to the very fast growth of the RISC-V developer community, it's clear how porting efforts have mainly focused on embedded and AI applications. An HPC special interest group (SIG) for RISC-V was founded in 2019 [180]. However, to the best of our knowledge, the demonstration of a complete software stack and HPC applications running on real hardware on RISC-V nodes in a multi-blade cluster is still missing. *Monte Cimone* aims at filling this gap.

In addition to libraries and tools for HPC application deployment, a production-ready HPC system must support fine-grain utilization, performance, and power monitoring of the computing resources to enable efficient computing, power, thermal management, and anomaly detection for reliability. Recently, several works have been proposed to extend the power monitoring attainable from the voltage regulator modules leveraging shunt resistors, current probes, and out-of-band telemetry [181]. In addition, Operational Data Analytics (ODA) [182] has been introduced, focusing on monitoring and managing large-scale HPC installations. Vertical solutions encompassing all layers (from data gathering and storage to processing and analysis) have been proposed in this area. Notable examples are OMNI [183], an infrastructure for extreme-scale operational data collection, and ExaMon [171], an ODA infrastructure leveraging: i) distributed sensing plugins (including node-level metrics, processing elements performance metrics, dedicated fine-grain power monitoring meters, facility data); ii) scalable storage backends; iii) visualization and analytics targetting anomaly detection and intrusion detection systems. Current ODA tools are available only for the dominant ARM and x86 environments. In this chapter we advance state of the art demonstaring a fully-operational port of the ExaMon ODA infrastructure to the *Monte Cimone* RISC-V cluster.

2.3 Hardware Architecture

Monte Cimone is based on the SiFive Freedom U740 RISC-V SoC HiFive Unmatched board integrated in an HPC node form factor (Figure 2.2). The board follows the Mini-ITX standard with a size of 170 mm × 170 mm. Each board features one SiFive Freedom U740 SoC, 16 GB of 64-bit DDR4 memory operating up to 1866 MT/s and high-speed interconnects with PCIe Gen3 x16 (but it's limited to x8 lanes), one Gigabit Ethernet, and four USB 3.2

Figure 2.1: The custom-built E4 RV007 Server Blade is based on a dual SiFive Freedom U740 SoC. The form factor is 4.44 cm (1 RackUnit) high, 42.5 cm wide, 40 cm deep. A dedicated power supply powers each board to account for future PCIe expansions.

Gen1.

The E4 RV007 blade prototype system, specifically designed to be the *Monte Cimone* building block, is a dual-board platform server, with a form factor of 4.44 cm (1 RackUnit) high, 42.5 cm wide, 40 cm deep (Figure 2.1). Two 250 W power supplies, one for each board (compute node), are installed inside the case. This allows turning on every compute node individually, and makes the system ready with abundant power headroom for future expansions with PCIe accelerators and network cards.

In the RV007 node deployed in *Monte Cimone* the M.2 expansion slot is occupied by a 1 TB NVME SSD storing the operating system (OS). The available Micro SD card slot is used only for the UEFI boot process.

The FU740-C000 is a Linux-capable SoC powered by SiFive U74-MC, the first (to the best of our knowledge) commercially available multi-core RISC-V core complex. It includes a single 64-bit S7 RISC-V (monitor) core with a high-performance dual-issue in-order execution pipeline and a peak sustainable execution rate of two instructions per clock cycle. It implements the `RV64IMAC` ISA. The FU740-C000 also features four 64-bit U74 RISC-V (application) cores, each having a high-performance dual-issue in-order exe-

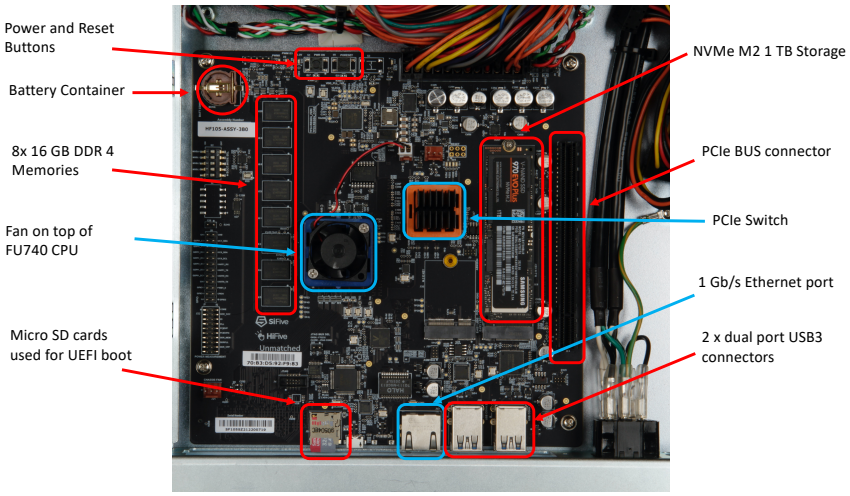


Figure 2.2: The HiFive Unmatched board based on the SiFive Freedom U740 SoC. The form factor follows the Mini-ITX standard (170 mm × 170 mm).

cution pipeline and a peak sustainable execution rate of two instructions per clock cycle. The U74 application core implements the `RV64IMAFDC` ISA. An essential feature for an HPC system is *observability*: the S7 processor core provides a hardware performance monitoring (HPM) unit. It supports two classes of counters: fixed-function and event programmable counters. These classes consist of a set of fixed counters and their counter-enable registers, as well as a set of event-programmable counters and their event selector registers. The registers are available to control the behaviour of the counters. We extended the Linux `perf_events` interface to be able to collect performance events via the Linux tooling (e.g., the `perf` command). Table 2.2 reports a complete list of the available events.

The SiFive Freedom board features a Microsemi VSC8541 chip to interconnect the SiFive Freedom U740 SoC with a single port gigabit Ethernet copper interface. Moreover, we equipped two of the compute nodes with an InfiniBand (IB) host channel adapter (HCA) widely used in large-scale HPC systems. We target an IB FDR HCA (56 Gbit/sec) to leverage RDMA communications among different nodes to improve the network throughput and the communication latency. We used a Mellanox ConnectX-4 FDR HCA interconnect through the PCIe interface on the compute node. This HCA supports 8 × PCIe Gen3 lanes. The first experimental results show that the kernel can recognize the device driver and mount the kernel module to man-

Package	Version
`gcc`	10.3.0
`openmpi`	4.1.1
`openblas`	0.3.18
`fftw`	3.3.10
`netlib-lapack`	3.9.1
`netlib-scalapack`	2.1.0
`hpl`	2.3
`stream`	5.10
`quantumESPRESSO`	6.8

Table 2.1: User-facing software stack deployed on *Monte Cimone*.

age the Mellanox OFED stack. We cannot use all the RDMA capabilities of the HCA due to device driver incompatibilities of the software stack and the kernel driver. Nevertheless, we successfully ran an IB `ping` test between two boards and between a board and an HPC server, showing that full IB support could be feasible.

In addition, the SiFive Freedom U740 SoC features 7 separate power rails, including the core complex, IOs, PLLs, DDR and PCIe subsystems. The HiFive Unmatched board implements separated shunt resistors in series with each of the SiFive U740 power rails and for the onboard memory banks [184]. We characterized the power consumption of the system under test exploiting the set of nine power lines available onboard with embedded shunt resistor for current monitoring.

2.4 Software Environment

Since our goal was to build a software environment as close as possible to a production HPC cluster, we leveraged the Spack [185] package manager to deploy the full software stack (Table 2.1) and make it available to all system users via environment modules [186]. Actual Spack architecture and micro-architecture support, in the form of platform-specific toolchain flags, is provided by the `archspec` [187] module. Explicit support for the `linux-sifive-u74mc` target triple was already present (`archspec` version 0.1.3) and tested to be working without modifications. The user-facing software stack installed successfully via Spack (version 0.17.0) and presented to users is listed in Table 2.1 (transitive dependencies omitted for brevity). All the nodes are running upstream Ubuntu 21.04 deployed from `riscv64` server images without modifications and mount a remote NFS.

2.4.1 HPC Software

We ported on *Monte Cimone* all the essential services needed to run HPC workloads in a production environment, namely NFS, LDAP, and the SLURM job scheduler. Porting all the necessary software to RISC-V was relatively straightforward, and we can hence claim that there is no obstacle in exposing *Monte Cimone* as a computing resource in a HPC facility. However, deployment in a data center requires integrating *Monte Cimone* within a holistic monitoring framework. For that purpose, we use the ExaMon [171] framework.

2.4.2 Power Monitoring Infrastructure

The typical configuration of ExaMon consists of installing plugins dedicated to data sampling, a broker for transport layer management and a database for storage. For *Monte Cimone* cluster both broker and database are installed in their basic configuration on a master node, while plugins have been specifically developed for *Monte Cimone* and deployed on the compute nodes. As a first step, we created a dedicated version of the `pmu_pub` [188] plugin to acquire the performance counters available in the Linux OS through the `perf_events` interface. In the current version of the Linux kernel tree, only the `instret` (number of retired instructions) and `cycle` (number of cycles) counters are exposed for the RISC-V architecture. By default, the remaining programmable counters available on the hardware performance monitoring (HPM) unit of the U740 SoC are disabled at boot time [184]. We have, therefore, modified the U-Boot bootloader to enable and program all available counters (Table 2.2). The counters are sampled for each core in user-mode by the `pmu_pub` plugin at regular intervals (2 Hz), and values are published for collection.

A second plugin has been installed and configured to collect operating system statistics, `stats_pub`. This plugin mainly accesses the `sysfs` and `procfs` filesystems to get useful metrics about system resources such as load, CPU usage, memory usage, and network bandwidth. In particular, the HiFive Unmatched board is equipped with three thermal sensors dedicated respectively to the SoC, the motherboard, and the NVME SSD. This plugin samples sensors data via the Linux `hwmon sysfs` with a frequency of 0.2 Hz.

Finally, data collected for each board is published via Grafana [171] in the same way a TOP500 system like Marconi100 keeps track of its operational data.

2.5 Assessment Experiments

To be able to both stress-test and characterize the system, we performed an array of experiments with the goal of fixing early design issues and driving

Event [Linux `perf_events` identifier]	
`cycle`	`floating_point_fused_multiply_add_retired`
`instret`	`floating_point_division_or_square_root_retired`
`integer_load_instruction_retired`	`other_floating_point_instruction_retired`
`integer_store_instruction_retired`	`instruction_cache_itim_busy`
`system_instruction_retired`	`data_cache_dtim_busy`
`conditional_branch_retired`	`branch_direction_misprediction`
`jal_instruction_retired`	`branch_jump_target_misprediction`
`jalr_instruction_retired`	`pipeline_flush_from_other_event`
`integer_arithmetic_instruction_retired`	`integer_multiplication_interlock`
`integer_multiplication_instruction_retired`	`floating_point interlock`
`integer_division_instruction_retired`	`instruction_cache_miss`
`floating_point_load_instruction_retired`	`memory_mapped_i_o_access`
`floating_point_store_instruction_retired`	`data_cache_write-back`
`floating_point_addition_retired`	`instruction_tlb_miss`
`floating_point_multiplication_retired`	`data_tlb_miss`

Table 2.2: Performance monitoring events of the SiFive Freedom U740 SoC exposed to the Linux `perf_events` interface by our custom `pmu_pub` plugin.

future evolution. Of particular importance is assessing the maturity of the HPC software stack on RISC-V systems. In Paragraph 2.5.1, we focus on the details of software stack tests. We also carried out extensive thermal and power experiments [65] that were instrumental in the early design phase; all measurements were performed via the ExaMon monitoring subsystem. In particular, thermal experiments highlighed several issues with the initial chassis layout and allowed further optimization, especially concerning insufficient heat dissipation from the power supplies due to flaws in the airflow design. Power experiments allowed detailed characterization of the SiFive Freedom U740 SoC for different workloads, measuring 4.81 W in idle, with 64 % due to core power (32 % of leakage power, 51 % dynamic and clock tree power and 17 % by the OS workload), 13 % related to DDR and 23 % to the PCI subsystem. The power consumption increases to 5.935 W under CPU-intensive workloads.

2.5.1 HPC Applications Performance

Considering the peak theoretical value of 1 GFLOP/s per core, inferred from the micro-architecture specification [184], leading to a 4 GFLOP/s peak value for a single chip, the upstream HPL [57] benchmark (built on top of the software stack presented in Section 2.4) reached a sustained value of 1.86 ± 0.04 GFLOP/s on a single node (on a `N=40704` and `NB=192` HPL configuration and a total runtime of $24\,105 \pm 587$ s; this amounts to 46.5 % of the theoretical peak, a result we deem to be promising considering the upstream, unmodified software stack used in this phase. The same experiment was run on both the Marconi100 [155] system at Cineca and the Armida [170] system at E4. Both upstream software stack (and no vendor libraries) and MPI topology (1 MPI task per physical core) were the same. This experiment obtained 59.7 % and 65.79 % of a single node's CPU-only theoretical peak respectively, a

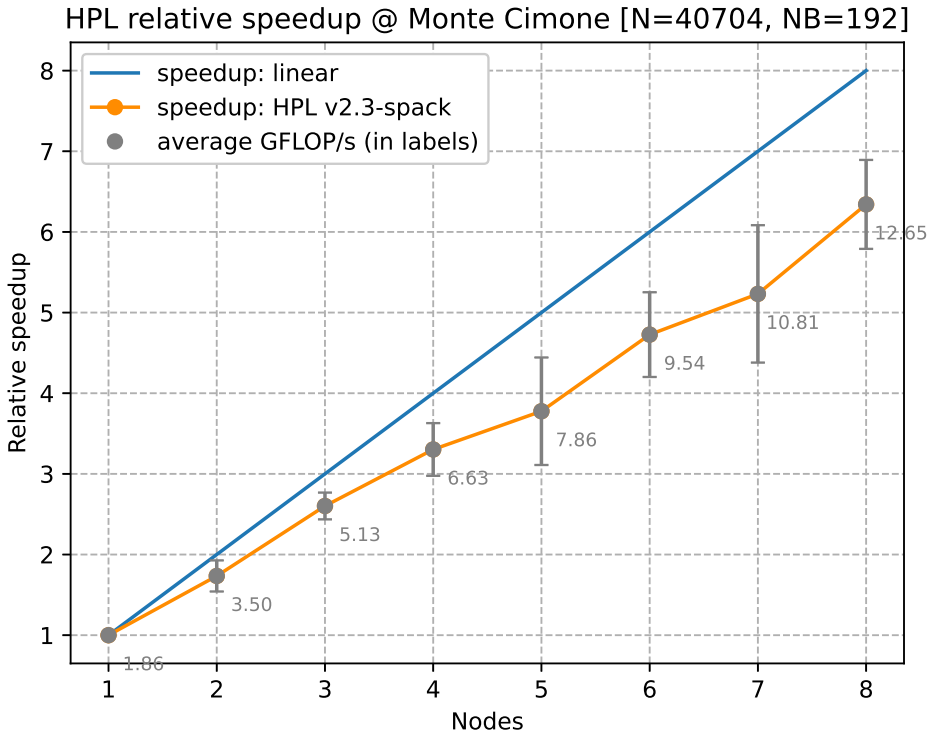


Figure 2.3: HPL strong scaling tests on *Monte Cimone*. Average attained throughput values are shown in labels. Standard deviations are calculated on 10 repetitions.

result that is comparable to what we observed on *Monte Cimone*. The same HPL configuration has been used to carry out a *Monte Cimone* full-machine benchmark experiment leveraging the 1 GB/s network currently available, reaching a sustained value of 12.65 ± 0.52 GFLOP/s using all of the eight nodes (with a total runtime of 3548 ± 136 s); this amounts to 39.5 % of the entire machine's theoretical peak and to 85 % of the extrapolated attainable peak in case of perfect linear scaling from the single-node case. Relative speedup obtained during the HPL strong scaling experiment are shown in Figure 2.3. Again, these results are promising and deserving both further optimization on the software side and tuning (or technology upgrade) on the interconnect side.

The STREAM [189] benchmark has been used to measure the attainable memory bandwidth on a single node. Out of the peak 7760 MB/s [184], a 4-thread experiment measured the values shown in Table 2.3. No topology configuration was needed due to the node being a uniform memory access (UMA) system. We consider the results attained via upstream, unmodified STREAM unsatisfactory: the results on *Monte Cimone* show an attained bandwidth of no more than 15.5 % of the available peak bandwidth. The

Test	STREAM.DDR	STREAM.L2
	1945.5 MiB/s [MB/s]	1.1 MiB/s [MB/s]
copy	1206 ± 3.26	7079 ± 2.11
scale	1025 ± 4.94	3558 ± 3.72
add	1124 ± 4.93	4380 ± 3.72
triad	1122 ± 5.63	4365 ± 3.56

Table 2.3: STREAM benchmark results for four threads on a single SiFive Freedom U740 SoC.

same experiment involving an upstream, unoptimized STREAM benchmark ran on both Marconi100 [155] and Armida [170] (using the same topology with 1 OpenMP thread per physical core) attained 48.2 % and 63.21 % of the peak bandwith respectively, suggesting that a result higher than the lower quartile should be easily attained with little to no effort. This observation is worth of further experimentation, in particular:
(i) the L2 prefetcher provided by the micro-architecture[184], being able to track up to eight streams per core, should be perfectly capable of reducing the gap between the two experiments shown in Table 2.3 (DDR-bound and L2-bound) given the large degree of spatial and temporal locality shown by the STREAM memory access patterns. Further analysis is needed to understand how the prefetcher is currently operating and the modifications needed to leverage it properly; (ii) the overall data size used by STREAM is currently limited by the RISC-V code model. The `medany` code model used by `RV64` requires that every linked symbol resides within a ± 2 GiB range from the `pc` register [184, 190]. Since the upstream, unmodified STREAM benchmark uses statically-sized data arrays in a single translation unit preventing the linker to perform *relaxed* relocations, their overall size cannot exceed 2 GiB. Further experiments on available workarounds for the absence of a *large code model* [191] and modifications to the STREAM source itself to overcome this limitation are needed;
(iii) while the architecture provides both the `Zba` and `Zbb` RISC-V bit manipulation standard extensions[184], the upstream GCC 10.3.0 toolchain isn't capable of emitting them nor the underlying GNU `as` assembler (shipped with GNU Binutils 2.36.1) is able to assemble them properly. Experiments with the latest upstream GCC version (*minimal* support for bit manipulations code generation landed in GCC 12 [192]) and the upstream development version of GNU Binutils (patches already merged [193], expected to be shipped with GNU Binutils 2.37.x) are needed to assess its impact on current STREAM measurements.

Regarding user applications, we carried out benchmarks for the quantum-ESPRESSO [64] suite, in particular using its LAX test driver, compiled with

OpenMPI, that performs a blocked (and optionally distributed) matrix diagonalization as a benchmark representative of the full-scale application workload. For a 512^2 input matrix, we obtained a value of 1.44 ± 0.05 GFLOP/s (36 % of the theoretical FPU efficiency) on a single node over a total test duration of 37.40 ± 0.14 s.

2.6 Heterogeneous HPC on RISC-V: Accelerating *Monte Cimone*

Deploying heterogeneous systems is the HPC trend observed in the TOP500 list. As discussed in Chapter 1, the vast majority of large-scale systems adopted nodes built around a capable host processor augmented with a diverse array of throughput-oriented floating point accelerators, attached either via PCIe or vendor-specific connections. In order to explore the viability of RISC-V for the future of HPC, building an heterogeneous prototype cluster is essential.

As a first step toward this goal, we extended the already deployed U740 partition of *Monte Cimone* with new nodes based on the Sophon SG2042 processor. The Sophon SG2042[1] is a high-performance CPU built on 64 XuanTie C920 cores, an evolution of the previous XuanTie C910 [50] core. These 64-bit RISC-V cores are designed for compute-intensive workloads, sporting a modern 12-stage pipeline with out-of-order execution and 5-way superscalar execution. The C920 implements the `RV64GCV` architecture (a shorthand for `RV64MAFDCV`): apart from single (`F`) and double (`D`) scalar floating point arithmetics, it provides an implementation of the RISC-V vector extension (`V`, also commonly referred to as `RVV`) ISA on a 128-bit vector register file. The `RVV` ISA implemented by XuanTie C920 is the pre-ratification version `0.7.1` instead of the ratified version `1.0`: this poses additional challenges on the software stack. None of the mainstream toolchains support `RVV` version `0.7.1`, so manually deploying the GCC fork[2] maintained by the CPU vendor is mandatory for efficient code generation. The memory hierarchy consists of separate 64 kB L1 caches for instructions and data per core, a 1 MB L2 cache shared within each four-core cluster, and a 64 MB of shared L3 cache. The SoC provides 32 PCIe Gen4 lanes, an essential feature to augment the system with accelerators. All the nodes Sophon SG2042 nodes are in the Milk-V Pioneer Box[3] form factor, all equipped with 128 GB of DDR4 memory.

With respect to the accelerated part of the system, we chose Occamy [165, 166, 19], a flexible, general-purpose, dual-chiplet system with two 16 GiB HBM2E stacks optimized for regular and irregular floating point intensive

[1] https://en.sophgo.com/sophon-u/product/introduce/sg2042.html
[2] https://github.com/XUANTIE-RV/xuantie-gnu-toolchain
[3] https://milkv.io/pioneer

workloads. Each chiplet integrates a CVA6 [194] RISC-V core and 216 lightweight Snitch [48] cores organized hierarchically in six groups, each containing four nine-core compute clusters. Snitch is a small, efficient, in-order, 32-bit RISC-V integer core with an accelerator port for augmenting its compute capabilities. Being optimized for floating point workloads, each Snitch core in Occamy is augmented with a large multi-precision FPU supporting both standard RISC-V scalar (`D`, `F` and `H`) and 64-bit-wide packed-SIMD custom instructions [195]. It supports a wide range of number formats: `FP64` (IEEE 754 [196] double precision), `FP32` (IEEE 754 single precision), `FP16` (IEEE 754 half precision), `FP16alt` (half precision alternate format with 8-bit exponent and 7-bit mantissa, also known as `bfloat16` [197]), `FP8` (custom quarter precision with 5-bit exponent and 2-bit mantissa), `FP8alt` (alternate format for custom quarter precision with 4-bit exponent and 3-bit mantissa). To provide energy-efficient computing, the Snitch core provides custom RISC-V extensions to allow the FPU accelerator to execute an instruction flow without relying on its small frontend integer core for fetch or decode, loads/stores and address calculations. Characteristics of the Snitch architecture and its novel features are discussed in detail in Section 3.2.

To bring up both the system and the user-space software stack, we synthesized part (2 full Snitch clusters in addition to the CVA6 host core) of the whole Occamy accelerator on the AMD Alveo™ U55C FPGA[4]. The synthesized accelerator has been then successfully connected to the Sophon SG2042 nodes via a PCIe Gen4 link. This accelerated partition is currently used as the bring-up testbed for device drivers and offload runtimes. We plan to use the Occamy-accelerated partition as the test-bed for the Snitch linear algebra compiler presented in Chapter 3.

2.7 RISC-V for HPC: Conclusion and Prospects

The new open and royalty-free RISC-V ISA is attracting interest across the computing continuum, from microcontrollers to supercomputers. High-performance RISC-V processors and accelerators have been announced, but RISC-V-based HPC systems will need a holistic co-design effort, spanning memory, storage hierarchy interconnects and full software stack. In this chapter, we presented *Monte Cimone*, a multi-blade computer prototype and hardware-software test-bed: it is, to the best of our knowledge, the first RISC-V cluster which is fully operational and supports a baseline HPC software stack, proving the maturity of the RISC-V ISA and the first generation of commercially available RISC-V components. The prototype is a heterogeneous, accelerated cluster based on two different partitions, seamlessly integrated through its HPC production software stack: a CPU-only partition

[4] https://www.amd.com/en/products/accelerators/alveo/u55c/a-u55c-p00g-pq-g.html

based on the SiFive Freedom U740 SoC, and an accelerated partition based on the XuanTie C920 processor augmented with a synthesized version of the Occamy accelerator. We also evaluated the support for IB network adapters which are recognized by the system but are not yet capable of supporting RDMA communication. We characterized in detail the power consumption of the SiFive Freedom U740 SoC for different workloads, measuring 4.81 W in idle, with 64 % due to core power (32 % of leakage power, 51 % dynamic and clock tree power and 17% by the OS workload), 13 % related to DDR and 23 % to the PCI subsystem. The power consumption increases to 5.935 W under CPU-intensive workloads. Furthermore, we ported the ExaMon ODA system on *Monte Cimone* and used it to detect thermal stability problems in the first configuration, which led to a thermal shutdown on the central node during the HPL run. We changed the enclosure design to provide higher airflow to mitigate the issue.

Monte Cimone does not aim to achieve strong floating point performance, but it was built with the purpose of *priming the pipe* and exploring the challenges of integrating a multi-node RISC-V cluster capable of providing an HPC production stack including interconnect, storage and power monitoring infrastructure on RISC-V hardware. We present the results of our hardware/software integration effort, which demonstrate a remarkable level of software and hardware readiness and maturity: this shows that the first-generation of RISC-V HPC machines may not be so far in the future.

Chapter 3

Multi-level SSA Compilers for RISC-V Accelerators

The need for energy efficiency affecting hardware designs is fueling the need for efficient software stacks. Each accelerator design that reaches the market comes with a wide variety of vendor-specific libraries that allow applications to efficiently leverage all the available domain-specific features in a straightforward way. A prime example of this trend is NVIDIA shipping a wide range of extremely optimized libraries for neural networks (cuDNN [198]), dense (cuBLAS) and sparse (cuSPARSE) linear algebra, tensor linear algebra for DL (TensorRT), tensor core computations (CUTLASS), and a wide variety of scientific acceleration libraries (e.g., for Fourier transforms, linear and non-linear solvers, etc.). All major accelerator vendors on the market (e.g., Intel with oneAPI, AMD with ROCm) adopt the same model: by providing high-performance libraries, they lower barriers for application developers to leverage their products efficiently. As a consequence, making the most effective use of vendor libraries has become a critical task for both HPC and ML developers. The need for careful usage of performance libraries combined with the sheer amount of native operators needed by mainstream ML frameworks[1] is becoming unsustainable [34]. It is clear that numerical computing needs full-stack approaches to deal with large corpora of operators, novel numerical formats [199, 197, 200, 201] and optimizations [30, 202] that are often combined in large parameter searches for extreme optimization: programming abstractions need maintainability and flexibility [203]. This sustainability issue is being tackled by means of a variety of novel compilation approaches [202]. Tensor-level IRs is used by XLA [32] and Glow [33] to transform tensor programs into predefined LLVM and CUDA operation templates (e.g., reductions, element-wise operations, etc.) using pattern matching. The polyhedral model [204] is used by Tensor Compre-

[1]For example, PyTorch v2.5.0 ships 2633 operators. Many of them rely on multiple overloads that dispatch to target-specific implementations.

hensions [28] to parameterize and automate the compilation of one or many DNN layers into LLVM and CUDA programs. Loop synthesis is used by Halide [29] and TVM [30] to transform tensor computations into loop nests that can be then manually optimized using user-defined transformations or parametric schedules. Other approaches focus on exposing fast, local memories directly into the programming model, e.g., Triton [34] has been built on the concept of tile-level operations.

