Automatic Code Rewriting for Performance Portability

by

Alister Johnson

A dissertation accepted and approved in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

in Computer Science

Dissertation Committee:

Allen Malony, Chair

Boyana Norris, Core Member

Zena Ariola, Core Member

Jon Brundan, Institutional Representative

Michael Wolfe, External Member

Camille Coti, External Member

University of Oregon

Spring 2024

DISSERTATION ABSTRACT

Alister Johnson

Doctor of Philosophy in Computer Science

Title: Automatic Code Rewriting For Performance Portability

Rewriting code for cleanliness, API changes, and new programming models is a common, yet time-consuming task. This is important for HPC applications that desire performance portability in particular, since these applications are usually very long lived and wish to run on many architectures, so they need to be written such that they can make good use of all the available hardware with minimal code changes. Furthermore, it is unknown what future supercomputer hardware and programming models will be, so they need to be written in such a way that they are "future proof" and will only need minimal rewrites in the future.

Localized or syntax-based changes are often mechanical and can be automated with text-based rewriting tools, like `sed`. However, non-localized or semantic-based changes require specialized tools that usually come with complex, hard-coded rules that require expertise in compilers. This means techniques for source rewriting are either too simple for complex tasks or too complex to be customized by non-expert users; in either case, developers are often forced to manually update their code instead.

This work describes a new approach to code rewriting which exposes complex and semantic-driven rewrite capabilities to users in a simple and natural way. Rewrite rules are expressed as a pair of parameterized "before-and-after" source code snippets, one to describe what to match and one to describe what the replacement looks like. Through this novel and user-friendly interface,

programmers can automate and customize complex code changes which require a deep understanding of the language without any knowledge of compiler internals.

This dissertation includes previously published and unpublished co-authored material.

CURRICULUM VITAE

NAME OF AUTHOR: Alister Johnson

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon, Eugene, OR, USA
Harvey Mudd College, Claremont, CA, USA

DEGREES AWARDED:

Doctor of Philosophy, Computer Science, 2024, University of Oregon
Master of Science, Computer and Information Science, 2020, University of Oregon
Bachelor of Science, Joint Mathematics and Computer Science, 2017, Harvey Mudd College

AREAS OF SPECIAL INTEREST:

Compilers
Performance Portability
High Performance Computing

PROFESSIONAL EXPERIENCE:

Graduate Student Research Aide, Argonne National Laboratory, 2021-2022
WJ Cody Research Assistant, Argonne National Laboratory, 2021
TAU Research Group, 2018-2021
GPU Compilers Intern, NVIDIA HPC SDK Team, 2018, 2019, 2020
Teaching Assistant – Computer Science III, University of Oregon, 2017
Harvey Mudd Clinic Program, Environmental Data Resources (EDR), 2016-2017

GRANTS, AWARDS AND HONORS:

Moursund Fellowship, Winter Term 2024

PUBLICATIONS:

Alister Johnson, Camille Coti, Allen D Malony, and Johannes Doerfert. MARTINI: The little match and replace tool for automatic code rewriting. Journal of Open Source Software, 7(76), 2022.

Alister Johnson, Camille Coti, Allen D. Malony, and Johannes Doerfert. MARTINI: The little match and replace tool for automatic application rewriting with code examples. In Jose Cano and Phil Trinder, editors, Euro-Par 2022: Parallel Processing, pages 19–34, Cham, 2022. Springer International Publishing.

Alister Johnson. Area exam: General-purpose performance portable programming models for productive exascale computing. Technical report, University of Oregon, Department of Computer and Information Sciences, June 2020.

Alister Johnson. Scaling collaborative filtering with PETSc. In 2018 IEEE International Conference on Big Data (Big Data), pages 4237–4244, December 2018.

ACKNOWLEDGEMENTS

Many thanks to my advisors and coauthors, Professors Allen Malony and Camille Coti, for their help and adivce, and for keeping me on the right track. Thanks also to Jan Hückelheim and Johannes Doerfert at Argonne National Lab for funding my work.

I would not have been able to complete this work without the unwavering support of my friends and family – in particular, my parents, and my dear friends Sam, Maddy, and Sun-Ae. Thank you all for never doubting me, and for making sure I know I'm the best to ever do it.

DEDICATION

To everyone who walked so I could run.

TABLE OF CONTENTS

Chapter Page

LIST OF FIGURES

Figure Page

Figure Page

LIST OF TABLES

CHAPTER I

INTRODUCTION

Rewriting and refactoring, for example to optimize performance, port to a new programming model, update an API, or add error handling, are common tasks that can take a great deal of time if performed by hand on a large code base, which is what the vast majority of high-performance computing (HPC) applications are. Some of these tasks are easily automated with existing text-based search-and-replace tools, like the `sed` stream editor or C preprocessor macros. This is especially true if the rewrite is localized and does not require any semantic information that is not also present through syntax. However, once rewrites span code ranges or require semantic reasoning, text-based tooling is inadequate or requires complex implementations (for example, tracking balanced parentheses with extended regular expressions).

Traditionally, this is where compiler-based tooling comes in (Quinlan and Liao (2011)). The compiler's frontend has parsing and semantic analysis capabilities that allow more complete understanding of the source code and, consequently, semantic-based rewriting over most arbitrary code ranges. However, developing and customizing such tooling has a high barrier to entry, requiring a deep understanding of the compiler and its rewriting infrastructure (if it even has one), which restricts the developer pool drastically (Murai, Sato, Nakao, and Lee (2018); Takizawa, Hirasawa, Hayashi, Egawa, and Kobayashi (2014)). Alternatively, developers can choose to learn the user interface of one of several existing rewriting tools, which are of a similar complexity and require similar knowledge to building a tool in a compiler (see Sec. 2.8.6), again making it difficult for the average

developer to get started. In the past, as long as the number of desired rewrites was small and customization was not required, this was sufficient.

Today, however, language standards are changing more rapidly and new parallel programming models are constantly being developed, meaning developers need to expend more effort to keep their applications up to date and use a streamlined refactoring process (Wright, Jasper, Klimek, Carruth, and Wan (2013)). Furthermore, recent HPC machines come from a variety of vendors and don't always support the same programming models, so developers have to choose between performance and portability for their application. Many of the new parallel programming models have begun adopting a philosophy of *performance portability*, which attempts to minimize that trade-off (Daniel and Panetta (2019); Deakin et al. (2019); Dreuning, Heirman, and Varbanescu (2018); Harrell et al. (2018); Pennycook, Sewall, and Lee (2016, 2019); Wolfe (2016b)). But developers must still port their applications to these new models, and we return to the problem of rewriting and refactoring.

Some rewriting tasks might be a simple matter of replacing one API call with another, but most often complex changes have to be made as well, especially if the application has any kind of parallelism. We will show that, in many cases, these changes often follow patterns, and if programmers are able to capture those patterns in some way, these tasks seem like they *should* be able to be automated.

A tool to automate code rewriting must have the ability to access and understand semantic context, allow the user to easily contribute semantic knowledge, and utilize both in code replacements. For sophisticated code transformations, this means using a compiler frontend is often the only solution. However, we believe that this does not preclude a user-friendly approach, since

developers can often write *what* they want to happen, though maybe not *how* it should happen.

The problem is essentially one of finding an *acceptable balance* between 1) minimizing the complexity of expressing the users' rewriting intentions, 2) maximizing the variety and customizability of rewrites available, and 3) maximizing the users' ability to realize their intentions via automated tooling (e.g., automate bulk edits on a large codebase). The central research issue is discovering what is possible with respect to this balance. Previous work has generally emphasized two of these three goals at the cost of the third (usually at the cost of (1)). We wish to discover if it is possible to achieve all three goals. To this end, we constrain our solution methods to those based solely on a high-level programming language (to meet goal (1)), those that do not hinder customization or variety of transformations (to meet goal (2)), and those that lend themselves to high levels of automation (to meet goal (3)), and investigate what can and cannot be done.

We are developing a system based on *semantic matching* and user-provided *code replacements* that are accessible to the average programmer. Similar to regular expressions, users can describe and customize code transformations naturally as "before-and-after" snippets of C++ code, which correspond to the two expressions used in search-and-replace schemes. The available context for searching and replacing is not restricted to syntax, though; it also contains semantic information extracted by the compiler. Our interface is designed to be intuitive for C++ developers by restricting its syntax to modern C++ and requiring no knowledge of compiler internals, unlike previous rewriting tools. It is also designed to give users a great deal of control over which changes are applied and where. The main contributions of this work will be:

– A C++ user interface, with syntax similar to an embedded DSL, that is both user-friendly *and* customizable, unlike many previous similar tools.

– MARTINI, the Little Match and Replace Tool, an open-source[1], extensible code rewriting tool built on top of Clang's tooling infrastructure and ASTs.

– Case studies that demonstrate the versatility of MARTINI and its applications to performance and performance portability.

The rest of this dissertation is organized as follows. Chapter II gives relevant background information on performance portability and a selection of performance portable programming models. Chapter III describes the methodology used in this work. Chapter IV gives a basic example of our automated rewriting framework, while Chapters V, VI, and VII describe several use cases and studies related to performance portability. Chapter VIII summarizes and concludes, while Chapter IX describes future work.

Chapter II contains material that was originally published solely by Alister Johnson (Johnson (2020)). Chapters III, IV, VI, and VII contain material that was originally published by Johnson, Coti, Malony, and Doerfert (2022). Chapters V and VI contain material that will be published by Johnson, Coti, Malony, and Hueckelheim (n.d.). Chapter VII contains material that will be published by Huck, Coti, Johnson, and Malony (n.d.).

[1] `https://github.com/ajohnson-uoregon/llvm-project/tree/feature-ajohnson/clang-tools-extra/clang-rewrite`.

CHAPTER II

BACKGROUND

This chapter contains material that was originally published by Johnson (2020).

This chapter will provide context and background information for the rest of this dissertation, including current trends in high performance computing (HPC), performance portability, productivity, and existing performance portable models and rewriting tools referenced elsewhere in this dissertation.

2.1 Current Goals in High Performance Computing

Before discussing performance portability, we must first discuss current trends in HPC, so we can understand why performance portability is important.

The current goal of HPC is to build machines capable of performing 10^{18} floating point operations per second – 1 exaFLOP. *Exascale computing* is essential for doing new research in many domains. It will allow simulations to have higher resolution, scientific computations to get results more quickly, and machine learning applications to train on more data.

Current supercomputers can, on the whole, only do on the order of 100 petaFLOPs (1 exaFLOP = 1,000 petaFLOPs). Frontier (Oak Ridge National Lab) is the first, and so far only, machine that can perform an exaFLOP, clocking in at roughly 1.2-1.6 exaFLOPs. Fugaku (RIKEN, Japan) can do ∼450-540 petaFLOPs; LUMI (EuroHPC/CSC) can do ∼300-430 petaFLOPs; Leonardo (EuroHPC/CINECA) can do ∼240-300 petaFLOPs; and Summit (Oak Ridge National Lab) can do ∼150-200 petaFLOPs (*Top500 List* (2023)). Aurora, which arrived at Argonne National Lab in late 2023, will theoretically be a second

exascale machine, able to provide 2 exaFLOPs of computation (*Aurora Exascale Supercomputer* (2023)).

2.1.1 Goals for Exascale. The U.S. Department of Energy (DoE) set a goal to build an exascale machine that has a hardware cost of less than $200M and uses less than 20MW of power (Shalf, Dosanjh, and Morrison (2011)). Aurora will not meet the cost goal, and it's unlikely it will meet the power goal, but there is hope for future machines.

Other (implicit) goals for exascale computing include (1) making exascale machines "easy" to program, (2) verifying that these machines can do a "useful" exaFLOP, and (3) verifying they can perform sustained exaFLOPs. The first of these is also a goal of performance portability, and will be discussed further in Sec. 2.3 (on productivity).

As for goals (2) and (3), the origin of these questions goes back to how supercomputer performance is measured. The measurement method used by Top500 is the LINPACK Benchmark, a dense linear algebra solver (*The LINPACK Benchmark* (2023)). LINPACK has been criticized for being overly specific and thus not representative of real applications that will be run on these machines. For example, LINPACK does not account for data transfers, which is one of the bottlenecks on current machines. Very few applications can achieve even close to the peak performance of LINPACK because they cannot make use of all the floating point units on a chip and/or they have to wait on data movement (Kindratenko and Trancoso (2011)).

The HPC community wants an exascale machine that can perform a "useful" exaFLOP with a real application that has these kinds of problems. If the machine can only do exaFLOPs with highly tuned, compute-bound programs like

LINPACK, that isn't helpful for domain scientists, whose applications are much more varied. If the machine cannot perform sustained exaFLOPs, but only burst to an exaFLOP under some circumstances (e.g., the kind of dense math performed by LINPACK), that also isn't helpful. Building an exascale machine *that can meet goals (2) and (3)* will be a challenge beyond merely building an exascale machine.

2.1.2 Challenges of Exascale. A DoE report on exascale computing (Lucas et al. (2014)) identified the following as the top ten challenges to building an exascale supercomputer. Many other works have also identified a subset of these as major difficulties for exascale systems (Kogge and Shalf (2013); Mo (2018); Shalf et al. (2011)).

1. Energy efficiency — the goal is to use only 20 MW of power, but simply scaling up current technology would use far more than this.

2. Interconnect technology — we need communication to be fast and energy efficient, otherwise an exascale machine "would be more like the millions of individual computers in a data center, rather than a supercomputer" (Lucas et al. (2014)).

3. Memory technology — we need to minimize data movement in our programs, make movement energy efficient, and have affordable high-capacity and high-bandwidth memory.

4. Scalable system software — current system software was not designed to handle as many cores and nodes as exascale systems will have. Systems also need better power management and resilience to faults.

5. **Programming systems — we need better programming environments that allow developers to express parallelism, data locality, and resilience, if they so choose.**

6. Data management — our software needs to be able to handle the volume, velocity, and diversity of data that will be produced by applications.

7. Exascale algorithms — current algorithms weren't designed with billion-way[1] parallelism in mind, and we need to rework them or design completely new algorithms.

8. Algorithms for discovery, design, and decision — we need software to be able to reason about uncertainty and optimizations (e.g., error propagation in physics simulations or the optimal instruction set to use for a machine learning algorithm).

9. Resilience and correctness — exascale computers will have many more nodes than current petascale computers, and hardware faults will therefore be more frequent. We need both machines and applications to be able to recover from these faults and guarantee correctness.

10. **Scientific productivity — we want to increase productivity of domain scientists with new tools and environments that let them work on exascale machines easily.**

The two bold challenges, (5) and (10), are of particular interest to performance portability research. Performance portability is concerned

[1]To get 10^{18} FLOPs with cores running at 1 GHz (a reasonable approximation of core frequencies in current petascale supercomputers), we would need at least 1 billion cores.

with creating programming models that run equally well on multiple architectures/machines; the more difficult question is, how can we do so while allowing developers to express parallelism, data locality, and fault tolerance in ways that won't tie them to a particular machine or get them bogged down in details?

2.1.3 A Brief History of Supercomputing. Before discussing how exascale computing and performance portability impact and inform each other, we need a brief digression to the history of supercomputer architectures.

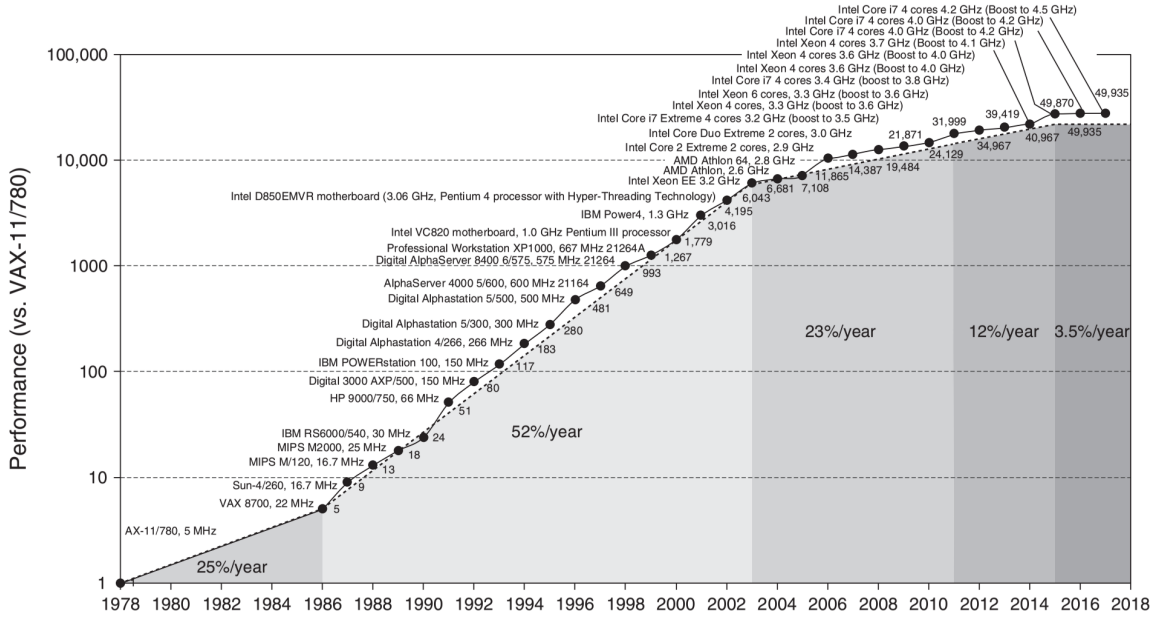


Figure 1. Forty years of processor performance improvements, as measured by SPECint. Image source: Hennessy and Patterson (2019).

In the early years, processor performance improvements were mainly due to technological advancements, and microprocessor performance doubled roughly every 3.5 years. In the mid-1980s, further advances in processor development led to greater increases in processor performance – performance began to double every 2 years (Moore's law). Dennard scaling, which relates the power density of transistors to transistor size, allowed chip manufacturers to drastically increase the number of

transistors per chip while increasing clock frequency, giving large performance gains with essentially the same base architecture. Figure 1 shows this steady growth beginning around 1986. The following decades of steady progress got developers used to performance improvements for "free" – if their application was too slow, they could just wait for the next generation of processor to come out, and (without modifying their code!) it would run faster.

However, in the early to mid-2000s, Dennard scaling began to break down. Chip designers began hitting physical limits, like the power wall: the power density of processors grew so high that scaling any further would make it physically impossible to dissipate the excess heat. Processor and compiler developers began seeing diminishing returns from instruction-level parallelism, and, as Figure 1 shows, progress began to slow. Processor performance was only doubling every 3.5 years again, and it was clear another solution would be needed, which began the transition to multicore chips. After this, performance growth came from increasing the number of cores per chip while clock rates stabilized (Kogge and Shalf (2013); Shalf et al. (2011)), and we continue to see decreases in performance gains. If the trend of the mid-2010s continues, processor performance will only double every 10-20 years.

Before the mid-2000s, supercomputers were made up of simple nodes (if they had nodes at all) and programmed with regular programming languages like C and Fortran or, as massively parallel and cluster architectures became more popular, with C and Fortran plus the MPI message passing library, for communication between nodes. Afterwards, as intra-node parallelism increased and MPI's scalability was called into question (Thakur et al. (2010)), MPI+X (where "X" is an intra-node parallel programming model) became the default.

2.1.4 Modern Architectures.

To cope with the end of Moore's law and Dennard scaling, three families of supercomputer architectures have emerged: heavyweight, based on pre-2004 models with a few powerful cores with high clock speeds; lightweight, with many less powerful cores and slower clock speeds (e.g., IBM's BlueGene architecture); and heterogeneous, a mix of heavy- and lightweight processors, like a CPU+GPU system. Performance projections from 2013 (Kogge and Shalf (2013)) implied that only heterogeneous systems had a hope of making the exascale compute goal within the power limit. Recent developments have borne this out – 8 out of the current top 10 systems have some kind of accelerator, and 7 of these use GPUs as their accelerator (*Top500 List* (2023)). All 10 of the top 10 most power efficient machines use GPU accelerators (*Green500 List* (2023)).

This proliferation of architectures is shown in Table 1, which lists the architectures of current and future supercomputers from around the world. The HPC community hasn't agreed yet on the best way to program these machines, but we can agree that we don't know what future exascale (and larger) machines will look like. Some of the US DoE machines most recently announced, Aurora (*Aurora Exascale Supercomputer* (2023)), Perlmutter (Harris (2021)), and Frontier (*Frontier supercomputer debuts as world's fastest, breaking exascale barrier* (2022)), have very different native programming models, and users will likely want to run their applications on all of them at some point.[2] Future machines may even have multiple types of accelerators in each node, each specialized for a different type of computation, or programmable coprocessors, like FPGAs (Yang and Fu (2018)), that users want to use at the same time. Therefore, the HPC community

[2]Aurora will have Intel CPUs and Intel GPUs, Perlmutter will have AMD CPUs and Nvidia GPUs, and Frontier will have AMD CPUs and AMD GPUs. Each type of GPU uses a different (incompatible) native programming model.

wants to "future-proof" its applications by developing new *performance portable* programming models.

