Université de Strasbourg

École Doctorale de Mathématiques, Sciences de l'Information et de l'Ingénieur

Laboratoire des Sciences de l'Ingénieur, de l'Informatique et de l'Imagerie

THÈSE présentée par :

Arun Thangamani

Soutenue le : **25 Septembre 2024**

Pour obtenir le grade de : **Docteur de l'Université de Strasbourg**

Discipline / Spécialité : **Informatique**

Génération de code optimisée pour des nids de boucles parallèles et polyédriques à l'aide de MLIR

Optimized Code Generation of Parallel and Polyhedral Loop Nests using MLIR

THÈSE dirigée par :

M. Stéphane Genaud	Professeur à l'Université de Strasbourg, France

RAPPORTEURS :

M. Guillaume Latu	Directeur de Recherche, CEA Cadarache, France
M^{me} Alexandra Jimborean	Ramon y Cajal Researcher, University of Murcia, Spain

EXAMINATEURS :

M. Sylvain Contassot-Vivier	Professeur à l'Université de Lorraine, France
M. Mark Potse	Research Professor, Université de Bordeaux, France

INVITÉ :

M. Vincent Loechner	Maître de Conférences à l'Université de Strasbourg, France (co-encadrant)

Dedicating this thesis to my Father Mr. Thangamani Subramanium.

Acknowledgements

First and foremost, I would like to thank my advisors Stéphane Genaud and Vincent Loechner for their continuous support, guidance, and patience. Their expertise and insights were invaluable in shaping the direction and quality of this work. It is good for me as it is hard to convince Stéphane with an idea. I ended up making my ideas stronger or new ideas while trying to convince him. I did admire the writing skills of Stéphane and learned a lot about writing from him. Without Vincent, I might not ended up doing a PhD. It is Vincent who offered me a PhD position at his lab on looking at my profile even though I have not applied for that position. He is the first one who ignited the fire for the PhD. His knowledge of polyhedral techniques is amazing, I learned a lot from him and applied them to this work.

I would like to thank my co-author Tiago Trevisian Jost for his discussion and support. Tiago and I had a lot of discussions at the initial stage of my PhD, we made certain decisions and those are believed to be the success factors of this work. Being new to France, Tiago and Vincent helped me to adapt to the French culture.

I would like to thank my thesis rapporteurs Guillaume Latu and Alexandra Jimborean for their proofreading and feedback on the manuscript. I also thank Sylvain Contassot-Vivier and Mark Potse for agreeing to be part of my thesis jury.

I would like to thank my thesis committee members: Laure Gonnord and Basile Sauvage. I thank Cédric Bastoul and Louis-Noël Pouchet for their technical discussion.

I am thankful to the MICROCARD project, Euro-HPC, and the European Union for funding my thesis and conference travels. I thank my other co-workers Bérenger Bramas, Raphaël Colin, Atoli Huppé, all members of the ICPS lab and the Inria CAMUS team, and the members of the MICROCARD project.

I thank my Father and Mother who are my first teachers to teach the basics of life. It's sad and unfortunately, my father is no more to read this acknowledgment but it's ok that is life. My Mother taught me the benefits of patience. Kavin Thangamani sponsored my first personal computer where all about the computers started for me.

Finally, I would like to thank the love of my life Gowthami Arun for being part of everything. She exposed me to the beautiful aspects of life other than research. And sorry I couldn't financially help you a lot during PhD but I'm sure you are very proud about being part of my PhD. And the arrival of our son Khaathwik Arun added furthermore beautiful colors to our life. He was the first teacher to teach me parenting is more difficult than doing a PhD. His presence and innocence made us leave our negative thoughts and overcome the difficult phases in our life. Our blessings to him. We will remember our happy days in a $14m^2$ studio in Illkirch, our frequent trips to Schengen zoos, trips to Paris, and trips to Cigoland. We will miss Strasbourg and France a lot with a hope to come back again.

This work was supported by the European High-Performance Computing Joint Undertaking EuroHPC under grant agreement No 955495 (MICROCARD) co-funded by the Horizon 2020 programme of the European Union (EU), the French National Research Agency ANR, the German Federal Ministry of Education and Research, the Italian ministry of economic development, the Swiss State Secretariat for Education, Research and Innovation, the Austrian Research Promotion Agency FFG, and the Research Council of Norway.

Abstract

In this thesis we show the benefits of the novel MLIR compiler technology to the generation of code from a DSL, namely EasyML used in openCARP, a widely used simulator in the cardiac electrophysiology community. Building on an existing work we deeply modified openCARP's native code generator to enable efficient vectorized CPU and GPU code generation (Nvidia CUDA and AMD ROCm). Generating optimized code for different accelerators requires specific optimizations and we review how MLIR has been used to enable multi-target code generation from an integrated generator. To our knowledge, this is the first work that deeply connects an optimizing compiler infrastructure to electrophysiology models of the human body, showing the potential benefits of using compiler technology in the simulation of human cell interactions. Additionally, we did a study on the polyhedral compilers and generalized our techniques using Polygeist to improve the vectorization and heterogeneous code generation of polyhedral compilers.

Résumé

Dans cette thèse, nous montrons les avantages de la nouvelle technologie de compilateur MLIR pour la génération de code à partir d'un DSL, à savoir EasyML utilisé dans openCARP, un simulateur largement utilisé dans la recherche en électrophysiologie cardiaque. S'appuyant sur un travail existant nous avons profondément modifié le générateur de code natif d'openCARP pour permettre une génération efficace de code CPU vectoriel et GPU (Nvidia CUDA et AMD ROCm). La génération de code optimisé pour différents accélérateurs nécessite des optimisations spécifiques et nous examinons comment MLIR a été utilisé pour permettre la génération de code multi-cible à partir d'un générateur intégré. À notre connaissance, il s'agit du premier travail qui relie profondément une infrastructure de compilateur d'optimisation aux modèles électrophysiologiques du corps humain, montrant les avantages potentiels de l'utilisation de technologies de compilation dans la simulation des interactions entre cellules humaines. De plus, nous avons réalisé une étude sur les compilateurs polyédriques et généralisé nos techniques en utilisant Polygeist pour améliorer la vectorisation et la génération de codes hétérogènes des compilateurs polyédriques.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

The impact of computers on human real-life applications is very wide and it revolutionizes the day-to-day activities of human life. Here are a few significant areas where computers have made an enormous impact: Healthcare - starting from electronic medical records, medical image processing, diagnostic tools to telemedicine; E-commerce - doorstep delivery of household items; Communication - starting from emails, video conferencing, social media platforms to virtual reality technology; Transport Systems - navigation tools, traffic management systems, and autonomous vehicles; Science and Research - climate modeling, genomics, drug discovery, and simulations. Computer technology has itself undergone several evolutions over the past 50 years and still evolving to make itself compatible with different domains. Particularly, the personal computing era in the 1980s saw the rapid growth of personal computers in homes and offices thereby igniting the fire of computing capabilities. Later advancements in the microprocessor and storage further add fuel to the computing power. The mobile and internet era in the early 2000s resulted in handheld smartphone computer devices for every individual. And, everybody knows the present cloud computing and artificial intelligence (AI) impact and its benefits. The French version of this chapter is available in Appendix B.

In this thesis, we want to discuss the advancements in computational systems targeting large scale applications and how the code generation and code optimization should adapt to these advancements to maximize the efficient usage of supercomputers.

1.1 Hardware Advances in Computation

Both hardware and software advance together to improve the computational power of computers. We now discuss a few advancements made in computer hardware:

Multi-Core Processors. Integrating multiple processing cores onto a single chip thereby allowing parallel execution of threads/tasks.

Graphics Processing Units. Specially designed electronic circuits initially intended for

graphics rendering in video games, but it has evolved a lot and now being used in many other application domains particularly to say machine learning, scientific simulations and 3D imaging.

Field-Programmable Gate Arrays. An FPGA is an integrated circuit that provides the capability of dynamic reconfiguration to perform a wide variety of tasks, particularly in digital signal processing.

Vector Instruction Sets. A specially designed instruction set to enhance the performance of similar operations on multiple data, commonly known as vector or SIMD (Single Instruction, Multiple Data) operations. Intel's SSE/AVX/AVX512, Arm's Neon and Nvidia's warps are a few examples.

Distributed Systems. Interconnection of several computers together that communicate and coordinate together to complete a common goal. These systems share computational tasks, storage, and processing nodes to provide high computational power and scalability.

Quantum Computing. Though in the early stages of development, quantum computing arises as a promising methodology for improving computational power far beyond the classical computing system. It uses the principles of quantum mechanics to perform computation on quantum bits instead of the classical bits used in traditional computers.

1.2 Supercomputers

Supercomputers are very high-performance computing systems specially designed to tackle complex computations that require massive parallelism and to handle large-scale data processing. Most supercomputers are heterogeneous systems as they are capable of supporting different architectures like x86, AArch64, RISC-V, and GPU. The target is scientific and engineering applications that are required to solve intensive problems that are far beyond the computational power of conventional computers. They typically consist of millions of cores running in parallel to achieve a high number of floating-point operations per second (Flop/s). They do have high-end parallel or distributed architectures with multiple processing units like CPUs, GPUs, and specialized accelerators (e.g., FPGAs). Supercomputers are highly scalable to handle large data, interconnected through high-speed networks, and have specialized architectures (like Tensor-processing Units) to support machine learning/AI applications. As they consume large amounts of electric power and generate a significant amount of heat during computation, they are equipped with efficient cooling systems.

The Top500[1] website ranks the world's available supercomputers in terms of their computing power. Since June 2022, the Frontier supercomputer hosted at Oak Ridge National Laboratory is the world's fastest supercomputer with 8.7 million cores and

[1] https://www.top500.org/

with a computing power of 1206 PFlop/s[2], that is more than an exaFlop/s. Now, it's our responsibility to effectively use these hardware resources to reap the full benefits of these supercomputers.

1.3 Compiler Proposed Code Optimizations

In this section, we discuss optimization techniques that could be used to automatically transform the software to leverage the capabilities of hardware, which can be better exploit parallelism or avoid bottlenecks.

Parallelism. Thread and task-level parallelization involves splitting the computation into smaller parts and executing these parts simultaneously using multiple processing units (cores, CPUs or GPUs). For example, OpenMP (Open Multi-Processing) is a parallel programming API widely used in C, C++, or Fortran that supports task-level and thread-level parallelism in shared memory systems. MPI (Message Passing Interface) is another widely used standard for parallel programming in distributed systems.

Vectorization (SIMD). It is an optimization technique to exploit the possible SIMD instructions in the code thereby enabling faster and more efficient computation with the help of a vector instruction set. Many standard compilers (like `gcc`, `clang`, and `icc`) do auto-vectorization with standard compiler analysis techniques to generate equivalent SIMD vector instructions. However, in many cases, the compilers cannot auto-vectorize code where manual vectorization could be applied. This involves invoking a few compiler analyses (like dependency analysis), rewriting code to include vector directives, or writing vector instruction at the intermediate representation (IR)/assembly level.

Cache Coherence. A system with multiple processing units (CPUs or cores) may share a common memory hierarchy. It needs to be ensured that the memory is consistent across them by maintaining coherence between the data stored in their respective caches. There are many cache coherence protocols (like MESI) being used to maintain the cache coherence.

Software Pipelining. Compiler can take advantage of instruction-level parallelism offered by processors to re-order instructions (called *out-of-order execution*. In this way the hardware pipeline of instructions can be fed more efficiently and realize parallelism at the instruction level.

Shared Resources. If multiple threads execute concurrently it is very important to manage the shared resources properly to avoid data races and deadlocks. The critical aspect is not only computational improvement but also to ensure the correctness of the program. Different synchronization mechanisms like mutexes, semaphores, critical sections, and atomic operations help us to effectively manage these shared resources.

Data Locality. Data management plays a vital role in distributed systems. However, achieving optimal data locality in a distributed system is quite challenging due to many

[2] https://top500.org/project/linpack/

parameters, like network latency, data partition, and system scalability. There are situations where the performance gains in computation have been ruined by inefficient data placement resulting in excessive data copies. Here are some key considerations the software developer has to investigate for data locality in distributed systems: data replication, data partitioning, data movement, and network topology.

The support for the multi-processing and distributed systems has resulted in the evolution of many parallel and distributed languages (like X10 [1, 2, 3], Chapel [4], Cilk [5], Erlang, Go [6]) to ease the programming effort needed to use these supercomputers. In these languages, the compiler has the ability to generate an optimized assembly code targeting those multi-core heterogeneous architectures.

A compiler is typically a computer program capable of translating a human-readable source code into an optimized assembly file which is later on converted into machine instructions and executed on the computer hardware. Thus, the compiler plays a very crucial role in software development as it not only transforms an input code into an executable but it also does apply a lot of optimizations to generate an efficient code. However, in many cases, the compiler is limited to applying safe optimization techniques to pieces of code it achieves to analyze, and, because of the complex nature of the program the developers need to manually intervene to optimize it further.

1.4 Context and Contributions

In this thesis, we introduce a new domain-specific heterogeneous compilation and code generation technique targeting exascale supercomputers, to optimize parallel loop structures in cardiac simulation applications. We provide hints to the compiler to generate vector code for complex loop statements (which are not auto-vectorized by the compiler) and execute them on the CPU using different vector instruction sets. We further enhance the code generation process to emit GPU code for those loop statements and execute them using the accelerators like Nvidia and AMD GPUs.

Compute Intensive Code

A scientific application spends generally most of the time in iterative computation phases as they contain the most statements to get executed. The statements are often compute-intensive kernels that require significant computational resources and heavily relies on CPU or GPU processing power. These compute-intensive kernels mostly involve:

- Simulations like molecular/fluid dynamics, weather patterns, and biological derivations,
- Data analysis: processing of very large data sets, statistical analysis or training an ML model that requires extensive computation,

- Mathematical computations like solving differential or integral equations,
- Image or signal processing.

Mostly, these kernels are enclosed inside loops with large iteration counts. Identifying and optimizing these kernels are essential for improving the performance and efficiency of these large applications. Developers often profile their applications to identify such compute-intensive kernels and optimize them, using standard optimization techniques or by proposing new hardware-specific optimizations. There are cases where the compiler itself could identify these patterns and apply optimizations (for example auto-vectorization).

Can those kernel optimizations be further improved?

Several optimization factors needed to be considered while generating an optimized code targeting supercomputers. The main focus would be invoking massive parallelism resulting in maximized usage of all the cores. Sometimes, the complex nature of the application results in difficulties applying these optimizations, for example the dependencies between instructions prevents some loop transformations. Moreover, the presence of complex statements like a function/library call or irregular control flow within the loop can prevent the compiler's auto-vectorization, even though the loop carries no dependencies and is annotated with vector directives.

Another aspect is code generation: many applications rely on their domain-specific language (DSL) as it is more convenient for their experts to write code with less programming knowledge. Then, it is the responsibility of the DSL compiler to generate efficient code. Most of the designed code generators target one architecture, either CPU or GPU, not both.

Compiler technology has evolved a lot and there are many new compiler frameworks introduced to help the developers generate optimized code for their applications, the MLIR [7] compiler is one such example. MLIR (Multi-Level Intermediate Representation) from LLVM [8] is the state-of-the-art compiler technology that aims to represent various levels of abstraction and optimization in a unified form, thanks to several coexisting intermediate representations. It is a very recent framework designed to address the challenges of modern compiler infrastructure, able to target heterogeneous computing systems where multiple programming languages, hardware targets, and optimization passes need to be coordinated efficiently.

Heterogeneous Code Generation for a Cardiac Simulation Application

Cardiac simulation refers to the computational modeling of the human heart's structure and its function. It involves creating mathematical ordinary differential equations (ODEs) representing the flow of `Iion` within the channels, which electrical activity drives the heartbeat (electrophysiology), and the physical behavior of heart tissues (mechanics). These simulations are mainly used as medical aid in diagnosing heart

conditions by providing insights into abnormal heart rhythms (arrhythmias), ischemia (reduced blood flow), and other cardiac disorders. The state-of-the-art cardiac simulators can simulate only a portion of the heart or simulate it roughly. Due to the advancements in the computer's computing power and the evolution of more powerful supercomputers, the research in cardiac technology will soon reach the precise simulation of the entire human heart. The heart is composed of approximately 2 billion muscular cells and simulating the entire heart involves processing those 2 billion cells. Even with the most powerful supercomputers, the simulation on 2 billion cells will be challenging considering the complex mathematical equations and also depend on how effectively the simulation code is optimized on a heterogeneous architecture. In Chapter 2, we address the limitations in the state-of-the-art cardiac simulator's code generation and optimization process. And in Chapter 3, we provide information about *dialects* and optimization passes in MLIR.

In Chapter 4, we introduce an optimized code compilation/generation technique with the help of MLIR to emit vectorized CPU and GPU code for cardiac simulation applications to target different architectures in supercomputers. We implemented our techniques on the openCARP [9] open-source cardiac simulator and used 48 different computational (ionic) models available in openCARP to test our implementation. Those 48 computational models are different combinations of ODE equations and are widely used in cardiac research. We evaluated their performance using two large-scale test beds: (i) Grid'5000 (`https://www.grid5000.fr`) and (ii) PlaFRIM (`https://plafrim.fr`). Both test beds contain the various architectures to execute the generated heterogeneous code. In Chapter 4 Section 4.2, the performance results of 48 models are reported for CPU vectorized code (SSE, AVX, AVX-512) and GPU code (Nvidia and AMD). Our execution results show that helping the compiler to generate an optimized code results in effective usage of multi-cores and heterogeneous architecture available in those large-scale machines. We also captured the energy consumption and explained how efficient our techniques were in reducing the power consumption. We published our techniques in two conferences [10] and [11].

This work presented in Chapter 4, is a collaborative work with a research engineer (initially started with Tiago Trevisan Jost and succeeded by Raphaël Colin). Also, this work led to a collaboration with the STORM research team at Inria Bordeaux where our compilation flow is integrated with StarPU [12], a task-based run-time system, to distribute the execution on the target architecture at run-time [13].

Improvements to Polyhedral Loop Optimization Techniques

With the promising improvements in performance of the cardiac simulation application, we generalized our techniques in polyhedral loop optimizations. Polyhedral optimization techniques are used to transform programs with regular loop nests like matrix operations, stencil computation, and image processing applications. They first start with representing the loop nest as polyhedra in a geometrical space and then perform a dependency analysis between statements. They mostly help to perform three loop optimizations: tiling, loop parallelism, and vectorization. Tiling is done by re-structuring

the loops and aim to improve data locality. The loops are annotated with parallel and vector directives for loop parallelism and vectorization, respectively.

There are many different types of polyhedral compilers available like general-purpose source-to-source compilers, built-in compilers, application-specific, and target-specific compilers. In Chapter 5, we first made a detailed survey on the available general-purpose polyhedral compilers using our large-scale test beds to find out their advantages and limitations. We used the widely recognized PolyBench/C [14] - a set of thirty numerical benchmarks targeting various domain applications to conduct a detailed study on the polyhedral compilers. The analysis using hardware performance counters provided insights and scope for further improvements in polyhedral compilers particularly in enhancing vector optimization and GPU code generation.

In Chapter 6, we address a couple of limitations in the state-of-the-art polyhedral compilers: (i) the vector annotations using directives are merely a recommendation to the compilers and are ignored in a few cases, and (ii) there is no unified approach to generate a heterogeneous code. We relied on Polygeist [15] to generalize our earlier introduced techniques as a solution to the two above-mentioned problems. We choose Polygeist [15], the reason being an MLIR/LLVM based framework to lower `C/C++` code to Polyhedral MLIR code and generalizing our techniques using Polygeist will require reduced effort as the required environment is already established. We modified the code generation flow of Polygeist to emit OpenMP SIMD MLIR loops for vector annotated loops. Our experimental results do not show any performance improvements on the PolyBench/C programs, because of limited OpenMP SIMD support by MLIR. The MLIR framework is undergoing very active development phases to support various compiler optimizations. So, in the very near future MLIR could provide complete optimization support for OpenMP SIMD loops and we hope that our proposed technique improves vectorization. We left the implementation of GPU code generation as future work. The study on polyhedral compilers is published in ACM TACO [16] and the Polygeist SIMD loop generation technique is accepted for publication at a PhD symposium [17].

This thesis outlines the advances in the computing power of supercomputers, limitations in code optimization techniques targeting those supercomputers, and a solution to overcome those limitations particularly enhancing vectorization and GPU code generation. Our goal is to show how optimizing code in very large applications can effectively use the massive resources in supercomputers for performance improvements of applications that have an impact on human real life. The remainder of this thesis exposes how we proceeded.

The thesis was funded by the European High-Performance Computing Joint Undertaking EuroHPC under grant agreement No 955495 (**MICROCARD**), co-funded by the Horizon 2020 programme of the European Union (EU), and France, Italy, Germany, Austria, Norway, and Switzerland (`https://microcard.eu`).

Chapter 2

Cardiac Simulation

In this chapter we discuss about the basics of cardiac electrophysiology simulation, the ionic models that drive the cardiac simulation, the available state-of-the-art cardiac simulators, the code generation process from ionic models, and its limitations.

2.1 Cardiac Electrophysiology

One common way of monitoring the cardiac electrical activity is electrocardiography which produces an electrocardiogram (ECG). Figure 2.1[1] shows three different responses of the human heart by ECG in which arrhythmia and ischemia are the abnormal behaviors. Arrhythmia is a state where the heartbeat is irregular. It can be a fast or slow or inconsistent heartbeat but occurs mainly because the heart's electrical system does not respond properly to the input electrical signals. Ischemia refers to heart weakening due to a reduced flow of blood into the heart thereby damaging the heart tissues. Coronary artery diseases are one of the main causes of ischemia.

The heart's electrical activity is defined by the current that flows into the cardiac tissue wrapping the heart. Figure 2.2 shows a microscopic view of real cardiac tissue. The left side of the image is a healthy tissue where the muscle fibers are in purple color and are separated by white sheets. Whereas the right side of the image is a damaged tissue and we can see heavy penetration of fatty tissue between the muscle fibers. These fatty tissues prevent electrical interconnectivity and weaken the electrical functionality of the human heart. These types of damaged tissues can lead to heart beat abnormalities such as arrhythmia.

Electrically active cells generate electrical signals named action potentials that are found in nerves and muscles mainly to communicate and synchronize cellular functions within them. The difference in ion concentration between the inside and outside of these cells generates brief changes in electrical potential across the cell membranes: ions cross the membrane through ion channels. The ion concentrations are slightly impacted if

[1]Image taken from https://istockphoto.com

there are a reasonable number of action potentials. There are transient openings in ion channels which are controlled by molecular mechanisms based on the transmembrane voltage. Thus, the transient openings and the conductivity in certain ions inside the channel can be represented using non-linear first-order differential equations.

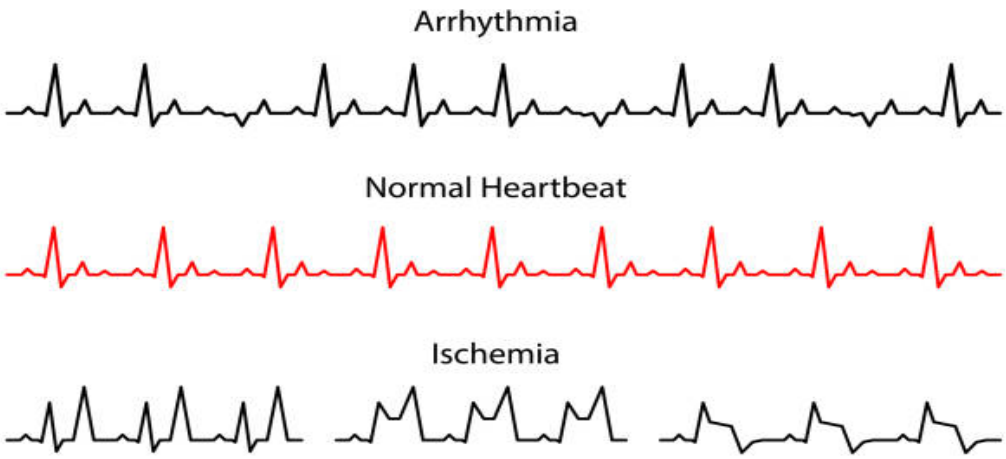


Figure 2.1: Different heart responses measured using ECG

The objective of cardiac simulation is to model and reproduce the observed phenomena, and then extrapolate behaviors in different scenarios. In reality, cardiac simulation resorts to model the current flows traversing the cardiac tissue, which amounts to compute the current intensity of ions and voltage that make the ions traverse membranes to flow from one cardiac cell to another. Figure 2.3[2] shows the different scales of the cardiac systems namely ion channel, membrane, tissue, organ, and organism.

About 2 billion muscle cells together constitute the human heart thus forming a network of electrically interconnected fibers.

2.2 Cardiac Simulation

In many scientific fields, numerical simulation is based on mathematical modeling of the physical behavior of the matter of interest. For instance, biologists want to model cells or protein interactions, physicists are interested in the behavior of particles not only governed by the Newtonian laws, but also in quantum physics, and mechanical engineers are concerned with mechanical structures such as bridges. One possibility for the mathematical modeling of such systems may use some abstract description language, typically a domain-specific language (DSL), which provides the necessary instrument to describe the computations.

[2]Image taken from https://microcard.eu

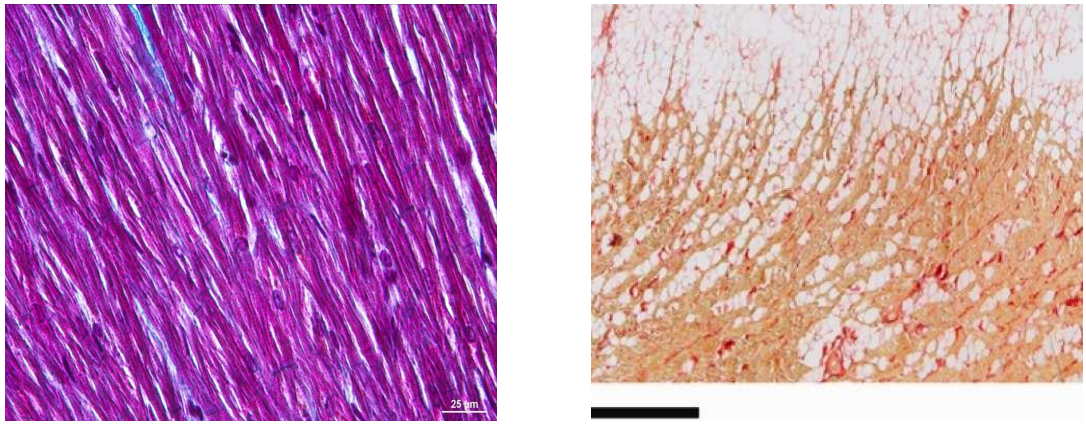


Figure 2.2: Left picture is the microscopic view of cardiac tissues. The muscle fibers are in purple color that are separated by collagen sheets (white). Damage to cardiac tissues causes collagen growth and results in connectivity loss. Image is from *Dr. D. Benoist*, IHU Liryc. Right picture shows a cardiac tissue heavily penetrated by fatty tissues (white). This type of tissue damage can lead to arrhythmia. Image is from *Dr. M. Hoogendijk*, AMC, Amsterdam.

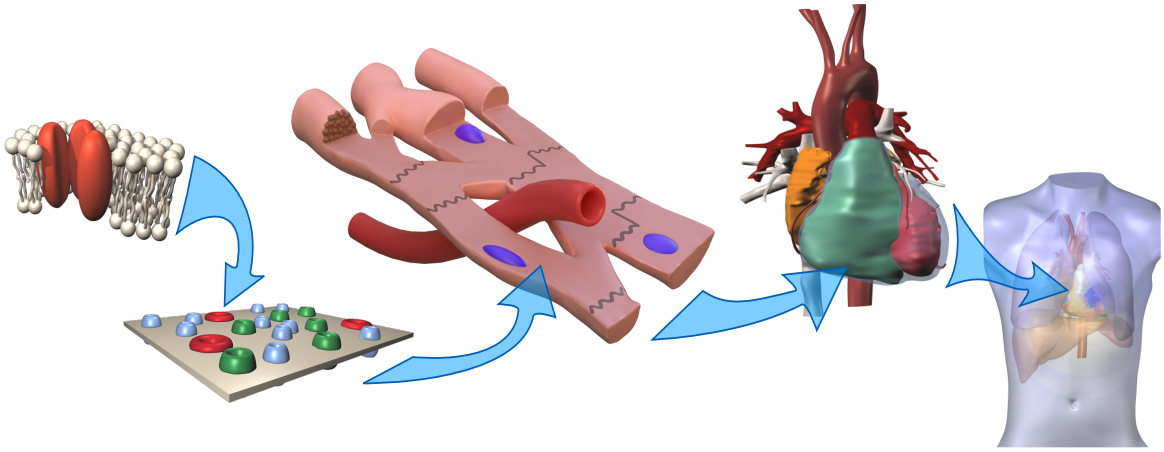


Figure 2.3: Different scales of cardiac system

2.3 Ionic Models and their Description

The modeling of the ionic flows in the cardiac tissue amounts to describe the evolution of the two physical quantities: the current I_{ion} and the voltage V_m over the cell membrane for a time period. This description is called an *ionic model*.

The transmembrane voltage is defined as $V_m = \phi_i - \phi_e$ where (i) ϕ_i is the intracellular potential, and (ii) ϕ_e is the extracellular potential. The sum of individual currents from the ion channel constitutes the *Ionic* current (I_{ion}). Hodgkin and Huxley [18] proposed a general form for action potential and described the above process as:

$$\begin{aligned} C_m \partial_t V_m + I_{ion}(V_m, \mathbf{y}) &= 0 \\ \partial_t \mathbf{y} &= \mathbf{F}(V_m, \mathbf{y}) \end{aligned} \tag{2.1}$$

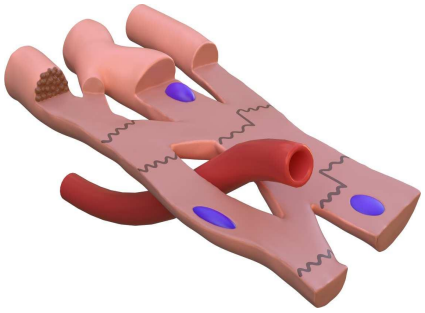


Figure 2.4: A sample image of cardiac tissue from microcard.eu project. A cardiac tissue is divided into cells and connected electrically with gap junctions (jagged lines).

where **F** is a non-linear vector-valued function, together I_{ion} and **F** constitutes a membrane (*Ionic*) model; **y** is the state vector which includes the gate information for each current and the ion concentrations. These systems of equations **F** can be very large and computationally expensive to evaluate, and it has to be computed for every element in the simulation and for thousands of time steps. Figure 2.4 shows cardiac muscle cells that are electrically inter-linked through gap junctions.

Central to the models are the ODEs, used to compute the evolution of I_{ion}. We describe hereafter the most commonly used methods to compute those ODEs. Over time, research in cardiac simulation has proposed more and more refined models, which imply the use of more and more ODEs in the models for sake of more accurate modeling, while leading to an increasing computational complexity.

2.3.1 Integration Methods (ODEs)

An important aspect of an ordinary differential equation lies in the selection of the appropriate method for the temporal discretization of an approximate solution. Here we list some of the commonly used methods in an ionic model [18]. In the below equations, h - is a step value; function $f()$ - is to compute the slope; t - is time;

- *Forward euler (fe)* [19] is a fast and explicit first-order method for solving ODEs. It is usually the default method used by simulators when none is specified by the user.

  One step of the Euler method is:

  $$y_{n+1} = y_n + hf(t_n, y_n)$$

- *Runge-Kutta with 2 steps (rk2)* [20] is an explicit second-order method. It provides better accuracy than *fe*, with twice as many computations (two calls to the `f` function).

  $$y_{n+1} = y_n + hf(t_n + \frac{1}{2}h, y_n + \frac{1}{2}f(t_n, y_n))$$

- *Runge-Kutta with 4 steps (rk4)* [20] is an explicit fourth-order method that pro-

vides more accuracy than *rk2* with more than twice computations,

$$y_{n+1} = y_n + \frac{1}{6}hf(k_1 + 2k_2 + 2k_3 + k_4)$$

where

$$k_1 = f(t_n, y_n), k_2 = f(t_n + \frac{h}{2}, y_n + h\frac{k_1}{2})$$

$$k_3 = f(t_n + \frac{h}{2}, y_n + h\frac{k_2}{2}), k_4 = f(t_n + h, y_n + hk_3)$$

- *Rush-Larsen* [21] is one of the most popular first-order methods for discretizing ODEs in dynamic models of cardiac electrophysiology [22]. Easy to implement and it is the preferred method for simulating *gates*, which represent the movement of proteins forming the ion channel in response to the membrane potential. It uses first-order.
- *Sundnes method* [23] is an extension of the Rush-Larsen in a second-order scheme, which is proven to be more efficient than its predecessor over stiff problems. This method uses second-order of RL.
- *Markov_be* is a backward method inspired by Euler. It uses an implicit first-order Runge-Kutta method, where models require values to be in between 0 and 1. A refinement process is used to keep values as precise as possible, so this method is used for models where accuracy is paramount.

2.3.2 DSL for Ionic Models

The widespread practice in this field is for bio-medical experts to describe their ionic model in a domain-specific language (DSL). The use of a DSL brings the obvious advantage to let experts concentrate on the modelling itself, relieving them from the programming details they would have to take care of if they used a general-purpose programming language. There are several DSL introduced by the cardiac research community. We can cite CellML [24], EasyML [25], MMT, and SBML [26]. Note that translators are available to translate from a DSL representation to another, like for example from EasyML to CellML (and vice-versa). Let us take EasyML DSL as a reference to outline its pre-eminent characteristics. EasyML:

1. uses an SSA (static single assignment) [27] representation, so that all variables are defined as mathematical equalities in an arbitrary order;
2. uses specific variables prefixes/suffixes (such as `_init`, `diff_`, etc.) to indicate the operational semantics of the variable using ODE;
3. offer calls to *math* library functions;
4. uses *markup* statements to specify various variable properties, such as: which method to use for integrating differential equations (`.method(m)`), whether to

```
1  V; .external(Vm); .nodal(); // external variables
2  Iion; .external(); .nodal(); // external variables
3
4  // initial values for ODE variables
5  V_init = -60.0;
6  m_init = 0.05;
7  h_init = 0.6;
8  n_init = 0.325;
9
10 E_R = -60.0;
11
12 GNa = 120.0; .param(); // parameters
13 E_Na = (E_R+ 115.0 );
14 INa = GNa*m*m*m*h*(V-E_Na);
15 alpha_m = ( 0.1 *(V+ 25.0 ))/(exp(( 0.1 *(V+ 25.0 )))- 1.0 );
16 beta_m = ( 4.0 *exp(V/ 18.0 ));
17 dm_dt = ((alpha_m*( 1.0 -m))-(beta_m*m));
18 alpha_h = ( 0.07 *exp(V/ 20.0 ));
19 beta_h =  1.0 /(exp(( 0.1 *(V+ 30.0 )))+ 1.0 );
20 dh_dt = ((alpha_h*( 1.0 -h))-(beta_h*h));
21
22 GK = 36.0; .param();  // parameters
23 E_K = (E_R- 12.0 );
24 alpha_n = ( 0.01 *(V+ 10.0 ))/(exp(( 0.1 *(V+ 10.0 )))- 1.0 );
25 beta_n = ( 0.125 *exp(V/ 80.0 ));
26 dn_dt = ((alpha_n*( 1.0 -n))-(beta_n*n));
27 IK = (GK*n*n*n*n*(V-E_K));
28
29 g_L = 0.3; .param();  // parameters
30 E_L = (E_R+ 10.613 );
31 IL = (g_L*(V-E_L));
32 dV_dt  =  -Iion;
33
34 Iion = (INa+IK+IL); // Updates to ion concentration
```

Listing 2.1: An ionic model proposed by Hodgkin and Huxley written in EasyML

pre-compute a lookup table of predefined values over a given interval (`.lookup`), which variables to output (`.trace`), etc.;

5. it is not Turing-complete since it does not let the programmer express loops, control flow, or sequence of elements – but there can be tests expressed as restricted `if/else` statements or as C-like ternary operators. All the control flow is fixed by the simulation framework.

Let us illustrate EasyML further with an example. Listing 2.1 shows an ionic model proposed by Hodgkin and Huxley [18]. If we look closer to the action items of this model, we can see:

1. Lines 1 and 2 defines external variables V_m and I_{ion}. At the beginning of the simulation, the V_m value is set to represent the initial voltage, that becomes afterward the difference between the intra and extracellular potential difference (ion changes). Similarly, I_{ion} gets updated with respect to the changes of V_m.

2. Lines 5 to 8 set the initial values for the four variables that will be used by ODE computation.

3. Lines 12, 22, and 29 define controlable parameters (they can be modified at runtime). For example, the parameter GNa is local for solving equations from lines 13 to 20. Similarly, GK is local to lines 23-27 and g_L is local to lines 30-31.

4. Lines 17, 20, 26, and 32 describe the ODE computation (d*_dt) using Rush Larsen method to finally compute `INa`, `IK`, and `IL`. We can see that the external variable V (V_m) is used at many places for evaluating intermediate variables.

5. Line 34 updates I_{ion} with the sum of its components `INa` + `IK` + `IL`.

2.4 Code Generation from Ionic Models

After having described a model using a DSL, the next stage consists in the generation of code that computes the model. This task is realized by a module called *code generator*. We can distinguish two phases for this task: the code generator first acts as a front-end to parse the ionic model description written in the DSL to produce the operational semantics of the model's specifics under the form of an Abstract Syntax Tree (AST). It then traverses the AST nodes that are translated into code in a given programming language. This generated code is inserted into a skeleton template code that implements the simulation mechanics common to all models. Examples of such generators can be found in Myokit [28] and openCARP [9]. Myokit accepts ionic models written using the MMT DSL and has a Python-based tool that can export models into multiple formats and programming models, such as C, OpenCL, CUDA, and Matlab. openCARP (the software we will be using in this work) takes as input ionic models written in EasyML and generates C and C++ source code. The whole source code is then compiled to produce the complete cardiac simulator.

Figure 2.5 shows the overview of a cardiac simulator. It is composed of the *Ionic Computation* and *Electric Potential (V_m) Computation (solver)* phases. From an initial voltage value, representing a small impulse of external voltage, the first ionic computation phase computes I_{ion}. Subsequently, this value is used in the electrical potential computation which solves for V_m, the electric potential of each cell membrane. Then, with the newly computed V_m, the gating/intermediate variables and ion are recomputed. This process of computing V_m and I_{ion} (current) is conducted in a cyclic fashion for a given period of time for each cell.

Notice that the electrical potential computation phase which solves a system of equations can rely on any general-purpose solver. There are various linear-solver software/frameworks available. In particular PETSc [29] and Ginkgo [30] are used in the openCARP cardiac simulation software. In our work, this phase is simply an invocation of a solver as an external library call. Therefore, it is seen as a black box over which we have no control and will be out of the scope of the cardiac simulator improvement. Our work hence focuses on the ionic computation part, which we detail in the upcoming section.

2.4.1 Ionic Computation Phase in OpenCARP

Let us now instantiate on the openCARP software, the principal elements that build up an ionic model, from its description to the executable simulation. The model

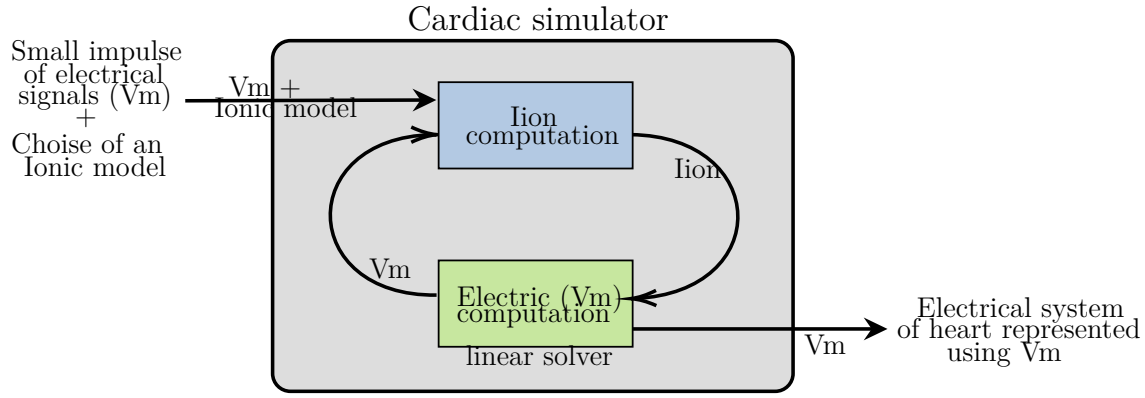


Figure 2.5: Overview of an cardiac simulator with two phases: (i) Ionic compute phase and (ii) Electric potential (V_m) computation phase.