Concerning compiler construction, traditional IRs based on a single, uniform level of abstraction proved themselves to be solid approaches for compilers that focused on relatively low-level frontend languages (i.e., C, C++) [35, 36]. Being fixed in their abstraction level, those representations usually settle on the greatest common set of features shared by all front-end languages, a set that often reduces to a handful of low-level, hardware-friendly concepts. Nevertheless, the need for efficient generated code forced compilers to introduce key features like fixed-size SIMD vectors[2] even though completely missing from front-end languages. These additions often require non-standard language extensions[3], an approach that, while being essential to support hardware evolution, present high engineering costs when data types become increasingly complex[4]. Moreover, ML programs are usually written in high-level, domain-specific languages where tensor algebra is a first-class citizens [37, 38]: to be able to sustain this evolution, compilation stacks are transitioning from the traditional low-level, fixed IRs to expressive, flexible representations.

Expanding the abstraction levels of an IR is not a novel concept. One notable example adopted in a production compiler is WHIRL [205], a multi-level IR that allows the input program to be represented at different levels of abstraction, each one designed to make a specific set of analyses and transformations more effective. The key concept highlighted by WHIRL authors is the importance of *designing the most efficient form of representation for each optimization phase to work on* [205]. The IR spans from very high-level forms designed to perform reasoning close to the input program semantics (e.g., MPI [206] communications optimizations or structure layout transformations [207]), to lower levels progressively close to the target machine. The effectiveness of this approach is witnessed by its traction during the dawn of the first two great computing disruptions of the last 20 years, namely *the multicore era* [41, 208] and GPGPU [42]. The power of multi-level IRs as tools to overcome the challenges of computing at the *end of Moore's Law* is proven by the fact that all novel ML compilation approaches leverage different forms of the multi-level IR concept as it enables carrying rich semantic information across the compilation pipeline and transforming it most effi-

[2] LLVM Language Reference Manual: Vector Types
[3] Clang Language Extensions: Vectors and Extended Vectors
[4] LLVM Language Reference Manual: `X86_amx` Type

ciently, avoiding any *semantic loss* [39, 40, 33]. One notable example is MLIR [39] (or *Multi-Level Intermediate Representation*), an SSA-based IR that expands on the multi-level concept by introducing two novel features: *extensibility*, as it can be extended with custom types and instructions (logically grouped in modules called *dialects*) that seamlessly integrate into the existing language, ecosystem of analyses and transformations; *progressive lowering*, as it allows multiple levels of abstractions, coming from different dialects, to coexist in the same IR program. These features enable compilation process to lower (or *spend*) precious semantic information only at the right time when the best possible transformation can be applied, such that a non-reversible *expenditure* of semantic information brings the most value to the lowering result. The effectiveness of MLIR as a compiler construction framework is proven by several innovative ML toolchains [34, 209, 210, 211].

The *back-end* or *code generation* is traditionally the last compilation stage before program emission and is the closest to the target hardware. Tasks like instruction selection, target-specific optimizations, and register allocation are essential for the quality and performance of the resulting program [212]. As discussed in Chapter 2, modular, extensible ISAs and domain-specific architectures are increasing the diversity of compilation targets.

In this chapter I present the work done by the research group led by Prof. Tobias Grosser at the University of Edimburgh (now at Cambridge University) and my research contributions [213] while visiting. The effort focused on investigating how *progressive lowering* can help in dealing with novel, application-specific targets and how SSA, multi-level IRs like MLIR can be leveraged to compile programs. The compilation target is Snitch [48], a RISC-V accelerator architecture designed by ETH Zurich to pursue extreme compute energy efficiency using novel features like floating point hardware loops to elide control flow and stream-semantics registers to reach perfect, software programmed prefetching. Snitch poses particularly interesting code generation challenges that require reasoning at multiple levels of abstraction simultaneously, e.g. affine expressions for memory accesses and target-specific register allocation constraints must be taken into account at the same time. In the resulting publication, Lopoukhine et al. [67] show how MLIR can be leveraged to build an efficient dense linear algebra kernel compiler for Snitch and how the concept of *progressive lowering* can be applied to traditional compiler backend tasks performed on a multi-level SSA IR.

3.1 Compiling at the *End of Moore's Law*: Introduction

Modern general-purpose compiler frameworks use a mid-level, target-agnostic, RISC-like IR as input to a generic and well-optimized backend. As a result, several languages targeting LLVM IR [35] all benefit from shared mid-

level optimizations and mature CPU backends. At the same time, modern DSL compilers are commonly based on multi-level IRs like MLIR [214] that allow for *progressive lowering* of the input program across domain-specific abstractions. While they successfully broaden the expressive power of IRs at the higher levels of the compilation stack, they commonly target LLVM as a backend. LLVM targets modern superscalar CPUs reasonably well, but its RISC-like IR fails to effectively model the domain-specific nature of modern hardware. It is indeed possible to augment the LLVM IR with an increasingly diverse set of target-specific features [44], but these features impact all targets and the implementation is usually a long-term community and engineering effort [215]. At the same time, while being modeled to target modern superscalar CPU architectures, the general approach to backend design struggles to generate extremely high-performance kernels, to the point that even estabilished CPU architecture vendors need to opt out of general-purpose compiler backends to emit high-performance kernels [216]. These critical operations are mainly related to linear algebra (i.e., matrix multiplications and convolutions [216]) where input tensors are of fixed size, and rely on various degrees of auto-tuning and kernel parameter space exploration to reach the best possible performance [216, 217]. This trend is even more severe for domain- and hardware-specific optimizations that are increasingly harder to express in traditional backends: as a result many domain-specific compilers, languages and libraries [198, 218, 219, 220, 217] sidestep the traditional compiler backend.

The approach proposed by Lopoukhine et al. [67] and described in this chapter, widens this *hourglass* model by proposing a multi-level backend which accepts, preserves, and exploits domain-specific information to target highly specialized hardware. It is based on a family of static single assignment (SSA) IRs modeling both the base target RISC-V ISA and structured IRs for domain-specific accelerator extensions. It supports structured control flow and non-standard register usage through a multi-level register allocator that operates across IR abstractions and demonstrate that spilling common in best-effort register allocation is unnecessary for peak performance. By lowering from a high-level DSL, it proves that a wide backend allows for direct lowering of domain-specific concepts to corresponding hardware features, simplifying code generation compared to traditional backends. These concepts are applied to a novel backend for Snitch [48], a scalable in-order RISC-V core augmented with floating point accelerators, custom extensions for streaming registers and hardware loops, and is capable of generating high-performance dense linear algebra kernels for Snitch.

This chapter covers the novel approach to backend construction introduced by Lopoukhine et al. [67]. In particular, Section 3.2 covers the Snitch [48] architecture, its unique ISA extensions and the current programming model (Paragraph 3.2.1). Section 3.3 introduces the MLIR language and ecosystem. Section 3.4 describes the overall approach by Lopoukhine et al. [67] with fo-

cus on my contributions related to stream semantic register (SSR) representation in MLIR (Paragraph 3.4.1), type legalization (Paragraph 3.4.2) and representation and lowering of SSR ISA extensions to assembly-level dialects (Paragraph 3.4.3). Section 3.5 describes the authors' approach to the extensive experimental evaluation needed to assess the performance of generated kernels, focusing on my contributions about the architecture's performance model (Paragraph 3.5.1), the performance metrics used (Paragraph 3.5.2) and how the overall continuous testing and benchmarking methodology is designed and implemented (Paragraph 3.5.3).

3.2 The Snitch Architecture

As technological challenges drive hardware designs towards vertical specialization, the RISC-V modular, extensible, and royalty-free instruction set architecture (ISA) is gaining popularity for domain-specific accelerators [2]. As shown in Chapter 2, RISC-V has become a key enabler for both architecture research [46, 47, 48, 49] and novel concepts in accelerator and processor design [50, 51, 52, 53, 54, 55]. The RISC-V ISA defines a simple load-store reduced instruction set computer (RISC) architecture [169]. The ISA is organized in small groups of logically related instructions, simplifying their hardware implementation and enabling composition. Custom extensions are increasingly adopted to ship specialized designs [51, 221, 176] ranging, for example, from multi-core cloud CPUs with hardware barriers, cache control and custom vector instructions [222, 52] to energy efficient architectures with multi-precision packed-SIMD instructions [223, 224].

Increasing hardware specialization creates challenges in efficient code generation for traditional compiler backends. This is due to the widening gap between the high-level, application-driven semantics of such extensions that are hard to represent with common IR and backend data structures [225].

Snitch [48] is an open-source, state-of-the-art RISC-V core design by ETH Zurich. It applies novel architectural solutions to address the compute scalability challenge in terms of energy efficiency, achieving more than double the floating-point (FP) performance per Watt of other leading commercial accelerators. It has been successfully used as the fundamental building block of large, multicore architectures, like Occamy [19], Manticore [177] and heterogeneous tiled accelerators [55]. The same accelerator architecture is at the core of the *Monte Cimone* accelerated partition presented in Chapter 2.

Snitch (Figure 3.2) comprises a lean, in-order integer core that can be expanded via an accelerator port. In our case, the Snitch core is augmented with a large FPU, capable of multi-precision and (non-standard) packed-SIMD instructions [195, 223]. Moreover, the integer core connects to the FP accelerator subsystem through a sequencer unit, which drives the FPU with a stream of instructions independently from the integer core. Its *key*

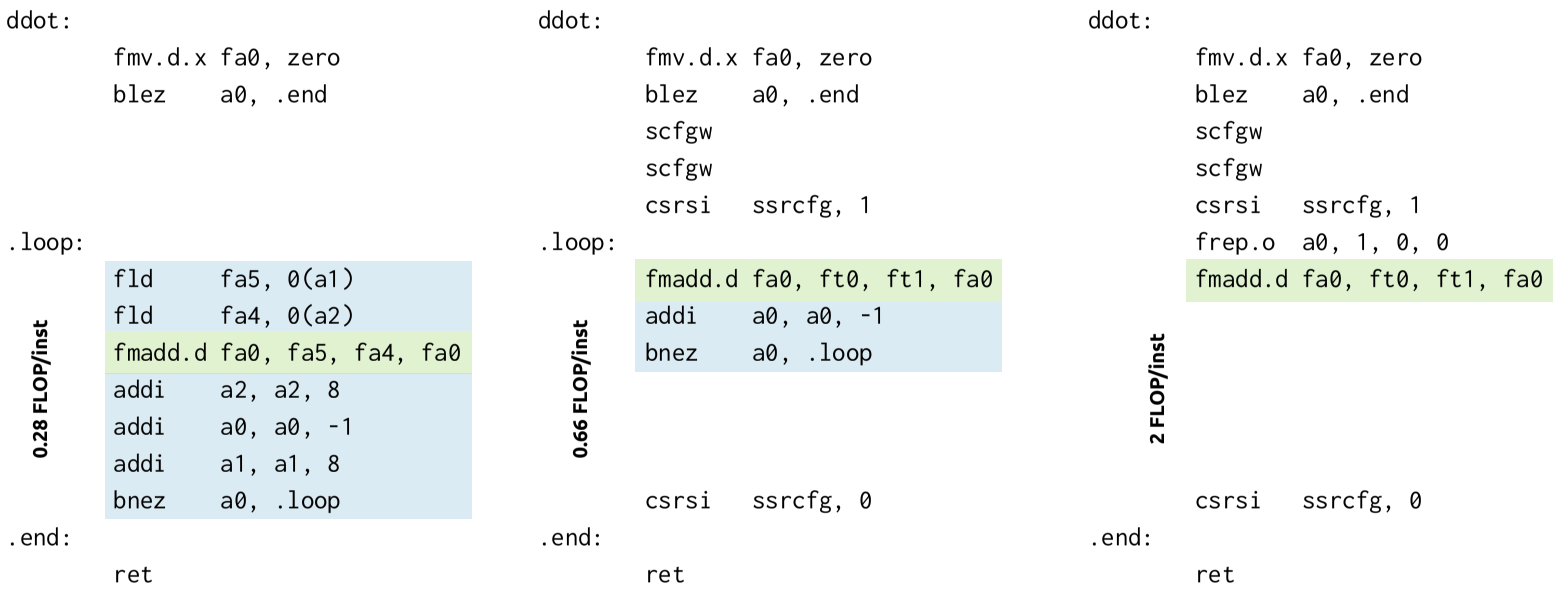


Figure 3.1: Double precision vector inner product (BLAS DDOT), increasingly optimized for Snitch. The baseline (left) implementation using RISC-V standard ISA extensions only (base ISA with `d`) reaches a theoretical peak of 0.28 FLOP/instruction in the loop body. The second implementation (center) introduces SSRs from the Snitch ISA, reaching a peak throughput of 0.66 FLOP/instruction. The third implementation (right) replaces explicit control loop with the `frep.o` hardware loop, reaching the architecture's theoretical peak throughput of 2 FLOP/instruction. This figure is from Lopoukhine et al. [67].

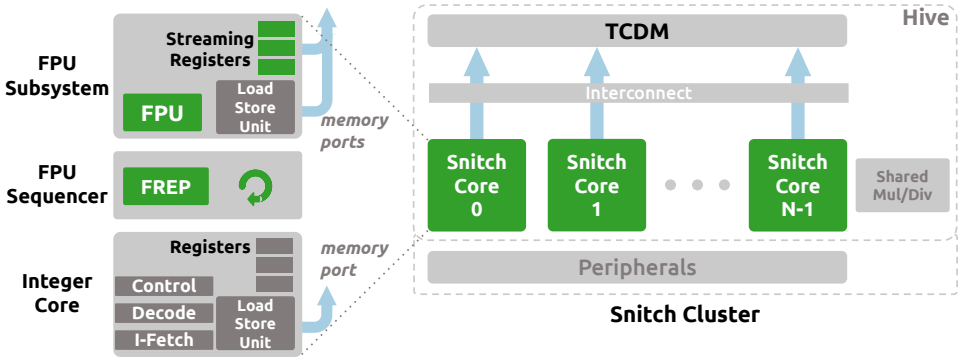


Figure 3.2: Simplified high-level overview of the Snitch micro-architecture [48]. This simplified model is used to define the performance model (Paragraph 3.5.1) used by the experimental evaluation. FPU utilization can be maximized using hardware loops (FREP) to remove explicit loop control flow and SSR to eliminate explicit FP load/stores for affine access patterns. This figure is from Lopoukhine et al. [67].

design idea is to maximize area and energy efficiency, by utilizing the FPU core for most of the computation. Two custom features enable optimal FPU utilization:

- **Stream semantic registers (SSRs)** implicitly handle FP loads/stores when adhering to affine linear memory accesses [226].
- **Floating-point repetition (FREP)** repeatedly executes an FP instruction sequence, removing the need for loop control flow (conditionals, jumps, induction variables).

Combining SSRs and FREP allows the FPU to execute instructions without waiting for the integer core to provide them, effectively making the architecture pseudo-dual issue. Lastly, Snitch uses a fast, energy-efficient, high-throughput tightly-coupled data memory (TCDM), acting as a software-managed L1 cache. When the FP accelerator subsystem is configured to execute streams of FP instructions independently, the integer core can continue its operation, making the architecture effectively pseudo-dual issue without any out-of-order execution in the micro-architecture. A hardware scoreboard enables staggering memory accesses tracking load-use dependencies, thereby achieving a significant degree of latency tolerance for memory operations at a minimal hardware cost [48].

With the help of a scalable interconnect, multiple Snitch cores are then aggregated in a *cluster*, where all cores share a single integer multiply and divide unit, an L1 instruction cache, and a fast, software-managed TCDM. While the core has no data cache, the shared TCDM is designed to be used as a software-managed scratchpad memory. Moreover, one core per cluster is dedicated to the management of direct memory access (DMA) asynchronous,

strided data transfers between the global memory (shared across all clusters) and the TCDM (shared across all cores of a cluster).

A Snitch core implements the `RV32IMAFD` ISA and, thanks to the extensibility provided by the RISC-V encoding, exposes all of its novel features via custom ISA extensions.

To show the effect of Snitch original extensions on instructions streams we consider a double precision vector inner product using standard `RV32IMAFD` instructions only (Figure 3.1). In this case, for each loop iteration, only 1 out of 7 instructions is actual computation while the rest are loads, induction register updates and control flow, allowing an attainable peak of 0.28 FLOP/instruction.

By introducing SSRs, all load instructions become implicit accesses to `ft0` and `ft1` registers, allowing the elision of load (`fld`) instructions. In this case, 1 out of 3 instructions is actual computation: while induction updates and control flow are still present, all loads and pointer arithmetics have been removed increasing the attainable peak to 0.66 FLOP/instruction. Essentially, SSRs are working as a prefetcher, directly injecting values from local memory into the FPU datapath without passing by the floating-point register file. Traditional hardware prefetchers are based on dedicated micro-architectural units whose goal is to infer access patterns of unknown instruction streams on the fly during execution [227]. In the case of SSRs no runtime inference is involved and it is the software itself that configures streaming units ahead of time to load (or store) data according to a linear affine pattern.

By introducing floating-point repetition (FREP), all control flow and induction updates can be removed, leaving a loop body containing a single `fmadd.d` instruction to be executed. This last form increases the attainable peak to 2 FLOP/instruction. It is worth noting that, by eliding all integer calculations, the introduction of both SSRs and FREP removed the use of cluster-shared resources like the single integer multiply and divide unit.

To enable Snitch-based architectures to efficiently support modern DL models, a wide range of FP formats are supported by the FPU attached to the integer core via its accelerator extension port. The FPU provides instructions operating on quarter (`Xf8`), half (`Xf16` and the IEEE 754 2008 [196] alternate `Xf16alt`), single (`f`) and double (`d`) precision formats. Since the data movement via SSRs happens in blocks of 64 bit, when working with data structures of elements smaller that 64 bit (i.e. FP formats of precisions lower than double), more than one element is transferred from memory to a FP register and vice-versa. In order to process smaller FP formats efficiently, the Snitch FPU implements custom *packed-SIMD* RISC-V extensions (`Xfvec` and `Xfaux`) with rich instructions like expanding inner products and a range of SIMD reductions. Unfortunately, the absence of shuffling instructions make some operations extremely inefficient (e.g.: matrix transpose).

3.2.1 Programming Model

In practice, Snitch extensions have proven difficult to implement in traditional compiler backends due to SSRs adding implicit memory side effects to potentially any FP instruction and FREP introducing a form of implicit iteration, requiring the reconstruction of a large set of code guarantees. By focusing on correctness rather than efficiency, previous attempts [228] made extensive use of global side effects to successfully avoid miscompilations. While producing correct code, cautious approaches tamper the efficiency of the result. Moreover, while both Clang builtins and LLVM intrinsics are available, they are hard to use efficiently and correctly. To date, no compiler provides analyses to automatically generate code for them. Implementing Snitch extensions in traditional compilers has proven challenging, as its capabilities introduce require reconstruction of code guarantees from backend analyses [48, 228].

The current backend compiler support is provided by means of an LLVM downstream fork [228]. As part of this work, the pre-existing PULP Project LLVM fork has been rebased on top of LLVM 18 and the *MC Layer* support for FREP, SSR and the full SmallFloat [195] ISA has been fixed and improved. Along Clang builtins and LLVM intrinsics, access to Snitch-specific features is provided via a C library shipped by the hardware maintainers. Figure 3.3 shows an example of an optimized BLAS SAXPY (single precision scaled vector multiplication) implementation for Snitch, using a mix of application programming interface (API) calls and inline assembly.

Stream semantics must be set up ahead of time (Figure 3.3: a) by specifying, for each needed data mover, the iteration domain. SSRs support any linear affine memory access pattern in the form $a_{i+1} = a_i + stride \times i + offset$. In the SAXPY example only a 1-dimensional iteration domain is needed while SSRs support up to four: in case of multiple dimensions, the n-th dimension stream is going to take into account outer dimensions in its own $offset$. Note that in Figure 3.3 all streams have the same 1-dimensional iteration domain, so a single call is needed to set all data movers to the same configuration.

Proper API calls (Figure 3.3: b) are then used to set the base address for each SSR. These calls translate to specific configuration instructions writing to memory-mapped registers.

After these two groups of setup operations, three streams are ready for operation and, inside a streaming region, `ft0` will be use for streaming reads from the first input vector, `ft1` for streaming reads from the second input vector and `ft2` for streaming writes to the output vector.

Moreover, being pipelined, the Snitch FPU benefits from an unroll factor (Figure 3.3: c) that is at least the cycle latency of the repeated instruction: avoiding read-after-write (RAW) hazards allows the FPU to retire 1 instruction per cycle in a steady state execution.

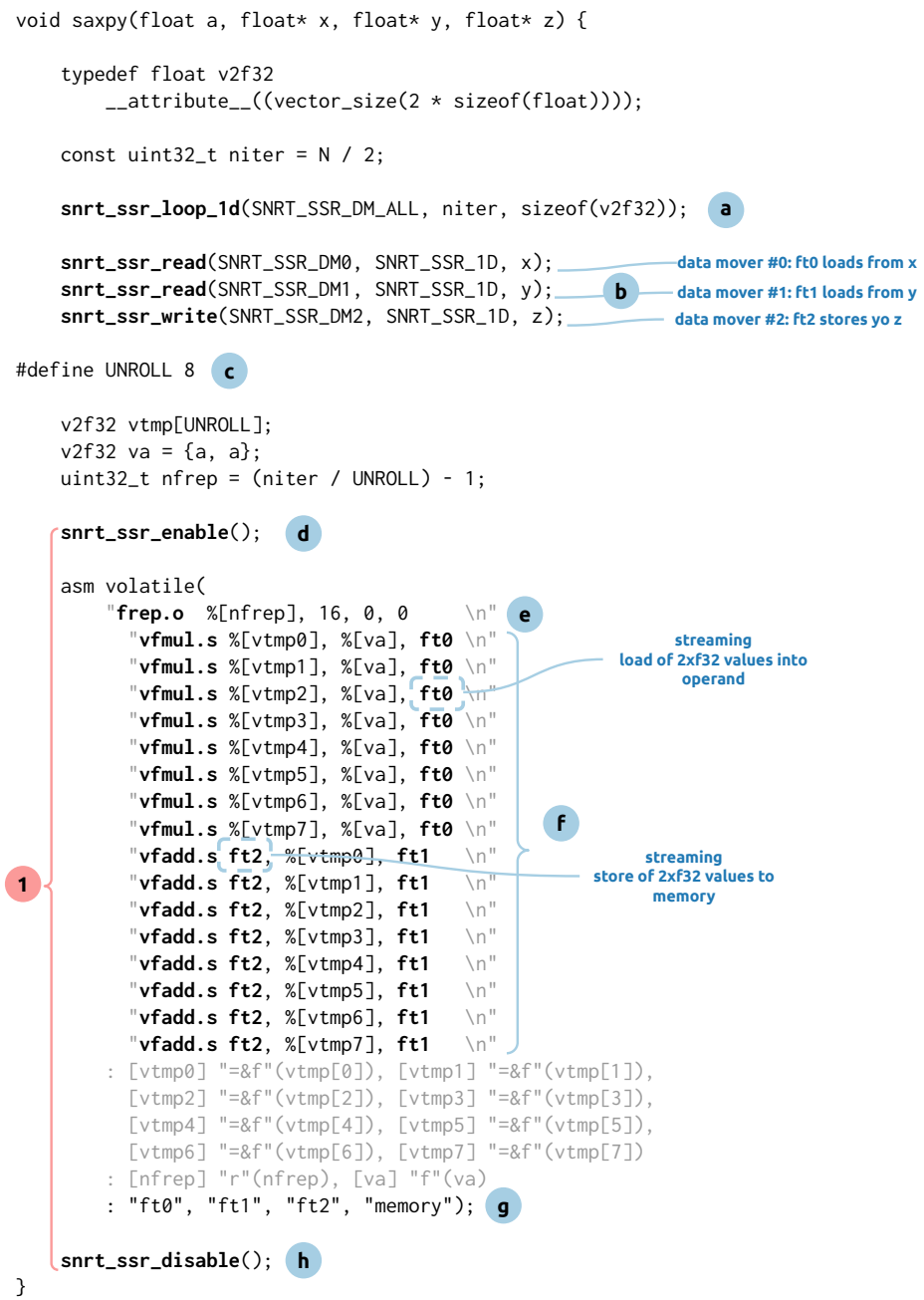


Figure 3.3: BLAS SAXPY operation in Snitch. This code is an optimized implementation of the single-core kernel utilizing read/write SSRs, FREP and an unrolled loop body performing packed-SIMD instructions on vectors of two single precision elements.

In order to remove any control flow from the hot loop (i.e. the streaming region), any iteration instruction (namely induction updates, comparisons and jumps) is replaced by the *floating-point repetition* instruction (the `frep.o` at Figure 3.3: e). In the example, the operands tell the Snitch instruction sequencer to repeat `nfrep` times the following 16 instructions. The two additional immediate operands set to zero tell the sequencer to perform no *register staggering*, a feature that allows to avoid data hazards stalls due to register dependencies: in our case we are manually breaking register dependencies (with the help of the `vtmp` temporary registers) for the sake of clarity.

In the loop body (Figure 3.3: f), the absence of a vector fused multiply-add instruction imposes the use of one vector multiplication (`vfmul.s`) and one addition (`vfadd.s`) for each pair of elements. As pointed out in Section 3.2, data movement via SSRs happens in blocks of fixed 64 bit size. In our example, vector elements are 32 bit wide. This *forces* the introduction of vector instructions since every streaming read from memory is going to load two contiguous single precision elements from memory in the same FP register, that must then treated as a vector of two 32 bit values.

The *streaming region* (Figure 3.3: 1) is delimited by API calls switching on (Figure 3.3: d) and off (Figure 3.3: h) the streaming semantics on the configured FP registers: `ft0` for the first input vector, `ft1` for the second input vector and `ft2` for the output vector. Inside the streaming region, each read or write operation to such registers triggers a read or write memory transfer and the subsequent update of the stream pointer to the next memory location according to the configured access patterns. For this reason, proper register clobbers (Figure 3.3: g) must be specified to ensure the register allocator doesn't allocate SSR-reserved registers to intermediate values. Moreover, a global memory clobber must be specified to avoid incorrect code generation: inside a streaming region, every FP instruction must be considered side-effecting and effectively a load/store according to the streaming register they operate on. Any kind of instruction hoisting or reordering must be prevented to avoid miscompilations.