	Lifespan	Architecture	CPU vendor	Accel. vendor
Aurora	2023–	Cray EX	Intel	Intel
Frontier	2022–	Cray EX	AMD	AMD
Fugaku	2021–	Fujitsu	Fujitsu	N/A
Perlmutter	2021–	Cray Shasta	AMD	Nvidia
Frontera	2019–	Dell C6420	Intel	N/A
Summit	2018–	IBM AC922	IBM	Nvidia
Sierra	2018–	IBM AC922	IBM	Nvidia
ABCI	2018–	Fujitsu CX2560 M4	Intel	Nvidia
Theta	2017–	Cray XC40	Intel	N/A
TaihuLight	2016–	Sunway MPP	Sunway	N/A
Cori	2016–	Cray XC40	Intel	N/A
Oakforest-PACS	2016–	Fujitsu CX1640 M1	Intel	N/A
Trinity	2015–	Cray XC40	Intel	Intel
Tianhe-2A	2013–	TH-IVB-FEP	Intel	NUDT
Titan	2012–2019	Cray XK7	AMD	Nvidia
Piz Daint	2012–	Cray XC50	Intel	Nvidia
Sequoia	2012–	BlueGene/Q	IBM	N/A
Mira	2012–	BlueGene/Q	IBM	N/A
K computer	2011–2019	Fujitsu	Fujitsu	N/A

Table 1. Recent supercomputers, from the 2015-2023 Top500 lists and various announcements.

2.1.5 Why Performance Portability Matters.

Currently, developers need to expend effort to port their applications to a new machine, and speedup is no longer guaranteed. Sometimes application development teams spend months porting and optimizing for a new architecture, only to have to repeat all that work again a year or two later when the next machine comes out. Many HPC applications have a lifespan measured in decades, while supercomputers usually last far less than that. Being forced to refactor for new machines every few years leaves these applications fragile and error-prone, since the time developers have to test, debug, and otherwise improve their code decreases significantly – the current

situation is actively harming work done by domain scientists (Majeed, Dastgeer, and Kessler (2013); Sedova, Eblen, Budiardja, Tharrington, and Smith (2018)).

The end goal of performance portability is to solve this problem and minimize the work users need to put into porting and optimizing their programs for future architectures. Instead of rewriting the same code again and again, developers will have time to improve the functionality of their application. To quote the OpenACC website: "more science, less programming" (OpenACC Group (n.d.)).

2.2 Performance Portability

Performance portability is not new, but increased interest in it is. Computer architectures have gone through (and are continuing to go through) so many shifts that programmers have had to port or rewrite their code multiple times, which isn't sustainable in the long run (see Sec. 2.1.5).

When developers port an application to a new architecture, they do not want to be locked into that architecture – they want to move between architectures with minimal porting effort. In addition, they want their application to perform well on new architectures with minimal optimization and performance tuning. Developers want their applications to be *performance portable*.

2.2.1 Defining Performance Portability.

Several different definitions of performance portability have been proposed in recent years. Some definitions, as listed by Pennycook et al. (2016):

1. An approach to application development, in which developers focus on providing portability between platforms without sacrificing performance (Pennycook et al. (2016)).

2. The ability of the same source code to run productively on a variety of different architectures (Larkin (2016)).

3. $P_n^{\mathscr{P}}(b \to t) = \frac{S_n^t}{S_n^b} \times 100\%$ for program $\mathscr{P}$, base system b, target system t, and speed-up on n nodes S_n (Zhu, Niu, and Gao (2007)).

4. The ability of an application to achieve a similar high fraction of peak performance across target devices (McIntosh-Smith, Boulton, Curran, and Price (2014)).

5. The ability of an application to obtain the same (or nearly the same) performance as a variant of the code that is written specifically for that device (Edwards, Trott, and Sunderland (2014)).

While each of these definitions has its strengths, each also has weaknesses. Definition (1) is intuitive and provides a good baseline for determining if an application can claim to be performance portable, but it is subjective and provides no way to measure how performance portable an application is. Definition (2) restricts the application code to a single version, which is desirable for many reasons, but suffers from the same problems as (1) (how should we define "productively?"). Definition (3) does provide a metric, but this metric is difficult to compare between applications, systems, and program inputs. Definition (4)'s metric is problematic because it is somewhat subjective, peak efficiency is difficult to measure (see Sec. 2.2.2 on architectural efficiency), and two architectures might be sufficiently different that achieving peak on one is simple, but on another is impossible (e.g., one machine has vector units that the application can't utilize). Definition (5) is similarly somewhat subjective, and might be difficult to measure if code version for a certain device doesn't exist.

An ideal definition of performance portability would be objective and provide an easy way to both measure *and* compare values for different applications.

The most commonly utilized performance portability definition is from Pennycook et al. (2016), because it comes with such a metric. However, there are still several criticisms of Pennycook et al.'s metric, which will be discussed in the next section.

2.2.2 Primary Metric. Pennycook et al.'s definition of performance portability is "a measurement of an application's performance efficiency for a given problem that can be executed correctly on all platforms in a given set" (Pennycook et al. (2016)). Pennycook et al. designed their definition to reflect both the performance and portability aspects. In addition, they specifically mention executing *correctly* on a *given problem* to ensure that applications ported incorrectly are not considered portable and to note that different inputs can yield different performance characteristics.

The corresponding metric is defined as the harmonic mean of the efficiency of the application on all supported platforms in a set (see Smith (1988) for the logic behind choosing the harmonic mean). When one or more platforms are not supported, the metric goes to zero:

$$\Phi(a, p, H) = \begin{cases} \dfrac{|H|}{\sum_{i \in H} \dfrac{1}{e_i(a, p)}}, & \text{if } i \in H \text{ is supported} \\ 0, & \text{otherwise} \end{cases}$$

where a is the application, p is the problem/input, H is the set of platforms, and $e_i(a, p)$ is the efficiency of application a solving problem p on platform $i \in H$. Higher is better — Φ will be high when an application ports well to all architectures in H, and low when it only ports well to a few (or none) of them.

The Φ metric can be used by comparing application performance to either the theoretical peak performance of the architecture — the *architectural efficiency* — or the best known performance of the application on any architecture

— the *application efficiency*. Pennycook et al. note that both these efficiencies are important since just looking at one can bias results. Only considering architectural efficiency can give artificially low values for Φ if an application physically cannot take advantage of architectural features (e.g., fused multiply-add instructions or vectorization), and only looking at application efficiency can give artificially high results when no truly efficient implementation exists.

The rest of this section will discuss some criticisms and improvements suggested since Pennycook et al. first published their metric.

2.2.2.1 Architectural Efficiency. Dreuning et al. (2018) note that, when using architectural efficiency, we must choose between comparing the application's achieved FLOPs *or* achieved memory bandwidth to the architecture's theoretical peaks. Choosing the wrong peak can lead to incorrectly high or low values of Φ depending on whether an application is compute or memory bound. Dreuning et al. suggest fixing this by using a performance portability model similar to the Roofline model (Williams, Waterman, and Patterson (2009)): compute bound applications should be compared to the platform's theoretical peak FLOPs, and memory bound applications to theoretical peak bandwidth. The operational intensity (operations per memory access) of the application compared to the hardware can be used to determine whether applications are compute or memory bound.

Yang et al. (2018) add that, when using the Roofline model, we need accurate theoretical ceilings. In particular, they note that vendors can be optimistic when reporting numbers, so real measurements should be used. In addition, application FLOPs should not be counted by hand, because compiler optimizations and special hardware units can make hand-counted FLOPs inaccurate. Yang et al.

demonstrate exactly how far off vendor estimates of their hardware's performance can be, as well as how different hand-counted FLOPs can be from measured FLOPs. Ofenbeck, Steinmann, Caparros, Spampinato, and Püschel (2014) similarly describe why using hardware counters is more accurate than (vendor) estimates, although still problematic for reasons that are outside the scope of this survey.

Yang et al. also note that we need to choose *relevant* ceilings when measuring performance portability. Using an unrealistic theoretical peak will give artificially bad results. For example, if the application is not using fused multiply-add (FMA) instructions, the ceiling measurement should also not use FMA. Knowing how close an application truly is to the platform's most relevant roofline will help developers decide how and where to optimize their code.

2.2.2.2 Application Efficiency.

Architectural efficiency can give a theoretical upper bound for application performance, but does not tell us whether an application is actually efficient. Application efficiency can give us a practical upper bound and does not require estimation, but can be prone to bias. Sometimes there is no good, efficient implementation, or the developers are unaware of an implementation that performs better than their chosen reference. Dreuning et al. note that the best implementation of an application may be from another programming model that developers are unfamiliar with (Dreuning et al. (2018)). There is also a long history in high performance computing of using statistics to obfuscate application performance results or make numbers appear more favorable (Bailey (1991); Gustafson (1991); Pakin (2011)), which only makes matters more murky.

2.2.2.3 Platform Set Choice.

Another criticism from Dreuning et al. is the need for an additional metric for platform set diversity (Dreuning et

al. (2018)). If an application does well on one type of architecture (e.g., a Xeon Phi), then a platform set containing only that architecture will give artificially high $\mathcal{P}$. Indeed, in their original paper, Pennycook et al. note that their metric is only useful when the platform set is known (Pennycook et al. (2016)). Researchers using Pennycook et al.'s metric need to carefully consider their choice of platform set.

Ideally, applications would support every architecture, but there are often trade-offs between optimizing for performance and optimizing for performance *portability* – what's good for one platform is not necessarily good for another. Optimizing for one platform will either improve performance on all platforms, or improve performance on a subset of the platforms to the detriment of the others. Optimization is likely to improve performance portability overall, but may actually decrease it, and there's no way of knowing what will happen beforehand.

Based on these observations, Dreuning et al. give three ways to improve an application's performance portability:

- Add platforms to the set, especially ones similar to platforms it already performs well on, since the application is more likely to perform well on those, and/or require less work to perform well.
- Remove poorly performing platforms from the set, if there are sufficiently few of them and/or the optimization effort required would outweigh the performance benefits.
- Do the work of improving performance on some or all platforms.

2.2.3 Other Metrics for Performance Portability. Other metrics for performance portability have been proposed as alternatives or extensions to Pennycook et al.'s metric. This section will describe two recent proposals.

2.2.3.1 $\mathbf{PP_{MD}}$.

The PP_{MD} metric is motivated by the experiences of Sedova et al. (2018) investigating the performance portability of various molecular dynamics applications. The applications they looked at are all best-of-class (or nearly so) and get high performance on many HPC platforms (so they all score very highly in Φ under application efficiency). However, these applications got there with high-effort ports and are thus not truly performance portable, since migrating to a new platform would likely result in a great deal more work specific to that platform. Sedova et al. desired a metric that would take different source versions and program components into account, so they developed their PP_{MD} metric.

Sedova et al.'s metric is based on the sources of speedup in a particular code. If speedup comes mostly from portable program components, like standard libraries, the application's PP_{MD} value will be high. If non-portable components, such as CUDA kernels, are responsible for most of the speedup, PP_{MD} will be low. The mathematical definition of their metric is similar to Pennycook et al.'s, using the harmonic mean of the fraction of speedup from non-portable components:

$$PP_{MD}(a,p,Q) = \begin{cases} \dfrac{|Q|}{\sum_{i \in Q} S_i(a,p)}, & \text{if } G \neq Q \text{ and } Q \neq \emptyset \\ 1, & \text{if } Q = \emptyset \\ 0, & \text{if } G = Q \end{cases}$$

where G is the set of all program components that contribute to speedup for application a on input p, Q is the subset of G that is non-portable, and S_i is the speedup over baseline application performance that component $i \in Q$ is responsible for. Ideally, all program components responsible for speedup will be portable, so Q will be empty and PP_{MD} will be 1. PP_{MD} doesn't depend on any concept of "peak performance," which is helpful since measuring peaks is difficult, but

assumes that program components don't interact with or influence each other and can be measured independently, which may not be the case. Sedova et al. calculated PP_{MD} for each of their applications, and none scored particularly well; the best only got around 40%, which is because many of them use CUDA or other vendor-specific libraries (vendor implementations of standard libraries can vary in performance). Sedova et al.'s PP_{MD} metric might be a good companion to Pennycook et al.'s Φ, since it explicitly penalizes applications for using non-portable programming models (in other words, keeping divergent, machine-specific code versions), whereas Φ does not.

2.2.3.2 $\mathbf{P_D}$.

Pennycook et al.'s metric gives different values for the same application on different inputs, which can make it difficult to see whether an application is truly performance portable, since different platforms might do better on different problem sizes. Daniel and Panetta (2019) propose an alternative metric to better cope with problem size variations, which they call P_D, performance portability divergence. They compute P_D as the root-mean-square of the relative error in performance compared to the "best" performance (using either architectural or application efficiency):

$$P_D = \frac{\sum_{i \in H} \Delta_{RMS}}{|H|}$$

$$\Delta_{RMS} = \sqrt{\frac{\sum_{s \in S} \delta(a, \alpha)^2}{|S|}}$$

where a and α are the Φ values for two applications on the same input, $\delta(a, \alpha)$ is the relative error, S is the set of all input sizes, and H is the set of all platforms. Ideally, an application will perform identically on all inputs across all architectures, and performance portability divergence will be zero.

While this does give users more information on how an application behaves across problem sizes, Daniel et al.'s definition of application does not exclude programs with multiple source versions, so applications can still get good P_D scores by writing code specific to one architecture. A combination of this metric and PP_{MD}, perhaps, might be a better extension to Φ than this metric alone, since it's already been decided keeping multiple code versions is not a sustainable solution.

2.2.4 More History of Performance Portability. Past research placed the burden of performance portability solely on the application, but more current work has shifted it to languages, compilers, and other programming tools (Wolfe (2016b)). For example, when the developers of the Weather Research and Forecast (WRF) model were trying to make the application more performance portable across vector-based and RISC-based computers in the late 1990s/early 2000s (Michalakes, Loft, and Bourgeois (2001)), they primarily looked into reordering the loops to improve memory access patterns, not introducing a new library or compiler that would do it for them.[3] However, most application teams today are looking for solutions that won't require modifying their code (or at least, won't require modifying it more than once), which (interestingly) may involve writing an application-specific performance portability layer (see Sec. 2.5.2), going back to placing more of the burden on applications. This chapter will discuss many other potential solutions.

2.3 Productivity

Developer productivity is an important (though often overlooked) aspect and motivating factor of performance portability. If it takes years to port an

[3]Interestingly, they did discuss looking into source-to-source translators that would automatically reorder loops based on the type of machine, but at the time decided not to since they were able to get satisfactory performance without translation. This is an avenue new work is exploring though; see Secs. 2.7.3 and 2.8 for more, as well as the rest of this dissertation.

application to a high-performance portable model, that model is not as useful as another that can be adopted more quickly (even at the cost of lower performance), especially if the port cannot be done incrementally so the application can still be used in the meantime. Performance portable models should *increase* developer productivity overall and allow developers to spend less time writing (and rewriting) code during the porting process. This section will discuss issues relating to defining and measuring productivity, as it relates to HPC and performance portability.

2.3.1 Defining Productivity. Productivity is a very subjective, qualitative concept and it means different things in different contexts, making it difficult to define and measure. An intuitive definition might be "getting higher performance with less time/effort," but how do we define (and measure) effort?

2.3.2 Measuring Productivity. There have been numerous attempts to measure software developer effort and productivity across both industry and academia, but measuring *HPC* developer productivity is a very different problem. Most industry productivity measurement tools look at how much code a developer writes, but in HPC, developer productivity isn't just time spent writing code – it's time spent optimizing, tuning, parallelizing, and porting existing code.

There are two main ways to measure developer effort and productivity: direct logging (e.g., Wienke, Miller, Schulz, and Müller (2016) and Harrell et al. (2018)) and indirect approximations. Lines of code (LOC) and code divergence (how "different" code versions are) are common metrics for approximating code complexity, and hence, developer effort, but they don't take into account how difficult writing a single line of code can be (e.g., a complex OpenMP pragma versus a simple variable declaration) or development time not spent coding, such as time spent making performance measurements or designing new features

with team members (Wienke et al. (2016)). It is not always clear whether these approximations are accurate, and they don't include all of the productivity data we want, which is why direct logging approaches are also important.

Development time logs would be the best metric for developer effort, since they contain data on what developers were working on and what their results were, but this data is very difficult to collect. Developer diaries sound like a good idea in theory, but in practice there are numerous problems. They require a high level of commitment from developers to collect the type, regularity, and granularity of data that would be useful, but this often doesn't happen because developers don't want to take the time, are inconsistent in their entries, or simply forget. At the other end of the spectrum, automated activity monitoring software that tracks data such as keystrokes and applications used requires no commitment from developers, but misses out on development activities that happen away from the computer, such as planning and training, as well as performance data.

Wienke et al.'s direct logging method (Wienke et al. (2016)) tries to combat these problems by creating a journal that pops up at predefined intervals (so it can't be forgotten) and provides a short form with multiple choice questions (so a quick entry still has useful data) and an open-ended comment section. However, their tool doesn't collect performance data very often, and doesn't seem to associate that data with the particular code version it came from.

Harrell et al.'s direct logging method (Harrell et al. (2018)) was motivated by the desire to keep productivity and performance data *with* their code and better track changes over time, improving on Wienke et al. They wanted to minimize disruption for developers, so they integrated the logging process with their projects' version control system (`git commit`). Harrell et al. used these logs to confirm their

intuition about using approximations such as LOC and code divergence as proxies for developer effort; in particular, they demonstrate that low divergence ports (e.g., adding OpenMP or OpenACC directives) are often less expensive in terms of developer effort than high divergence ports (e.g., rewriting all compute loops as CUDA or OpenCL kernels).[4]

2.3.3 Productivity and Performance Portability. Harrell et al.'s primary criticism (Harrell et al. (2018)) of $\bar{\mathrm{P}}$ is similar to Sedova et al. (2018): it does not penalize applications that keep separate code versions for each platform, which are difficult to maintain and require much more developer effort to create. Indeed, one of the primary goals when developers were first porting applications to the Titan supercomputer (one of the first major accelerator-based machines) was to avoid "version bifurcation," which Joubert et al. equated with a loss of maintainability (Joubert et al. (2015)). Maintaining separate code paths or separate optimizations in a single version runs into similar problems. This is a good representation of the trade-off between programming for performance and programming for productivity and maintainability (Joubert et al. (2015)) – one goal of performance portability is to do away with this trade-off.

Harrell et al. suggest there is a need for a metric to measure developer effort to achieve performance. Such a metric could be used to penalize applications and programming models that require high amounts of developer effort to obtain and/or maintain performance portability. They note again that there are often

[4]As an example of this intuition, a survey of the CAAR teams, who were porting applications to Titan for the first time, revealed that 85% of the developers (most of whom were using CUDA, which is known for being difficult to adopt) felt the amount of effort to needed to get good performance on accelerator architectures was "moderate to high." Many expressed interest in moving to directive-based models like OpenACC (Joubert et al. (2015)), which is generally considered easier to adopt, even though there was little evidence to back this interest up at the time.

trade-offs between programming model abstraction levels (a proxy for portability) and performance. Models with higher abstraction levels are usually easier to port to new architectures, but have lower performance. Models with low levels of abstraction have better performance, but are much more difficult to port. A metric for productivity would help quantify this trade-off.

2.4 Some Non-(Performance) Portable Programming Models

The rest of this chapter will often mention various non-portable programming models (or non-performance portable models) as comparison points for the performance portable models discussed. These non-portable models have generally been highly tuned for their one architecture, and can provide a good estimate of peak application efficiency and performance. This section will describe the main non-portable models mentioned.

2.4.1 CUDA. CUDA (Nvidia (2023)) is Nvidia's proprietary C++-based GPU programming language, which naturally only works on Nvidia devices. CUDA allows developers to write compute kernels in extended C++ syntax that will be run on a machine's GPU. The downsides of CUDA are that (for optimal performance) it requires users to manually manage memory movement between the host and GPU, and the parallelism model for writing kernels can be non-intuitive for newcomers. This makes CUDA code difficult to read and debug, and porting an existing application to CUDA requires major changes and restructuring. Nvidia has released several libraries for CUDA, including cuBLAS (dense linear algebra), cuSOLVER (linear system solvers), cuDNN (neural networks), and Thrust (Bell and Hoberock (2012)), a high-level, productivity-oriented library containing general purpose algorithms, in an attempt to address these problems, however, CUDA is still not portable.

2.4.2 OpenCL. OpenCL (Khronos OpenCL Working Group (2023)) is a programming language similar to CUDA and developed by the Khronos Group, but as an open standard, it is implemented for far more devices. OpenCL supports GPUs, CPUs, and FPGAs from multiple vendors, including AMD, Intel, and Nvidia. OpenCL requires a great deal of boilerplate code to set up data structures and kernels, and kernels are represented as strings in the application, which makes debugging difficult. Like CUDA, OpenCL also has many libraries dedicated to reducing the severity of these problems. Unfortunately, since OpenCL is so low-level (arguably more so than CUDA), different code is required to get good performance on different architectures. While OpenCL is portable, it is not *performance* portable.

2.4.3 OpenMP 3. OpenMP (OpenMP Architecture Review Board (2011)) is an open standard for directive-based extensions to C, C++, and Fortran that enables developers to add parallelism to their applications by annotating their code with various pragmas. Early versions of OpenMP (≤ 3) were solely for CPUs, and since it is much simpler than almost every other parallel programming model available, OpenMP became very popular. Vendors heavily optimized their implementations, and as a result OpenMP is generally very high performance, but only on CPUs. OpenMP 4.0 began to add support for other architectures, including GPUs (mostly Nvidia GPUs) and Intel's Xeon Phi accelerator, making OpenMP 4+ a good candidate for being performance portable, and this is discussed in more detail in Sec. 2.7.1.