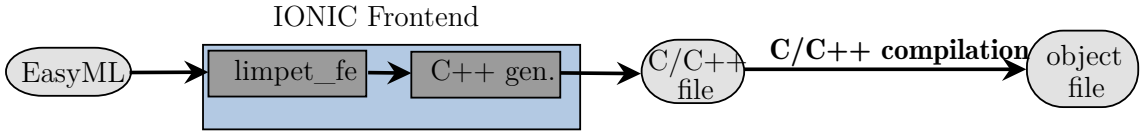


Figure 2.6: openCARP simulator code generation flow

description is written in EasyML. It is then parsed by the code generator, implemented as a python program called `limpet_fe`, which produces an AST using the `ast` python module. From the AST, the code generator emits `C/C++` code with (i) functions to initialize parameters, lookup tables, and state variables, and (ii) a `compute` function that scans all cells in a *for* loop, to calculate the output `Iion` current and update the state variables of each cell. Finally, the generated code is compiled using a standard `C/C++` compiler and the object file is used for simulation. Figure 2.6 sketches these steps. We call this original compilation flow of openCARP the *baseline*.

Listing 2.2 shows the `C` code emitted by the ionic computation phase of openCARP for the Hodgkin and Huxley ionic model in listing 2.1. The analysis of the generated code is as follows:

1. The constants are defined and declared in lines from 2 to 9.

2. Initialization of V_m, I_{ion}, and local parameters are done in function `initialize()`.

3. The I_{ion} computation happens in function `compute_ionic_ODE()`. Lines 23 to 37 do the computation for solving Rush Larsen ODE. Lines 40 to 42 updates the cell's state variables. In line 45, $p{\rightarrow}GK$ is the usage of the parameter. The external variables: V is loaded from V_ext (line 20) which holds the value computed by the *solver*; I_{ion} - line 49 stores the ionic flow calculated with the Rush Larsen method to an external variable $Iion\_ext$. Later, the *solver* uses this value to re-compute the electric potential. During the simulation, openCARP has an outer loop that iterates over a fixed time stamp to call the function `compute_ionic_ODE()` followed by a call to the linear-system solver.

2.4.2 Compute-intensive Kernels

A critical part of the generated code is the one related to the computation of the ODEs that contain many complex mathematical operations on real numbers, thus making it a compute intensive kernel. The code generator simply translates the expressions in EasyML to the C code shown in Listing 2.2, lines 20 to 49. For example, we can see that `exp` and `expm1` are heavily used. In our investigation of the 48 different ionic models available in openCARP, we found that the following math operations were frequently used: `pow`, `log`, `tanh`, `cube`, `fabs`, `exp`, `sqrt`, and `expm1`. These kinds of kernels consume a major part of the execution time apart the solver and optimizing them is expected to largely contribute to the performance improvement.

Moreover, the above code is the ODE calculation for one cell membrane. As told previously, the heart is composed of 2 billion (approx.) such cells and if the target is to model individual cells, using model elements that are much smaller than a cell, then we need to do the ODE calculation for 2 billion times. Even the very basic methodology of simulating a part of the organ would require processing at least 10 million cells. So, the minimum number of times the ODEs are calculated is 10 millions times the number of timesteps for the whole simulation. This confirms 99.9% of ionic computation time would be spent in this function.

2.5 Critical Parts of Cardiac Simulators

The cardiac simulator generates code from ionic models with a standard structure. It contains a loop over time steps and for each time step a *compute* function is called. This compute function iterates over all the cells and does ion calculation. As each cell computation is independent, the iterations over cells can be parallel. The loop is made parallel by the code generator by inserting the OpenMP directives (for example `#pragma omp parallel for`, shown in line 19 of Listing 2.2). Here we propose two advancements to the code generators of cardiac simulators: (i) vectorization, and (ii) GPU code generation.

Vectorization. Both openCARP and Myokit generate parallel loops but they depend on the standard compiler for vectorization of the parallel loop. It is not always the case that compilers auto-vectorize a parallel loop. They have to get positive results on vector-cost model analysis and the loop should not contain any complex statements that would result in ambiguity to apply vectorization. We did an initial investigation on 48 different ionic models available in openCARP and found that the compiler cannot vectorize the parallel loop in many cases as the loop contains complex statements because of ODEs. Here is the list of a few warnings shown by the compilers (`gcc` and `clang`) while trying for auto-vectorization: (i) pointer/array deference, (ii) irregular control flow, (iii) function/library calls, (iv) switch statements, and (v) couldn't determine the loop iterations. Even annotating the loop with vector directives doesn't solve the problem. Listing 2.3 shows the warning by the clang compiler when trying to auto-vectorize the code emitted by openCARP for the Hodgkin and Huxley model. Line 3

```c
// Define all constants
#define E_R (double)(-60.0)
#define V_init (double)(-60.0)
#define h_init (double)(0.6)
#define m_init (double)(0.05)
#define n_init (double)(0.325)
#define E_K (double)((E_R-(12.0)))
#define E_L (double)((E_R+10.613))
#define E_Na (double)((E_R+115.0))

void initialize(...)
{
    // This function is to initialize Vm, Iion, and local parameters.
}

void compute_ionic_ODE(...)
{
 #pragma omp parallel for schedule(static)
 for (int cell_id=start; cell_id<end; cell_id++) {
  double V = V_ext[cell_id];
  ...
  //C code for Rush Larsen Update
  double alp_h = (0.07*(exp((V/20.0))));
  double alp_m = ((0.1*(V+25.0))/((exp((0.1*(V+25.0))))-(1.0)));
  double alp_n = ((0.01*(V+10.0))/((exp((0.1*(V+10.0))))-(1.0)));
  double beta_h = (1.0/((exp((0.1*(V+30.0))))+1.0));
  double beta_m = (4.0*(exp((V/18.0))));
  double beta_n = (0.125*(exp((V/80.0))));
  double h_rush_larsen_A = (((-alp_h)/(alp_h+beta_h))*(expm1(((-dt)*(alp_h+beta_h))
    )));
  double h_rush_larsen_B = (exp(((-dt)*(alp_h+beta_h))));
  double m_rush_larsen_A = (((-alp_m)/(alp_m+beta_m))*(expm1(((-dt)*(alp_m+beta_m))
    )));
  double m_rush_larsen_B = (exp(((-dt)*(alp_m+beta_m))));
  double n_rush_larsen_A = (((-alp_n)/(alp_n+beta_n))*(expm1(((-dt)*(alp_n+beta_n))
    )));
  double n_rush_larsen_B = (exp(((-dt)*(alp_n+beta_n))));
  double h_new = h_rush_larsen_A+h_rush_larsen_B*sv->h;
  double m_new = m_rush_larsen_A+m_rush_larsen_B*sv->m;
  double n_new = n_rush_larsen_A+n_rush_larsen_B*sv->n;

  // Update to state variables.
  sv->h = h_new;
  sv->m = m_new;
  sv->n = n_new;

  // Update Iion
  double IK = (((((p->GK*sv->n)*sv->n)*sv->n)*sv->n)*(V-(E_K)));
  double IL = (p->g_L*(V-(E_L)));
  double INa = (((((p->GNa*sv->m)*sv->m)*sv->m)*sv->h)*(V-(E_Na)));
  Iion = ((INa+IK)+IL);
  Iion_ext[cell_id] = Iion;
 }
}

void output_trace(...)
{
    // Emit output or debugging trace
}
```

Listing 2.2: An equivalent `C` code emitted by a cardiac simulator for Hodgkin and Huxley ionic model shown in Listing 2.1.

```
1 [ 20%] Building CXX object physics/limpet/CMakeFiles/limpet.dir/src/imps_src/HH.cc.
      o
2
3 /openCAR/physics/limpet/CMakeFiles/limpet.dir/src/imps_src/HH.cc.o:393:42: remark:
      loop not vectorized:call instruction cannot be vectorized[-Rpass-analysis=loop-
      vectorize]
4 Ca_i_row[KsCa_idx] = (1.0+(0.6/(1.0+(pow((3.8e-5/Ca_i),1.4)))));
5
6 /openCAR/physics/limpet/CMakeFiles/limpet.dir/src/imps_src/HH.cc.o:386:3: remark:
      loop not vectorized:could not determine number of loop iterations[-Rpass-
      analysis=loop-vectorize]
7 for (int __i=Ca_i_tab->mn_ind; __i<=Ca_i_tab->mx_ind; __i++) {
8
9 /openCAR/physics/limpet/CMakeFiles/limpet.dir/src/imps_src/HH.cc.o:785:5: remark:
      loop not vectorized:loop contains a switch statement[-Rpass-analysis=loop-
      vectorize]
10 LUT_interpRow(&IF->tables[Ca_i_TAB], sv->Ca_i, __i, Ca_i_row);
11
12 /openCAR/physics/limpet/CMakeFiles/limpet.dir/src/imps_src/HH.cc.o:778:22: remark:
      loop not vectorized:value that could not be identified as reduction is used
      outside the loop[-Rpass-analysis=loop-vectorize]
13 GlobalData_t v = v_ext[__i];
```

Listing 2.3: Warning by clang compiler while trying to auto-vectorize the code generated by openCARP for Hodgkin and Huxley model.

shows it cannot vectorize the statement of line 4 as it contains a math library call (kind of a function call) to perform power operation. It cannot vectorize the statement of line 7 as it cannot determine the number of loop iterations because the loop-induction variable is pointer deference. Though the clang can inline function in line 10, it cannot vectorize it because of the switch statement (irregular control flow) within the function. As it is not able to identify the reduction variable it doesn't vectorize the statement in line 13.

GPU code generation. Only Myokit can generate code targeting different computer architectures. They emit `C`, openCL, and CUDA kernels. Also, they have translators that would help to convert an ionic model in one DSL representation to another DSL representation. Though Myokit is the only simulator to emit CUDA code targeting GPU architecture its purpose is different. In recent advancements, there are many GPU architectures but Myokit targets only CUDA. OpenCARP only has a CPU code generator, not the GPU or other architectures.

With the rapid advancements in computing technology and access to exascale supercomputers, the cardiac community urges the importance of advancing from part of organ simulation to whole organ simulation. OpenCARP is one such initiative for the simulation of the entire human heart. But, openCARP neither has a very optimized code generation process nor a heterogeneous code targeting different architectures.

Chapter 4 addresses all these limitations and introduces an optimized heterogeneous code generation and compilation flow targeting the different hardware architectures in exascale supercomputers. We guided standard compilers with vectorization hints, packed 2, 4, or 8 cells together to exploit the SIMD instructions, and relied on the vector instruction sets (SEE, AVX2, and AVX-512 respectively) available in those supercomputers. Unlike the other code generators, we followed a unified approach to emit one abstraction of code for CPU/GPU, and with the help of MLIR (further described in Chapter 3) the code is lowered to target either CPU vectorized machine,

Nvidia GPU machine, or AMD GPU machine, respectively.

There are many techniques available to generate optimized and heterogeneous code (like compiler intrinsics), we chose the recent MLIR [7] compiler technology because of its flexibility to represent a domain application at higher levels of abstraction. With *lowering* and *translation* passes available in MLIR, the high-level abstraction can be lowered to any specific architecture. Compiler intrinsics are low-level functions or operations provided by a compiler to directly access specific hardware features or low-level operations. They are specific to a particular compiler and target architecture, and also each time different intrinsics have to be generated concerning the target. In contrast, MLIR requires generating one high-level representation with which the code can be lowered to either CPU, Vector CPU, openMP CPU, Nvidia GPU, AMD GPU, or OpenCL code. In the near future, if MLIR supports any other architecture, with minimal effort and with the same abstraction the new architecture can be targeted. It is very convenient and easy for the developers to write/generate code at high-level IR with less knowledge of target architecture, rather than directly emitting low-level IR using intrinsics. With built-in support of *python* bindings, MLIR further eases the code generation process for users.

Because of many advantages over other techniques, MLIR emerges to be the best choice for heterogeneous code generation and is proven to correct on looking at the performance improvements by these techniques on cardiac simulation applications (shown in Chapter 4 Section 4.2 Experiments and Results). Chapter 3 introduces MLIR with its flexibility in representing a problem irrespective of any domain, optimization passes, and novelty.

Chapter 3

Multi-Level Intermediate Representation

MLIR (Multi-Level Intermediate Representation) [7] is a compiler infrastructure developed by the LLVM [8] framework. MLIR offers a means to express code operations and types through an extensible set of different common *intermediate representations* (IR), called *dialects*, each dedicated to a specific concern, at different levels of abstraction. Those dialects can be used by various other compilers and tools within the compiler stack. MLIR is designed to be flexible and extensible, allowing developers to define custom dialects and optimizations tailored to their specific needs. The different abstraction levels allow MLIR to accommodate a wide range of languages, hardware targets, and optimization techniques. MLIR has gained attention in the compiler community for its potential to simplify the development of compilers and optimization passes, as well as for its ability to facilitate the integration of different compiler components. It is used in various projects, including TensorFlow, Swift for TensorFlow, and LLVM. Developers can build on top of MLIR to create compilers tailored to their needs without having to develop a compiler from scratch, thanks to the LLVM compiler back-end. For example, MLIR can be used to create compilers for domain-specific languages (DSLs), where custom dialects can be defined to represent the semantics of the DSL efficiently.

MLIR has also driven the idea of using it as a building block for new IRs in order to permit interleaving levels of abstractions and further optimization opportunities. Gysi et al. [31] propose a hierarchy of dialects (IRs) for GPU-based stencil computations that is effective in weather and climate applications. A multi-level rewriting flow is proposed to progressively lower abstractions level-by-level, applying optimizations at the most appropriate abstraction. The approach has shown significant speedup in comparison to state-of-the-art solutions for climate and weather simulation, proving that extra levels of abstractions can help to devise new optimizations. Sommer et al. [32] propose a dialect, and a lowering process to optimize sum-product network inference in both CPUs and GPUs. DistIR [33] is an IR for distributed computation that employs MLIR to optimize neural networks. Recently, many works have proposed to extend MLIR with new dialects to analyze, optimize and accelerate heterogeneous

applications in a variety of domains [34, 35, 36]. We make use of MLIR as an enabling tool for our code generator and compiler transformations, similar to some previous works [37, 38, 39]: Bondhugula [38], and Katel et al. [37] present evaluations of the modularity of MLIR with sequences of transformations and customizable passes to optimize matrix multiplications. Vasilache et al. [39] builds composable abstractions for leveraging tensor algebra computation. Polygeist [15] acts as a `C/C++` frontend to MLIR and generates `affine` dialect code to better utilize the polyhedral optimization and code generation available in MLIR.

3.1 MLIR Dialects and Passes

In this section, we discuss *dialects* and optimization passes that distinguish MLIR from other compiler frameworks. They are as follows:

3.1.1 Dialects

MLIR supports various types of dialects, each serving a purpose or representing different aspects of programming languages, hardware architectures, or optimization passes. Let us classify the MLIR dialects into four categories:

1. *High-level Dialects*: These dialects provide a high-level representation for expressing the semantics of programming languages or domain-specific constructs in a concise and structured manner. These dialects facilitate the representation and optimization of programs written in high-level languages within the MLIR framework. They include constructs specific to these languages, such as data types, control flow statements, and language-specific operations. A few high-level dialects in MLIR are: the `ml_program` dialect contains operations and types for machine learning frameworks such as TensorFlow, PyTorch, and JAX; The `tensor` dialect is intended for the creation and manipulation of tensors targeting AI and ML application domains; The `linalg` dialect aims at expressing linear algebra operations and computations like matrix multiplication, convolution, and tensor contractions.

2. *Standard Dialects*: MLIR comes up with a set of standard dialects that define basic operations and types commonly used across other dialects. Such a standard dialect serves as a foundation for building custom dialects and ensures interoperability between different parts of the compiler stack. They include primitive operations like arithmetic operations, control flow constructs, and data types such as integers and floats. Example standard dialects in MLIR are: `arith` - to represent basic operations like add, subtract, multiply, and division; `math` - to perform scientific operations like exponential, power, absolute value and others; `scf` - to represent standard control flow structures like if-then, for loop, and while loop.

3. *Specific Dialects*: There are dialects which are created specifically for a particular framework but are widely used across many applications. For example, the `omp` and `mpi` dialects are used to represent the constructs of OpenMP and MPI frameworks, respectively. The `vector` dialect is to represent vector instructions and related vector operations.

4. *Backend Dialects*: Backend dialects are used to represent the target hardware architecture or instruction set architecture (ISA) for code generation. They include operations and constructs specific to the target platform, enabling the translation of high-level IR into low-level machine code or intermediate representations specific to the target. Among backend dialects there are: `gpu`, `nvgpu`, `nvvm`, and `rocdl` dialects targeting different GPU architectures, or the `x86vector`, `arm_neon`, and `arm_sve` dialects to target different vector instruction sets.

3.1.2 Lowering

Lowering a dialect in MLIR refers to the process of transforming operations and constructs from a higher-level dialect into a lower-level representation that is closer to the target execution environment or hardware platform. MLIR has a wide range of lowering passes to transform high-level operations from the source dialect into equivalent or semantically similar operations in the target dialect. These passes may involve pattern matching, rewriting rules, or translation routines to map high-level constructs to lower-level representations.

An example of a high-level abstraction dialect is `linalg` which defines operations on matrices, and comes with a set of optimizations that can take advantage of some mathematical properties. For instance `linalg` offers a matrix multiplication operation, whose arguments are two input matrices and one output matrix, expressed as either memory areas or tensors. Below is an example with memory areas specified for fixed sized 8x8 matrices of floats.

```
%r= linalg.matmul { %A, %B, %C }: memref<8x8xf32>, memref<8x8xf32>, memref<8x8xf32>
```

The way this matrix multiplication is implemented is deferred to a lowering process implemented as an MLIR pass. Figure 3.1 illustrates the three available options to lower `linalg.matmul` in a more concrete expression (the relevant passes' names are indicated on arrows).

These lowering passes have in common to generate loop nests to implement the matrix multiplication, but the one to choose depends on the type of code transformation desired afterwards. It might me a lowering to: (a) `affine.for` to further apply polyhedral-related optimizations on the loop nests; (b) `affine.parallel` to evince parallel execution, that can later translate into the `omp` dialect for example; or (c) `scf.loop` which is the standard way of representing control flows. Finally, the code is lowered in the `llvm` dialect.

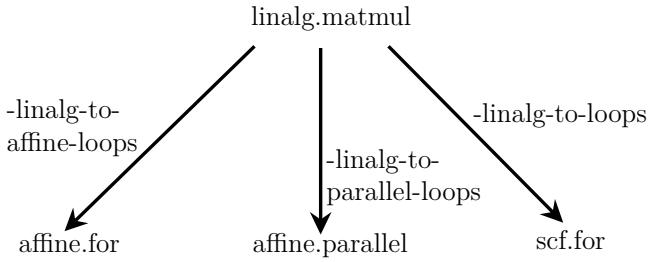


Figure 3.1: Lowering of `linalg` dialect.

3.1.3 Translation

The final phase is the translation of the `llvm` dialect MLIR code into LLVM IR. MLIR has a translation pass that helps the user to achieve this transformation. Then, with standard LLVM compilation the translated code is compiled into an object code.

3.1.4 Example

Let us illustrate the *lowering* and *translation* passes with an example shown in Listing 3.1. We show the MLIR representation at different abstraction levels and the successive lowering.

1. *Example.* Listing 3.1 is a C program consisting of a two-level nested loop that contains one arithmetic operation. A and B are two-dimensional arrays, with A being updated with B along with a constant value.

2. *MLIR representation.* Listing 3.2 is the hand written MLIR representation for the example `C` program, shown in listing 3.1. The two-level nested loops are expressed using the `affine` dialect and the operations using the `arith` dialect. Note that the loop could also have been represented using `scf`, but to apply optimization with affine transformation the loops need to be in `affine`. For example, we can extract polyhedral representation from these affine loops and apply polyhedral-related optimizations.

3. *Polyhedral optimization.* Listing 3.3 is the polyhedral tiled `affine` loops. The tile size is set to 32 and the loops are tiled accordingly.

4. *Lowering.* The nested `affine` (high-level abstraction) loops are lowered to `scf` (mid-level abstraction) loops. Listing 3.4 shows the lowered MLIR with the help of *lowering* passes on Listing 3.3.

5. *Lowering.* Listing 3.5 shows the further lowering of MLIR and all the statements are represented in the `llvm` (low-level abstraction) dialect. The lowering has made the loops be represented using basic blocks with branch statements denoting the control flow. The arithmetic operations are represented using `llvm` arithmetic operations.

```
value = 4.0;
for (int i = 0; i < 416; i++) {
 for (int j = 0; j < 416; j++) {
  A[i][j] = B[j][i] + value;
 }
}
```

Listing 3.1: An example C program with loops and arithmetic operations.

```
affine.for %arg3 = 0 to 416 {
 affine.for %arg4 = 0 to 416 {
  %cst = arith.constant 4.000000e+00 : f32
  %0 = affine.load %arg2[symbol(%arg4), symbol(%arg3)] : memref<?x416xf32>
  %1 = arith.addf %0, %cst : f32
  affine.store %1, %arg0[symbol(%arg3), symbol(%arg4)] : memref<?x416xf32>
 }
}
```

Listing 3.2: MLIR representation of example C program (Listing 3.1) using `affine` and `arith` dialects.

6. *Translation.* Listing 3.6 is the final translated LLVM IR from Listing 3.5 with the help of a MLIR *translation* pass. Then, the LLVM IR can be further optimized with the help of LLVM optimization passes and compiled into an object file.

Overall, MLIR's unified representation (*dialects*), modular architecture, and optimization infrastructure make it well-suited for developing efficient and scalable compiler pipelines for a wide range of programming languages and target platforms.

3.2 MLIR Representation with its Compilation Flow

In Chapter 2 we discussed the compute intensive kernels as they consume most of the computation time and optimizing them results in high performance gains. So in our approach, the main focus is on optimizing compute intensive kernels using MLIR and let the rest of the code be in `C/C++`.

Two issues have to be addressed in order to generate MLIR code for the compute intensive kernels. The first one is to extract the code of those kernels from the rest of the application. Such extraction is carried out on the AST. An AST is constructed

```
#map = affine_map<(d0) -> (d0 * 32)>
#map1 = affine_map<(d0) -> (d0 * 32 + 32)>
affine.for %arg3 = 0 to 13 {
 affine.for %arg4 = 0 to 13 {
  affine.for %arg5 = #map(%arg3) to #map1(%arg3) {
   affine.for %arg6 = #map(%arg4) to #map1(%arg4) {
    %cst = arith.constant 4.000000e+00 : f32
    %0 = affine.load %arg2[symbol(%arg6), symbol(%arg5)] : memref<?x416xf32>
    %1 = arith.addf %0, %cst : f32
    affine.store %1, %arg0[symbol(%arg5), symbol(%arg6)] : memref<?x416xf32>
   }
  }
 }
}
```

Listing 3.3: The `affine` loops in Listing 3.2 are tiled with the help of polyhedral representation.

```
%c0 = arith.constant 0 : index
%c13 = arith.constant 13 : index
%c1 = arith.constant 1 : index
scf.for %arg3 = %c0 to %c13 step %c1 {
 scf.for %arg4 = %c0 to %c13 step %c1 {
  %c32 = arith.constant 32 : index
  %0 = arith.muli %arg3, %c32 : index
  %1 = arith.muli %arg3, %c32 : index
  %2 = arith.addi %1, %c32 : index
  scf.for %arg5 = %0 to %2 step %c1 {
   %3 = arith.muli %arg4, %c32 : index
   %4 = arith.muli %arg4, %c32 : index
   %5 = arith.addi %4, %c32 : index
   scf.for %arg6 = %3 to %5 step %c1 {
    %cst = arith.constant 4.000000e+00 : f32
    %0 = memref.load %arg2[%arg6, %arg5] : memref<?x416xf32>
    %1 = arith.addf %0, %cst : f32
    memref.store %1, %arg0[%arg5, %arg6] : memref<?x416xf32>
   }
  }
 }
}
```

Listing 3.4: The `affine` dialect (high-level abstraction) from Listing 3.3 is lowered to `scf` dialect (mid-level abstraction) using MLIR lowering pass.

```
  %16 = llvm.mlir.constant(0 : index) : i64
  %17 = llvm.mlir.constant(13 : index) : i64
  %18 = llvm.mlir.constant(1 : index) : i64
  llvm.br ^bb1(%16 : i64)
 ^bb1(%19: i64):  // 2 preds: ^bb0, ^bb11
  %20 = llvm.icmp "slt" %19, %17 : i64
  llvm.cond_br %20, ^bb2, ^bb12
 ^bb2:  // pred: ^bb1
  llvm.br ^bb3(%16 : i64)
 ^bb3(%24: i64):  // 2 preds: ^bb2, ^bb10
  %25 = llvm.icmp "slt" %24, %17 : i64
  llvm.cond_br %25, ^bb4, ^bb11
// some code skipped ...
  %116 = llvm.mlir.constant(4.000000e+00 : f32) : f32
  %117 = llvm.extractvalue %15[1] : !llvm.struct<(ptr, ptr, i64, array<2 x i64>,
    array<2 x i64>)>
  %118 = llvm.mlir.constant(416 : index) : i64
  %119 = llvm.mul %arg8, %118  : i64
  %120 = llvm.add %119, %arg7  : i64
  %121 = llvm.getelementptr %117[%120] : (!llvm.ptr, i64) -> !llvm.ptr, f32
  %122 = llvm.load %121 : !llvm.ptr -> f32
  %123 = llvm.fadd %122, %116  : f32
  %124 = llvm.extractvalue %7[1] : !llvm.struct<(ptr, ptr, i64, array<2 x i64>,
    array<2 x i64>)>
  %125 = llvm.mlir.constant(416 : index) : i64
  %126 = llvm.mul %arg7, %125  : i64
  %127 = llvm.add %126, %arg8  : i64
  %128 = llvm.getelementptr %124[%127] : (!llvm.ptr, i64) -> !llvm.ptr, f32
  llvm.store %123, %128 : f32, !llvm.ptr
```

Listing 3.5: The `scf` and `arith` dialects from Listing 3.4 is further lowered to `llvm` dialect (last low-level abstraction) using MLIR lowering pass.

```
36:                                               ; preds = %77, %21
  %37 = phi i64 [ %78, %77 ], [ 0, %21 ]
  %38 = icmp slt i64 %37, 13
  br i1 %38, label %39, label %79
39:                                               ; preds = %36
  br label %40
40:                                               ; preds = %75, %39
  %41 = phi i64 [ %76, %75 ], [ 0, %39 ]
  %42 = icmp slt i64 %41, 13
  br i1 %42, label %43, label %77
43:                                               ; preds = %40
  %44 = mul i64 %37, 32
  %45 = mul i64 %37, 32
  %46 = add i64 %45, 32
  br label %47
// some code skipped ...
  %131 = extractvalue { ptr, ptr, i64, [2 x i64], [2 x i64] } %30, 1
  %132 = mul i64 %8, 416
  %133 = add i64 %132, %7
  %134 = getelementptr float, ptr %131, i64 %133
  %135 = load float, ptr %134, align 4
  %136 = fadd float %135, 0.000000e+00
  %137 = extractvalue { ptr, ptr, i64, [2 x i64], [2 x i64] } %23, 1
  %138 = mul i64 %7, 416
  %139 = add i64 %138, %8
  %140 = getelementptr float, ptr %137, i64 %139
  store float %136, ptr %140, align 4
```

Listing 3.6: Finally the MLIR translation pass converts the `llvm` dialect in MLIR representation to LLVM IR.

by a front-end, which might be the one built by a general purpose compiler, such as Clang, or a specialized one when we use a DSL. In the former case, the identification of the kernel is most often realized by the programmer who is responsible for marking the code, for instance with directives. We will see in Chapter 6 that the marked sections can be *static control parts*, marked with `#pragma scop`. In the latter case, the program's structure most often implies a same template for all programs, in which the compute intensive kernels are always contained in some specific constructs. In our case of the EasyML DSL, we know that the code generator (the frontend) always generates a *compute* function that represents the compute intensive part of the model (see for example the code of Listing 2.2 generated from Listing 2.1).

The second issue is to generate an equivalent MLIR code for the compute intensive functions in a reasonably portable way. First, we must be able to map all compute and control instructions from the AST to some equivalent instructions available in any of the MLIR dialects. All along our work, we found that it was possible to establish a one-to-one mapping between the AST-encoded instructions and MLIR instructions available in the `arith`, `math`, and `affine` dialects.

Once this mapping established, we could output raw MLIR code (that is output native MLIR instructions directly to a file). However, given the fast evolution of MLIR that results in frequent changes in both syntax and structure of dialects, we take advantage of the *python bindings* that come with MLIR. The MLIR python bindings[1] provide a simpler and more portable means to generate MLIR code.

[1]More details about python bindings is available at https://mlir.llvm.org/docs/Bindings/Python/

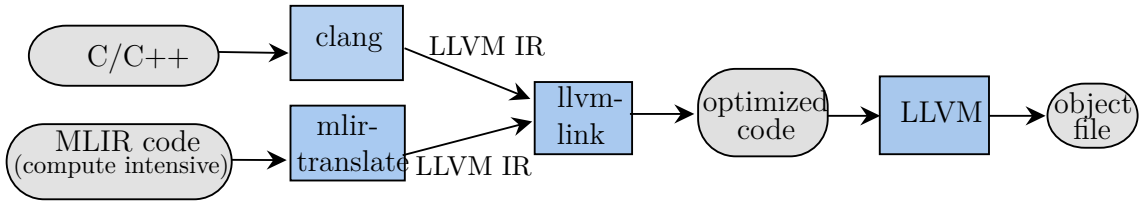


Figure 3.2: Generalized compilation view of MLIR and `C/C++` code

Thus, instead of generating raw MLIR, we generate a python program that in turn will output the raw MLIR. For example, assume we have the AST corresponding to the program presented in Listing 3.1. When reaching the AST node containing the addition instruction inside the loop nest (line 3), we add to the python program the method call `dialects.arith.AddFOp(op1, op2)` so that when executed it registers an arithmetic floating-point addition to the current context. Eventually, the execution of the python program generates for this line an equivalent `arith.addf %0, %cst : f32` MLIR instruction, where `%0` is the array deference value (`B[j][i]`) and `%cst` is the `value`. Both of type `f32` and the instruction does floating-point addition.

The python bindings support different functions to map different types for an operation. Likewise, we can generate MLIR representation for other `C/C++` statements. And, if the compute intensive code is a parallel loop with complex statements then those statements can be represented directly using MLIR vector instructions and that will force the compiler to vectorize compute instructions.

Compilation Flow. Figure 3.2 shows the generalized compilation view of MLIR and `C/C++` code using the LLVM compiler infrastructure. From the MLIR translation pass, we have the final LLVM IR for the compute intensive kernels and with the help of LLVM infrastructure we emit LLVM IR for the rest of the code that is in `C/C++`. Then, we link both of them together into an optimized IR. The standard LLVM compiler further compiles them to an object code. Finally, at the time of software execution the optimized MLIR represented compute intensive code can be executed with a function call from the `C/C++` file. This MLIR representation and compilation technique can be applied to any kernel irrespective of the application domain.

In this Chapter, we showed the advantages of MLIR dialects and their optimization passes with an example. We also discussed the techniques to generate MLIR representation for a piece of kernels and a generalized view of compiling MLIR and `C/C++` code together to an object file. In the next Chapter, we will see how MLIR and the discussed techniques get incorporated into a cardiac simulator for generating heterogeneous code.

Chapter 4

Heterogeneous Code Generation for Cardiac Simulation Application

In this chapter, we apply the MLIR code generation techniques seen in the previous chapter to the openCARP cardiac simulator. One of the challenges addressed here is to enable code generation for two types of accelerators, namely vector units of CPUs, and GPUs. First, in Section 4.1, we show how the techniques to generate MLIR representation (discussed in Chapter 3) get incorporated into openCARP and also the other required changes in openCARP to support the overall compilation flow. Then, a detailed evaluation is reported in Section 4.2 using the 48 different ionic models available in openCARP. We show improvements both in terms of performance and energy efficiency, for (a) an Intel machine with an AVX-512 vector instruction set, (b) an Nvidia GPU, and (c) an AMD GPU.

4.1 Overview of CPU Vectorized and GPU Code Compilation Flow

In this section we detail our contribution that extends the initial openCARP code generator. As explained in Section 2.4.1, the `limpet_fe` python script of openCARP parses a model written in EasyML as an AST encoded with python data structures, from which C/C++ code is generated. Our extension consists in extending `limpet_fe` with an auxiliary script *limpetMLIR*, capable of substituting part of the code generation to produce MLIR code instead. From this MLIR code we can then choose to emit vectorized CPU or GPU code.

Based on the principles discussed in section 3.2, especially the dedicated view of the compilation process of Figure 3.2, we develop the global compilation flow depicted in Figure 4.1, showing how our extension fits in the framework. Let us describe the different phases as numbered in the figure:

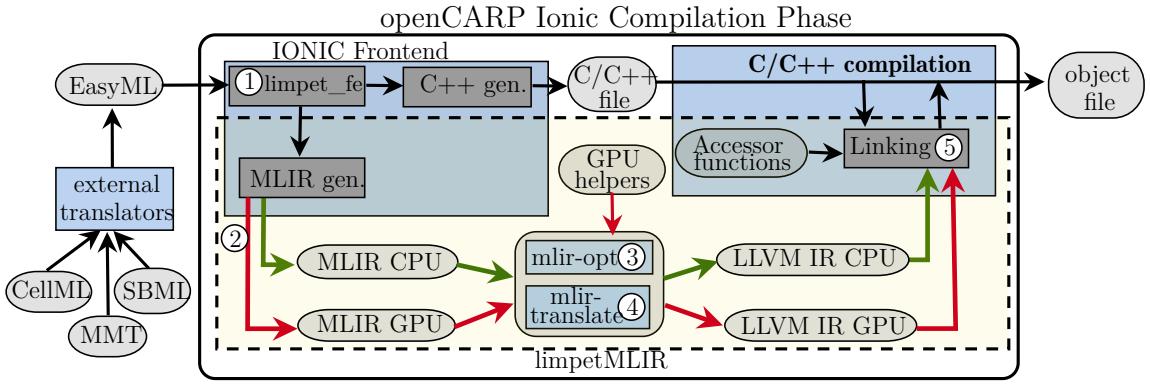


Figure 4.1: Overview of the code generation, from the EasyML model to an object file. The dashed line box shows how *limpetMLIR* fits into the original code generation process, to emit optimized code for CPU and GPU.

① From the EasyML description the `limpet_fe` python program creates an AST, which serves as a common entry point for the *baseline* openCARP and *limpetMLIR*.

② The *limpetMLIR* code generator emits MLIR code for the compute-intensive kernel loop using the `scf`, `arith`, `math`, `vector` and `memref` dialects. The emitted code is slightly different for CPU or GPU:

- For CPU vector code generation: we let the user choose the vector size -of either 2, 4, or 8- with the help of an environment variable and following which all the emitted MLIR instructions will be of type `vector<?xf64>`. In listing 4.1, the vector size is chosen to be 8 targeting AVX-512 vector instruction set with `double` types.
- For GPU code generation: the type used is `<f64>` and the control flow expressed in `scf` allows the latter MLIR passes to lower it to a parallel control flow in the `gpu` dialect. In listing 4.2, lines 2-3 show the two `scf.for` loops, which will be translated into an outer loop and an inner loop iterating over the GPU `blocks` and `threads` within a block respectively.

③ The MLIR lowering pass converts the MLIR code either to:

- an OpenMP CPU vectorized code where the `scf.for` shown in line 2 of listing 4.1 is lowered to an OpenMP work-sharing and parallel loop.
- a GPU device code part using a specific GPU low-level dialect. This low-level dialect can be either `nvvm` (the Nvidia CUDA IR) or `rocdl` (the AMD ROCm IR) depending on the target GPU architecture.

④ The MLIR translator pass converts the MLIR code (represented using the `llvm` dialect) into a vectorized CPU or GPU device (as part of GPU compilation) code part to LLVM IR.

⑤ Last is the linking phase, where `C/C++` and LLVM IR CPU and GPU files are linked together into an object file using LLVM.

```
1  %c8 = arith.constant 8 : index
2  scf.for %arg9 = %0 to %1 step %c8 { // parallel loop
3    %14 = func.call load_vector(%13, %c0_i32_3) : (memref<1xi8>, i32) -> vector<8xf64
       >
4    // ... code skipped for space
5    %cst_8 = arith.constant dense<7.000000e-02> : vector<8xf64>
6    %cst_9 = arith.constant dense<2.000000e+01> : vector<8xf64>
7    %35 = arith.divf %20, %cst_9 : vector<8xf64>
8    %36 = math.exp %35 : vector<8xf64>
9    %37 = arith.mulf %cst_8, %36 : vector<8xf64>
10   %cst_10 = arith.constant dense<1.000000e-01> : vector<8xf64>
11   %cst_11 = arith.constant dense<2.500000e+01> : vector<8xf64>
12   %38 = arith.addf %20, %cst_11 : vector<8xf64>
13   %39 = arith.mulf %cst_10, %38 : vector<8xf64>
14   %40 = arith.addf %20, %cst_11 : vector<8xf64>
15   %41 = arith.mulf %cst_10, %40 : vector<8xf64>
16   %42 = math.exp %41 : vector<8xf64>
17   %cst_12 = arith.constant dense<1.000000e+00> : vector<8xf64>
18   %43 = arith.subf %42, %cst_12 : vector<8xf64>
19   %44 = arith.divf %39, %43 : vector<8xf64>
20   %cst_13 = arith.constant dense<1.000000e-02> : vector<8xf64>
21   %cst_14 = arith.constant dense<1.000000e+01> : vector<8xf64>
22   %45 = arith.addf %20, %cst_14 : vector<8xf64>
23   %46 = arith.mulf %cst_13, %45 : vector<8xf64>
24   %47 = arith.addf %20, %cst_14 : vector<8xf64>
25   %48 = arith.mulf %cst_10, %47 : vector<8xf64>
26   %49 = math.exp %48 : vector<8xf64>
27   %50 = arith.subf %49, %cst_12 : vector<8xf64>
28   %51 = arith.divf %46, %50 : vector<8xf64>
29   // ... code skipped for space
30   %104 = arith.mulf %99, %15 : vector<8xf64>
31   %105 = math.fma %90, %94, %104 : vector<8xf64>
32   func.call store_vector(%8, %105, %c16_i32_22):(memref<1xi8>, vector<8xf64>, i32)
       -> ()
33   %c0_i32_23 = arith.constant 0 : i32
34   func.call store_vector(%11, %34, %c0_i32_23):(memref<1xi8>, vector<8xf64>, i32)
       -> ()
35    // ... code skipped for space
```

Listing 4.1: MLIR code snippet for vectorized CPU generated by *limpetMLIR* for the Hodgkin and Huxley ionic model shown in listing 2.1.

Notice that the final system linking phase requires a vector-capable mathematical library (that is not the case for the standard *libm*) in order for the mathematical function calls to be vectorized, even though they have been vectorized in the MLIR code. We rely on Intel's Short Vector Math Library (SVML) library for this purpose in our experiments presented below.

Listing 4.1 and listing 4.2 show the vector CPU and GPU code snippets generated by *limpetMLIR* for the parallel loop for the Hodgkin and Huxley ionic model shown in listing 2.2. At this point, we cannot avoid to take into account the required parallel loop structure which depends on the hardware target.

For the CPU code, we target an OpenMP `parallel for` loop, so we generate a single `scf.loop`, as seen in listing 4.1, line 2. Regarding the compute instructions, the target vector instruction set is AVX-512 and 8 is chosen as the vector size. Lines 5-9 are the equivalent MLIR code using the `arith` and `math` dialects for line 21 in listing 2.2. The vector loads and stores are done with the help of *Accessor* functions (discussed later) seen in lines 3, 32, and 34.

For the GPU code, lines 2 and 3 in listing 4.2 are the two `scf.for` loops within

```
%7 = memref.load %6[%c0_3] : memref<?xf64>
scf.for %arg9 = %0 to %3 step %c1 { // iterate over blocks
 scf.for %arg10 = %c0 to %c512 step %c1 { // iterate over threads
 // ... code skipped for space
  scf.execute_region {
   %11 = arith.cmpi slt, %9, %1 : index
   cf.cond_br %11, ^bb1, ^bb2
  ^bb1:  // pred: ^bb0
   // ... code skipped for space
   %cst_22 = arith.constant 7.000000e-02 : f64
   %cst_23 = arith.constant 2.000000e+01 : f64
   %50 = arith.divf %31, %cst_23 : f64
   %51 = math.exp %50 : f64
   %52 = arith.mulf %cst_22, %51 : f64
   %cst_24 = arith.constant 1.000000e-01 : f64
   %cst_25 = arith.constant 2.500000e+01 : f64
   %53 = arith.addf %31, %cst_25 : f64
   %54 = arith.mulf %cst_24, %53 : f64
   %55 = arith.addf %31, %cst_25 : f64
   %56 = arith.mulf %cst_24, %55 : f64
   %57 = math.exp %56 : f64
   %cst_26 = arith.constant 1.000000e+00 : f64
   %58 = arith.subf %57, %cst_26 : f64
   %59 = arith.divf %54, %58 : f64
   // ... code skipped for space
   %118 = arith.mulf %113, %25 : f64
   %119 = math.fma %104, %108, %118 : f64
   %125 = arith.truncf %119 : f64 to f32
   %c0_42 = arith.constant 0 : index
   memref.store %125, %124[%c0_42] : memref<?xf32>
   %127 = memref.view %22[%c0_46][%c4_47] : memref<1xi8> to memref<?xf64>
   %c0_48 = arith.constant 0 : index
   memref.store %31, %127[%c0_48] : memref<?xf64>
   // ... code skipped for space
```

Listing 4.2: MLIR code snippet for GPU generated by *limpetMLIR* for the Hodgkin and Huxley ionic model shown in listing 2.1.

which all the loop statements are embedded, such that the outer loop iterates over blocks and the inner loop over the threads within blocks on the GPU. For the GPU computations, we use the default `<f64>` (double) type instead of the `vector` type.