3.3 The MLIR Ecosystem

Amidst the challenges posed by the *end of Moore's law*, software stacks struggle to adapt, especially at the infrastructure level, where the lack of modularity forces constant rebuilding of similar features, resulting in high engineering costs and limited user flexibility. This issue is particularly evident in the ML space: frameworks, programming languages, and performance libraries are developed in *vertical silos* [229] that neither compose nor interoperate.

MLIR (Multi-Level Intermediate Representation) [39, 214] is a new compiler infrastructure that drastically reduces the entry cost to define and in-

troduce new abstraction levels for building domain-specific IRs. It is part of the LLVM project and follows decades of established practices in production compiler construction. As such, MLIR is an ideal solution to the missing infrastructure problem.

While details of MLIR as an IR are discussed later in this section, we can identify its core principles:

Extensibility. The system is built with extensibility as a fundamental requirement, not an afterthought. Every component, from operations to types to transformation passes, can be extended or customized while kept modular and interoperable.

Progressive lowering. Rather than forcing immediate translations between abstraction levels, MLIR enables gradual lowering of programs, preserving high-level information until it's no longer needed.

Unified infrastructure. Common compiler tasks like generic optimization (e.g., constant propagation or common subexpression elimination), analysis (e.g., dataflow), and transformations (e.g., loop unrolling or tiling) are provided through a shared infrastructure, reducing duplication and increasing reliability. For example, in Section 3.5, we describe how parts of the MLIR core infrastructure can be effectively integrated into a domain-specific compiler.

MLIR has found application in various domains, and it is becoming the *lingua franca* of compiler construction: from tensor algebra compilers [230, 231, 232] to full-stack kernel [68] and graph [233] compilers for DL; from sparse tensor code generators [234] to stencil programs compilers for discretized systems of differential equations [235]. MLIR is also used for target-specific tasks, i.e., to drive automatic code generation for NVIDIA GPU tensor cores [236].

Regarding the traditional compiler *infrastructure*, existing toolchains have been built as isolated efforts, where abstractions, transformations, and representations exist only in the domain of their specific code base. On the other hand, the *interoperability by-design* introduced by MLIR allows for seamless sharing of infrastructure and abstractions. A notable example is xDSL[5]: introduced by Fehr et al. [237], it's a pure-Python compiler construction toolkit that integrates into the MLIR infrastructure. Its design goal is rapid DSL prototyping, IR design and compiler construction, something that requires a non-negligible engineering effort when done in upstream MLIR. xDSL is built on top of IR definition language (IRDL) [238], an MLIR dialect (or *meta dialect*) designed to describe MLIR dialects. Upstream MLIR can build dialect definitions from IRDL sources[6], integrating them in the existing infrastructure at run time. By being completely interoperable, both when transforming programs (by reading and emitting MLIR IR) and defin-

[5] https://xdsl.dev

[6] https://mlir.llvm.org/docs/Dialects/IRDL/

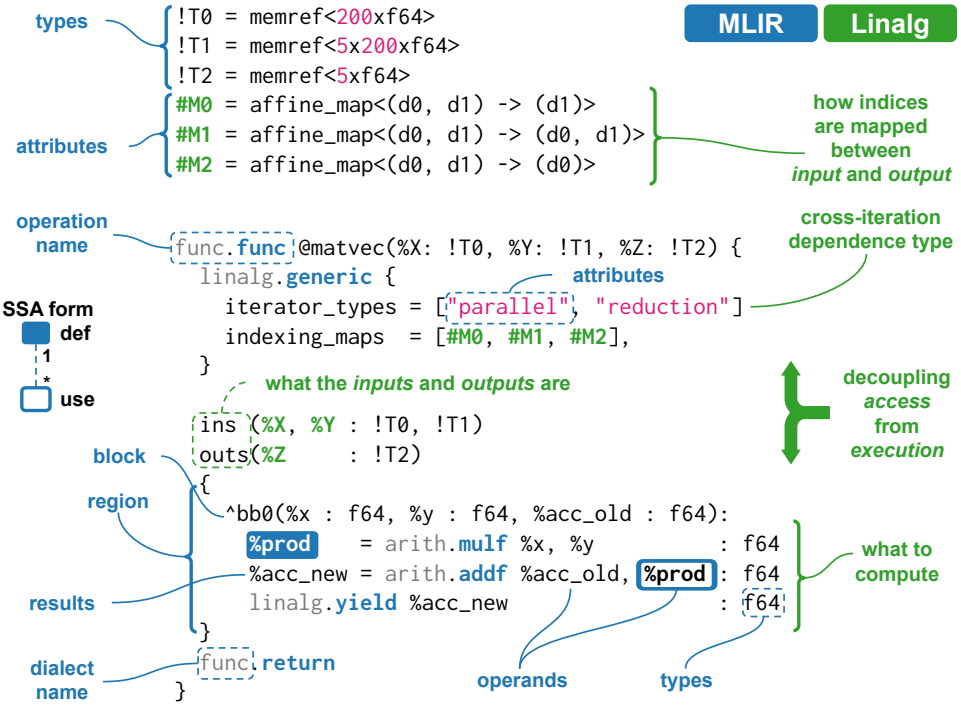


Figure 3.4: Organizing program abstractions as SSA-based IRs enables a modular approach for compiler construction. The above vector-matrix product in MLIR makes the use-def relationships explicit and obviates the need for intricate analyses by capturing information at the right abstraction level (e.g., directly expressing iteration types in `linalg.generic`). This figure is from Lopoukhine et al. [67].

ing new dialects (by emitting dialect definitions in IRDL), xDSL is a notable example of the *unified infrastructure* enabled by MILR. The validity of this approach is proven by both previous work [239] and by Lopoukhine et al. [67], where a mix of xDSL dialects and transformation passes are used alongside a selection of upstream MLIR's to build a kernel compiler for the Snitch architecture (Section 3.2).

As the compiler construction landscape continues to evolve to adapt to the new challenges coming from the *end of Moore's law*, MLIR flexible design and shared infrastructure make it a powerful tool for addressing the challenges of modern computing.

3.3.1 IR Structure

Static single assignment (SSA) intermediate representations (IRs) are widely-used across modern research and industrial compilers (e.g., LLVM [35], GCC [240], Cranelift [241], xDSL [242]), thanks to the broadly-accepted benefits that explicit data flow information offers [212, 243]. *Values* in SSA form are associated with a *unique* name, and each use of a value refers to a *unique*

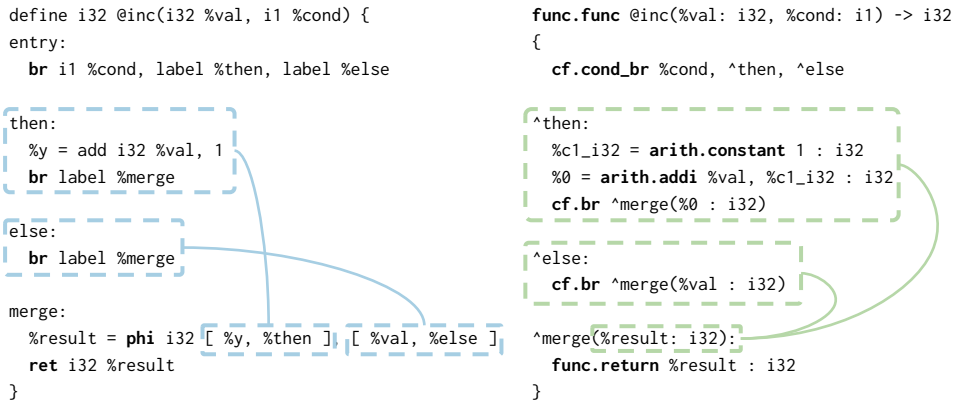


Figure 3.5: The LLVM phi-based SSA form compared to an equivalend MLIR block-based SSA form. In LLVM different SSA values defined in different incoming branches of the control flow are merged by the `phi` instruction. The MLIR snippet uses *blocks with arguments* instead of ϕ nodes: the `^merge` block is entered with different SSA values as its argument. The MLIR snippet uses the `cf` dialect to represent unstructured control flow to keep the two forms as similar as possible

definition. We use SSA IR as implemented in MLIR [214].

Operations outline computation alongside their SSA values (i.e., results and operands) (Figure 3.4). A *dialect* forms a namespace for a set of related types and operations. Operations are prefixed with their dialect name (e.g., `arith.addf`) and may contain *attributes*, a key-value map of compile-time constants.

Operations organized in *blocks* correspond to straight-line code (i.e., basic blocks [212]). Blocks can represent bodies of functions or `for` loops, and may have values as arguments. A *region* is a list of blocks associated with an operation. The semantics of their parent operation defines complex control flow between regions and blocks. For instance, `scf.for` embodies a typical `for` loop, with an induction variable incrementing within an integer range (Figure 3.4). Combining regions as a first-class IR element with SSA allows the direct encoding of nested hierarchical structures in the IR, without any restrictions in the combination of dialects used to express a program.

At the language level, the choice of MLIR authors of a block-based SSA form over traditional alternatives has profound effects on the readability, compactness and effectiveness of the IR language. The LLVM IR, for example, uses a form based on *phi* (ϕ) functions that are introduced to merge values from multiple predecessor blocks [243, 212]. If two paths lead to a block, the ϕ-function selects the appropriate value for the variable depending on the incoming control flow. On the other hand, the SSA form based on *blocks*, the one selected by MLIR authors, eliminates explicit ϕ-functions by treating variables as arguments to basic blocks: instead of a ϕ-function, the predecessors pass the appropriate values as arguments when transferring

control to a block. This makes the IR more human-readable and compact, the control flow more explicit and simplifies analyses ans transformations, as the merging of values is tied to parameter passing rather than separate statements. Comparison between the two approaches is shown with a simplified example in Figure 3.5.

To build a linear algebra kernel compiler for Snitch that is based on MLIR, Lopoukhine et al. [67] use a selection of existing upstream dialects that model common programming abstractions such as functions (`func`), structured control flow (`scf`), memory buffers with reference semantics (`memref`), immutable tensors with value semantics (`tensor`) and SIMD vectors (`vector`).

3.3.2 Linear Algebra Programs in MLIR

The *linear algebra dialect* (`linalg`) is a common lowering destination for high-level, ML-oriented IRs (e.g., `onnx`, `pytorch`) [233, 244]. Its value as an entry point [245] to the MLIR IR is rooted in the ability to concisely captures high-level linear algebra computations using a single, versatile operation, `linalg.generic`, encoding the following properties: i) explicit iterator types encoding data dependencies (e.g., whether the iteration of a specific dimension is element-wise or performs a reduction), ii) affine mappings between iteration space and operand data, iii) an iteration space completely defined by input/output operands, and iv) a lambda specifying the computation (Figure 3.4). These properties are hard, or impossible, to reconstruct from low-level encodings [246, 229, 247]. The value of having those informations fully preserved from the input language and readily available during the lowering process is shown by the results in Lopoukhine et al. [67], where the `linalg` dialect is the selected entry point for programs that effectively utilize the target Snitch accelerator (Section 3.5).

3.4 A Multi-Level Compiler Backend

Lopoukhine et al. [67] introduce a novel RISC-V backend representing target-specific concepts at multiple levels of abstraction, designed and implemented from scratch by leveraging the MLIR infrastructure throughout the compilation flow. Target assembly is modeled in SSA with a set of MLIR dialects: the lower level `rv` and `rv_cf` dialects and encode semantics (e.g., structured control flow in `rv_scf` and call conventions `rv_func`) with the higher ones. In contrast to traditional, monolithic backends, the MLIR-based infrastructure is split into components that are easy to reason about and extend.

Notably, the `rv` dialect uses MLIR's extensible type system, denoting assembly instructions as operations where source and destination registers correspond, respectively, to operands and results (Figure 3.7). Some operations, such as `rv.get_register`, are not printed in the assembly; these exist

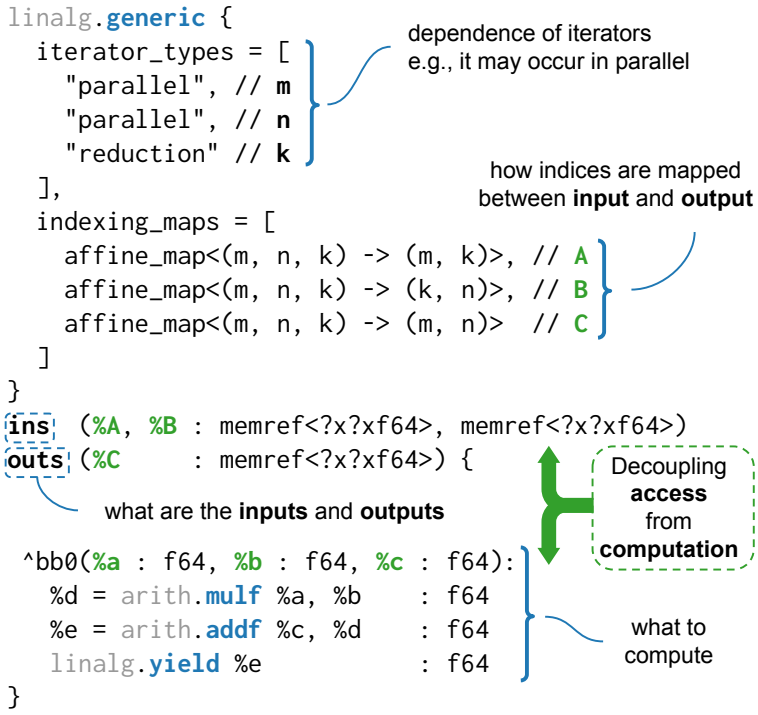


Figure 3.6: The presented approach leverages valuable information that *explicitly* captures accesses and computation when expressed as an MLIR `linalg.generic` operation. For this matrix multiplication, the reduction dimension `k` along with how it maps to the input and output matrices is clearly expressed. This figure is from Lopoukhine et al. [67].

to create SSA values in the IR, bridging SSA semantics and the representation of registers in the type system (Figure 3.7). Operations in the `rv_cf` dialect model unstructured control flow via jump instructions to other basic blocks in the IR.

Along their novel representation of the RISC-V ISA in MLIR, authors introduce additional, higher level RISC-V dialects allow to preserve more semantic information that is useful for target-specific optimizations. For example, the `rv_func.func` operation (Figure 3.7) encodes the application binary interface (ABI) constraint of requiring function arguments and results to be passed in *A* registers. Similarly, the `rv_scf.for` operation represents a for loop in a structured way, easing optimizations and live range construction during register allocation. These dialects are designed to mirror the existing `func` and `scf`, making lowering from higher abstractions straightforward.

In contrast to monolithic compiler backends, the modular approach introduced by Lopoukhine et al. [67] eases the addition of new capabilities. In order to target Snitch unique features, the same MLIR backend structure is augmented for ISA extensions by following a similar multi-level approach, explicitly encoding accelerator semantics in the IR.

Snitch packed-SIMD operations, streaming configuration, and FREP loops (Section 3.2) are modeled in the `rv_snitch` dialect. SSRs add memory effects to previously pure arithmetic operations, a semantics that is modeled with MLIR operations interacting with streaming registers (Figure 3.7). Snitch SmallFloat [195] packed-SIMD instructions are modeled similarly to the standard FP instructions as they both operate on scalar FP registers. FREP hardware loops are modeled with a region body and iteration count operand, along with a mechanism to accumulate results (Figure 3.7), adding the constraint that only instructions on FP registers and stream operations are allowed in the loop body. Additionally, Lopoukhine et al. [67] introduce the `snitch_stream.streaming_region` operation to encapsulate the streaming configuration and the region where streaming is enabled (Figure 3.7). These operations provide convenient targets for higher-level compiler passes. The representation of stream configurations as compile-time constants allows to easily perform some key performance optimizations (Figure 3.7) like constant folding on configurations: since the memory-mapped registers storing data movers parameters are stateful, redundant configuration sequences can be folded to avoid useless overheads. A single construct for upper bounds and strides allows detecting and removing contiguous accesses, reducing the number of generated assembly operations for accelerator configuration. Similarly, a stride of 0 in the last dimension represents a repeated memory access to the same location, for which the Snitch ISA extensions provide a dedicated optimization, reducing the pressure on the memory interconnect. Declarative, high-level representations of accelerator capabilities allow the compiler to use simple peephole rewrites for custom optimizations.

Regarding other traditional backend tasks like scheduling, Snitch's in-

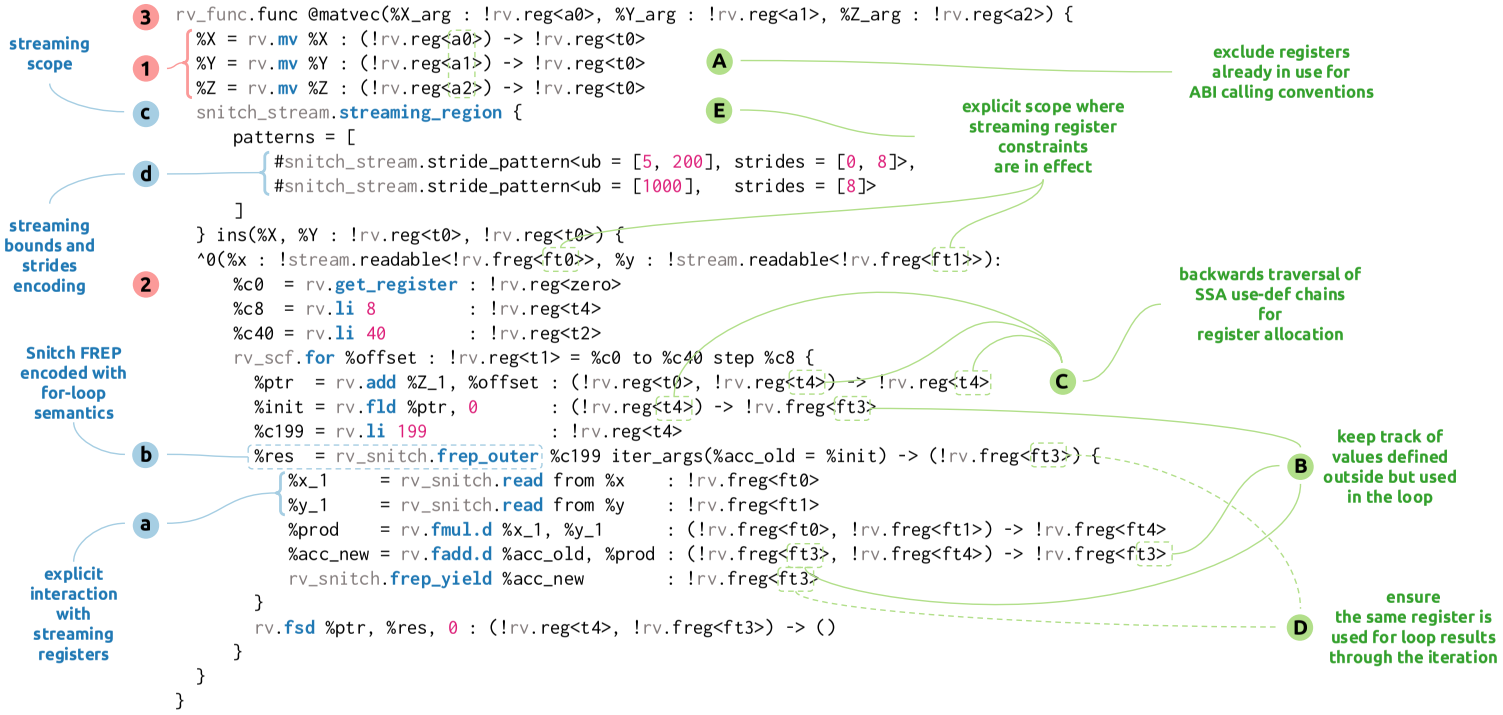


Figure 3.7: The presented multi-level backend uses a mix of SSAs-based IRs to represent different levels of abstraction around the RISC-V ISA for a matrix-vector calculation. The SSA formulation of the ISA empowers the compiler to employ well-understood analyses and transformations and, when combined with regions, to encode further information control flow information (e.g., for loops) while staying close to the semantics of the ISAs. This figure is from Lopoukhine et al. [67].

order core, software-managed L1 memory and the absence of caches on any level of the memory hierarchy make its performance predictable. This hardware design principle, followed by Snitch designers to reach the best possible compute efficiency both in terms of power and area, allows compiler writers to rely on a simple scheduling pipeline without the need for expensive schedule space exploration or sophisticated cost models.

Several other optimizations are introduced by Lopoukhine et al. [67]. For example, to avoid accumulating intermediate results in memory, reduction indices are excluded from iteration space specifications, guiding lowering to loops with local values for accumulation. RAW conflicts are mitigated using unroll-and-jam, interleaving multiple iterations in innermost loops, balancing code size and register pressure for performance (Figure 3.8). The optimal unroll factor depends on pipeline depth: for Snitch, the FPU has three stages, so stalls are minimized with an unroll factor of at least four. Fixed iteration space enables separation between stream setup and computation, lowered to `scf` loops, with inner operations working on streams instead of memory.

Moreover, Lopoukhine et al. [67] introduce a spill-free register allocation approach. The allocator is based on three linear passes operating direclty on MLIR and highlights a key feature of this novel *progressive allocation* approach: the Snitch ISA extensions impose additional constraints on register allocation. Snitch reserves the use of some registers during streaming, and the set of reserved registers depends on the number of streams configured and enabled in a specific streaming region. Those constraints can be expressed locally and naturally by each lowering pass by partially allocating SSA values to registers known to be required ahead of time. Another important feature comes from the fact that the SSA form guarantees that a linear walk respects the order of use-def relations: this property extends to MLIR's SSA with regions, and enable allocation of whole function bodies in a single backward walk. Moreover, since the handling of unstructured control flow is not needed when operating on MLIR, no kind of liveness computation is needed. Finally, another key point claimed by Lopoukhine et al. [67] is that a register allocator for kernel compilers can afford to be *spill-free*. Since the pipeline is designed to compile kernels supposed to be the hottest spots in a linear algebra program, spilling is not taken into account: as soon as a specific program runs out of allocatable registers, the compiler is allowed to fail due to the resulting micro-kernel code being automatically inefficient.

In their work, Lopoukhine et al. [67] present a multi-level, modular approach to compiler construction that facilitates the extension of backends to target accelerators with custom ISA extensions by leveraging shared abstractions and infrastructure.

Details of concepts developed as part of this work are presented later in this section. In particular, we introduce the `memref_stream` MLIR dialect that is the operation that carries all the information needed to exploit SSRs and hardware loops in later lowering steps (Paragraph 3.4.1). We

then discuss a traditional backend transformation like type legalization and how it is performed on MLIR, and how the operations introduced by the `memref_stream` dialect make tasks like vectorization (essential for correct execution of non-double precision FP code on Snitch) an efficient transformation (Paragraph 3.4.2). Finally, we present how Snitch ISA extensions are represented and directly lowered to target assembly code (Paragraph 3.4.3).

3.4.1 Representing SSRs

Dialects introduced by Lopoukhine et al. [67] represent concepts at different level of abstraction to support an effective lowering strategy. The `memref_stream` dialect bridges `linalg` input dialect and `snitch_stream` operations (Figure 3.8). The `memref_stream.generic` operation is based on its `linalg` counterpart, except an explicit encoding of the iteration bounds, in contrast to `linalg`'s approach of inferring bounds from input shapes (Section 3.3.2).

Designed to extend the `linalg.generic` operation from MLIR upstream, `memref_stream.generic` is the entry point of the Snitch lowering process. The `memref_stream` dialect bridges `linalg` abstractions and `snitch_stream` operations (Figure 3.8). The `memref_stream.generic` operation is based on its `linalg` counterpart, except an explicit encoding of the iteration bounds, in contrast to `linalg`'s approach of inferring bounds from input shapes (Section 3.3.2). Figure 3.8 shows the structure of a `memref_stream.generic` operation.

3.4.2 Type Legalization

Type legalization is the process of transforming data types into lower-level, hardware-supported types that can be directly handled by the target machine [212]. The process typically involves decomposing complex types (e.g., structures, vectors, or large integers) into smaller, simpler types that conform to the target architecture's constraints: this usually simplifies later backend transformations, e.g., instruction selection patterns can be defined just on the operand types that the target hardware actually supports. For example, a 128 bit integer might be split into two 64 bit integers if the hardware lacks native support for 128 bit operations. Similarly, vector types may be expanded, split, or aligned to match the hardware's SIMD capabilities.

As detailed in Section 3.2, the Snitch architecture integrates a data mover unit performing either register-to-memory and memory-to-register streaming transfers operating only on blocks of data of fixed 64 bit size. This means that all input structures (either `memref` or `tensor`) to a `memref_stream.generic` structured operation with element types other than `f64` must be be analysed for legality. In particular, if the scalar block arguments to the payload body are of types other than `f64`, then the whole `memref_stream.generic` must

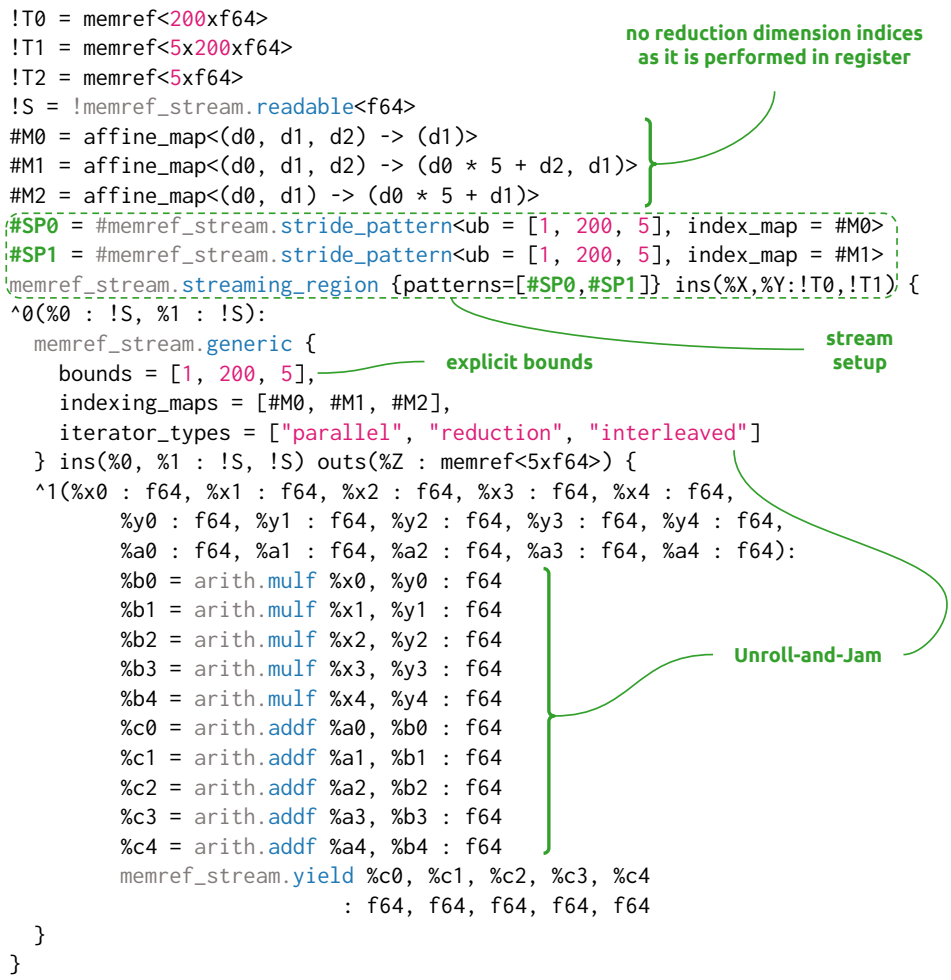


Figure 3.8: The `memref_stream` abstractions bridge the gap between high-level linear algebra abstractions and Snitch accelerator capabilities, allowing the scheduling of computation operations before separating access from execution. This figure is from Lopoukhine et al. [67].

```
func.func public @sumf32(%m0 : memref<8x16xf32>,
                         %m1 : memref<8x16xf32>,
                         %m2 : memref<8x16xf32>)
                      -> memref<8x16xf32> {
  memref_stream.generic {
    bounds = [8, 16],
    indexing_maps = [
      affine_map<(d0, d1) -> (d0, d1)>,
      affine_map<(d0, d1) -> (d0, d1)>,
      affine_map<(d0, d1) -> (d0, d1)>
    ],
    iterator_types = ["parallel", "parallel"]
  } ins(%m0, %m1 : memref<8x16xf32>, memref<8x16xf32>)
    outs(%m2 : memref<8x16xf32>) {
  ^0(%in0: f32, %in1: f32, %out: f32):
    %0 = arith.addf %in0, %in1: f32
    memref_stream.yield %0: f32
  }
  func.return %m2 : memref<8x16xf32>
}
```

Figure 3.9: Example of element-wise addition of 2-dimensional matrices (in the form of `memref` memory buffers) represented with a `memref_stream.generic` operation. Element types are single precision FP scalars, a data type that is handled by the Snitch FPU but becomes *illegal* with respect to SSR memory transfers. This program needs to be legalized (Figure 3.10).

be transformed to achieve legality. Figure 3.9 shows an input IR for a simple element-wise addition of two 2-dimensional memory buffers.