2.4.4 MPI and SHMEM. MPI (MPI Forum (2015)) is the most commonly used model for communication between processors and nodes on supercomputers. It is based on a two-sided message passing model, where both

processes need to be involved in sending and receiving messages. MPI is highly portable, and will generally get high performance on any machine without source code changes, making it very performance portable by some standards. However, MPI alone doesn't allow users to take advantage of all a machine's hardware (like accelerators), doesn't allow users to fine-tune their parallelism, and has some scalability problems (Thakur et al. (2010)), meaning MPI alone is not enough to write performance portable exascale programs.

SHMEM, standardized by OpenSHMEM (OpenSHMEM Contributors Committee (2017)), is a newer communication model, based on the partitioned global address space (PGAS) model, that is gaining popularity. SHMEM uses a one-sided message passing model, where only one process puts (or gets) data into (from) another process's memory space, without interfering with the other process's execution. Like MPI, SHMEM is highly portable, and in some ways performance portable, but it also cannot access all a machine's hardware on its own.

Both MPI and SHMEM need to be used with another programming model to become truly performance portable, but beyond that, both are too low-level to be very productive programming models. Many of the performance portable models below are built on top of MPI or SHMEM, but work at an even higher abstraction level to be productive as well.

2.5 Performance Portable Programming Models

This section will survey several performance portable programming models. The survey will be restricted to those relevant to the rest of this dissertation. For a larger survey, see Johnson (2020).

2.5.1 Libraries. This section describes two libraries designed specifically to provide performance portability across multiple domains.

```
/* Dot product */
/* create skeletons */
SkelCL::Reduce<float> sum (
    "float sum (float x,float y){return x+y;}");
SkelCL::Zip<float> mult(
    "float mult(float x,float y){return x*y;}");

/* allocate, initialize host arrays */
float *a_ptr = new float[ARRAY_SIZE];
float *b_ptr = new float[ARRAY_SIZE];
fillArray(a_ptr, ARRAY_SIZE);
fillArray(b_ptr, ARRAY_SIZE);

/* create input vectors */
SkelCL::Vector<float> A(a_ptr, ARRAY_SIZE);
SkelCL::Vector<float> B(b_ptr, ARRAY_SIZE);

/* execute skeletons */
SkelCL::Scalar<float> C = sum( mult( A, B ) );

/* fetch result */
float c = C.getValue();
```

Listing 2.1 SkelCL code sample.

2.5.1.1 Skeletons. Skeleton programming is a concept borrowed from functional programming, where higher-order functions can take other functions as arguments to specialize their operation. Since imperative languages are much more common in HPC, developers have begun implementing them in these languages as well (Ernsting and Kuchen (2014)). Skeleton libraries tend to be modeled as arbitrary task graphs that provide various communication and computation patterns (skeletons) as higher-order functions. These skeletons generally implement data parallel patterns, but not loop parallelism, and make use of various distributed data structures.

SkelCL. SkelCL (Steuwer, Kegel, and Gorlatch (2011)) (code sample in Listing 2.1) is a high level library for programming heterogeneous systems, built on top of OpenCL. The primary design philosophy of SkelCL is to abstract away all the tricky parts of writing (multi) GPU code to make programming easier. SkelCL provides a modest set of data parallel skeletons, which users can specialize with their own C-like functions, including `Map` (apply a function to all elements of a collection), `Zip` (apply a function to a pair of collections), `Reduce` (condense the elements of a collection to a single value), and `Scan` (prefix-sum). Later works (Breuer, Steuwer, and Gorlatch (2014); Steuwer and Gorlatch (2013)) add `MapOverlap` (simple stencil), `Stencil` (more complex stencils), and `AllPairs` (apply a function to each pair of elements in two sets; n-body) skeletons to better support some linear algebra applications. The two different stencil-like skeletons allow users more flexibility in defining their stencil computations – e.g., `MapOverlap` uses padding around the edges of the vector/matrix to minimize branching, while `Stencil` allows users to specify the number of iterations to run the stencil and various synchronization properties. All skeletons can take additional arguments aside from the defaults to give users even more flexibility. For example, if a user wants to define a generic "add x" function for vectors, they can use the map skeleton plus an additional argument for x.

SkelCL also provides a generic vector class that works with both CPU and GPU code and hides data transfers from users – the vector class will do data copies lazily (only when data is actually used) to eliminate unnecessary data transfers. The vector class can also automatically distribute data to multiple GPUs based on a set of predefined data distribution patterns, including `single` (only one GPU gets a copy), `block` (split evenly among all GPUs), and `copy` (each GPU has a full

copy) (Steuwer, Kegel, and Gorlatch (2012)). A later paper (Steuwer and Gorlatch (2013)) adds an `overlap` distribution that automates halo exchanges for stencils, as well as a generic 2-dimensional matrix class that supports the same distributions along the rows of the matrix.

All skeletons support all data distributions, and the GPUs automatically cooperate on `block` and `overlap` distributed data structures. If the user doesn't specify a data distribution with multiple GPUs, each skeleton has preferred distributions for their pre-defined arguments, but the user or runtime can override this (for additional, user-defined arguments, the user must always specify a distribution since the runtime can't reason about the semantics of user-defined arguments).

2.5.1.2 Parallel Loop Libraries.

This section will discuss a library based on parallel loop abstractions. These types of libraries mostly fall under the data parallelism and index space abstraction strategies, and many use C++ templating to provide a more generic interface. In general, these libraries do not provide computational patterns, and users must write all computation themselves.

Kokkos. Kokkos (Edwards and Sunderland (2012); Edwards et al. (2014)) is a loop-based data parallelism library for performance portability that uses C++ templating capabilities. Kokkos has back ends for various other programming models, including CUDA, OpenMP, and pthreads, which allow users to target a wide variety of architectures, such as Intel's Xeon Phi and Nvidia GPUs, without modifying their code. These back ends can be swapped out with minimal changes to application code so users can easily run on multiple architectures.

Kokkos abstracts away parallelism so architecture-specific optimizations exist outside the main application. Kokkos' abstractions can be loosely grouped

```
// Heat conduction stencil (TeaLeaf)
#define DEVICE Kokkos::OpenMP

// initialize a view
Kokkos::View<double*, DEVICE> p("p", x*y);

// lambda-style loop
Kokkos::parallel_for(x*y, KOKKOS_LAMBDA (int index) {
    const int kk = index;
    const int jj = index / x;

    // if in view interior...
    if (kk >= pad & kk < x - pad &&
        jj >= pad && jj < y - pad) {

        // recalculate value
        p(index) = beta*p(index) + r(index);
    }
});
```

Listing 2.2 Kokkos code sample.

into memory and execution abstractions. Memory layout and execution environment are the most common program elements that need to change for each new architecture, so the Kokkos developers chose to abstract them away to decrease porting effort.

Kokkos' abstractions are multidimensional array views (array layouts) (Edwards, Sunderland, Amsler, and Mish (2011)), memory spaces, and execution spaces. The array views allow developers to change the memory layout and access patterns to best suit the underlying architecture at compile time, so the compiler can make inlining and vectorizing optimizations. Memory spaces keep track of where data resides in heterogeneous architectures, and similarly, execution spaces contain information on the type of hardware code should run on. Execution spaces only have access to memory spaces that make sense; e.g., a CPU execution space

cannot access a GPU-only memory space, but can access a host burst buffer memory space. All of these can be modified to add optimizations for a particular application on a specific architecture. For example, users could write their own tiled matrix layout and add it to an application by swapping out array views. Kokkos keeps these configurations consolidated and out of application code, which is excellent for portability and productivity.

2.5.2 Application-Specific Libraries. In a recent paper, Holmen, Peterson, and Berzins (2019) suggest a way for large legacy codes to incrementally adopt performance portability models by adding a dedicated, application-specific *performance portability layer* (PPL), which acts as an application-specific performance portability library. The primary goal of a PPL is to improve long-term portability for legacy code, or even a newer application that is expected to have a long lifetime. A PPL insulates users from changes in the underlying programming models by consolidating all performance portability-related code into a single place – the application only needs to port to the PPL once, and all future ports take place inside the PPL. This allows applications to experiment with multiple programming models without disrupting users and domain developers. If the user base for the application's chosen model(s) dies out (as the user bases for some early programming models did (Han and Abdelrahman (2009))), the application can migrate to new models in the PPL, not in the application code. The PPL also allows an application to specialize its use of performance portable models.

Holmen et al. describe the adoption of a PPL into the Uintah fluid-structure interaction simulator. The Uintah PPL is based heavily on Kokkos and contains parallel loop abstractions similar to Kokkos' (including an iteration range, execution policy, and lambda kernel function), as well as application-level tags that

denote which loops support which back ends (e.g., CUDA vs. OpenMP), and build-level support for selective compilation of loops that enables incremental refactoring and even simultaneous use of multiple underlying models. Currently, their PPL only supports Kokkos, but the authors see no reason why it couldn't also support other models.

While their PPL is very similar to Kokkos, the developers didn't want to directly adopt Kokkos because they wanted to preserve legacy code, simplify the abstractions for their domain developers, and make re-working their implementation or adopting another performance portable model later easier. They've succeeded at some of these goals already, since their domain developers have liked the new parallel loop interface and haven't had much trouble with it, even though they don't know much about parallel programming. The new portable implementation also demonstrates better speedup (generally at least 2x) and scalability on both CPU and GPU architectures.

Since the Uintah code base is so large (over one million lines of code), the authors did several small-scale refactors to validate their PPL and standardize an adoption process, and came up with general advice for porting to performance portable models:

- Put a build configuration system in place before fully migrating to a PPL to avoid additional refactors.

- Include a tagging system for loops to let the build system know which loops have PPL implementations for which interfaces; this will make porting go much smoother.

– Have a thorough testing apparatus in place to verify correctness pre-, during, and post-port.

The Uintah developers also acknowledge that, while adding a PPL solves many problems, it also raises new questions, including how to use domain libraries (e.g., PETSc) in both the application and PPL simultaneously, how to manage increasingly complex build configurations, how to make intelligent, optimal use of underlying programming models, and how to efficiently manage memory and execution while potentially using multiple underlying programming models. In some ways, adding a PPL is moving responsibility for performance portability back to application teams, where it was originally (see Sec. 2.2.4), though with a PPL the performance and portability aspects of the code are separated from its functionality. This raises yet more questions about the roles of applications, compilers, and programming models in performance portability.

2.6 Parallel (C/C++-like) Languages

In an ideal world, sequential languages could automatically be parallelized to get high performance on modern machines, but this is an exceptionally difficult problem, and debatably not one worth solving,[5] so numerous explicitly parallel languages have been created instead. These languages vary widely in their feature sets and parallel constructs, from high-level languages like Chapel to low-level languages like OpenCL. The barrier to entry for new languages is higher than for all the other models discussed in this paper, since developers of new applications are reluctant to use a language that might not exist in a few years, and incremental

[5] As some have noted (Chamberlain, Callahan, and Zima (2007)), sequential and parallel programming are fundamentally different, so why should we try to use the same tools for both?

```
// Buffer example
int b[N], c[N];

// put data in buffers
buffer<int, 1> B(b, range<1>{N});
buffer<int, 1> C(c, range<1>{N});

// submit this to a device queue
myQueue.submit([&](handler& h) {
    // request buffer access
    auto accB = B.get_access<access::mode::read>(h);
    auto accC = C.get_access<access::mode::write>(h);

    // create kernel
    h.parallel_for<class computeC>(range<1>{N}, [=](id<1> ID)
   {
        accC[ID] = accB[ID] + 2;
    });
});
```

Listing 2.3 SYCL code sample with buffers.

porting to a new language is difficult if not impossible (Wolfe (2011)). This section describes a language that is built on C/C++ and beginning to break into HPC.

2.6.1 SYCL and DPC++. SYCL (Wong, Richards, Rovatsou, and Reyes (2016)) (code sample in Listing 2.3) is a high-level language built on OpenCL and designed to improve OpenCL's usability so it can be used to write single-source C++ for accelerators. This means host and device code live in the same source file, which can be analyzed and optimized by the same compiler for better performance. SYCL is built on C++14 and intended to follow the C++ specification as much as it can, so it does not change or extend C++ syntax in any way (i.e., a CPU-only implementation could work with any C++ compiler). It allows C++ libraries to work with OpenCL, and allows OpenCL kernels to work with C++ features, such as templates and lambdas. SYCL is intended to be easy to use while still giving the

performance and control of OpenCL, and focuses on intra-node parallelism, leaving inter-node parallelism to other models like MPI.

SYCL takes an explicit data parallelism approach, where the user declares what can be run in parallel with Kokkos-style `parallel_for` templated function calls. Kernels can be written as lambdas, functors, OpenCL code, or loaded as SPIR binaries. SYCL can output kernels at compile time as device-specific executables, or as SPIR, so kernels can be compiled at run time for any device. To manage data movement, SYCL uses buffers, which are an abstraction over C++ objects, and can represent one or more objects at any given time (as opposed to a specific memory location). Inside a kernel, users can request access to a buffer with a set of permissions, and SYCL will automatically handle data movement to and from the device based on those permissions. While this can be cumbersome, it does allow SYCL to optimize data transfers to the device, instead of copying "just in time."

SYCL kernels are run by submitting them to device queues, which allow for some task parallelism, since by default there is no ordering enforced between kernels in different queues. Each device can have multiple queues running kernels on it (although each queue can only be bound to one device). The data dependencies and access permissions of each kernel are used to enforce an ordering on each queue, or users can explicitly specify dependencies between kernels. SYCL queues only specify which kernels *can* be run in parallel, but provide no guarantees about what kernels will be run in parallel.

Data Parallel C++ (DPC++) (Reinders et al. (2019)) (code sample in Listing 2.4) is a language from Intel built on top of SYCL (it's "SYCL with extensions"). Intel's new oneAPI (Intel Corp. (2020)) is also built primarily on

```
// Shared memory example
// create host and shared arrays
int* hostArray = (int*) malloc_host(N*sizeof(int));
int* sharedArray = (int*) malloc_shared(N*sizeof(int));

// submit this to a device queue
myQueue.submit([&](handler& h) {

    // create kernel
    h.parallel_for<class myKernel>(range<1>{N}, [=](id<1> ID)
   {
        int i = ID[0];

        // access shared and host array on device
        // shared array will be copied over; host array will
   not
        sharedArray[i] = hostArray[i] + 1;
    });
});
```

Listing 2.4 DPC++ code sample with shared memory.

DPC++, with many additional libraries and interoperability with languages other than C++. DPC++ is intended to be a place to experiment with new parallelism features that could be added to the C++ standard. Some of the features already added include support for hierarchical parallelism inside kernels, ordered device queues, per-device versions of kernels, and unified shared memory.

Unified shared memory is perhaps the most interesting of these. It removes the need for specifying data use permissions or manually managing data movement by allowing users to create host, device, and shared buffers via `malloc_host`, `malloc_device`, and `malloc_shared` functions. Device buffers are allocated only on the device and can't be accessed by the host, while host buffers are allocated only the host, but can be accessed by the device *without being transferred*, so accesses are likely to be very slow. Data can be explicitly moved by copying a host buffer

```
// Heat conduction (TeaLeaf)
// set up data environment, copy r and p to device
#pragma omp target data map(to: r[:r_len]) map(tofrom:
   p[:p_len])
{
    // describe loop parallelism
    #pragma omp target teams distribute collapse(2)
    for (int jj = pad; jj < y-pad; ++jj) {
        for (int kk = pad; kk < x-pad; ++kk) {
            const inst index = jj*x + kk;
            p[index] = beta*p[index] + r[index];
        }
    }
} // p gets copied back
```

Listing 2.5 OpenMP code sample.

into a device buffer inside a kernel, or vice versa. Shared buffers can migrate data automatically between the host and device whenever the data is accessed, so the first few accesses are slow, but afterwards accesses can happen from fast device memory. This greatly lessens the burden on the developer, so they can quickly prototype an application, then optimize memory transfers as necessary.

2.7 Directive-based Models

Directives are language extensions, although sometimes they're considered languages in and of themselves. Directives are annotations for base languages, usually C, C++, and Fortran, that allow users to give the compiler extra information, such as which loops can be parallelized or what arrays need to be moved to the GPU. While some (Peccerillo and Bartolini (2017); Steuwer and Gorlatch (2013)) consider directives to be low-level because users must still specify data movement and describe parallelism themselves, directives are still more concise, portable, and high-level than models like OpenCL or pthreads. This section will describe the two most popular directive-based models.

2.7.1 OpenMP.

As described in Sec. 2.4, OpenMP (OpenMP Architecture Review Board (2011, 2015, 2018)) (code sample in Listing 2.5) began as a way to simplify programming shared memory CPUs, specifically targeting parallel loops with directives to tell the compiler how loops should be parallelized. (Version 3 also added directives for task-based parallelism.) Version 4.0 began adding support for heterogeneous, accelerator-based computing and performance portability features, a trend which has continued for Version 5.0 – both of these are discussed further below. OpenMP follows a more "prescriptive" programming model, where developers explicitly define how their code should be mapped onto the hardware for parallel execution, but is beginning to add some "descriptive" directives that give the compiler more freedom. (This idea will be revisited in Sec. 2.7.2 on OpenACC.)

OpenMP's original goals when it was created in the mid-1990s were to provide portable, consistent, parallel, shared-memory computing for Fortran, C, and C++, maintain independence from the base language, make a minimal specification, and enable serial equivalence (de Supinski et al. (2018)). OpenMP has mostly kept to these goals, with the exception of adding support for models other than data parallelism, which has made it difficult to keep the specification small, and abandoning serial equivalence as unrealistic for modern architectures.[6] There is an ongoing debate about which parallelism models OpenMP should support, and exactly how many basic programming constructs should be added to OpenMP to support these other forms of parallelism; some expect that OpenMP will move closer to becoming a general-purpose language in its own right in the future (de Supinski et al. (2018)).

[6] A subset of OpenMP does maintain serial equivalence, but keeping to this subset severely restricts what programmers can do.

2.7.1.1 OpenMP 4.x.

As mentioned earlier, OpenMP's primary abstraction is parallel loops, and OpenMP provides directives for users to describe how their loops should be mapped onto parallel hardware. OpenMP 4.0 added `target` and `map` directives to denote that code regions and data should be offloaded to an accelerator, as well as various clauses to modify how code is mapped to the device. Version 4.0 also added the `simd` directive to force vectorization (auto-vectorization can vary by compiler and greatly influence performance) and improvements for task-based parallelism and error handling (de Supinski et al. (2018)).

OpenMP 4.5 further improved accelerator support by adding unstructured data regions, so `map` directives can be moved into functions, which improves readability and usability. However, OpenMP 4.5 also modified how certain types of data (e.g., scalars) are copied onto the device, which can make porting between OpenMP 4.0 and 4.5 error-prone, as noted by Martineau and McIntosh-Smith (2017). The 4.x specification also does not define support for copying pointers in data structures to the device ("deep copy" support), but some compilers still support it, which can make porting between compilers difficult.

OpenMP 4+ has been struggling with implementation support. Even though the 4.0 specification was released in 2013, it has taken many years for some compilers to offer even basic support for offloading directives, and performance can still vary wildly between compilers. As Martineau et al. note, developers who were used to consistent high performance from OpenMP 3 will be in for a surprise, and may be better served by waiting until compiler vendors have had more time to improve their implementations.

2.7.1.2 *OpenMP 5.x.*

OpenMP 5.0 adds several new features specifically for performance portability and to make supporting multiple architectures easier. Version 5.0 adds `metadirective`, `declare variant`, and `requires` constructs that allow users to denote that some directives or functions should only be used on certain hardware. While semantically similar to preprocessor directives like `#ifdef`, these constructs give the compiler more information to reason about which version should be used, instead of naively copy-pasting code (Pennycook, Sewall, and Duran (2018)). Version 5 also adds support for deep copying (to handle data structures with pointers, as mentioned above), iterator-based ranges, and interacting with the memory hierarchy, all of which are becoming more widely used in modern code.

Version 5.0 is also adding some more "descriptive" constructs, such as `loop` (similar to OpenACC's `parallel loop` construct) and `order(concurrent)`, so users can opt to let the compiler make choices for them. One goal of OpenMP 5.0 is not to interfere with threading and memory models that are currently being added to the base languages, so allowing the compiler (which should know more about these models) to make more choices could be helpful.

2.7.1.3 *Future OpenMP.*

OpenMP is still evolving, and there are many features that may be added in the future (de Supinski et al. (2018)). Some of these features include: data transfer pipelining, memory affinity, user-defined memory spaces, event-driven programming, and lambda support. The standards committee is also considering adding "free-agent" threads that can help with load balancing by joining parallel regions.

2.7.1.4 *OpenMP 3 to GPGPU.*

As an interesting aside, even before OpenMP moved to officially support heterogeneous computing, there were

```
// Head conduction (TeaLeaf)
// set up data environment, copy r and p to device
#pragma acc data copyin(r[:r_len]) copy(p[:p_len])
{
    // describe loop parallelism
    #pragma acc kernels loop independent collapse(2)
    for (int jj = pad; jj < y-pad; ++jj) {
        for (int kk = pad; kk < x-pad; ++kk) {
            const inst index = jj*x + kk;
            p[index] = beta*p[index] + r[index];
        }
    }
} // copy p back
```

Listing 2.6 OpenACC code sample.

efforts to enable OpenMP-based GPU computing. Lee et al. (Lee and Eigenmann (2010); S. Lee, Min, and Eigenmann (2009)) created OpenMPC, a source-to-source translator from OpenMP 3 to CUDA. Their translator took in OpenMP code, transformed it so it was better organized for the CUDA programming model, then translated parallel loops into optimized CUDA kernels with the appropriate data transfers. They provided extra directives so users could control and further optimize the translation into CUDA, if desired. Lee et al. compared their optimized, generated CUDA against hand-tuned CUDA, and found that the average performance gap was less than 12%. They noted that the generated CUDA took significantly less effort to make, demonstrating that directives can greatly increase productivity, if developers are willing to take a small performance hit (though hopefully in the future, performance will be more similar).