4.1.1 *LimpetMLIR* for CPU and GPU

In this section, we describe the additional changes that are required for the compilation of generated MLIR code along with openCARP.

Data access. *Accessor* functions are implemented to retrieve values of external variables of ionic models, and state variables of a cell. Stride accesses are enabled by `gather` and `scatter` operations from the `vector` dialect, allowing to fetch state variables stored in non-contiguous memory addresses. We also generate *accessor* functions for single-valued broadcasts, and contiguous memory accesses, the latter being necessary for our code transformation discussed in section 4.1.2. Lines 3 and 32 from listing 4.1 show examples of *accessors* for contiguous and non-contiguous memory accesses that fetch data from eight cells in parallel.

Multimodel support. Electrophysiology simulations also allow multiple models to interact, accessing the same data. This leads to a hierarchy of cells relying on a parent-

offspring relation. Offspring cells are allowed to access and modify the content (or state) of their parent. In the openCARP framework, this feature is supported through a combination of conditional statements that check the existence of the parent and its values, and function pointers that connect the appropriate parent data with its offspring. We support this feature by conditionally accessing data from the parent through MLIR `gather` and `scatter` operations that also handle such conditions. If the parent information cannot be found, it falls through the common local variable storage.

The code generation process described previously supports all of the features found in the original openCARP code generator. More precisely, all 48 ionic models for cardiac cell simulation are supported, illustrating the flexibility and completeness of our code design.

Helper Functions. During the compute stage (so inside the kernel) openCARP typically performs function calls that cannot be inlined with MLIR. By analyzing the generated assembly code and checking the hardware performance counters in our benchmarks, we noticed that in many models, these function call were costly. The first type are calls are to the lookup table (LUT) to retrieve pre-computed interpolated values for complex mathematical functions (in order to speed-up computations). LUT related functions are called very often and found in almost all models. In addition, there are calls in some models to specific integration methods, such as *Rosenbrock* (to perform LU decomposition and integration). These types of calls escapes the MLIR code generation because it is very hard to automatically generate MLIR code for those function calls.

Initially, we followed a naive approach and relied on the original scalar implementation of these functions but there was a considerable slowdown in both CPU and GPU codes. This is because of the execution shift from vector to scalar and vice-versa for CPU execution, and the system calls and memory copy between device and host for GPU execution. To overcome this issue, we implemented a fully vectorized MLIR version of LUT interpolation and Rosenbrock functions, leading to a considerable gain in CPU performance. For GPU, we write their respective implementations in GPU device code such that they are called and executed on GPU without any call back to CPU. For example in listing 4.3, lines 5 and 7 are the respective function calls.

GPU Memory Management. One well-known pitfall regarding performance is the data transfers between host and device because of the PCIe bus bottleneck. These necessary transfers are not part of the MLIR code generation process, but are inserted into the code that wraps the ionic model computation for the following reasons: (i) we want to keep the structure of the MLIR code as similar as possible for all types of devices, so we focus on generating MLIR for the compute function only, (ii) other openCARP software parts access this memory (e.g solvers), and (iii) we want to precisely control the data movements behavior regarding performance. We implement the memory management preferably using unified memory with `cudaMallocManaged` or `hipMallocManaged`. As a side note, it happened on our AMD test platform that `hipMallocManaged` is not supported and falls back to inefficient data transfers. In that case, we could easily change it to explicit allocation and memory copies between host and device.

```
// ... code skipped for space
scf.execute_region {
 // ... code skipped for space
 %147 = memref.view %146[%c0_127][] : memref<?xi8> to memref<1xi8>
 func.call LUT_interpRow(%144,%122,%147):(f64,i32,memref<1xi8>)->()
 // ... code skipped for space
 func.call rosenbrock_StepX(%814,%830,%c3):(memref<1xi8>,f32, i32)->()
 // ... code skipped for space
```

Listing 4.3: MLIR code snippet generated for GPU by *limpetMLIR* for the Bondarenko model

Implementation Effort. For our implementation, we wrote about 10k source lines of code (39% python, 26% MLIR, 23% `C++`, some GPU kernel and CMake code). The total auto generated lines of code for all 48 ionic models are as follows: *baseline*: 39621; vectorized *limpetMLIR*: 111883; GPU: 78025.

4.1.2 Data Layout Transformation

A data layout transformation is implemented to avoid the effects of non-consecutive data storage of variables within ionic models in memory. From the data perspective, ionic models are described as a combination of shared and private information among cells. While the former is defined as a read-only region that delivers no optimization opportunities (SIMD memory loads of a single data are usually efficiently implemented by the hardware), the latter has been originally modeled as to regroup values of a single cell in a contiguous manner (an array-of-structures, AoS). This design becomes non-optimal when multiple cells are processed in parallel, as is the case of vector memory accesses in our solution.

We implemented a data layout transformation to avoid the effects of non-consecutive data storage of cell variables within ionic models. This classical approach consists of rearranging the same state variable from successive ionic cells consecutively: data is stored in an array-of-structures-of-blocks (or array-of-structures-of-arrays, AoSoA) form [40, 41], a combination of the classical array-of-structures (AoS, non-consecutive) and structure-of-arrays (SoA, completely consecutive but large) forms. Using the AoSoA data storage format, we:

1. avoid memory operations on addresses that are far from one another - and thus avoid TLB misses,
2. improve data locality - and thus improve cache accesses,
3. enable contiguous efficient vector load/store hardware operations.

Figure 4.2 shows the proposed data layout optimization for the vector size of 4. The data storage is transformed to array-of-structures-of-arrays (AoSoA), such that with respect to the vector size the cell elements are packed together. As a result during

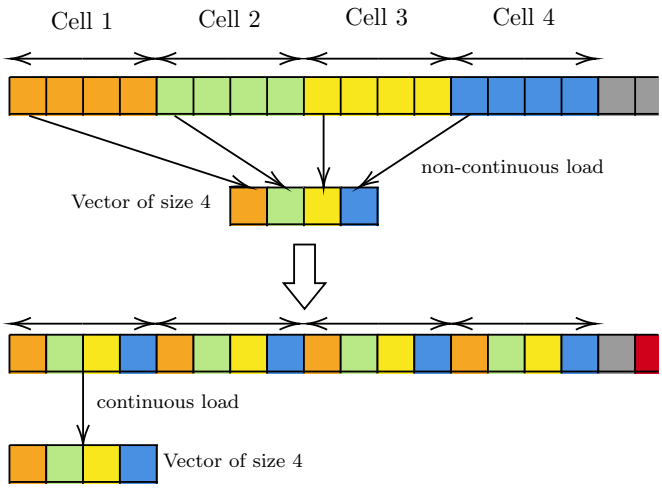


Figure 4.2: Data layout optimization for vector size of 4.

vector load/store, we could perform a contiguous load/store with respect to the vector size. This transformation is implemented as part of the code generation process, and can be enabled/disabled through a compiler flag.

4.2 Experiments and Results

We implemented *limpetMLIR* on top of the openCARP source from the git repository. We compile them using the LLVM compiler infrastructure tag `15.0.2`, which has all necessary compilation tools including Clang and MLIR. *LimpetMLIR* targets heterogeneous code generation, hence the following CPU and GPU machines were used to evaluate our generated optimized code:

- a 2x 18-core Cascade Lake Intel Xeon Gold 6240 @2.6GHz (850 GFLOP/s, 2x 150W), turbo boost and hyperthreading disabled, 192GB of RAM @2933MT/s,
- an A100 Nvidia GPU (9,700 GFLOP/s peak performance on *double*s, 400W),
- an AMD Radeon Instinct MI50 GPU (6,600 GFLOP/s, 300W).

The Intel Xeon Cascade Lake processor architecture supports all the three `SSE`, `AVX2`, and `AVX-512` vector instruction sets that we tested for CPU vectorized code. We run all 48 ionic models available in the openCARP benchmarks, using the `bench` executable to run the compute step alone and get a trace every 100 steps. Each model is run five times, the two extreme measures are eliminated and the remaining three are averaged. In the following section 4.2.1, the performance of CPU vectorized code on different vector architectures across threads 1 to 32 are discussed, in section 4.2.2 we report GPU performance improvements, and finally in section 4.2.3 we discuss the energy efficiency of the *limpetMLIR* code generator.

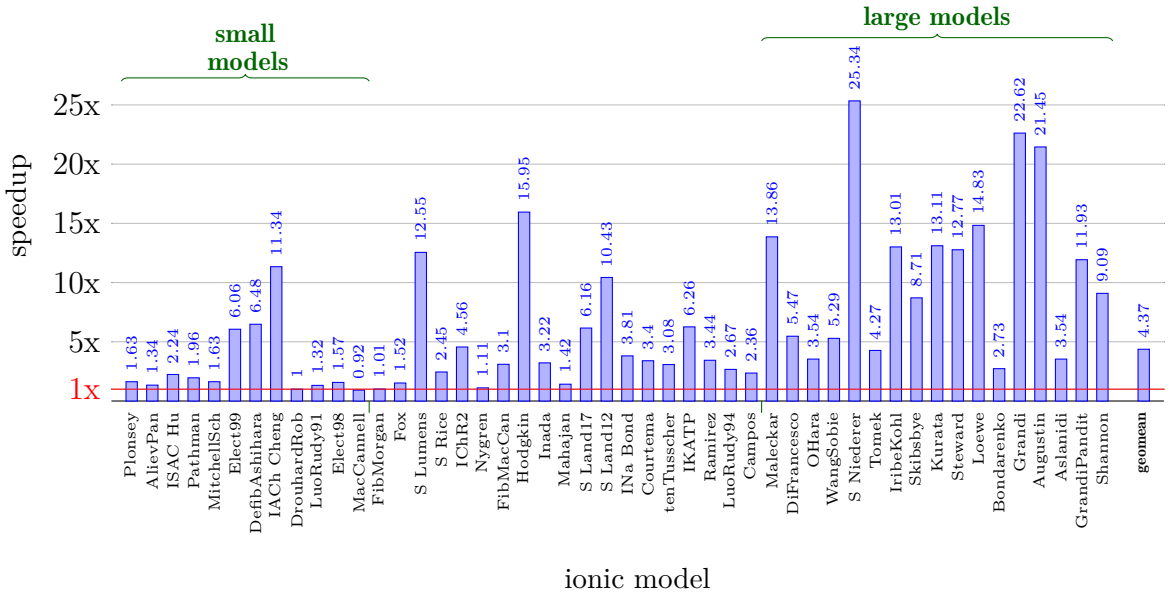


Figure 4.3: Speedup of the *limpetMLIR* vectorized CPU version of the code compared to the baseline openCARP version, using one single thread (sequential) on an `AVX-512` architecture.

4.2.1 CPU Vectorization Performance

Each model is run using a small total of 8,192 cells with a 100,000 step simulation, in order for the largest models not to take more than two hours to execute, and thus, limiting the duration of the experiments to a few hours.

Experiments were conducted using three vector architectures: `SSE` - with a vector size of two doubles; `AVX2` (four doubles); `AVX-512` (eight doubles). On each architecture, we evaluated the codes on a number of threads (and cores) ranging from 1 to 32, and used the geometric mean (*geomean*) to average speedup results in all cases.

Single Thread Execution. Figure 4.3 shows speedups comparing the execution time of the *baseline* openCARP version to our *limpetMLIR* version, both on one thread. The horizontal axis is composed of 48 ionic models, ordered from the shortest to the longest execution time from the *baseline* openCARP version. We arbitrarily split those models into three sets of *small*, *medium*, and *large* ionic models. The small set is composed of the twelve models running in less than a minute on our experimental platform, the medium one of 19 models running in 1-5 minutes, and the large one of 17 models taking more than 5 minutes. Large models are usually the most precise and close to the physiology, and as such, they are the most relevant ones for many practical applications, *e.g.*, virtual drug testing in cardiac research.

The *limpetMLIR* vectorized CPU version achieves a geomean speedup of 4.37× on `AVX-512` architecture. These results show one important aspect of our code generation: the acceleration can be much higher than the size of the vectors, up to more than 25×. Although it may sound surprising, the effects of our optimizations go far beyond the raw vectorization and CPU computation power, reaching also how memory

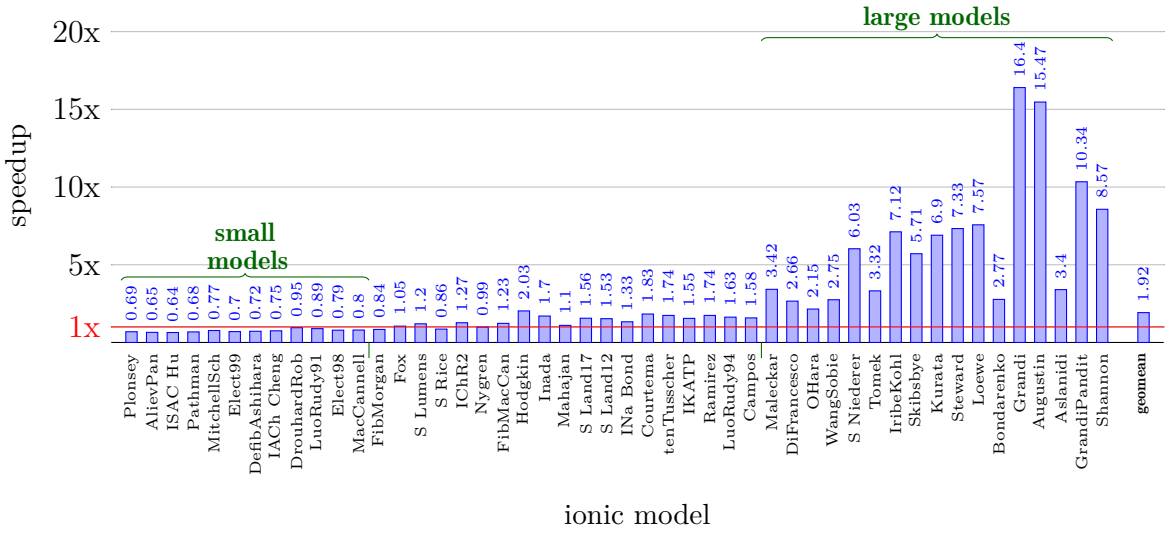


Figure 4.4: Speedup of the *limpetMLIR* vectorized CPU version of the code compared to the baseline openCARP version, using 32 OpenMP threads on a 32 cores `AVX-512` architecture.

is accessed (simultaneous load/store assembly instructions) and taking advantage of our data layout optimization (efficient use of the memory caches). A different version of the code might also trigger other compiler optimizations that affect, *e.g.*, register allocation, pipelined execution, and out-of-order execution.

The observed speedups are low and irregular in small models, and more significant and consistent for larger models. This is expected: on short codes executing short optimized loops in less than a fraction of a millisecond, it is more difficult to achieve good performance than on longer loops containing more computationally-expensive operations. Some notable exceptions (*e.g.*, ISAC Hu) are more operationally intensive than appeared to be: they share the characteristics of (1) calling costly mathematical functions that were efficiently vectorized by our optimizer *and* (2) not using lookup-tables (LUT).

Thirty-two Threads Execution. Figure 4.4 presents the speedup results on a 32 OpenMP threads execution (using 32 physical cores). The measured speedups compare the baseline and *limpetMLIR* versions in the same conditions, *both running in parallel on 32 threads*: the 1× line represent the execution time of the baseline openCARP parallel code on 32 cores. The *limpetMLIR* version achieves a geomean speedup of 1.92×, but only 0.83× on small models, 1.34× on medium models, and a very good 6.03× on large models. Smaller models with very short execution times suffer a slow-down, mainly because of the synchronization and optimization overheads, or because they are by nature memory-bound and not compute-bound.

Execution Time and Speedup over Varying Number of Threads. We confirmed our analysis of those differences between *small*, *medium*, and *large* models in fig. 4.5. We compare the average execution times (y-axis) of the three classes of models running on 1 to 32 cores (x-axis). The dashed lines represent linear speedup. All models running in less than some 50 seconds suffer a slowdown compared to the dashed line. In small

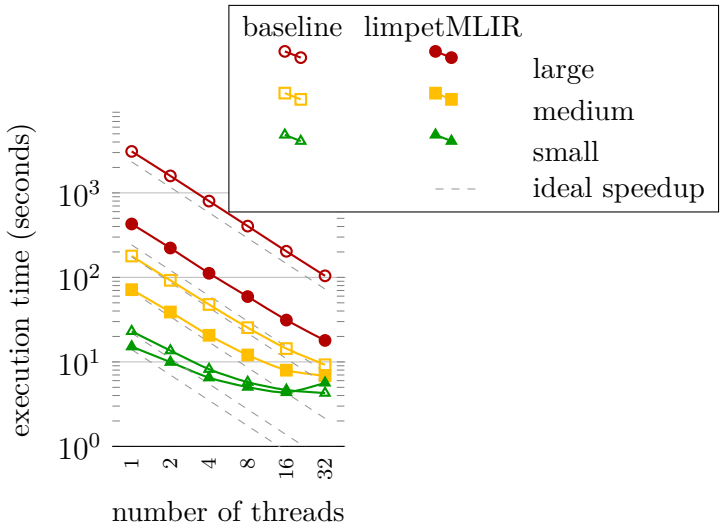


Figure 4.5: Average execution times of three classes of ionic models: small, medium, large (on `AVX-512`)

models, we observe that the scalability is very poor due to the execution of very short parallel loops: the overhead of synchronizations between threads is very high compared to the computation time itself, and the curve flattens as the number of cores increases. Although our optimizations (filled symbols) have some positive effect using a small number of threads, they induce a slowdown when reaching 32 cores, as the gain due to parallelism also completely disappears. This is less perceptible in medium models, but still present starting at 8 cores: the *limpetMLIR* execution times get closer to the baseline version at this point. In large models, the *limpetMLIR* version consistently executes $8-10\times$ faster than the baseline, along with an almost ideal parallel speedup both on the baseline and the optimized version.

Figure 4.6 shows the geometric mean speedups, with respect to the *baseline* openCARP version, achieved by the *limpetMLIR* version on all the three (`SSE`, `AVX2`, and `AVX-512`) vector architectures across varying threads from 1 to 32. In all cases the `AVX-512` architecture outperforms `AVX2` and `AVX2` outperforms `SSE` (in overall geomean). This behavior is expected as `AVX-512` calculates eight values for one hardware operation, whereas `AVX2` calculates four, and `SSE` calculates two (`double` types). Notice that this is true even if instructions cost and frequency might differ between these three vector architectures.

The difference flattens as the number of cores increases, mainly due to the slowdowns of small models. When restricting those results to the set of large models, we get consistent speedups of $3.80\times$ on `SSE`, $5.13\times$ on `AVX2`, and $6.03\times$ on `AVX-512` on 32 cores. The overall geomean speedup over all models and all architectures is $2.90\times$.

Impact of Data Layout Optimization. We found that the data layout optimization was essential in increasing the speedups of medium and large ionic models. This happens because they access more memory (state value) than smaller models. For instance, the S_Niederer model has its speedup increased from $4.98\times$ to $6.03\times$ in a 32-thread `AVX-512` configuration. The geomean speedup of all models, in a 1 to 32 thread `AVX-512` configuration, goes from $3.12\times$ to $3.37\times$ thanks to the data layout optimization.

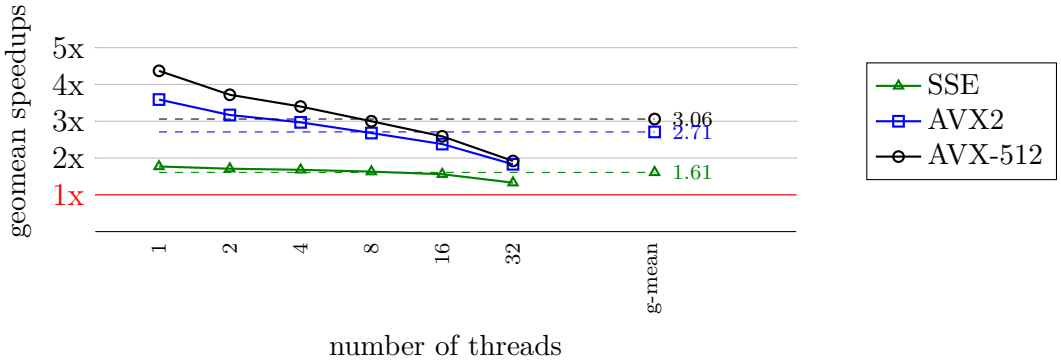


Figure 4.6: Geomean speedups for `SSE`, `AVX2`, and `AVX-512` across varying threads (in the power of two)

Roofline Model. Figure 4.7 shows the roofline model for our various ionic models. The operational intensity on the x-axis is the number of floating point arithmetic operations divided by the number of memory operations (in Flops/Byte). The number of memory loads and stores were extracted by instrumenting the generated MLIR code of the ionic models. The number of arithmetic operations were measured for each ionic model using the processor performance counters.

The y-axis of fig. 4.7 represents the GFlops/s performance, as the number of arithmetic operations divided by the execution time, on our 32 cores `AVX-512` platform. The peak performance using 32 cores was measured experimentally with the *Empirical Roofline Tool (ERT)* [42] as 760GFlops/s, DRAM bandwidth as 199GB/s, and L1 cache bandwidth as 1052GB/s. Notice that the maximum DRAM bandwidth according to the architecture specification is 140.8GB/s (shown as the lowest gray dashed line in the figure).

One can observe in fig. 4.7 that many of these codes have an operational intensity lower than the DRAM bandwidth *versus* performance limit (around 4 Flops/Byte); the majority of them are memory-bound. Codes of the large class perform quite well: those on the right of the figure are compute-bound and a bit lower the peak 760 GFlops/s limit (GrandiPanditVoigt for example), and those on the left are also close to the memory maximum bandwidth limit (OHara and WangSobie for example). OHara and some medium models (*e.g.*, Courtemanche) exceed the DRAM bandwidth thanks to their efficient cache usage.

There are models with less than 20 GFlops/s performance from the *small* and *medium* models and they are mostly memory-bound. The DrouhardRoberge model in particular does 19 GFlops/s, but its operational intensity is less than 1/4 Flops/Byte. We observed the slowdown for the small models (the *limpetMLIR* version is below the baseline version), as explained earlier (fig. 4.5) by their very short execution time.

Overall, those roofline results are as expected. The roofline model just confirms our previous analyses and shows that many of our optimized codes are reasonably close to the maximal performance of this architecture. Some improvements still seem possible in some of them (*e.g.*, Hodgkin, Maleckar), and this will be investigated in the future.

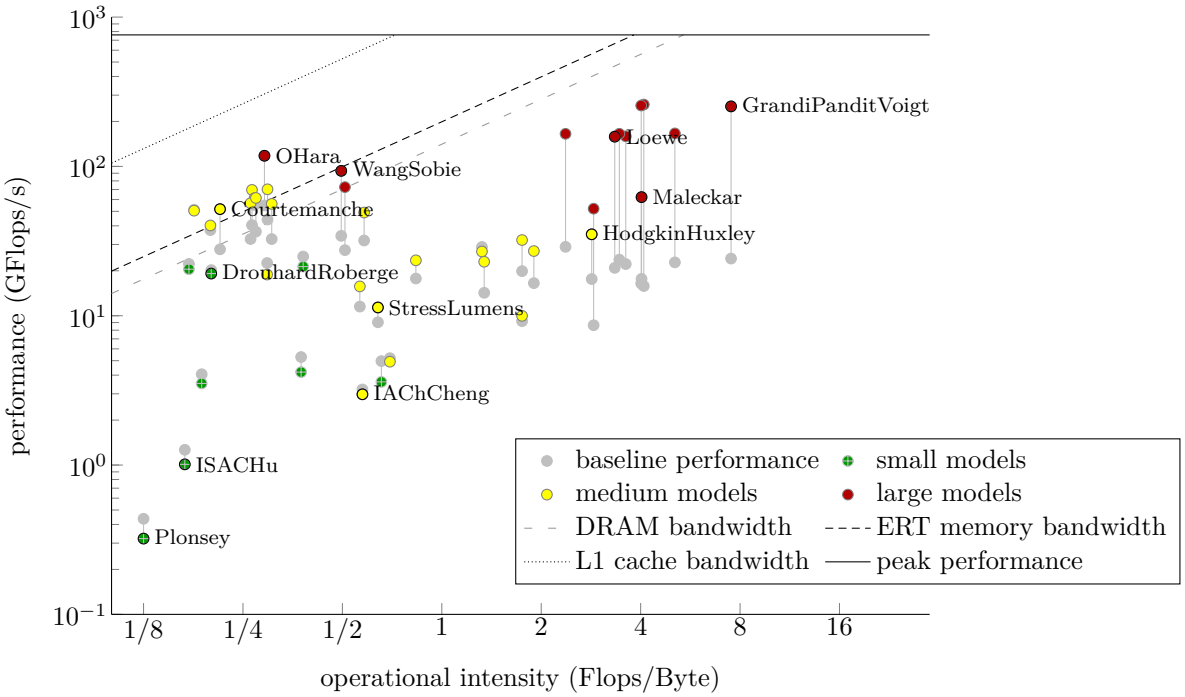


Figure 4.7: Roofline model for the different ionic models with `AVX-512` vectors on 32 cores when compared to baseline openCARP (peak performance of our experimental platform: 760 GFlops/s, DRAM bandwidth: 199 GB/s, L1 cache bandwidth: 1052 GB/s)

Comparison with Auto-vectorization. We did force the standard compilers to vectorize the parallel for loop by annotating with OpenMP `simd` directive. However, `clang` and `gcc` failed to vectorize the loop along with aggressive optimization options. Intel `icc` 19.1.3 could vectorize the loop when annotated with the OpenMP `simd` directive, but only reached an overall AVX-512 geomean speedup of 2.19×, much lower than *limpetMLIR* (3.06× as seen in fig. 4.6).

4.2.2 GPU Performance

We used 819,200 cells (higher number than the CPU experiments) with a 10,000 step simulation, in order to report the real performance benefits of GPU machine which have higher computational power. On CPU we ran (i) the 36 OpenMP threads baseline openCARP, and (ii) the 36 OpenMP threads `AVX-512` vectorized CPU *limpetMLIR* version.

The total number of floating point operations necessary to run each ionic model was measured with the hardware counters on the CPU. The GPU probably does less operations due to mathematical functions being optimized, but we used the same baseline value measured on CPU for a fair comparison.

For the GPU execution we chose a block and thread dimension of 1, a `number`

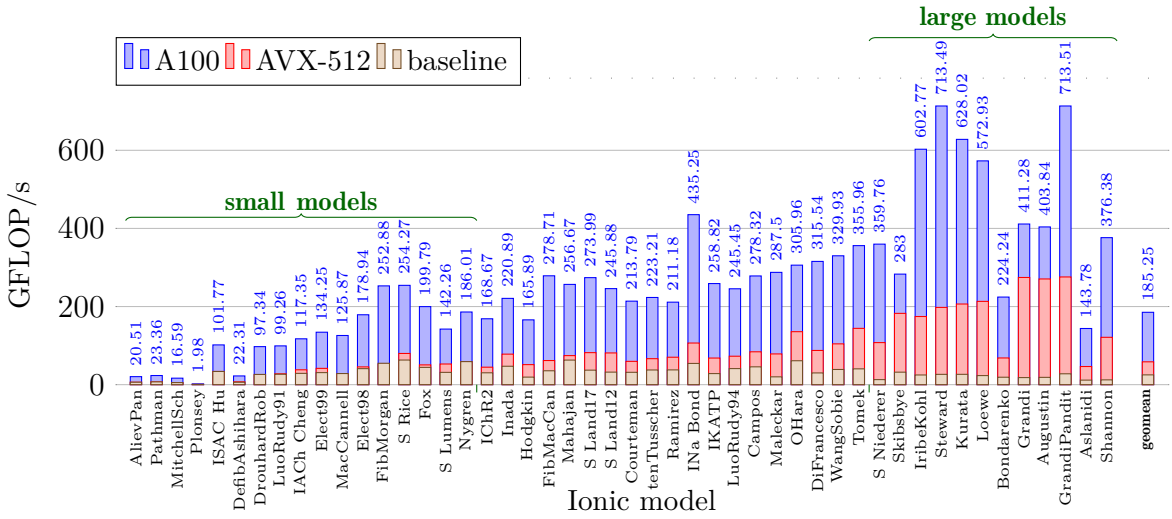


Figure 4.8: Performance on Nvidia A100, in giga floating operations per second

`of threads per block` (CTA size) of 64, and a `number of blocks` of {number of cells/64}. We empirically determined that this provides the best performance results on our platforms on average for all models. This value can be easily adjusted if running on different hardware.

Nvidia CUDA Performance. Figure 4.8 shows the floating-point operations executed per second by the CPU baseline, *limpetMLIR* `AVX-512` vectorized, and A100 GPU versions of openCARP. The x-axis lists all 48 ionic models and the y-axis is the GFLOP/s performance. On the `x-axis`, we sorted the ionic models from the shortest to the longest execution time (of the baseline).

From fig. 4.8 we can observe with no surprise that the GPU code performs better than the CPU openCARP versions in all ionic models. GPU optimized codes report the highest GFLOP/s for the large models, that perform the most computations. Overall, considering the geometric mean, we reach 185 GFLOP/s on this platform, the GPU optimized code executes 3.17× faster than the vectorized CPU code, and 7.4× faster than the baseline openCARP.

However, the model that exhibits the best performance reaches 713 GFLOP/s, that is only 1/13 of the raw performance the A100 can deliver. Also, the A100 has 11.4x the raw performance of our test bed CPU (850 GFLOP/s) so the average gain of 3.17x seems pretty low. The reason is that those ionic models, taken from a real simulation application, have a pretty low compute intensity: a geomean of 0.35 flop/byte. This means that they execute many memory operations along with floating point calculations. Better performance on large models is explained by their greater compute intensity: a geomean of 3.02 flop/byte if we exclude Bondarenko and Aslanidi. Those two specific models have lower performance results than the other ones, as they have in common to call a memory intensive integration method (Rosenbrock). Overall, the low compute intensity explains that the GPU performance is far from the maximal hardware performance, and that the CPU with multiple levels of fast and large caches is better at this.

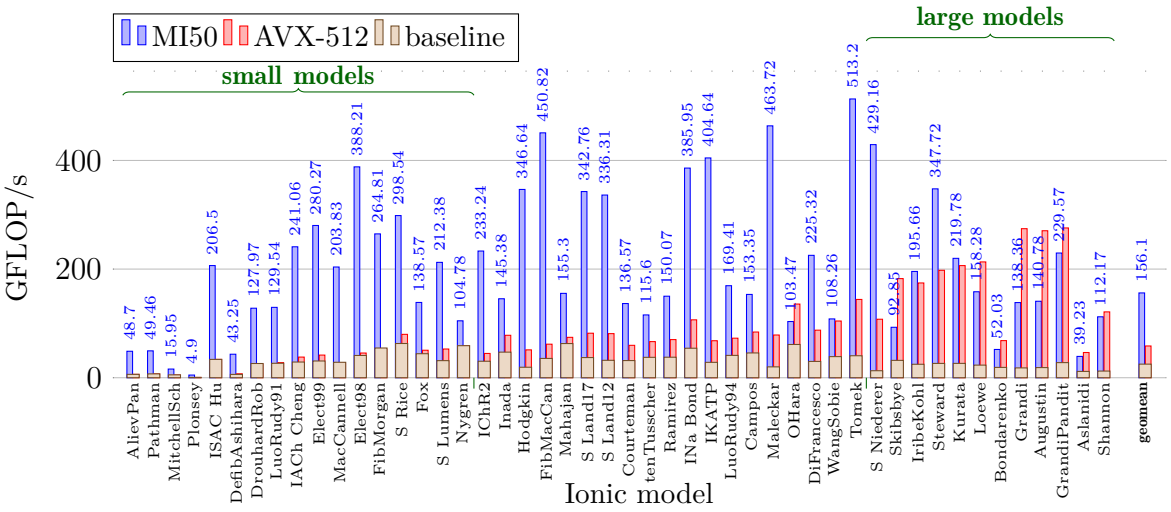


Figure 4.9: Performance on AMD MI50, in giga floating operations per second

We also report that one of those best performing model (Steward) was manually written in CUDA by our HPC expert, using explicit memory copies. We measured very similar performance between this code and the *limpetMLIR* generated code (the handwritten code is less than 5% faster).

AMD ROCm Performance. We did the same experiments on the AMD MI50, as reported in fig. 4.9. We reach a geomean performance of 156 GFLOP/s on this platform, and the overall results are pretty similar except for one point: the MI50 performs better than the A100 on small and medium ionic models while we can observe the opposite for the large models. The AMD version is sometimes even outperformed by the CPU vectorized version (for example on Grandi and Augustin). The difference in memory management (see section 4.1.1 GPU memory management) between the CUDA and ROCm implementations is the main reason for this lower performance on large models and better performance on small models.

Considering the geometric mean, the AMD ROCm *limpetMLIR* code executes 2.67× faster on MI50 than the vectorized CPU version. This number compares to 3.17× on A100, since the A100 has almost 50% more maximal raw performance than the MI50.

4.2.3 Energy Efficiency

We reported GFLOP/s raw performance results as it is good practice, but those numbers are not very meaningful when comparing completely different architectures with very different raw computing power. The FLOP per consumed Joule is a much better scale to compare them with the perspective of running on energy-aware supercomputers. We measured the total energy consumption by running the benchmarks on the CPU using the hardware counters, as the sum of package and RAM consumption; on

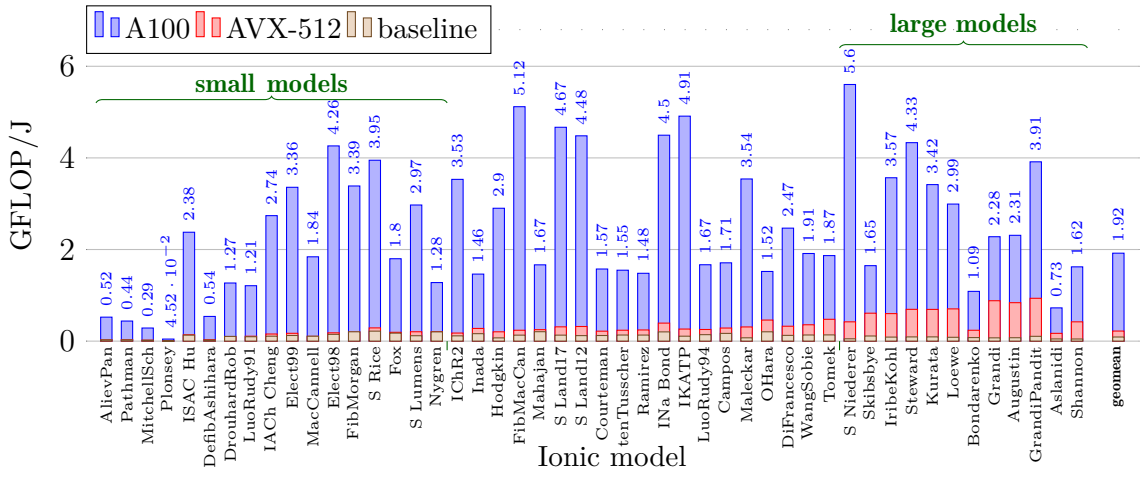


Figure 4.10: Energy efficiency on Nvidia A100, in GFLOP per Joule

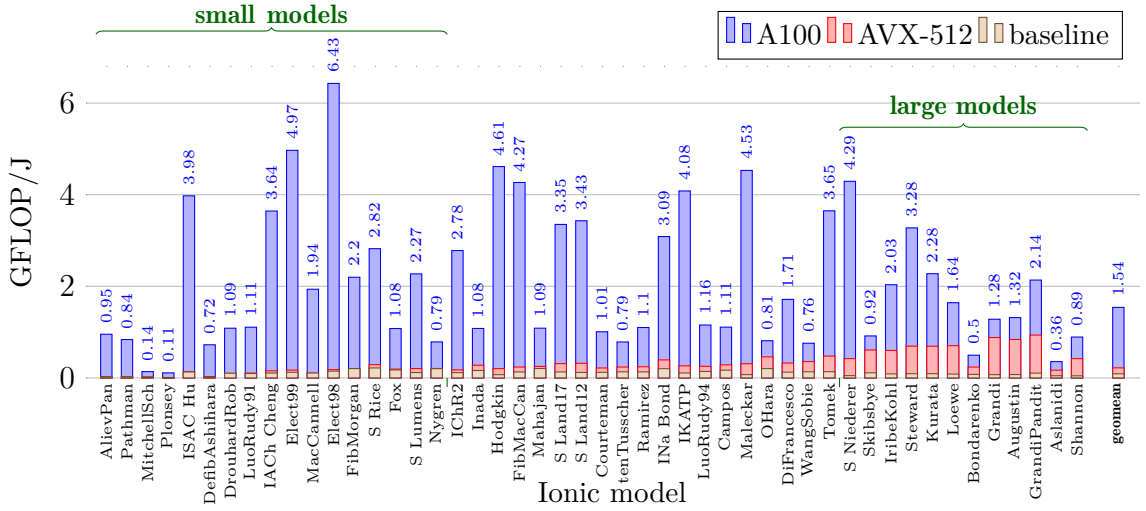


Figure 4.11: Energy efficiency on AMD MI50, in GFLOP per Joule

the Nvidia GPU, we used the `nvidia-smi` command to regularly poll instant power consumption during the kernels execution, and averaged them; a similar command (`rocm-smi`) was used on AMD GPUs.

Figure 4.10 shows the energy efficiency (`y-axis`) of the *limpetMLIR* generated code on the A100 GPU compared to the CPU baseline and vectorized versions, for all 48 ionic models (`x-axis`). A first remark from this figure is that the difference between small, medium, and large models is much less significant than on the previous figures. Only the very small models performing very few floating point operations, and in general the ones that have a small compute-intensity (e.g., as already noticed, Bondarenko and Aslanidi), have a low efficiency on GPU. The numbers reported in this plot pretty closely relate to the compute-intensity of those different benchmarks. For example, ISAC Hu (col. 4) has an intensity of 1.6 flop/byte, while DefibAshihara (col. 5) has only 0.28 flop/byte and is much less power efficient.

The geomean energy gain of the CPU `AVX-512` vectorized version compared to

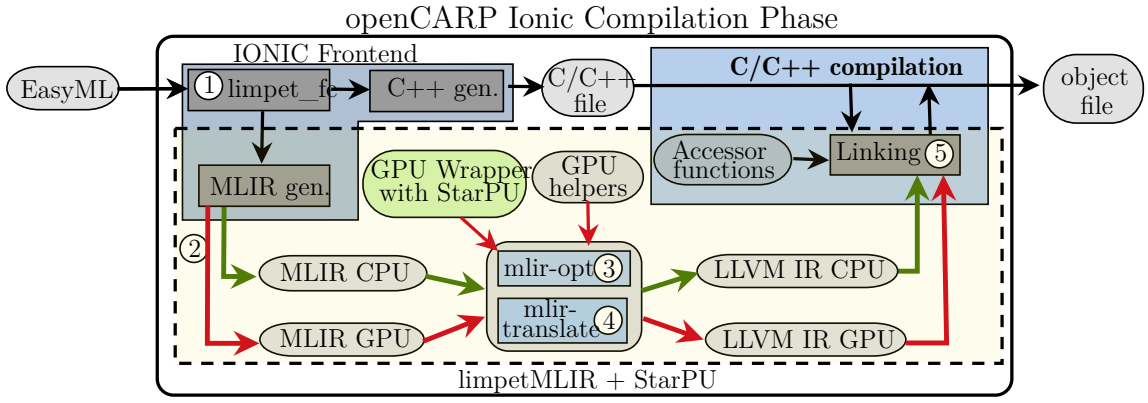


Figure 4.12: *LimpetMLIR* with StarPU GPU wrappers to support load balancing on heterogeneous systems during runtime.

the baseline openCARP CPU version is 2.3×, and it is especially significant on the large models (7.1×): on CPU, the largest benchmarks have the most energy gain when vectorized. On the other hand, for all benchmarks, the efficiency of the GPU is consistently higher than the best CPU version. Considering all ionic models the geomean energy gain of the A100 GPU over the vectorized CPU version is 8.72×.

We performed the same measurements on the AMD MI50 GPU (shown in fig. 4.11) and obtained similar results: as reported before, the MI50 is faster and has also better energy efficiency than the A100 on small models, but worse efficiency on the medium and large ones. The geomean power efficiency of the MI50 is 1.54 GFLOP/J, a bit lower than the A100 1.92 GFLOP/J, but still much better than the vectorized CPU version 0.22 GFLOP/J.

4.3 *LimpetMLIR* Integration with a Task-based Run-time System

The *limpetMLIR* is capable of generating optimized codes targeting different machine architectures. Taking advantage of this, the optimizing capability of *limpetMLIR* can be further improved by integrating it with a task-based runtime systems like StarPU [12]. And this integration will allow *limpetMLIR* for simultaneous exploration of CPU and GPU architectures, so able to run experiments at larger scales.