The legalization process on MLIR is achieved via a transformation process that works as follows:

1. analyze the input data types and compute the tiling factor for the last dimension according to the bit width of the element type (i.e. the number of actual lanes in the packed-SIMD vectors);

2. modify iteration bounds according to the tiling factor. On a structured loop, this equals to tiling the stride-1 dimension and applying an equivalent unrolling factor;

3. modify the affine access maps specified in the `memref_stream.generic` operation according to the last dimension's tiling factor. This is needed to account for the iteration bounds modified in the previous step;

4. modify the payload body introducing `vector` types of proper size according to the tiling factor. This is done with simple rewrites following

```
func.func public @sumf32(%m0: memref<8x16xf32>,
                         %m1: memref<8x16xf32>,
                         %m2: memref<8x16xf32>)
                      -> memref<8x16xf32> {
  memref_stream.generic {
    bounds = [8, 8],
    indexing_maps = [
      affine_map<(d0, d1) -> (d0, (d1 * 2))>,
      affine_map<(d0, d1) -> (d0, (d1 * 2))>,
      affine_map<(d0, d1) -> (d0, (d1 * 2))>
    ],
    iterator_types = ["parallel", "parallel"]
  } ins(%m0, %m1: memref<8x16xf32>, memref<8x16xf32>)
    outs(%m2: memref<8x16xf32>) {
  ^0(%in0: vector<2xf32>, %in1: vector<2xf32>, %out: vector<2xf32>):
    %0 = arith.addf %in0, %in1: vector<2xf32>
    memref_stream.yield %0: vector<2xf32>
  }
  func.return %m2: memref<8x16xf32>
}
```

Figure 3.10: Result of Snitch legalization applied on the input example of element-wise addition of 2-dimensional matrices shown in Figure 3.9. Highlighted changes are adaptation of static iteration bounds, affine access maps and payload body's block argument types. The resulting IR represents a `generic` operation (from the `memref_stream` dialect) that has been tiled by a factor of 2 and vectorized in its computational payload.

use-def chains from the block arguments to the payload body until both the yielded type and block arguments are all of legal `vector` types.

Figure 3.10 shows the result of the legalization transformation, where only legal types are present in the payload body of the `memref_stream.generic` operation.

Vectorization is a challenging problem in traditional compilers due to its needs of analyzing data dependencies, memory access patterns, and control flow to ensure correctness while transforming scalar operations into vector instructions [212]. In our case, the process starts from a `memref_stream.generic` that carries the same information as a `linalg.generic` operation (as shown in Section 3.3), augmented with additional metadata needed for Snitch lowering (i.e. static iteration bounds). The decoupling between memory accesses and actual computation, with the former being completely defined by explicit iterator types, affine mappings and input/output operands, make transformations that are traditionally difficult like multi-dimensional tiling and vectorization, efficient. The advantage of having such rich information readily available during the lowering process (as shown in Section 3.3, informations

that are hard, or impossible, to reconstruct from low-level encodings [246, 229, 247]) is clearly demonstrated by the legalization process: correct type legalization and vectorization (via tiling) become straightforward analyses and transformations.

3.4.3 Configuring Software-Managed Prefetchers

The ability to represent assembly instructions in MLIR can be used to model Snitch SSR setup instructions. SSR-specific instructions are used to configure and control streaming behaviour, thus defining the boundaries of streaming regions in the instruction stream (Paragraph 3.2.1). Among other target-specific dialects discussed in previous sections, the `snitch` dialect is introduced for this purpose. This dialect can be seen as a *target dialect*, as its operations are directly mapped to target functionalities (or instructions as in this case). Figure 3.11 shows the direct lowering performed by a simple transofmation pass to RISC-V assembly with Snitch extensions. Several examples of this class of target dialects are present in upstream MLIR (i.e. `nvgpu`, `amdgpu`, `arm_sve`, and others), even though their output is usually in the form of LLVM intrinsics instead of machine instructions. Figure 3.11 shows an input IR program that configures a three-dimensional read stream (Figure 3.11: a, by setting respective *bound*, *stride* and *source address*) and a one-dimensional write stream (Figure 3.11: b). It's worth noting that the first stream is also configured with a repetition value. After the setup phase, it defines a streaming region in the straight-line operation sequence by switching on and off (Figure 3.11: c) the streaming semantics for the respective FP registers. In Figure 3.11 is also shown the IR result of the `lower-snitch` transform pass. The produced IR uses operations from both `riscv` and `riscv_snitch` dialects, where the former is used to represent instructions part of the RISC-V standard set of extensions (Figure 3.11: 1) and the latter for Snitch-specific ISA extensions (Figure 3.11: 2). It is worth noting that both dialects (`riscv` and `riscv_snitch`), introduced in Lopoukhine et al. [67], are *assembly-level dialects*: each operation represent a specific ISA instruction and operands are either *attributes* (for immediate/constant arguments) or proper instruction operands of the `riscv.reg` type, that represent SSA values that still need to be allocated to registers. This snippet highlights key features of MLIR, where multiple dialects can coexist in the same IR program and a dialect can extend existing dialects by leveraging its types and transformations.

3.5 Experimental Evaluation

Lopoukhine et al. [67] present an extensive experimental evaluation to assess the ability of assembly-level RISC-V dialects to represent high-performance linear algebra kernels and to prove that their multi-level register allocator

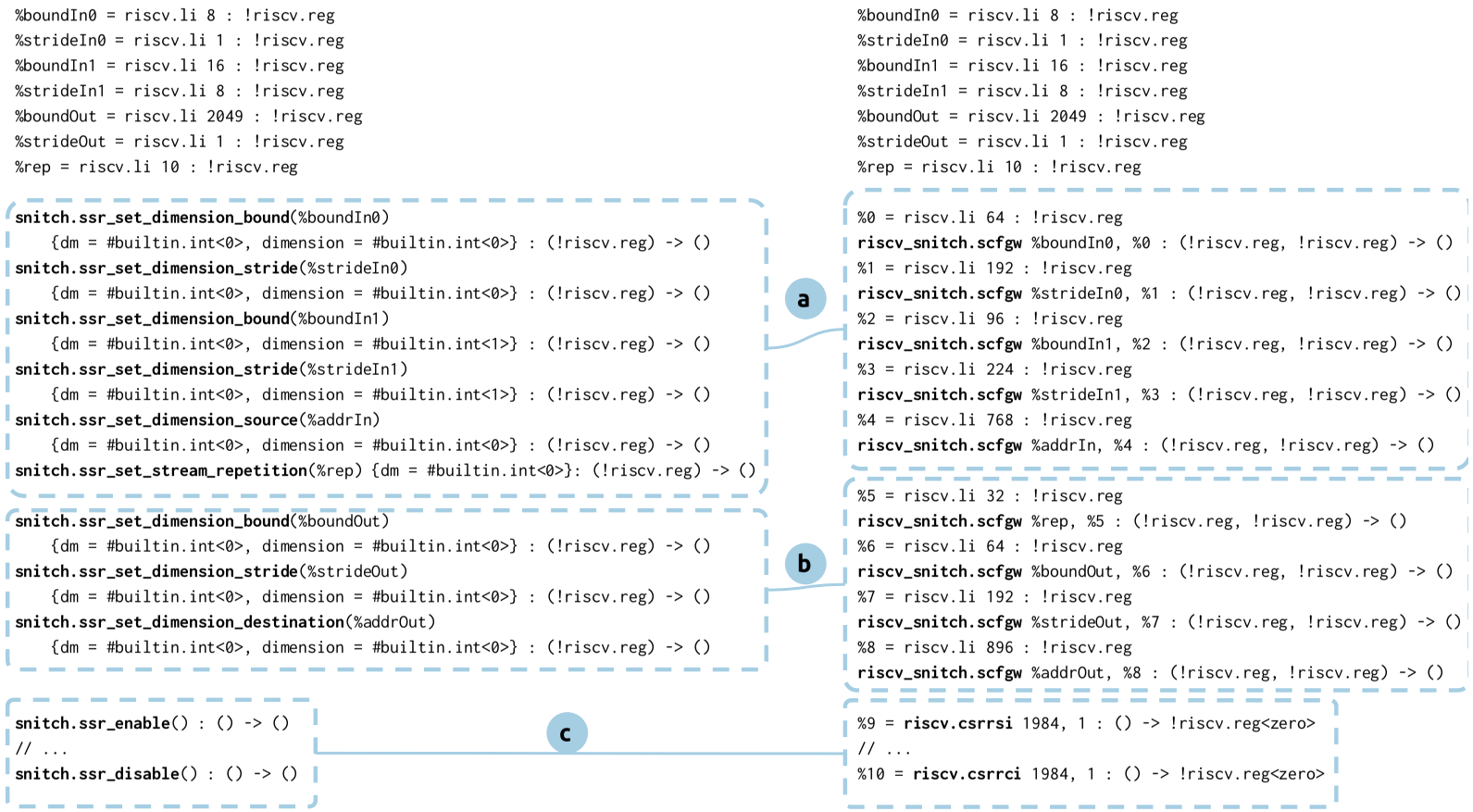


Figure 3.11: `snitch` lowering to *assembly-level dialects* dialects (`riscv` and `riscv_snitch`). The program configures a three-dimensional read stream by setting respective *bound*, *stride* and *source address*, and a one-dimensional write stream. The input stream is also configured with a repetition value. Finally, streaming semantics is turned on and off. On the right is the IR result of the `lower-snitch` transform pass.

can support code generation even without support for register spilling. Experimental methodology, infrastructure, results and findings are summarized in this sections.

The benchmarking kernels selected by authors aim at being representative of mainstream DNN models. Namely, the benchmark set contains kernels from two DNNs: NSNet2 [248], a noise suppression model, and AlexNet [249], an image classification model. The selection ensures that input kernels cover a wide range of operations: element-wise and reduction computations on tensors, linear and non-linear memory accesses, nested loops. All kernels are manually implemented in both C and MLIR, the latter using only the `linalg` dialect operating on tensors, similar to the MLIR IR emitted by mainstream DL frameworks. An additional variant, implemented in C with inline assembly following the Snitch programming model (as shown in Paragraph 3.2.1) and optimized to reach peak performance on Snitch, is provided for a subset of selected kernels.

As described in Section 3.2, the compilation target is Snitch [48]. The same accelerator architecture is at the core of the *Monte Cimone* accelerated partition (Chapter 2). The open-source reference SystemVerilog implementation of Snitch is compiled with Verilator [250] to generate the register transfer level (RTL) *cycle-accurate* simulator. According to the Snitch micro-architecture (Figure 3.2), the FPU can execute one instruction per cycle peak or two floating-point operations (FLOPs) per cycle peak in case of the fused multiply-add (FMA) instruction), when operating on 64 bit values. For smaller types, a corresponding number of vector operations can be executed. As an in-order, *bare-metal* platform with no runtime or operating system, all measurements on the Snitch platform are deterministic.

The PULP Project provides an LLVM toolchain [228]that is capable of targeting Snitch, enabling the assembly-based programming model described in Paragraph 3.2.1. It's worth noting that no existing compilers can automatically generate code that leveraged Snitch ISA extensions. The evaluation considers two alternative compilation flows, both leveraging the LLVM RISC-V backend: i) a pipeline using upstream MLIR passes, lowering the same inputs as this prototype compiler, and ii) a naive C reimplementation of the same kernel.

A key advantage in constructing MLIR compilation pipelines for the evaluation is the ability to take advantage of MLIR's rich ecosystem of compiler components and tools. The proposed backend is implemented in the xDSL open-source compiler framework (v0.21.1) [237, 242], the *Pythonic* counterpart of MLIR, enabling native integration of core MLIR constructs (i.e., SSA, regions) within the language. Interoperability between xDSL and MLIR is achieved via the common text IR format. For C implementations, we use the LLVM toolchain provided by the Snitch architects [228], containing both the assembler and linker used in all our kernel implementations. The programming model used in C implementations is introduced in Paragraph 3.2.1.

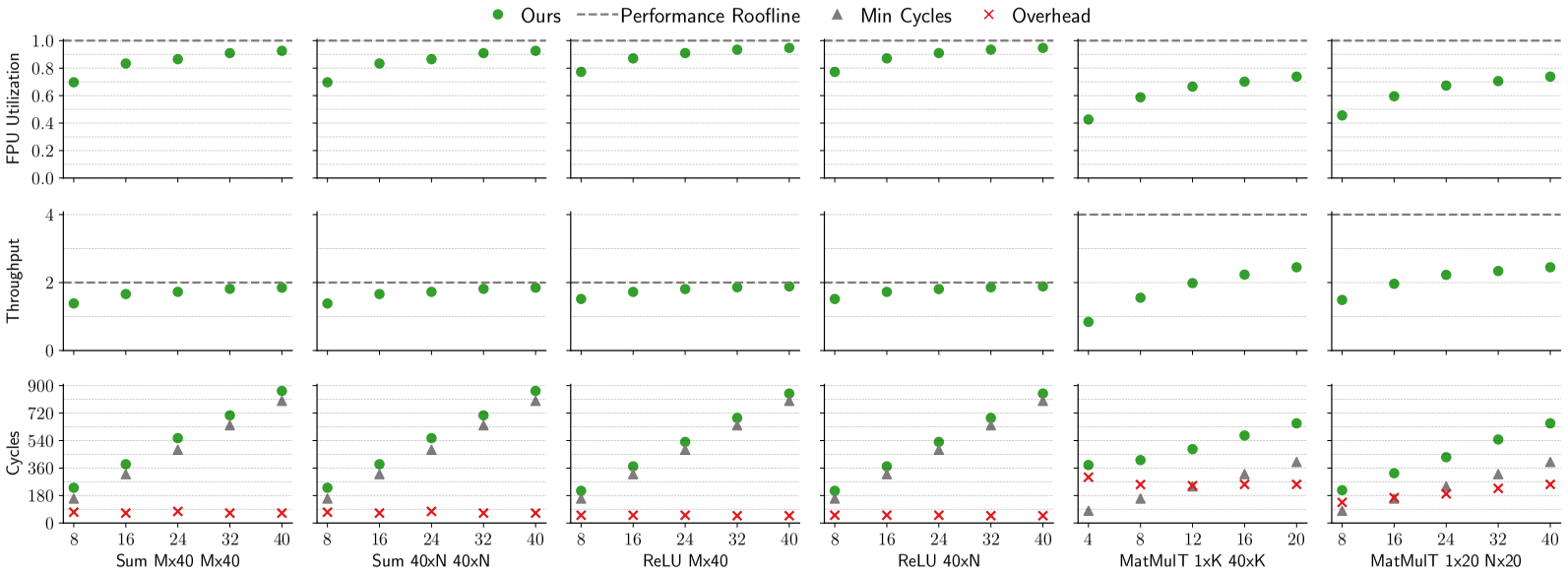


Figure 3.12: The proposed low-level representation is flexible enough to represent linear algebra operations commonly used in machine learning (ML) reaching high FPU utilization, reaching 95% peak FPU utilization and 94% of theoretical maximum throughput. Despite the high FPU utilization, the MatMulT kernel only reaches 2.45 FLOP/cycle throughput due to extra vector packing instructions. This figure is from Lopoukhine et al. [67].

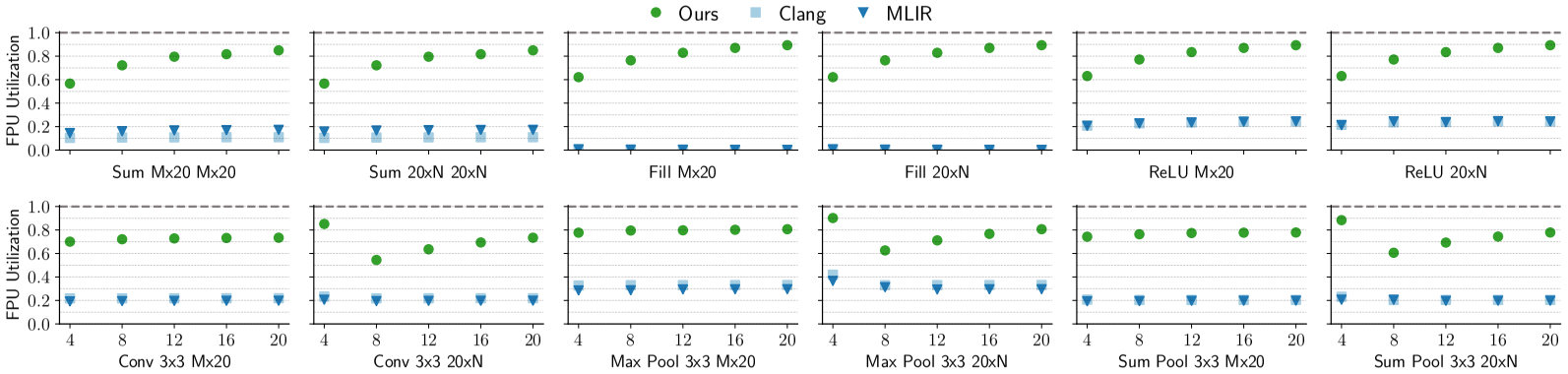


Figure 3.13: Selected micro-kernels compiled with the proposed end-to-end prototype compiler reach up to 95% FPU utilization. In contrast, MLIR does not outperform a naive C implementation compiled with Clang on this platform. This figure is from Lopoukhine et al. [67].

Leveraging existing tools popular in industry eases the adoption of our work in research and production.

3.5.1 Performance Model

To define a *roofline* performance model [126] for Snitch, peak values for both compute throughput and memory bandwidth are defined according to the micro-architecture (Figure 3.2), as follows. The pipelined FPU is capable of executing one instruction per cycle or, in terms of double precision (the largest supported format) arithmetic operations, 2 FLOP/cycle since the ISA provides the FMA instruction that performs both an addition and a multiplication in a single cycle (in the pipeline steady state).

Each core can issue 5 load/store instructions per cycle: 3 coming from SSRs and 2 coming from additional load-store units (LSUs), one in the frontend integer core and one in the FPU proper. While the shared low-latency memory can serve 32 requests per cycle, either a load or a store, each core is connected via 3 memory ports to the low-latency memory, hence at most 3 requests coming from a single core can be served every cycle. Moreover, memory transfers are performed in fixed-size transactions of 8 B, therefore the peak attainable memory bandwidth by a single core is 24 B/cycle. The 8 B fixed transaction has profound effects on both the current assembly-based programming model and the actual code generation: every kernel operating on floating-point precisions lower than double, must deal with packed-SIMD instructions (Paragraph 3.2.1). Since this work doesn't deal with the accelerator's global memory (HBM2E in the case of Occamy [19]), the only memory roof needed refers to the L1 low-latency memory.

The roofline plot for the double precision matrix multiplication kernel is reported in Figure 3.14. The input program is MLIR `linalg` and the resulting RISC-V assembly kernel is obtained via the lowering pipeline presented in this chapter. The plot shows data from 500 simulations with varying tensor shapes: with $C_{M\times N} = A_{M\times K}B_{K\times N}$, experiments range from $(M = 4, K = 4, N = 8)$ to $(M = 8, K = 64, N = 64)$. The limit to the shape space exploration is set by the available L1 memory on the Snitch cluster. Moreover, to overcome the large computational needs of the Verilator simulator, the benchmark harness (described in detail in Paragraph 3.5.3) is run on the Leonardo [5] supercomputer. The plot highlights how all (but one) data points are above the double precision theoretical peak of the architecture (1 FLOP/cycle) thanks to the extensive use of FMA instructions. Figure 3.14 supports the findings depicted in Figure 3.17, Figure 3.18 and Figure 3.19: while kernels operating on larger tensors are near-optimal (in terms of FMA theoretical peak), smaller tensors progressively decrease the sustained performance due to constant overheads (function calls, SSRs setup sequences). It's also worth noting how SSRs behave on the kernel characteristics. Since Snitch's data movers operate as software-managed L1 prefetchers,

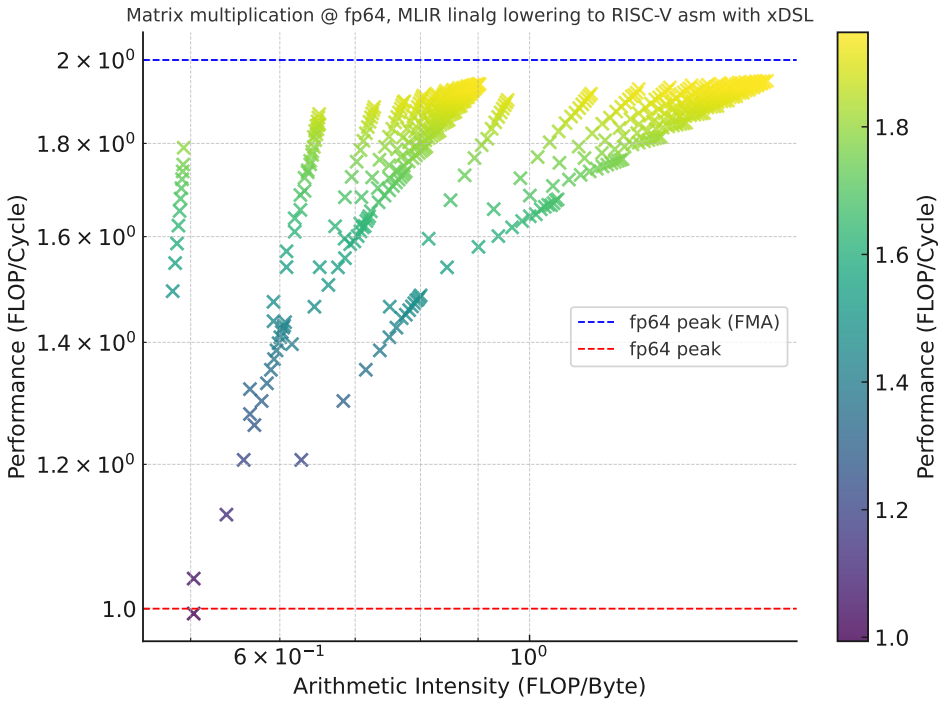


Figure 3.14: Roofline plot of the double precision matrix multiplication kernel. The input program is MLIR `linalg` and the resulting RISC-V assembly kernel is obtained via the lowering pipeline presented in this chapter. The plot shows data from 500 simulations with varying tensor shapes: with $C_{M\times N} = A_{M\times K}B_{K\times N}$, experiments range from $(M = 4, K = 4, N = 8)$ to $(M = 8, K = 64, N = 64)$. Almost all data points are above the double precision theoretical peak of the architecture, highlighting an extensive use of FMA instructions.

the optimal multi-dimensional SSR configuration generated by the proposed compiler backend make the majority of generated kernels compute bound: the operands needed by each floating point instruction are always available at minimal latency at the time of instruction issue.

3.5.2 Performance Metrics

With respect to *throughput* measurements, both instruction/cycle and FLOP/cycle are taken into account. Another important metric is *FPU utilization*, expressed as the ratio of cycles spent in the FPU executing arithmetic instructions over the total execution latency. These three metrics assess the kernel execution speed and the *efficient* use of the available compute resources.

To obtain accurate performance data, we need to measure the cycle count, throughput, and FPU utilization for each kernel execution. Each simulation in the benchmark set is run through the compiled Verilator simulator producing a detailed execution trace. The Verilator trace is disassembled to

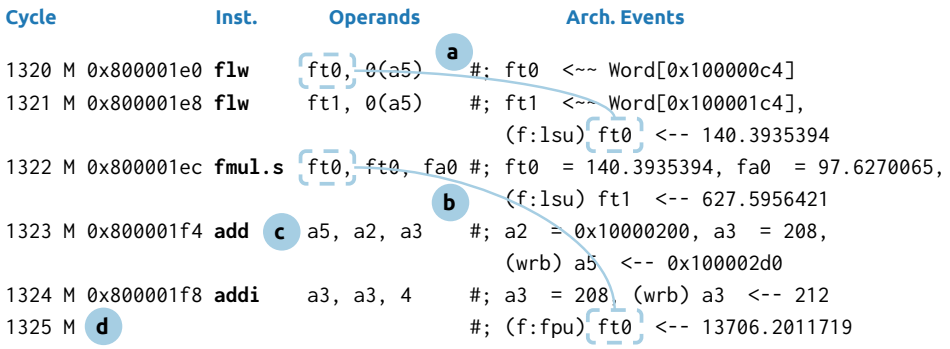


Figure 3.15: Snitch execution trace. This format is produced by both disassembling the Verilator traces and post-processing the result via the Snitch repository tooling.

get instructions and operands from opcodes, and then post-processed via the tooling provided by the Snitch authors along with the SystemVerilog sources. The resulting output is a condensed execution trace in the format presented in Figure 3.15. The trace format reports several information, including cycle count from the beginning of the simulation, the value of the program counter at that cycle, the instruction along with its operands and, more importantly, all the architectural events occurred at each clock cycle. This latter is of particular importance since it allows to measure instruction and memory latencies. For example, Figure 3.15: a highlights the single cycle latency needed to load a single-precision floating point value from L1 memory, from the issue of the `flw` instruction to the actual write back of the destination operand `ft0` by the FPU own LSU (`f:lsu`). For the actual multiplication instruction (`fmul.s`) shown in the example, the latency spans 4 cycles (Figure 3.15: b) and the two subsequent integer additions (`add` and `addi` in Figure 3.15: b) highlight the pseudo-dual issue capability of Snitch since they are executed by the integer core while the FPU is still busy. The actual retirement of the multiplication instruction and write back of the destination register (`ft0`) happens 4 cycles later, in a clock cycle that sees no new instructions issued (Figure 3.15: d), marking it as a stall. From the analysis of this post-processed trace, we can extract all performance counters needed by our experimental evaluation. The full overview of available counters and derived metrics is presented in Table 3.1.