2.7.2 OpenACC. OpenACC (OpenACC Standards Group (2011, 2018, 2019)) (code sample in Listing 2.6) is a directive-based model originally designed for GPU computing, although there are now implementations that target multicore, Xeon Phis, and FPGAs (Lambert, Lee, Kim, Vetter, and Malony (2018);

S. Lee and Vetter (2014a, 2014b); Wolfe et al. (2017)). OpenACC's main goals were to enable easy, directive-based, portable accelerated computing (which OpenMP didn't have at the time) and to merge several individual efforts into a single standard. OpenACC came out of a combination of CAPS' OpenHMPP (Dolbeau, Bihan, and Bodin (2007)), PGI Accelerator (Wolfe (2010)), and Cray's extensions to OpenMP. Like OpenMP, OpenACC supports C, C++, and Fortran as base languages.

Unlike OpenMP, however, OpenACC follows a more "descriptive" approach, whereas OpenMP is "prescriptive." OpenMP requires developers to explicitly define how parallelism is mapped onto hardware, while OpenACC leaves most choices up to the compiler. There has been much discussion about which approach is best for performance portability, but in truth (as noted by de Supinski et al. (2018)), this is a false binary, and these models exist on a spectrum. While it would be nice if users could add a few descriptive directives and have things "just work," often that isn't possible and prescriptive models are still necessary. OpenACC has recently begun adding more prescriptive constructs so users can have more control over tuning their code.

2.7.2.1 OpenACC 2.x. The original OpenACC standard had fairly basic directives for annotating parallel loops and describing data movement. The 2.x versions added support for asynchronous compute regions, function calls within compute regions, atomics, an interface for profiling tools, and other usability improvements, such as modifying the semantics of `copy/copyin/copyout` to include checking whether data was already present to minimize unnecessary data transfers. Basic support for user-defined deep copy operations on data

structures containing pointers was also added, which is very important for scientific applications that use deeply nested data structures.

2.7.2.2 OpenACC 3.0 and Future Versions. Version 3.0 adds more support for multi-GPU configurations, which are becoming more common, and lambdas in compute regions. It also introduces somewhat stricter rules for which parallelism clauses can appear together. OpenACC is generally adding more prescriptive options, such as loop scheduling policies and optimization directives (e.g., `unroll`), to allow users to make decisions where the compiler can't.

2.7.3 Customizable Directives. In addition to the standardized and otherwise pre-packaged directives described above, there have been efforts to allow users to define their own, specialized directives. Allowing users to define their own directives can give them more control over their application. This section will describe a framework that helps users define their own directives and an example of how user-defined directives can improve performance portability.

2.7.3.1 Xevolver. Xevolver (Takizawa et al. (2014)) is a source-to-source translation tool built on top of ROSE (Quinlan and Liao (2011)) (see Sec. 2.8.3) that allows users to define their own (parameterized) code transformations and directives to specify where those transformations should be applied. The main goal of Xevolver is to separate optimizations/transformations from application code – these transformations can be kept outside application code bases, which removes the need to keep platform-specific versions of code and consolidates knowledge about how to optimize for a given platform. Transformations can be shared across applications as well, which reduces the need to re-implement optimizations for each program.

The authors of Xevolver note that existing solutions for writing specialized code for each architecture aren't sufficient, since they generally involve keeping separate source versions or directly modifying application code. As an example, using C preprocessor macros to conditionally compile different code versions (even ones kept in the same file) can quickly devolve into "the so-called `#ifdef` hell" (Takizawa et al. (2014)). Users want specialized code, but they don't want to *write* specialized code, so having a set of transformations to pull from could be extremely helpful.

Performance tests of Xevolver-enhanced applications on various architectures confirm that adding these transformations helps performance. One motivation for creating Xevolver was to help users with applications optimized for vector machines port to other architectures (Xevolver also enables incremental porting). The authors demonstrated that it takes a set of non-trivial but consistent transformations to port these codes, and that transformations that help one architecture can be detrimental to another. Since these transformations don't actually modify the source code, they help make applications more performance portable.

Xevolver has been used to port part of a weather simulation to OpenACC (Komatsu et al. (2016)) and migrate a numerical turbine code (Suda, Takizawa, and Hirasawa (2016)), among other things.

2.7.3.2 The CLAW DSL.

The CLAW DSL (Clement et al. (2018)) is an example of application specific, user-defined directives meant to enable performance portability for weather and climate models. CLAW is based on the Omni source-to-source Fortran translator (Murai et al. (2018)) (see Sec. 2.8.2) and takes advantage of certain domain properties of most climate modeling programs.

The CLAW compiler outputs OpenMP or OpenACC annotated Fortran and is interoperable with normal OpenMP or OpenACC code to enable incremental porting.

Climate models have been using OpenMP and OpenACC for performance portability, but different architectures require different directives for optimal performance. The differences are consistent, however, so the DSL provides directives to abstract away these differences.

The CLAW port of a portion of one climate model outperformed the original serial implementation and a naive OpenMP implementation, and matched the corresponding hand-tuned OpenACC implementation. This proof of concept demonstrates how user-defined transformations can improve the performance portability of an application.

2.8 Source-to-source Translators and Existing Rewriting Tools

This section will discuss source-to-source translators specifically designed to improve the performance portability of applications, similar to the work done by the rest of this dissertation. Source-to-source translators are considered here to be programs that transform one input language into another, but *do not themselves* compile it down to an executable; i.e., translators need another base compiler to work.

Several translators have been mentioned already, particularly when discussing the implementations of some directive based models in Sec. 2.7. This section will discuss how those translators work, as well as some other frameworks for code translation for performance portability. We will compare our work to these in the summary in Chapter VIII.

2.8.1 Early Translators.

Source-to-source translators are not a new concept, and several that did not directly address performance portability were introduced that could nevertheless improve it. This section will discuss a couple of those.

2.8.1.1 Qilin.

The Qilin compiler (Luk, Hong, and Kim (2009)) was intended to solve the problem of load balancing computation on heterogeneous systems – in other words, to decide what fraction of computation should happen on the CPU vs. on the GPU. Even for a single application, the ideal fraction can change for different inputs, and, of course, different hardware. Qilin automates this mapping process by doing it adaptively at run time.

Qilin provides two APIs for writing parallel applications; the compiler doesn't have to extract parallelism, only map it to the hardware. The first API is the stream API, which provides data parallel algorithms (similar to skeleton libraries), and the second is the threading API, which allows users to provide parallel implementations in the underlying programming models, Intel TBB and CUDA. The compiler dynamically translates these API calls into native code and decides a mapping using an adaptive algorithm.

This algorithm is based on a database of execution time projections that Qilin maintains for all programs it has seen. The first time Qilin sees a program, it builds a model of how that code performs when different percentages of work (per kernel) are run on the CPU vs. the GPU. For future runs (even on different problem sizes), Qilin refers to that model to find the mapping that will minimize run time. This adaptive mapping is always faster than GPU-only or CPU-only execution, and within 94% of the best manual mapping (at granularities of 10%). This kind of adaptive mapping can improve performance portability by providing

good performance regardless of architecture changes, and it improves productivity by automating the process.

2.8.1.2 R-Stream. The R-Stream compiler and translator for C (Meister et al. (2009); Schweitz, Lethin, Leung, and Meister (2006)) is somewhat unique among all the other models discussed here, in that it requires no source code modifications at all, not even annotations like OpenMP or Bones (see Sec. 2.8.4). R-Stream has been used to target several processors and accelerators, including IBM's Cell processor, ClearSpeed's processors, and Nvidia GPUs (Leung et al. (2010)). The compiler takes in unmodified C and outputs C with parallelizable portions in the chosen parallel back end. R-Stream is based partially on the polyhedral optimization model.[7] Its general compilation flow is as follows:

- Parse in C, translate it to a static single assignment (SSA) internal representation (IR).
- Run optimizations.
- Run analyses to determine which portions could be mapped onto an accelerator.
- "Raise" map-able portions into a polyhedral IR and optimize them under the polyhedral model to find parallelism.
- Lower map-able portions back to SSA IR.
- Emit non-mapped code as part of the master thread, and emit mapped code in target language (e.g., CUDA).

[7]It is not important to understand the polyhedral model for this dissertation, but more information can be found in Griebl, Lengauer, and Wetzel (1998).

Interestingly, R-Stream generates the best parallel code when given what the authors call "textbook" C code – code without any optimizations or clever implementation strategies, just the basic algorithm as you might find in a textbook. This implies that, to create performance portable code, less is more, and simple, high-level expressions of algorithms can be more useful (in some ways, at least) than optimized versions.

2.8.2 Omni. The Omni compiler (Murai et al. (2018)) is a source-to-source translator for Fortran and C (C++ is in development) that is used by XcalableMP (Nakao, Lee, Boku, and Sato (2012)), XcalableACC (Nakao et al. (2014); Tabuchi, Nakao, Murai, Boku, and Sato (2017)), and the CLAW DSL, among others. Omni is based on the idea of metaprogramming: it allows users to write code to transform their code. Not all compilers support every optimization, or can determine whether an optimization is safe, so metaprogramming allows users to transform their code so it is easier for compilers to analyze or to directly apply optimizations themselves.

Omni is composed of three main pieces: a front end, which parses in C and Fortran and turns them into XcodeML, Omni's IR (based on XML); a translator, which turns the XcodeML IR into Xobject (Java-based XML objects) and applies transformations to it; and a back end, which translates XcodeML back into C or Fortran which can be compiled by a regular compiler. Omni could theoretically support multiple "meta-languages" for users to define what transformations should be done, and where, including an Xobject-based meta-language, an XML-based meta-language, or a new DSL; however, since the Xobject-based meta-language was simplest to implement, Murai et al. chose to only implement that one. To define a transformation in their Xobject-based language, users must write a Java class

that implements an Omni-specific interface, which describes the transformation to perform on the Xobject(s). To choose where that transformation is applied to their code, users add an Omni directive to their application. This is how XcalableMP and XcalableACC, two other directive-based models, were implemented.

Unfortunately, this interface isn't terribly user-friendly, especially for non-compiler-expert users and domain scientists, although the vast majority of high-level compiler optimizations, including loop unrolling and array-of-struct to struct-of-array transformations, can be defined using it. Murai et al. realize this, though, and note that future work includes building a nicer interface. Perhaps another solution would be for compiler experts to write an open source collection of transformations that the HPC community could use and modify for their own purposes.

2.8.3 ROSE. The ROSE translator (Quinlan and Liao (2011)) is different from other translators listed here in that it is designed to specifically support analysis and optimization for source code *and* binaries. The ROSE front end supports many languages, including C, C++, Fortran, Python, OpenMP, UPC, and Java, as well as both Linux and Windows binaries. ROSE is meant to support rewriting large DoE applications for future architectures and programming models, as well as research into new compiler optimizations, automatic parallelization, software-hardware codesign, and proof-based compilation techniques for software verification. ROSE has multiple levels of interfaces for working with source code ASTs, and many optimizations and analyses have been implemented to help users modernize their code.

One of the more interesting features of ROSE, from a compiler design perspective, is its IR, which is based on the Sage family of IRs. Even though ROSE

supports a wide variety of languages with very different feature sets, around 80% of the IR is shared between all these languages, and only 10% is for special cases. That such a variety of languages can all be represented by one IR is very promising for compilers that wish to support multiple programming models.

Some projects that have been built on ROSE include Xevolver (see Sec. 2.7.3.1), a fault-tolerance research tool (Lidman, Quinlan, Liao, and McKee (2012)), a benchmark suite for data race detection (Liao, Lin, Schordan, and Karlin (2018)), and more.

2.8.4 Bones. Bones (Nugteren and Corporaal (2014); Nugteren, Custers, and Corporaal (2013)) is a source-to-source compiler for C, and targeting OpenMP, OpenCL, and CUDA, based on the concepts of algorithmic species and skeletons, similar to the skeleton library described earlier in Sec. 2.5.1.1. Nugteren et al. (Nugteren and Corporaal (2014); Nugteren, Custers, and Corporaal (2013)) criticize existing compilers for parallelism as lacking in one of these three aspects: they aren't fully automatic and require users to change their code, they produce binaries or otherwise non-human-readable code, or they don't generate highly efficient code. Bones is intended to fix these problems by using algorithm species[8] and knowledge of traditional compiler optimizations to drive a skeleton-based translation process.

The Bones translation process goes like this:

1. Extract algorithmic species information from source code, either by hand or (preferably) using an automated tool like A-Darwin (Nugteren, Corvino, and

[8] Algorithm species are similar to skeletons, but at higher granularity – species describe memory access patterns on all data structures in a particular loop nest. See Nugteren et al. (Nugteren and Corporaal (2014); Nugteren, Custers, and Corporaal (2013)) for more.

Corporaal (2013)) or ASET (Custers (2012)), and add species annotations to source code.

2. Pass annotated source code to BONES compiler, which uses species annotations to pick appropriate algorithmic skeletons.

3. BONES uses skeleton information to perform source code optimizations and outputs transformed C code.

BONES skeletons are not quite like the skeleton functions found in the library in Sec. 2.5.1.1, but are more like pieces of template or boilerplate code into which the compiler can insert pieces of user code; e.g., types are left as parameters, loop bodies are empty, and so on. Multiple species can map to a single skeleton, since species provide much higher granularity than skeletons, in general, and the number of species is near infinite.

BONES can do many different optimizations based on skeleton information, such as multi-dimensional array flattening, loop collapsing, and thread coarsening. The optimizations BONES does based on the skeleton information can also be target-dependent. For example, when targeting a GPU (either with OpenCL or CUDA), BONES can do data analysis on the species (kernels) identified to determine which data needs to be moved to the GPU, as well as optimize data transfers and synchronization events. One uncommon benefit of BONES' optimizations is that they need not be just permutations of the original code – BONES can add extra code to, e.g., ensure data accesses on GPUs are properly coalesced. However, BONES does still rely on the optimizations the underlying C compiler can do, since not all optimizations (like vectorization) can be written

as skeletons, and not all performance relevant data (like register pressure) can be contained in species or skeletons.

2.8.5 OpenACC to OpenMP. There have been several proposals for translating between OpenACC and OpenMP, including those from Pino, Pollock, and Chandrasekaran (2017), Sultana, Calvert, Overbey, and Arnold (2016), and Denny, Lee, and Vetter (2018). Many current machines have only one of OpenMP or OpenACC, or have a poor implementation of one but a good implementation of the other, so being able to translate between them would alleviate the problems this causes. However, because of semantic differences between OpenMP and OpenACC, mechanical translation from OpenMP to OpenACC is generally not possible or advisable (Wolfe (2016a)).

OpenMP was designed when most machines used multi-processor architectures only (as opposed to the modern CPU+GPU), and processor vendors wanted to provide a unified interface for programming multi-processors. Therefore, the meaning of each OpenMP directive was very important and the OpenMP specification has a detailed, prescriptive definition of what each one means. OpenACC, on the other hand, was designed when there were many diverse, heterogeneous architectures, so the OpenACC specification decided to leave many more (architecture-specific) choices to the compiler and only provide a descriptive definition of what each directive does/means. OpenMP also has many synchronization primitives and atomics, which OpenACC does not; some OpenMP concepts simply cannot be expressed in OpenACC. Going from OpenACC to OpenMP, though, is possible, as this direction doesn't have these problems.

While there is a simple, one-to-one mapping for OpenACC and OpenMP data directives, the same is not so for compute directives. Even though many

OpenMP directives seem similar to OpenACC, they have different definitions and meanings. For example, OpenMP can't vectorize within a thread with only `parallel for` unless the user specifies it (this is why OpenMP needed to add the `simd` directive), while OpenACC can – the `parallel loop` directive guarantees there are no data dependencies across iterations. This means that translating loops from OpenMP to OpenACC is non-trivial and requires data dependence analysis, while going the other way is simple and always valid.

The difficult part of writing an OpenACC to OpenMP translator is adding the right prescriptive OpenMP keywords to loop nests (to ensure the loops are effectively mapped to hardware). To make matters more interesting, these keywords may be different for each particular device, as the next sections describe.

2.8.5.1 Sultana et al.'s Translator. Sultana et al. (2016) made a prototype tool for automatically translating OpenACC to OpenMP, focusing on translating for the *same* target device, e.g., Nvidia GPUs. They demonstrated that some parts of the translation are indeed mechanical, but others require more work. One of their goals was to provide a deterministic translation, i.e., the same set of OpenACC directives will always be translated to the same set of OpenMP directives. To this end, they created a deterministic set of translation rules for each OpenACC data and compute directive, as well as deterministic rules for adding `gang`, `worker`, and `vector` clauses to OpenACC loops that did not already possess them. This was necessary because, while OpenACC allows the compiler to decide how to parallelize nested loops, OpenMP requires more direction. In addition, OpenMP has no equivalent for OpenACC's `seq` directive, so the translator needed to remove directives from these sequential loops, and redistribute any reductions or private clauses on these loops.

However, Sultana et al. ran into problems with changing compilers when going from OpenACC to OpenMP (specifically, OpenMP offloading), including large performance differences. Some of this is expected, as both OpenMP and OpenACC rely heavily on compiler implementation details. They used PGI as their baseline OpenACC compiler and Clang as their OpenMP compiler, and noticed that the kernels PGI generated were almost always faster than the kernels Clang generated. This could be because, at the time, PGI's OpenACC implementation was much more mature than Clang's OpenMP implementation, but it does show that compiler implementations of these models can have significant impact on overall performance.

Furthermore, as future work, Sultana et al. believe that they may need to add device specific translation rules, echoing Wolfe (2016a). For example, the rules for an Nvidia GPU will probably not be optimal for a Xeon Phi, and vice versa.

2.8.5.2 Clacc. A more recent project, Clacc (Denny et al. (2018)), desires to build a production-quality, open-source OpenACC compiler, built on Clang, and to generally improve OpenACC and GPU support in Clang. Clacc translates OpenACC into OpenMP to take advantage of existing OpenMP support in Clang. The authors of Clacc note that this both improves code portability between machines without good OpenACC or OpenMP implementations (as Sultana et al. (2016) also describe), but also opens up possibilities for using existing OpenMP tools to analyze OpenACC. As OpenACC is much newer than OpenMP, tool support for OpenACC is less mature, and this could significantly benefit OpenACC developers. Many developers are also worried that OpenACC will soon be subsumed into OpenMP, which is much more popular, and that porting

to OpenACC will therefore be a waste of effort, so being able to translate from OpenACC to OpenMP easily and automatically would ease their concerns.

To implement OpenACC support in Clang, the Clacc developers first translate OpenACC code to an OpenACC AST inside Clang, then create shadow OpenMP sub-trees, which can be compiled to an executable or used to output an OpenMP AST (or source code) equivalent to the OpenACC input.

While their implementation is very new and doesn't yet fully support GPUs or languages other than C, the Clacc team has been able to get performance comparable to PGI's on multi-core, with the exception of one benchmark, as long as `gang`, `worker`, and `vector` clauses are specified.

2.8.6 Generic Translators. While not originally designed for performance portability, the translators described here still have significant applications to performance portability. They can be used to optimize and maintain code, making it possible to automate applying different optimizations for different machines and keeping code updated. They could also theoretically be used to automate porting from one programming model to another, in the event that an application's current model isn't as performance portable as the developers would like.

The tools described in this section are closest to the work done on MARTINI in this dissertation. There are a wide variety of code rewriting tools that can do *some* of what MARTINI can do (e.g., other tools that can insert OpenMP pragmas include DawnCC (Mendonça et al. (2017)), PPCG (Verdoolaege et al. (2013)), and Par4All (Amini et al. (2012))). However, this section will only cover more generic rewriting tools that share MARTINI's varied applications.

2.8.6.1 *Regular Expressions: sed, awk, etc..*

Text editing tools that use regular expressions, like `sed` (*sed, a stream editor* (n.d.)), `awk` (*The GNU Awk User's Guide* (n.d.)), and others, are mainstays of automated editing due to their string processing power. At a high level, the user provides a specification for the "before" string(s) they are looking for and a specification for what the string(s) should look like "after;" the complexity of providing these specifications as regular expressions can be anywhere from quite simple to extremely difficult, though. Regular expressions are purely text-based, and as such these tools have no semantic understanding of what they're processing beyond a stream of characters. Despite this, they can still be useful for simple code changes, such as renaming a function across all callsites. However, for any change that requires semantic information, especially information that is syntactically distant from the location of the change, using regular expressions and other text-based tools becomes extremely difficult if not impossible.

The rest of the tools discussed in this section are compiler-based so as to make use of the semantic information available in compilers, which overcomes this difficulty of text-based tools.