Our code generation techniques are extended in collaboration with the STORM research team at Inria Bordeaux and we introduced GPU wrappers in *limpetMLIR* to integrate with StarPU. Additionally, the STORM team proposed and implemented two distribution algorithms in StarPU: (i) dynamic load-balancing to distribute the workloads efficiently across nodes and (ii) automatic resource dimensioning to select a good number of computing resources for running this simulation. Figure 4.12 shows the modified *limpetMLIR* with GPU wrappers supporting StarPU integration. They evaluated their implementation with multiple Nvidia GPUs (up to 8 GPUs) and obtained about 13× geo-mean speedups compared to vectorized CPU *limpetMLIR* version. They also

evaluated it using a hybrid approach where a combination of CPUs and Nvidia GPUs are used for computation and obtained geo-mean speedups of 9.97×.

The *limpetMLIR* integration with StarPU is not part of this thesis and more details about integration, algorithms, and evaluations can be found in our collaborative publication [13]. It is very promising to see that our techniques are being used and extended by other research labs.

Chapter 5

A Survey of General-purpose Polyhedral Compilers

In previous chapters, we introduced heterogeneous code generation of parallel loops using the MLIR compiler framework for a cardiac application. The evaluation results show promising performance improvements and hence we tried to generalize those techniques in polyhedral compilation. Polyhedral techniques are effective in optimizing regular loop nests and there are many polyhedral compilers available, with differences in nature: general-purpose, built-in, application-specific, and target-specific compilers. We first made a detailed survey on those available general-purpose polyhedral compilers to find out their advantages and limitations, then generalize our technique concerning those needs.

In this chapter, we discuss the polyhedral model, brief information about different polyhedral compilers, detailed study on general-purpose polyhedral compilers using PolyBench/C benchmark programs and limitations of these general-purpose polyhedral compilers.

5.1 The Polyhedral Model Terminology

In computer science, many compiler optimizations from high level languages to efficient machine dependent code were introduced long years back. With the evolution of distributed, multi-processor, and multi-core architectures many optimization techniques proposed by computer scientists target compute intensive loops. The polyhedral model [43] appeared in the early 1990's, and since then many loop optimization techniques using the polyhedral framework have been proposed.

Polyhedral scheduling originates from the seminal work of Feautrier [44]: the dependence analysis of a SCoP permits to statically determine which computations depend on other ones, and therefore the constraints to be respected by a linear schedule for the transformed program to be valid, *i.e.* to perform the same computations as the

```
1  for (int i = 0; i < N; i++)
2     for (int j = 0; j < N; j++)
3         C[i][j] = C[i][j] * B[i][j];  //S1
4
5  for (int k = 0; k < N; k++)
6     for (int l = 0; l < N; l++)
7         A[k][l] = A[k][l] + C[l][k];  //S2
```

Listing 5.1: Original nested loop code

original one, possibly in a different order. The problem of finding a valid linear schedule reduces to solving a non-linear system of constraints, that can be linearized to an ILP (integer linear programming) with the help of the Farkas lemma, and solved by an ILP solver like PIP [45], the Omega Library [46], FPL [47], or ISL [48] for example.

Several specific objective functions can be added to the ILP system to be solved in order to, *e.g.*, maximize outer- or inner-loop parallelism, increase data locality, or reduce control. One of the key idea to expose such objective functions in the solver was proposed by Bondhugula [49], who first implemented it in the source-to-source Pluto compiler. Polyhedral optimizing compilers usually rely on three main steps:

Step 1: *Raising a static control part (*SCoP*) of a program into its polyhedral representation.* SCoPs are program parts whose control flow can be statically determined at compile-time. They can contain loop nests and conditions, but all bounds and tests must be affine functions of surrounding loop iterators and constant parameters. For precise dependence analysis, they also require the accesses to arrays to be affine functions of the loop iterators and parameters. While some (often *built-in*) polyhedral compilers can automatically detect all SCoPs in any input code, other (usually *source-to-source*) compilers require to manually annotate the regions of the source code to take into account as SCoPs. At the end of this step, a polyhedral representation of a SCoP is a set of program statements, each of them associated to a polyhedron representing the iterations and execution time of those statements. The polyhedral representation incorporate the following information for each loop statement:

Iteration Domain. The iteration domain refers to the set of all points (or iterations) in an iteration space that corresponds to the execution of a particular statement enclosed within loops in a program. They are represented mathematically for loop statements using polyhedra, which are multi-dimensional geometric shapes defined by a set of linear inequalities. The iteration domain can also be a union for example if the input code splits into two control flows then the iteration domain will be the union of disjoint polyhedra. Equation (5.1) shows the iteration domain for the statement S1 of the original code in Listing 5.1. Its coordinate values are taken from induction variables i and j.

$$D_{S1}(N) = \left\{ () \to \begin{pmatrix} i \\ j \end{pmatrix} \in \mathbb{Z}^2 \middle| \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 1 & -1 \\ 0 & -1 & 1 & -1 \end{bmatrix} \begin{bmatrix} i \\ j \\ N \\ 1 \end{bmatrix} \geq \vec{0} \right\} \tag{5.1}$$

Step 2: *Finding an optimizing schedule for this problem.* The central step of a

polyhedral compiler is to find a schedule, that will associate a new execution time to each statement such that (a) the original program semantics is respected and (b) some objective functions are optimized. This step usually relies on finding a valid linear schedule, and applying pre- and post- processing to the problem (*e.g.*, loop fusion/-fission, tiling, parallelism, etc.). It is to be noted that the iteration domain does not provide information about the time to execute loop statements.

Mapping Relations. The mapping relation $f : D \longrightarrow S$ is a linear map, from the iteration domain to a multidimensional time domain. It describes a relation between loop statements and the time at which the statements have to be executed. An identity schedule function refers to the execution of a loop statement in the lexicographical ordering of the original loop statement. If two statements map to the same logical time, they can be executed in any order (or in parallel). There are also other types of mapping functions available: (i) Affine mapping - To represent loop transformations such as skewing, reversal, and interchange; (ii) Parallel mapping - describes the relation of mapping iterations of loop nests to parallel processing units (multi-core or distributed systems). (iii) Spatial mapping - map to represent the spatial parallelism and data locality optimizations (like tiling).

$$\left(\begin{array}{cccccc} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ \hline 1 & 0 & 0 & -1 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 \end{array}\right) \left[\begin{array}{c} i \\ j \\ k \\ l \\ \hline N \\ 1 \end{array}\right] \begin{array}{ccc} \geq & 0 & \\ \geq & 0 & \\ \geq & 0 & \\ \geq & 0 & \\ \hline = & 0 & \#i = l \\ = & 0 & \#j = K \end{array} \tag{5.2}$$

Access Relation. Access relation is used to model memory reference for a given access. It could be an array reference or a statement reference to memory. These relations capture data accesses made by each iteration of the loop nests. Polyhedral techniques perform dependency testing (like the GCD test) on these access relation models to find potential data dependencies between the loop iterations. Then, with the help of dependency testing results it performs dependency analysis and constructs dependency polyhedra to represent data dependencies between two loop statements. Equation (5.2) shows the dependency polyhedra between the statements $S1$ and $S2$ for Listing 5.1. There is a dependency from P and Q if (i) there exists an instance $P[i]$ and $Q[j]$ such they perform read or write to the same memory location and one of them is a write, and (ii) the statement $P[i]$ may happen before $Q[j]$, that means $i \prec j$. There are four dependency types available which are shown in Table 5.1. Input$^{(i)}$ (RAR) is only read and this dependency can be ignored. With the help of standard compiler optimizations (like renaming) anti$^{(a)}$ (WAR) and output$^{(o)}$ (WAW) dependencies can be removed. Only Flow$^{(f)}$ (RAW) is the actual dependency that needs to be taken care of.

Code Transformations. The dependency polyhedra help to identify dependencies of the loop statements between the iterations of the loop nest. Without breaking dependencies, we can apply different transformations to the polyhedral representation that help to apply loop optimizations like tiling, parallel loops, and loop vectorization.

Name	Definition
Input$^{(i)}$	When both $P[i]$ and $Q[i]$ are reading (RAR)
Flow$^{(f)}$	When $P[i]$ writes and $Q[i]$ reads (RAW)
Anti$^{(a)}$	When $P[i]$ reads and $Q[i]$ writes (WAR)
Output$^{(o)}$	When both $P[i]$ and $Q[i]$ are writing (WAW)

Table 5.1: Dependency Types

Step 3: ***Lowering the resulting polyhedral representation into code.*** The last step is to revert to a classical program form, by converting back the polyhedral representation with its associated schedule into code. There are various code generators available, e.g. CLooG [50], ISL [48], or CodeGen [51] which produce an AST from polyhedral representation. The AST takes into account the iteration domain and the schedule generated in the optimizing phase. As said earlier, polyhedral techniques mainly do three loop optimizations: loop tiling, parallelization, and loop vectorization. Tiling is achieved by re-structuring the loop nest. Parallel loops can be expressed by inserting *omp parallel for* OpenMP directives and vectorization loop opportunities can be marked with *vector always* or *ivdep* directives.

Listing 5.2 shows the transformed code by the Pluto [49] polyhedral compiler for the original code shown in Listing 5.1. From the listing, it is visible that the three loop optimizations (tiling, parallelism, and vectorization) discussed above are applied to the original code with the help of polyhedral techniques. The size of the tile is chosen to be 32 by default. Line 4 is the *pragma omp parallel for* annotation to the for loop at line 5, such that the iterations of the loop run in parallel. The loop at line 12 is annotated with vector directives and is a recommendation to the compiler for auto-vectorization.

Over the years, many polyhedral compilers were implemented in different contexts [52, 53, 54, 55, 56, 57]. They have been built on top of different libraries developed by polyhedral research community: PIP [45], Omega [46], PolyLib [58], ISL [48], OpenSCOP [59], CLooG [50], and PET [60]. In the upcoming subsections, we will briefly describe the three categories of available polyhedral compilers, their status and a short brief about each of them.

5.1.1 General-purpose Source-to-source Polyhedral Compilers

These types of polyhedral compilers listed hereafter are open-source and perform source-to-source polyhedral optimizations. Most of these compilers accept `C` code as input in which the nested loop structures are enclosed within `scop` directives. The generated optimized code can be either: `C`, OpenMP parallel `C` code, CUDA, or OpenCL. The emitted code can be compiled into an executable by any standard compiler like `gcc`, `clang`, or `icc`. A brief introduction about the available general-purpose source-to-source compilers is as follows:

Pluto. Pluto [49] is an actively maintained automatic polyhedral source-to-source transformation framework to optimize imperfectly nested loops in regular C programs.

```
if (N >= 1) {
  lbp=0;
  ubp=floord(N-1,32);
#pragma omp parallel for private(lbv,ubv,t2,t3,t4,t5)
  for (t1=lbp;t1<=ubp;t1++) {
    for (t2=0;t2<=floord(N-1,32);t2++) {
      for (t3=32*t1;t3<=min(N-1,32*t1+31);t3++) {
        lbv=32*t2;
        ubv=min(N-1,32*t2+31);
#pragma ivdep
#pragma vector always
        for (t4=lbv;t4<=ubv;t4++) {
          C[t3][t4] = C[t3][t4] * B[t3][t4];;
          A[i][j] = A[i][j] + C[j][t4];;
        }
      }
    }
  }
}
```

Listing 5.2: Transformed code (by the Pluto polyhedral compiler) of original C code in listing 5.1

With the help of affine transformations Pluto finds an efficient way to tile parallel loop(s) for parallelism and data locality. Pluto extracts polyhedral information with the help of PET [60] or clan [61], uses the Integer Set Library (ISL) [48] or candl [62] as a dependency tester and uses ISL [48] or PIP [45] as an ILP solver. After applying its own tiling optimization technique it generates code using CLooG [50]. It also provides options to control the loop fusion heuristic. Different command line options are available to choose different tiling options (standard tile, two level tile, and diamond tile), ILP solver, dependency tester, and many others. The diamond tile option supports load-balanced tile execution across cores for stencil computations.

PoCC. PoCC [63] is a collection of different source-to-source translators, tools and other libraries to provide support for polyhedral optimization. It includes Pluto, and the Legal transformation Space explorator (LetSee) permits iterative compilation using different schedules. PoCC provides multiple options to try various combinations of polyhedral extractors, dependency testers, ILP solvers, schedulers and code generators. It does also include POnOS [64] - a loop optimizer with the help of convex formulation. PoCC is maintained by the developer and provides supports to users on how to use the compiler effectively.

ROSE/PolyOpt. `ROSE` [54] is an open source compiler infrastructure developed at Lawrence Livermore National Laboratory (LLNL). PolyOpt [65] is a polyhedral loop optimization framework, integrated within the ROSE compiler. It can auto extract the SCoPs that can be optimized using the polyhedral model based on PoCC [55] (for dependency analysis, code generation, parametric tiling and others). It uses Pluto available in PoCC to perform tiling. It is maintained along with the `ROSE` compiler.

PPCG. Polyhedral Parallel Code Generator (PPCG) [57] is a source-to-source compiler generating C, OpenCL and CUDA GPU code from the input program. PPCG is the only polyhedral compiler in this category to generate all 4 different types of optimized code. It takes C programs as input, and (i) extracts a polyhedral representation with the help of the Polyhedral Extraction Tool (PET) [60], (ii) does dependency testing using ISL, (iii) exposes tiling opportunities with the same algorithm as Pluto, (iv) de-

cides the host and GPU code, including memory management, and (v) finally generates C/OpenMP, OpenCL or CUDA code with its own code generator. PPCG is maintained by the ISL development team and provides support to report bugs, feature requests, and user questions.

CHiLL. Composable High-Level Loop (CHiLL) [56] is a source-to-source compiler to improve the performance of nested loop calculations. It uses the `ROSE` compiler to parse the input program. Contrarily to the above polyhedral compilers it is not fully automatic: the user has to manually write a python script which specifies the loops and transformations to be applied. There is also a CUDA version of CHiLL.

TRACO. TRACO [66] is a source-to-source compiler to increase program locality and generate parallel tiled code on nested loop sequences, via polyhedral techniques and Iteration Space Slicing [67]. TRACO uses the transitive closure of loop nest dependency graph to perform tiling, so that all loop dependencies are preserved. It uses the CLooG code generator to emit the final parallel-tiled code.

5.1.2 General-purpose Built-in Polyhedral Compilers

The next set of polyhedral compilers are the ones that are built-in within a compiler. They are integrated within the compiler and are invoked like an optimization pass. Unlike the source-to-source compilers which would require the user to enclose the loop nest within `scop`, the built-in compilers can automatically detect the `scop` from the input code and apply polyhedral transformations. We have very few general-purpose built-in compilers that are integrated within either `gcc` or `clang`. They are as follows:

Gcc/GRAPHITE. GRAPHITE [52] is the first research project which has implemented the polyhedral techniques into a widely used compiler, `gcc` [68]. It follows a three step approach: (i) extract polyhedral information from GIMPLE (the IR of `gcc`), (ii) apply polyhedral optimizations using ISL, and (iii) finally generate code using CLooG. `Gcc` has the Polyhedral Compilation Package (PCP) [69] interface to interact with GRAPHITE. It is maintained along with the `gcc` compiler.

LLVM/Polly. Polly [53] is an actively maintained project from the LLVM [8] compiler infrastructure which proved itself an effective tool to apply the benefits of polyhedral optimizations at the intermediate representation (IR) level. Similarly to `gcc`/GRAPHITE, Polly uses a three step approach: it takes LLVM-IR as the input and (i) identifies the static control parts (SCoPs) of interest, using $\mathbb{Z}$-polyhedra provided by ISL for representing polyhedral domains; (ii) applies inbuilt polyhedral optimizations, and also provides support to export the polyhedral representation, call an external polyhedral compiler and import it back; (iii) generates a generic AST from the optimized polyhedral representation with the help of CLooG, and the final optimized LLVM-IR is created by merging it with the original AST.

MLIR/Polygeist. Polygeist raises C/C++ to Polyhedral MLIR [15] as an LLVM/MLIR [7] based approach. It is introduced recently and in active development phase. Polygeist takes an input source program, and (i) extract the high level information,

(ii) converts them to MLIR representation, (iii) extracts polyhedral representation, and allows a call to an external polyhedral optimizer like Pluto, and (iv) finally converts it to LLVM IR. As it operates from high-level structure to low-level IR of the program, it has the opportunity to explore optimizations at both levels. Polygeist also supports additional transformations like statement splitting and reduction parallelization.

5.1.3 Application-specific and Target-specific Compilers

These set of compilers are either application-specific or target-specific, e.g. Tensor Comprehensions is an application-specific polyhedral compiler mainly focusing on optimizing deep learning-related applications. Many of these compilers do not accept the standard programming languages as input, they accept DSL as an input or the code should be converted to the DSL representation. Most of these compilers are developed by technology-supported companies and are either in-house (specifically developed for their company products) or commercial. Those compilers are as follows:

IMB XL. The IBM XL C/C++ compilers contain an internal polyhedral pass using a Pluto-like scheduling algorithm. This suite of compilers targets only IBM specific systems and processors.

R-Stream [discontinued]. R-Stream [70] is a source-to-source compiler from Reservoir Labs to optimize loop nests manipulating dense matrices and arrays in high-performance scientific and embedded computing. It accepts C programs as input and generates generalized dependence graph (GDG) - a kind of polyhedral representation to model the computations, dependences and memory references of loop nests. It performs parallel-tile optimizations on the nested loops using a scheduling algorithm inspired from Pluto, and generates parallel code for multi-cores and various accelerators (including GPUs and FPGAs). The R-Stream team joined Qualcomm and they now develop the Qualcomm polyhedral mapper, an iterative hierarchical scheduler specialized to target AI applications. It enables code generation for the specific Qualcomm Hexagon processor, exhibiting parallelism at multiple levels: nodes distribution, multi-thread, and SIMD units.

Tensor Comprehensions. Facebook (Meta) develops Tensor Comprehensions [71, 72], a C++ library and a domain-specific language for deep learning models. Their polyhedral just-in-time compiler based on PPCG targets CUDA kernels, and includes operator fusion and size specialization. Unfortunately, the project is now in defunct stage.

Huawei's MindSpore/AKG. MindSpore [73] is a deep learning framework developed by Huawei, targeting their Ascend processor, Nvidia GPUs and x86 CPUs. AKG (Auto Kernel Generator) [74] includes a polyhedral scheduler that permits to fuse AI operators, tile, parallelize, and vectorize the resulting code.

Cerebras. Cerebras develops a target specific polyhedral compiler [75] for AI applications to generate efficient multi-level SIMD instructions for their CS-1 architecture.

Tiramisu. Tiramisu [76, 77] is a compiler framework using polyhedral techniques to optimize complex loop patterns for dense and sparse deep learning model (RNN, CNN)

applications and data parallel algorithms. They target optimized code generation for heterogeneous architectures.

PolyMage. PolyMage labs [78] has developed PolyBlocks which is used to build compilers and optimized code generators for high performance computing targeting the domain of Artificial Intelligence and Machine Learning. PolyBlocks is built using the MLIR infrastructure and it performs efficient computations for high-dimensional data space application with the help of polyhedral optimization techniques.

APOLLO. Automatic speculative POLyhedral Loop Optimizer [79] is based on the LLVM compiler and performs complex loop optimizations with the help of polyhedral techniques at run-time. APOLLO's run-time system (i) profiles a few iterations of loop chunks, (ii) identifies a linear prediction model, (iii) encodes the model in its polyhedral representation, and finally (iv) performs loop transformations with the help of Pluto. The prediction correctness is checked at runtime and in case it is wrong, it rollbacks to the original code.

Intel's PlaidML. PlaidML is an open-source tensor compiler developed and supported by Intel for deep learning applications. They introduce a domain-specific IR named Stripe [80] to represent tensor operations and generate efficient machine learning kernels using the nested polyhedral model.

5.1.4 A Study on Polyhedral Compilers

With the difference in nature of polyhedral compilers, a detailed study on them would help the polyhedral community to spread their research. Additionally, it will help to identify the existing limitations in the polyhedral compilers and provides the path for further improvements.

There are many [81, 82, 83, 84, 85, 86, 87, 88, 89, 90] surveys and empirical studies available on compiler optimizations, but only very few of them discuss polyhedral loop optimizations. Schneck [86] did a survey on the compiler optimizations and it is considered as one of the earliest such survey work. Mustafa [91] did a survey on parallel applications, in which he conducted a brief study on polyhedral compilers. The survey lists the available polyhedral compilers and only compares performance of Cetus [92] against Pluto. Simbürger et al. [93] did an empirical study on polyhedral optimization on various applications. The authors took many real-world applications and studied the benefits of polyhedral optimization (at run-time and compile-time) using Polly on those applications. Pfaffe et al. [94] did a detailed study on the *gemm* benchmark running on GPU. The authors used Polly, combined with their tuner to exploit the benefits of polyhedral optimization. They did performance tests on PolyBench/C. Fontaine et al. [95] did a case study on polyhedral dataflow programming, combining dataflow programming with polyhedral compilation in order to exploit the parallelism in loop nests. They used Pluto for polyhedral optimization and the $\sum$C dataflow language for their case study. All of these works have a range of evaluation limited to one or two polyhedral compiler, while our survey compares most of them.

In Section 5.3, a very detailed study on the general-purpose polyhedral compilers using PolyBench/C [14] set of benchmarks is presented. An intensive evaluation using Intel and AMD machines, detailed performance analysis with the help of hardware counters, and profiling the source-to-source optimized code for finding the degree of parallelismt that has been made. We listed the limitations with the current polyhedral compilers and mentioned the need for new set of benchmarks for testing AI/ML related polyhedral compilers.

Furthermore, we try to address one of the listed limitations in the polyhedral compilers. For polyhedral vector optimizations, the loops are marked with *vector always* and *ivdep* directives. This latter directive is merely a recommendation for the compilers to vectorize the upcoming loop structure but we observed that in few cases standard compilers do not actually vectorize it. The study found that the main reasons reported by compilers to not vectorize the marked code were: (i) could not determine the number of loop iterations, (ii) complex loop statements (like function calls, irregular control flow, and pointer/array deference), (iii) multiple nested loops, (iv) control flow inside loops, and (v) inner-loop count not invariant. Further, in a few cases the vectorization is not applied because the vector cost-model estimates that it is not profitable. Thus, we are not able to fully reap the benefits of polyhedral techniques. Another aspect is the heterogeneous code generation, only PPCG (in general-purpose compiler) is able to generate CUDA targeting only Nvidia machines.

In Chapter 6, we introduce an optimization technique with the help of MLIR and Polygeist to enhance the vectorization opportunities in polyhedral techniques. Similarly, we propose an heterogeneous compilation flow in that a unified approach is followed to emit one abstraction of code for CPU/GPU from the polyhedral representation and with the help of MLIR the code is lowered to target either CPU vectorized machine or Nvidia GPU machine or AMD GPU machine, respectively.

5.2 Benchmark Suite

We chose PolyBench/C [14] amongst a wide range of loop nests benchmarks: NAS Parallel Benchmarks [96], UTDSP Benchmarks [97], Livermoore Benchmarks [98], and many others. PolyBench/C is the most widely recognized benchmark in the polyhedral research community.

It comprises a set of thirty numerical benchmarks targeting various real application domains, available in both `C` and `Fortran` languages. Table 5.2 from the `README` of PolyBench/C [14] shows the thirty available benchmark programs. PolyBench/C supports various features like five different problem sizes, options to flush the cache, and dump-out the benchmark results to verify their correctness. It also provides various options to the user like timing/profiling options (such as PAPI [99] support), stack or heap memory allocation, array padding, and using the C99 standard. We used the following options for our evaluations:

- `POLYBENCH_TIME` - to measure the kernel execution time,

Benchmark	Description
	linear-algebra/kernels
2mm	2 Matrix Multiplications (alpha * A * B * C + beta * D)
3mm	3 Matrix Multiplications ((A*B)*(C*D))
atax	Matrix Transpose and Vector Multiplication
bicg	BiCG Sub Kernel of BiCGStab Linear Solver
doitgen	Multi-resolution analysis kernel (MADNESS)
mvt	Matrix Vector Product and Transpose
	linear-algebra/blas
gemm	Matrix-multiply C=alpha.A.B+beta.C
gemver	Vector Multiplication and Matrix Addition
gesummv	Scalar, Vector and Matrix Multiplication
symm	Symmetric matrix-multiply
syr2k	Symmetric rank-2k update
syrk	Symmetric rank-k update
trmm	Triangular matrix-multiply
	linear-algebra/solvers
cholesky	Cholesky Decomposition
durbin	Toeplitz system solver
gramschmidt	Gram-Schmidt decomposition
lu	LU decomposition
ludcmp	LU decomposition followed by Forward Substitution
trisolv	Triangular solver
	datamining
correlation	Correlation Computation
covariance	Covariance Computation
	medley
deriche	Edge detection filter
floyd-wars	Computes shortest path in a graph data structure
nussinov	Dynamic programming algorithm for sequence alignment
	stencils
adi	Alternating Direction Implicit solver
fdtd-2d	2-D Finite Different Time Domain Kernel
head-3d	Heat equation over 3D data domain
jacobi-1D	1-D Jacobi stencil computation
jacobi-2D	2-D Jacobi stencil computation
seidel	2-D Seidel stencil computation

Table 5.2: Thirty different numerical computations programs from different real world problem in Polybench 4.2 suite (from the PolyBench 4.2 documentation).

- `EXTRALARGE_DATASET` - we chose the maximum available input data-size for our experiments,
- `POLYBENCH_DUMP_ARRAYS` - to emit the output of benchmarks,
- `POLYBENCH_USE_SCALAR_LB` - to use scalar loop bounds (not parametric bounds),
- `POLYBENCH_USE_RESTRICT` - to tell the compiler that no aliasing occurs between arrays,
- `POLYBENCH_USE_C99_PROTO` - to use standard `C99` function prototypes.

The thirty benchmarks are divided into six categories based on their nature: kernels (linear-algebra), blas (linear-algebra), solvers (linear-algebra), datamining, medley, and stencils. In the further sections, we present our results and analysis with respect to those categories as they have similar characteristics within their group.

5.3 Evaluations

Our evaluation regarding performance, mostly expressed as speedups relative to the native code of the eight available general-purpose polyhedral compilers on the PolyBench/C benchmark is done from several view points.

First, we evaluate the overall performance of each polyhedral compiler, averaged over the thirty programs of PolyBench/C. This evaluation is done for both sequential and parallel generated codes, on an Intel processor and an AMD processor (Section 5.3.2). Second, we present (Section 5.3.3) the detailed performance for each of the six categories of PolyBench/C. Third, we choose one representative benchmark in each category to better understand its behavior. For that purpose, we provide an analysis based on hardware counter values (Section 5.4). Fourth, as Pluto provides different tiling options, we present the impact of each in a separate section (Section 5.5). Finally, we report the performance for PPCG's CUDA and OpenCL code on GPU (Section. 5.6).

5.3.1 Experimental Setup and Method

In this section, we present the hardware setup and the polyhedral compilers tested in our experiments. We used:

- a 2x20-cores CascadeLake Intel Xeon Gold 5218R @2.1GHz CPU,
- a 1x48-cores Zen 2 AMD EPYC 7642 @2.3 GHz CPU,
- an A100 Nvidia GPU.

Compiler	Version	Option(s)
Polly	LLVM 15.0.2	--polly, --polly-parallel
Polygeist	LLVM 13.0.2[1]	--demote-loop-reduction, --extract-scop-stmt, --canonicalize, --pluto-opt='parallelize=1', -reg2mem, -insert-redundant-load
GRAPHITE	GCC 12.1.1	--fgraphite, --floop-parallelize-all
PolyOpt	0.11.100	--polyopt-fixed-tiling, --polyopt-scalar-privatization, --polyopt-safe-math-func
Pluto	0.11.4	--tile/--l2tile/--diamond-tile, --parallel, --smartfuse, --prevector
PoCC	1.6.0	--pluto-tile, --pluto-parallel, --pragmatizer, --vectorizer, --pluto-scalpriv, --pluto-fuse smartfuse
PPCG	0.09.1	--tile, --target=c (or cuda, or opencl), --openmp
CHiLL	7.4	--tile, --loop-split, --loop-fusion

[1] Polygeist uses an older version of LLVM than Polly in its distribution

Table 5.3: Polyhedral compilers with their versions and tested options

Both CPUs turbo boost and hyper-threading were disabled. We carried out the experiments with eight polyhedral compilers (GRAPHITE, Polly, Polygeist, PolyOpt, Pluto, PoCC, PPCG, and CHiLL). There were some challenges, especially while installing TRACO as the code is pretty old and facing run-time errors while applying tiling. We contacted most of the developers/maintainers of those eight polyhedral compilers and applied the right flag options for our experiments, such that the polyhedral compilers perform best against the benchmarks. The detailed version of the compilers and the optimization flags we used are shown in Table 5.3.

For Pluto the benchmarks are compiled in three flavours using the `--tile` (called *Pluto-tile* in the following), `--l2tile` (*Pluto-l2tile)*, and `--diamond-tile` (*Pluto-diatile)* along with the other options; for PPCG we used options to generate code for `C`, `CUDA` and `OpenCL`. For CHiLL we used `--tile` (along with fusion for a few benchmarks, see below). PolyOpt's `-fixed-tiling` is a combination of fuse, tile, parallel, and prevector from Pluto. We did our experiments for both sequential and parallel versions, and for sequential runs we excluded the compiler options which emits parallel code. We used `gcc` (12.1.1), `clang` (15.0.2), and `icc` (2021.4.0) to compile the polyhedral optimized code from source-to-source translators (Pluto, PPCG, PoCC, and CHiLL), and the `rose` compiler (0.11.100) for PolyOpt.

We evaluated all the thirty benchmark programs available in PolyBench/C against the available eight polyhedral compilers, along with aggressive compiler optimizations (`-O3 -march=native -mprefer-vector-width=256 -ffast-math`). We fixed the vector size to 256 to select the AVX2 instruction set and activated fast-math, for all compilers to be evaluated fairly: by default the -O3 optimization level does not select them in the same manner. We chose the highest problem size (`EXTRALARGE`) from PolyBench/C. Execution times were measured by running each benchmark five times, eliminating the two extremes, and averaging the remaining three.

Errors. We came across errors which prevented to get results for some experiments. PolyOpt throws an assertion error during affine conversion for *cholesky* and «fails to find hyper-planes» for *seidel-2d*. Pluto fails to extract polyhedra for the *adi* benchmark. We also had to stop Pluto during its source-to-source compilation of *heat-3d* with diamond tile option activated after two hours of execution without a result. PoCC throws a candl error for *ludcmp, deriche,* and *nussinov* and a clan error for *adi*. The PPCG emitted CUDA code of *nussinov* fails to compile. Some kernels pose difficulties regarding the data dependency analysis and no parallel code is generated. This is the case for *cholesky* (Polygeist), *durbin* (PolyOpt, Pluto-tile), *ludcmp* (Polygeist, Pluto-tile, PPCG), *trisolv* (Polygeist), *deriche* (Polygeist), *nussinov* (Polygeist, PolyOpt). CHiLL generates codes that suffers segmentation faults at run-time for *gramschmidt, ludcmp* and *nussinov*. CHiLL also refuses to fuse two loops inside a loop in all benchmarks. It cannot apply tiling for the inner loop statement as it finds a dependence violation (*fdtd-2d*, *heat-3d*, *jacobi-1d/2d*, and *seidel-2d*). For many benchmarks it cannot apply fusion and tiling together. For some benchmarks with tiling activated, CHiLL fails to calculate the loop upper bound value, warns the user and stops. Table 5.4 summarizes errors and warnings which occurred during the translation, compilation, and execution of PolyBench/C using the polyhedral compilers.

Output Verification. We used the `POLYBENCH_DUMP_ARRAYS` option from PolyBench/C and dumped all output to files for both sequential and parallel executions, and compared them to their reference standard compiler. For example, we compared the output of Polly with the output of `clang`, and similarly the Pluto (-tile) `gcc` output with the output of the original code using `gcc`. Because of the aggressive optimization options (`-ffast-math`) the output of a few benchmarks differ slightly in precision, and we did not consider those as failures.

Polly generates wrong outputs for *gemver* and *trmm* (in parallel only). PolyOpt emits wrong outputs for *symm*, *ludcmp*, and *deriche*. PoCC also emits wrong outputs for *symm* and *durbin*. Polygeist generates wrong outputs for the parallel execution of *lu*. CHiLL generated wrong results for seven benchmarks (*atax*, *doitgen*, *gesummv*, *symm*, *durbin*, *adi*, and *fdtd-2d*).

Considering errors and output verification Pluto, Polly, and Polygeist fail in one benchmark (*adi*, *gemver*, and *lu* respectively). PolyOpt and PoCC fail in 5 benchmarks whereas CHiLL fails in 10 benchmarks. The other compilers (GRAPHITE and PPCG) generates valid output for all 30 benchmarks in PolyBench/C.

Methodology for Comparison. In the rest of the chapter, we evaluate several performance metrics for both built-in and source-to-source compilers using the following baseline.

For built-in optimizing-compilers, we compare the performance of the binary produced by the optimizer against the binary produced by the optimizer's respective standard compiler: a binary compiled by Polly is compared against the one generated by `clang`, and GRAPHITE against `gcc`.

Benchm.	PolyOpt	Plulto-tile	Pluto-l2tile	Pluto-dia-tile	PoCC	PPCG	Polygeist	CHiLL
symm	-	Failed to parallelize	-	-	-	-	-	
cholesky	Assertion error during affine conversion	-	-	-	-	-	Failed to parallelize	-
durbin	Failed to parallelize	Failed to parallelize	-	-	-	-	-	
gramschmidt-		-	-	-	-	-	-	Segmentation error
ludcmp	-	Failed to parallelize	-	-	Candl error: schedule is not identity 2d+1 shaped	Failed to parallelize	Failed to parallelize	Segmentation error
trisolv	-	-	-	-	-	-	Failed to parallelize	-
correlation	-	-	-	-	-	Segmentation violation: parallel execution in `icc` compiler	-	-
deriche	-	-	-	-	Candl error: schedule is not identity 2d+1 shaped	-	Failed to parallelize	-
nussinov	Failed to parallelize	-	-	-	Candl error: schedule is not identity 2d+1 shaped	Segmentation violation: parallel execution in `icc` compiler. CUDA compiler error.	Failed to parallelize	Segmentation error
fwarshall	-	-	-	-	-	Failed to parallelize	-	
adi	Failed to parallelize	Error while extracting polyhedra	Error while extracting polyhedra	Error while extracting polyhedra	Clan error	-	-	-
jacobi-1d/2d	-	-	-	-	-	Failed to parallelize	-	-
seidel-2d	PLuTo cannot find any more hyperplanes	-	-	-	-	Failed to parallelize	-	-
heat-3d	-	-	-	Time-Out: `C` file not emitted in 2 hrs	-	Segmentation violation: parallel execution in `icc`	-	-

Table 5.4: Compile-time errors, run-time errors, warnings, and parallelization issues of PolyBench/C with respect to different polyhedral compilers.

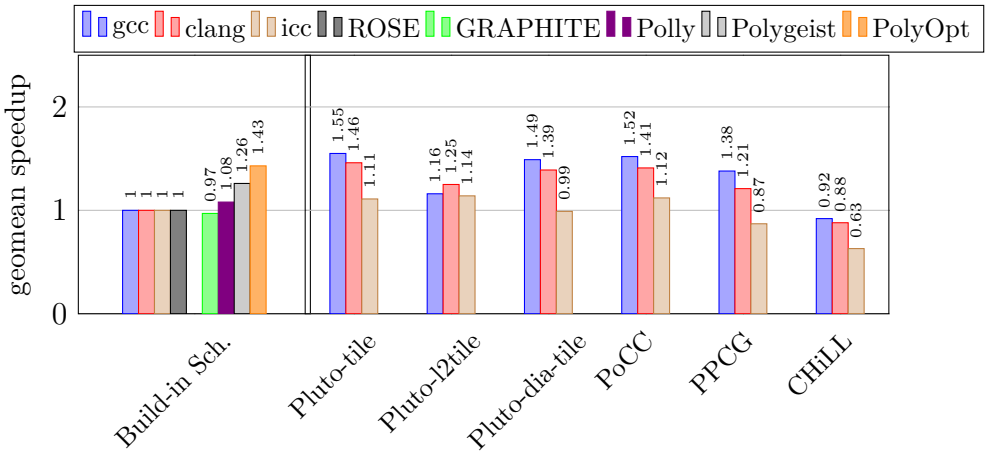


Figure 5.1: Geomean **sequential** execution speedups on PolyBench/C on an Intel CPU

For source-to-source polyhedral compilers, once the native source code is transformed into an optimized source code, it is compiled with `gcc`, `clang` and `icc`. Their three executions are then compared to the ones obtained by compiling the original source code with these respective compilers. When ambiguous, we name a compiled transformed source after the scheduler and compiler used to produce it, for example `PoCC+gcc` or `PoCC+clang`. For PolyOpt, the optimized code is compiled and compared only with `ROSE`.

5.3.2 Overall Performance of Sequential and Parallel Codes

Sequential. Figures 5.1 and 5.2 show the sequential geomean speedups of polyhedral optimized codes on Intel and AMD processors respectively of all benchmarks in PolyBench/C. The left box groups the results for built-in polyhedral compilers (GRAPHITE, Polly, Polygeist), and PolyOpt, while the right box presents results for source-to-source polyhedral compilers, for each binary compiled with `gcc`, `clang` and `icc` respectively. The left box also contain bars with value set to 1 for the standard compilers (`gcc`, `clang`, `icc` and `ROSE`) as a reference. For errors, wrong results and time-outs happening using polyhedral compilers, we have replaced their results with the execution time of their respective baseline compiler (yielding a speedup of 1 for benchmarks with an error).

From the figures, we can see that Polly, Polygeist and PolyOpt perform better than their baseline compiler on both our Intel (1.08×, 1.26×, and 1.43× resp.) and AMD (1.27×, 1.51×, and 1.47× resp.) machines. On the contrary, GRAPHITE binaries are slower than those produced by `gcc` alone with speedups of 0.97× on Intel and 0.95× on AMD.

Among source-to-source compilers, PLuTo's and PoCC's optimized codes perform significantly better using `gcc` and `clang`. When using `icc` we see a modest gain on the Intel processor and a slight slowdown on AMD. `PPCG+gcc` and `PPCG+clang` perform better on both processor types, but `PPCG+icc` is always slower than the native source compiled with `icc`. CHiLL performs on average worse using all three compilers.

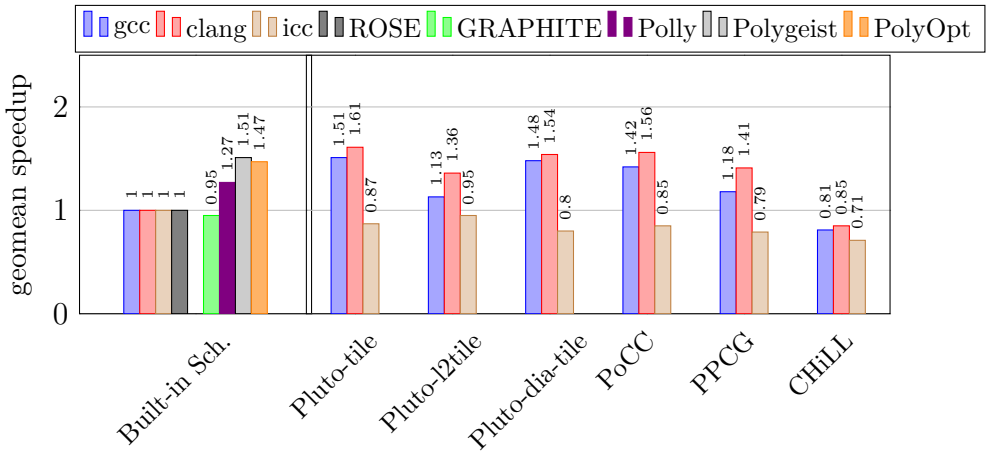


Figure 5.2: Geomean **sequential** execution speedups on PolyBench/C on AMD Machine.

Parallel. Figures 5.3 and 5.4 show the parallel geomean speedups of 21 benchmarks in PolyBench/C against polyhedral compilers. A few benchmarks have very short execution time and are not suitable for parallelization. So, we considered only 21 benchmarks[2] with an execution time greater than 1.5 seconds using the `gcc`-compiled sequential version, for inclusion in the geomean speedup calculation reported here. CHiLL does not provide support for automatic parallel code generation and therefore is not included in those figures. We observed that Polly (geomean 5.5×), PolyOpt, PoCC, and Pluto (-tile with `gcc` geomean: 15×) are quite efficient in parallelizing this set of codes with geomean speedups of 4.6× to 18× on both our platforms (40-cores Intel and 48-cores AMD).

We report speedups with respect to the *baseline sequential* code, as the end-user is mainly interested in the overall gain obtained from its original code. By averaging the speedups we get a global picture of the performance one can expect from these automatic polyhedral compilers. However, if we express the speedups listed in those figures as efficiency (the ratio of speedup to number of cores), they range from a modest 0.35 to a problematic 0.05, regardless of the test platform (AMD or Intel). Even without computing a reference to the machine best reachable performance, some benchmarks obviously pose serious issues. As we will detail later (Section 5.4.2) these averages conceal extremely varied performance depending on the benchmark.