3.5.3 Continuous Testing and Benchmarking Infrastructure

During the early stages of this work we realized that building a kernel compiler requires continuous feedback both in terms of correctness and performance of the generated code. For this reason, we introduced a deterministic testing and benchmarking harness that is then used in a continuous inte-

Counter	Unit	Scope	Description
`tstart`	cycles	cc	The global simulation time when the `mcycle` instruction opening the current measurement region is issued.
`tend`	cycles	cc	The global simulation time when the `mcycle` instruction closing the current measurement region is issued.
`start`	cycles	cc	The core complex cycle count when the `mcycle` instruction opening the current measurement region is issued.
`end`	cycles	cc	The core complex cycle count when the `mcycle` instruction closing the current measurement region is issued.
`end_fpss`	cycles	cc > fpss	The core complex cycle count when the last FP operation issued in the current measurement region retires.
`snitch_issues`	inst	cc > snitch	Total number of instructions issued by Snitch, excluding those offloaded to the FPSS (see `snitch_fseq_offloads`).
`snitch_occupancy`	inst/cycle	cc > snitch	IPC of the Snitch core, calculated as `snitch_issues / cycles`.
`snitch_fseq_offloads`	inst	cc > snitch	No. of instructions offloaded by the Snitch to the FPSS.
`snitch_fseq_rel_offloads`	%	cc > snitch	The ratio between `snitch_fseq_offloads` and the total number of instructions issued by Snitch core proper, i.e., `snitch_issues + snitch_fseq_offloads`.
`snitch_load_latency`	cycles	cc > snitch	Cumulative latency of all loads issued by Snitch's own LSU. The latency of a load is measured from the cycle the load is issued to the cycle it is retired.
`snitch_avg_load_latency`	cycles	cc > snitch	Average latency of a load issued by Snitch's own LSU.
`snitch_loads`	inst	cc > snitch	No. of load instructions retired by the Snitch own LSU.
`fseq_yield`	%	cc > fseq	The ratio between `fpss_issues` and `snitch_fseq_offloads`.
`fpss_issues`	inst	cc > fpss	Total number of instructions issued by the FPSS.
`fpss_fpu_issues`	inst	cc > fpss > fpu	Similar to `fpss_issues`, but counts only instructions destined to the FPU proper.
`fpss_fpu_fmadd_issues`	inst	cc > fpss > fpu	Similar to `fpss_fpu_issues`, but counts only double-precision fused multiply and add.
`fpss_fpu_latency`	cycles	cc > fpss > fpu	Cumulative latency of all FPU instructions.
`fpss_avg_fpu_latency`	cycles	cc > fpss > fpu	Average latency of an FPU instruction.
`fpss_load_latency`	cycles	cc > fpss	Cumulative latency of all loads issued by FPSS own LSU.
`fpss_avg_load_latency`	cycles	cc > fpss	Average latency of a load issued by FPSS own LSU.
`fpss_loads`	inst	cc > fpss	No. of load instructions retired by the FPSS own LSU.
`fpss_section_latency`	cycles	cc > fpss	`max(end_fpss - end, 0)`.
`fpss_occupancy`	inst/cycle	cc > fpss	IPC of the FPSS, calculated as `fpss_issues / cycles`.
`fpss_fpu_occupancy`	inst/cycle	cc > fpss > fpu	IPC of the FPU, calculated as `fpss_fpu_issues / cycles`.
`fpss_fpu_rel_occupancy`	%	cc > fpss > fpu	The ratio between `fpss_fpu_occupancy` and `fpss_occupancy`.
`cycles`	cycles	cc	Overall cycles spent in the current measurement region.
`total_ipc`	inst/cycle	cc	The overall IPC of the core complex, calculated as `snitch_occupancy + fpss_occupancy`.

Table 3.1: Snitch performance counters and derived metrics produced by the simulation traces post-processor provided alongside Verilog sources. Additional post-processing is performed specifically for this work to compute metrics relevant for SIMD profiling. The micro-architectural *scope* (where *cc* stands for *core complex*, *snitch* for the integer core, *fpss* for *FP sub-system* and *fpu* for just the FPU itself) and a description of each counter/metric are also reported.

gration infrastructure to provide feedback on every change introduced in the code generation pipeline. Given the set of selected kernels, the pipeline explores the parameter space of input tensor shapes and, for each kernel instance, checks for results correctness and produces all performance counters computed from cycle-accurate Verilator simulation traces (Paragraph 3.5.1). Moreover, in order to assess the contribution of each optimization pass and the optimal pass schedule, the test harness explores the parameter space of optimization passes by adding each optimization incrementally. Results of this exploration are reported in Table 3.2.

The pipeline is implemented in the Snakemake [251] workflow language and depicted in Figure 3.16. In particular, `kernel_generate_*` tasks drive the parametric kernel generator to explore the space of input tensor shapes. `optimization_pipelines` generates variants of the lowering pipeline by incremental addition of optimizations. All tasks downstream of `verilator` (that runs the actual simulation) are devoted to post-processing of execution traces and computation of performance counters.

The Snitch Verilog source repository provides utilities to decode simulation traces: post-processing the cycle-accurate execution traces allows us to compute all the performance metrics needed to support our benchmarking model without any hardware support (like an actual *performance monitoring unit*).

To support the parameter space exploration needed to guide both the implementation and tuning of optimization passes, we also introduced a kernel templating system. While not directly implemented in MLIR but instead as an external tool, our kernel generator works similarly to the `mlir-gen` generator by Golin et al. [68] in that it instantiates variants of kernel programs based on parametric input tensor shapes and, at the same time, provides test input data along with reference results to ensure correctness.

3.5.4 Experimental Results

The first set of experiments presented by Lopoukhine et al. [67] have the goal of assessing the performance of kernels expressed using low-level RISC-V dialects. Sum, ReLU, and MatMulT kernels on 32 bit FP elements are expressed using the `snitch_stream`, `rv_snitch` and structured `rv` dialects, and lowered to assembly using the presented backend passes. The Sum and ReLU kernels display similar performance behavior and attain 95% FPU utilization. These kernels are element-wise operations on one or two operands, have no reductions, and operate in linear manner, resulting in a minimal and constant overhead for the accelerator setup and a simpler control flow structure, reaching near-100% FPU utilization. The MatMulT kernel reaches 74% FPU utilization, but only attains a throughput of 2.45 FLOP/cycle. All MLIR kernels match the performance of the optimized, handwritten assembly versions, proving that the low-level MLIR dialects representing both

Figure 3.16: Compiler continuous testing and benchmarking pipeline. Vertices represent tasks, edges represent data dependencies. Tasks in the form `kernel_generate_`$*$ drive the parametric kernel generator to explore the space of input tensor shapes. `optimization_pipelines` generates variants of the lowering pipeline by incremental addition of optimizations. All tasks downstream of `verilator` (that runs the actual simulation) are devoted to post-processing of execution traces and computation of performance counters.

standard and Snitch RISC-V ISAs are capable of matching expert-written and tuned kernels.

Register allocation is a critical task for compiler backends: the second set of experiments has the goal of assessing the feasibility of a linear, spill-free register allocator operating directly on MLIR. The key realization is avoiding the management of register spilling altogether, due to the high-performance nature of the considered micro-kernels: spilling to L1 memory (or even to global memory) would prevent any usage of Snitch ISA features altogether, making the generated code automatically inefficient. The presented experimental evaluation analyzes the effectiveness of this approach across various data types and shape sizes to assess its suitability. For double-precision kernels, register pressure remains manageable, with available registers left unallocated even considering the reserved floating point registers required by SSRs. For single-precision, packed-SIMD kernels, register usage is generally higher. Deeper loop nests due to higher-dimensional tensors and some peephole optimizations, like loop unrolling, are known sources of increased register pressure [212]. Despite these combined factors, the measurements reported by the authors show how spill-free register allocation is fit for high-performance linear algebra micro-kernels.

The third set of experiments has the goal of assessing the capacity of the proposed multi-level compilation approach to generate target-optimized, high-efficiency kernels from high-level abstractions. The effectiveness of code produced by the proposed compiler backend is evaluated in comparison to code compiled with MLIR and a naiv C implementation compiled using Clang. The objective is to minimize kernel execution time while maximizing FPU utilization.

Empirical results indicate that both MLIR and Clang achieve similar performance, with a peak utilization of approximately 42%. This limitation arises because both compilation flows rely on the LLVM RISC-V backend which, as pointed out, is unable to automatically emit ISA extensions from input C or LLVM IR programs. Consequently, the generated assembly code exhibits severe inefficiencies such as explicit load/store operations and RAW hazards, leading to poor performance. While the Max Pool kernel benefits the most from LLVM's backend optimizations, its FPU utilization remains below 50%. The fundamental insight is that despite leveraging high-level, domain-specific MLIR optimizations, the LLVM IR and backend dictate and ultimately constrain the micro-kernels' performance.

In contrast, the proposed compilation strategy achieves high FPU utilization even for smaller kernel sizes, reaching up to 90%. For element-wise kernels (Sum, Fill, and ReLU), FPU utilization increases with input size, approaching 100%. Reduction kernels (Conv, Max Pool, and Sum Pool) also experience an increase in FPU utilization as input width grows, though at a slower rate, stabilizing between 70-80%. All kernels are modified to handle four reductions simultaneously, resulting in an outlier of approximately 90%

utilization when $N = 4$, due to the elimination of the outermost loop overhead and dimensional reduction in accelerator setup. The remaining kernels exhibit behavior analogous to their parallel counterparts, with utilization rising steadily as kernel width increases.

Further measurements are conducted on the MatMul kernel to assess performance across a broader range of input shapes (Figure 3.17, Figure 3.18, and Figure 3.19) and to evaluate the impact of stream setup instructions. When either the inner dimension or the number of columns in the second operand is minimal, accelerator setup costs dominate execution time, leading to a lower throughput that does not exceed 80% of the theoretical peak. As input sizes increase, throughput improves, as setup and function call overheads become negligible relative to computation time.

To evaluate the impact of individual high- and low-level optimizations, authors incrementally applied optimization passes to the MatMul kernel (Table 3.2). The kernel consists of two `linalg.generic` operations: one for initializing the output matrix to zero and another for performing matrix multiplication. The baseline pipeline, which applies direct lowering without schedule optimizations while targeting the standard RISC-V ISA, results in an FPU occupancy of less than 3%, a performance degradation of approximately 36× compared to the fully optimized pipeline.

The introduction of SSRs halves the cycle count. While the number of explicit load/store instructions is reduced by 66%, overall performance remains suboptimal. The introduction of additional optimizations like `Scalar Replacement` enhances performance by a factor of over 4×, as explicit memory operations are hoisted and minimized. Applying `FRep` at this stage provides only marginal improvement, as execution time remains dominated by the innermost loop's dot product computation rather than loop setup overhead. Moreover, `Unroll-and-Jam` allows to break cross-iteration RAW dependencies in the reduction dimension and pipeline stalls are eliminate by unrolling and interleaving with a depth of at least five iterations.

The combined application of high- and low-level optimization passes results in near-optimal performance for linear algebra micro-kernels, with FPU utilization ranging between 73% and 90%. The multi-level approach to backend construction proposed by Lopoukhine et al. [67] effectively utilizes the high-level semantics embedded in `linalg.generic` operations to target custom ISA extensions, such as SSR and FREP in the Snitch architecture.

The results presented by the authors clearly demonstrate that micro-kernels expressed as assembly-level dialects can be effectively compiled to the Snitch accelerator, reaching up to 95% of the theoretical peak performance. At the same time, the proposed register allocation shows how, by being consistently spill-free, high-performance micro-kernels don't benefit from sophisticated spilling management. Finally, experiments show how the presented multi-level compiler backend can efficiently target the Snitch from a higher-level DSL obtaining 90% FPU utilization.

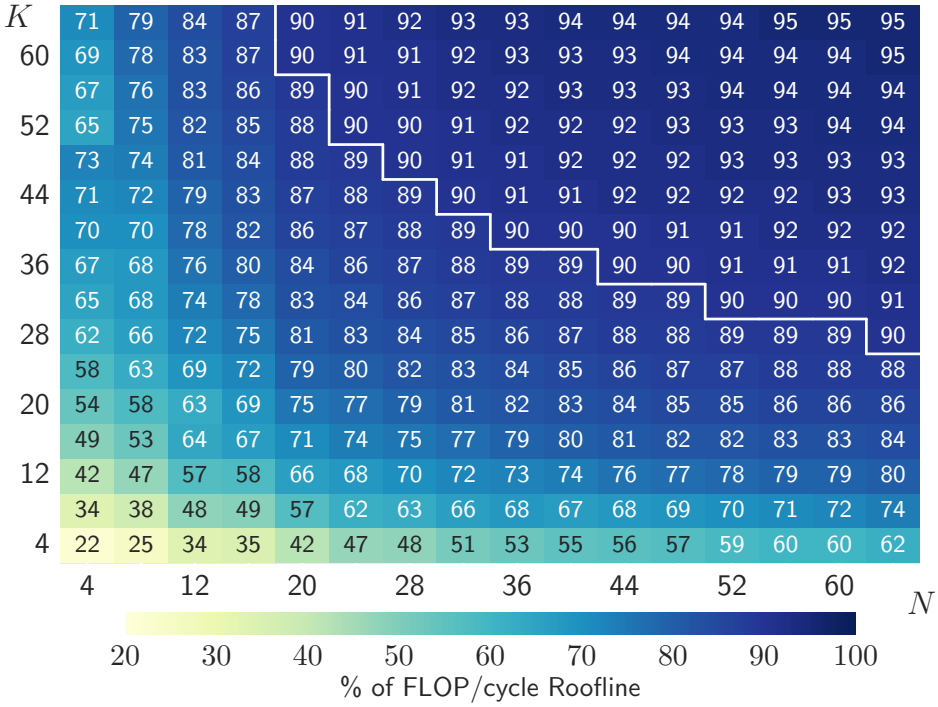


Figure 3.17: Sustained throughput of the double precision MatMul kernel ($C_{M\times N} = A_{M\times K}B_{K\times N}$ when $M = 1$). When compiled via the proposed approach, it achieves a throughput of over 90% ($\geq$ 1.8 FLOPs/cycle) of the theoretical peak (above the white border) as shape sizes increase, indicating that the computation offsets constant overheads. This figure is from Lopoukhine et al. [67].

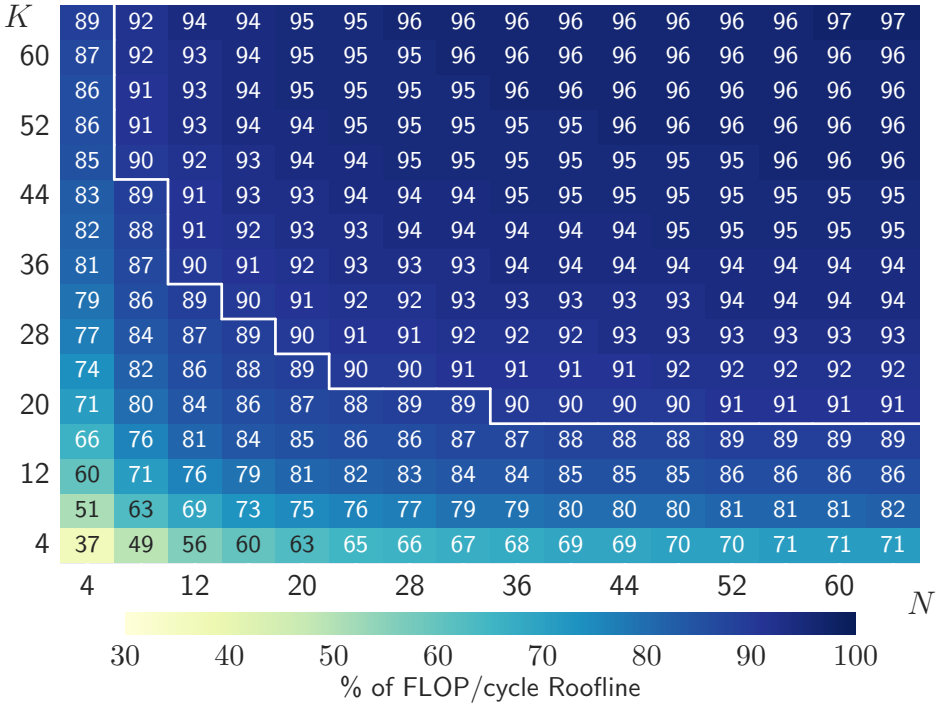


Figure 3.18: Sustained throughput of the double precision MatMul kernel ($C_{M\times N} = A_{M\times K}B_{K\times N}$ when $M = 4$). When compiled via the proposed approach, it achieves a throughput of over 90% ($\geq$ 1.8 FLOPs/cycle) of the theoretical peak (above the white border) as shape sizes increase, indicating that the computation offsets constant overheads. This figure is from Lopoukhine et al. [67].

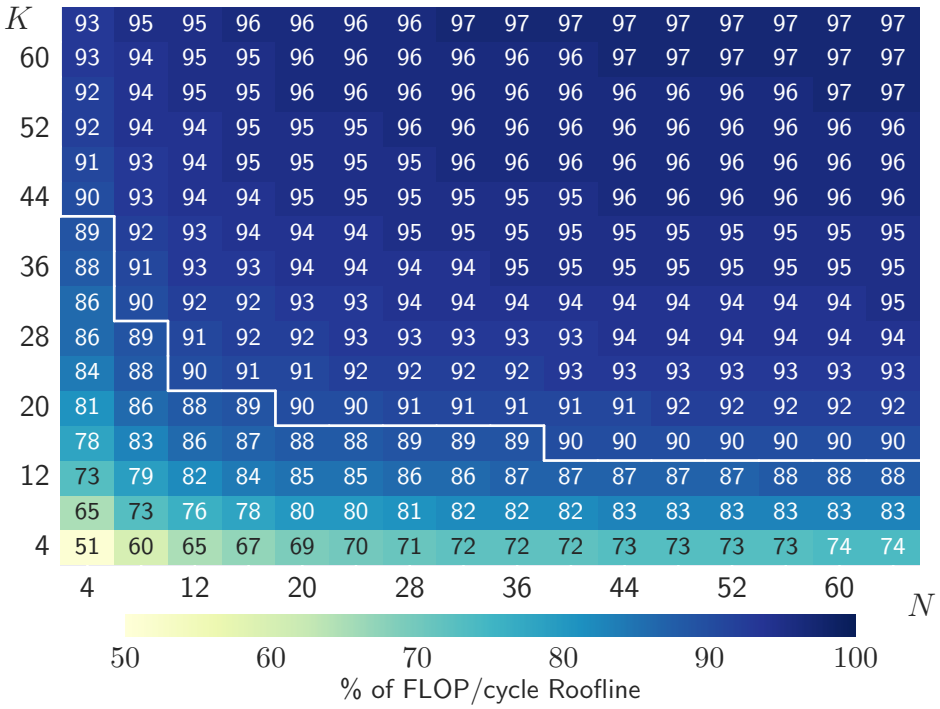


Figure 3.19: Sustained throughput of the double precision MatMul kernel ($C_{M\times N} = A_{M\times K}B_{K\times N}$ when $M = 8$). When compiled via the proposed approach, it achieves a throughput of over 90% ($\geq$ 1.8 FLOPs/cycle) of the theoretical peak (above the white border) as shape sizes increase, indicating that the computation offsets constant overheads. This figure is from Lopoukhine et al. [67].

3.6 Related Work

Traditional compiler designs have converged toward a three-tier architecture, effectively separating implementation concerns and enabling independent resolution of problems; tasks like instruction selection and register allocation are traditionally pertaining to the backend. However, this approach has resulted in increasingly complex and monolithic compiler stages.

Related to the work by Lopoukhine et al. [67] is another micro-kernel compiler by Castello et al. [252] that leverages a DSL [253] but is constrained to matrix multiplication and relies on manual user input for scheduling, data management, and transformations. In contrast, the authors' approach automates these processes across a broader spectrum of linear algebra operations.

Certain approaches within the MLIR ecosystem have explored the generation of optimized code tailored to specific micro-architectures, yet they frequently integrate existing general-purpose compiler backends. Notably, Bondhugula [232] applies high-level loop transformations within MLIR to reproduce optimizations known to be effective in producing basic linear algebra subprograms (BLAS) kernels for Intel CPUs. Kuzma et al. [254] utilize an end-to-end compilation process with C/C++ and LLVM IR types, while Varoumas [255] focuses on ARM-based architectures through profile-driven transformations exploration in the `vector` dialect. On the other hand, Lopoukhine et al. [67] claim that their approach represents the first MLIR-based compiler backend that uses accelerator-specific abstractions while, at the same time, implementing a complete compiler backend.

Several library-based approaches [218, 216, 217] rely on ahead-of-time (AOT), just-in-time (JIT) compilation or kernel templates augmented with parameter space exploration to provide the best kernel corpus for a specific target micro-architecture. None of these approaches provide the same flexibility of a multi-level compiler backend.

Recent years have seen a proliferation of domain-specific languages and corresponding tooling aimed at managing computational aspects such as scheduling and memory placement [30, 256, 257, 29, 229, 258, 259, 260, 253], influenced by methodologies like Halide [29]. This trend has also fostered the integration of autotuning and analytical techniques for optimizing configuration choices [261, 262, 263, 34]. These DSLs often replicate tightly coupled compiler infrastructures [34, 264], depend on general-purpose compiler backends for actual code generation [253], or necessitate external tools for interoperability [265]. In contrast, the approach presented by Lopoukhine et al. [67] is fully integrated in the MLIR ecosystem, and leverages its open and reusable infrastructure.

Table 3.2: Incremental performance improvements by optimization passes from the proposed compilation pipeline. The prototype backend achieves over 90% FPU occupancy for the MatMul kernel, operating on 1×200 and 200×5 64 bit inputs. Incrementally adding each optimization minimizes and, eventually eliminates, explicit memory operations, while reducing execution time (cycles) and maximizing FPU utilization. This table is from Lopoukhine et al. [67].

Optimizations	**Allocated Registers (#)**		**Assembly Operations (#)**				**Performance**	
	FP	**Integer**	**Loads**	**Stores**	**FMAdd**	**FRep**	**Cycles (#)**	**Occupancy (%)**
Baseline (for MatMul)	3/20	13/15	3 000	1 005	1 000	0	40 161	2.49
`+ Streams`	3/20	11/15	1 000	1 000	1 000	0	19 165	5.25
`+ Scalar Replacement`	3/20	10/15	5	5	1 000	0	4 147	24.28
`+ FRep`	3/20	9/15	5	5	1 000	2	4 124	24.42
`+ Fuse Fill`	5/20	8/15	0	0	1 000	1	4 130	24.5
`+ Unroll-and-Jam`	8/20	7/15	0	0	1 000	1	1 115	**90.67**

3.7 Compiling at the *End of Moore's Law*: Conclusion

The initial motivation for the work by Lopoukhine et al. [67] is the realization that the strict separation of frontends and backends in the design of modern general-purpose compilers results in an information bottleneck between high-level code abstractions and targeted novel hardware features. This leads to performance experts often bypassing the compiler backend altogether with hand-written kernels, often written in assembly language or from templates. The cost and effort in hand-tuning kernels is exacerbated by a recent proliferation of specialized hardware, driven by the breakdown of Moore's law and Dennard scaling in modern silicon technologies. The MLIR-based prototype presented by Lopoukhine et al. [67] showcases an efficient methodology for creating modular and expressive compiler backends that combine domain knowledge with hardware capabilities. Authors show rhat a novel approach to compiler backend construction, based on a structured, abstraction-driven strategy with multi-level SSA-based IRs, is able to generate high-performance assembly kernels for RISC-V accelerators on a set of real-world workloads from the DL domain. Given the steep learning curve posed by the Snitch assembly-based programming model and the complexity of analyses and transformations needed to leverage its unique ISA features, the proposed approach is a valuable prospect to unlock the performance of novel hardware platforms both in industry and computing architectures research.

Final Conclusions

This thesis presented published works, along with my contributions, in an attempt at answering the questions posed by our *post-Moore* era.

> How task-based, embarrassingly-parallel scientific workloads like virtual screening can efficiently scale up to entire pre-exascale, state-of-the-art GPU-accelerated HPC systems?

Chapter 1 explored virtual screening of large molecular sets, a real-world example of an embarrassingly-parallel, task-based workload on pre-exascale HPC systems. Unlike tightly coupled GPU workloads common in classical HPC applications, this problem requires a different acceleration strategy. For GPU efficiency, a *latency*-optimized approach, which distributes a single task (ligand-protein pair) across multiple processing elements, improves task-level performance but hinders overall time-to-solution. On the other hand, a *throughput*-optimized strategy, despite increasing single-task latency, enhances GPU occupancy, achieving up to $5\times$ better throughput. Thus, maximizing GPU occupancy is crucial for handling task-based workloads efficiently. This is done by means of warp-synchronous kernels, static allocation of kernel resources and off-device load balancing. The presented works also demonstrated how extreme-scale virtual screening can accelerate drug discovery during global pandemics emergencies. Using the first GPU port of LiGen, authors screened over 70 billion ligands across 15 binding sites of 12 SARS-CoV-2 proteins. Running on both Marconi100 [155] at CINECA and HPC5 [156] at ENI S.p.A., Europe's two most powerful supercomputers at the time, aggregating around 81 PFLOP/s, the *one-trillion-docking experiment* was completed in just 60 hours of continuous production, the largest virtual screening campaign ever conducted.

> How an extreme-scale HPC system, based on a heterogeneous offering of highly specialized accelerators, can be sustainable?

If RISC-V can be an answer to the challenge of extreme hardware specialization, how future post-exascale systems based on RISC-V will look like?