2.8.6.2 *LLVM and Polly.*

The LLVM project (*The LLVM Compiler Infrastructure* (n.d.)) is a collection of versatile compiler and compiler toolchain technologies. The Clang compiler is the C/C++ frontend to LLVM and enables users to implement their own C/C++ manipulation tools either as operations on the Clang AST or the LLVM internal representation (IR). The Clang AST is more high level and closely tied to the source code, and the tools implemented on it are more separate from the compilation process. See Sec. 2.8.6.3 for more on tools using the Clang AST.

LLVM IR (Lattner and Adve (2004)) is more low level and has more powerful transformations already implemented over it that are closely integrated with the compilation process. These are known as "LLVM passes" since they serve the same purpose as optimization passes in other compilers and are implemented as C++ classes. A user can implement arbitrary code modifications in an LLVM pass, but at the cost of working with LLVM IR, which is akin to generalized assembly and not intended to be seen by non-compiler experts. LLVM's extensive collection of libraries eases the burden somewhat, but writing an LLVM pass is still not ideal for the average C++ developer.

Polly (Grosser, Groesslinger, and Lengauer (2012)) is another part of the LLVM project which uses the polyhedral model to implement a suite of LLVM passes for code transformation and optimization. The polyhedral model uses mathematical representations of loop bounds as definitions of polyhedra to perform transformations on the user's code. While powerful, these transformations are not terribly general, as they require the code to already be in a specific form and only apply to loop nests. There are many, many other tools, such as PPCG (Verdoolaege et al. (2013)), that also make use of the polyhedral model, but since they are not as generic as we desire, they will not be discussed.

2.8.6.3 ClangMR and Clang::Transformer.

ClangMR (Wright et al. (2013)) and the Clang Transformer library (Clang Developers (n.d.)) are two similar tools that perform transformations based on the Clang AST. Both use Clang's AST matchers, which are a functional-style library for finding structural matches in the AST, as a frontend. ClangMR uses callbacks in the AST matcher library to perform rewrites and transformations, while the Transformer library continues to use functional-style calls very similar to the AST matchers themselves

to define rewrite rules. As both use the Clang AST as a user interface, neither is terribly friendly to the average C++ developer, though both are very powerful and can perform arbitrary code transformations.

2.8.6.4 Cetus. Cetus (Dave et al. (2009); S.-I. Lee, Johnson, and Eigenmann (2003)) is a research compiler originally intended for automatically parallelizing C code, though it can also be used to perform arbitrary transformations. The Cetus IR is implemented as a Java class hierarchy, and users can manipulate a program by writing their own Java classes. While Cetus allows users to perform arbitrary transformations and has a full-featured library of operations and analyses on its IR, this interface is not very user-friendly as it requires both knowing Java in addition to C and working directly with the source code's Cetus IR.

2.8.6.5 Stratego/XT. Stratego/XT (Bravenboer, Kalleberg, Vermaas, and Visser (2008)) is a powerful suite of tools for program transformation. It includes a domain-specific language for describing transformations, a compiler for that language, libraries of transformations, a translator for turning Stratego programs into C, various analyses, and more. Users can define the language they wish to use in Stratego, then define transformations on that language and generate parsers, compilers, and more. Several projects in multiple languages, including an optimizer for C/C++, have been implemented in Stratego/XT. Stratego/XT perhaps has the most features of any tool described here, but its use is hindered by the fact that users must learn its DSL interface, which is based heavily in the theory of programming languages and thus not overly friendly for the average C++ developer.

More recently, Stratego/XT has been integrated into the Spoofax ecosystem (Kats and Visser (2010)), which is a set of tools for developing domain-specific languages, though it could (in theory, and with some work) also be used to perform program transformations on existing languages.

2.8.6.6 CHiLL. CHiLL (Chen, Chame, and Hall (2008)) is an optimizing compiler for Fortran and C that, unlike most optimizing compilers, performs code transformations based on user input or compiler decisions, iteratively measures performance of code variants generated from those transformations, and finally generates an optimized version of the code. CHiLL makes use of many of the same loop transformations as polyhedral optimization (see Sec. 2.8.6.2), but is not limited to those. The transformation scripts that can be either given to CHiLL or derived by it are essentially a DSL, and each portion of the script can be applied in any order, or composed with other parts of the script. However, it may not always be clear to the average developer which part of the script affects which loops in the code, though the DSL is fairly self-explanatory to those familiar with loop optimization.

The main goal of CHiLL is to create compiler-optimized code that has performance equivalent or better than hand-optimized or existing-compiler-optimized code. In several case studies, it has succeeded at this. MARTINI hopes to emulate this success but with a more friendly user interface.

2.8.6.7 Coccinelle. Coccinelle (Padioleau, Lawall, Hansen, and Muller (2008)) was originally created to propagate local patches and other API changes across the Linux kernel and drivers, but more recently has found use as a generic refactoring tool in scientific applications. While these use cases may at first seem quite different, both involve performing systematic transformations

over large codebases, which is, at a high level, what Coccinelle was designed for. Coccinelle uses a variation on the `patch` syntax to describe rewrite rules; the main modification Coccinelle makes is to include a set of "metavariables" at the start of a rule that can match actual variables in the source code. This syntax can be used to perform arbitrary code transformations, and is well-suited to performing rewrites en masse – Coccinelle can parse the entire Linux kernel in a matter of minutes.

In their case study (Martone and Lawall (2021)) on performing an Array-of-Struct (AoS) to Struct-of-Array (SoA) transform on a scientific application, the Coccinelle developers note that their use of patch syntax to store the transformations allows developers to continue working with the familiar but lower performance AoS code while running their application (post-transformation) with the higher performance SoA code. This separation of concerns leads to code that is both high performance and easy to maintain and extend. The Coccinelle rewrite rules that now exist for the AoS to SoA transformation could also, with some work, be applied to other applications. MARTINI seeks to emulate Coccinelle in these respects.

2.8.6.8 Orio. Orio (Hartono, Norris, and Sadayappan (2009)) is a performance tuning system that can perform a wide variety of source-to-source transformations controlled by annotations in an application's code. Orio will dynamically load the correct Python modules based on the names in the annotations, then perform an automated search among those modules for a sequence of code transformations that gives optimal or near-optimal performance.

These transformations are represented as Python modules that directly operate on the code's AST. Orio's developers have deliberately made it possible for users to implement their own Python modules, so it could theoretically be

used for arbitrary code rewriting as well. However, as it still requires a developer to go through their application and insert these annotations, it is not well suited for automating a large number of edits. It also requires the developer to know Python in addition to C/C++, which, while Python is generally regarded as an easy language to learn, is not ideal.

2.8.6.9 Nobrainer. The most similar work to that done in this dissertation is Nobrainer (V. Savchenko et al. (2019); V. V. Savchenko et al. (2020)), which also uses C/C++ code snippets and Clang's AST matchers to match user source code and describe how to modify it. Nobrainer allows developers to use specialized C/C++ syntax to write before-and-after code snippets describing the changes they would like to make to their code. The before snippets are used to generate Clang AST matchers, which can be used to search the code for structural matches to the before snippet. When a match is found, the actual names from that match are bound to parameters in the AST matcher, then copied into the corresponding places in the after snippet.

Savchenko et al. enforce several rules on these snippets (e.g., to match a single expression, that expression must be returned) so that they can ensure their transformations are (type-)safe. Nobrainer's design philosophy for these transformations is to make matchers as specific as possible and force users to add generality – they assume all names in a matcher are literals unless they are specified as parameters and do their best to enforce safety via rules surrounding types and semantics.

2.8.6.10 Selected Rewriting Tools for Other Languages. Though this work is necessarily focused on C and C++, as we wish to study performance portability and other effects of code rewriting technology on HPC

applications, there are of course code rewriting tools for other languages. A few of these are described here, since there are important lessons to be taken from them.

Haskell. Haskell is a functional language with a relatively rare property: rewrite rules are supported in its most common compiler, and can be written directly in the language (GHC Team (n.d.)). Since users do not have to learn another language or use a different compiler to utilize these rewrite rules, Haskell has perhaps the most friendly user interface out of all described here. MARTINI seeks to emulate this by having rewrite rules defined in C/C++, while providing even more powerful and varied transformations than Haskell allows. For example, it is very difficult if not impossible to change the type of an expression or insert arbitrary code via Haskell's rewrite rules.

Java: Sydit and Lase. Sydit (Meng, Kim, and McKinley (2011)) is a plugin for the Java IDE Eclipse which assists developers in performing *systematic edits*. Sydit works somewhat differently to many of the other tools described here: users provide an example edit that is specific to one class or method, then Sydit generalizes it into a transformation that can be applied anywhere. The user can then specify where else they would like this transformation applied.

To generate the transformation, Sydit runs a difference algorithm (similar to `diff`) on the original and edited ASTs, then finds any context (e.g., dependent statements, variables used) relevant to the changes, abstracts away specifics like class names, types, and line numbers, and finally generates an edit script that can check a target for suitability and, if it is found suitable, apply the transformation with specifics from the target inserted.

As Sydit is an IDE tool, it is not very well suited to bulk edits, since the user must select and check every instance where the transformation should be applied. While acceptable for an IDE-based tool, this is less ideal for situations such as inserting instrumentation around all function calls or changing the signature of a commonly used function, where bulk editing should be highly automated. MARTINI aims to give users the same level of control as Sydit, but with bulk edits enabled.

Lase (Meng, Kim, and McKinley (2013)) is very similar to Sydit (and in fact shares several authors) in that it infers edit scripts from examples, but Lase requires two examples so it can more accurately create an editing script. It then automatically searches for similar code in a program and suggests edits to the user. While more suited to bulk editing than Sydit, it does still run into the same problem of user approval on each edit.

2.8.6.11 Summary of Generic Rewriting Tools. As was discussed in Chapter I, rewriting tools generally have three goals: to minimize the complexity of their interface, maximize what users can express, and allow automating a wide variety of use cases. Figure 2 shows how the tools discussed in this section measure up. As can be seen, none of the tools mentioned meet all seven sub-goals, demonstrating a lack in current technology.

2.9 Summary

This section gives a brief summary of the content of this chapter.

2.9.1 "The Three Ps". Performance, portability, and productivity are often considered together when discussing performance portability, and are often known as the "Three Ps." This section will summarize what was discussed in this chapter on this subject.

	Friendly UI	Custom rewrites	Bulk edits	Original code preserved	Arbitrary rewrites	Rewritten code visible	UI not another language
sed/awk	×	✓	✓	✓	×	✓	×
LLVM pass	×	✓	✓	✓	✓	×	×
Polly and polyhedral model	×	×/varies	✓	✓	×	×/varies	×
Transformer library and ClangMR	×	✓	✓	✓	✓	✓	×
Omni	×	✓	✓	✓	✓	✓	×
ROSE	×	✓	✓	✓	✓	✓	×
Xevolver	×	✓	✓	✓	✓	✓	×
BONES	×	×/tricky	✓	✓	×	✓	✓
Cetus	×	✓	✓	✓	✓	✓	×
Stratego/XT	✓	✓	✓	✓	✓	✓	×
CHiLL	×	✓	✓	✓	✓	✓	×
Coccinelle	✓	✓	✓	✓	✓	✓	×
Orio	✓	✓/tricky	×/slow	✓	✓	✓	×
Nobrainer	✓	✓	✓	✓	×	✓	✓
LIFT	×	✓/tricky	✓	✓	×	✓	×
SYDIT	✓	✓	×/slow	×	✓	✓	✓
LASE	✓	✓	×/slow	×	✓	✓	✓
Haskell	✓	✓/tricky	✓	✓	×/tricky	×	✓

Figure 2. A comparison of several code transformation tools. Boxes with an extra note can be read as "generally yes/no, but it's tricky/slow/varies."

2.9.1.1 Portability. Most of the models described here are very portable, and many of the ones that aren't are actively working on adding support for more architectures. Portability is in many ways a prerequisite for performance portability, since a model can't be performance portable if it only runs on one or two architectures. Regardless, all claims of portability should be taken with a grain of salt. A recent, large-scale study on performance portability by Deakin et al. (2019) notes that one of the most difficult parts of comparing performance portability values for different programming models is simply getting

applications to run on a variety of architectures in the first place. Immature implementations or lack of testing mean that, while small benchmarks might run, larger applications may still fail; Deakin et al.'s analysis had to cope with multiple failures on some of their applications. In addition, for many models, portability varies by implementation, and to migrate to a new architecture, users may have to switch implementations. This can be problematic since the implementations' support for the standard may vary, and performance can also vary, but this is an implementation problem and not a problem with the model itself.

Perhaps the most important aspect of the models themselves to consider, with respect to portability, is how easily they can be extended to support new architectures – particularly architectures that aren't just a variation on traditional CPUs, GPUs, or clusters. We don't know what HPC machines will look like in the future, so it's worth considering how extendable a model is and whether it has the right abstractions for parallelism, computation, and communication to be "future-proof," regardless of what happens to the underlying hardware, including CPUs, GPUs, the memory hierarchy, configurable computing such as FPGAs, and a host of other special purpose processors.

Higher-level models that internally utilize other, lower-level models, such as high-level libraries (e.g., SkelCL, Kokkos) or high-level languages, are a good example of extensibility – if a new architecture with a new programming model comes out, they can implement a new back end and their users can automatically take advantage of it. Other models, like OpenMP and the C++ STL, have had to work much harder to adapt, but have ultimately succeeded. The users of these models have also had to work to update their code, however, so this is not a sustainable path, as it can greatly impact the model's productivity during the

changeover. If a programming model has to go through a major API change every time there is a shift in supercomputer architectures, it isn't truly portable.

2.9.1.2 Performance. Most of the models discussed here also give high performance, but this is much more inconsistent, since performance portability is still a struggle for many. (See Sec. 2.2 for a discussion of how optimizing for performance vs. for portability can be at odds.) Furthermore, optimizations that are good for one architecture might hurt performance on another, and this can be very difficult to reason about.

Because of this, it's important for programming models to give users lots of performance knobs to tune (if they so desire), but to keep those knobs *out of the application code* and preferably in one centralized location. Almost all the models described here do a good job of this, with the exceptions of OpenMP and OpenCL, to some degree. The primary two ways models abstract out performance tuning are by providing high-level interfaces with specialized back ends (e.g., OpenACC and Kokkos) or by allowing users to define their own specialized mappings to hardware (e.g., PPLs and custom directives). Both of these methods are functional, although the latter is more flexible at the cost of putting more responsibility on the user. Better support for parallelism in compilers could reduce this burden by making it easier for compilers to reason about parallelism, so users don't have to provide extra information about hardware mappings.

2.9.1.3 Productivity. Productivity is perhaps the most difficult of the Three Ps. There isn't yet a good definition or way to measure it (see Sec. 2.3), which means most discussion of productivity (even in this paper) is qualitative, based on individuals' opinions and experiences. However, there is still a general consensus that even the performance portable models with the steepest learning

curves are still better than device-specific models like CUDA and OpenCL in terms of productivity, if only because porting from one machine to another takes less work. Keeping these more portable models productive is good for performance portability – it forces them to stay higher-level and avoid falling into the "OpenCL trap" of becoming overly-specific, so they have a better chance of achieving consistent performance on multiple architectures.

Models that support incremental porting (directives and libraries) are in many ways the most productive for porting applications, since a small part of the application can be ported to test out a model without committing to rewriting thousands of lines of code, even though high-productivity languages like Chapel (Chamberlain et al. (2007)) might be more succinct. Tools that can automate the porting process (like Bones and the other translators) are also a boon for productivity.

Perhaps the best solution for productivity, though, is an application-specific performance portability layer (see Sec. 2.5.2), since it can allow parallelism experts to play with different models in the background while domain scientists get on with their work, and it removes all dependence on a specific performance portable model from the application code itself. While the concept is still relatively new, separation of concerns between application programming and application tuning seems to be a very productive path to take. (And is, in fact, the path taken in the rest of this dissertation.)

The tools that go with these models, such as debuggers, testing apparatuses, and performance measurement tools, are also essential for productivity, but they are outside the scope of this dissertation.

To conclude this part of the summary, a list of programming model features that are good for performance portability and productivity:

- High-level abstractions.
- Simple front end that targets multiple (non-performance portable) back ends.
- Keeping configuration separate from code.
- Multiple abstraction levels.
- Allow users to extend anything and everything.
- Automate, but let users override.
- Scale down as well as up.
- Standardization.
- Separate expressing parallelism from mapping it to hardware.

2.9.2 Performance Portable Models.

This section will summarize the performance portable models discussed in this chapter.

2.9.2.1 Libraries.

There are two main classes of libraries designed for performance portability: skeleton libraries and loop-based libraries. Skeleton libraries provide higher-order functions based on common parallel patterns users can customize to run computations in parallel, while loop-based libraries abstract away iteration spaces and data structures to run loops in parallel. Application-specific performance portability layers can further insulate users and application code from the details of parallelism and changes in the underlying performance portability models.

2.9.2.2 *Languages.*

While the barriers to entry for parallel languages are higher than those for other models, some languages have been successful. These languages are primarily task-based, but most include data parallelism as well, since modern hardware relies heavily on vectorization for its performance. Some of these languages are very high-level while others are less so (e.g., DPC++).

2.9.2.3 *Directives.*

The two most popular directive-based models are OpenMP and OpenACC, which fall on opposite ends of the prescriptive vs. descriptive spectrum, although both have been moving closer to the middle. Other directive-based models, like OpenMC and XcalableACC, haven't been nearly as successful, but have unique features OpenMP and OpenACC could learn from. There are also tools that let users define their own directives, to give them more control over what transformations and optimizations happen to their code.

2.9.2.4 *Translators.*

Many of the other models in this paper were built on source-to-source translators like Omni and ROSE, which were both designed to enhance user productivity by helping users transform their code into more performance portable versions. Other translators, like Bones and Clacc, were designed more as compilers than translators, but also provide translation capabilities so users can further tune their code before compiling it. However, most translators do not meet all the goals we have set for them. Filling this gap is the goal of this dissertation.

CHAPTER III

METHODOLOGY

This chapter contains material originally published by Johnson et al. (2022).

This chapter will discuss the methodology used in this dissertation, including motivation, design, and implementation details.

3.1 Motivation

The work in this dissertation was motivated by an existing, specialized code rewriting tool, PDT (Lindlan et al. (2000)). PDT can be used to insert calls to a performance measurement library, TAU (Shende and Malony (2006)), at various predefined places within the user's code, including function entry and exits, and around for loops in a given file or function. PDT allows for *selective* instrumentation of source code, as opposed to full instrumentation, which is what most other compiler-based tools provide.

Selective instrumentation is important for two reasons. First, it gives users more control over the data they collect, so they can only collect data of interest, instead of being overwhelmed by data on everything. Second, it can significantly reduce runtime overheads from instrumentation, which can, in some cases, make it infeasible to run a fully instrumented application. By giving users a manageable amount of data to work with and allowing them to actually run their instrumented code, selective instrumentation opens up opportunities in performance measurement for many codes.

However, PDT can only insert predefined calls at predefined locations, based on the syntax of a specification file. It cannot insert arbitrary code at arbitrary locations, and as such it is very limited in its use cases. A tool that *could* do these things, that could allow users to perform arbitrary rewrite operations on their code,

would be even more useful in even more situations. Unfortunately, existing code rewriting tools are not exactly user friendly, as discussed in Sec. 2.8.

What might happen if we *did* have a tool capable of all these things, *with* a user-friendly interface?

This dissertation aims to investigate the following questions:

- How can we most intuitively express a wide variety of source-to-source code transformations natively in C++?
- Can we perform those transformations using only information available at the AST level in an unmodified mainstream compiler?
- What effect will such transformations (e.g., optimizations, ports to different models, measurements, refactors, and more) have on the performance and performance portability potential of high-performance applications?

3.2 Design and Implementation of MARTINI

This section will discuss the particulars of how the tool described in the previous section was created.

3.2.1 Design Philosophy. We chose to implement MARTINI, the Little Match and Replace Tool, as a front end, AST-based tool as opposed to farther along the compilation toolchain because the AST is more closely tied to the source and contains more detailed source information than even LLVM IR. In LLVM IR, subtle differences in code structure are obfuscated or disappear completely; it can be difficult to tell the difference between, for example, a while loop and a for loop, or a switch statement and an extended if-else if-else chain. To provide the semantic matching capabilities we do, we need that extra information. We also wished to make use of Clang's existing and easily extensible AST matching

capabilities. There is little support for performing similar structural matching on LLVM IR.

It is important to us that our tool be source-to-source, since we want users to be able to maintain a simple codebase and add complexity via only-as-needed transformations. While this work deals primarily with performance optimizations and porting from one programming model to another, there are many other use cases for MARTINI, such as producing a one-off instrumented version of an application for performance tuning. In each of these cases, it is important for the original code to be maintained, so that the user can undo any changes they make. We want MARTINI to be applicable to as many use cases as possible, hence we want both the original and transformed code to be visible to the user.

For similar reasons, we chose to leave correctness and verification mostly in the hands of the user. To enable the widest variety of transformations, we decided not to enforce any ideas of correctness, since it's easy to imagine use cases where correctness is *not* guaranteed, such as introducing multiple precision calculations. We want MARTINI to be applicable to those use cases as well.

3.2.2 User Interface Design. The most intuitive starting point was simple "before and after" code snippets, similar to the regular expressions given to `sed`. From there, we've begun developing syntax based on these rules:

1. The syntax must be available in vanilla C++.
2. Clang must be able to parse all transformation specifications without modification.
   (a) Minimal modifications, e.g., giving Clang knowledge of new attributes, are allowed.