5.3.3 Performance per Category

We now refine the observations with a breakdown of the performance averaged by category. Each figure in this section presents both sequential and parallel versions, using `gcc`, `clang`, `icc`, and `ROSE` when applicable. We did not report CHiLL since it fails to optimize many of these benchmarks and is not fully automatic. Like in the previous sub-section, we considered only 21 suitable benchmarks for parallel execution.

[2]The excluded benchmarks are *atax*, *bicg*, *doitgen*, *mvt*, *gemver*, *gesummv*, *durbin*, *trisolv*, and *jacobi-1d*.

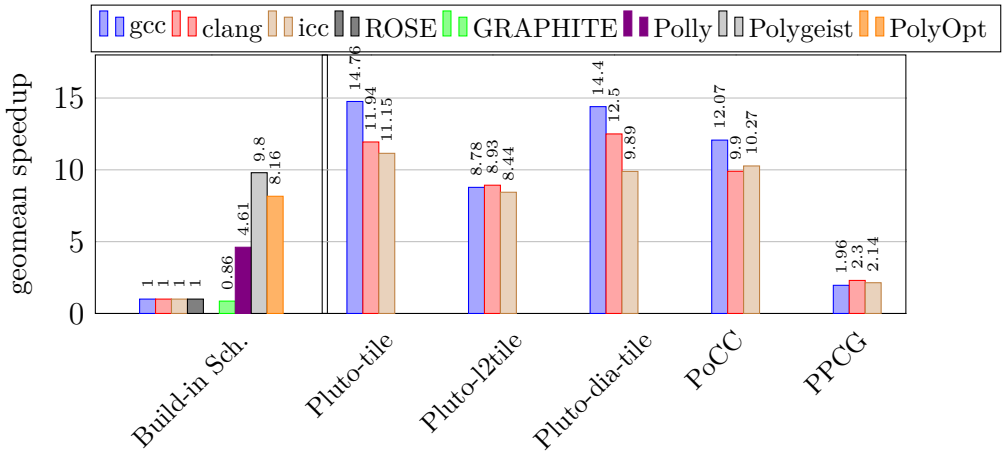


Figure 5.3: Geomean **parallel** execution speedups on 21 PolyBench/C on an Intel CPU (40 cores)

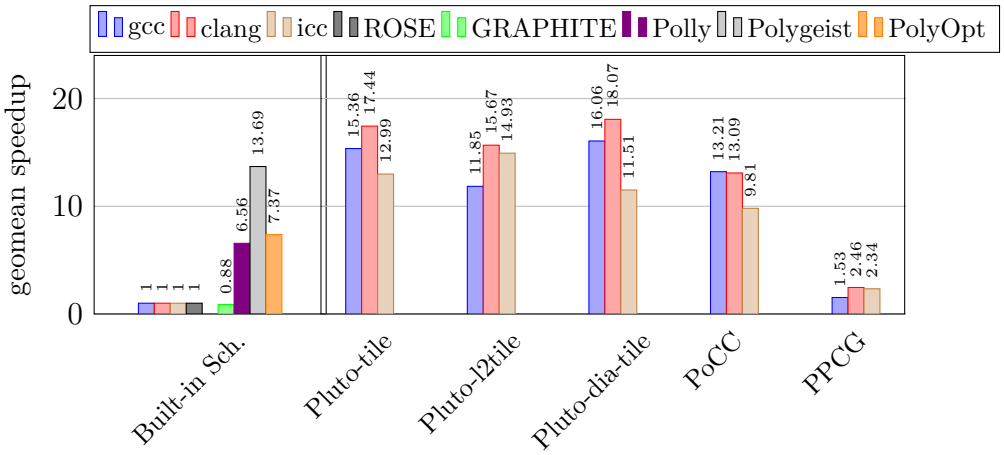
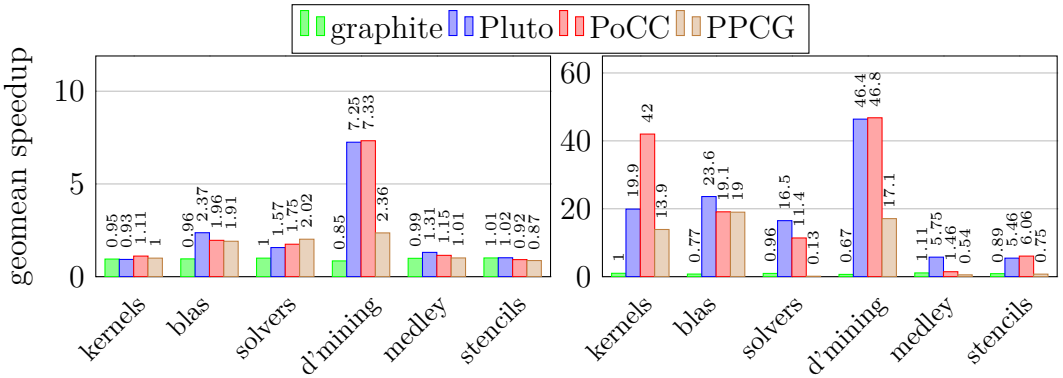
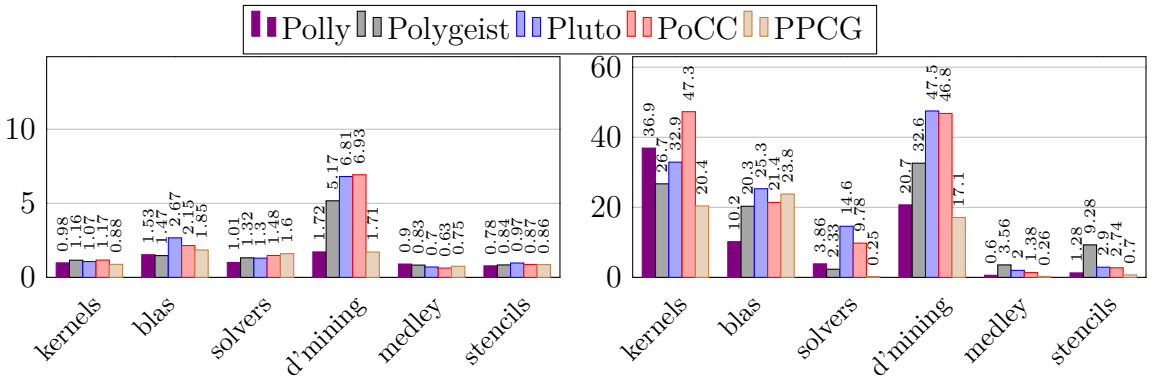


Figure 5.4: Geomean **parallel** execution speedups on 21 PolyBench/C on AMD Machine (48 cores).



(a) sequential (b) parallel

Figure 5.5: Geomean speedups for GRAPHITE, Pluto (-tile), PoCC, and PPCG using `gcc`. On the `x-axis` are the six categories of benchmarks in PolyBench/C.



(a) sequential (b) parallel

Figure 5.6: Geomean speedups of Polly, Polygeist, Pluto (-tile), PoCC, and PPCG using `clang`

Gcc. Figure 5.5 shows the sequential and parallel speedups obtained by the polyhedral compilers (GRAPHITE, Pluto (-tile), PoCC, and PPCG) using `gcc` for the six PolyBench/C categories. GRAPHITE cannot effectively optimize any set of benchmarks compared to the baseline `gcc` (speedups: 0.67×-1.11×). Pluto-tile and PoCC produce effective optimizations for the *blas*, *solvers*, *datamining*, and *medley* categories. They fail to optimize *kernels* and *stencils* in sequential execution, but they report speedups (5×-42×) in parallel[3]. PPCG reports speedups for *kernels* (parallel), *blas*, *solvers* (sequential), and *datamining*. PPCG adds overheads[4] to the parallelized code and shows a lower performance using `gcc` for *solvers*, *medley*, and *stencils*.

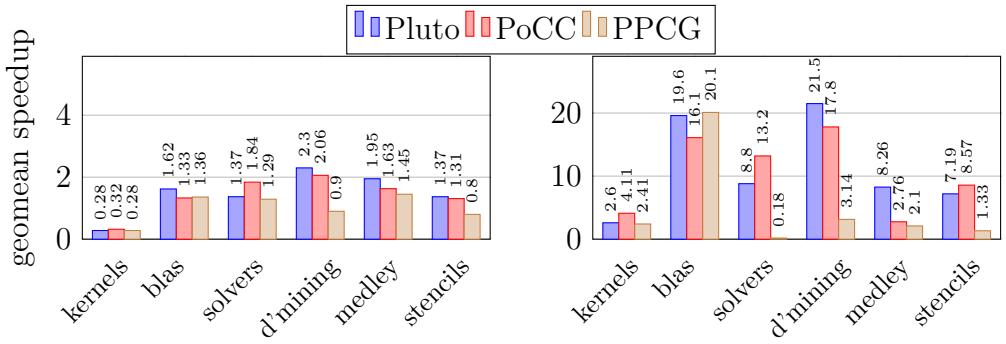
Clang. Figure 5.6 shows the speedups with `clang` in the same conditions as above. Polly and Polygeist report speedups in sequential for *kernels*, *blas*, *solvers*, and *datamining*, but fails for the other two categories. Except for the *medley* category, Polly has a good parallelization impact. Polygeist failed to parallelize a few loops in the *solvers* benchmarks (as mentioned in section 5.3.1 errors) and that is the reason for low speedups. Pluto-tile and PoCC fail to optimize *medley* and *stencils* in sequential, but for all other categories and in parallel they perform good. PPCG sequential optimized code fails to perform better for the *kernels*, *medley*, and *stencils* categories. Similarly, PPCG's parallel codes show slowdowns for *solvers*, *medley*, and *stencils*.

Icc. Figure 5.7 shows the speedups using `icc` in the same conditions as above. The baseline `icc` compiler is very efficient in optimizing the *kernels* category benchmarks, leaving no space for optimization. The polyhedral compilers transform those *kernels* codes, that are no longer recognized by `icc` and eventually perform worse than the baseline in sequential. They report only small speedups in parallel. For all the other categories Pluto-tile and PoCC improve performance. PPCG underperforms the baseline for *datamining* (sequential), *stencils* (sequential), and *solvers* (parallel).

ROSE. Figure 5.8 shows the sequential and parallel speedups obtained by PolyOpt

[3]Benchmarks *atax*, *bicg*, *doitgen*, *mvt* from kernels and *jacobi-1d* from stencils are excluded in speedup calculation.

[4]Due to more thread creation/synchronization or inefficient usage of processor cores, as discussed below in section 5.4.2.



(a) sequential (b) parallel

Figure 5.7: Geomean speedups of Pluto (-tile), PoCC, and PPCG using `icc`

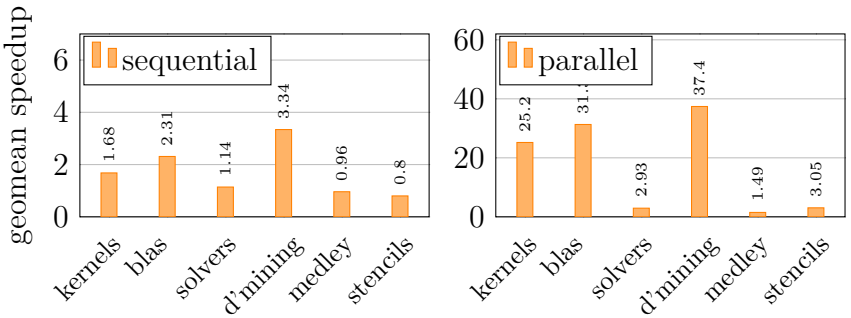


Figure 5.8: Geomean speedups of ROSE/PolyOpt compared to the baseline `ROSE` compiler

compared to `ROSE`. PolyOpt reports speedups for most of the categories excluding *medley* (sequential) and *stencils* (sequential).

5.4 Performance for Specific Benchmarks

Observation metrics. In this section, we pick one benchmark from each of the six categories to find the reasons for the differences in performance observed among different versions. The representative benchmarks are: *2mm* (kernels), *syr2k* (blas), *lu* (solvers), *covariance* (datamining), *floyd-warshall* (medley), and *jacobi-2d* (stencils). We chose them on conditions that they have long execution time, and they are being compiled and executed without error by all polyhedral compilers. We provide an analysis based on the observation of some execution data collected with the hardware counters using the linux `perf` profiling tool. Table 5.5 presents measurements collected for sequential runs. They can be split in two categories of metrics:

(1) Scalar and Vector Instructions The *scalar-ins* and *vector-ins* lines of the table report the ratio of the number of scalar (resp. vector) instructions to the number of floating-point instructions counted in the baseline. The number of vector instructions is computed counting each vector instruction as its equivalent number of floating-point

operations on doubles, that is 2, 4 and 8 for respectively 128, 256 and 512 bits vector instruction sets. We chose this representation to easily see on one hand, to what extent the native compiler could vectorize the code and the nature of the benchmark's computations (some benchmarks like *floyd-warshall* only compute on integers), and on another hand how these shares change on the transformed code.

For instance, we can read in the Table 5.5 that for *2mm*, the compilers `gcc`, `clang` and `icc` could vectorize almost all floating-point instructions, or that the binary produced from Polly executes more floating-point vector instructions (+17%) than the baseline compiled with `clang`. The trend is comparable between the baseline and the optimized codes: the vectorization opportunities have not been diminished by the transformations. On the contrary, in *jacobi-2d* for example, the code produced by Polygeist results in only 6% vector instructions compared to 99.5% in the base version (`clang`). In this case, it is clear that the vectorization opportunities were not detected by the compiler in the transformed code. We provide an in-depth analysis with vector debug passes in the following sections for this poor vectorization.

There are multiple factors that explain the variations in vectorization. On one hand, source-to-source polyhedral compilers add vectorization directives[5] by exploiting the result of their dependency analysis, that can enable more vectorization opportunities. However, these are merely recommendations to the compiler which may just ignore them. On the other hand, the codes produced by polyhedral compilers generally expose more complex loop nests which may annihilate vectorization attempts by the auto-vectorizers.

The total number of floating-point instructions may vary for diverse reasons. The `-O3` optimization level generally requests the auto-vectorizer to generate adaptive code driven by a cost model[6]. The cost model computations themselves are costly when repeated frequently to control an inner-loop with few iterations. Another example is that fused operations such as `fma` is possible in one version of the code and not in another.

(2) LLC-miss, page-fault and branch misprediction These metrics globally reflect the data-locality and branch prediction efficiency of the execution. In Table 5.5, the lines *LLC-miss*, *page-fault* and *branch-miss* report ratios of event counts of the optimized version over the base version. We choose to report *LLC-miss* over *L2-miss*, because an *LLC-miss* involves an access to main memory, that has a high impact on performance.

[5] for example `#pragma ivdep`, `#pragma vector always`, or `#pragma simd`.

[6] `gcc` for instance sets the switch `-fvect-cost-model` to `dynamic` at the `-O3` level, while it is set to `very-cheap` with `-O2`. With `dynamic` a runtime code guards the vectorized code-path to enable vectorization only when estimated profitable. The `very-cheap` model only vectorizes the code if the scalar iteration count is known to be a multiple of the vector size.

			gcc					clang					ROSE		icc			
Benchm.	**metric**	**gcc**	**GRA**	**Pluto**	**PoCC**	**PPCG**	**clang**	**Polly**	**P'gst**	**Pluto**	**PoCC**	**PPCG**	**ROSE**	**P'opt**	**icc**	**Pluto**	**PoCC**	**PPCG**
2mm	scalar-ins	0.013%	0.013%	0.758%	0.775%	1.599%	0.358%	1.652%	0.620%	0.816%	0.758%	1.661%	100.0%	0.606%	0.209%	1.394%	0.703%	3.175%
(kernel)	vector-ins	99.99%	100.0%	128.3%	101.8%	107.3%	99.64%	117.1%	82.66%	117.2%	90.69%	116.7%	0.000%	80.99%	99.79%	125.4%	93.36%	134.6%
	LLC-miss	2.8E+08	1.091	0.141	0.143	0.042	3.5E+08	0.052	0.131	0.107	0.091	0.052	3.6E+08	0.103	2.9E+06	11.96	9.894	5.867
	page-fault	3.1E+03	1.001	0.987	0.982	0.983	3.2E+03	0.995	0.808	1.003	1.153	1.155	3.7E+03	0.859	3.1E+03	1.028	1.534	1.190
	branch-miss	4.5E+06	0.996	1.504	0.105	1.389	6.8E+06	1.983	0.126	1.000	0.978	1.475	6.8E+06	2.070	6.8E+06	1.958	1.980	1.981
	speedups	1	1.032	2.104	3.804	2.356	1	1.729	5.158	1.884	2.536	1.871	1	2.179	1	0.174	0.200	0.145
syr2k	scalar-ins	99.99%	99.88%	0.633%	0.634%	0.657%	0.117%	0.116%	120.3%	0.643%	0.650%	0.651%	0.038%	0.036%	0.020%	1.281%	1.278%	1.272%
(blas)	vector-ins	0.010%	0.010%	102.6%	101.5%	103.0%	99.88%	100.8%	0.000%	111.8%	111.5%	109.4%	99.96%	100.7%	99.98%	105.6%	105.6%	103.8%
	LLC-miss	1.0E+09	0.997	0.007	0.006	0.029	1.0E+09	0.029	0.001	0.030	0.029	0.031	1.1E+09	0.025	2.3E+08	0.129	0.141	0.136
	page-fault	2.1E+03	0.998	0.933	1.176	1.174	3.1E+03	0.670	0.660	0.808	0.642	0.808	2.1E+03	0.959	2.5E+03	0.782	0.983	0.984
	branch-miss	2.5E+06	0.995	2.716	2.714	0.213	4.3E+06	1.605	1.621	1.635	1.631	1.576	4.7E+06	1.482	3.4E+06	1.987	2.032	2.009
	speedups	1	0.980	6.253	6.168	7.953	1	3.380	2.310	7.831	8.503	7.408	1	4.680	1	2.338	2.659	2.625
lu	scalar-ins	25.02%	24.99%	0.028%	0.165%	0.019%	24.94%	0.059%	0.111%	24.55%	24.79%	0.165%	24.97%	12.51%	12.60%	24.73%	0.482%	24.86%
(solver)	vector-ins	74.98%	75.05%	100.10%	99.91%	100.7%	75.06%	99.87%	99.78%	75.33%	75.19%	99.91%	75.03%	87.50%	87.40%	73.39%	99.60%	74.54%
	LLC-miss	1.1E+10	0.993	0.787	0.790	0.788	1.4E+09	0.883	0.195	0.173	0.180	0.198	6.5E+09	0.672	1.4E+09	0.004	0.042	0.004
	page-fault	2.7E+03	1.000	1.007	1.337	0.987	3.3E+03	0.996	0.839	1.001	1.000	0.982	3.1E+03	0.981	2.2E+03	0.958	0.90	0.96
	branch-miss	3.2E+07	1.002	0.850	0.933	0.827	3.7E+07	0.662	0.479	0.485	0.486	1.052	3.2E+07	1.318	1.7E+07	1.390	0.739	1.491
	speedups	1	1.010	11.62	11.54	10.71	1	1.184	5.483	2.616	2.597	6.807	1	2.949	1	2.199	6.773	1.834
covari-	scalar-ins	0.017%	0.017%	0.091%	0.148%	1.601%	0.299%	1.840%	1.690%	1.541%	1.119%	1.854%	100.0%	0.058%	0.112%	0.121%	0.079%	3.159%
ance	vector-ins	99.98%	100.2%	104.2%	104.9%	100.8%	99.70%	122.2%	101.4%	101.7%	103.3%	121.5%	0.000%	100.8%	99.89%	102.7%	97.80%	122.1%
(datam.)	LLC-miss	4.8E+08	0.956	0.042	0.043	0.039	4.9E+08	0.066	0.044	0.041	0.040	0.047	4.5E+08	0.026	4.6E+08	0.037	0.037	0.045
	page-fault	2.2E+03	1.000	0.674	0.675	0.950	1.6E+03	1.265	0.899	1.222	1.223	0.965	2.2E+03	0.933	1.6E+03	0.975	0.973	1.328
	branch-miss	3.5E+06	1.000	0.114	0.115	2.747	3.5E+06	3.026	0.271	3.782	3.630	2.956	3.5E+06	2.898	7.1E+06	1.467	1.491	1.448
	speedups	1	0.993	8.475	8.478	2.743	1	1.726	5.050	6.724	6.861	1.708	1	3.407	1	2.045	2.142	0.901
floyd-	scalar-ins	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
warshall	vector-ins	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
(medley)	LLC-miss	6.0E+07	1.004	54.51	54.08	39.59	5.2E+07	0.947	158.4	84.02	84.33	165.9	5.9E+07	148.4	1.3E+10	0.031	0.031	0.030
	page-fault	1.8E+03	1.001	1.218	0.935	1.184	3.4E+04	0.052	0.040	0.062	0.097	0.053	1.8E+03	0.967	2.3E+03	0.796	0.791	1.010
	branch-miss	3.2E+07	1.001	6.483	6.533	0.107	3.2E+07	0.998	0.228	5.348	4.448	0.211	3.2E+07	5.282	3.5E+07	4.952	4.947	31.066
	speedups	1	0.999	1.471	1.529	1.780	1	0.998	0.483	0.250	0.251	0.735	1	1.096	1	4.329	4.352	3.316
jacobi-	scalar-ins	0.000%	0.000%	0.077%	0.071%	99.87%	0.500%	99.77%	93.65%	100.00%	99.89%	99.90%	0.000%	0.071%	0.357%	0.150%	0.144%	99.97%
2d	vector-ins	100.00%	100.0%	99.97%	99.97%	0.000%	99.50%	0.000%	6.239%	0.035%	0.037%	0.017%	100.00%	100.1%	99.64%	100.1%	100.2%	0.019%
(stencil)	LLC-miss	9.2E+08	1.017	0.065	0.004	0.001	1.0E+09	0.070	0.037	0.009	0.009	0.001	1.0E+09	0.006	9.9E+08	0.006	0.006	0.001
	page-fault	2.4E+03	1.000	0.768	0.792	1.000	1.9E+03	1.390	1.185	0.999	1.505	1.298	3.8E+03	0.494	1.9E+03	0.999	0.938	1.031
	branch-miss	5.7E+06	1.000	0.130	1.524	1.574	5.7E+06	87.47	4.241	1.600	1.668	41.388	5.7E+06	1.472	5.7E+06	1.480	1.486	1.642
	speedups	1	1.036	1.011	0.915	0.825	1	0.334	0.992	1.068	0.975	0.777	1	0.927	1	1.038	2.188	0.826

Table 5.5: Performance ratios for various profiling metrics against their respective compiler for six benchmarks using the `perf` profiling tool on an Intel machine (**sequential execution**). GRA - GRAPHITE; Pluto - Pluto with --tile; P'gst - Polygeist; P'opt - PolyOpt.

The raw numbers of misses or page-faults are given in the base version column (first column of each group of compiler, in blue). We compare different versions compiled with the same compiler and options. A lower than 1 ratio is an indication that the compiled code does better in terms of locality or branch prediction. For instance, the *LLC-miss* count of the code generated by Polly for *2mm* is only 5.2% (0.052×) the one observed when executing the base version *2mm* (both compiled with `clang`). A decrease in branch miss is an indication that the branch prediction is effective and the optimized code performs better. For example, the branch-miss count of the code generated by Polygeist is only 12.6% (0.126×) the one observed when executing the base version of *2mm*.

(3) Speedup Finally, the table gives for each benchmark the speedup of the transformed version compared to the baseline compiler version.

5.4.1 Sequential Execution Analysis.

2mm (kernels). Overall, we see in Table 5.5 a slight increase (above 100%) in the number of vector instructions for one polyhedral compiler out of two, and sometimes a slight decrease, but these small variations do not account significantly in the observed speedups. Notice that the `ROSE` compiler does not vectorize the native *2mm* code at all, while PolyOpt's generated code exhibits a number of vector instructions similar to the other polyhedral compilers. Most of the optimized codes (except those compiled with GRAPHITE and `icc`) do show a significant speedup thanks to a 10× reduction in *LLC-misses*. On top of that, `PoCC+gcc` and Polygeist generated codes have better branch prediction and show the best speedups. The optimized codes compiled with `icc` have execution times on par with other compilers and the observed slowdowns in the table are due to the fact that the original code generated by `icc` performs about 10× faster than its counterparts. This is due to a 100× reduction in LLC-misses compared to `gcc` and `clang`. This confirms the observation we made in section 5.3.3, that `icc` optimizes very effectively the native `kernels` codes.

syr2k (blas). We observe important discrepancies regarding vectorization between the compilers. While almost all floating-point instructions can be vectorized, `gcc` fails to detect them on the native code, while it achieves it on all the transformed codes (except GRAPHITE). For the other compilers the vectorization rate is 100% on the native and on the optimized codes, except for Polygeist where the `clang` compiler does not vectorize them. The optimized codes also show a significant reduction in the *LLC-misses*. We can pinpoint in this benchmark the effect of *branch-misses* as `Pluto+gcc`, `PoCC+gcc` and `PPCG+gcc` show similar counts for all the hardware counters except *branch-misses*: a reduction of 10× in this count (`PPCG+gcc`) leads to an increased speedup of 27-29%. As in *2mm*, the better use of cache mostly contributes to the observed speedups of Pluto, PoCC and PPCG. Polly had an increase in the total number of instructions (not shown in table) compared to `clang` and that is the reason for smaller speedups than its counterparts.

For *symm* (another *blas* benchmark), only PPCG produces a code with better performance by dividing the counts for *LLC-misses* by 35; the other polyhedral compilers perform similar to the baseline. For all the other benchmarks in the *blas* category, most source-to-source polyhedral compilers generate code that performs better than the baseline.

lu (solvers). A large part of the optimized codes show significant speedups (2×-11×). What appears to contribute most to the speedup is the number of floating-point instructions promoted from scalar to vector instructions. For instance the `Pluto+gcc` code achieves to vectorize almost all floating-point instructions (less than 0.03% scalar instructions remain) resulting in a 11.6× speedup. On the contrary, `Pluto+clang` code executes with the same number of scalar and vector instructions as the native code and has a speedup of 2.6× only. Still, this is a significant speedup due to a better cache utilization: the number of *LLC-misses* is 0.17× the one of the native code. We may underline here the effect of reduction in *LLC-misses* in this benchmark as Polly and Polygeist show similar counts for all counters except *LLC-misses*: comparing those two versions, a reduction of 4.5× in *LLC-misses* leads to a speedup of 4.6×.

Similar to *symm*, for *ludcmp (solvers)* the PPCG optimized code has better execution time compared to all other polyhedral compilers.

covariance (datamining). Most optimized codes show a significant speedup of 2×-8×. The general trend in this benchmark is again a massive reduction in LLC-misses. Polly, `PPCG+clang`, and `PPCG+icc` increase the total number of executed instructions (not shown in the table) and that is the reason for reduced speedups than their counterparts. One particular case here is, like in `2mm`, the `ROSE` compiler not being able to vectorize the native code while PolyOpt can. The inefficient branch prediction in `PPCG+gcc` reduces the speedups as compared to its counterparts (`Pluto+gcc` and `PoCC+gcc`). The Polygeist code generates less branch misses than `Pluto/PoCC+clang`, but reports lower speedups because of a greater number of total instructions.

floyd-warshall (medley). The optimized codes show almost no speedup in this benchmark. There are no floating-point operations at all in this code. Although `Pluto+icc` and `PoCC+icc` have speedups of 4×, this result is misleading because the `icc` native compiled code takes 6× more execution time compared to `gcc` and `clang`. We observe a large number of *LLC-misses* in the `icc` native version. After optimization, the *LLC-misses* decrease and there is a 4× speedup. Nonetheless the execution time is twice as long as the `Pluto+gcc` version. Using the other compilers, all polyhedral compilers increase the number of *LLC-misses* for this benchmark, but it is three orders of magnitude smaller than the one of the `icc` version.

We observe no correlation between the speedup and the hardware counters presented in this table. However, we observe a strong linear correlation between the ratios of cycle stalls and the speedups. The reason for those cycle stalls is probably a different instruction reordering resulting in worse pipeline usage, or a less performing register

allocation. In both cases, the `icc` compiler on the base code performs better but at the cost of a huge increase in *LLC-misses* ending up to a slowdown.

jacobi-2d (stencils). The polyhedral optimizations improve data locality by *tiling* the code, but they also hinder vectorization opportunities, for the `clang` compiler and for the PPCG generated code with any compiler. As a result, the speedup of these versions is reduced. Further analysis with the help of `clang` and `gcc` vector debugging pass shows a forward-dependency across the innermost loop in the polyhedral optimized *jacobi-2d* code. This dependency prevents vectorization by `clang`, but `gcc` achieved to unroll and re-order those loop statements to vectorize them.

Although almost all polyhedral compilers largely reduced the *LLC-miss* counts, this does not induce any speedup. Considering each compiler separately, the version that has the less *page-faults* is among the ones that have the best speedup, but the correlation is pretty weak and does not hold amongst different compilers. Moreover, the `Pluto+gcc` version for example has much lower values for all *cache-misses* (0.06×), *page-faults* (0.77×) and *branch-misses* (0.13×) than the `gcc` baseline, and still only achieves a 1% speedup! We suspect, as in *floyd-warshall*, that another factor like efficient instruction pipelining or register allocation of this stencil code is the main factor of performance. Overall, most versions show very small variations in speedup, which stays around 1×. Only `PoCC+icc` reaches a 2× speedup although there is no special improvement in vectorization, LLC-misses, pages-faults, nor branch misses compared to the other versions.

5.4.2 Parallel Execution Analysis.

The benefit of parallelization by polyhedral compilers varies a lot depending on the perspective we adopt to analyze the results: either with respect to the base code (which is the user's primary concern) or with respect to the sequential optimized code. In the former case, the benefit of polyhedral compilation combines the benefits of all loop transformations and parallelization, and in the latter case we see the sole effect of parallelization.

Table 5.7 shows in the main columns the speedups obtained for 7 benchmarks (the 6 discussed in sequential analysis + *heat-3d*) in the two cases for all three compilers (`gcc`, `clang`, `icc`), and three polyhedral source-to-source parallelizing compilers (Pluto, PoCC, PPCG). PPCG is not mentioned for *floyd-warshall* and *jacobi-2d* because it did not produce a parallel code. We can see that compared to the sequential optimized version, the efficiency when using all 40 cores ranges from low (about 0.4 for a 16× speedup) to very low (about 0.05 for a 2× speedup)[7]. However, comparing the parallel execution times with the base sequential version leads to results ranging from super-linear speedups (over 40×) to some rare slowdowns.

[7]The only exception is *lu* with `Pluto+clang` and `PoCC+clang` with an efficiency of more than 0.8.

		gcc					clang						ROSE		icc			
Benchm.	**metric**	**gcc**	**GRA**	**Pluto**	**PoCC**	**PPCG**	**clang**	**Polly**	**P'gst**	**Pluto**	**PoCC**	**PPCG**	**ROSE**	**P'opt**	**icc**	**Pluto**	**PoCC**	**PPCG**
2mm	scalar-ins	0.013%	0.013%	73.73%	0.839%	118.2%	0.358%	1.713%	0.680%	40.71%	44.17%	93.98%	100.0%	0.626%	0.209%	1.035%	0.652%	2.625%
(kernel)	vector-ins	99.99%	99.4%	52.98%	113.9%	0.000%	99.64%	112.2%	73.78%	62.07%	38.91%	0.033%	0.000%	85.58%	99.79%	98.52%	92.46%	135.6%
	LLC-miss	2.8E+08	1.025	0.017	0.015	0.014	3.5E+08	0.013	0.027	0.008	0.006	0.011	3.6E+08	0.015	2.9E+06	4.755	3.781	5.807
	page-fault	3.1E+03	0.999	1.166	1.166	1.327	3.2E+03	0.047	0.346	0.448	0.448	0.410	3.7E+03	4.152	3.1E+03	1.013	1.013	1.027
	branch-miss	4.5E+06	1.002	43.81	2.090	3.512	6.8E+06	1.720	1.484	1.306	1.380	2.672	6.8E+06	2.031	6.8E+06	2.319	2.216	1.859
	speedups	1	1.050	25.60	47.01	15.12	1	33.38	25.65	25.08	34.18	18.88	1	21.04	1	2.452	4.135	2.273
syr2k	scalar-ins	99.99%	100.04%	99.45%	99.81%	99.41%	0.117%	0.118%	119.6%	0.651%	0.629%	1.256%	0.038%	0.038%	0.020%	0.893%	0.827%	1.133%
(blas)	vector-ins	0.010%	0.010%	0.000%	0.000%	0.000%	99.88%	100.4%	0.095%	115.9%	96.73%	102.5%	99.96%	109.1%	99.98%	83.39%	69.23%	72.91%
	LLC-miss	1.0E+09	1.000	0.002	0.002	0.001	1.0E+09	0.035	0.001	0.021	0.020	0.021	1.1E+09	0.013	2.3E+08	0.093	0.075	0.073
	page-fault	2.1E+03	1.241	1.198	0.958	0.956	3.1E+03	0.045	0.433	0.576	0.575	0.584	2.1E+03	9.532	2.5E+03	0.596	0.596	0.604
	branch-miss	2.5E+06	0.992	2.902	2.856	2.798	4.3E+06	1.574	2.363	2.380	2.002	2.307	4.7E+06	1.598	3.4E+06	2.655	2.460	2.376
	speedups	1	0.813	50.46	50.86	39.32	1	59.02	25.86	71.42	68.10	83.76	1	73.39	1	34.55	34.68	37.44
lu	scalar-ins	25.02%	24.99%	0.027%	0.174%	0.019%	24.94%	0.155%	-	0.18%	0.30%	0.215%	24.97%	12.52%	12.60%	24.55%	0.422%	24.77%
(solver)	vector-ins	74.98%	74.99%	100.6%	102.6%	99.92%	75.06%	99.60%	-	99.86%	100.9%	99.60%	75.03%	87.58%	87.40%	70.73%	93.10%	71.83%
	LLC-miss	1.1E+10	1.007	0.807	0.803	1.244	1.4E+09	0.072	-	0.765	0.673	1.421	6.5E+09	0.798	1.4E+09	0.019	0.028	2.476
	page-fault	2.7E+03	1.227	1.257	1.258	8.436	3.3E+03	0.016	-	1.043	0.989	1.962	3.1E+03	23.274	2.2E+03	0.892	0.900	1.741
	branch-miss	3.2E+07	1.008	1.329	1.192	55.61	3.7E+07	0.458	-	1.202	0.601	35.93	3.2E+07	1.360	1.7E+07	1.786	0.951	70.71
	speedups	1	1.029	129.4	110.5	0.153	1	19.75	-	90.95	85.70	0.545	1	53.66	1	28.05	70.84	0.368
covari-	scalar-ins	0.017%	99.78%	0.034%	0.024%	102.4%	0.299%	1.890%	1.107%	1.013%	0.970%	1.635%	100.0%	0.036%	0.112%	0.065%	0.052%	3.290%
ance	vector-ins	99.98%	0.000%	108.5%	109.0%	0.000%	99.70%	124.4%	102.0%	112.2%	113.0%	134.3%	0.000%	106.6%	99.89%	115.7%	112.7%	117.4%
(datam.)	LLC-miss	4.8E+08	1.020	0.043	0.042	0.070	4.9E+08	0.088	0.066	0.045	0.046	0.057	4.5E+08	0.021	4.6E+08	0.043	0.040	0.053
	page-fault	2.2E+03	1.327	1.265	1.291	0.955	1.6E+03	0.978	1.074	1.502	1.508	1.436	2.2E+03	1.304	1.6E+03	1.291	1.603	1.520
	branch-miss	3.5E+06	1.014	3.128	3.145	3.138	3.5E+06	3.053	4.878	4.150	4.016	3.910	3.5E+06	3.163	7.1E+06	1.614	1.541	1.775
	speedups	1	0.782	47.41	52.94	19.93	1	21.04	34.92	46.48	45.96	17.17	1	36.99	1	22.04	17.83	9.850
floyd-	scalar-ins	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
warshall	vector-ins	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
(medley)	LLC-miss	6.0E+07	1.038	80.23	80.46	38.17	5.2E+07	0.870	157.4	59.63	59.69	171.0	5.9E+07	103.4	1.3E+10	0.224	0.226	0.022
	page-fault	1.8E+03	0.999	2.155	2.064	0.902	3.4E+04	1.006	5.271	2.698	2.657	41.69	1.8E+03	1.651	2.3E+03	2.083	3.187	32.82
	branch-miss	3.2E+07	0.998	12.77	12.87	0.110	3.2E+07	1.003	50.28	16.74	16.29	0.211	3.2E+07	8.158	3.5E+07	13.15	13.70	5.012
	speedups	1	0.734	3.305	3.126	1.692	1	1.002	2.470	2.630	2.641	0.713	1	3.362	1	21.05	21.03	4.195
jacobi-	scalar-ins	0.000%	0.000%	0.078%	99.98%	99.90%	0.500%	99.77%	92.31%	99.48%	101.0%	99.83%	0.000%	0.073%	0.357%	0.172%	0.137%	99.56%
2d	vector-ins	100.0%	100.0%	100.1%	0.038%	0.000%	99.50%	0.000%	6.246%	0.034%	0.034%	0.046%	100.0%	99.37%	99.64%	97.97%	95.70%	0.032%
(stencil)	LLC-miss	9.2E+08	1.014	0.096	0.003	0.001	1.0E+09	0.070	0.015	0.008	0.008	0.001	1.0E+09	0.009	9.9E+08	0.006	0.007	0.001
	page-fault	2.4E+03	1.260	0.933	0.905	1.002	1.9E+03	1.575	0.965	1.211	1.216	40.44	3.8E+03	9.078	1.9E+03	1.497	1.073	40.25
	branch-miss	5.7E+06	1.010	1.633	1.900	1.573	5.7E+06	6.148	6.541	4.466	4.310	41.45	5.7E+06	1.876	5.7E+06	3.639	2.758	1.711
	speedups	1	0.982	10.39	11.71	0.825	1	3.525	15.53	5.168	5.593	0.768	1	9.965	1	11.08	22.62	0.787

Table 5.6: Performance ratios for various profiling metrics against their respective compiler for six benchmarks using the `perf` performance profiling tool on an Intel machine (**parallel execution on 40 cores**). GRA - GRAPHITE; Pluto - Pluto with --tile; P'gst - Polygeist; P'opt - PolyOpt.

Bench.	polyhedral compiler	#parallel region	core-used	gcc-speedups base	gcc-speedups opt	clang-speedups base	clang-speedups opt	icc-speedups base	icc-speedups opt
2mm	Pluto	2	all 40	25.60	12.17	25.08	13.31	2.452	14.09
	PoCC	2	all 40	47.01	12.36	34.18	13.48	4.135	20.67
	PPCG	2	all 40	15.12	6.418	18.88	10.09	2.273	15.66
syr2k	Pluto	2	all 40	50.46	8.070	71.42	9.121	34.55	14.78
	PoCC	2	all 40	50.86	8.245	68.10	8.009	34.68	13.04
	PPCG	1	all 40	39.32	4.944	83.76	11.31	37.44	14.27
lu	Pluto	250	all 40	129.4	11.13	90.95	34.77	28.05	12.75
	PoCC	250	all 40	110.5	9.58	85.70	32.99	70.84	10.46
	PPCG	21.08m	32	0.153	0.014	0.545	0.080	0.368	0.201
covari-ance	Pluto	7	all 40	47.41	5.594	46.48	6.912	22.04	10.78
	PoCC	7	all 40	52.94	6.245	45.96	6.700	17.83	8.32
	PPCG	5	all 40	19.93	7.265	17.17	10.05	9.850	10.94
floyd-warshall	Pluto	1.96m	all 40	3.305	2.247	2.630	10.52	21.05	4.863
	PoCC	1.96m	all 40	3.126	2.044	2.641	10.51	21.03	4.832
	PPCG	-	-	-	-	-	-	-	-
jacobi-2d	Pluto	182	30	10.39	10.27	5.168	4.839	11.08	10.67
	PoCC	182	30	11.71	12.81	5.593	5.735	22.62	10.34
	PPCG	-	-	-	-	-	-	-	-
heat-3d	Pluto	100	3	1.543	1.024	1.052	0.704	4.483	1.546
	PoCC	100	3	1.389	2.100	1.004	0.598	4.350	1.559
	PPCG	31.08m	32	0.012	0.011	0.023	0.012	1.000	0.217

Table 5.7: The total number of parallel regions, number of cores utilized, and speedups of the parallel optimized code of source-to-source polyhedral compilers compared to the baseline version (base) and the sequential optimized version (opt). Pluto with --tile; m - is millions.

While we have collected the same hardware counters as for the sequential executions (reported in Table 5.6) we found out that two other factors are mostly governing the performance of parallel executions: the total number of OpenMP parallel regions, and the maximum number of cores (out of 40) utilized during a run. Both are reported in Table 5.7.

Based on these two factors, we separate the benchmarks in two groups:

- the *promising cases* gather executions that create a reasonable number of parallel regions and use most of the available cores;
- the *ill cases* contain the executions with an excessive number of parallel regions (inducing large synchronization overhead) or distribute the work to a small fraction of the available cores (inducing low parallelism).