Chapter 2 introduced *Monte Cimone*, a multi-blade computer prototype and hardware/software testbed, the first fully operational RISC-V cluster supporting a baseline HPC software stack. The heterogeneous cluster integrates a CPU-only partition and an accelerated partition based on a synthesized, scaled-down version of the Occamy [19] RISC-V accelerator, within a seamless HPC production stack. The extensive experimental evaluation campaign showed how mainstream HPC applications on *Monte Cimone* exhibit near-linear, full-system strong scaling, demonstrating a remarkable readiness of the whole HPC software stack. While *Monte Cimone* is not optimized for high floating-point performance, it serves to explore the integration of a multi-node RISC-V cluster with a full HPC production stack. The presented results highlight significant software and hardware maturity, suggesting that the first generation of RISC-V HPC machines may be closer than expected.

Can a multi-level IR represent RISC-V domain specific extensions for novel linear algebra accelerators?

Can the multi-level approach to compiler construction enable SSA compiler backends to generate high-performance kernels leveraging custom, application-specific hardware features?

Finally, in Chapter 3, Lopoukhine et al. [67] show how the strict separation of frontends and backends in modern general-purpose compilers creates an information bottleneck, limiting optimization for novel hardware. The effort originated from the goal of being able to close the gap between high-level linear algebra programs and Snitch [48], a RISC-V accelerator architecture designed by ETH Zurich to pursue extreme compute energy efficiency by means of novel features like stream-semantics registers to reach perfect, software-programmed prefetching, and floating point hardware loops to elide control flow. The same architecture is the building block of the Occamy [19] RISC-V accelerator deployed on *Monte Cimone* (Chapter 2). The presented MLIR-based prototype kernel compiler explores a backend methodology that integrates domain knowledge with hardware capabilities. With the support of an extensive experimental evaluation, authors demonstrate that a structured, abstraction-driven approach to compiler backend construction, based on a multi-level SSA-based IRs, is able to generate high-performance assembly kernels for the Snitch architecture. Given the steep learning curve of the Snitch assembly-based programming model and the complexity of required analyses and transformations, this approach offers a promising prospect to

unlock performance for emerging domain-specific hardware in both industry and computing research.

Bibliography

[1] Charles E. Leiserson, Neil C. Thompson, Joel S. Emer, Bradley C. Kuszmaul, Butler W. Lampson, Daniel Sanchez, and Tao B. Schardl. "There's Plenty of Room at the Top: What Will Drive Computer Performance after Moore's Law?" In: *Science* 368.6495 (June 5, 2020), eaam9744. DOI: 10.1126/science.aam9744.

[2] John L. Hennessy and David A. Patterson. "A New Golden Age for Computer Architecture". In: *Communications of the ACM* 62.2 (Jan. 28, 2019), pp. 48–60. DOI: 10.1145/3282307.

[3] Thomas N. Theis and H.-S. Philip Wong. "The End of Moore's Law: A New Beginning for Information Technology". In: *Computing in Science & Engineering* 19.2 (Mar. 2017), pp. 41–50. DOI: 10.1109/MCSE.2017.29.

[4] Elie Track, Nancy Forbes, and George Strawn. "The End of Moore's Law". In: *Computing in Science & Engineering* 19.2 (Mar. 2017), pp. 4–6. DOI: 10.1109/MCSE.2017.25.

[5] Matteo Turisini, Giorgio Amati, and Mirko Cestari. *LEONARDO: A Pan-European Pre-Exascale Supercomputer for HPC and AI Applications*. July 31, 2023. URL: http://arxiv.org/abs/2307.16885. Pre-published.

[6] LUMI Consortium. *The LUMI Supercomputer*. URL: https://www.lumi-supercomputer.eu/lumi_supercomputer/.

[7] Elon Musk. *This Weekend, the @xAI Team Brought Our Colossus 100k H100 Training Cluster Online. From Start to Finish, It Was Done in 122 Days. Colossus Is the Most Powerful AI Training System in the World. Moreover, It Will Double in Size to 200k (50k H200s) in a Few Months. Excellent Work by the Team, Nvidia and Our Many Partners/Suppliers.* x.com. Sept. 2, 2024. URL: https://x.com/elonmusk/status/1830650370336473253.

[8] Mark Zuckerberg. *Some Updates on Our AI Efforts. Our Long Term Vision Is to Build General Intelligence, Open Source It Responsibly, and Make It Widely Available so Everyone Can Benefit. We're Bringing Our Two Major AI Research Efforts (FAIR and GenAI) Closer*

Together to Support This. We're Currently Training Our next-Gen Model Llama 3, and We're Building Massive Compute Infrastructure to Support Our Future Roadmap, Including 350k H100s by the End of This Year – and Overall Almost 600k H100s Equivalents of Compute If You Include Other GPUs. Also Really Excited about Our Progress Building New AI-centric Computing Devices like Ray Ban Meta Smart Glasses. Lots More to Come Soon. Instagram. Jan. 18, 2024. URL: https://www.instagram.com/p/C2QARHJR1sZ/.

[9] Andrew A. Chien and Vijay Karamcheti. "Moore's Law: The First Ending and a New Beginning". In: *Computer* 46.12 (Dec. 2013), pp. 48–53. DOI: 10.1109/MC.2013.431.

[10] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. "Dark Silicon and the End of Multicore Scaling". In: *Proceedings of the 38th Annual International Symposium on Computer Architecture*. ISCA '11. San Jose, California, USA: Association for Computing Machinery, June 4, 2011, pp. 365–376. DOI: 10.1145/2000064.2000108.

[11] Michael B. Taylor. "Is Dark Silicon Useful? Harnessing the Four Horsemen of the Coming Dark Silicon Apocalypse". In: *Proceedings of the 49th Annual Design Automation Conference*. DAC '12: The 49th Annual Design Automation Conference 2012. San Francisco California: ACM, June 3, 2012, pp. 1131–1136. DOI: 10.1145/2228360.2228567.

[12] *TOP500 Supercomputer Sites*. URL: https://www.top500.org/.

[13] "Intel's 7nm Slip Raises Questions About Ponte Vecchio GPU, Aurora Supercomputer". In: *HPC Wire* (July 30, 2020).

[14] Luca Benini. "Open Platforms for Energy-Efficient Scalable Computing". Invited talk. The International Conference for High Performance Computing, Networking, Storage, and Analysis (Dallas, TX, USA). Nov. 17, 2022.

[15] Norman P. Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, et al. "TPU v4: An Optically Reconfigurable Supercomputer for Machine Learning with Hardware Support for Embeddings". Version 1. In: (2023). DOI: 10.48550/ARXIV.2304.01433.

[16] Gokul Krishnan, Sumit K. Mandal, Chaitali Chakrabarti, Jae-sun Seo, Umit Y. Ogras, and Yu Cao. "In-Memory Computing for AI Accelerators: Challenges and Solutions". In: *Embedded Machine Learning for Cyber-Physical, IoT, and Edge Computing*. Ed. by Sudeep Pasricha and Muhammad Shafique. Cham: Springer International Publishing, 2024, pp. 199–224. DOI: 10.1007/978-3-031-19568-6_7.

[17] Zhe Jia, Blake Tillman, Marco Maggioni, and Daniele Paolo Scarpazza. "Dissecting the Graphcore IPU Architecture via Microbenchmarking". Dec. 6, 2019.

[18] Gary Lauterbach. "The Path to Successful Wafer-Scale Integration: The Cerebras Story". In: *IEEE Micro* 41.6 (Nov. 1, 2021), pp. 52–57. DOI: 10.1109/MM.2021.3112025.

[19] Gianna Paulin, Paul Scheffler, Thomas Benz, Matheus Cavalcante, Tim Fischer, Manuel Eggimann, Yichao Zhang, Nils Wistoff, Luca Bertaccini, Luca Colagrande, et al. "Occamy: A 432-Core 28.1 DP-GFLOP/s/W 83% FPU Utilization Dual-Chiplet, Dual-HBM2E RISC-V-Based Accelerator for Stencil and Sparse Linear Algebra Computations with 8-to-64-Bit Floating-Point Support in 12nm FinFET". In: *2024 IEEE Symposium on VLSI Technology and Circuits (VLSI Technology and Circuits)*. 2024 IEEE Symposium on VLSI Technology and Circuits (VLSI Technology and Circuits). Honolulu, HI, USA: IEEE, June 16, 2024, pp. 1–2. DOI: 10.1109/VLSITechnologyandCir46783.2024.10631529.

[20] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S. Vetter. "NVIDIA Tensor Core Programmability, Performance & Precision". In: *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). Vancouver, BC: IEEE, May 2018, pp. 522–531. DOI: 10.1109/IPDPSW.2018.00091.

[21] Melissa Riddle, Tom Sorensen, and Earl Joseph. *Forecast Update: GPU and Accelerator Growth in HPC*. Hyperion Research, Feb. 2023.

[22] Baolin Li, Rohin Arora, Siddharth Samsi, Tirthak Patel, William Arcand, David Bestor, Chansup Byun, Rohan Basu Roy, Bill Bergeron, John Holodnak, et al. "AI-Enabling Workloads on Large-Scale GPU-Accelerated System: Characterization, Opportunities, and Implications". In: *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA). Seoul, Korea, Republic of: IEEE, Apr. 2022, pp. 1224–1237. DOI: 10.1109/HPCA53966.2022.00093.

[23] Shidi Tang, Ruiqi Chen, Mengru Lin, Qingde Lin, Yanxiang Zhu, Ji Ding, Haifeng Hu, Ming Ling, and Jiansheng Wu. "Accelerating AutoDock Vina with GPUs". In: *Molecules* 27.9 (May 9, 2022), p. 3041. DOI: 10.3390/molecules27093041.

[24] 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* (2022), pp. 1–12. DOI: 10.1109/TETC.2022.3187134.

[25] Joseph C. Jacob, Daniel S. Katz, G. Bruce Berriman, John C. Good, Anastasia C. Laity, Ewa Deelman, Carl Kesselman, Gurmeet Singh, Mei Hui Su, Thomas A. Prince, and Roy Williams. "Montage: A Grid Portal and Software Toolkit for Science-Grade Astronomical Image Mosaicking". In: *International Journal of Computational Science and Engineering* 4.2 (2009), p. 73. DOI: 10.1504/IJCSE.2009.026999.

[26] S. Agostinelli, J. Allison, K. Amako, J. Apostolakis, H. Araujo, P. Arce, M. Asai, D. Axen, S. Banerjee, G. Barrand, et al. "Geant4—a Simulation Toolkit". In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 506.3 (July 2003), pp. 250–303. DOI: 10.1016/S0168-9002(03)01368-8.

[27] Ben Langmead and Steven L Salzberg. "Fast Gapped-Read Alignment with Bowtie 2". In: *Nature Methods* 9.4 (Apr. 2012), pp. 357–359. DOI: 10.1038/nmeth.1923.

[28] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. *Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions*. June 28, 2018. URL: http://arxiv.org/abs/1802.04730. Pre-published.

[29] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. "Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines". In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI '13*. The 34th ACM SIGPLAN Conference. Seattle, Washington, USA: ACM Press, 2013, p. 519. DOI: 10.1145/2491956.2462176.

[30] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 578–594.

[31] Intel. *PlaidML*. Intel, 2017.

[32] Chris Leary and Todd Wang. "XLA: TensorFlow, Compiled!" TensorFlow Dev Summit. 2017.

[33] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, et al. "Glow: Graph Lowering Compiler Techniques for Neural Networks". Apr. 3, 2019.

[34] Philippe Tillet, H. T. Kung, and David Cox. "Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations". In: *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. PLDI '19: 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. Phoenix AZ USA: ACM, June 22, 2019, pp. 10–19. DOI: 10.1145/3315508.3329973.

[35] C. Lattner and V. Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* International Symposium on Code Generation and Optimization, 2004. CGO 2004. San Jose, CA, USA: IEEE, 2004, pp. 75–86. DOI: 10.1109/CGO.2004.1281665.

[36] Jason Merrill. *GENERIC and GIMPLE: A New Tree Representation for Entire Functions*. 2003.

[37] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems". Mar. 16, 2016.

[38] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. Dec. 3, 2019. URL: http://arxiv.org/abs/1912.01703. Pre-published.

[39] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. "MLIR: A Compiler Infrastructure for the End of Moore's Law". Feb. 29, 2020.

[40] Scott Cyphers, Arjun K. Bansal, Anahita Bhiwandiwalla, Jayaram Bobba, Matthew Brookhart, Avijit Chakraborty, Will Constable, Christian Convey, Leona Cook, Omar Kanawi, et al. "Intel nGraph: An Intermediate Representation, Compiler, and Executor for Deep Learning". Jan. 29, 2018.

[41] Sun C. Chan, Guang R. Gao, Barbara Chapman, Tony Linthicum, and Anshuman Dasgupta. "Open64 Compiler Infrastructure for Emerging Multicore/Manycore Architecture All Symposium Tutorial". In: *2008 IEEE International Symposium on Parallel and Distributed Processing*. Distributed Processing Symposium (IPDPS). Miami, FL, USA: IEEE, Apr. 2008, pp. 1–1. DOI: 10.1109/IPDPS.2008.4536577.

[42] Mike Murphy. "NVIDIA's Experience with Open64". In: IEEE/ACM International Symposium on Code Generation and Optimization (CGO). Boston, MA, Apr. 16, 2008.

[43] William J. Starke, Brian W. Thompto, Jeff A. Stuecheli, and Jose E. Moreira. "IBM's POWER10 Processor". In: *IEEE Micro* 41.2 (Mar. 2021), pp. 7–14. DOI: 10.1109/MM.2021.3058632.

[44] Dounia Khaldi, Yuanke Luo, Bing Yu, Alexey Sotkin, Bruno Morais, and Milind Girkar. "Extending LLVM IR for DPC++ Matrix Support: A Case Study with Intel ® Advanced Matrix Extensions (Intel ® AMX)". In: *2021 IEEE/ACM 7th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*. 2021 IEEE/ACM 7th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC). St. Louis, MO, USA: IEEE, Nov. 2021, pp. 20–26. DOI: 10.1109/LLVMHPC54804.2021.00008.

[45] Finn Wilkinson and Simon McIntosh-Smith. "An Initial Evaluation of Arm's Scalable Matrix Extension". In: *2022 IEEE/ACM International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. 2022 IEEE/ACM International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS). Dallas, TX, USA: IEEE, Nov. 2022, pp. 135–140. DOI: 10.1109/PMBS56514.2022.00018.

[46] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. *The Rocket Chip Generator*. UCB/EECS-2016-17. Apr. 2016.

[47] Davide Rossi, Antonio Pullini, Igor Loi, Michael Gautschi, Frank Kagan Gurkaynak, Adam Teman, Jeremy Constantin, Andreas Burg, Ivan Miro-Panades, Edith Beigne, et al. "Energy-Efficient Near-Threshold Parallel Computing: The PULPv2 Cluster". In: *IEEE Micro* 37.5 (Sept. 2017), pp. 20–31. DOI: 10.1109/MM.2017.3711645.

[48] Florian Zaruba, Fabian Schuiki, Torsten Hoefler, and Luca Benini. "Snitch: A Tiny Pseudo Dual-Issue Processor for Area and Energy Efficient Execution of Floating-Point Intensive Workloads". In: *IEEE*

Transactions on Computers 70.11 (Nov. 1, 2021), pp. 1845–1860. DOI: 10.1109/TC.2020.3027900.

[49] Blaise Tine, Fares Elsabbagh, Krishna Yalamarthy, and Hyesoon Kim. "Vortex: Extending the RISC-V ISA for GPGPU and 3D-GraphicsResearch". Oct. 20, 2021.

[50] Chen Chen, Xiaoyan Xiang, Chang Liu, Yunhai Shang, Ren Guo, Dongqi Liu, Yimin Lu, Ziyi Hao, Jiahui Luo, Zhijian Chen, et al. "Xuantie-910: A Commercial Multi-Core 12-Stage Pipeline Out-of-Order 64-Bit High Performance RISC-V Processor with Vector Extension : Industrial Product". In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA). Valencia, Spain: IEEE, May 2020, pp. 52–64. DOI: 10.1109/ISCA45697.2020.00016.

[51] Dylan Patel. *Tenstorrent Wormhole Analysis - A Scale Out Architecture for Machine Learning That Could Put Nvidia On Their Back Foot*. SemiAnalysis. 2021. URL: https://www.semianalysis.com/p/tenstorrent-wormhole-analysis-a-scale.

[52] Ventana Micro. *Ventana Veyron V1*. 2023. URL: https://www.ventanamicro.com/technology/risc-v-cpu-ip/.

[53] Amin Firoozshahian, Joel Coburn, Roman Levenstein, Rakesh Nattoji, Ashwin Kamath, Olivia Wu, Gurdeepak Grewal, Harish Aepala, Bhasker Jakka, Bob Dreyer, et al. "MTIA: First Generation Silicon Targeting Meta's Recommendation Systems". In: *Proceedings of the 50th Annual International Symposium on Computer Architecture*. ISCA '23: 50th Annual International Symposium on Computer Architecture. Orlando FL USA: ACM, June 17, 2023, pp. 1–13. DOI: 10.1145/3579371.3589348.

[54] Stavros Kalapothas, Manolis Galetakis, Georgios Flamis, Fotis Plessas, and Paris Kitsos. "A Survey on RISC-V-Based Machine Learning Ecosystem". In: *Information* 14.2 (Jan. 21, 2023), p. 64. DOI: 10.3390/info14020064.

[55] The European Processor Initiative. *The European Processor Initiative Accelerator Processor Stream*. The European Processor Initiative. 2023. URL: https://www.european-processor-initiative.eu/accelerator/.

[56] Sally Ward-Foxton. "Jim Keller on AI, RISC-V, Tenstorrent's Move to Edge IP". In: *EE Times* (Sept. 6, 2023).

[57] Antoine Petitet, Clint Whaley, and Jack Dongarra. *HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers*. Innovative Computing Laboratory, 2008.

[58] Jack Dongarra, Michael A Heroux, and Piotr Luszczek. "High-Performance Conjugate-Gradient Benchmark: A New Metric for Ranking High-Performance Computing Systems". In: *The International Journal of High Performance Computing Applications* 30.1 (Feb. 2016), pp. 3–10. DOI: 10.1177/1094342015593158.

[59] 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* (Dec. 2023), p. 104819. DOI: 10.1016/j.jpdc.2023.104819.

[60] Andrew Emerson, Federico Ficarelli, Gianluca Palermo, and Francesco Frigerio. "The High-Performance Computing Resources for the EXSCALATE4CoV Project". In: *Exscalate4CoV*. Ed. by Silvano Coletti and Gabriella Bernardi. Cham: Springer International Publishing, 2023, pp. 27–34. DOI: 10.1007/978-3-031-30691-4_4.

[61] Andrea R. Beccari, Carlo Cavazzoni, Claudia Beato, and Gabriele Costantino. "LiGen: A High Performance Workflow for Chemistry Driven de Novo Design". In: *Journal of Chemical Information and Modeling* 53.6 (June 24, 2013), pp. 1518–1527. DOI: 10.1021/ci400078g.

[62] Giulio Vistoli, Candida Manelfi, Carmine Talarico, Anna Fava, Arieh Warshel, Igor V. Tetko, Rossen Apostolov, Yang Ye, Chiara Latini, Federico Ficarelli, et al. "MEDIATE - Molecular DockIng at homE: Turning Collaborative Simulations into Therapeutic Solutions". In: *Expert Opinion on Drug Discovery* (July 10, 2023), pp. 1–13. DOI: 10.1080/17460441.2023.2221025.

[63] Andrea Bartolini, Federico Ficarelli, Emanuele Parisi, Francesco Beneventi, Francesco Barchi, Daniele Gregori, Fabrizio Magugliani, Marco Cicala, Cosimo Gianfreda, Daniele Cesarini, et al. "Monte Cimone: Paving the Road for the First Generation of RISC-V High-Performance Computers". In: *2022 IEEE 35th International System-on-Chip Conference (SOCC)*. 2022 IEEE 35th International System-on-Chip Conference (SOCC). Belfast, United Kingdom: IEEE, Sept. 5, 2022, pp. 1–6. DOI: 10.1109/SOCC56010.2022.9908096.

[64] Paolo Giannozzi, Oscar Baseggio, Pietro Bonfà, Davide Brunato, Roberto Car, Ivan Carnimeo, Carlo Cavazzoni, Stefano de Gironcoli, Pietro Delugas, Fabrizio Ferrari Ruffino, et al. "Quantum ESPRESSO toward the Exascale". In: *The Journal of Chemical Physics* 152.15 (Apr. 21, 2020), p. 154105. DOI: 10.1063/5.0005082.

[65] Federico Ficarelli, Andrea Bartolini, Emanuele Parisi, Francesco Beneventi, Francesco Barchi, Daniele Gregori, Fabrizio Magugliani, Marco Cicala, Cosimo Gianfreda, Daniele Cesarini, et al. "Meet Monte Cimone: Exploring RISC-V High Performance Compute Clusters". In: *Proceedings of the 19th ACM International Conference on Computing Frontiers*. CF '22: 19th ACM International Conference on Computing Frontiers. Turin Italy: ACM, May 17, 2022, pp. 207–208. DOI: 10.1145/3528416.3530869.

[66] Federico Ficarelli. "Monte Cimone: Towards RISC-V High Performance Compute Clusters". In: HiPEAC 23. Tolouse, France, Jan. 18, 2023.

[67] 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: IEEE/ACM International Symposium on Code Generation and Optimization (CGO). 2025.

[68] Renato Golin, Lorenzo Chelini, Adam Siemieniuk, Kavitha Madhu, Niranjan Hasabnis, Hans Pabst, Evangelos Georganas, and Alexander Heinecke. *Towards a High-Performance AI Compiler with Upstream MLIR*. Apr. 15, 2024. URL: http://arxiv.org/abs/2404.15204. Pre-published.

[69] Gordon E. Moore. "Cramming More Components onto Integrated Circuits, Reprinted from Electronics, Volume 38, Number 8, April 19, 1965, Pp.114 Ff." In: *IEEE Solid-State Circuits Society Newsletter* 11.3 (Sept. 2006), pp. 33–35. DOI: 10.1109/N-SSC.2006.4785860.

[70] Gordon E. Moore. "Progress in Digital Integrated Electronics [Technical Literaiture, Copyright 1975 IEEE. Reprinted, with Permission. Technical Digest. International Electron Devices Meeting, IEEE, 1975, Pp. 11-13.]" In: *IEEE Solid-State Circuits Society Newsletter* 11.3 (Sept. 2006), pp. 36–37. DOI: 10.1109/N-SSC.2006.4804410.

[71] R.H. Dennard, F.H. Gaensslen, Hwa-Nien Yu, V.L. Rideout, E. Bassous, and A.R. LeBlanc. "Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions". In: *IEEE Journal of Solid-State Circuits* 9.5 (Oct. 1974), pp. 256–268. DOI: 10.1109/JSSC.1974.1050511.

[72] Mark Bohr. "A 30 Year Retrospective on Dennard's MOSFET Scaling Paper". In: *IEEE Solid-State Circuits Newsletter* 12.1 (Win. 2007), pp. 11–13. DOI: 10.1109/N-SSC.2007.4785534.

[73] Luiz André Barroso and Urs Hölzle. "The Case for Energy-Proportional Computing". In: *Computer* 40.12 (Dec. 2007), pp. 33–37. DOI: 10.1109/MC.2007.443.

[74] Shekhar Borkar and Andrew A. Chien. "The Future of Microprocessors". In: *Communications of the ACM* 54.5 (May 2011), pp. 67–77. DOI: 10.1145/1941487.1941507.

[75] Alexander A. Conklin and Suhas Kumar. "Solving the Big Computing Problems in the Twenty-First Century". In: *Nature Electronics* 6.7 (July 20, 2023), pp. 464–466. DOI: 10.1038/s41928-023-00985-1.

[76] Wu-chun Feng and Kirk Cameron. "The Green500 List: Encouraging Sustainable Supercomputing". In: *Computer* 40.12 (Dec. 2007), pp. 50–55. DOI: 10.1109/MC.2007.445.

[77] William J. Dally, Stephen W. Keckler, and David B. Kirk. "Evolution of the Graphics Processing Unit (GPU)". In: *IEEE Micro* 41.6 (Nov. 1, 2021), pp. 42–51. DOI: 10.1109/MM.2021.3113475.

[78] David Luebke. "CUDA: Scalable Parallel Programming for High-Performance Scientific Computing". In: *2008 5th IEEE International Symposium on Biomedical Imaging: From Nano to Macro*. 2008 5th IEEE International Symposium on Biomedical Imaging (ISBI 2008). Paris, France: IEEE, May 2008, pp. 836–838. DOI: 10.1109/ISBI.2008.4541126.

[79] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, and J.C. Phillips. "GPU Computing". In: *Proceedings of the IEEE* 96.5 (May 2008), pp. 879–899. DOI: 10.1109/JPROC.2008.917757.

[80] Wen-mei Hwu, David Kirk, and Izzat El Hajj. *Programming Massively Parallel Processors: A Hands-on Approach*. Fourth edition. Cambridge, MA: Elsevier : Morgan Kauffmann, 2023. 551 pp.

[81] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. "A Survey of General-Purpose Computation on Graphics Hardware". In: *Computer Graphics Forum* 26.1 (Mar. 2007), pp. 80–113. DOI: 10.1111/j.1467-8659.2007.01012.x.

[82] Michael Garland and David B. Kirk. "Understanding Throughput-Oriented Architectures". In: *Communications of the ACM* 53.11 (Nov. 2010), pp. 58–66. DOI: 10.1145/1839676.1839694.

[83] Chengbin Fan, Hui Deng, Feng Wang, Shoulin Wei, Wei Dai, and Bo Liang. "A Survey on Task Scheduling Method in Heterogeneous Computing System". In: *2015 8th International Conference on Intelligent Networks and Intelligent Systems (ICINIS)*. 2015 8th International Conference on Intelligent Networks and Intelligent Systems (ICINIS). Tianjin, China: IEEE, Nov. 2015, pp. 90–93. DOI: 10.1109/ICINIS.2015.42.

[84] Markus Steinberger, Bernhard Kainz, Bernhard Kerbl, Stefan Hauswiesner, Michael Kenzel, and Dieter Schmalstieg. "Softshell: Dynamic Scheduling on GPUs". In: *ACM Transactions on Graphics* 31.6 (Nov. 2012), pp. 1–11. DOI: 10.1145/2366145.2366180.

[85] Andrew Davidson, Sean Baxter, Michael Garland, and John D. Owens. "Work-Efficient Parallel GPU Methods for Single-Source Shortest Paths". In: *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. 2014 IEEE International Parallel & Distributed Processing Symposium (IPDPS). Phoenix, AZ, USA: IEEE, May 2014, pp. 349–359. DOI: 10.1109/IPDPS.2014.45.