(b) Including our own additional headers in the specification file, e.g., for function-like macros, is allowed.

3. The syntax must be minimal so as not to obfuscate the transformation.

4. The syntax should be self-explanatory; every new attribute, macro, etc. should say what it is.

We've taken inspiration (both for what we do and don't want our syntax to be) where we can from existing tools. Nobrainer (ref. Sec. 2.8.6.9), the most similar tool to our own, has been our most useful point of comparison. Sadly, it is not open source, so we cannot make a direct comparison.

To specify code modifications, users provide two sets of C++ functions containing parameterized code snippets: *matchers* and *replacers*, also called *transformations*. Matchers describe which code to modify, and are distinct from Clang's *AST* matchers. We will always refer to the latter as AST matchers in this work to avoid ambiguity. Replacers describe how matched code should be rewritten. Figure 3 illustrates the workflow of our tool. Our contributions, which include the AST matcher generator, the replacer transform generator, and code rewriting functionality, are highlighted with bold, green outlines. We reused Clang's AST generation and AST matcher utilities, as well as some of their rewriting infrastructure. The user provides the program source (to be rewritten) as well as the matchers and replacers. The latter is shown as separate files but one file can contain any number of matchers and replacers.

To declare matchers and replacers, we introduce a few new C++ attributes (Figure 4) and DSL-like dummy functions (Figure 5) in our `clang_rewrite` namespace. Through the use of native C++ to embed control structures, we can

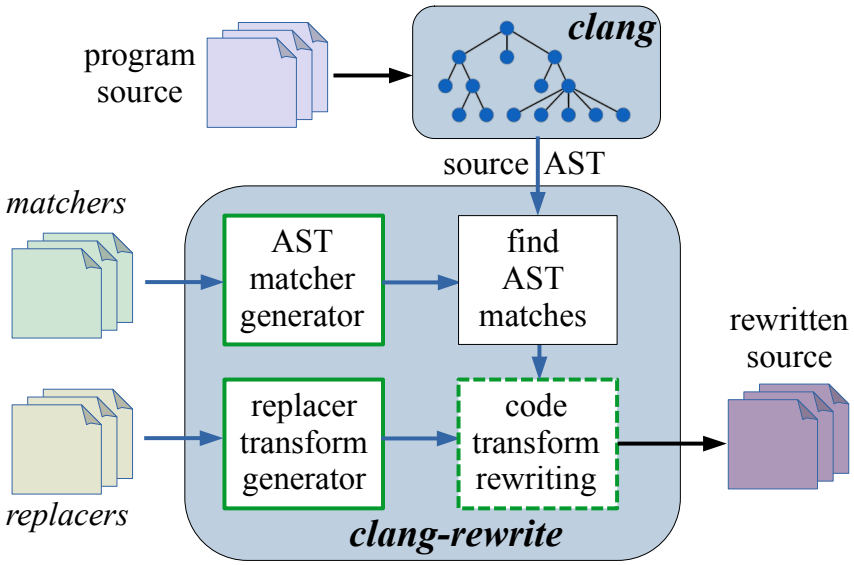


Figure 3. The workflow of our tool. Our contributions have bolded, green outlines. The dashed outline indicates that, while we reused some existing infrastructure, we also made significant contributions.

work with an otherwise unmodified Clang. Similarly, users can verify the validity of their matchers and replacers through existing verification tools in in the Clang front end (e.g., `clang -cc1 -verify`). This is important, as we expect all user-given code to be valid in its respective language, since a Clang AST can only be generated for valid inputs.

3.2.3 MARTINI Design. We want any technology we develop to be "minimally invasive" in the compiler toolchain we use to implement it – in other words, we don't want to have to modify the compiler itself to get the information we need to develop our tool. We also want to preserve the user's original application code, and for the transformed code to be compiler independent. To that end, we've chosen to create a source-to-source transformation tool built on Clang and the Clang AST. Clang's frontend tooling infrastructure is the most mature and has the most features out of all the mainstream compilers, making

`[[clang::matcher("<matcher_name>")]]`
for matchers; identifies a function as a matcher specification with the given name.

`[[clang::replace("<matcher list>")]]`
for replacers; identifies a function as a replacer associated with all listed matchers.

`[[clang::insert_before("<matcher list>")]]`
for transformations; identifies a function as a replacer that inserts code before a match.

`[[clang::insert_after("<matcher list>")]]`
for transformations; identifies a function as a replacer that inserts code after a match.

`[[clang::rewrite_setup]]`
for matchers, replacers, and transformations; identifies setup statement(s) (i.e., variable declarations) that are required for a `matcher_block` to parse.

`[[clang::matcher_block]]`
for matchers, replacers, and transformations; identifies statement(s) to be matched, replaced, or inserted. Outside code can provide declarations to ensure valid C++. If not present, all statements are used, though support for this is incomplete.

Figure 4. The C++ attributes used to declare matchers and replacers in user-provided code snippets. Through use of native C++, these control attributes are naturally embedded in the source and can be handled by an otherwise unmodified Clang.

it a very common choice for tool development. Its AST has sufficient ties back to the source for our needs, while still providing a wealth of information about code structure. So far, the only modifications to Clang we've needed to make are implementing additional AST matchers (which needn't actually be in Clang, but are easiest to implement there, hence our choice to do so) and adding knowledge of the new attributes from our syntax to Clang's parser so they will appear in the AST. We've judged this as acceptable, since the modifications are small and don't change any of Clang's functionality otherwise.

`loop_body()`
in a matcher, will match the entire loop body; in a replacer, will copy in the loop body with no modifications.

`loop_body(vector<pair<T,T>>)`
overload of the above, where `T` is a type parameter; in a replacer, will copy in the loop body with simple modifications expressed as a pair of statements, e.g., replacing all uses of a variable `i` with `j`.

`loop_body(T predicate)`
another overload, where `T` is a type parameter; in a matcher, will check whether the predicate given is true.

`contains(code_structure s)`
a predicate for use in the above that will ensure the given code structure is in the match.

`not_contains(code_structure s)`
another predicate for use in the above that will ensure the given code structure is *not* in the match (implementation incomplete).

Figure 5. Signatures of the functions used as additional control structures inside matchers and replacers. With familiar C++ syntax, they allow users to express more types of transformations in a DSL-like way.

3.2.4 MARTINI Implementation. MARTINI uses the previously mentioned syntax, "compiles" it into Clang AST matchers, and uses the AST information gathered by those matchers to perform the described transformations. MARTINI makes heavy use of Clang's frontend tooling infrastructure, particularly the AST matchers and Rewriter library. Development has been focused on incrementally adding support for the full C++ standard via performing case studies.

Our AST matcher generator, shown in Fig. 3, "compiles" the statements in the matcher block (ref. `[[clang::matcher_block]]` in Fig. 4) of functions annotated as matchers (ref. `[[clang::matcher("...")]]` in Fig. 4) into a Clang

AST matcher. These AST matchers will match source code that has the same semantic structure as the input code snippet, regardless of syntactic differences. Consequently, potential hazards like arbitrarily complex sub-expressions, line breaks and spacing, and inline comments, are automatically dealt with. Matchers are by default parametric, but can also look for literal names (through the `hasName()` AST matcher). Parameters are bound (via the `bind()` AST matcher) to the source code they match.

To turn a code snippet into an AST matcher, the snippet's AST is traversed and converted node-by-node based on the node's semantics. For example, the AST matchers generated for the operands of a `CallExpr` node (function call) are connected via a `hasArgument()` AST matcher to the `CallExpr`'s AST matcher. We maintain an internal tree data structure with AST matchers as nodes to generate the final AST matcher.

Replacers are read by the replacer transform generator also shown in Fig. 3. Only the source code of the matcher block in a replacer (ref. `[[clang::replace("...")]]` in Fig. 4) is kept.

Matchers and replacers are tied together through names in their respective attributes written as string literals (ref. `"<matcher_name>"` and `"<matcher list>"` in Fig. 4). Replacers and transformations can be tied to any number of matchers, and a single matcher can be tied to multiple replacers, e.g., both `insert_before` and `insert_after`. A match found by a matcher will be rewritten with the code of the replacer with the matcher's name in its list, with appropriate names and values in the code replaced with those from the match.

MARTINI takes in a specification file describing matchers and replacers, performs AST matcher generation, parses the replacers, then uses the existing

AST matcher framework to search the user's source code for matches. Matches are processed in the order they are found by the Clang infrastructure, which uses top-down AST traversal. When it finds a match, it uses Clang's existing Rewriter utilities to replace the match with the code from the replacer, and replaces any identifiers in the replacement code with the code bound to that identifier by the matcher. An entirely new source file is produced to simplify experimentation with different transformation specifications and preserve the original source in case of mishaps. We put very few restrictions on the kinds of transformations users can write, even those that may produce invalid C++ output, to give users as much flexibility as possible. We have implemented a few safety checks for `insert_before` and `insert_after` (for example, we will not insert code into the conditions of an if statement), but in general users must verify the correctness of their rewritten code.

As mentioned, all identifiers, such as variables and function names, are treated as matcher parameters unless they are marked as *literals* by the user. This means users can choose to, e.g., match all functions that take two arguments and use the name of the matched function as a parameter in the replacer. Alternatively, a user can choose to match all calls to a specific function named `foo` that takes two arguments by making `foo` a literal. An identifier is marked as literal through a declaration in a special namespace (i.e., `namespace clang_rewrite_literals { void foo(int a, int b); }` – note `a` and `b` are *not* literals as they are parameters, not explicitly declared in the namespace), or by putting it in a special vector (i.e., `vector<string> clang_rewrite_literal_names {"foo"};`). We are considering adding another attribute for declaring literals for users that do not want to use the namespace or vector. All non-literal variable names will be bound

to the name in the source code that matched, and that name will be used in the rewritten code wherever that variable appears in the replacer.

3.3 Evaluation Procedure

We evaluate MARTINI in several ways in the rest of this dissertation. In earlier chapters, we evaluate MARTINI and the code it produces compared to traditional compiler tools, both in terms of the code required to perform a transformation and in terms of the correctness of the transformation. In later chapters, especially where there is no clear tool to compare MARTINI to, we evaluate the complexity of the code required to perform transformations and the performance of the resulting code compared to hand-written code. While a true productivity and usability study is out of the scope of this dissertation, we do attempt to estimate when MARTINI would provide a productivity gain.

CHAPTER IV

A BASIC REWRITING TASK

This chapter contains material originally published by Johnson et al. (2022).

This chapter serves as an introduction to the rest of this dissertation by describing a seemingly simple rewriting task and how it can be done with both traditional tools and MARTINI.

4.1 An Example: modernize-use-nullptr

```
int* test() {
 int* a = 0;
 double b, *c;
 b = 0;
 c = 0;
 return 0;
}
```

(a) Example snippet in which the `0`-literal is used for pointer and non-pointer values.

```
int* test() {
 int* a = nullptr;
 double b, *c;
 b = 0;
 c = nullptr;
 return nullptr;
}
```

(b) The same snippet with the `0`-literal replaced by `nullptr` in all pointer contexts.

Figure 6. Example to showcase the "modernize-use-nullptr" `clang-tidy` rewrite rule, which replaces `0`-literal pointers with `nullptr`. While the initialization of `a` can be reasonably found with text-based search-and-replace techniques, the other two replacements require non-local, semantic reasoning.

Consider the "simple" rewriting task done by the `clang-tidy` rule "modernize-use-nullptr"[1]. This rule replaces constants, like `NULL` and `0`, assigned to pointer variables with the C++11 `nullptr` keyword, which is both safer and more readable. Figure 6 illustrates the changes `clang-tidy` can perform. The first replacement, where `a` is initialized to `0`, could be done with a text-based tool, like `sed`, although a generic regular expression to match arbitrary types and variable names could get very complex, like this `sed` expression: `sed -i`

[1] `https://clang.llvm.org/extra/clang-tidy/checks/modernize-use-nullptr.html`

`'s/\([[:print:]]\+\)\*\([[:print:]]\+\)[[:space:]]*=[[:space:]]*0[[:space:]]*;/\1 * \2 = nullptr;/g'` (where `[[:print:]]` is all alphanumeric characters, punctuation, and single-space, and `[[:space:]]` is all space characters). However, the other two replacements, in the assignment to `c` and the `return` statement, are difficult if not impossible for a purely text-based tool to handle, since it does not have semantic context. The physical distance between the type of the variable and the `0`-literal can cross file boundaries, and most languages allow for various other complexities, like shadowing declarations with different types. This semantic context is out of reach for purely text-based search-and-replace tools – we need a tool that can understand more of the complexities of semantics.

Once a rewriting task reaches a complexity beyond the capabilities of text-based tools, programmers are often left with no choice but to develop specialized modules in a compiler, where the semantic information needed is most readily available. The "modernize-use-nullptr" `clang-tidy` rule shown above is one of many such modules implemented using the Clang compiler's tooling infrastructure, which provides access to semantic information from the Clang AST. The code for this rule, though, is roughly 125 lines of complex C++ and Clang AST matchers (excluding comments and code to handle `NULL` macros) which require Clang AST specific knowledge. Other rewriting software (ref. Section 2.8) expects similar specialized knowledge, as it also operates directly on the AST.

For sophisticated code transformations, using a compiler front end is often the only solution. However, we believe that this does not preclude a user-friendly approach, since developers can often write *what* they want to happen, though maybe not *how* it should happen. To this end, we developed a system built on

Clang and based on *semantic matching* and user-provided *code replacements* that is accessible to the average programmer. Similar to regular expressions, users can describe and customize code transformations naturally as "before-and-after" snippets of C++ code, which correspond to the two expressions used in search-and-replace schemes. The available context for searching and replacing is not restricted to syntax, though; it also contains semantic information extracted by a compiler. Our interface is designed to be intuitive for C++ developers by restricting its syntax to modern C++ and requiring no knowledge of compiler internals, unlike previous rewriting tools. It is also designed to give users a great deal of control over which changes are applied and where.

Continuing the above example, we can mimic most of the functionality of `clang-tidy`'s "modernize-use-nullptr" rule (except `NULL` macros – these require further checks due to how the Clang preprocessor works) through the three code pairs shown in Fig. 7. On the left hand side are the "matchers" which describe what should be replaced, and on the right hand side are corresponding "replacers" which contain the desired code with references back to the matched input. This example demonstrates how our approach provides semantic context for code rewriting and allows the average programmer to automate more complex rewriting tasks – and does so much more simply than Clang.

```
template <typename T>
[[clang::matcher("nptr-decl")]]
auto null_match() {
  [[clang::matcher_block]]
  T* var = 0;
}
```

(a) Matcher for initializing a pointer-typed variable to a 0-literal.

```
template <typename T>
[[clang::replace("nptr-decl")]]
auto null_replace() {
  [[clang::matcher_block]]
  T* var = nullptr;
}
```

(b) Replacement using `nullptr` for the matcher in (a).

```
template <typename T>
[[clang::matcher("nptr-asgn")]]
auto null2_match() {
  T* var = nullptr;
  [[clang::matcher_block]]
  var = 0;
}
```

(c) Matcher for a 0-literal assignment to a pointer-typed variable.

```
template <typename T>
[[clang::replace("nptr-asgn")]]
auto null2_replace() {
  T* var = nullptr;
  [[clang::matcher_block]]
  var = nullptr;
}
```

(d) Replacement using `nullptr` for the matcher in (b).

```
template <typename T>
[[clang::matcher("nptr-ret")]]
T* null3_match() {
  [[clang::matcher_block]]
  return 0;
}
```

(e) Matcher for a pointer-typed return statement using a 0-literal.

```
template <typename T>
[[clang::replace("nptr-ret")]]
T* null3_replace() {
  [[clang::matcher_block]]
  return nullptr;
}
```

(f) Replacement using a `nullptr` for the matcher in (e).

Figure 7. The three matcher-replacer pairs we used to mimic (most of) the functionality of clang-tidy's "modernize-use-nullptr" rule. Applied to Fig. 6a, the "modernized" version in Fig. 6b is produced. The variable name `var` is a parameter of the matcher block, and the original variable name in the matched program fragment (e.g., `a`, `b`, and `c` in Fig. 6) is bound to it for use in the replacement. While our matchers are by default type-agnostic, and hence fully polymorphic, we enable type-based reasoning for template type parameters, here `T`. As a result, the matchers on the left are restricted to pointer-typed values.

CHAPTER V

REWRITING FOR OPTIMIZATION

This chapter contains work that will be published by Johnson et al. (n.d.). Alister Johnson did all development on MARTINI and related writing, while Camille Coti designed and performed the performance experiments, including creating the figures, and helped with the related writing.

This chapter will discuss how MARTINI can be used to optimize applications using the lens of loop optimizations.

5.1 Introduction

Loop optimization is an important subset of performance optimization, as most (if not all) large scientific applications spend most of their time in loops. However, optimizing for performance portability is often very different from optimizing for performance, since optimizations that work well on one machine may not on another. Developers often wish to perform different loop optimizations per machine, or parameterize their optimizations.

Most loop optimizations are possible, but very difficult and error-prone, for humans to write, as they involve complex updates to loop indices in the loop body, so ideally they would be left to the compiler. All mainstream compilers implement a wide variety of loop optimizations, from unrolling and vectorizing to loop fusion or fission. The compiler uses heuristics developed by expert engineers to choose which optimizations to apply and the value of any parameters for that particular optimization (e.g., the number of times to unroll). However, not all compilers implement the same optimizations, and furthermore, the heuristics they use can vary wildly, leading to performance variations across both compilers and machines. Especially for application developers striving for performance portability

(Deakin, Poenaru, Lin, and Mcintosh-Smith (2020); Hollman et al. (2019); Hornung and Keasler (2013)), this is undesirable, so developers often end up implementing these optimizations by hand anyway. But an optimization that works well on one machine may be detrimental on another, so how then does a development team maintain their codebase?

Many codes do not maintain a single codebase, but rather versions specific to each machine they wish to run on. This can severely impact developer productivity though, as any bug fixes and new features must be applied to all versions, and versions can diverge over time, leading to a serious accumulation of technical debt if the developers ever decide to unify their codebases. Other development teams turn to (performance) portability libraries and language extensions like Kokkos (Edwards et al. (2014)) or OpenMP (OpenMP Architecture Review Board (2018)). However, these libraries can still have serious performance variation across different platforms, if they even support the platform at all (Deakin et al. (2019, 2020)), or in the case of OpenMP, still require different code per platform. This has improved drastically in recent years as libraries mature and performance portability features are added (e.g., OpenMP 5's new `metadirective` construct (Pennycook, Sewall, and Hammond (2018))), but is still a concern for high-performance application developers, who wish to achieve day-one high performance on the newest machines, often before libraries and compilers fully support optimizations on those machines.

Of particular concern is maintaining developer productivity in this environment, especially for domain scientists who may not be expert coders. Many loop optimizations, particularly ones that introduce extra loops or change indices, like tiling, fission, or vector intrinsics, can obfuscate the true purpose of a piece of

code for anyone not extremely well-versed in performance optimization, such as a domain scientist. Most domain scientists (and, to be fair, computer scientists too) would prefer to write and work with the "textbook" version of their algorithms – the simplest version, with no obfuscating optimizations, as might be seen in a textbook on the subject.

Our work aims to enable this by separation of concerns: the functionality of an application can be kept separate from its optimization via automatic code transformation with MARTINI. Instead of keeping optimizations in the main codebase, they can be kept separate as matchers and replacers that can be used to generate an optimized version of the base application at will. These matchers and replacers can be applied per-platform and are compiler independent, so developers can optimize their performance on any given machine without depending on a compiler or library to implement the particular set of optimizations they need, allowing them to achieve performance portability while maintaining a simple, "textbook" codebase, along with its rewrite rules.

The rest of this chapter is organized as follows. Sections 5.2, 5.3, and 5.4 will discuss three different kinds loop optimizations and how they can be done with MARTINI. Our goal is to demonstrate the variety of applications for our tool and describe its features. These examples were chosen because they are well-known, but difficult for compilers to consistently apply, either due to safety concerns or challenging implementation. Section 5.5 will evaluate how these optimizations impact a selection of benchmarks.

5.2 Loop Peeling

```
[[clang::matcher("peel_inner")]]
auto peelinm() {
  [[clang::rewrite_setup]]
    int max_i, min_i, max_j, min_j;
```

```
  [[clang::matcher_block]] {
    for (int j = min_j; j < max_j; j++) {
      for (int i = min_i; i < max_i; i++) {
        clang_rewrite::loop_body();
      }
    }
  }
}

[[clang::replace("peel_inner")]]
auto peelinr() {
  [[clang::rewrite_setup]]
    int max_i, min_i, max_j, min_j, x, y;
  [[clang::matcher_block]] {
    for (int j = min_j; j < max_j; j++) {
      for (int i = min_i; i < max_i - 1; i++) {
        clang_rewrite::loop_body({{x%y, x}});
      }
      for (int i = max_i - 1; i < max_i; i++) {
        clang_rewrite::loop_body({{x%y, x-y}});
      }
    }
  }
}
```

Listing 5.1 Loop peeling matcher and replacer, inner loop.

Loop peeling, or splitting, moves some iterations outside of the loop to simplify the treatment inside the loop. For instance, a test on the iteration number can be removed by taking the iterations that require a specific treatment outside of the loop.