Hereunder, we group the parallelized benchmarks based on these criteria and discuss their behavior at the light of their parallel efficiency, that is the ratio of speedup with respect to their *sequential optimized* version to the number of available cores:

promising cases:

- *2mm (kernels)* and *syr2k (blas)* are effectively parallelized with a minimal number of parallel regions and utilize all the available cores. However, the efficiency is not very good (0.3 and 0.2 resp.). For *symm* (also in the *blas* category, not shown in table), only PPCG can emit correct parallel code, but only one statement out of two is parallelized due to dependencies in the other one. As a result it shows a very low efficiency (0.05). Pluto is unable to parallelize it. PoCC emits an invalid parallel code.

- For *lu (solvers)*, Pluto and PoCC achieve to parallelize the second loop of the 6-level loop nest (after tiling) with a contrasted efficiency: it ends up with the best efficiency over all results with `clang` (more than 0.8) and only 0.35 with `gcc`. PPCG parallelizes an inner loop which results in millions of parallel regions creations and hence a performance collapse. This is the main reason for PPCG's large slowdown in the *solvers* category (Figs. 5.5-5.7 (b)).

- *covariance (datamining)* is effectively parallelized, but shows a limited efficiency for all polyhedral compilers (e.g., 0.15 for `gcc+Pluto`), which can be explained by the large increase in the number of branch-misses in the parallel executions (see Table 5.6).

- *jacobi-2d (stencils)*, similarly to *lu*, spawns parallel regions in the order of a hundred. However the parallelism is limited to 30 cores and the efficiency is lower than with *lu* (0.127 with `clang`, 0.275 with `gcc`).

ill cases:

- For *floyd-warshall (medley)*, Pluto and PoCC generate a code that creates a huge number of parallel regions resulting in a low efficiency of 0.05. PPCG does not parallelize it.

- *heat-3d (stencils)*: though the Pluto and PoCC versions have a reasonable number of parallel regions, they parallelize a (parametric) loop with 3 iterations and therefore utilize only 3 out of the 40 cores.

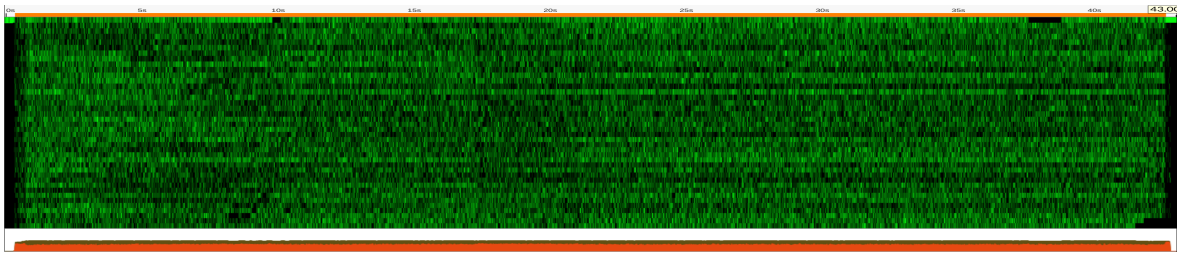
- not shown in table, *ludcmp (solvers)*: none of the polyhedral compilers could parallelize this code because of a dependency on an intermediate variable.

- not shown in table, *doitgen (kernels)* has a longer execution time in parallel than in sequential. Its execution both generates an excessive number of parallel regions and makes a low utilization of cores, due to the dependencies which prevent parallelization of the two outer loops.

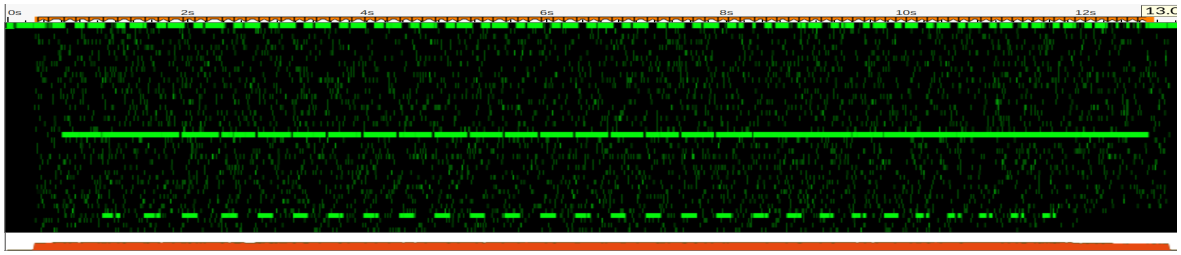
Parallel Performance Analysis. We used Intel's VTune and Pluto-tile compiled with icc to collect the cores' utilization for some of our benchmarks exposing different parallel behaviors. Figures 5.9 and 5.10 show the utilization for promising and ill cases respectively. Time (in seconds) is represented horizontally and the bar(s) over the timeline are the parallel region(s); the vertical axis represents the threads; the green color intensity denotes the cores' activity; and the bottom bars shows the average (over 40 cores) CPU utilization. For *2mm* (Figure 5.9a), the left and upper regions show a perfect utilization of cores. The imbalance observed at bottom right is due to a low number of iterations of the parallel loop (after 40 iterations run on 40 threads, only 10 remain in the second part of the parallel region). In *lu* (Figure 5.9b), we observe a large number of parallel regions and a low compute intensity. This leads to a performance reduction even though we classified it as a promising case. *Floyd-warshall*

(a) 2mm (b) lu

Figure 5.9: Intel VTune reports of cores utilization for promising cases



(a) floyd-warshall



(b) heat-3d

Figure 5.10: Intel VTune reports of cores' utilization for ill cases

(Figure 5.10a) has a very large number of parallel regions (1.96 million) and a very low compute intensity, as expected. For *heat-3d* (Figure 5.10b), we observe a very low utilization of cores: only 3 threads are active throughout the execution. These measurements just confirm the different behaviours that we analyzed in the previous paragraph.

5.5 Pluto with Different Tiling Options

Figure 5.11 shows geomean speedups of the --tile, --l2tile, and --diamond-tile options of Pluto compared against the baseline standard compilers for 21 Polybench/C benchmarks. The Pluto-tile and Pluto-dia-tile versions show very similar speedups, except for *stencils*: using `clang` Pluto-dia-tile generates more efficient (+22%) code than Pluto-tile, but using `icc` it is the opposite (-40%). We can also see that the --tile and --diamond-tile options perform better than the --l2tile option for all categories except *stencils*. We investigated the hardware counter values and found out that the number of LLC-misses is often higher using --l2tile. We observe the same behaviour in sequential runs where --l2tile performs worse even for *stencils*.

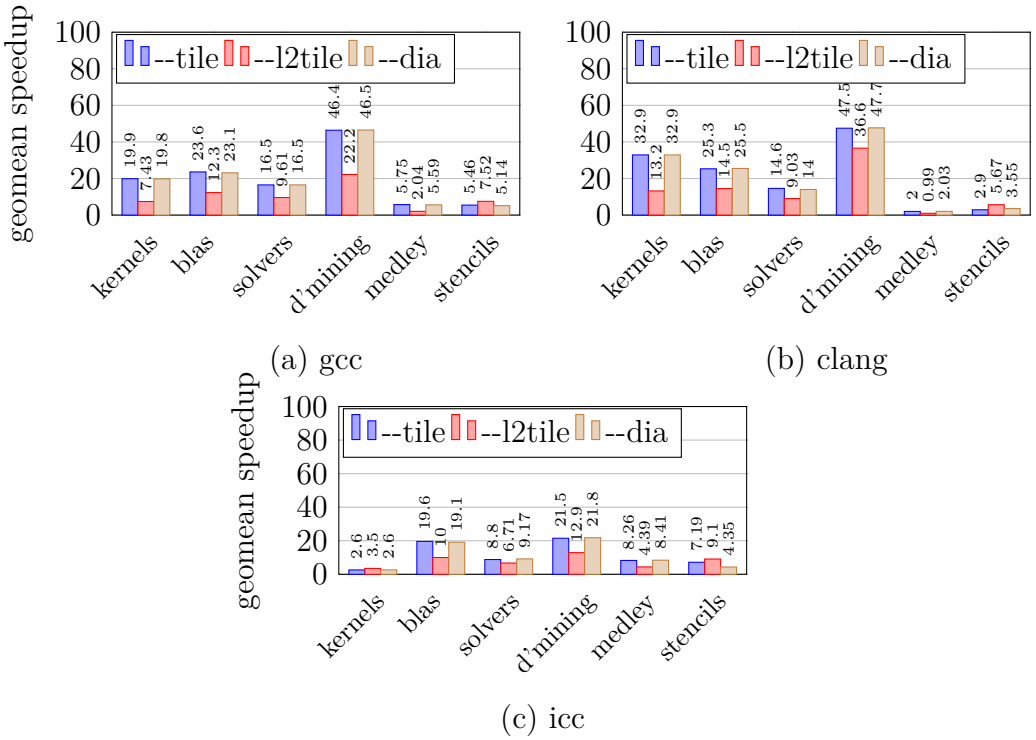


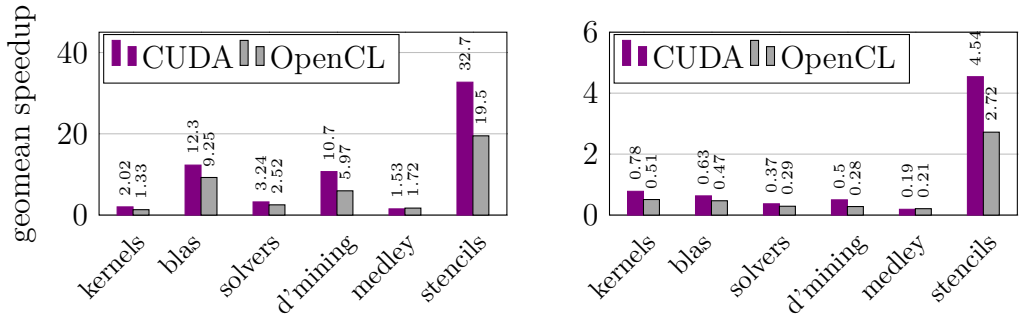
Figure 5.11: Geomean parallel speedups of Pluto with different tiling options using different compilers

Considering some individual benchmarks in sequential run (not shown in the figure), there are some exceptions: for a few tiny benchmarks like *atax*, *bicg*, *doitgen*, *gemver* and *gesummv* the --l2tile code is best. The reason is a reduction in total instruction count and branch-misses of these versions. In parallel runs, the Pluto-l2tile optimized codes of most benchmarks are slower than with the other tiling options. This can be further explained by a lower degree of parallelism and more imbalance, since there are fewer tiles of larger size than with the other tiling options. Only *nussinov (medley)*, and the stencils *seidel-2d*, *fdtd-2d*, *heat-3d* with --l2tile perform better. In particular, *heat-3d* shows a speedup of more than 3× because when activating --l2tile Pluto chooses another parallel loop that utilizes all 40 cores, while the other tiling versions could not utilize all of them.

5.6 PPCG: CUDA and OpenCL

PPCG is the only polyhedral open-source compiler that we could use to generate an effective GPU code. We compare this PPCG GPU generated codes to the CPU runs in sequential and in parallel, on the same 21 PolyBench/C benchmarks used in Section 5.3.2. The emitted CUDA code is executed on an A100 Nvidia GPU, OpenCL code using OpenCL version 3.0.1 on the same GPU, and the CPU code on the same 40-core Intel CPU as above.

Figure 5.12 shows the geomean speedups of PPCG emitted CUDA and OpenCL



(a) PPCG CUDA/OpenCL vs icc(b) PPCG CUDA/OpenCL vs Pluto+icc

Figure 5.12: Geomean speedups of PPCG CUDA and OpenCL codes compared to the best: (left fig.) sequential code compiled with `icc`; and (right fig.) parallel code produced by Pluto-tile + icc.

codes for PolyBench/C. We compare the performance of these optimized codes running on GPU to the baseline `icc` CPU sequential code (left figure), since it had the best geomean execution time in sequential. We also compare (right figure) with the parallel CPU execution of `Pluto+icc` with the --tile option.

5.6.1 CUDA

The PPCG generated CUDA codes perform better than the sequential CPU versions (Fig. 5.12(a)), but they report small speedups (1.5× - 3.2×) for `kernels`, `solvers` and `medley`, and the other ones report 10× - 33× speedups. Comparing with the parallel `Pluto+icc` versions (Fig. 5.12(b)), the CUDA optimized codes report slowdowns for all of them except for the *stencils* category.

In their paper from 2013 [57], the authors of PPCG report similar experimental results. Despite the fact that we do not find the same speedups due to the evolution of the hardware since this publication, we observe the same trends. The authors report speedups compared to the sequential code of more than 100× on some benchmarks using the MEDIUM_DATASET (*correlation*, *covariance*, *2mm*, *3mm*, *gemm*) while our measures are in the order of 10× for those. Even though we used the EXTRALARGE_DATASET, the execution times on GPU in most cases (4 out of 5) are lower than 1 second, which explains the low ratio of computation to data transfer and initialization times. All the benchmarks have a constant 0.2 to 0.3 seconds overhead for initialization (memory allocation and copy), which is a large part of the total execution time for many of them. We tried to manually increase the size of the datasets and observed better speedups in some of those cases.

Regarding the pathological cases reported in the original paper, we noted the same codes that could not be parallelized efficiently on GPU due to dependencies or due to many CPU-GPU synchronizations.

To investigate further, we used the Nvidia profiler (`nvprof`) to better analyze the executed CUDA codes. Here are our findings:

- many of those benchmarks are much too short to be executed efficiently on GPU in the categories *kernels*, *blas*, *datamining*, and *medley*.

- the kernels that were not parallel on CPU obviously could not be parallelized on GPU. This is the case for *ludcmp*, *cholesky* and *durbin* (*solvers* category), all of them containing at least one loop that could not be run in parallel on the GPU.

- the PPCG emitted *nussinov* CUDA code throws a compiler error.

- *floyd-warshall* and *doitgen* have too many synchronizations between CPU and GPU. They make many calls to cudaLaunchKernel since the parallelization is not efficient: the two outer loops are not parallel.

- the only cases where the GPU code runs faster than the CPU parallel code, are *symm (blas)* and the *stencils* category benchmarks (all of them except *jacobi-1d*, which is too short).

  However, the speedup obtained for those six benchmarks, when compared to the CPU parallel execution, is 6.3× on geomean. This is low compared to the 19× performance ratio (theoretical peak GFLOP/s) between the 40-core CPU and the A100 GPU.

5.6.2 OpenCL

We compiled and executed the PPCG OpenCL codes using the `gcc` compiler (with `-lOpenCL`). Overall, the OpenCL codes report a small slowdown compared to their counterpart CUDA versions, due to longer initialization and kernels launch times, but the overall trends are similar to the ones presented above.

The PPCG emitted *nussinov* OpenCL code compiles and runs correctly while the CUDA code did not. But this benchmark is short and reports a slowdown compared to the parallel `Pluto+icc` version.

5.7 Discussion

In this section we summarize the main findings of this survey. Most of the benchmarks for which polyhedral optimization does not bring an improvement are those whose execution times are short. We include those tiny benchmarks in the comparison although it is debatable to optimize such short executions. Overall, we observe that for sequential execution polyhedral compilers eventually produce faster codes for 19 out of 30 benchmarks, and win in all but two benchmarks if we do not consider the tiny benchmarks. Hence, it can be concluded that polyhedral compilers are very effective at optimizing those sequential codes. Moreover, they automatically generate parallel code.

Among polyhedral compilers, Pluto ranks best in many benchmarks (10 in sequential execution and 14 in parallel execution). We can also notice that the source-to-source polyhedral compilers win over built-in polyhedral compilers in almost all cases.

5.7.1 Sequential Code Optimization

The benefit from polyhedral optimization on a sequential code can be summarized in light of our analysis in Section 5.4.1. We identify three factors that contribute to the speedups: (i) data locality (LLC-hits, page-hits), (ii) branch prediction, and (iii) vectorization.

Data locality is greatly improved by polyhedral compilers and is by far the prime contributor for increased performance. Loop tiling shows a reduction in LLC misses to a notable extent. The level of tiling has some influence, the one level tiling and diamond tiling often being more efficient in reducing the cache-misses than the two level tiling. There is also a reduction in number of overall cache references (not shown in the tables) for most of the benchmarks, which means that register allocation is more efficient. There is a significant reduction in page faults for a few benchmarks.

A minor counterpart to the data locality benefit is an increased number of branch instructions and branch misses. Indeed, restructuring the code into more complex loop nests as part of the tiling optimization leads to more numerous tests in many of the benchmarks (*syr2k* for instance).

Polyhedral transformations do not favor in general the vectorization opportunities for current compilers auto-vectorizers. On one hand we found cases where the code restructuring actually favors vectorization, in particular PolyOpt allows the `ROSE` compiler to partially vectorize loops that `ROSE` did not vectorize in the native code because of the depth (at least 3) of the loop nest and its complex access pattern (2-d array access). Another example is *lu* in which `gcc`, `clang`, and `icc` have further vectorized the loops produced by Polly, Polygeist, `Pluto+gcc`, `PoCC+gcc`, and `PPCG+gcc`. On the other hand, we observed many cases where the transformation hinders vectorization as for example for the *stencil* benchmarks. The auto-vectorization pass reports the following reasons to not vectorize some loops that were vectorized in the native code: «multiple loop nests» (in `gcc`), «unsafe dependent memory operations in loop», or «loop nests having two or more consecutive inner loops» for example. The typical *jacobi-2d* benchmark is perfectly vectorized in its native version but almost not in the transformed versions.

5.7.2 Parallel Code Optimization

In the multi-threaded CPU experiments, we find that the parallelization performed by the polyhedral compilers exhibits only modest speedups or even slowdowns in some ill cases. Those ill cases are typically characterized by a parallel loop nested inside sequential loops, resulting in a huge number of parallel regions creation and destruction,

or low parallelism. Given the short execution times of some benchmarks, this becomes an overwhelming overhead. In favorable cases when the parallel dimension can be chosen as one of the outer loops, the distribution of memory-bound computations to 40 cores can benefit from distributed cache, but can conversely lead to RAM access congestion and cache false sharing overheads, reducing the benefit of tiling. In addition, we observe that some of these parallel codes are not vectorized by the compiler while they are vectorized in the sequential code. The vector cost model of the compiler is probably responsible for that, and an explicit vectorization directive generated by the source-to-source polyhedral compiler could help in these cases.

In the GPU experiments, we observe that, for most of the codes that run efficiently in parallel on CPU, the `EXTRALARGE_DATASET` size of PolyBench/C is too small to observe good speedups on current GPUs, except for *stencils*.

5.7.3 Benchmark Suite

Although PolyBench/C is favoured by the polyhedral community, the benchmark suite has been created long years back and there were no recent significant improvements or extensions made to it. In recent advancements, the polyhedral techniques are widely used in AI/ML applications and there are many new ML related polyhedral compilers. PolyBench/C does not include benchmarks involving ML affine computation patterns (like dense matrices and dense tensors) that are prevalent in most domains of natural language processing, AI vision, and generative AI. Few polyhedral compilers are used to optimize convolution-based neural networks (CNN) and there are no benchmarks to test convolution with common point-wise operation.

Another limitation is the size of the datasets. Though PolyBench/C supports five different sizes, they are now outdated when running on today's available computing power. The sizes of some benchmarks must be increased by orders of magnitude to target high-performance computing nodes. However this has to be done thoughtfully for each benchmark, to avoid huge memory allocation increase.

Chapter 6

Extending Polygeist to Generate OpenMP SIMD and GPU MLIR Code

In the previous chapter, we noticed that the state-of-the-art source-to-source polyhedral schedulers annotate loops that can be vectorized with directives, which are merely recommendations to the compiler. However, standard compiler auto-vectorizers may fail to vectorize them because of the complexity of the loop structure or nested statements in the restructured code. With the help of vector debugging passes, we find the main reasons reported by compilers to not vectorize the marked code was, for example: (i) could not determine the number of loop iterations, (ii) multiple nested loops, (iii) complex loop statements (like function calls, irregular control flow, and pointer/array deference), (iv) control flow inside loops, and (v) inner-loop count not invariant. Further, in a few cases, the vectorization is not applied because the vector cost-model of the back-end compiler estimates that it is not profitable.

PPCG is the only polyhedral compile capable of emitting heterogeneous code, but it uses many code generation and compilation flows with respect to the target architecture. The optimization techniques introduced in Chapter 4 could help solve these two limitations as we directly emit vector instructions for loop statements and with one abstract level of MLIR representation, the code can be lowered to different target architectures including different GPUs.

We choose Polygeist to generalize our technique as it is based on the MLIR compiler and would require reduced effort for generalization because the development environment (MLIR) is already set up by Polygeist. It generates polyhedral optimized (tiling and parallel loops) MLIR code, but it neither annotates the loops with vectorization directives nor auto-generates the vectorized code. Our generalization would benefit Polygeist and thereby enhance the polyhedral optimization techniques.

In this chapter, we describe a proposal to extend Polygeist to generate OpenMP SIMD MLIR code for vector loops. Later, these OpenMP SIMD loops can be lowered to vector instructions. We also want to further extend the code generation process to support GPU MLIR code thereby targeting accelerated architectures.

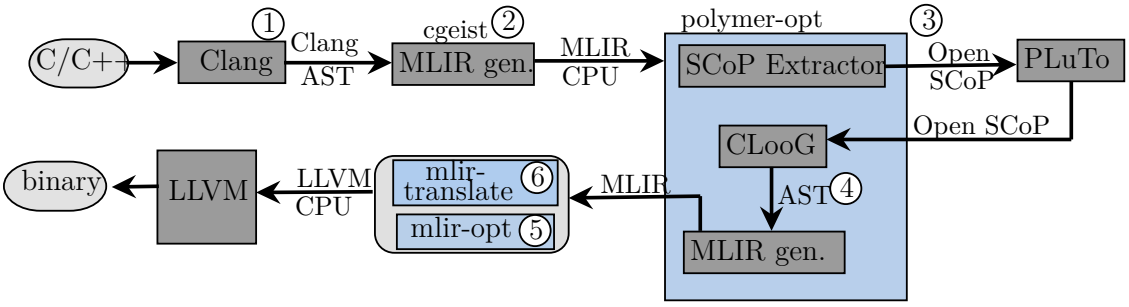


Figure 6.1: Overview of Polygeist compilation-flow.

6.1 Polygeist

Polygeist [15] is an MLIR based compilation tool that lowers `C/C++` code to Polyhedral MLIR code. Figure 6.1 shows the Polygeist compilation process. It takes an input source program, and

① extracts the high level-information using `clang` and generates a Clang AST,

② with a code generator named *cgeist*, Polygeist generates MLIR equivalent code from the Clang AST,

③ with a built-in tool named *polymer-opt*, the polyhedral representation is extracted (via *SCoP Extractor*), and using Pluto (or another external polyhedral scheduler) the extracted polyhedral representation is optimized for loop tiling and parallelization,

④ from the optimized `OpenSCoP` file, the CLooG code generator emits an AST which serves as an input for *cgeist* to generate MLIR code. *Cgeist* retains the information of the parallel loop identified by Pluto by marking it with the `scop.parallel` loop attribute,

⑤ an MLIR pass lowers the `scop.parallel` attribute loop to an OpenMP parallel loop,

⑥ finally the lowered MLIR code is translated into LLVM IR, that is passed to LLVM to generate an object file.

Listing 6.2 shows the polyhedral MLIR code generated by Polygeist for the loops of the *syr2k* kernel (from PolyBench/C [14]) shown in Listing 6.1. The `scop.parallel` annotated `affine.for` loop at line 2 and 8 will be converted to MLIR OpenMP parallel loop during lowering. Lines 3-5 and 9-13 are tiled loops executing statements S0 and S1, respectively. It is to be noticed that Pluto identifies the `affine.for` loop in line 5 of Listing 6.2 as vectorizable using vector directives, but Polygeist does not use this information as it can not lower the loop statements to an MLIR vector code.

Considering this context, we propose to extend the MLIR code generation process of Polygeist to convert the loops which are vector annotated by the polyhedral compiler to

```
#pragma scop
for (i = 0; i < n; i++) {
 for (j = 0; j <= i; j++)
  C[i][j] *= beta;  // S0
 for (k = 0; k < m; k++)
  for (j = 0; j <= i; j++)
   C[i][j] += A[j][k]*alpha*B[i][k] + B[j][k]*alpha*A[i][k]; // S1
 }
#pragma endscop
```

Listing 6.1: Compute intensive loops of the syr2k kernel.

```
// ... code skipped for space
affine.for %arg7 = 0 to #map()[%0] {scop.parallel}
 affine.for %arg8 = 0 to #map1(%arg7)
  affine.for %arg9 = #map2(%arg7) to min #map3(%arg7)[%0]
   affine.for %arg10 = #map2(%arg8) to min #map4(%arg9, %arg8)
    S0

affine.for %arg7 = 0 to #map()[%0] {scop.parallel}
 affine.for %arg8 = 0 to #map1(%arg7)
  affine.for %arg9 = 0 to #map()[%1]
   affine.for %arg10 = #map2(%arg7) to min #map3(%arg7)[%0]
    affine.for %arg11 = #map2(%arg8) to min #map4(%arg10, %arg8)
     affine.for %arg12 = #map2(%arg9) to min #map3(%arg9)[%1]
      S1
```

Listing 6.2: Polyhedral optimized MLIR code generated by Polygeist for the loops shown in Listing 6.1.

an MLIR OpenMP SIMD loop, thereby emitting SIMD instructions for loop statements with the help of OpenMP.

6.2 OpenMP SIMD Code Generation

This section presents three major changes in CPU code generation to be able to generate SIMD code, as follows:

① Identify which loop(s) can be vectorized from the Pluto output,

② Annotate those `affine.loop`'s with the attribute `scop.vector`,

③ A new optimization pass that converts those marked loops to `omp.simdloop`. The pass should compute the arguments of the simd constructs: (i) size of the vector (`simdlen`), (ii) the step value between the loop iterations, and (iii) the lower/upper bound loop variables.

The green compilation flow in Figure 6.2 shows the modified/extended approach of our MLIR CPU code generation technique. We rely on the `affine` and `omp` dialects for simd loop creation. Listing 6.3 line 11 shows the `affine.for` loop in line 5 of Listing 6.2 is converted to `omp.simdloop` by our modified approach. Firstly, Pluto scheduler identifies the loops that could be vectorized and informs Polygeist. We modified *cgeist* so that it annotates the vectorizable loop(s) with the `scop.vector` attribute. Our simd converter pass then converts those annotated loops to `omp.simd`

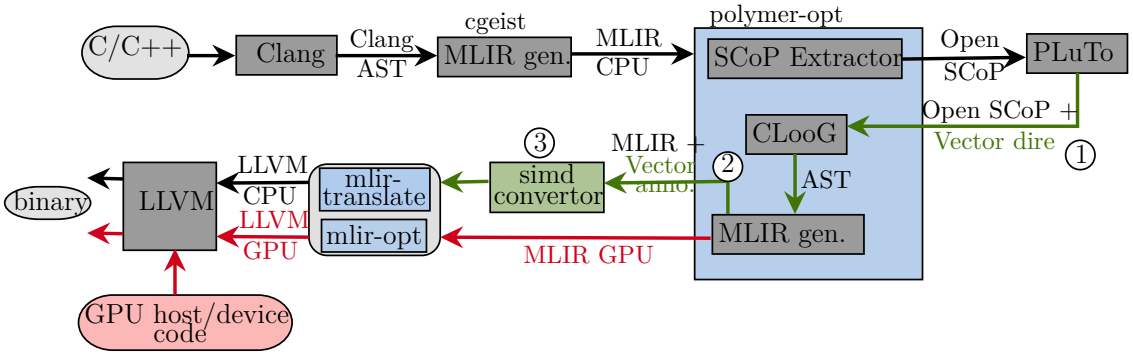


Figure 6.2: Overview of Polygeist compilation-flow being modified by our proposed technique. Green arrows show our extended/modified version of CPU code generation. Red arrows show our newly proposed GPU code generation

```
1  affine.for %arg7 = 0 to #map()[%0]
2   affine.for %arg8 = 0 to #map1(%arg7)
3    affine.for %arg9 = #map2(%arg7) to min #map3(%arg7)[%0]
4     %c1 = arith.constant 1 : index
5     %2 = arith.muli %arg8, %c32 : index
6     %3 = arith.addi %arg9, %c1 : index
7     %4 = arith.muli %arg8, %c32 : index
8     %5 = arith.addi %4, %c32 : index
9     %6 = arith.cmpi slt, %3, %5 : index
10    %7 = arith.select %6, %3, %5 : index
11    omp.simdloop simdlen(4) for (%arg10):index = (%2) to (%7) step (%c1){
12     S0
13     omp.yield
14    }
15 // ... code of S1 skipped for space
```

Listing 6.3: OpenMP SIMD loop generation by our proposed technique for vector annotated loop(s) from Listing 6.2.

loop. Using an environmental variable the pass allows the user to choose the `simdlen` (set to either 2, 4, 8, or 16 targeting `SSE`, `AVX2`, or `AVX512` vector architectures). In line 11 of Listing 6.3, the simd length is set to 4 targeting `AVX2` architecture with using `double` types. The `step` value is set to the default value of one. Arguments `%2` and `%7` in line 11 are the lower and upper bound values of the simd loop, respectively. In Listing 6.2, the lower and upper bounds are expressed as affine maps (#map2 and #map4), such expression is not defined in the OpenMP dialect. Therefore, we have to substitute the map definition with arithmetic operations as shown in lines 4-10 of Listing 6.3. Finally, the Polygeist existing compilation flow will lower the simd loop to LLVM IR.

6.3 Preliminary Experiments

We implemented the proposed CPU compilation flow on top of the Polygeist source from the git repository version `18.0.0`. We used a 2x20-cores CascadeLake Intel Xeon Gold 5218R @2.1GHz CPU and `AVX2` vector architecture set for our evaluation. We choose six benchmarks (*2mm*, *syr2k*, *gramschmidt*, *correlation*, *nussinov*, and *heat-3D*), one from each category of PolyBench/C [14] with `EXTRALARGE` data-set for evaluating

our implementation for CPU code generation. Unfortunately, the OpenMP SIMD optimized MLIR code took the same execution time as the code without our optimization and does not improve the use of the vector instructions. Indeed, the `omp.simd` loop was not vectorized. Since this functionality is newly added to MLIR, it has limited support. The MLIR framework is undergoing very active development phases to support various compiler optimizations. So, in a very near future MLIR could provide complete optimization support for OpenMP SIMD loops and we hope that our proposed technique improves vectorization.

Another work-around would be to emit the vector instructions directly with the help of MLIR's `vector` dialect instead of relying on SIMD loops, but this would duplicate the work that `omp.simd` should do unless the loop contains complex statements (like function calls, irregular control flow, and pointer/array deference).

6.4 Discussion

Using the multiple code generation capabilities of MLIR, we will further extend Polygeist (red compilation flow in Figure 6.2) to implement the polyhedral GPU MLIR code generation for nested loop structures. We propose to generate MLIR GPU code for the loops enclosed within `scop` and let the rest of the code be in `C/C++`. The generated MLIR GPU code will be enclosed within two nested `scf.for` loops which are later lowered into an outer and an inner loop iterating over the GPU `blocks` and `threads`, respectively. The MLIR GPU code is first lowered to its GPU specific low-level dialect. This can be for example either `nvvm` (the Nvidia CUDA IR) or `rocdl` (the AMD ROCm IR) with respect to the target GPU architecture. Next is the MLIR translation pass that converts the low level MLIR code (`llvm` dialect) into an LLVM IR representation. The last step is the linking phase, where the `C/C++` emitted LLVM IR host code with the help of `clang` and the MLIR translated optimized LLVM IR are linked together into an object file using the LLVM compiler framework.

One challenge we may face is the function or library calls inside the nested loops. An equivalent GPU code is required for those function calls to keep the computations within the GPU rather than making frequent and costly context switches between host and device. Another challenge is the memory management on GPU. We have to choose between the unified memory model or a manual allocation and data transfers between the host and device for efficient management. The implementation of GPU code generation is left as a future work.

Chapter 7

Conclusion and Perspectives

Summary of Contributions

We have presented how the quite recent *multi-level intermediate representation* (MLIR) concept that arose from the compilation research community can be applied to a production-level scientific application, namely openCARP, a cardiac electrophysiology simulator (Chapter 4). The challenge is to integrate into the existing code base the generation of highly optimized code for both CPU and GPU. We have explained the modifications we have brought to the code generation process which originally generated `C/C++` code from the ionic models expressed with a DSL. The thesis discusses the design choices that arise when it comes to choosing among the available dialects to represent the code structures and statements at the appropriate abstraction level. We show that we were able to factorize a large part of the MLIR-generated code that was used for vectorization in CPU, and we explained how the necessary additions to generate GPU code are implemented through the lowering passes. Finally, MLIR allows us to produce code that has the same simulation results as native code but in a more portable way to various hardware targets. The evaluation of our vectorized CPU and GPU version is carried out on the full set of ionic models shipped with openCARP and brings a significant performance improvement over the baseline non-optimized version both in terms of execution time and energy efficiency (Section 4.2 Chapter 4). We have artifacts to reproduce our results for vectorized CPU [100] and GPU [101].

Moreover, we could see that our work has been extended by a research team, and taking advantage of the heterogeneous code generation nature of our techniques they integrated our approach with the StarPU task-based runtime system to allow dynamic load balancing and resource allocation. They reported good performance improvements by evaluating their methodology using the ionic models on multi-GPU and hybrid architectures. This work has resulted in a joint publication [13].

Building up on this achievement regarding performance improvements brought to the ionic models generated code in openCARP, we wanted to extend our findings to the optimization of loop nests in the polyhedral framework. In the last decades, the

research community has developed a rich set of techniques and tools to transform regular loop nests so as to extract potential parallelism and tiled loops to improve data cache locality. We wanted to explore the possibility to use the polyhedral analysis results and extend it to generate MLIR code. As there are many polyhedral compilers available with differences in nature we started a survey of these polyhedral compilers to find out if we could have a chance to make a difference. In this survey, we have presented an up-to-date study on general-purpose static polyhedral compilers and their optimizations on the PolyBench/C set of benchmarks. We have compared eight different general-purpose polyhedral compilers, compiled the generated codes using three standard compilers, and reported their failures and benefits (Chapter 5).

Overall, even if polyhedral techniques can dramatically improve performance on some of the benchmarks, especially when loop tiling results in optimal data locality and use of cache, the loop restructuring leads to side effects that are difficult to predict. For instance, we observed that this restructuring favors auto-vectorization by the standard compiler in some cases while it hinders it in other cases. In automatic parallelization, many of the benchmarks effectively used the multi-core hardware, but there are also cases where (i) the degree of parallelism is low and (ii) the large number of fork-join regions (entering and exiting a parallel region) leads to synchronization overheads. The polyhedral compilers would benefit from an improved code generation, making use of a wider range of OpenMP directives. Scheduling algorithms explicitly targeting vectorization and explicit vector directives in the generated code would also be useful to generate more efficient code.

Based on this study, we identified Polygeist to be the most appropriate polyhedral framework to extend, in order to bring in the techniques developed within openCARP. The principal argument is that Polygeist already produces its intermediate representation in MLIR. In Chapter 6 we have presented our efforts to extend Polygeist so that identified (by Pluto) vectorizable computations inside loop nests can be explicitly expressed as such by emitting MLIR code that can further translate to OpenMP SIMD instructions. Though our evaluations do not show performance improvements because of the limited support for the OpenMP SIMD loop by MLIR, we hope the rapid ongoing development of MLIR will quickly address this issue and will enable to evaluate the potential benefits.

Perspectives

Other vector architectures. The vectorized CPU code generation studied in this thesis targets only x86 architecture. However, there have been important developments in the recent years regarding vector architectures. For instance, the ARM SVE architecture [102] offers registers with variable length allowing implementations to choose a vector register length between 128 and 2048 bits. Similarly, the RISC-V architecture through its RVV extension offers a vector register than can contain from 64 or 128 to 65,536 bits. MLIR already provides support for the ARM SVE architecture with

the `arm_sve` dialect. We estimate that modifications of our existing compilation flow to add support for the ARM SVE vectors would hence only require a limited effort. For RISC-V architecture, we depend on the basic `vector` dialect and as of now would require additional passes to lower the MLIR code to target a RISC-V architecture.

Wider accelerator targets. Another possible area worth exploring is heterogeneous code generation. Extending our techniques to support the generation of OpenACC MLIR code using the `acc` dialect will help to target parallel multi-core and accelerator machines in general. Furthermore, if MLIR supports any new architecture, we can modify our code generator with minimal effort to provide support for those new architectures. And, that is the prime reason for having chosen the MLIR framework in this work.

MLIR as a standard. Broad adoption of MLIR requires standardizing certain aspects of its design to ensure compatibility and interoperability between different projects and tools. Encouraging widespread use and contributing to open-source initiatives can help in establishing MLIR as a standard in compiler infrastructure. While MLIR facilitates advanced optimizations, ensuring that these transformations lead to significant performance gains on various architectures remains an ongoing challenge. Continuous benchmarking, profiling, and tuning are necessary to maximize the benefits of MLIR in real-world applications.

Polygeist extension for GPU. In Chapter 6 (Figure 6.2) we proposed heterogeneous code generation targeting GPU architectures and left the implementation as a future work. The first priority would be to implement the compilation flow for GPU architectures targeting Nvida and AMD GPU machines. Another interesting aspect could be improving the support of OpenMP SIMD loops in the MLIR framework that would automatically improve the performance of our CPU SIMD code generation using Polygeist.

Extend polyhedral benchmarks. Another broad perspective would be the improvement of benchmark programs to evaluate various polyhedral compilers. The present PolyBench/C set of benchmarks is pretty old and outdated. The new set of benchmarks should have very large data sets targeting supercomputers and should include benchmarks to evaluate ML/AI-related applications.

Improve polyhedral scheduling. Recent advancements [73, 75] explore integrating machine learning techniques in polyhedral compilation to predict the most effective transformations for a given code segment. This integration aims to further automate the optimization process and adapt to diverse hardware architectures dynamically. Polyhedral compilation intersects with various fields such as computational geometry, linear algebra, and combinatorial optimization. Collaborative research across these

disciplines can lead to novel optimization techniques and broader applicability of the polyhedral model.

The mathematical complexity of the polyhedral model can be seen as a barrier to its wider adoption. Simplifying the model without losing its expressiveness is an ongoing challenge. Extending support for polyhedral compilation in modern programming languages and paradigms, such as functional and reactive programming, is also an interesting direction.

Bibliography

Personal Bibliography

[10] Arun Thangamani, Tiago Trevisan Jost, Vincent Loechner, Stéphane Genaud, and Bérenger Bramas. "Lifting Code Generation of Cardiac Physiology Simulation to Novel Compiler Technology". In: *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization (CGO)*. 2023, pp. 68–80. DOI: 10.1145/3579990.3580008.

[11] Tiago Trevisan Jost, Arun Thangamani, Raphaël Colin, Vincent Loechner, Stéphane Genaud, and Bérenger Bramas. "GPU Code Generation of Cardiac Electrophysiology Simulation with MLIR". In: *Proceedings of the 29th International Conference on Parallel and Distributed Computing (Euro-Par)*. Limassol, Cyprus, 2023, pp. 549–563. DOI: 10.1007/978-3-031-39698-4_37.

[13] Vincent Alba, Olivier Aumage, Denis Barthou, Raphael Colin, Marie-Christine Counilh, Stephane Genaud, Amina Guermouche, Vincent Loechner, and Arun Thangamani. "Performance Portability of Generated Cardiac Simulation Kernels Through Automatic Dimensioning and Load Balancing on Heterogeneous Nodes". In: *25th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC)*. 2024. DOI: 10.1109/IPDPSW63119.2024.00171.

[16] Arun Thangamani, Vincent Loechner, and Stéphane Genaud. "A Survey of General-purpose Polyhedral Compilers". In: *ACM Trans. Archit. Code Optim.* (June 2024). DOI: 10.1145/3674735.

[17] Arun Thangamani, Vincent Loechner, and Stéphane Genaud. "Extending Polygeist to Generate Efficient Vectorized and GPU MLIR Code". In: *PhD Symposium in Proceedings of the 30th International Conference on Parallel and Distributed Computing (Euro-Par)*. 2024.

Artifact Bibliography

[100] Arun Thangamani, Tiago Trevisan Jost, Vincent Loechner, Stéphane Genaud, and Bérenger Bramas. *Artifact for Lifting Code Generation of Cardiac Physiology Simulation to Novel Compiler Technology*. 2023. URL: https://doi.org/10.1145/3554349.

[101] Tiago Trevisan Jost, Arun Thangamani, Raphaël Colin, Vincent Loechner, Stéphane Genaud, and Bérenger Bramas. *Artifact for GPU Code Generation of Cardiac Electrophysiology Simulation with MLIR*. 2023. URL: https://doi.org/10.6084/m9.figshare.23546157.

General Bibliography

[1] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. "X10: an object-oriented approach to non-uniform cluster computing". In: *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA '05. San Diego, CA, USA, 2005, pp. 519–538. DOI: 10.1145/1094811.1094852.

[2] Arun Thangamani and V Krishna Nandivada. "Optimizing remote data transfers in X10". In: *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*. PACT '18. Limassol, Cyprus, 2018. DOI: 10.1145/3243176.3243209.

[3] Arun Thangamani and V. Krishna Nandivada. "Optimizing Remote Communication in X10". In: *ACM Trans. Archit. Code Optim.* 16.4 (Oct. 2019). DOI: 10.1145/3345558.