[86] Muhammad Amber Hassaan, Martin Burtscher, and Keshav Pingali. "Ordered vs. Unordered: A Comparison of Parallelism and Work-Efficiency in Irregular Algorithms". In: *ACM SIGPLAN Notices* 46.8 (Sept. 7, 2011), pp. 3–12. DOI: 10.1145/2038037.1941557.

[87] Yuxin Chen, Benjamin Brock, Serban Porumbescu, Aydın Buluç, Katherine Yelick, and John D. Owens. *Atos: A Task-Parallel GPU Dynamic Scheduling Framework for Dynamic Irregular Computations*. Nov. 30, 2021. URL: http://arxiv.org/abs/2112.00132. Pre-published.

[88] Kshitij Gupta, Jeff A. Stuart, and John D. Owens. "A Study of Persistent Threads Style GPU Programming for GPGPU Workloads". In: *2012 Innovative Parallel Computing (InPar)*. 2012 Innovative Parallel Computing (InPar). San Jose, CA, USA: IEEE, May 2012, pp. 1–14. DOI: 10.1109/InPar.2012.6339596.

[89] Wilson W.L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow". In: *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*. 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007). Chicago, IL, USA: IEEE, 2007, pp. 407–420. DOI: 10.1109/MICRO.2007.30.

[90] Yuan Lin and Vinod Grover. *Using CUDA Warp-Level Primitives*. NVIDIA Technical Blog. Jan. 15, 2018. URL: https://developer.nvidia.com/blog/using-cuda-warp-level-primitives/.

[91] Julie R. Schames, Richard H. Henchman, Jay S. Siegel, Christoph A. Sotriffer, Haihong Ni, and J. Andrew McCammon. "Discovery of a Novel Binding Trench in HIV Integrase". In: *Journal of Medicinal Chemistry* 47.8 (Apr. 1, 2004), pp. 1879–1881. DOI: 10.1021/jm0341913.

[92] David E Clark. "What Has Virtual Screening Ever Done for Drug Discovery?" In: *Expert Opinion on Drug Discovery* 3.8 (Aug. 2008), pp. 841–851. DOI: 10.1517/17460441.3.8.841.

[93] A.M. MacConnachie. "Zanamivir (Relenza®) — A New Treatment for Influenza". In: *Intensive and Critical Care Nursing* 15.6 (Dec. 1999), pp. 369–370. DOI: 10.1016/S0964-3397(99)80031-7.

[94] Enrico Glaab. "Building a Virtual Ligand Screening Pipeline Using Free Software: A Survey". In: *Briefings in Bioinformatics* 17.2 (Mar. 2016), pp. 352–366. DOI: 10.1093/bib/bbv037.

[95] Todd J.A. Ewing, Shingo Makino, A. Geoffrey Skillman, and Irwin D. Kuntz. "DOCK 4.0: Search Strategies for Automated Molecular Docking of Flexible Molecule Databases". In: *Journal of Computer-Aided Molecular Design* 15.5 (2001), pp. 411–428. DOI: 10.1023/A:1011115820450.

[96] Garrett M. Morris, Ruth Huey, William Lindstrom, Michel F. Sanner, Richard K. Belew, David S. Goodsell, and Arthur J. Olson. "AutoDock4 and AutoDockTools4: Automated Docking with Selective Receptor Flexibility". In: *Journal of Computational Chemistry* 30.16 (Dec. 2009), pp. 2785–2791. DOI: 10.1002/jcc.21256.

[97] Richard A. Friesner, Jay L. Banks, Robert B. Murphy, Thomas A. Halgren, Jasna J. Klicic, Daniel T. Mainz, Matthew P. Repasky, Eric H. Knoll, Mee Shelley, Jason K. Perry, et al. "Glide: A New Approach for Rapid, Accurate Docking and Scoring. 1. Method and Assessment of Docking Accuracy". In: *Journal of Medicinal Chemistry* 47.7 (Mar. 1, 2004), pp. 1739–1749. DOI: 10.1021/jm0306430.

[98] Ingo Schellhammer and Matthias Rarey. "FlexX-Scan: Fast, Structure-based Virtual Screening". In: *Proteins: Structure, Function, and Bioinformatics* 57.3 (Nov. 15, 2004), pp. 504–517. DOI: 10.1002/prot.20217.

[99] René Thomsen and Mikael H. Christensen. "MolDock: A New Technique for High-Accuracy Molecular Docking". In: *Journal of Medicinal Chemistry* 49.11 (June 1, 2006), pp. 3315–3321. DOI: 10.1021/jm051197e.

[100] Gareth Jones, Peter Willett, Robert C Glen, Andrew R Leach, and Robin Taylor. "Development and Validation of a Genetic Algorithm for Flexible Docking 1 1Edited by F. E. Cohen". In: *Journal of Molecular Biology* 267.3 (Apr. 1997), pp. 727–748. DOI: 10.1006/jmbi.1996.0897.

[101] Ming Liu and Shaomeng Wang. "MCDOCK: A Monte Carlo Simulation Approach to the Molecular Docking Problem". In: *Journal of Computer-Aided Molecular Design* 13.5 (1999), pp. 435–451. DOI: 10.1023/A:1008005918983.

[102] P. Nuno Palma, Ludwig Krippahl, John E. Wampler, and José J.G. Moura. "BiGGER: A New (Soft) Docking Algorithm for Predicting Protein Interactions". In: *Proteins: Structure, Function, and Genetics* 39.4 (June 1, 2000), pp. 372–384. DOI: 10.1002/(SICI)1097-0134(20000601)39:4<372::AID-PROT100>3.0.CO;2-Q.

[103] Dong Dong, Zhijian Xu, Wu Zhong, and Shaoliang Peng. "Parallelization of Molecular Docking: A Review". In: *Current Topics in Medicinal Chemistry* 18.12 (Sept. 18, 2018), pp. 1015–1028. DOI: 10.2174/1568026618666180821145215.

[104] Xiaohua Zhang, Sergio E. Wong, and Felice C. Lightstone. "Message Passing Interface and Multithreading Hybrid for Parallel Molecular Docking of Large Databases on Petascale High Performance Computing Machines". In: *Journal of Computational Chemistry* 34.11 (Apr. 30, 2013), pp. 915–927. DOI: 10.1002/jcc.23214.

[105] Shuxing Zhang, Kamal Kumar, Xiaohui Jiang, Anders Wallqvist, and Jaques Reifman. "DOVIS: An Implementation for High-Throughput Virtual Screening Using AutoDock". In: *BMC Bioinformatics* 9.1 (Dec. 2008), p. 126. DOI: 10.1186/1471-2105-9-126.

[106] Mengran Fan, Jian Wang, Huaipan Jiang, Yilin Feng, Mehrdad Mahdavi, Kamesh Madduri, Mahmut T. Kandemir, and Nikolay V. Dokholyan. "GPU-Accelerated Flexible Molecular Docking". In: *The Journal of Physical Chemistry B* 125.4 (Feb. 4, 2021), pp. 1049–1060. DOI: 10.1021/acs.jpcb.0c09051.

[107] Bharat Sukhwani and Martin C. Herbordt. "GPU Acceleration of a Production Molecular Docking Code". In: *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*. GPGPU '09: Second Workshop on General-Purpose Computation on Graphics Processing Units. Washington D.C. USA: ACM, Mar. 8, 2009, pp. 19–27. DOI: 10.1145/1513895.1513898.

[108] Oliver Korb, Thomas Stützle, and Thomas E. Exner. "Accelerating Molecular Docking Calculations Using Graphics Processing Units". In: *Journal of Chemical Information and Modeling* 51.4 (Apr. 25, 2011), pp. 865–876. DOI: 10.1021/ci100459b.

[109] Ye Fang, Yun Ding, Wei P. Feinstein, David M. Koppelman, Juana Moreno, Mark Jarrell, J. Ramanujam, and Michal Brylinski. "Geaux-Dock: Accelerating Structure-Based Virtual Screening with Heterogeneous Computing". In: *PLOS ONE* 11.7 (July 15, 2016). Ed. by Alexandre G. De Brevern, e0158898. DOI: 10.1371/journal.pone.0158898.

[110] Irene Sánchez-Linares, Horacio Pérez-Sánchez, José M Cecilia, and José M García. "High-Throughput Parallel Blind Virtual Screening Using BINDSURF". In: *BMC Bioinformatics* 13.S14 (Sept. 2012), S13. DOI: 10.1186/1471-2105-13-S14-S13.

[111] Scott LeGrand, Aaron Scheinberg, Andreas F. Tillack, Mathialakan Thavappiragasam, Josh V. Vermaas, Rupesh Agarwal, Jeff Larkin, Duncan Poole, Diogo Santos-Martins, Leonardo Solis-Vasquez, et al. "GPU-Accelerated Drug Discovery with Docking on the Summit Supercomputer: Porting, Optimization, and Application to COVID-19 Research". In: *Proceedings of the 11th ACM International Conference on Bioinformatics, Computational Biology and Health Informatics*. BCB '20: 11th ACM International Conference on Bioinformatics, Computational Biology and Health Informatics. Virtual Event USA: ACM, Sept. 21, 2020, pp. 1–10. DOI: 10.1145/3388440.3412472.

[112] Diogo Santos-Martins, Leonardo Solis-Vasquez, Andreas F Tillack, Michel F Sanner, Andreas Koch, and Stefano Forli. "Accelerating A Uto D Ock 4 with GPUs and Gradient-Based Local Search". In: *Journal of Chemical Theory and Computation* 17.2 (Feb. 9, 2021), pp. 1060–1073. DOI: 10.1021/acs.jctc.0c01006.

[113] Leonardo Solis-Vasquez, Diogo Santos-Martins, Andreas F. Tillack, Andreas Koch, Jerome Eberhardt, and Stefano Forli. "Parallelizing Irregular Computations for Molecular Docking". In: *2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. 2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3). GA, USA: IEEE, Nov. 2020, pp. 12–21. DOI: 10.1109/IA351965.2020.00008.

[114] Jens Glaser, Josh V Vermaas, David M Rogers, Jeff Larkin, Scott LeGrand, Swen Boehm, Matthew B Baker, Aaron Scheinberg, Andreas F Tillack, Mathialakan Thavappiragasam, et al. "High-Throughput Virtual Laboratory for Drug Discovery Using Massive Datasets". In: *The International Journal of High Performance Computing Applications* 35.5 (Sept. 2021), pp. 452–468. DOI: 10.1177/10943420211001565.

[115] Yuejiang Yu, Chun Cai, Jiayue Wang, Zonghua Bo, Zhengdan Zhu, and Hang Zheng. "Uni-Dock: GPU-Accelerated Docking Enables Ultralarge Virtual Screening". In: *Journal of Chemical Theory and Computation* 19.11 (June 13, 2023), pp. 3336–3345. DOI: 10.1021/acs.jctc.2c01145.

[116] Gabin Schieffer and Ivy Peng. "Accelerating Drug Discovery in AutoDock-GPU with Tensor Cores". In: *Euro-Par 2023: Parallel Processing*. Ed. by José Cano, Marios D. Dikaiakos, George A. Papadopoulos, Miquel Pericàs, and Rizos Sakellariou. Vol. 14100. Cham: Springer Nature Switzerland, 2023, pp. 608–622. DOI: 10.1007/978-3-031-39698-4_41.

[117] Davide Gadioli, Gianluca Palermo, Stefano Cherubin, Emanuele Vitali, Giovanni Agosta, Candida Manelfi, Andrea R. Beccari, Carlo Cavazzoni, Nico Sanna, and Cristina Silvano. "Tunable Approximations to Control Time-to-Solution in an HPC Molecular Docking Mini-App". In: *The Journal of Supercomputing* 77.1 (Jan. 2021), pp. 841–869. DOI: 10.1007/s11227-020-03295-x.

[118] Sebastian Ruder. *An Overview of Gradient Descent Optimization Algorithms*. Version 2. 2016. DOI: 10.48550/ARXIV.1609.04747. URL: https://arxiv.org/abs/1609.04747. Pre-published.

[119] NVIDIA. *CUDA C++ Programming Guide*. Oct. 1, 2024.

[120] Emanuele Vitali, Davide Gadioli, Gianluca Palermo, Andrea Beccari, Carlo Cavazzoni, and Cristina Silvano. "Exploiting OpenMP and OpenACC to Accelerate a Geometric Approach to Molecular Docking in Heterogeneous HPC Nodes". In: *The Journal of Supercomputing* 75.7 (July 2019), pp. 3374–3396. DOI: 10.1007/s11227-019-02875-w.

[121] Laxmikant V. Kalé, Abhinav Bhatele, Eric J. Bohm, James C. Phillips, David H. Bailey, Ananth Y. Grama, Joseph Fogarty, Hasan Aktulga, Sagar Pandit, David Padua, et al. "NAS Parallel Benchmarks". In: *Encyclopedia of Parallel Computing*. Ed. by David Padua. Boston, MA: Springer US, 2011, pp. 1254–1259. DOI: 10.1007/978-0-387-09766-4_133.

[122] Chunye Gong, Jie Liu, Jin Qin, Qingfeng Hu, and Zhenghu Gong. "Efficient Embarrassingly Parallel on Graphics Processor Unit". In: *2010 2nd International Conference on Education Technology and Computer*. 2010 2nd International Conference on Education Technology and Computer (ICETC). Shanghai, China: IEEE, June 2010, pp. V4-400-V4–404. DOI: 10.1109/ICETC.2010.5529656.

[123] Herman Van Vlijmen, Jean-Yves Ortholand, Volkhart M.-J. Li, and Jon S.B. De Vlieger. "The European Lead Factory: An Updated HTS Compound Library for Innovative Drug Discovery". In: *Drug Discovery Today* 26.10 (Oct. 2021), pp. 2406–2413. DOI: 10.1016/j.drudis.2021.04.019.

[124] *FooDB: The Largest and Most Comprehensive Resource on Food Constituents*.

[125] James Stewart. *MOPAC*. Stewart Computational Chemistry, 2016.

[126] Nan Ding and Samuel Williams. "An Instruction Roofline Model for GPUs". In: *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. 2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS). Denver, CO, USA: IEEE, Nov. 2019, pp. 7–18. DOI: 10.1109/PMBS49563.2019.00007.

[127] *NVIDIA Nsight Compute Kernel Profiling Guide*.

[128] Vasily Volkov. "Better Performance at Lower Occupancy". NVIDIA GPU Technology Conference (GTC). 2010.

[129] World Health Organization. *WHO Coronavirus Disease (COVID-19) Dashboard*. 2020. URL: https://covid19.who.int/.

[130] Marcello Allegretti, Maria Candida Cesta, Mara Zippoli, Andrea Beccari, Carmine Talarico, Flavio Mantelli, Enrico M. Bucci, Laura Scorzolini, and Emanuele Nicastri. "Repurposing the Estrogen Receptor Modulator Raloxifene to Treat SARS-CoV-2 Infection". In: *Cell Death & Differentiation* 29.1 (Jan. 2022), pp. 156–166. DOI: 10.1038/s41418-021-00844-6.

[131] Silvano Coletti and Gabriella Bernardi, eds. *Exscalate4CoV: High-Performance Computing for COVID Drug Discovery*. SpringerBriefs in Applied Sciences and Technology. Cham: Springer International Publishing, 2023. DOI: 10.1007/978-3-031-30691-4.

[132] Tilak Agerwala. "Exascale Computing: The Challenges and Opportunities in the next Decade". In: *ACM SIGPLAN Notices* 45.5 (May 2010), pp. 1–2. DOI: 10.1145/1837853.1693454.

[133] William Gropp. "MPI at Exascale: Challenges for Data Structures and Algorithms". In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Ed. by Matti Ropo, Jan Westerholm, and Jack Dongarra. Vol. 5759. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 3–3. DOI: 10.1007/978-3-642-03770-2_3.

[134] Pierre Darme, Manuel Dauchez, Arnaud Renard, Laurence Voutquenne-Nazabadioko, Dominique Aubert, Sandie Escotte-Binet, Jean-Hugues Renault, Isabelle Villena, Luiz-Angelo Steffenel, and Stéphanie Baud. "AMIDE v2: High-Throughput Screening Based on AutoDock-GPU and Improved Workflow Leading to Better Performance and Reliability". In: *International Journal of Molecular Sciences* 22.14 (July 13, 2021), p. 7489. DOI: 10.3390/ijms22147489.

[135] Andy B. Yoo, Morris A. Jette, and Mark Grondona. "SLURM: Simple Linux Utility for Resource Management". In: *Job Scheduling Strategies for Parallel Processing*. Ed. by Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn. Red. by Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen. Vol. 2862. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 44–60. DOI: 10.1007/10968987_3.

[136] Baldomero Imbernón, Antonio Serrano, Andrés Bueno-Crespo, José L Abellán, Horacio Pérez-Sánchez, and José M Cecilia. "METADOCK 2: A High-Throughput Parallel Metaheuristic Scheme for Molecular Docking". In: *Bioinformatics* 37.11 (July 12, 2021). Ed. by Alfonso Valencia, pp. 1515–1520. DOI: 10.1093/bioinformatics/btz958.

[137] Verónica G. Vergara Larrea, Wayne Joubert, Michael J. Brim, Reuben D. Budiardja, Don Maxwell, Matt Ezell, Christopher Zimmer, Swen Boehm, Wael Elwasif, Sarp Oral, et al. "Scaling the Summit: Deploying the World's Fastest Supercomputer". In: *High Performance Computing*. Ed. by Michèle Weiland, Guido Juckeland, Sadaf Alam, and Heike Jagode. Vol. 11887. Cham: Springer International Publishing, 2019, pp. 330–351. DOI: 10.1007/978-3-030-34356-9_26.

[138] Maciej Wójcikowski, Pedro J. Ballester, and Pawel Siedlecki. "Performance of Machine-Learning Scoring Functions in Structure-Based Virtual Screening". In: *Scientific Reports* 7.1 (Apr. 25, 2017), p. 46710. DOI: 10.1038/srep46710.

[139] *BlazingDB: High Performance GPU Database for Big Data SQL*. 2015.

[140] Anubhav Jain, Shyue Ping Ong, Wei Chen, Bharat Medasani, Xiaohui Qu, Michael Kocher, Miriam Brafman, Guido Petretto, Gian-Marco Rignanese, Geoffroy Hautier, et al. "FireWorks: A Dynamic Workflow System Designed for High-throughput Applications". In: *Concurrency and Computation: Practice and Experience* 27.17 (Dec. 10, 2015), pp. 5037–5059. DOI: 10.1002/cpe.3505.

[141] The MPI Forum. "MPI: A Message Passing Interface". In: *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing - Supercomputing '93*. The 1993 ACM/IEEE Conference. Portland, Oregon, United States: ACM Press, 1993, pp. 878–883. DOI: 10.1145/169627.169855.

[142] Tiejun Cheng, Qingliang Li, Zhigang Zhou, Yanli Wang, and Stephen H. Bryant. "Structure-Based Virtual Screening for Drug Discovery: A Problem-Centric Review". In: *The AAPS Journal* 14.1 (Mar. 2012), pp. 133–141. DOI: 10.1208/s12248-012-9322-0.

[143] Daniel F. Veber, Stephen R. Johnson, Hung-Yuan Cheng, Brian R. Smith, Keith W. Ward, and Kenneth D. Kopple. "Molecular Properties That Influence the Oral Bioavailability of Drug Candidates". In: *Journal of Medicinal Chemistry* 45.12 (June 1, 2002), pp. 2615–2623. DOI: 10.1021/jm020017n.

[144] Emanuele Vitali, Davide Gadioli, Gianluca Palermo, Andrea Beccari, and Cristina Silvano. "Accelerating a Geometric Approach to Molecular Docking with OpenACC". In: *Proceedings of the 6th International Workshop on Parallelism in Bioinformatics - PBio 2018*. The 6th International Workshop. Barcelona, Spain: ACM Press, 2018, pp. 45–51. DOI: 10.1145/3235830.3235835.

[145] Claudia Beato, Andrea R. Beccari, Carlo Cavazzoni, Simone Lorenzi, and Gabriele Costantino. "Use of Experimental Design To Optimize Docking Performance: The Case of LiGenDock, the Docking Module of Ligen, a New De Novo Design Program". In: *Journal of Chemical Information and Modeling* 53.6 (June 24, 2013), pp. 1503–1517. DOI: 10.1021/ci400079k.

[146] David Weininger. "SMILES, a Chemical Language and Information System. 1. Introduction to Methodology and Encoding Rules". In: *Journal of Chemical Information and Computer Sciences* 28.1 (Feb. 1, 1988), pp. 31–36. DOI: 10.1021/ci00057a005.

[147] Marc Snir, Robert W Wisniewski, Jacob A Abraham, Sarita V Adve, Saurabh Bagchi, Pavan Balaji, Jim Belak, Pradip Bose, Franck Cappello, Bill Carlson, et al. "Addressing Failures in Exascale Computing". In: *The International Journal of High Performance Computing Applications* 28.2 (May 2014), pp. 129–173. DOI: 10.1177/1094342014522573.

[148] Wesley Bland, Aurelien Bouteiller, Thomas Herault, George Bosilca, and Jack Dongarra. "Post-Failure Recovery of MPI Communication Capability: Design and Rationale". In: *The International Journal of High Performance Computing Applications* 27.3 (Aug. 2013), pp. 244–254. DOI: 10.1177/1094342013488238.

[149] Roberto Rocco, Davide Gadioli, and Gianluca Palermo. "Legio: Fault Resiliency for Embarrassingly Parallel MPI Applications". In: *The Journal of Supercomputing* 78.2 (Feb. 2022), pp. 2175–2195. DOI: 10.1007/s11227-021-03951-w.

[150] PBS Works. *OpenPBS: Industry-leading Workload Manager and Job Scheduler for High-Performance Computing*. 2016.

[151] Silvia Gervasoni, Giulio Vistoli, Carmine Talarico, Candida Manelfi, Andrea R. Beccari, Gabriel Studer, Gerardo Tauriello, Andrew Mark Waterhouse, Torsten Schwede, and Alessandro Pedretti. "A Comprehensive Mapping of the Druggable Cavities within the SARS-CoV-2 Therapeutically Relevant Proteins by Combining Pocket and Docking Searches as Implemented in Pockets 2.0". In: *International Journal of Molecular Sciences* 21.14 (July 21, 2020), p. 5152. DOI: 10.3390/ijms21145152.

[152] Thomas A. Halgren. "Merck Molecular Force Field. I. Basis, Form, Scope, Parameterization, and Performance of MMFF94". In: *Journal of Computational Chemistry* 17.5–6 (Apr. 1996), pp. 490–519. DOI: 10.1002/(SICI)1096-987X(199604)17:5/6<490::AID-JCC1>3.0.CO;2-P.

[153] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Andreas Müller, Joel Nothman, Gilles Louppe, et al. "Scikit-Learn: Machine Learning in Python". Version 4. In: (2012). DOI: 10.48550/ARXIV.1201.0490.

[154] H. M. Berman. "The Protein Data Bank". In: *Nucleic Acids Research* 28.1 (Jan. 1, 2000), pp. 235–242. DOI: 10.1093/nar/28.1.235.

[155] CINECA. *The Marconi100 Supercomputer*. 2017. URL: https://www.hpc.cineca.it/hardware/marconi100.

[156] ENI S.p.A. *HPC5, Supercomputers Serving Research Activities*. 2020. URL: https://www.eni.com/en-IT/actions/energy-transition-technologies/supercomputing-artificial-intelligence/supercomputer.html.

[157] Stefano Markidis, Davide Gadioli, Emanuele Vitali, and Gianluca Palermo. "Understanding the I/O Impact on the Performance of High-Throughput Molecular Docking". In: *2021 IEEE/ACM Sixth International Parallel Data Systems Workshop (PDSW)*. 2021 IEEE/ACM Sixth International Parallel Data Systems Workshop (PDSW). St. Louis, MO, USA: IEEE, Nov. 2021, pp. 9–14. DOI: 10.1109/PDSW54622.2021.00007.

[158] Tatsumi Aoyama, Ken-Ichi Ishikawa, Yasuyuki Kimura, Hideo Matsufuru, Atsushi Sato, Tomohiro Suzuki, and Sunao Torii. "First Application of Lattice QCD to Pezy-SC Processor". In: *Procedia Computer Science* 80 (2016), pp. 1418–1427. DOI: 10.1016/j.procs.2016.05.457.

[159] Sparsh Mittal. "A Survey on Evaluating and Optimizing Performance of Intel Xeon Phi". In: *Concurrency and Computation: Practice and Experience* 32.19 (Oct. 10, 2020), e5742. DOI: 10.1002/cpe.5742.

[160] Intel. *What Is Intel® Advanced Matrix Extensions (Intel® AMX)?* Intel Blog. URL: https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/what-is-intel-amx.html.

[161] RISC-V International. *RVA32 Profiles*. Oct. 17, 2024.

[162] RISC-V International. *RISC-V Announces Ratification of the RVA23 Profile Standard*. Oct. 21, 2024. URL: https://riscv.org/announcements/2024/10/risc-v-announces-ratification-of-the-rva23-profile-standard/.

[163] Joseph K. L. Lee, Maurice Jamieson, Nick Brown, and Ricardo Jesus. *Test-Driving RISC-V Vector Hardware for HPC*. Apr. 20, 2023. URL: http://arxiv.org/abs/2304.10319. Pre-published.

[164] 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. Vol. 13999. Cham: Springer Nature Switzerland, 2023, pp. 433–443. DOI: 10.1007/978-3-031-40843-4_32.

[165] Gianna Paulin, Matheus Cavalcante, Paul Scheffler, Luca Bertaccini, Yichao Zhang, Frank Gurkaynak, and Luca Benini. "Soft Tiles: Capturing Physical Implementation Flexibility for Tightly-Coupled Parallel Processing Clusters". In: *2022 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 2022 IEEE Computer Society Annual Symposium on VLSI (ISVLSI). Nicosia, Cyprus: IEEE, July 2022, pp. 44–49. DOI: 10.1109/ISVLSI54635.2022.00021.

[166] Gianna Paulin, Florian Zaruba, Stefan Mach, Manuel Eggimann, Matheus Cavalcante, Paul Scheffler, Yichao Zhang, Tim Fischer, Nils Wistoff, Luca Bertaccini, et al. *Occamy*. The IIS Chip Gallery. 2022. URL: http://asic.ethz.ch/2022/Occamy.html.

[167] Mario Kovač. "European Processor Initiative: The Industrial Cornerstone of EuroHPC for Exascale Era". In: *Proceedings of the 16th ACM International Conference on Computing Frontiers - CF '19*. The 16th ACM International Conference. Alghero, Italy: ACM Press, 2019, pp. 319–319. DOI: 10.1145/3310273.3323432.