For example, computing a wavelet transform uses N consecutive elements of an array, and moves in this array by steps of two elements. Hence, when $N > 2$, the last elements require *wrapping* on the elements of this array. Indices can be computed using a modulo; however, Coti, Falcou, and Matei (2020) show how computing this modulo at every step harms performance. To avoid this, we can peel the loop iterations to remove the modulo entirely. For the iterations that do

not require wrapping, we can simply use the indices as given, without the modulo. For iterations that do require wrapping, we can instead subtract the array size from the indices that would go past the end of the array.

Listing 5.1 shows how this can be done, and showcases a feature of MARTINI: inline matchers and replacers. The `loop_body()` function is a generic matcher that will match any set of statements inside the loop. The expressions given as arguments to the function (inside the double {}) are a nested, inline matcher and replacer set for simple replacements inside the loop body, such as renaming indices or, in our case, removing a modulo.

5.3 Loop Fission

Loops can be unrolled automatically and vectorized. However, some operations prevent this vectorization, for instance, if a test is performed in the loop. The excerpt shown in Listing 5.2 performs an operation on every element of an array and searches for the maximum resulting value. The two tests in the loop body prevent the compiler from vectorizing the loop iterations. The first test defines a specific instruction for the first iteration. It can be removed by taking this first iteration outside of the loop, i.e., peeling it. Then the loop body consists of two steps: computing the new value of `A[i]`, and comparing it to the current maximum. This can be distributed into two loops: 1) computing the new value and 2) searching for the maximum (Listing 5.3). As a result of splitting this loop, we expect the compiler to be able to optimize at least the first loop.

In order to transform this code, we need two pairs of matchers and replacers: one to peel the loop, and one to distribute it. We have already presented how peeling can be achieved in Section 5.2; the matcher and replacer look quite similar. Listing 5.6 presents the matcher and replacer for loop fission on this example.

```
for( auto i = 0 ; i < M ; i++ ){
    A[i] = oper( A[i], M );
    if( 0 == i ){
        maxsq = A[0];
    } else {
        if( maxsq < A[i] ){
            maxsq = A[i];
        }
    }
}
```

Listing 5.2 Original loop to be fissioned.

```
for( auto i = 0 ; i < M ; i++ ){
    A[i] = oper( A[i], M );
}
maxsq = A[0];
for( auto i = 0+1 ; i < M ; i++ ){
    if( maxsq < A[i] ){
        maxsq = A[i];
    }
}
```

Listing 5.3 Loop after fission.

```
for (int i = 0; i < M; i++) {
        r = calculate(A[i]);

        if (r > max) {
            max = r;
            index = i;
        }

        sum += r;
    }
```

Listing 5.4 Another loop to be split.

```
double* tmp = (double*)malloc(sizeof(double) * M );
for (int i = 0; i < M; i++) {
  r = calculate(A[i]);
  tmp[i] = r;
  sum += r;
}
for (int i = 0; i < M; i++) {
  if (tmp[i] > max) {
```

```
    index = i;
    max = r;
  }
}
free(tmp);
```

Listing 5.5 Second loop after fission.

```
[[clang::matcher("fission")]]
auto modm() {
    [[clang::rewrite_setup]]
    int max, min;
    [[clang::rewrite_setup]]
    double mm;
    [[clang::rewrite_setup]]
    double* tab;
    [[clang::matcher_block]] {
        for (auto w = min; w < max; w++) {
            tab[w] = oper( tab[w], max );
            if( min == w ){
                mm = tab[w];
            } else {
                if( mm < tab[w] ){
                    mm = tab[w];
                }
            }
        }
    }
}

[[clang::replace("fission")]]
auto modr() {
    [[clang::rewrite_setup]]
    int max, min;
    [[clang::rewrite_setup]]
    double mm;
    [[clang::rewrite_setup]]
    double* tab;
    [[clang::matcher_block]] {
        for( auto w = min ; w < max ; w++ ){
            tab[w] = oper( tab[w], max );
        }
        mm = tab[min];
        for( auto w = min+1 ; w < max ; w++ ){
                if( mm < tab[w] ){
                    mm = tab[w];
```

```
            }
        }
    }
}
```

Listing 5.6 Fission: matcher and replacer for first example.

We execute these two transformations sequentially, with two invocations of `clang-rewrite`, as opposed to writing both as a single transformation or assuming the transformations will compose correctly if run together with a single invocation of `clang-rewrite`. We do plan on adding the capability to compose transformations in the future, and it can be emulated now by multiple runs of our tool, or by including both transformations in a single matcher/replacer pair.

A more complex example is given by Listing 5.4: in some cases, the loop keeps two integers, the index and a maximum. However, the two-loop implementation (Listing 5.5) has the extra cost of another memory allocation.

5.4 Loop Tiling

Loop tiling is a common optimization to improve code performance by improving memory locality, particularly in the processor cache. Most caches load data in *blocks*, so when one entry in a multi-dimensional array is loaded, so are entries in the rows and columns around it. Tiling takes advantage of this by performing loop calculations in the order cache blocks are loaded, instead of in the order of array indices. While this can give significant performance improvements due to fewer cache misses, tiling requires major changes to loop structure and indices, which are difficult for humans to perform and can obfuscate the loop's purpose.

Furthermore, the size of the tiles is machine-dependent, as different processors have different cache layouts. This can lead to users needing multiple,

per-machine variants of a loop or having to write a parameterized loop with different settings for each machine. Our tool can assist with this by allowing users to keep a single base code version, with the processes for generating variants stored as matchers and replacers that can be applied based on which machine the code is being run on.

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

Listing 5.7 Original matrix transposition code.

```
for (int ii = 0; ii < n; ii+=TILE_SIZE) {
  for (int jj = 0; jj < n; jj+=TILE_SIZE) {
    for (int i = ii; i < MIN(n, ii + TILE_SIZE); i++) {
      for (int j = jj; j < MIN(n, jj + TILE_SIZE); j++) {
        B[i][j] = A[j][i];
      }
    }
  }
}
```

Listing 5.8 Tiled matrix transposition code.

A simple example where loop tiling can be taken advantage of is matrix transposition: the matrix is transposed block by block in order to exploit data locality as much as possible. From the simple code given in Listing 5.7, the loops can be executed by blocks (the tiles) of size `TILE_SIZE`, with a tile size chosen small enough to fit in the machine's cache. `TILE_SIZE` can be a constant chosen by the user at replacement time, or a parameter set per-machine, as with a preprocessor macro. Listing 5.8 gives the expected tiled code, with the inner loops transposing the matrix by tiles.

The corresponding matcher and replacer, given by Listing 5.9, are straightforward: we replace the original loops with outer loops that progress

by bigger steps, and inner loops that work inside of the tiles. The loop body is unchanged.

```
[[clang::matcher("tiling")]]
auto tilingm() {
    [[clang::rewrite_setup]]
    int max_i, min_i, max_j, min_j;
    [[clang::rewrite_setup]]
    double** A, **B;
    [[clang::matcher_block]] {
        for (int i = min_i; i < max_i; i++) {
            for (int j = min_j; j < max_j; j++) {
                    clang_rewrite::loop_body();
            }
        }
    }
}
[[clang::replace("tiling")]]
auto tilingr() {
    [[clang::rewrite_setup]]
    int max_i, min_i, max_j, min_j;
    [[clang::rewrite_setup]]
    double** A, **B;
    [[clang::matcher_block]] {
        for (int ii = min_i; ii < max_i; ii+=TILE_SIZE) {
            for (int jj = min_j; jj < max_j; jj+=TILE_SIZE) {
                for (int i = ii; i < MIN(max_i, ii +
   TILE_SIZE); i++) {
                    for (int j = jj; j < MIN(max_j, jj +
   TILE_SIZE); j++) {
                        clang_rewrite::loop_body();
                    }
                }
            }
        }
    }
}
```

Listing 5.9 Loop tiling matcher and replacer.

5.5 Evaluation

In this section, we examine the performance of the code generated by MARTINI to evaluate how efficient this approach is on straightforward

computation kernels. We compiled both codes (original and transformed) with gcc's `-O3` optimization level: our goal is to provide optimizations that cannot be performed by the compiler, and let it optimize what it can optimize. Ideally, our optimizations will allow the compiler to make better inferences about the user's code, leading to higher performance overall. Indeed, in almost all cases, our automatically-applied optimizations allowed the compiler to generate better performing code, even compared to `-O3`.

We ran all our experiments on a machine featuring two 16-core, 32-thread Intel(R) Xeon(R) CPU E5-2697 v4 CPUs running at 2.30GHz and 128 GB of memory. All the code was compiled using gcc 12.2.0 on a Linux RHEL running a kernel v. 4.18. We measured time using either `clock_gettime`, or reading the hardware timestamp counter. All the measurements were taken 10 times, and we present the mean and standard deviation on the plots shown in this section.

5.5.1 Loop peeling. We implemented a straightforward wavelet transform function of length 4. The basic implementation uses a modulo, as explained in Section 5.2. We used the matchers and replacers presented in the aforementioned section to remove these modulos by peeling both the inner and the outer loops. Computing these modulos takes a high proportion of CPU cycles in this computation kernel; hence, removing them thanks to loop peeling saves a high proportion of the computation time, as shown in Fig. 8.

5.5.2 Loop fission. We compared the performance of the maximum search code presented in Section 5.3 with the generated code that uses loop fission. In our example, the operation performed in the loop is simple: it computes `ceill( a*a )`. The performance is shown Figure 9. As expected, the code using two separate loops takes advantage of the vector capabilities of the CPU, and runs

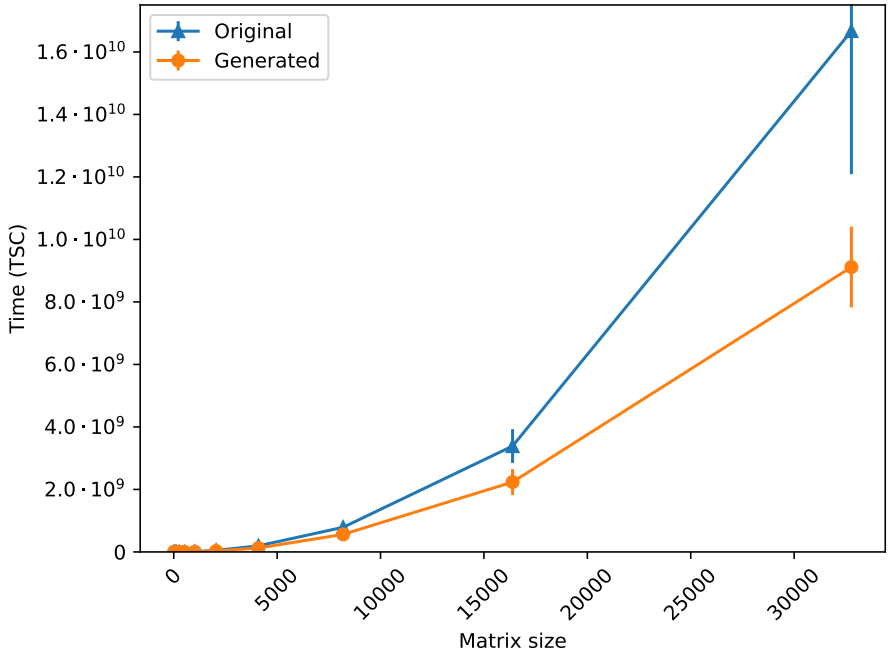


Figure 8. Loop peeling: performance obtained by the original code and the generated code. The vertical axis is clock cycles/nanoseconds.

faster... up to a certain array size, after which the fused-loop implementation is faster. The two-loop version uses twice as much memory in cache and therefore the array no longer fits in the cache, resulting in a higher number of cache faults that make the two-loop implementation slower.

A more computation-intensive loop, which calculates `cos(sqrt(fabs(in)))`, is evaluated in Figure 10. Since the computational portion of the loop is more expensive than with the simple maximum search use-case, this computation takes more advantage of the vector computation capabilities of the CPU, and the performance gain is higher, even with larger data arrays.

5.5.3 Loop tiling. Figure 11 presents an evaluation of the performance of the matrix transposition code presented Section 5.4. We compared the result of the original code (with two simple nested loops) with the generated

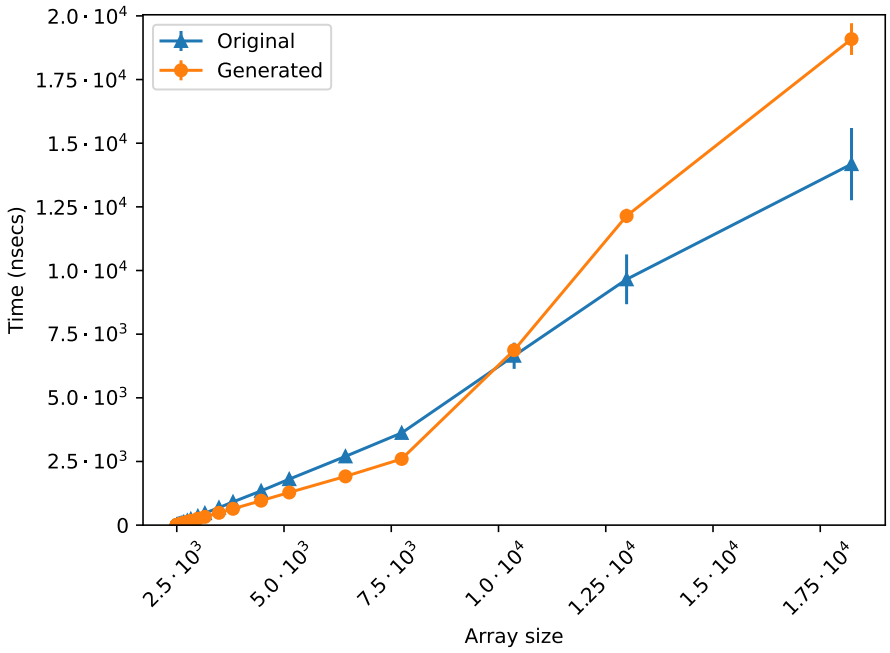


Figure 9. Loop fission: performance obtained by the original code and the generated code.

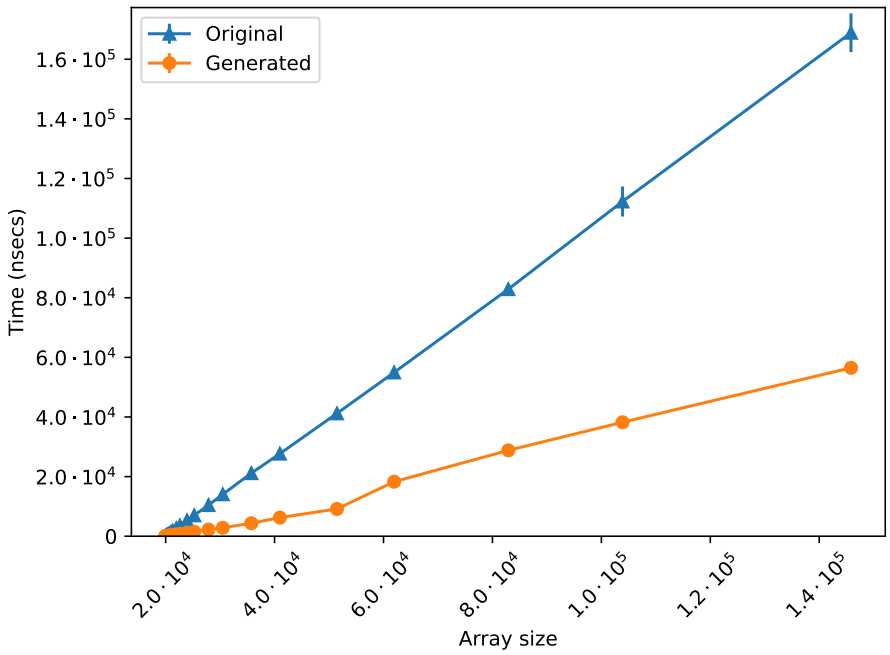


Figure 10. Loop fission (compute heavy): performance obtained by the original code and the generated code..

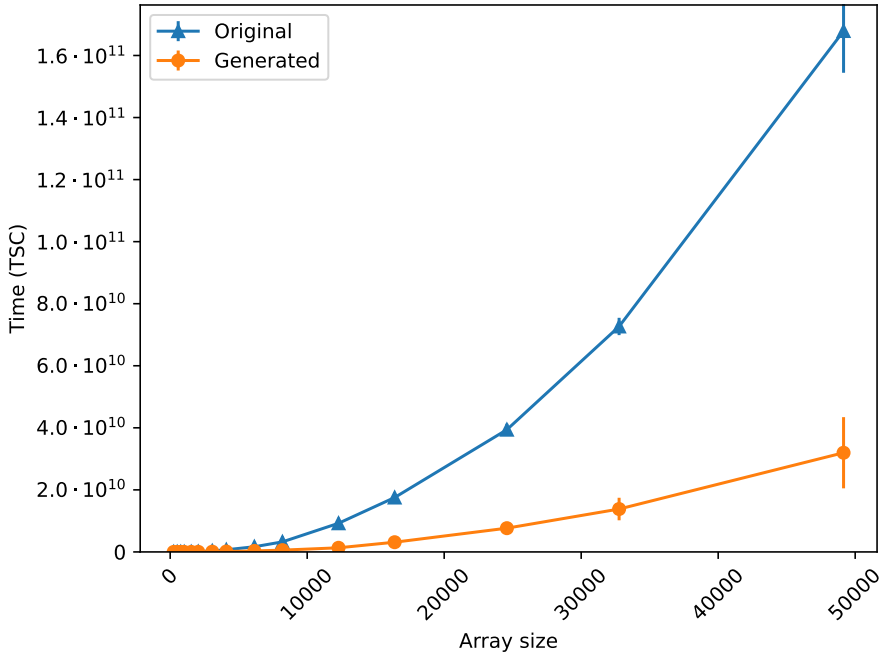


Figure 11. Loop tiling: performance obtained by the original code and the generated code. The vertical axis is clock cycles/nanoseconds.

code (working on tiles), and, as we can see, the generated code outperforms the original one, once again demonstrating that MARTINI can be useful as an optimization tool.

5.5.4 Aside: Autotuners. While autotuning frameworks, such as Orio (Hartono et al. (2009)), can perform many of the same tasks and optimizations MARTINI can, they still require hand-made edits to the users' code to either insert annotations on loops to tune or to add parameters to loops, such as tile size for tiled code or an unroll factor for loop unrolling. This is less than ideal, for reasons already mentioned: humans are bad at modifying loops, and an excess of annotations can obfuscate the purpose of the code. With MARTINI, all this complexity is hidden in matchers and replacers, and furthermore, users are not limited by the transformations implemented by the autotuner. MARTINI may

in fact be a good companion to existing autotuners, as it can be used to insert annotations or parameters automatically.

CHAPTER VI

PORTING TO NEW PROGRAMMING MODELS

This chapter contains work published by Johnson et al. (2022). It also contains work that will be published by Johnson et al. (n.d.). Alister Johnson did all development on MARTINI and related writing, while Camille Coti designed and performed all performance experiments and helped with related writing.

This chapter will describe how MARTINI can be used to port existing programs to new programming models.

6.1 Introduction

One of the main challenges facing performance portability today is the sheer variety of programming models that exist. Not every model is a good fit for every application, and it can be difficult for new and even existing but evolving models to gain traction in the community. Developers are wary of porting to a model that may no longer be developed or supported in a year or two. Furthermore, even among the most popular, widely-supported models, performance portability is not guaranteed (Deakin et al. (2019, 2020)), either due to machine-specific optimizations in the application or the model's implementation (or lack thereof) for a given architecture.

The underlying issue here is developer *productivity*. Even if the ideal performance portable language or library existed, developers of large, existing applications would be reluctant to port to it because of the effort involved, even with incremental ports. The average development team doesn't have the time or compiler building expertise required to make a program to automate the port, and existing tools they could use also require compiler experts.

But many of the changes made during porting follow patterns, so it seems like automation *should* be possible. MARTINI allows users to capture these patterns to automate the porting process. MARTINI's matchers and replacers allow for separation of concerns with respect to implementing and parallelizing an application. Developers can write the simplest, "textbook" version of their algorithms, and then transform it as needed into an optimized, parallel version in a performance portable model, like OpenMP or Kokkos. Being able to store porting strategies as transformations allows developers to maintain a simple codebase while still achieving high performance and performance portability.

The rest of the chapter is organized as follows. Section 6.2 will discuss an early case study of porting CUDA (Nvidia's proprietary GPU programming language) to HIP (AMD's more portable GPU programming language). Section 6.3 will describe how MARTINI can add OpenMP pragmas to users' code. Section 6.4 will discuss porting two miniapps from OpenMP to Kokkos, and Section 6.5 will further discuss porting one of those miniapps from Kokkos to SYCL. Section 6.6 will evaluate these ports.