[4] B.L. Chamberlain, D. Callahan, and H.P. Zima. "Parallel Programmability and the Chapel Language". In: *Int. J. High Perform. Comput. Appl.* 21.3 (Aug. 2007), pp. 291–312. DOI: 10.1177/1094342007078442.

[5] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. "The implementation of the Cilk-5 multithreaded language". In: PLDI '98. Montreal, Quebec, Canada, 1998, pp. 212–223. DOI: 10.1145/277650.277725.

[6] Jeff Meyerson. "The Go Programming Language". In: *IEEE Software* 31.5 (2014), pp. 104–104. DOI: 10.1109/MS.2014.127.

[7] 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 Int. Symp. on Code Generation and Optimization (CGO)*. 2021, pp. 2–14. DOI: 10.1109/CGO51591.2021.9370308.

[8] Chris Lattner and Vikram Adve. “LLVM: a compilation framework for lifelong program analysis & transformation”. In: *Int. Symp. on Code Generation and Optimization, (CGO)*. 2004, pp. 75–86. DOI: 10.1109/CGO.2004.1281665.

[9] Gernot Plank, Axel Loewe, Aurel Neic, Christoph Augustin, Yung-Lin Huang, Matthias A.F. Gsell, Elias Karabelas, Mark Nothstein, Anton J. Prassl, Jorge Sánchez, Gunnar Seemann, and Edward J. Vigmond. “The openCARP simulation environment for cardiac electrophysiology”. In: *Computer Methods and Programs in Biomedicine* 208 (2021), p. 106223. DOI: 10.1016/j.cmpb.2021.106223.

[10] Arun Thangamani, Tiago Trevisan Jost, Vincent Loechner, Stéphane Genaud, and Bérenger Bramas. “Lifting Code Generation of Cardiac Physiology Simulation to Novel Compiler Technology”. In: *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization (CGO)*. 2023, pp. 68–80. DOI: 10.1145/3579990.3580008.

[11] Tiago Trevisan Jost, Arun Thangamani, Raphaël Colin, Vincent Loechner, Stéphane Genaud, and Bérenger Bramas. “GPU Code Generation of Cardiac Electrophysiology Simulation with MLIR”. In: *Proceedings of the 29th International Conference on Parallel and Distributed Computing (Euro-Par)*. Limassol, Cyprus, 2023, pp. 549–563. DOI: 10.1007/978-3-031-39698-4_37.

[12] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. “StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures”. In: *CCPE - Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009* 23 (2 Feb. 2011), pp. 187–198. DOI: 10.1002/cpe.1631.

[13] Vincent Alba, Olivier Aumage, Denis Barthou, Raphael Colin, Marie-Christine Counilh, Stephane Genaud, Amina Guermouche, Vincent Loechner, and Arun Thangamani. “Performance Portability of Generated Cardiac Simulation Kernels Through Automatic Dimensioning and Load Balancing on Heterogeneous Nodes”. In: *25th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC)*. 2024. DOI: 10.1109/IPDPSW63119.2024.00171.

[14] Louis-Noël Pouchet and Tomofumi Yuki. *PolyBench/C version 4.2.1-beta*. 2022. URL: http://polybench.sf.net.

[15] William S. Moses, Lorenzo Chelini, Ruizhe Zhao, and Oleksandr Zinenko. “Polygeist: Raising C to Polyhedral MLIR”. In: *30th Int. Conf. on Parallel Architectures and Compilation Techniques (PACT)*. 2021, pp. 45–59. DOI: 10.1109/PACT52795.2021.00011.

[16] Arun Thangamani, Vincent Loechner, and Stéphane Genaud. “A Survey of General-purpose Polyhedral Compilers”. In: *ACM Trans. Archit. Code Optim.* (June 2024). DOI: 10.1145/3674735.

[17] Arun Thangamani, Vincent Loechner, and Stéphane Genaud. “Extending Polygeist to Generate Efficient Vectorized and GPU MLIR Code”. In: *PhD Symposium in Proceedings of the 30th International Conference on Parallel and Distributed Computing (Euro-Par)*. 2024.

[18] Hodgkin A. L. and Huxley A. F. “A quantitative description of membrane current and its application to conduction and excitation in nerve”. In: *The Journal of physiology* 117(4) (), pp. 500–544. DOI: 10.1113/jphysiol.1952.sp004764.

[19] Biplab Biswas, S. Chatterjee, S. Mukherjee, and S. Pal. “A discussion on Euler method: A review”. In: *Electronic Journal of Mathematical Analysis and Applications*. 2013. URL: https://api.semanticscholar.org/CorpusID:124717919.

[20] Ernst Hairer, Syvert P. Norsett, and Gerhard Wanner. *Solving Ordinary Differential Equations I. Nonstiff Problems*. eng. 2nd rev. ed. 1993. Corr. 3rd printing. ID: unige:12346. Berlin: Springer, 1993. URL: https://archive-ouverte.unige.ch/unige:12346.

[21] Stanley Rush and Hugh Larsen. “A Practical Algorithm for Solving Dynamic Membrane Equations”. In: *IEEE Transactions on Biomedical Engineering* BME-25.4 (1978), pp. 389–392. DOI: 10.1109/TBME.1978.326270.

[22] Megan E. Marsh, Saeed Torabi Ziaratgahi, and Raymond J. Spiteri. “The Secrets to the Success of the Rush–Larsen Method and its Generalizations”. In: *IEEE Transactions on Biomedical Engineering* 59.9 (2012), pp. 2506–2515. DOI: 10.1109/TBME.2012.2205575.

[23] Joakim Sundnes, Robert Artebrant, Ola Skavhaug, and Aslak Tveito. “A Second-Order Algorithm for Solving Dynamic Cell Membrane Equations”. In: *IEEE Transactions on Biomedical Engineering* 56.10 (2009), pp. 2546–2548. DOI: 10.1109/TBME.2009.2014739.

[24] Catherine M Lloyd, Matt DB Halstead, and Poul F Nielsen. “CellML: its future, present and past”. In: *Progress in biophysics and molecular biology* 85.2-3 (2004), pp. 433–450. DOI: 10.1016/j.pbiomolbio.2004.01.004.

[25] Edward Vigmond. *EasyML*. https://opencarp.org/documentation/examples/01_ep_single_cell/05_easyml. 2021.

[26] Michael Hucka, Andrew Finney, Herbert M Sauro, Hamid Bolouri, John C Doyle, Hiroaki Kitano, Adam P Arkin, Benjamin J Bornstein, Dennis Bray, Athel Cornish-Bowden, et al. “The systems biology markup language (SBML): a medium for representation and exchange of biochemical network models”. In: *Bioinformatics* 19.4 (2003), pp. 524–531. DOI: 10.1093/bioinformatics/btg015.

[27] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. "An Efficient Method of Computing Static Single Assignment Form". In: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '89. Austin, Texas, USA: ACM, 1989, pp. 25–35. DOI: 10.1145/75277.75280.

[28] Michael Clerx, Pieter Collins, Enno de Lange, and Paul G.A. Volders. "Myokit: A simple interface to cardiac cellular electrophysiology". In: *Progress in Biophysics and Molecular Biology* 120.1 (2016), pp. 100–114. DOI: 10.1016/j.pbiomolbio.2015.

[29] Satish Balay and et al. *PETSc Web page*. 2022. URL: https://petsc.org/.

[30] Hartwig Anzt, Terry Cojean, Goran Flegar, Fritz Göbel, Thomas Grützmacher, Pratik Nayak, Tobias Ribizel, Yuhsiang Mike Tsai, and Enrique S. Quintana-Ortıé. "Ginkgo: A Modern Linear Operator Algebra Framework for High Performance Computing". In: *ACM Trans. Math. Softw.* 48.1 (Feb. 2022). DOI: 10.1145/3480935.

[31] 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 Trans. Archit. Code Optim.* (Sept. 2021). DOI: 10.1145/3469030.

[32] Lukas Sommer, Cristian Axenie, and Andreas Koch. "SPNC: An Open-Source MLIR-Based Compiler for Fast Sum-Product Network Inference on CPUs and GPUs". In: *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2022, pp. 1–11. DOI: 10.1109/CGO53902.2022.9741277.

[33] Keshav Santhanam, Siddharth Krishna, Ryota Tomioka, Andrew Fitzgibbon, and Tim Harris. "DistIR: An Intermediate Representation for Optimizing Distributed Neural Networks". In: *Proceedings of the 1st Workshop on Machine Learning and Systems*. 2021, pp. 15–23. DOI: 10.1145/3437984.3458829.

[34] Martin Lücke, Michel Steuwer, and Aaron Smith. "Integrating a functional pattern-based IR into MLIR". In: *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*. 2021, pp. 12–22.

[35] Erdal Mutlu, Ruiqin Tian, Bin Ren, Sriram Krishnamoorthy, Roberto Gioiosa, Jacques Pienaar, and Gokcen Kestor. "Comet: A domain-specific compilation of high-performance computational chemistry". In: *International Workshop on Languages and Compilers for Parallel Computing*. Springer. 2020, pp. 87–103.

[36] Ryan Kabrick, Diego A Roa Perdomo, Siddhisanket Raskar, Jose M Monsalve Diaz, Dawson Fox, and Guang R Gao. "CODIR: towards an MLIR codelet model dialect". In: *2020 IEEE/ACM Fourth Annual Workshop on Emerging Parallel and Distributed Runtime Systems and Middleware (IPDRM)*. IEEE. 2020, pp. 33–40.

[37] 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 2022. Seoul, South Korea, 2022, pp. 117–128. DOI: 10.1145/3497776.3517770.

[38] Uday Bondhugula. *High Performance Code Generation in MLIR: An Early Case Study with GEMM*. 2020. arXiv: 2003.00532 [cs.PF].

[39] Nicolas Vasilache, Oleksandr Zinenko, Aart J. C. Bik, Mahesh Ravishankar, Thomas Raoux, Alexander Belyaev, Matthias Springer, Tobias Gysi, Diego Caballero, Stephan Herhut, Stella Laurenzo, and Albert Cohen. “Composable and Modular Code Generation in MLIR: A Structured and Retargetable Approach to Tensor Compiler Construction”. In: *CoRR* abs/2202.03293 (2022). URL: https://arxiv.org/abs/2202.03293.

[40] Nicolas Weber and Michael Goesele. “Auto-tuning complex array layouts for GPUs”. In: *Proceedings of the 14th Eurographics Symposium on Parallel Graphics and Visualization*. PGV ’14. Swansea, Wales, United Kingdom: Eurographics Association, 2014, pp. 57–64. DOI: 10.2312/pgv.20141085.

[41] Klaus Kofler, Biagio Cosenza, and Thomas Fahringer. “Automatic data layout optimizations for gpus”. In: *European Conference on Parallel Processing*. Springer. 2015, pp. 263–274. URL: https://api.semanticscholar.org/CorpusID:2597244.

[42] Samuel Williams, Andrew Waterman, and David Patterson. “Roofline: an insightful visual performance model for multicore architectures”. In: *Commun. ACM* 52.4 (Apr. 2009), pp. 65–76. DOI: 10.1145/1498765.1498785.

[43] Paul Feautrier and Christian Lengauer. “Polyhedron Model”. In: *Encyclopedia of Parallel Computing*. Ed. by David Padua. Boston, MA, 2011, pp. 1581–1592. DOI: 10.1007/978-0-387-09766-4_502.

[44] Paul Feautrier. “Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time”. In: *International Journal of Parallel Programming* 21.6 (Dec. 1992), pp. 389–420. DOI: 10.1007/BF01379404.

[45] *PIP - The Parametric Integer Programming’s Home*. http://www.piplib.org/. 2009.

[46] William Pugh. “A Practical Algorithm for Exact Array Dependence Analysis”. In: *Commun. ACM* 35.8 (Aug. 1992), pp. 102–114. DOI: 10.1145/135226.135233.

[47] Arjun Pitchanathan, Christian Ulmann, Michel Weber, Torsten Hoefler, and Tobias Grosser. “FPL: fast Presburger arithmetic through transprecision”. In: *Proc. ACM Program. Lang.* 5.OOPSLA (Oct. 2021). DOI: 10.1145/3485539.

[48] Polly Labs. *Integer Set Library*. https://www.openhub.net/p/isl. 2014.

[49] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. "A Practical Automatic Polyhedral Parallelizer and Locality Optimizer". In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '08. Tucson, AZ, USA, 2008, pp. 101–113. DOI: 10.1145/1375581.1375595.

[50] Cédric. Bastoul. "Code generation in the polyhedral model is easier than you think". In: *Proceedings. 13th International Conference on Parallel Architecture and Compilation Techniques (PACT),* 2004, pp. 7–16. DOI: 10.1109/PACT.2004.1342537.

[51] Chun Chen. "Polyhedra scanning revisited". In: *SIGPLAN Not.* 47.6 (June 2012), pp. 499–508. DOI: 10.1145/2345156.2254123.

[52] Sebastian Pop, Albert Cohen, Cédric Bastoul, Sylvain Girbal, Georges-André Silber, and Nicolas Vasilache. "GRAPHITE : Polyhedral Analyses and Optimizations for GCC". In: GCC Developer Summit, 2006. URL: https://www.cri.ensmp.fr/people/silber/docs/PopS2006b.pdf.

[53] Tobias Grosser, Armin Größlinger, and Christian Lengauer. "Polly - Performing Polyhedral Optimizations on a Low-Level Intermediate Representation." In: *Parallel Process. Lett.* 22.4 (2012). URL: https://doi.org/10.1142/S0129626412500107.

[54] Dan Quinlan and Chunhua Liao. "The ROSE source-to-source compiler infrastructure". In: *Cetus users and compiler infrastructure workshop, in conjunction with PACT*. Vol. 2011. Citeseer. 2011, p. 1. URL: https://engineering.purdue.edu/Cetus/cetusworkshop/papers/4-1.pdf.

[55] Louis-Noël Pouchet. *PoCC - The Polyhedral Compiler Collection*. http://web.cs.ucla.edu/~pouchet/software/pocc/. 2012.

[56] Michelle Mills Strout, Mary Hall, and Catherine Olschanowsky. "The Sparse Polyhedral Framework: Composing Compiler-Generated Inspector-Executor Code". In: *Proceedings of the IEEE* 106.11 (2018), pp. 1921–1934. DOI: 10.1109/JPROC.2018.2857721.

[57] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. "Polyhedral Parallel Code Generation for CUDA". In: 9.4 (Jan. 2013). DOI: 10.1145/2400682.2400713.

[58] Vincent Loechner. *PolyLib: a library of polyhedral functions*. http://icps.u-strasbg.fr/polylib/. 1999.

[59] *OpenScop - A Specification and a Library for Data Exchange in Polyhedral Compilation Tools*. http://icps.u-strasbg.fr/~bastoul/development/openscop/index.html. 2014.

[60] Sven Verdoolaege and Tobias Grosser. “Polyhedral Extraction Tool”. In: *2nd International Workshop on Polyhedral Compilation Techniques* (2012). URL: https://impact-workshop.org/impact2012/workshop_IMPACT/verdoolaege.pdf.

[61] *CLAN - A Polyhedral Representation Extraction Tool for C-Based High Level Languages*. http://icps.u-strasbg.fr/~bastoul/development/clan/index.html. 2014.

[62] *Candl - Data Dependence Analysis Tool in the Polyhedral Model*. http://icps.u-strasbg.fr/people/bastoul/public_html/development/candl/. 2014.

[63] Louis-Noël Pouchet and al. *PoCC version 1.6.0-alpha*. 2022. URL: http://pocc.sf.net.

[64] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, P. Sadayappan, and Nicolas Vasilache. “Loop Transformations: Convexity, Pruning and Optimization”. In: *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’11. Austin, Texas, USA, 2011, pp. 549–562. DOI: 10.1145/1926385.1926449.

[65] Louis-Noël Pouchet and P. Sadayappan. *PolyOpt: A Framework for Polyhedral Optimization in the ROSE compiler*. 2022. URL: https://github.com/rose-compiler/rose/tree/v0.11.100.0/projects/PolyOpt2.

[66] Marek Palkowski, Tomasz Klimek, and Wlodzimierz Bielecki. “TRACO: An automatic loop nest parallelizer for numerical applications”. In: *2015 Federated Conference on Computer Science and Information Systems (FedCSIS)*. 2015, pp. 681–686. DOI: 10.15439/2015F34.

[67] William Pugh and Evan Rosser. “Iteration Space Slicing and Its Application to Communication Optimization”. In: *Proceedings of the 11th International Conference on Supercomputing*. ICS ’97. Vienna, Austria, 1997, pp. 221–228. DOI: 10.1145/263580.263637.

[68] GCC Compiler developers. *GCC - The GNU Compiler Collection*. https://gcc.gnu.org/. 2011.

[69] Jan Sjödin, Sebastian Pop, Harsha Jagasia, Tobias Grosser, and Antoniu Pop. “Design of graphite and the Polyhedral Compilation Package”. In: GCC Developer Summit, 2009. URL: https://minesparis-psl.hal.science/hal-00408759/fr/.

[70] Benoit Meister, Nicolas Vasilache, David Wohlford, Muthu Manikandan Baskaran, Allen Leung, and Richard Lethin. “R-Stream Compiler”. In: *Encyclopedia of Parallel Computing*. Ed. by David Padua. Boston, MA, 2011, pp. 1756–1765. DOI: 10.1007/978-0-387-09766-4_515.

[71] Meta (Facebook) Research. *Tensor Comprehensions*. https://github.com/facebookresearch/TensorComprehensions. 2018.

[72] 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*. 2018. arXiv: 1802.04730 [cs.PL].

[73] Huawei Technologies. *MindSpore AI computing Framework*. https://github.com/mindspore-ai/mindspore. 2023.

[74] Jie Zhao, Bojie Li, Wang Nie, Zhen Geng, Renwei Zhang, Xiong Gao, Bin Cheng, Chen Wu, Yun Cheng, Zheng Li, Peng Di, Kun Zhang, and Xuefeng Jin. "AKG: Automatic Kernel Generation for Neural Processing Units Using Polyhedral Transformations". In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2021. Virtual, Canada, 2021, pp. 1233–1248. DOI: 10.1145/3453483.3454106.

[75] Sven Verdoolaege, Manjunath Kudlur, Rob Schreiber, and Harinath Kamepalli. "Generating SIMD Instructions for Cerebras CS-1 using Polyhedral Compilation Techniques". In: 10th International Workshop on Polyhedral Compilation Techniques, Jan. 2020. URL: https://cerebras.net/wp-content/uploads/2021/04/IMPACT_2020_paper_3.pdf.

[76] 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: *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*. CGO 2019. Washington, DC, USA: IEEE Press, 2019, pp. 193–205. URL: https://doi.ieeecomputersociety.org/10.1109/CGO.2019.8661197.

[77] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. *Tiramisu Compiler*. https://github.com/Tiramisu-Compiler/tiramisu. 2019.

[78] PolyMage lab developers. *PolyMage*. https://www.polymagelabs.com/. 2023.

[79] Philippe Clauss, Matthew Wahab, Juan Manuel Martinez Caamano, Aravind Sukumaran-Rajam, Artiom Baloian, Manuel Selva, and Raquel Lazcano. *APOLLO: Automatic speculative POLyhedral Loop Optimizer*. https://webpages.gitlabpages.inria.fr/apollo/about. 2017.

[80] Tim Zerrell and Jeremy Bruestle. *Stripe: Tensor Compilation via the Nested Polyhedral Model*. 2019. arXiv: 1903.06498 [cs.DC].

[81] Matthew Arnold, Michael Hind, and Barbara G. Ryder. "An Empirical Study of Selective Optimization". In: *Languages and Compilers for Parallel Computing*. Ed. by Samuel P. Midkiff, José E. Moreira, Manish Gupta, Siddhartha Chatterjee, Jeanne Ferrante, Jan Prins, William Pugh, and Chau-Wen Tseng. 2001, pp. 49–67. DOI: 10.1007/3-540-45574-4_4.

[82] Zhide Zhou, Zhilei Ren, Guojun Gao, and He Jiang. “An empirical study of optimization bugs in GCC and LLVM”. In: *Journal of Systems and Software* 174 (2021), p. 110884. DOI: https://doi.org/10.1016/j.jss.2020.110884.

[83] Xiaolei Ren, Michael Ho, Jiang Ming, Yu Lei, and Li Li. “Unleashing the Hidden Power of Compiler Optimization on Binary Code Difference: An Empirical Study”. In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2021. Virtual, Canada, 2021, pp. 142–157. DOI: 10.1145/3453483.3454035.

[84] Pierre-Yves Péneau, Rabab Bouziane, Abdoulayse Gamatié, Erven Rohou, Florent Bruguier, Gilles Sassatelli, Lionel Torres, and Sophiane Senni. “Loop optimization in presence of STT-MRAM caches: A study of performance-energy tradeoffs”. In: *2016 26th International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS)*. 2016, pp. 162–169. DOI: 10.1109/PATMOS.2016.7833682.

[85] Amir H. Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. “A Survey on Compiler Autotuning Using Machine Learning”. In: *ACM Comput. Surv.* 51.5 (Sept. 2018). DOI: 10.1145/3197978.

[86] Paul B. Schneck. “A Survey of Compiler Optimization Techniques”. In: *Proceedings of the ACM Annual Conference*. ACM ’73. Atlanta, Georgia, USA, 1973, pp. 106–113. DOI: 10.1145/800192.805690.

[87] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. “A Survey of Compiler Testing”. In: *ACM Comput. Surv.* 53.1 (Feb. 2020). DOI: 10.1145/3363562.

[88] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. “Compiler Transformations for High-Performance Computing”. In: *ACM Comput. Surv.* 26.4 (Dec. 1994), pp. 345–420. DOI: 10.1145/197405.197406.

[89] Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, Lin Gan, Guangwen Yang, and Depei Qian. “The Deep Learning Compiler: A Comprehensive Survey”. In: *IEEE Transactions on Parallel and Distributed Systems* 32.3 (2021), pp. 708–727. DOI: 10.1109/TPDS.2020.3030548.

[90] Saleh M. Alnaeli, Abdulkareem Alali, and Jonathan I. Maletic. “Empirically Examining the Parallelizability of Open Source Software System”. In: *Proceedings of the 2012 19th Working Conference on Reverse Engineering*. WCRE ’12. USA: IEEE Computer Society, 2012, pp. 377–386. DOI: 10.1109/WCRE.2012.47.

[91] Dheya Mustafa. “A Survey of Performance Tuning Techniques and Tools for Parallel Applications”. In: *IEEE Access* 10 (2022), pp. 15036–15055. DOI: 10.1109/ACCESS.2022.3147846.

[92] Sang-Ik Lee, Troy A. Johnson, and Rudolf Eigenmann. "Cetus – An Extensible Compiler Infrastructure for Source-to-Source Transformation". In: *Languages and Compilers for Parallel Computing*. Ed. by Lawrence Rauchwerger. 2004, pp. 539–553. DOI: 10.1007/978-3-540-24644-2_35.

[93] Andreas Simbürger, Sven Apel, Armin Größlinger, and Christian Lengauer. "The potential of polyhedral optimization: An empirical study". In: *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2013, pp. 508–518. DOI: 10.1109/ASE.2013.6693108.

[94] Philip Pfaffe, Tobias Grosser, and Martin Tillmann. "Efficient Hierarchical Online-Autotuning: A Case Study on Polyhedral Accelerator Mapping". In: *Proceedings of the ACM International Conference on Supercomputing*. ICS '19. Phoenix, Arizona, 2019, pp. 354–366. DOI: 10.1145/3330345.3330377.

[95] Romain Fontaine, Laure Gonnord, and Lionel Morel. "Polyhedral Dataflow Programming: A Case Study". In: *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. 2018, pp. 171–179. DOI: 10.1109/CAHPC.2018.8645947.

[96] NASA Advanced Supercomputing (NAS) Division. *NAS Parallel Benchmarks*. https://www.nas.nasa.gov/software/npb.html. -.

[97] Corinna G. Lee. *UTDSP Benchmark Suite*. https://www.eecg.utoronto.ca/~corinna/DSP/infrastructure/UTDSP.html. 1998.

[98] Lawrence Livermore National Laboratory. *Livermore Benchmarks*. https://netlib.org/benchmark/livermore. 1993.

[99] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. "Collecting Performance Data with PAPI-C". In: *Tools for High Performance Computing 2009*. Ed. by Matthias S. Müller, Michael M. Resch, Alexander Schulz, and Wolfgang E. Nagel. 2010, pp. 157–173. DOI: 10.1007/978-3-642-11261-4_11.

[100] Arun Thangamani, Tiago Trevisan Jost, Vincent Loechner, Stéphane Genaud, and Bérenger Bramas. *Artifact for Lifting Code Generation of Cardiac Physiology Simulation to Novel Compiler Technology*. 2023. URL: https://doi.org/10.1145/3554349.

[101] Tiago Trevisan Jost, Arun Thangamani, Raphaël Colin, Vincent Loechner, Stéphane Genaud, and Bérenger Bramas. *Artifact for GPU Code Generation of Cardiac Electrophysiology Simulation with MLIR*. 2023. URL: https://doi.org/10.6084/m9.figshare.23546157.

[102] Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martinez, Nathanaël Prémillieu, Alastair Reid, Alejandro Rico, and Paul Walker. "The ARM Scalable Vector Extension". In: *CoRR* abs/1803.06185 (2018). arXiv: 1803.06185. URL: http://arxiv.org/abs/1803.06185.

Appendix A

Execution Time of General-purpose Polyhedral Compilers

Benchm.	CUDA	OpenCL
linear-algebra/kernels		
atax	0.412	0.558
2mm	0.451	0.663
3mm	0.471	0.740
bicg	0.388	0.540
doitgen	1.730	2.752
mvt	0.400	0.560
linear-algebra/blas		
gemm	0.435	0.670
gemver	0.416	0.586
gesummv	0.427	0.580
symm	0.432	0.623
syr2k	0.562	0.664
syrk	0.495	0.639
trmm	0.641	0.780
linear-algebra/solvers		
cholesky	0.720	1.193
durbin	0.901	1.105
gschmidt	1.929	2.165
lu	0.968	1.647
ludcmp	963.1	833.5
trisolv	0.446	0.688
datamining		
correlation	0.450	0.680
covariance	0.489	1.041
medley		
deriche	0.522	0.945
nussinov	ERR	3.683
fwarshall	206.6	883.0
stencils		
adi	5.186	5.434
jacobi-1d	0.400	0.600
seidel-2d	1.532	1.670
fdtd-2d	0.789	1.659
jacobi-2d	0.668	1.498
heat-3d	0.867	2.095

Table A.1: Execution time in seconds of all PolyBench/C benchmarks with respect to PPCG's CUDA and OpenCL optimized code on an A100 GPU; ERR - error.

									Pluto(tile)			Pluto(l2tile)			Pluto(dia)			PoCC			PPCG			CHiLL		
Benchm.	**gcc**	**clang**	**icc**	**rose**	**GRA**	**Polly**	**P'gst**	**P'opt**	**gcc**	**clang**	**icc**	**gcc**	**clang**	**icc**	**gcc**	**clang**	**icc**	**gcc**	**clang**	**icc**	**gcc**	**clang**	**icc**	**gcc**	**clang**	**icc**
linear-algebra/kernels																										
atax	0.003	0.003	0.003	0.006	0.003	0.006	0.008	0.008	0.008	0.007	0.008	0.005	0.005	0.005	0.007	0.008	0.008	0.007	0.008	0.008	0.004	0.004	ERR	WR	WR	WR
2mm	8.254	12.73	0.722	19.48	7.997	7.364	2.468	8.937	3.923	6.756	4.147	13.60	12.45	0.723	3.927	6.806	4.204	2.170	5.020	3.607	3.504	6.802	4.973	3.877	7.306	6.928
3mm	9.143	19.62	1.201	32.79	9.537	9.630	5.189	14.91	5.824	5.693	7.633	16.41	20.59	1.171	5.838	5.666	7.847	3.019	3.891	4.920	5.484	7.623	7.449	5.715	10.42	9.951
bicg	0.003	0.002	0.002	0.009	0.003	0.006	0.009	0.008	0.007	0.008	0.008	0.005	0.005	0.005	0.008	0.008	0.008	0.007	0.008	0.008	0.008	0.011	0.011	0.003	0.003	0.003
doitgen	1.321	1.596	0.555	3.880	1.320	1.580	0.833	0.862	0.777	0.843	0.945	0.398	0.596	0.589	0.778	0.845	0.917	0.775	0.841	0.886	1.914	3.490	2.448	WR	WR	WR
mvt	0.036	0.042	0.019	0.066	0.048	0.033	0.062	0.054	0.045	0.044	0.061	0.059	0.055	0.053	0.044	0.044	0.061	0.053	0.044	0.056	0.022	0.034	0.041	0.063	0.056	0.058
linear-algebra/blas																										
gemm	6.065	6.148	6.521	6.597	6.123	2.003	2.802	2.636	1.690	2.028	2.203	5.821	6.016	0.662	1.688	2.059	2.179	1.786	2.027	3.174	3.496	6.620	4.795	6.952	10.56	7.742
gemver	0.047	0.052	0.029	0.081	0.059	WR	0.050	0.083	0.035	0.038	0.051	0.021	0.021	0.020	0.034	0.039	0.053	0.034	0.038	0.052	0.043	0.051	0.048	0.092	0.067	0.105
gesummv	0.009	0.009	0.009	0.019	0.009	0.017	0.022	0.013	0.015	0.013	0.013	0.010	0.009	0.009	0.015	0.013	0.013	0.013	0.014	0.013	0.017	0.017	0.018	WR	WR	WR
symm	12.42	12.63	12.56	12.43	12.97	12.43	13.01	WR	12.48	12.56	12.43	12.61	12.61	12.43	12.81	12.43	12.48	WR	WR	WR	5.039	6.645	5.274	WR	WR	WR
syr2k	17.39	17.95	6.550	18.98	17.74	5.310	7.770	4.055	2.782	2.292	2.801	6.482	6.061	6.519	2.809	2.143	2.541	2.820	2.111	2.463	2.187	2.423	2.496	5.254	6.495	6.280
syrk	6.403	6.210	2.870	6.579	6.282	3.517	4.998	2.089	1.927	1.132	1.305	2.709	2.473	2.499	1.978	1.350	1.321	2.063	1.281	1.295	1.072	1.222	1.293	4.647	6.223	5.471
trmm	7.138	7.515	6.083	7.199	7.193	3.704	1.331	1.055	1.090	0.970	1.227	2.187	2.223	2.219	1.088	0.962	1.243	4.324	3.954	3.840	9.585	3.677	3.967	5.566	5.588	4.369
linear-algebra/solvers																										
cholesky	12.16	11.97	12.28	12.12	12.10	11.96	7.877	ERR	7.631	8.218	6.909	12.21	12.02	12.19	7.625	8.142	6.741	7.700	8.262	6.963	3.338	6.475	6.717	8.093	8.253	ERR
durbin	0.005	0.004	0.004	0.005	0.005	0.004	0.011	0.005	0.005	0.004	0.004	0.005	0.004	0.004	0.005	0.004	0.004	WR	WR	WR	0.005	0.004	0.004	WR	WR	WR
gschmidt	15.10	15.49	14.71	18.46	14.93	17.40	8.078	13.25	9.127	13.73	10.72	15.19	15.61	14.93	8.555	13.37	10.86	6.297	6.080	6.247	12.09	13.76	12.42	ERR	ERR	ERR
lu	46.29	39.16	27.76	45.81	45.84	33.06	7.141	15.53	3.982	14.97	12.62	12.59	11.18	13.93	3.955	14.94	12.57	4.012	15.08	4.098	4.322	5.753	15.13	50.71	28.76	51.94
ludcmp	28.70	27.96	28.38	28.95	28.86	27.40	39.43	WR	28.61	27.48	28.86	28.78	28.34	29.01	28.87	28.32	29.08	ERR	ERR	ERR	12.67	12.71	12.77	ERR	ERR	ERR
trisolv	0.010	0.010	0.010	0.010	0.010	0.010	0.007	0.020	0.019	0.008	0.007	0.010	0.010	0.011	0.021	0.008	0.007	0.015	0.009	0.007	0.016	0.018	0.019	0.021	0.019	0.013
datamining																										
correlation	8.889	13.87	4.973	15.60	12.20	8.114	2.618	4.768	1.433	2.014	1.930	4.991	5.144	4.875	1.431	1.990	1.934	1.404	1.979	2.504	4.373	8.067	5.555	9.895	13.32	5.213
covariance	12.10	13.86	5.069	15.66	12.17	8.026	2.744	4.596	1.427	2.061	2.479	4.884	5.266	5.240	1.422	2.027	2.458	1.427	2.020	2.366	4.409	8.114	5.629	8.001	10.55	8.562
medley																										
deriche	1.505	0.790	0.579	1.522	1.504	1.073	0.993	WR	1.538	0.836	0.572	1.547	0.789	0.574	1.504	0.870	0.571	ERR	ERR	ERR	1.386	0.802	0.571	1.081	1.081	1.057
nussinov	33.56	31.48	40.36	35.42	34.61	31.47	21.09	39.33	21.56	21.98	23.76	35.08	31.42	41.91	21.53	22.08	23.16	ERR	ERR	ERR	62.70	53.95	44.67	ERR	ERR	ERR
fwarshall	109.6	112.9	669.2	109.5	109.8	113.1	233.9	99.90	74.54	451.7	154.6	109.5	112.8	668.5	71.66	451.2	153.8	71.70	449.4	153.8	61.61	153.5	158.8	93.18	94.83	113.8
stencils																										
adi	72.54	65.84	48.58	78.71	72.65	81.20	89.03	77.92	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	85.81	87.98	90.38	WR	WR	WR
jacobi-1d	0.002	0.004	0.002	0.002	0.002	0.004	0.008	0.009	0.008	0.006	0.002	0.009	0.005	0.002	0.010	0.007	0.005	0.009	0.006	0.006	0.007	0.007	0.007	0.023	0.023	0.036
seidel-2d	72.37	64.69	140.6	72.40	72.29	64.71	127.7	ERR	44.75	141.8	122.2	72.37	96.66	140.7	44.6	141.8	122.2	44.4	141.7	122.1	64.6	144.0	66.29	72.5	64.7	140.8
fdtd-2d	20.44	20.02	20.17	20.29	20.30	23.79	13.38	12.27	12.73	11.21	12.18	12.69	12.03	9.996	18.01	17.93	33.01	8.840	20.61	10.89	13.61	13.03	63.93	WR	WR	WR
jacobi-2d	19.32	19.21	18.92	19.49	18.64	57.48	19.37	21.03	19.10	17.99	18.23	21.13	20.02	18.12	24.29	22.12	41.07	21.12	19.70	8.65	23.40	24.71	22.90	55.42	74.96	83.52
heat-3d	27.51	25.93	51.73	28.25	26.89	25.91	17.70	47.07	18.26	17.36	17.85	19.45	18.84	19.88	TO	TO	TO	41.60	15.45	18.55	25.43	13.46	11.21	63.78	77.46	135.5

Table A.2: Sequential execution time in seconds of all PolyBench/C benchmarks with respect to baseline compilers and the polyhedral compilers on an Intel machine. GRA - GRAPHITE; P'gst - Polygeist; P'opt - PolyOpt; WR - wrong output; ERR - error; TO - timed-out.

					Pluto(tile)			Pluto(l2tile)			Pluto(dia)			PoCC			PPCG		
Benchm.	**GRA**	**Polly**	**P'gst**	**P'opt**	**gcc**	**clang**	**icc**	**gcc**	**clang**	**icc**	**gcc**	**clang**	**icc**	**gcc**	**clang**	**icc**	**gcc**	**clang**	**icc**
linear-algebra/kernels																			
atax	0.002	0.005	0.014	0.001	0.003	0.015	0.014	0.059	0.034	0.042	0.002	0.018	0.015	0.003	0.015	0.016	0.002	0.017	0.016
2mm	7.863	0.381	0.496	0.925	0.322	0.508	0.294	0.897	0.901	0.156	0.326	0.507	0.295	0.176	0.372	0.174	0.546	0.674	0.317
3mm	9.627	0.482	0.706	1.089	0.589	0.455	0.435	1.523	1.583	0.455	0.588	0.454	0.436	0.243	0.299	0.294	0.720	0.892	0.469
bicg	0.002	0.004	0.017	0.001	0.003	0.015	0.017	0.063	0.033	0.032	0.004	0.016	0.016	0.003	0.018	0.016	0.004	0.016	0.015
doitgen	1.328	1.644	17.35	2.866	2.961	0.930	1.438	444.0	115.4	143.2	2.853	0.943	1.467	2.967	0.997	1.200	2.062	1.153	0.939
mvt	0.047	0.007	0.024	0.005	0.009	0.022	0.020	0.011	0.026	0.019	0.009	0.021	0.020	0.007	0.021	0.019	0.009	0.025	0.019
linear-algebra/blas																			
gemm	5.427	0.100	0.301	0.108	0.101	0.150	0.133	0.381	0.381	0.399	0.101	0.150	0.133	0.101	0.152	0.131	0.373	0.439	0.243
gemver	0.078	WR	0.025	0.005	0.008	0.021	0.021	0.107	0.057	0.056	0.008	0.023	0.018	0.008	0.022	0.018	0.009	0.027	0.019
gesummv	0.010	0.004	0.018	0.002	0.004	0.021	0.010	0.003	0.015	0.013	0.004	0.016	0.014	0.004	0.016	0.016	0.004	0.016	0.013
symm	13.62	12.85	1.090	WR	13.85	12.52	13.72	15.05	12.80	15.11	15.01	12.48	15.75	WR	WR	WR	2.696	2.806	2.726
syr2k	21.40	0.304	0.694	0.259	0.345	0.251	0.190	0.621	0.707	0.535	0.352	0.249	0.187	0.342	0.264	0.189	0.442	0.214	0.175
syrk	6.718	0.206	0.499	0.135	0.264	0.129	0.108	0.434	0.236	0.235	0.265	0.130	0.107	0.269	0.131	0.120	0.275	0.126	0.102
trmm	20.29	WR	0.164	0.053	0.064	0.102	0.087	0.139	0.123	0.121	0.064	0.099	0.089	0.203	0.220	0.230	0.200	0.255	0.240
linear-algebra/solvers																			
cholesky	13.27	12.20	0.863	ERR	0.453	0.477	0.479	0.642	0.591	0.343	0.452	0.477	0.486	0.523	0.474	0.479	5467	1422	1498
durbin	0.005	0.004	1.369	0.006	0.005	0.004	0.004	0.005	0.004	0.004	0.005	0.004	0.004	WR	WR	WR	0.122	0.068	0.065
gschmidt	16.55	1.685	4.490	13.42	0.713	0.787	1.064	1.372	1.602	4.031	0.711	0.824	1.079	2.289	3.655	0.888	41.10	41.74	43.05
lu	45.00	1.982	WR	0.854	0.358	0.431	0.990	1.133	1.138	1.105	0.357	0.475	0.989	0.419	0.457	0.392	303.1	71.87	75.41
ludcmp	28.64	22.39	45.63	WR	28.69	27.99	46.98	28.70	28.29	45.84	28.67	28.20	38.72	ERR	ERR	ERR	12.64	12.69	12.70
trisolv	0.010	0.010	0.033	0.005	0.005	0.018	0.020	0.096	0.050	0.059	0.005	0.018	0.019	0.006	0.017	0.018	3.037	1.091	1.149
datamining																			
correlation	15.47	0.680	0.455	0.412	0.195	0.286	0.238	0.503	0.395	0.409	0.195	0.285	0.232	0.215	0.290	0.279	0.608	0.819	ERR
covariance	15.47	0.659	0.397	0.423	0.255	0.298	0.230	0.435	0.364	0.371	0.256	0.297	0.229	0.228	0.301	0.284	0.607	0.807	0.515
medley																			
deriche	0.883	1.444	0.220	WR	0.855	1.473	0.876	0.848	1.419	0.871	0.850	1.417	0.876	ERR	ERR	ERR	0.745	1.313	0.262
nussinov	30.66	81.82	6.177	39.12	1.028	5.563	0.998	0.901	4.480	0.643	1.032	5.769	0.997	ERR	ERR	ERR	733.7	776.4	ERR
fwarshall	149.4	112.6	45.7	32.57	33.18	42.92	31.79	854.4	456.3	329.6	36.13	40.93	30.16	35.08	42.75	31.82	64.79	158.3	159.5
stencils																			
adi	87.74	60.01	5.39	78.84	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	5.270	4.951	5.528
jacobi-1d	0.003	0.005	0.093	0.004	0.006	0.093	0.020	0.089	0.816	0.054	0.003	0.071	0.014	0.005	0.079	0.016	0.007	0.007	0.007
seidel-2d	101.4	64.73	10.03	ERR	2.870	10.79	4.422	1.891	3.116	2.735	2.860	10.76	4.421	2.721	11.12	4.424	64.88	143.7	66.33
fdtd-2d	20.80	21.19	1.317	1.084	1.705	3.195	1.660	1.652	2.281	1.618	1.571	2.210	3.195	1.083	4.240	1.361	12.76	12.80	69.86
jacobi-2d	19.68	5.450	1.237	1.956	1.860	3.717	1.708	2.320	2.985	2.199	1.778	1.861	2.429	1.649	3.435	0.836	23.40	25.00	24.03
heat-3d	27.29	27.81	7.020	19.83	17.83	24.65	11.54	4.523	5.188	4.579	TO	TO	TO	19.81	25.83	11.89	2285	1121	ERR

Table A.3: Parallel execution time in seconds of all PolyBench/C benchmarks with respect to the polyhedral compilers on an Intel machine. GRA - GRAPHITE; P'gst - Polygeist; P'opt - PolyOpt; WR - wrong output; ERR - error; TO - timed-out.