[168] Alexander Dörflinger, Mark Albers, Benedikt Kleinbeck, Yejun Guan, Harald Michalik, Raphael Klink, Christopher Blochwitz, Anouar Nechi, and Mladen Berekovic. "A Comparative Survey of Open-Source Application-Class RISC-V Processor Implementations". In: *Proceedings of the 18th ACM International Conference on Computing Fron-*

tiers. CF '21: Computing Frontiers Conference. Virtual Event Italy: ACM, May 11, 2021, pp. 12–20. DOI: 10.1145/3457388.3458657.

[169] RISC-V International. "The RISC-V Instruction Set Manual Volume I: Unprivileged Architecture". In: (Apr. 11, 2024).

[170] E4 Computer Engineering. *The Armida HPC System at E4*. 2016.

[171] Andrea Bartolini, Francesco Beneventi, Andrea Borghesi, Daniele Cesarini, Antonio Libri, Luca Benini, and Carlo Cavazzoni. "Paving the Way Toward Energy-Aware and Automated Datacentre". In: *Workshop Proceedings of the 48th International Conference on Parallel Processing*. ICPP 2019: Workshops. Kyoto Japan: ACM, Aug. 5, 2019, pp. 1–8. DOI: 10.1145/3339186.3339215.

[172] Nikola Rajovic, Alejandro Rico, Filippo Mantovani, Daniel Ruiz, Josep Oriol Vilarrubi, Constantino Gomez, Luna Backes, Diego Nieto, Harald Servat, Xavier Martorell, et al. "The Mont-Blanc Prototype: An Alternative Approach for HPC Systems". In: *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*. SC16: International Conference for High Performance Computing, Networking, Storage and Analysis. Salt Lake City, UT, USA: IEEE, Nov. 2016, pp. 444–455. DOI: 10.1109/SC.2016.37.

[173] Mitsuhisa Sato, Yutaka Ishikawa, Hirofumi Tomita, Yuetsu Kodama, Tetsuya Odajima, Miwako Tsuji, Hisashi Yashiro, Masaki Aoki, Naoyuki Shida, Ikuo Miyoshi, et al. "Co-Design for A64FX Manycore Processor and "Fugaku"". In: *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. SC20: International Conference for High Performance Computing, Networking, Storage and Analysis. Atlanta, GA, USA: IEEE, Nov. 2020, pp. 1–15. DOI: 10.1109/SC41405.2020.00051.

[174] Ravi Sahita, Vedvyas Shanbhogue, Andrew Bresticker, Atul Khare, Atish Patra, Samuel Ortiz, Dylan Reid, and Rajnesh Kanwal. "CoVE: Towards Confidential Computing on RISC-V Platforms". In: *Proceedings of the 20th ACM International Conference on Computing Frontiers*. CF '23: 20th ACM International Conference on Computing Frontiers. Bologna Italy: ACM, May 9, 2023, pp. 315–321. DOI: 10.1145/3587135.3592168.

[175] Roger Espasa. "Introducing SemiDynamics High Bandwidth RISC-V IP Cores". RISC-V Global Forum. 2020.

[176] David R. Ditzel. "Accelerating ML Recommendation With Over 1,000 RISC-V/Tensor Processors on Esperanto's ET-SoC-1 Chip". In: *IEEE Micro* 42.3 (May 1, 2022), pp. 31–38. DOI: 10.1109/MM.2022.3140674.

[177] Florian Zaruba, Fabian Schuiki, and Luca Benini. "Manticore: A 4096-Core RISC-V Chiplet Architecture for Ultraefficient Floating-Point Computing". In: *IEEE Micro* 41.2 (Mar. 1, 2021), pp. 36–42. DOI: 10.1109/MM.2020.3045564.

[178] Colin Schmidt, John Wright, Zhongkai Wang, Eric Chang, Albert Ou, Woorham Bae, Sean Huang, Anita Flynn, Brian Richards, Krste Asanovic, et al. "4.3 An Eight-Core 1.44GHz RISC-V Vector Machine in 16nm FinFET". In: *2021 IEEE International Solid- State Circuits Conference (ISSCC)*. 2021 IEEE International Solid- State Circuits Conference (ISSCC). San Francisco, CA, USA: IEEE, Feb. 13, 2021, pp. 58–60. DOI: 10.1109/ISSCC42613.2021.9365789.

[179] RISC-V International. ""V" Standard Extension for Vector Operations, Version 1.0". In: *The RISC-V Instruction Set Manual Volume I: Unprivileged Architecture* (Apr. 11, 2024), pp. 273–371.

[180] Nick Brown. *RISC-V for HPC: Where We Are and Where We Need to Go*. Version 1. 2024. DOI: 10.48550/ARXIV.2406.12398. URL: https://arxiv.org/abs/2406.12398. Pre-published.

[181] Antonio Libri, Andrea Bartolini, and Luca Benini. "*pAElla* : Edge AI-Based Real-Time Malware Detection in Data Centers". In: *IEEE Internet of Things Journal* 7.10 (Oct. 2020), pp. 9589–9599. DOI: 10.1109/JIOT.2020.2986702.

[182] Alessio Netti, Woong Shin, Michael Ott, Torsten Wilde, and Natalie Bates. "A Conceptual Framework for HPC Operational Data Analytics". In: *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. 2021 IEEE International Conference on Cluster Computing (CLUSTER). Portland, OR, USA: IEEE, Sept. 2021, pp. 596–603. DOI: 10.1109/Cluster48925.2021.00086.

[183] Elizabeth Bautista, Melissa Romanus, Thomas Davis, Cary Whitney, and Theodore Kubaska. "Collecting, Monitoring, and Analyzing Facility and Systems Data at the National Energy Research Scientific Computing Center". In: *Workshop Proceedings of the 48th International Conference on Parallel Processing*. ICPP 2019: Workshops. Kyoto Japan: ACM, Aug. 5, 2019, pp. 1–9. DOI: 10.1145/3339186.3339213.

[184] SiFive. *SiFive U74-MC Core Complex Manual*. 2021.

[185] Todd Gamblin, Matthew LeGendre, Michael R. Collette, Gregory L. Lee, Adam Moody, Bronis R. de Supinski, and Scott Futral. "The Spack Package Manager: Bringing Order to HPC Software Chaos". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC15: The International Conference for High Performance Computing, Networking,

Storage and Analysis. Austin Texas: ACM, Nov. 15, 2015, pp. 1–12. DOI: 10.1145/2807591.2807623.

[186] John Furlani. "Modules: Providing a Flexible User Environment". In: 1991.

[187] Massimiliano Culpo, Gregory Becker, Carlos Eduardo Arango Gutierrez, Kenneth Hoste, and Todd Gamblin. "Archspec: A Library for Detecting, Labeling, and Reasoning about Microarchitectures". In: *2020 2nd International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*. 2020 2nd International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC). Atlanta, GA, USA: IEEE, Nov. 2020, pp. 45–52. DOI: 10.1109/CANOPIEHPC51917.2020.00011.

[188] Francesco Beneventi, Andrea Bartolini, Carlo Cavazzoni, and Luca Benini. "Continuous Learning of HPC Infrastructure Models Using Big Data Analytics and In-Memory Processing Tools". In: *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. 2017 Design, Automation & Test in Europe Conference & Exhibition (DATE). Lausanne, Switzerland: IEEE, Mar. 2017, pp. 1038–1043. DOI: 10.23919/DATE.2017.7927143.

[189] John McCalpin. "Memory Bandwidth and Machine Balance in High Performance Computers". In: *IEEE Technical Committee on Computer Architecture Newsletter* (Dec. 1995), pp. 19–25.

[190] RISC-V International. *RISC-V ABIs Specification*. Nov. 30, 2021.

[191] SiFive. *RISC-V Large Code Model Software Workaround*.

[192] GNU Compiler Collection. *RISC-V: Minimal Support of Bitmanip Instructions*.

[193] GNU Binutils. *RISC-V: Add Support for Zbs Instructions*.

[194] Florian Zaruba and Luca Benini. "The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-Nm FDSOI Technology". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27.11 (Nov. 2019), pp. 2629–2640. DOI: 10.1109/TVLSI.2019.2926114.

[195] Giuseppe Tagliavini, Stefan Mach, Davide Rossi, Andrea Marongiu, and Luca Benini. "Design and Evaluation of SmallFloat SIMD Extensions to the RISC-V ISA". In: *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2019 Design, Automation & Test in Europe Conference & Exhibition (DATE). Florence, Italy: IEEE, Mar. 2019, pp. 654–657. DOI: 10.23919/DATE.2019.8714897.

[196] *IEEE Standard for Floating-Point Arithmetic*. 2008. DOI: 10.1109/IEEESTD.2008.4610935.

[197] Dhiraj Kalamkar, Dheevatsa Mudigere, Naveen Mellempudi, Dipankar Das, Kunal Banerjee, Sasikanth Avancha, Dharma Teja Vooturi, Nataraj Jammalamadaka, Jianyu Huang, Hector Yuen, et al. *A Study of BFLOAT16 for Deep Learning Training*. Version 3. 2019. DOI: 10.48550/ARXIV.1905.12322. URL: https://arxiv.org/abs/1905.12322. Pre-published.

[198] NVIDIA. *NVIDIA cuDNN*. NVIDIA, 2023.

[199] Yann LeCun. "1.1 Deep Learning Hardware: Past, Present, and Future". In: *2019 IEEE International Solid- State Circuits Conference - (ISSCC)*. 2019 IEEE International Solid- State Circuits Conference - (ISSCC). San Francisco, CA, USA: IEEE, Feb. 2019, pp. 12–19. DOI: 10.1109/ISSCC.2019.8662396.

[200] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. "In-Datacenter Performance Analysis of a Tensor Processing Unit". Apr. 16, 2017.

[201] Naigang Wang, Jungwook Choi, Daniel Brand, Chia-Yu Chen, and Kailash Gopalakrishnan. "Training Deep Neural Networks with 8-Bit Floating Point Numbers". In: *Proceedings of the 32nd International Conference on Neural Information Processing Systems*. NIPS'18. Red Hook, NY, USA: Curran Associates Inc., 2018, pp. 7686–7695.

[202] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, et al. "PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation". In: *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. ASPLOS '24: 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2. La Jolla CA USA: ACM, Apr. 27, 2024, pp. 929–947. DOI: 10.1145/3620665.3640366.

[203] Paul Barham and Michael Isard. "Machine Learning Systems Are Stuck in a Rut". In: *Proceedings of the Workshop on Hot Topics in Operating Systems*. HotOS '19: Workshop on Hot Topics in Operating Systems. Bertinoro Italy: ACM, May 13, 2019, pp. 177–183. DOI: 10.1145/3317550.3321441.

[204] M. Griebl, C. Lengauer, and S. Wetzel. "Code Generation in the Polytope Model". In: *Proceedings. 1998 International Conference on Parallel Architectures and Compilation Techniques (Cat. No.98EX192)*. 1998 International Conference on Parallel Architectures and Compilation Techniques. Paris, France: IEEE Comput. Soc, 1998, pp. 106–111. DOI: 10.1109/PACT.1998.727179.

[205] Silicon Graphics, Inc. “WHIRL Intermediate Language Specification”. In: *The SGI Pro64™ Compiler*.

[206] Anthony Danalis, Lori Pollock, Martin Swany, and John Cavazos. “Implementing an Open64-based Tool for Improving the Performance of MPI Programs”. In: IEEE/ACM International Symposium on Code Generation and Optimization (CGO). Bostop, MA, Apr. 16, 2008.

[207] Gautam Chakrabarti and Fred Chow. “Structure Layout Optimizations in the Open64 Compiler: Design, Implementation and Measurements”. In: IEEE/ACM International Symposium on Code Generation and Optimization (CGO). Boston, MA, Apr. 16, 2008.

[208] Zhou Shuchang, Liu Ying, Lu Fang, Yin Le, Huang Lei, Li Shuai, Ma Chunhui, Gao Zhitao, and Lian Ruiqi. “Open64 on MIPS: Porting and Enhancing Open64 for Loongson II”. In: ().

[209] Hsin-I Cindy Liu, Marius Brehler, Mahesh Ravishankar, Nicolas Vasilache, Ben Vanik, and Stella Laurenzo. “TinyIREE: An ML Execution Environment for Embedded Systems From Compilation to Deployment”. In: *IEEE Micro* 42.5 (Sept. 1, 2022), pp. 9–16. DOI: 10.1109/MM.2022.3178068.

[210] *OpenXLA*.

[211] Modular Inc. *Mojo: A Language for next-Generation Compiler Technology*. Oct. 19, 2024. URL: https://docs.modular.com/mojo/why-mojo.

[212] Keith D. Cooper and Linda Torczon. *Engineering a Compiler*. Third edition. Cambridge, MA: Morgan Kaufmann Publishers, an imprint of Elsevier, 2023. 820 pp.

[213] Federico Ficarelli. “Taming Custom RISC-V Extensions with Multi-level Compilers”. In: HiPEAC 24. Munich, Germany, Jan. 17, 2024.

[214] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. “MLIR: Scaling Compiler Infrastructure for Domain Specific Computation”. In: *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). Seoul, Korea (South): IEEE, Feb. 27, 2021, pp. 2–14. DOI: 10.1109/CGO51591.2021.9370308.

[215] Sander de Smalen. “Optimizing Code for Scalable Vector Architectures”. 2021 LLVM Developers’ Meeting. 2021.

[216] Alexander Heinecke, Greg Henry, Maxwell Hutchinson, and Hans Pabst. "LIBXSMM: Accelerating Small Matrix Multiplications by Runtime Code Generation". In: *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*. SC16: International Conference for High-Performance Computing, Networking, Storage and Analysis. Salt Lake City, UT: IEEE, Nov. 2016, pp. 981–991. DOI: 10.1109/SC.2016.83.

[217] Field G. Van Zee, Tyler M. Smith, Bryan Marker, Tze Meng Low, Robert A. Van De Geijn, Francisco D. Igual, Mikhail Smelyanskiy, Xianyi Zhang, Michael Kistler, Vernon Austel, et al. "The BLIS Framework: Experiments in Portability". In: *ACM Transactions on Mathematical Software* 42.2 (June 3, 2016), pp. 1–19. DOI: 10.1145/2755561.

[218] Intel. *Intel oneDNN*. Intel, 2023.

[219] Angelo Garofalo, Manuele Rusci, Francesco Conti, Davide Rossi, and Luca Benini. "PULP-NN: A Computing Library for Quantized Neural Network Inference at the Edge on RISC-V Based Parallel Ultra Low Power Clusters". In: *2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*. 2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS). Genoa, Italy: IEEE, Nov. 2019, pp. 33–36. DOI: 10.1109/ICECS46596.2019.8965067.

[220] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. *cuDNN: Efficient Primitives for Deep Learning*. Dec. 18, 2014. URL: http://arxiv.org/abs/1410.0759. Pre-published.

[221] Florian Zaruba. *Harnessing the RISC-V Wave: The Future Is Now*. Axelera AI. 2023. URL: https://www.axelera.ai/harnessing-the-risc-v-wave-the-future-is-now/.

[222] T-Head Semiconductor Co. *T-Head ISA Extension Specification (Xthead*)*. 2023.

[223] Michael Gautschi, Pasquale Davide Schiavone, Andreas Traber, Igor Loi, Antonio Pullini, Davide Rossi, Eric Flamand, Frank K. Gurkaynak, and Luca Benini. "Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25.10 (Oct. 2017), pp. 2700–2713. DOI: 10.1109/TVLSI.2017.2654506.

[224] Antonio Pullini, Davide Rossi, Igor Loi, Giuseppe Tagliavini, and Luca Benini. "Mr.Wolf: An Energy-Precision Scalable Parallel Ultra Low Power SoC for IoT Edge Processing". In: *IEEE Journal of Solid-*

State Circuits 54.7 (July 2019), pp. 1970–1981. DOI: 10.1109/JSSC.2019.2912307.

[225] Rachit Nigam, Samuel Thomas, Zhijing Li, and Adrian Sampson. "A Compiler Infrastructure for Accelerator Generators". In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. Virtual USA: ACM, Apr. 19, 2021, pp. 804–817. DOI: 10.1145/3445814.3446712.

[226] Fabian Schuiki, Florian Zaruba, Torsten Hoefler, and Luca Benini. "Stream Semantic Registers: A Lightweight RISC-V ISA Extension Achieving Full Compute Utilization in Single-Issue Cores". In: *IEEE Transactions on Computers* 70.2 (Feb. 1, 2021), pp. 212–227. DOI: 10.1109/TC.2020.2987314.

[227] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Sixth edition. Morgan Kaufmann, 2019.

[228] LLVM authors PULP Project. *Snitch Target Support in LLVM*. GitHub, 2023.

[229] Nicolas Vasilache, Oleksandr Zinenko, Aart J. C. Bik, Mahesh Ravishankar, Thomas Raoux, Alexander Belyaev, Matthias Springer, Tobias Gysi, Diego Caballero, Stephan Herhut, et al. *Composable and Modular Code Generation in MLIR: A Structured and Retargetable Approach to Tensor Compiler Construction*. Feb. 7, 2022. URL: http://arxiv.org/abs/2202.03293. Pre-published.

[230] Evangelos Georganas, Dhiraj Kalamkar, Sasikanth Avancha, Menachem Adelman, Cristina Anderson, Alexander Breuer, Jeremy Bruestle, Narendra Chaudhary, Abhisek Kundu, Denise Kutnick, et al. "Tensor Processing Primitives: A Programming Abstraction for Efficiency and Portability in Deep Learning Workloads". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '21: The International Conference for High Performance Computing, Networking, Storage and Analysis. St. Louis Missouri: ACM, Nov. 14, 2021, pp. 1–14. DOI: 10.1145/3458817.3476206.

[231] Lorenzo Chelini, Henrik Barthels, Paolo Bientinesi, Marcin Copik, Tobias Grosser, and Daniele G. Spampinato. *MOM: Matrix Operations in MLIR*. Version 1. 2022. DOI: 10.48550/ARXIV.2208.10391. URL: https://arxiv.org/abs/2208.10391. Pre-published.

[232] Uday Bondhugula. "High Performance Code Generation in MLIR: An Early Case Study with GEMM". Mar. 1, 2020.

[233] Tian Jin, Gheorghe-Teodor Bercea, Tung D. Le, Tong Chen, Gong Su, Haruki Imai, Yasushi Negishi, Anh Leu, Kevin O'Brien, Kiyokuni Kawachiya, and Alexandre E. Eichenberger. *Compiling ONNX Neural Network Models Using MLIR*. Sept. 30, 2020. URL: http://arxiv.org/abs/2008.08272. Pre-published.

[234] Aart J. C. Bik, Penporn Koanantakool, Tatiana Shpeisman, Nicolas Vasilache, Bixia Zheng, and Fredrik Kjolstad. "Compiler Support for Sparse Tensor Computations in MLIR". Feb. 9, 2022.

[235] Tobias Gysi, Christoph Müller, Oleksandr Zinenko, Stephan Herhut, Eddie Davis, Tobias Wicky, Oliver Fuhrer, Torsten Hoefler, and Tobias Grosser. "Domain-Specific Multi-Level IR Rewriting for GPU: The Open Earth Compiler for GPU-accelerated Climate Simulation". In: *ACM Transactions on Architecture and Code Optimization* 18.4 (Dec. 31, 2021), pp. 1–23. DOI: 10.1145/3469030.

[236] Navdeep Katel, Vivek Khandelwal, and Uday Bondhugula. "MLIR-based Code Generation for GPU Tensor Cores". In: *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*. CC '22: 31st ACM SIGPLAN International Conference on Compiler Construction. Seoul South Korea: ACM, Mar. 19, 2022, pp. 117–128. DOI: 10.1145/3497776.3517770.

[237] Mathieu Fehr, Michel Weber, Christian Ulmann, Alexandre Lopoukhine, Martin Lücke, Théo Degioanni, Michel Steuwer, and Tobias Grosser. *Sidekick Compilation with xDSL*. Version 3. 2023. DOI: 10.48550/ARXIV.2311.07422. URL: https://arxiv.org/abs/2311.07422. Pre-published.

[238] Mathieu Fehr, Jeff Niu, River Riddle, Mehdi Amini, Zhendong Su, and Tobias Grosser. "IRDL: An IR Definition Language for SSA Compilers". In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. San Diego CA USA: ACM, June 9, 2022, pp. 199–212. DOI: 10.1145/3519939.3523700.

[239] George Bisbas, Anton Lydike, Emilien Bauer, Nick Brown, Mathieu Fehr, Lawrence Mitchell, Gabriel Rodriguez-Canal, Maurice Jamieson, Paul H. J. Kelly, Michel Steuwer, and Tobias Grosser. "A Shared Compilation Stack for Distributed-Memory Parallelism in Stencil DSLs". In: *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. ASPLOS '24: 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3. La Jolla CA USA: ACM, Apr. 27, 2024, pp. 38–56. DOI: 10.1145/3620666.3651344.

[240] GNU Project. *GCC, the GNU Compiler Collection*. 2023.

[241] *Cranelift Code Generator*. Bytecode Alliance, 2023.

[242] *xDSL: A Python-native SSA Compiler Framework*. xDSL Project, 2023.

[243] Florent Bouchez Tichadou and Fabrice Rastello, eds. *SSA-based Compiler Design*. 1st ed. 2022. Cham: Springer Nature Switzerland AG, 2023. 382 pp.

[244] *The Torch-MLIR Project*. 2024.

[245] Jeff Niu and Mehdi Amini. "MLIR Dialect Design and Composition for Front-End Compilers". 2023 European LLVM Developers' Meeting (Glasgow, UK). May 11, 2023.

[246] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures: A Dependance-Based Approach*. Transferred to digital print. San Francisco, Calif.: Morgan Kaufmann, 2011. 790 pp.

[247] MLIR Authors. *MLIR Documentation: 'linalg' Dialect*. Multi-Level IR Compiler Framework. 2023. URL: https://mlir.llvm.org/docs/Dialects/Linalg.

[248] Sebastian Braun and Ivan Tashev. "Data Augmentation and Loss Normalization for Deep Noise Suppression". In: *Speech and Computer*. Ed. by Alexey Karpov and Rodmonga Potapova. Vol. 12335. Cham: Springer International Publishing, 2020, pp. 79–86. DOI: 10.1007/978-3-030-60276-5_8.

[249] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Communications of the ACM* 60.6 (May 24, 2017), pp. 84–90. DOI: 10.1145/3065386.

[250] Veripool. *Verilator*.

[251] Felix Mölder, Kim Philipp Jablonski, Brice Letcher, Michael B. Hall, Christopher H. Tomkins-Tinch, Vanessa Sochat, Jan Forster, Soohyun Lee, Sven O. Twardziok, Alexander Kanitz, et al. "Sustainable Data Analysis with Snakemake". In: *F1000Research* 10 (Jan. 18, 2021), p. 33. DOI: 10.12688/f1000research.29032.1.

[252] Adrián Castelló, Julian Bellavita, Grace Dinh, Yuka Ikarashi, and Héctor Martínez. "Tackling the Matrix Multiplication Micro-Kernel Generation with Exo". In: *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). Edinburgh, United Kingdom: IEEE, Mar. 2, 2024, pp. 182–193. DOI: 10.1109/CGO57630.2024.10444883.

[253] Yuka Ikarashi, Gilbert Louis Bernstein, Alex Reinking, Hasan Genc, and Jonathan Ragan-Kelley. "Exocompilation for Productive Programming of Hardware Accelerators". In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. San Diego CA USA: ACM, June 9, 2022, pp. 703–718. DOI: 10.1145/3519939.3523446.

[254] Braedy Kuzma, Ivan Korostelev, João P. L. De Carvalho, José E. Moreira, Christopher Barton, Guido Araujo, and José Nelson Amaral. "Fast Matrix Multiplication via Compiler-only Layered Data Reorganization and Intrinsic Lowering". In: *Software: Practice and Experience* 53.9 (Sept. 2023), pp. 1793–1814. DOI: 10.1002/spe.3214.

[255] Steven Varoumas. "Using MLIR to Optimize Basic Linear Algebraic Subprograms". 2023 Euro LLVM Developer's Meetingh (Glasgow, UK). May 10, 2023.

[256] Bastian Hagedorn, Archibald Samuel Elliott, Henrik Barthels, Rastislav Bodik, and Vinod Grover. "Fireiron: A Data-Movement-Aware Scheduling Language for GPUs". In: *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*. PACT '20: International Conference on Parallel Architectures and Compilation Techniques. Virtual Event GA USA: ACM, Sept. 30, 2020, pp. 71–82. DOI: 10.1145/3410463.3414632.

[257] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. "The Tensor Algebra Compiler". In: *Proceedings of the ACM on Programming Languages* 1 (OOPSLA Oct. 12, 2017), pp. 1–29. DOI: 10.1145/3133901.

[258] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. "GraphIt: A High-Performance Graph DSL". In: *Proceedings of the ACM on Programming Languages* 2 (OOPSLA Oct. 24, 2018), pp. 1–30. DOI: 10.1145/3276491.

[259] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. "Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code". In: *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). Washington, DC, USA: IEEE, Feb. 2019, pp. 193–205. DOI: 10.1109/CGO.2019.8661197.

[260] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. "LIFT: A Functional Data-Parallel IR for High-Performance GPU Code Generation". In: *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). Austin, TX, USA: IEEE, Feb. 2017, pp. 74–85. DOI: 10.1109/CGO.2017.7863730.

[261] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. "Learning to Optimize Tensor Programs". In: *Proceedings of the 32nd International Conference on Neural Information Processing Systems*. NIPS'18. Red Hook, NY, USA: Curran Associates Inc., 2018, pp. 3393–3404.

[262] Rui Li, Yufan Xu, Aravind Sukumaran-Rajam, Atanas Rountev, and P. Sadayappan. "Analytical Characterization and Design Space Exploration for Optimization of CNNs". In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. Virtual USA: ACM, Apr. 19, 2021, pp. 928–942. DOI: 10.1145/3445814.3446759.

[263] Nicolas Tollenaere, Guillaume Iooss, Stéphane Pouget, Hugo Brunie, Christophe Guillon, Albert Cohen, P. Sadayappan, and Fabrice Rastello. "Autotuning Convolutions Is Easier Than You Think". In: *ACM Transactions on Architecture and Code Optimization* 20.2 (June 30, 2023), pp. 1–24. DOI: 10.1145/3570641.

[264] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. "Spatial: A Language and Compiler for Application Accelerators". In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '18: ACM SIGPLAN Conference on Programming Language Design and Implementation. Philadelphia PA USA: ACM, June 11, 2018, pp. 296–311. DOI: 10.1145/3192366.3192379.

[265] Perry Gibson and José Cano. "Transfer-Tuning: Reusing Auto-Schedules for Efficient Tensor Program Code Generation". In: *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. PACT '22: International Conference on Parallel Architectures and Compilation Techniques. Chicago Illinois: ACM, Oct. 8, 2022, pp. 28–39. DOI: 10.1145/3559009.3569682.