									Pluto(tile)			Pluto(l2tile)			Pluto(dia)			PoCC			PPCG			CHiLL		
Benchm.	**gcc**	**clang**	**icc**	**rose**	**GRA**	**Polly**	**P'gst**	**P'opt**	**gcc**	**clang**	**icc**	**gcc**	**clang**	**icc**	**gcc**	**clang**	**icc**	**gcc**	**clang**	**icc**	**gcc**	**clang**	**icc**	**gcc**	**clang**	**icc**
linear-algebra/kernels																										
atax	0.002	0.002	0.009	0.007	0.002	0.006	0.008	0.006	0.005	0.005	0.017	0.003	0.003	0.006	0.005	0.005	0.018	0.005	0.005	0.017	0.004	0.004	ERR	WR	WR	WR
2mm	7.371	16.35	9.255	19.73	7.380	3.393	1.886	8.090	2.891	4.678	9.973	11.147	10.925	9.260	2.890	4.593	9.992	1.832	4.205	9.587	3.332	3.325	13.74	3.359	9.795	15.68
3mm	9.187	27.29	15.32	32.29	9.298	5.268	4.152	13.12	4.899	4.169	19.90	14.30	22.01	15.31	4.899	4.164	20.10	2.518	3.207	15.087	5.328	4.609	22.46	5.401	15.86	33.05
bicg	0.002	0.002	0.006	0.016	0.002	0.003	0.009	0.006	0.005	0.005	0.017	0.003	0.003	0.006	0.005	0.005	0.017	0.005	0.005	0.017	0.007	0.007	0.016	0.002	0.002	0.006
doitgen	1.416	1.452	2.564	3.660	1.448	1.429	0.852	0.923	0.849	0.843	2.797	0.437	0.609	2.539	0.843	0.907	2.870	0.885	0.882	2.837	2.315	2.853	5.441	WR	WR	WR
mvt	0.020	0.033	0.024	0.042	0.034	0.028	0.038	0.023	0.029	0.031	0.063	0.033	0.034	0.040	0.029	0.031	0.068	0.038	0.028	0.061	0.017	0.028	0.097	0.035	0.034	0.037
linear-algebra/blas																										
gemm	3.645	4.683	8.390	5.066	3.699	1.851	2.380	2.277	1.480	1.789	9.008	3.671	4.672	8.452	1.482	1.797	8.962	1.519	1.797	8.227	3.035	3.128	12.79	8.650	12.13	14.91
gemver	0.029	0.041	0.050	0.051	0.042	WR	0.034	0.080	0.017	0.026	0.100	0.013	0.015	0.045	0.018	0.026	0.100	0.018	0.026	0.105	0.027	0.029	0.062	0.042	0.038	0.064
gesummv	0.005	0.005	0.010	0.052	0.005	0.010	0.018	0.015	0.014	0.012	0.029	0.007	0.006	0.013	0.014	0.012	0.029	0.013	0.012	0.029	0.009	0.010	0.031	WR	WR	WR
symm	17.12	13.88	20.96	16.77	17.66	13.78	12.65	WR	14.42	13.81	22.51	14.45	13.80	22.44	14.54	13.79	22.98	WR	WR	WR	6.588	8.441	9.604	WR	WR	WR
syr2k	11.73	15.94	9.142	15.90	11.94	3.124	4.980	4.018	2.947	1.499	9.299	3.991	4.231	9.034	2.934	1.502	9.292	2.961	1.508	9.431	1.797	1.579	10.77	5.537	7.389	9.471
syrk	4.562	7.451	5.343	7.372	4.600	1.823	4.788	2.193	1.412	0.922	5.289	2.477	2.191	5.357	1.401	0.920	5.288	1.298	0.914	5.387	0.983	0.956	6.935	5.293	7.378	8.430
trmm	8.488	10.028	9.912	8.500	8.494	4.766	1.319	0.858	1.325	1.120	3.689	2.052	1.870	3.773	1.315	1.130	3.686	3.625	5.033	5.873	13.00	4.781	5.944	6.600	8.148	8.176
linear-algebra/solvers																										
cholesky	11.17	17.37	17.29	11.27	11.12	17.37	6.680	ERR	6.764	9.004	12.61	11.44	17.39	17.57	6.764	8.964	12.46	6.689	8.964	12.44	3.788	7.651	12.02	5.621	5.940	ERR
durbin	0.006	0.004	0.011	0.005	0.005	0.004	0.010	0.005	0.005	0.004	0.013	0.005	0.004	0.013	0.005	0.004	0.013	WR	WR	WR	0.005	0.004	0.011	WR	WR	WR
gschmidt	19.54	19.53	21.45	19.76	19.61	16.05	5.059	12.06	7.689	9.969	15.91	19.38	19.47	21.24	7.604	10.074	16.07	5.050	4.645	20.15	16.83	17.00	18.82	ERR	ERR	ERR
lu	32.47	41.95	31.93	32.26	32.31	26.43	6.20	16.17	3.68	16.61	19.94	11.02	9.197	17.42	3.677	16.59	19.88	3.492	16.378	20.69	3.333	4.691	22.30	51.12	32.07	58.69
ludcmp	28.83	28.61	28.79	28.69	28.82	28.60	32.14	WR	28.70	28.58	28.77	28.69	28.49	28.82	28.85	28.54	28.93	ERR	ERR	ERR	16.36	16.51	16.68	ERR	ERR	ERR
trisolv	0.009	0.013	0.013	0.009	0.009	0.013	0.013	0.007	0.013	0.014	0.020	0.009	0.013	0.013	0.013	0.014	0.020	0.016	0.014	0.019	0.010	0.012	0.016	0.008	0.008	0.010
datamining																										
correlation	8.614	16.88	8.904	17.02	16.82	3.803	1.994	5.229	1.338	1.584	6.784	3.411	4.531	8.874	1.339	1.581	6.783	1.337	1.579	6.936	3.726	3.778	10.89	8.577	16.92	8.773
covariance	16.77	16.83	8.900	17.05	16.83	3.776	1.988	5.181	1.316	1.577	6.905	3.430	4.548	8.829	1.326	1.576	6.932	1.316	1.577	6.940	3.722	3.769	10.54	13.55	13.93	16.45
medley																										
deriche	0.530	0.702	0.710	0.554	0.529	1.072	0.487	WR	0.530	0.707	0.991	0.531	0.701	0.711	0.529	0.712	1.002	ERR	ERR	ERR	0.421	0.606	1.007	0.605	0.802	0.964
nussinov	38.55	56.42	36.96	38.40	38.52	56.04	29.32	40.64	18.77	30.54	18.92	38.51	56.50	37.24	18.95	30.54	18.77	ERR	ERR	ERR	84.26	70.14	48.70	ERR	ERR	ERR
fwarshall	114.7	138.2	304.0	114.6	114.6	137.8	104.1	103.9	75.00	283.1	252.0	114.6	137.8	301.4	74.90	281.2	252.4	74.59	281.5	256.3	74.43	94.86	250.8	98.48	131.0	108.4
stencils																										
adi	73.14	94.11	150.3	99.51	73.17	83.91	90.79	99.95	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	94.75	111.7	178.0	WR	WR	WR
jacobi-1d	0.002	0.005	ERR	0.002	0.002	0.005	0.008	0.008	0.008	0.011	0.015	0.013	0.011	0.015	0.010	0.011	0.010	0.009	0.011	0.015	0.011	0.011	0.014	0.021	0.023	0.051
seidel-2d	67.72	89.61	276.1	67.35	67.51	89.61	134.2	ERR	37.20	151.2	243.2	67.73	75.03	275.0	37.00	150.0	242.9	37.07	153.8	242.6	64.46	153.3	242.0	68.02	90.66	276.3
fdtd-2d	10.83	10.99	18.42	11.12	10.89	14.01	11.32	13.08	14.90	10.06	17.96	12.82	10.01	33.27	18.40	28.33	92.35	9.446	11.69	18.39	23.72	24.47	79.28	WR	WR	WR
jacobi-2d	11.36	10.83	24.49	10.98	11.44	55.63	15.08	17.86	16.45	14.98	24.62	20.84	13.67	24.67	22.31	16.95	70.64	17.71	15.21	25.16	23.75	19.02	29.23	51.59	73.23	111.3
heat-3d	12.30	11.46	52.39	12.52	12.33	11.44	13.61	39.19	13.54	10.65	53.53	15.24	8.465	52.54	TO	TO	TO	38.15	10.69	54.23	24.25	17.88	28.19	73.63	88.82	173.4

Table A.4: Sequential execution time in seconds of all PolyBench/C benchmarks with respect to baseline compilers and the polyhedral compilers on an AMD machine. GRA - GRAPHITE; P'gst - Polygeist; P'opt - PolyOpt; WR - wrong output; ERR - error; TO - timed-out.

					Pluto(tile)			Pluto(l2tile)			Pluto(dia)			PoCC			PPCG		
Benchm.	**GRA**	**Polly**	**P'gst**	**P'opt**	**gcc**	**clang**	**icc**	**gcc**	**clang**	**icc**	**gcc**	**clang**	**icc**	**gcc**	**clang**	**icc**	**gcc**	**clang**	**icc**
linear-algebra/kernels																			
atax	0.003	0.005	0.011	0.001	0.003	0.012	0.012	0.052	0.024	0.026	0.003	0.012	0.013	0.003	0.012	0.012	0.003	0.011	0.014
2mm	8.805	0.161	0.349	0.640	0.265	0.360	0.448	0.434	0.500	0.375	0.267	0.359	0.448	0.106	0.309	0.456	0.397	0.582	0.495
3mm	11.62	0.234	0.489	0.554	0.464	0.439	0.808	0.847	1.013	0.604	0.464	0.438	0.807	0.159	0.237	0.661	0.607	0.881	0.680
bicg	0.003	0.004	0.011	0.001	0.003	0.012	0.013	0.052	0.025	0.026	0.004	0.012	0.013	0.004	0.013	0.014	0.003	0.011	0.013
doitgen	1.486	1.455	12.08	1.986	2.004	0.696	0.922	364.2	104.0	109.2	2.002	0.707	0.957	2.023	0.633	0.921	1.802	1.120	0.867
mvt	0.034	0.006	0.014	0.005	0.006	0.013	0.014	0.005	0.014	0.013	0.007	0.013	0.015	0.005	0.015	0.014	0.007	0.016	0.017
linear-algebra/blas																			
gemm	3.762	0.069	0.247	0.090	0.073	0.122	0.329	0.343	0.331	0.348	0.073	0.123	0.328	0.071	0.120	0.331	0.319	0.448	0.408
gemver	0.047	WR	0.015	0.008	0.007	0.015	0.018	0.107	0.042	0.050	0.007	0.014	0.017	0.007	0.015	0.016	0.010	0.019	0.019
gesummv	0.006	0.003	0.010	0.000	0.003	0.011	0.013	0.002	0.011	0.011	0.003	0.011	0.013	0.003	0.011	0.012	0.003	0.011	0.011
symm	17.59	14.12	0.673	WR	14.83	13.94	23.28	14.47	13.90	21.00	14.40	13.82	21.93	WR	WR	WR	3.683	3.682	3.829
syr2k	12.20	0.147	0.274	0.185	0.295	0.150	0.408	0.494	0.423	0.519	0.294	0.150	0.405	0.310	0.148	0.407	0.306	0.136	0.468
syrk	5.321	0.096	0.238	0.105	0.274	0.080	0.245	0.286	0.230	0.302	0.274	0.080	0.247	0.222	0.081	0.246	0.201	0.087	0.290
trmm	8.70	WR	0.093	0.039	0.041	0.057	0.125	0.093	0.066	0.132	0.041	0.056	0.124	0.110	0.160	0.172	0.144	0.160	0.197
linear-algebra/solvers																			
cholesky	11.20	17.09	0.534	ERR	0.388	0.474	0.604	0.411	0.565	0.544	0.388	0.475	0.604	0.345	0.487	0.605	4152	1134	1146
durbin	0.005	0.004	1.129	0.005	0.005	0.004	0.013	0.005	0.004	0.013	0.005	0.005	0.013	WR	WR	WR	0.111	0.051	0.048
gschmidt	20.14	0.613	2.295	12.83	0.697	0.633	0.666	0.778	0.784	0.724	0.599	0.625	0.665	0.879	1.589	1.812	33.85	33.78	28.79
lu	32.42	0.506	WR	0.479	0.219	0.318	0.905	0.279	0.237	0.541	0.218	0.317	0.907	0.268	0.322	0.896	253.7	64.89	63.44
ludcmp	28.70	22.73	34.75	WR	28.75	28.64	28.78	28.69	28.54	28.73	28.85	28.54	28.79	ERR	ERR	ERR	16.36	16.38	16.63
trisolv	0.009	0.013	0.025	0.006	0.005	0.012	0.012	0.088	0.033	0.045	0.005	0.012	0.014	0.005	0.012	0.014	3.060	0.839	0.827
datamining																			
correlation	17.18	0.204	0.277	0.252	0.136	0.151	0.357	0.590	0.495	0.497	0.131	0.156	0.358	0.133	0.153	0.371	0.334	0.298	ERR
covariance	17.34	0.194	0.268	0.255	0.137	0.161	0.359	0.526	0.483	0.480	0.135	0.154	0.359	0.106	0.163	0.372	0.333	0.305	0.552
medley																			
deriche	0.535	0.831	0.036	WR	0.514	0.691	0.849	0.518	0.690	0.843	0.516	0.691	0.854	ERR	ERR	ERR	0.409	0.502	0.777
nussinov	38.65	56.11	1.644	41.64	0.696	1.016	0.668	0.658	1.970	0.749	0.693	1.017	0.666	ERR	ERR	ERR	746.4	375.2	ERR
fwarshall	115.8	137.9	148.7	25.95	25.95	18.31	16.13	742.4	207.8	222.9	25.31	18.38	16.23	25.84	18.31	16.32	74.57	95.3	277.2
stencils																			
adi	117.8	71.00	11.62	99.63	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	6.141	5.034	6.163
jacobi-1d	0.002	0.005	0.018	0.003	0.004	0.012	0.012	0.085	0.034	0.036	0.003	0.012	0.011	0.004	0.012	0.012	0.011	0.011	0.014
seidel-2d	113.1	89.64	7.615	ERR	1.805	7.45	11.80	1.160	1.715	5.860	1.802	7.46	11.80	1.824	7.44	11.81	64.02	151.4	242.5
fdtd-2d	11.09	12.89	1.591	0.780	1.677	1.753	1.378	0.694	0.696	0.549	1.281	1.475	4.354	0.800	1.419	1.272	23.68	24.00	77.07
jacobi-2d	11.39	0.742	0.713	1.401	1.188	1.106	1.685	0.617	0.509	0.723	1.104	0.934	2.709	1.123	1.213	1.717	23.83	18.94	29.85
heat-3d	12.25	11.67	3.741	20.41	14.77	14.65	28.19	1.542	0.957	2.134	TO	TO	TO	19.23	14.96	28.15	1735	503.9	ERR

Table A.5: Parallel execution time in seconds of all PolyBench/C benchmarks with respect to the polyhedral compilers on an AMD machine. GRA - GRAPHITE; P'gst - Polygeist; P'opt - PolyOpt; WR - wrong output; ERR - error; TO - timed-out.

Benchmark	Polyhedral compiler(s)/Std. compiler(s) with best execution time	
	sequential	parallel
	linear-algebra/kernels	
atax*	Std. compilers	Std. compilers
2mm	`icc`	Pluto(-l2)+`icc`, PoCC+`gcc`, PoCC+`icc`
3mm	`icc`, Pluto(-l2)+`icc`	PoCC+`gcc`, PoCC+`clang`, PoCC+`icc`
bicg*	Std. compilers	Std. compilers
doitgen*	Pluto(-l2)+gcc	Std. compilers
mvt*	Std. compilers	Std. compilers
	linear-algebra/blas	
gemm	Pluto(-l2)+`icc`	Polly, Pluto(-tile)+`gcc`, Pluto(-dia)+`gcc`, PolyOpt, PoCC+`gcc`
gemver*	Std. compilers	Std. compilers
gesummv*	Std. compilers	Std. compilers
symm	PPCG+`gcc`, PPCG+`icc`	Polygeist
syr2k	PoCC+`clang`, Pluto(-dia)+`clang`, PPCG+`gcc`	PPCG+`icc`, Pluto(-tile)+`icc`, Pluto(-dia)+`icc`, PoCC+`icc`
syrk	PPCG+`gcc`, Pluto(-tile)+`clang`, PPCG+`clang`	PPCG+`icc`, Pluto(-tile)+`icc`, Pluto(-dia)+`icc`
trmm	Pluto(-dia)+`clang`, Pluto(-tile)+`clang`, PolyOpt	PolyOpt, Pluto(-tile)+`gcc`, Pluto(-dia)+`gcc`
	linear-algebra/solvers	
cholesky	PPCG+`gcc`	Pluto(-l2)+`icc`
durbin*	Std. compilers	Std. compilers
gramschmidt	PoCC+`gcc`, PoCC+`clang`, PoCC+`icc`	Pluto(-tile)+`gcc`, Pluto(-dia)+`gcc`
lu	Pluto(-dia)+`gcc`, Pluto(-tile)+`gcc`, PoCC+`gcc`, PoCC+`icc`	Pluto(-tile)+`gcc`, Pluto(-dia)+`gcc`
ludcmp	PPCG+`gcc`, PPCG+`clang`, PPCG+`icc`	PPCG+`gcc`, PPCG+`clang`
trisolv*	Std. compilers	Std. compilers
	datamining	
correlation	PoCC+`gcc`, Pluto(-dia)+`gcc`, Pluto(-tile)+`gcc`	PoCC+`gcc`, Pluto(-dia)+`gcc`, Pluto(-tile)+`gcc`
covariance	PoCC+`gcc`, Pluto(-dia)+`gcc`, Pluto(-tile)+`gcc`	PoCC+`gcc`, Pluto(-dia)+`icc`, Pluto(-tile)+`icc`
	medley	
deriche	`icc`, Pluto+`icc`, PPCG+`icc`	Polygeist, PPCG+`icc`
nussinov	Polygeist	Pluto(-l2)+`icc`
floyd-wars	PPCG+`gcc`	Pluto(-dia)+`icc`
	stencils	
adi	`icc`	PPCG+`clang`, PPCG+`gcc`, Polygeist
jacobi-1d*	Std. compilers	Std. compilers
seidel-2d	PoCC+`gcc`, Pluto(-dia)+`gcc`, Pluto(-tile)+`gcc`	Pluto(-l2)+`gcc`
fdtd-2d	PoCC+`gcc`	PolyOpt, PoCC+`gcc`
jacobi-2d	PoCC+`icc`	PoCC+`icc`
heat-3d	PPCG+`icc`	Pluto(-l2)+`gcc`, Pluto(-l2)+`clang`, Pluto(-l2)+`icc`

Table A.6: Best execution time (both sequential and parallel) for all benchmarks in PolyBench/C on Intel machine across standard compilers and polyhedral compilers; (*) Tiny benchmarks with execution time shorter than 1.5s (`gcc`).

Appendix B

Résumé en Français

L'impact des ordinateurs sur les applications humaines est très vaste et cela révolutionne les activités quotidiennes de la vie. Voici quelques domaines importants dans lesquels les ordinateurs ont eu un énorme impact : Santé - à partir du dossier médical électronique, de l'image médicale, des outils de diagnostic à la télémédecine; E-commerce - livraison à domicile d'articles ménagers; Communication - à partir des e-mails, des visioconférences, des plateformes de médias sociaux à la technologie de la réalité virtuelle; Systèmes de transports - outils de navigation, systèmes de gestion du trafic et véhicules autonomes; Science et recherche – modélisation du climat, génomique, découverte de médicaments et simulations physiques. La technologie informatique elle-même a connu plusieurs évolutions au cours des cinquante dernières années pour devenir compatible avec différents domaines. En particulier, l'ère de l'informatique personnelle dans les années 1980 a vu la croissance rapide des ordinateurs dans les maisons et les bureaux, explosant ainsi les capacités informatiques. Des progrès dans les microprocesseurs et le stockage ajoutent encore à la puissance de calcul. L'ère du mobile et de la démocratisation d'internet au début des années 2000 a donné naissance aux smartphones pour chaque individu. Et tout le monde connaît à présent l'impact du cloud computing et de l'intelligence artificielle et ses avantages.

Dans cette thèse, nous discutons des avancées dans les systèmes informatiques ciblant des applications scientifiques à large échelle et comment le processus de génération de code ou d'optimisation du code doit s'adapter à ces avancées pour maximiser l'utilisation des supercalculateurs.

B.1 Contributions Matérielles au Calcul

Le matériel et les logiciels progressent ensemble pour améliorer la puissance de calcul des ordinateurs. Nous abordons maintenant les quelques avancées réalisées en matière de matériel informatique :

Processeurs multicœurs. Intégration de plusieurs cœurs de traitement sur une seule

puce permettant ainsi l'exécution parallèle de threads/tâches par un CPU.

Unités de traitement graphique. Circuits électroniques initialement destinés au rendu graphique dans les jeux vidéo, mais qui ont beaucoup évolué et sont maintenant utilisés dans de nombreux autres domaines d'application, notamment pour l'apprentissage automatique, les simulations scientifiques et l'imagerie 3D.

Field-Programmable Gate Arrays. Un FPGA est un circuit intégré qui fournit la capacité de reconfiguration dynamique pour effectuer une grande variété de tâches, en particulier en traitement du signal numérique,

Jeux d'instructions vectorielles. Une architecture de jeu d'instructions spécialement conçue pour améliorer les performances d'opérations similaires sur plusieurs données, communément appelées opérations vectorielles ou SIMD (Single Instruction, Multiple Data). SSE/AVX/AVX-512 d'Intel, Neon d'ARM et CUDA de Nvidia en sont quelques exemples.

Systèmes distribués. Interconnexion de plusieurs ordinateurs entre eux qui communiquent et se coordonnent pour atteindre un objectif commun. Ces systèmes partagent tâches de calcul, nœuds de stockage et de traitement pour fournir une puissance de calcul élevée et évolutive.

Informatique quantique. Même si nous en sommes aux premiers stades de son développement, l'informatique quantique apparaît comme un champ de recherche prometteur, qui pourrait révolutionner les notions de puissance de calcul que nous connaissons actuellement.

B.2 Super calculateurs

À l'heure actuelle, on appelle supercalculateurs les systèmes informatiques regoupant un très grand nombre d'unités de calcul reliées par un réseau de communication très performant. Ils se composent de milliers, voire de millions de cœurs fonctionnant en parallèle pour atteindre un nombre élevé d'opérations en virgule flottante par seconde (Flop/s). Ils disposent d'architectures parallèles ou distribuées haut de gamme avec plusieurs unités de traitement telles que des processeurs, des GPUs et potentiellement des accélérateurs spécialisés (par exemple les FPGAs ou les TPU). Ces systèmes sont spécialement conçus pour traiter des calculs très complexes qui nécessitent un parallélisme massif et pour gérer des données massives. Leurs applications cibles sont les applications scientifiques et techniques résolvant des problèmes nécessitant des calculs intensifs qui dépassent de loin la puissance de calcul des ordinateurs classiques, comme par exemple des simulations de phénomènes physiques ou aujourd'hui l'apprentissage automatique en intelligence artificielle.

Le site Top500[1] classe les supercalculeurs existants dans le monde en termes de puissance de calcul. Depuis juin 2022, Frontier, hébergé au laboratoire national d'Oak Ridge, est la machine la plus puissante au monde grâce à ses 8,7 millions de cœurs et une

[1] https://www.top500.org/

puissance de calcul de 1206 PFlop/s, soit plus d'un exaFlop/s. Désormais, il incombe au logiciel d'utiliser efficacement ces ressources matérielles pour tirer pleinement parti de ces performances.

B.3 Optimisations de Code par le Compilateur

Dans cette section, nous discutons des techniques d'optimisation que le logiciel doit intégrer pour utiliser les avancées du matériel.

Parallélisme. La parallélisation au niveau des threads et des tâches distribuées implique de diviser le calcul en parties plus petites et exécuter ces parties simultanément en utilisant plusieurs cœurs. Par exemple, OpenMP (Open Multi-Processing) est une API de programmation parallèle largement utilisée en C, C++ ou Fortran qui prend en charge des tâches et des threads dans les systèmes multi-cœurs à mémoire partagée. MPI (Message Passing Interface) est une autre norme largement utilisée pour la programmation parallèle des systèmes distribués.

Ressources partagées. Si plusieurs threads ou tâches s'exécutent simultanément, il est essentiel de gérer correctement les ressources partagées pour éviter les accès concurrents aux données non synchronisés (provoquant de potentiels *datarace*) et les interblocages. L'aspect critique n'est pas seulement l'amélioration du calcul, mais aussi la garantie de l'exactitude du programme. Différents mécanismes de synchronisation (comme les mutex et les sémaphores), sections critiques, et les opérations atomiques nous aident à gérer efficacement ces ressources partagées.

Vectorisation. C'est une technique d'optimisation exploitant les instructions SIMD permettant ainsi un calcul plus rapide et plus efficace à l'aide de jeux d'instructions vectorielles. De nombreux compilateurs standards (comme `gcc`, `clang` et `icc`) effectuent de l'auto-vectorisation avec les techniques standard d'analyse du compilateur pour générer des instructions SIMD équivalentes aux instructions arithmétiques classiques. Dans certains cas, les compilateurs ont du mal à vectoriser automatiquement le code là où une vectorisation manuelle pourrait être appliquée. Cela implique de faire réaliser des analyses par le compilateur (comme l'analyse des dépendances), de réécrire le code pour inclure des directives vectorielles ou d'écrire des instructions vectorielles directement dans le code source.

Cohérence des Caches. Un système avec plusieurs unités de traitement (CPU ou cœurs) peut partager partiellement une hiérarchie mémoire. Il faut s'assurer que la mémoire est cohérente à travers l'ensemble de ces unités en maintenant la cohérence entre les données stockées dans leurs caches respectifs. De nombreux protocoles de cohérence de cache (comme MESI) sont utilisés pour maintenir la cohérence du cache.

Software Pipelining. C'est une technique pour transformer des itérations de boucles séquentielles en un pipeline d'instructions concurrentes, maximisant ainsi le parallélisme au niveau des instructions processeur et réduisant le coût de leur exécution.

Localité des Données. La gestion des données joue un rôle essentiel dans les systèmes

distribués. Cependant, atteindre une localité optimale des données dans un système distribué complexe est difficile en raison de nombreux paramètres, comme la latence du réseau, de la partition des données et du passage à l'échelle. Il existe des situations où les gains de performances de calcul sont anéantis par des placement de données inadéquats, nécessitant des copies excessives de données. Voici quelques considérations clés pour la localité des données dans les systèmes distribués : réplication de données, partitionnement de données, mouvement de données et topologie du réseau.

La prise en charge des systèmes multi-cœurs et distribués a abouti à l'évolution de nombreux langages parallèles et distribués (comme X10, Chapel, Cilk, Erlang, Go), qui ont réduit l'effort de programmation nécessaire à l'utilisation de ces supercalculateurs. Dans ces langages, le compilateur est capable de générer un code assembleur ciblant des architectures hétérogènes multicœurs.

Un compilateur est un programme informatique qui traduit un code source lisible par l'homme en un code assembleur lisible par machine qui est ensuite converti en instructions machine et exécuté sur le matériel informatique. Le compilateur joue donc un rôle crucial dans développement de logiciels car il transforme non seulement un code d'entrée en exécutable, mais applique également beaucoup d'optimisations pour générer un code efficace. Cependant, dans de nombreux cas, le compilateur se limite à appliquer certaines techniques d'optimisation sûres sur le code qu'il parvient à analyser, et en raison de la nature complexe du programme nécessite l'intervention du développeur pour l'optimiser davantage.

B.4 Contexte et Contributions

Dans cette thèse, nous introduisons une nouvelle technique de compilation et de génération de code pour architectures hétérogènes, ciblant les supercalculateurs exascale, afin d'optimiser des structures de boucles dans des applications spécifiques de simulation cardiaque. Nous fournissons au compilateur la possibilité de générer du code vectoriel pour instructions de boucles complexes (qui ne sont pas automatiquement vectorisées par le compilateur) et les exécuter sur un CPU en utilisant différents jeux d'instructions vectoriels. Nous étendons encore le processus de génération de code pour produire du code GPU pour ces boucles et les exécuter sur des GPU Nvidia et AMD.

Code de calcul intensif

Une application scientifique passe l'essentiel de son temps d'exécution en phase de calcul car il contient les instructions les plus nombreuses et complexes à être exécutées. Ces instructions complexes sont souvent des noyaux de calcul qui nécessitent des ressources importantes et reposent fortement sur la puissance de traitement du CPU ou du GPU. Ces noyaux gourmands en calcul impliquent principalement:

- des simulations telles que la dynamique moléculaire/fluidique, les modèles météorologiques et les dérivations,

- l'analyse des données : le traitement de très grands ensembles de données, l'analyse statistique ou les modèles d'apprentissages (ML),
- les calculs mathématiques qui nécessitent la résolution différentielle ou intégrale d'équations,
- le traitement d'image ou du signal.

La plupart du temps, ces noyaux sont enfermés dans des boucles avec un grand nombre d'itérations. L'identification et l'optimisation de ces noyaux sont essentielles pour améliorer les performances et l'efficacité de ces grandes applications. Les développeurs profilent souvent leurs applications pour identifier ces noyaux gourmands en calcul et les optimiser, à l'aide de techniques d'optimisation standard, ou en proposant de nouvelles optimisations spécifiques au matériel ou au logiciel. Il existe des cas où le compilateur lui-même suffit à les identifier et les appliquer (par exemple avec l'auto-vectorisation).

Ces optimisations du noyau peuvent-elles être encore améliorées ?

Plusieurs facteurs d'optimisation doivent être pris en compte lors de la génération d'un code optimisé ciblant les supercalculateurs. L'objectif principal est d'invoquer un parallélisme massif résultant en une utilisation efficace de tous les cœurs. Parfois, la nature complexe du logiciel rend l'application de ces optimisations difficile, par exemple lorsque les dépendances entre les instructions empêche certaines transformations de boucles. De plus, la présence d'instructions complexes comme un appel de fonction/bibliothèque ou un flux de contrôle irrégulier dans une boucle peut empêcher la vectorisation automatique du compilateur, même si la boucle ne porte aucune dépendance et est annotée avec des directives vectorielles.

Un autre aspect est la génération de code : de nombreuses applications s'appuient sur un langage dédié (*Domain Specific Language*, ou *DSL*) car il est plus commode pour les experts de ces domaines d'écrire du code avec moins de connaissances en programmation. Ensuite, c'est la responsabilité du compilateur pour ces DSL de générer un code efficace. La plupart de ces générateurs de code ciblent une architecture CPU ou GPU, mais rarement les deux.

La technologie des compilateurs a beaucoup évolué et il existe de nombreux nouveaux environnements de compilation pour aider les développeurs à générer du code optimisé pour leurs applications, par exemple le compilateur MLIR [7]. MLIR (*Multi-Level Intermediate Representation*) de LLVM [8] est une technologie de compilation innovante qui vise à représenter différents niveaux d'abstraction et d'optimisation sous une forme unifiée, à l'aide de plusieurs représentations intermédiaires coexistantes. Il s'agit d'une infrastructure très récente conçue pour relever les défis des compilateurs modernes, capable de cibler les systèmes hétérogènes, où plusieurs langages de programmation, cibles matérielles et passes d'optimisation doivent être coordonnés efficacement. Dans le chapitre 3 de ce manuscrit, nous fournissons une vue d'ensemble des *dialectes* et des passes d'optimisation de MLIR.

Génération de code hétérogène pour une application de simulation cardiaque

La simulation cardiaque fait référence à la modélisation informatique de la structure du cœur humain et de son fonctionnement. Il s'agit de créer des équations différentielles ordinaires (ODE) représentant l'anatomie cardiaque, dont l'activité électrique détermine le battement du cœur (électrophysiologie cardiaque) et le comportement physique des tissus cardiaques (mécanique). Ces simulations sont principalement utilisées comme aide médicale au diagnostic des maladies cardiaques, et pour permettre de mieux comprendre les rythmes cardiaques anormaux (arythmies), l'ischémie (réduction du flux sanguin), et d'autres troubles cardiaques. Les simulateurs cardiaques actuels ne peuvent simuler qu'une partie du cœur humain, ou le faire de manière assez imprécise. En raison des progrès de la puissance de calcul des ordinateurs et de l'évolution des supercalculateurs exascale, la recherche en simulation cardiaque permettra bientôt la simulation précise d'un cœur humain entier. Le cœur est composé d'environ 2 milliards de cellules, et simuler le cœur entier implique de traiter ces 2 milliards de cellules. Même avec les plus gros supercalculateurs, ces simulations sont difficiles compte tenu des équations mathématiques complexes et de l'efficacité avec laquelle le code de simulation est optimisé sur une architecture hétérogène. Dans le chapitre 2, nous abordons les limitations dans le processus de génération et d'optimisation de code des simulateurs cardiaques de pointe.

Dans le chapitre 4, nous introduisons une technique de compilation et de génération de code optimisée avec l'aide de MLIR (chapitre 3), pour émettre du code vectoriel CPU et GPU pour les applications de simulation cardiaque ciblant différentes architectures de supercalculateurs. Nous avons implémenté nos techniques sur le simulateur cardiaque open source openCARP [9] et utilisé les 48 modèles informatiques (modèles ioniques) disponibles dans ce simulateur pour tester notre implémentation. Ces 48 modèles ioniques sont différentes combinaisons d'équations (ODE) et sont largement utilisés dans la recherche cardiaque. Nous avons réalisé des évaluations de performances sur deux plate-formes d'expérimentation à grande échelle : (i) Grid'5000 (`https://www.grid5000.fr`) et (ii) PlaFRIM (`https://plafrim.fr`). Ces deux plate-formes contiennent les diverses architectures nécessaires pour exécuter le code hétérogène généré. Dans le chapitre 4, section 4.2 les résultats de performances des 48 modèles sont rapportés pour le code vectoriel CPU (SSE, AVX, AVX-512) et le code GPU (Nvidia et AMD). Nos résultats expérimentaux montrent qu'en aidant le compilateur à générer un code optimisé on obtient une utilisation très efficace de l'architecture multicœur et hétérogène disponible dans les supercalculateurs. Nous avons également mesuré la consommation d'énergie et montré l'efficacité énergétique de nos techniques. Nous avons publié ces résultats dans deux conférences internationales [10, 11].

Ce travail présenté dans le chapitre 4 est un travail collaboratif avec un ingénieur de recherche (initialement commencé par Tiago Trevisan Jost, puis poursuivi par Raphaël Colin). Il a conduit à une collaboration avec l'équipe de recherche STORM de l'Inria Bordeaux, où notre flux de compilation est intégré à StarPU [12], un système d'exécution dynamique basé sur des tâches, pour distribuer dynamiquement l'exécution sur l'architecture cible [13].

Améliorations des techniques d'optimisation des boucles polyédriques

Après les améliorations prometteuses des performances des applications de simulation cardiaque, nous avons généralisé nos techniques dans les optimisations de boucles polyédriques. Les techniques d'optimisation polyédriques sont utilisées pour transformer les programmes avec des nids de boucles régulières comme les opérations matricielles, le calcul de stencils et les applications de traitement d'images. Elles commencent par représenter le nid de boucles sous forme de polyèdres dans un espace géométrique, puis effectuent l'analyse de dépendance entre les itérations de boucles. Elles aident surtout à réaliser trois optimisations : pavage, parallélisme de boucles et vectorisation. Le pavage se fait en restructurant les boucles dans le but d'améliorer la localité des données. Les boucles sont annotées avec des directives parallèles et vectorielles pour le parallélisme de boucle et la vectorisation, respectivement.

Il existe de nombreux types différents de compilateurs polyédriques disponibles comme des compilateurs source-à-source généraux, ceux intégrés à des compilateurs généraux, ou encore spécifiques à une application ou spécifiques à une cible. Dans le chapitre 5, nous avons d'abord effectué une étude détaillée des compilateurs polyédriques à usage général disponibles en utilisant nos plate-formes d'expérimentation à grande échelle, pour connaître leurs avantages et leurs limites. Nous avons utilisé PolyBench/C [14], un ensemble de trente benchmarks numériques largement reconnus, ciblant divers domaines d'application pour mener une étude détaillée sur les compilateurs polyédriques. L'analyse à l'aide de compteurs de performances matérielles a fourni des informations et des directions d'amélioration supplémentaire des compilateurs polyédriques, en particulier dans l'amélioration de la vectorisation et la génération de code GPU.

Dans le chapitre 6, nous abordons quelques limitations des compilateurs polyédriques de pointe : (i) les directives de compilation vectorielles ne sont qu'une recommandation aux compilateurs et sont ignorées dans certains cas, et (ii) il n'y a pas d'approche unifiée pour générer des codes hétérogènes. Nous nous sommes appuyés sur Polygeist [15] pour généraliser notre technique précédente comme solution aux deux problèmes mentionnés ci-dessus. Nous avons choisi Polygeist [15], étant un environnement basé sur MLIR/LLVM pour transformer un code `C/C++` en code MLIR polyédrique, et généraliser nos techniques à l'aide de Polygeist nécessitera un effort réduit car l'environnement requis est déjà établi. Nous avons modifié le flux de génération de code de Polygeist pour émettre une boucle OpenMP SIMD en MLIR pour les boucles annotées vectorielles. Nos résultats expérimentaux ne montrent aucune amélioration des performances sur les programmes PolyBench/C, à cause de la prise en charge limitée des directives OpenMP SIMD de MLIR. L'environnement MLIR est en phase de développement très active pour prendre en charge diverses optimisations du compilateur. Donc, dans un futur très proche MLIR pourrait fournir un support d'optimisation complet pour les boucles OpenMP SIMD et nous espérons que notre technique proposée améliore la vectorisation. Nous avons laissé l'implémentation de la génération de code GPU dans le cadre de travaux futurs. L'étude sur les compilateurs polyédriques est publiée dans ACM TACO [16], et la technique de génération de boucles Polygeist SIMD est acceptée à un symposium [17].

Cette thèse présente les progrès de la puissance de calcul des supercalculateurs, les limites des techniques d'optimisation de code ciblant leurs architectures hétérogènes et une solution pour surmonter ces limitations, ciblant en particulier la vectorisation de code CPU et la génération de code GPU. Notre objectif est de montrer comment un code optimisé dans des applications complexes peut utiliser efficacement les ressources massives des supercalculateurs pour améliorer les performances d'applications ayant un impact sur la vie réelle humaine.

Cette thèse a été financée par le *European High-Performance Computing Joint Undertaking EuroHPC*, sous la convention de subvention numéro 955495 (**MICROCARD**), co-financé par le programme Horizon 2020 de l'Union Européenne, ainsi que par la France, l'Allemagne, l'Italie, la Suisse, l'Autriche, et la Norvège (https://microcard.eu).

Arun THANGAMANI

Génération de code optimisée pour des nids de boucles parallèles et polyédriques à l'aide de MLIR

Abstract

In this thesis we show the benefits of the novel MLIR compiler technology to the generation of code from a DSL, namely EasyML used in openCARP, a widely used simulator in the cardiac electrophysiology community. Building on an existing work we deeply modified openCARP's native code generator to enable efficient vectorized CPU and GPU code generation (Nvidia CUDA and AMD ROCm). Generating optimized code for different accelerators requires specific optimizations and we review how MLIR has been used to enable multi-target code generation from an integrated generator. To our knowledge, this is the first work that deeply connects an optimizing compiler infrastructure to electrophysiology models of the human body, showing the potential benefits of using compiler technology in the simulation of human cell interactions. Additionally, we did a study on the polyhedral compilers and generalized our techniques using Polygeist to improve the vectorization and heterogeneous code generation of polyhedral compilers.

Résumé

Dans cette thèse, nous montrons les avantages de la nouvelle technologie de compilateur MLIR pour la génération de code à partir d'un DSL, à savoir EasyML utilisé dans openCARP, un simulateur largement utilisé dans la recherche en électrophysiologie cardiaque. S'appuyant sur un travail existant nous avons profondément modifié le générateur de code natif d'openCARP pour permettre une génération efficace de code CPU vectoriel et GPU (Nvidia CUDA et AMD ROCm). La génération de code optimisé pour différents accélérateurs nécessite des optimisations spécifiques et nous examinons comment MLIR a été utilisé pour permettre la génération de code multi-cible à partir d'un générateur intégré. À notre connaissance, il s'agit du premier travail qui relie profondément une infrastructure de compilateur d'optimisation aux modèles électrophysiologiques du corps humain, montrant les avantages potentiels de l'utilisation de technologies de compilation dans la simulation des interactions entre cellules humaines. De plus, nous avons réalisé une étude sur les compilateurs polyédriques et généralisé nos techniques en utilisant Polygeist pour améliorer la vectorisation et la génération de codes hétérogènes des compilateurs polyédriques.