6.2 HIPIFY

For a realistic first evaluation we performed a case study against `HIPIFY` (ROCm Developers (n.d.)), a state-of-the-art source code rewriting tool that ports CUDA codes to the (very similar, but more portable) HIP programming model. `HIPIFY` exists in two versions, as a "legacy" perl script, `hipify-perl`, and as an extension to the Clang front end, `hipify-clang`. Both versions have been used by researchers and application developers to port code, with generally positive results (Brown, Abdelfattah, Tomov, and Dongarra (2020); Dufek et al. (2021); Sun et al. (2018)).

```
// 31 lines of C++ code (removed)
OS << kern;
if (caleeDecl->isTemplateInstantiation())
  OS << ")";
OS << ", ";

// Next up are the four kernel configuration parameters, the
   last two of which are optional and default to 0.

// Copy the two dimensional arguments verbatim.
for (unsigned int i = 0; i < 2; ++i) {
 string sArg = readSourceText(*SM,
   config->getArg(i)->getSourceRange()).str();
 bool bDim3 = equal(sDim3.begin(), sDim3.end(), sArg.c_str());
 OS << (bDim3 ? "" : sDim3) << sArg << (bDim3 ? "" : ")") <<
   ", ";
}
// 31 lines of C++ code (removed)
```

Listing 6.1 Excerpt of `hipify-clang` source translating CUDA kernel launches to HIP. The replacement function alone (`HipifyAction::cudaLaunchKernel`) is 73 lines (excluding comments and helper functions). This snippet pretty-prints the grid and block dimensions to the output file. It still inspects the input code string (e.g., by scanning for the string `sDim3 = "dim3("`), despite AST matching of CUDA kernels being done earlier elsewhere. This kind of string matching code is hard to read, hard to modify, and overall fragile as typedefs or syntactic deviations (e.g., spaces) impact it easily.

To bootstrap our `HIPIFY` clone, we used the rewrite rules already defined in the `hipify-clang` source. We limited our `HIPIFY` to CUDA's runtime API for now to keep the number of matchers and replacers manageable for debugging our prototype. The existing tables from `hipify-clang` allowed us to automatically generate matcher-replacer pairs for CUDA runtime calls and types, including all simple renames, such as `cudaMalloc` to `hipMalloc`. However, `hipify-clang` requires that transformations more complex than renaming, such as kernel calls, be implemented explicitly with strong coupling to the Clang AST. Listing 6.1 illustrates this with an excerpt of the `hipify-clang` source code for rewriting

CUDA kernel launches. These particular lines pretty-print the thread grid and block dimensions to the output stream (`OS`). For brevity we omit 62 lines of this function, as well as all helpers and the logic that creates and applies the AST matcher. Still, all of this complexity is required just to replace CUDA kernel calls.

In Figure 12 we illustrate our alternative approach, which does not require any interaction with an AST, or any other complexity. In Fig. 12a, the matcher for a CUDA kernel launch with two kernel arguments and three launch parameters is shown. Since we currently do not support optional arguments, we automatically generate matchers and replacers for all supported numbers of arguments and launch parameters explicitly with a script. While we will support such variability more concisely in the future, one can already see how our approach is fundamentally simpler and more natural to non-compiler experts. Neither the matcher nor the replacer require interaction with the AST or other compiler internals, but all benefits over text-based search-and-replace approaches are preserved. For example, all matcher parameters (e.g., `kern` and `nthreads`) can bind to arbitrary complex expressions in the user's code. Figure 12b shows the associated replacer pattern. The kernel name (`kern`), together with the launch parameters (`nthreads` converted to `dim3`), are moved to argument positions in HIP's kernel launch function.

Importantly, the replacer pattern is written directly in the target language, which makes it easy for any developer to change the argument order, adjust default values (here, 0 in the HIP kernel launch), or modify the transformation in other ways. As an example, the shown replacer will not only port a CUDA kernel launch to HIP, but also double the number of threads to account for the (usually) larger wave size on AMD GPUs compared to the warp size on NVIDIA GPUs.

The two characters added for this modification are the "`* 2`" in the definition of `nthreads3D` in Fig. 12b.

While our prototype emits the literal code in the `clang::matcher_block`, including the conditional that converts integer grid sizes to `dim3` types, we intend to add further capabilities to replacers such that compile-time constant expressions like this can be simplified. In this example, the `is_integer` condition can be determined at replacement time, which would allow the ternary expression to be simplified.

6.3 Inserting OpenMP Pragmas

A popular way of improving loop performance is by adding parallelism, for example with a pragma-based programming model such as OpenMP or OpenACC. These models allow users to add parallelism to their code with minimal changes that keep the code clean. However, due to the variability in compiler support and differences in target architectures, the same pragmas may give very different performance results on different machines. OpenMP in particular requires different directives to make use of GPU hardware. Some support has been added to ameliorate this, but a solution such as ours that could automate inserting per-machine directives would strongly preferable.

We can, for instance, decide to parallelize the outer loop of a nest of three loops. When we know that the iterations are independent from each other, we can also insert the keyword `simd`. For instance, we can parallelize the outer loop of a matrix-matrix multiplication, after making sure the loops are in the best order. Listing 6.2 presents the matcher and replacer to insert pragmas.

```
[[clang::matcher("omp")]]
auto loopm() {
    [[clang::rewrite_setup]]
    int M, N, K;
```

```
    [[clang::matcher_block]] {
        for( int i = 0 ; i < M ; i++ ){
            for( int k = 0 ; k < K ; k++ ){
                for( int j = 0 ; j < N ; j++ ){
                    clang_rewrite::loop_body();
                }
            }
        }
    }
}

[[clang::replace("omp")]]
auto loopr() {
    [[clang::rewrite_setup]]
    int M, N, K;
    [[clang::matcher_block]] {
#pragma omp parallel for simd
        for( int i = 0 ; i < M ; i++ ){
            for( int k = 0 ; k < K ; k++ ){
                for( int j = 0 ; j < N ; j++ ){
                    clang_rewrite::loop_body();
                }
            }
        }
    }
}
```

Listing 6.2 OpenMP: insertion of loop parallelization pragmas.

6.4 OpenMP to Kokkos

This section will discuss porting two miniapps, TeaLeaf and BabelStream, from OpenMP to Kokkos – Kokkos is often considered more performance portable than OpenMP because it does not require different code to target GPUs, only choosing a different back end. TeaLeaf solves a 2-dimensional linear heat conduction equation on a regular grid using a 5-point stencil and various solvers (McIntosh-Smith et al. (2017)). BabelStream is a collection of implementations of the classic STREAM benchmark (McCalpin (n.d.)) in various programming models.

6.4.1 TeaLeaf. Despite the seeming complexity of TeaLeaf compared to BabelStream, because TeaLeaf does not use classes to differentiate between programming models, it was actually significantly easier to translate. The translation was done with two matcher/replacer pairs, one for simple loops and the other for loops with reductions. The matcher and replacer for simple loops are shown in Listing 6.4; the matcher and replacer for reductions are very similar and thus omitted for brevity.

In TeaLeaf, the vast majority of computation is done in doubly-nested for loops (see Listing 6.3 for two examples) that iterate over one dimensional arrays of size `x*y`, where `x` and `y` are the height and width of a matrix, or sub-arrays of these. To make a generic matcher and replacer for all these loops, we used the index arithmetic shown in the replacer, where the `x`- and `y`-indices are calculated from a singular index, and then checked against the old loop bounds. This index arithmetic was challenging to arrive at and implement correctly (a bug we encountered will be discussed in Sec. 6.4.1.1), and if a human programmer had to do this by hand for all of TeaLeaf's loops, they would almost certainly have introduced bugs. Since we only had to write this code twice, however, it was easy for us to verify that both were correct, then apply the transformations globally. This same index arithmetic is how TeaLeaf was translated by hand. Listing 6.5 shows the result of translating the kernels from Listing 6.3 with our matchers and replacers, one of which is shown in Listing 6.4. This matcher and replacer set makes use of our knowledge of the TeaLeaf code, which always uses variables named `x` and `y` for the maximum loop bounds. We use these variables as literals in the replacer.

```
#pragma omp parallel for
    for(int jj = halo_depth; jj < y-1; ++jj)
```

```
    {
        for(int kk = halo_depth; kk < x-1; ++kk)
        {
            const int index = kk + jj*x;
            kx[index] = rx*(w[index-1]+w[index]) /
                (2.0*w[index-1]*w[index]);
            ky[index] = ry*(w[index-x]+w[index]) /
                (2.0*w[index-x]*w[index]);
        }
    }

    double rro_temp = 0.0;

#pragma omp parallel for reduction(+:rro_temp)
    for(int jj = halo_depth; jj < y-halo_depth; ++jj)
    {
        for(int kk = halo_depth; kk < x-halo_depth; ++kk)
        {
            const int index = kk + jj*x;
            const double smvp = SMVP(u);
            w[index] = smvp;
            r[index] = u[index]-w[index];
            p[index] = r[index];
            rro_temp += r[index]*p[index];
        }
    }

    // Sum locally
    *rro += rro_temp;
```

Listing 6.3 TeaLeaf: Sample computation kernels.

```
[[clang::matcher("kokkos")]]
auto kokkos_m() {
    [[clang::rewrite_setup]]
        int k, N, l, M;

    [[clang::matcher_block]] {
        #pragma omp parallel for
        for (int i = k; i < N; ++i) {
            for (int j = l; j < M; ++j) {
                clang_rewrite::loop_body();
            }
        }
    }
}
```

```
[[clang::replace("kokkos")]]
auto kokkos_r() {
    [[clang::rewrite_setup]]
        int k, N, l, M, x, y;

    [[clang::matcher_block]] {
        Kokkos::parallel_for(x*y, KOKKOS_LAMBDA (const int
   idx) {
            const size_t kk = idx % x;
            const size_t jj = idx / x;

            if (k-1 < jj && jj < N && l-1 < kk && kk < M) {
                clang_rewrite::loop_body();
            }
  });
    }
}
```

Listing 6.4 TeaLeaf: OpenMP to Kokkos, doubly-nested for loops.

```
    Kokkos::parallel_for(x*y, KOKKOS_LAMBDA (const int idx) {
      const size_t kk = idx % x;
      const size_t jj = idx / x;

      if (halo_depth-1 < jj && jj < y-1 && halo_depth-1 < kk
   && kk < x-1) {
            const int index = kk + jj*x;
            kx[index] = rx*(w[index-1]+w[index]) /
                (2.0*w[index-1]*w[index]);
            ky[index] = ry*(w[index-x]+w[index]) /
                (2.0*w[index-x]*w[index]);
        }
    });

    double rro_temp = 0.0;

    Kokkos::parallel_reduce(x*y, KOKKOS_LAMBDA (const int
   idx, double& intermed) {
  const size_t kk = idx % x;
        const size_t jj = idx / x;

        if (halo_depth-1 < jj && jj < y-halo_depth &&
   halo_depth-1 < kk && kk < x-halo_depth) {
            const int index = kk + jj*x;
            const double smvp = SMVP(u);
```

```
            w[index] = smvp;
            r[index] = u[index]-w[index];
            p[index] = r[index];
            intermed += r[index]*p[index];
        }
    },
    rro_temp);

    // Sum locally
    *rro += rro_temp;
```

Listing 6.5 TeaLeaf: Example kernels translated to Kokkos.

6.4.1.1 Hand Edits Required. Only minor changes needed to be done by hand to produce a working, translated application. The main modification we did by hand was changing all TeaLeaf's arrays of doubles into Kokkos views, which was not strictly necessary but enables the application to make better use of Kokkos' various back ends. We also added calls to `Kokkos::initialize()` and `Kokkos::finalize()` and added the Kokkos includes.

The other fix required was in fact a bug introduced by MARTINI to the application, which involved the index arithmetic described above. For the loops that go from `0` to `x*y`, this index arithmetic will have `kk` and `jj`, both unsigned `size_t` variables, compared to `0-1`, which evaluates as false when it should be true. We had to manually fix these loops, by commenting out the comparison, to make the application correct. We could write a matcher/replacer pair specifically for the case when `k` and/or `l` is `0`, but since we currently have no way to control whether that matcher or the one shown in Listing 6.4 has precedence, or a way to omit the `if` in the replacer when `k` or `l` is `0`, we had to make this fix by hand. In the future, we plan to add more control structures to matchers and replacers so that bugs like this can be avoided.

6.4.1.2 Summary of TeaLeaf Translation.

MARTINI correctly translated 6 out of 7 kernels for TeaLeaf's CG solver, 3 out of 3 for the Chebyshev solver, 3 out of 4 for the Jacobi solver, 3 out of 3 for the PPCG solver, and 4 out of 4 for some shared methods, giving us a grand total of 19 out of 21 kernels translated correctly. The two incorrect translations were due to the bad signedness comparison discussed above (Sec. 6.4.1.1), and were simple to fix by hand.

6.4.2 BabelStream.

Translation of OpenMP to Kokkos was done with four matcher/replacer pairs, three of which are shown here (the fourth is a trivial deletion). The first matcher/replacer set, shown in Listing 6.8, transforms allocation of one of OpenMP's arrays into allocation (example shown in Listing 6.6) and setup of multiple Kokkos views (for device and host, example shown in Listing 6.7), illustrating the first challenges we encountered and a new feature implemented to resolve it. We wished to create variable names (`device_x` and `hostmirror_x`) based on an existing variable name (`x`), but had no way to do so. Thus, we introduced the new `code_literal()` syntax. When binding code snippets from the original source to parameters in the matcher, MARTINI will look for declarations assigned a `code_literal()` and create a new binding based on the strings passed. The `to_str()` dummy function exists to ensure users can create a new variable name based off a variable of any type and still have it type-check. While we would prefer to have the `code_literal()` calls embedded in the `matcher_block`, the requirements of C++ make this extremely difficult, if not impossible in some cases, so we opted to use the syntax shown instead.

```
template <class T>
OMPStream<T>::OMPStream(const int ARRAY_SIZE, int device)
{
  array_size = ARRAY_SIZE;

  // Allocate on the host
```

```
  this->a = (T*)aligned_alloc(ALIGNMENT,
   sizeof(T)*array_size);
  this->b = (T*)aligned_alloc(ALIGNMENT,
   sizeof(T)*array_size);
  this->c = (T*)aligned_alloc(ALIGNMENT,
   sizeof(T)*array_size);
}
```

Listing 6.6 BabelStream: Variable allocation in OpenMP.

```
template <class T>
OMPStream<T>::OMPStream(const int ARRAY_SIZE, int device)
{
  Kokkos::initialize();

  array_size = ARRAY_SIZE;

  // Allocate on the host
    device_a = new Kokkos::View<T*>("label change me",
   array_size);
    hostmirror_a = new typename
   Kokkos::View<T*>::HostMirror();
    *hostmirror_a = Kokkos::create_mirror_view(*device_a);
  ;;
    device_b = new Kokkos::View<T*>("label change me",
   array_size);
    hostmirror_b = new typename
   Kokkos::View<T*>::HostMirror();
    *hostmirror_b = Kokkos::create_mirror_view(*device_b);
  ;;
    device_c = new Kokkos::View<T*>("label change me",
   array_size);
    hostmirror_c = new typename
   Kokkos::View<T*>::HostMirror();
    *hostmirror_c = Kokkos::create_mirror_view(*device_c);
  ;;
}
```

Listing 6.7 BabelStream: Automated translation of variable allocation into Kokkos.

This particular matcher/replacer pair is also part of a dummy class (`FakeClass`), because of the way BabelStream is implemented. BabelStream uses classes to determine which programming model is used, so the arrays it operates on are class members. This means they have different AST nodes from regular

variables, and in order to match on them, the variables in the matcher and replacer must also be class members. Hence, the matcher and replacer are made members of a simple, dummy class, as is the variable allocation they match.

```
template <class T>
[[clang::matcher("alloc")]]
auto FakeClass::decl_m() {
    [[clang::rewrite_setup]]
        int align, array_size;

    [[clang::matcher_block]] {
        this->x = (T*)aligned_alloc(align,
   sizeof(T)*array_size);
    }
}

template <class T>
[[clang::replace("alloc")]]
auto FakeClass::decl_r() {
    [[clang::rewrite_setup]]
        int array_size;
    [[clang::rewrite_setup]]
        T* x;
    [[clang::rewrite_setup]]
        T* device_x = &clang_rewrite::code_literal("device_"
   + clang_rewrite::to_str(x));
    [[clang::rewrite_setup]]
        T* hostmirror_x =
   &clang_rewrite::code_literal("hostmirror_" +
   clang_rewrite::to_str(x));

    [[clang::matcher_block]] {
        device_x = new Kokkos::View<T*>("label change me",
   array_size);
        hostmirror_x = new typename
   Kokkos::View<T*>::HostMirror();
        *hostmirror_x = Kokkos::create_mirror_view(*device_x);
    }
}
```

Listing 6.8 BabelStream: OpenMP to Kokkos, variable allocation.

The second matcher/replacer pair, shown in Listing 6.9, transforms OpenMP parallel loops (example shown in Listing 6.10) into Kokkos parallel loops, as well

as creating Kokkos views for the variables used (utilizing our knowledge of the variable names used by BabelStream) and inserting a call to `Kokkos::fence()` (example shown in Listing 6.11). The latter two transformations are required because of how BabelStream is implemented; we wished to produce code as close to the hand-written version as possible, so we respected the developers' choice to have the class representing their Kokkos implementation contain pointers to Kokkos views, and then for each kernel, initialize the actual views used. The call to `Kokkos::fence()` ensures that the BabelStream timing infrastructure can record accurate runtimes for each kernel. The `loop_body` syntax will match whatever is inside the for loop in the original code and copy it directly into the rewritten code.

```
[[clang::matcher("kokkos")]]
auto kokkos_m() {
    [[clang::rewrite_setup]]
        int k, N;

    [[clang::matcher_block]] {
        #pragma omp parallel for
        for (int i = k; i < N; i++) {
            clang_rewrite::loop_body();
        }
   }
}

template <class T>
[[clang::replace("kokkos")]]
auto kokkos_r() {
    [[clang::rewrite_setup]]
        int k, N, j;
    [[clang::rewrite_setup]]
        Kokkos::View<T*> *device_a, *device_b, *device_c;

    [[clang::matcher_block]] {
        Kokkos::View<T*> a(*device_a);
        Kokkos::View<T*> b(*device_b);
        Kokkos::View<T*> c(*device_c);

        Kokkos::parallel_for(N-k, KOKKOS_LAMBDA (const int i)
   {
```

```
            clang_rewrite::loop_body();
        });
        Kokkos::fence();
    }
}
```

Listing 6.9 BabelStream: OpenMP to Kokkos, basic parallel for loop.

```
template <class T>
void OMPStream<T>::mul()
{
  const T scalar = startScalar;

  #pragma omp parallel for
  for (int i = 0; i < array_size; i++)
  {
    b[i] = scalar * c[i];
  }
}
```

Listing 6.10 BabelStream: Example OpenMP kernel.

```
template <class T>
void OMPStream<T>::mul()
{
  const T scalar = startScalar;

    Kokkos::View<T*> a(*device_a);
    Kokkos::View<T*> b(*device_b);
    Kokkos::View<T*> c(*device_c);

    Kokkos::parallel_for(array_size-0, KOKKOS_LAMBDA (const
   int i) {
        b[i] = scalar * c[i];
    });
    Kokkos::fence();
}
```

Listing 6.11 BabelStream: Example kernel translated to Kokkos.

The third matcher/replacer pair, shown in Listing 6.12, translates a reduction kernel and demonstrates an extra feature of the `loop_body()` syntax, which is inline matchers and replacers. Since Kokkos requires an intermediate reduction variable for each thread executing the kernel, we cannot directly copy

the OpenMP reduction loop body, we must modify it slightly first. The argument pair given to `loop_body()` represents a simple matcher/replacer pair, where the first item in the pair is the matcher and the second is the replacer. In this case the original reduction variable (`red`) is replaced with the new intermediate variable from the lambda arguments (`intermed`).

```
[[clang::matcher("reduction")]]
auto red_m() {
    [[clang::rewrite_setup]]
        int k, N;
    [[clang::rewrite_setup]]
        double red;

    [[clang::matcher_block]] {
        #pragma omp parallel for reduction (+:red)
        for (int i = k; i < N; ++i) {
            clang_rewrite::loop_body();
        }
    }
}

template <class T>
[[clang::replace("reduction")]]
auto red_r() {
    [[clang::rewrite_setup]]
        int k, N, j;
    [[clang::rewrite_setup]]
        double red;
    [[clang::rewrite_setup]]
        Kokkos::View<T*> *device_a, *device_b, *device_c;

    [[clang::matcher_block]] {
        Kokkos::View<T*> a(*device_a);
        Kokkos::View<T*> b(*device_b);
        Kokkos::View<T*> c(*device_c);

        Kokkos::parallel_reduce(N-k, KOKKOS_LAMBDA (const int
   i, T& intermed) {
            clang_rewrite::loop_body({{red, intermed}});
        },
        red);
    }
}
```

Listing 6.12 BabelStream: OpenMP to Kokkos, loop with reduction.

The final matcher/replacer pair, not shown for brevity, simply deletes calls to `free()` on the original arrays which are no longer needed now that the arrays are Kokkos views.

6.4.2.1 Hand Edits Required. Despite the seeming dissimilarity between OpenMP and Kokkos, very few hand-done edits were required to produce functioning Kokkos from our automatic translation. The bulk of the edits were restricted to a single kernel, `read_arrays`, which is drastically different from the rest of the kernels in BabelStream. We currently have no way to differentiate that kernel from the rest, so it had to be fixed by hand. We intend to add syntax to differentiate this kernel from the rest so it can be automatically translated as well. The rest of the edits required were minor, such as adding the Kokkos includes, inserting a call to `Kokkos::initialize()` and `Kokkos::finalize()`, and changing the types of variable declarations in the header file.

6.5 Kokkos to SYCL

This section discusses translating BabelStream from Kokkos to SYCL. TeaLeaf has no SYCL implementation so we chose to restrict our discussion to BabelStream.

6.5.1 BabelStream. Though Kokkos and SYCL are, at a surface level, more similar than OpenMP and Kokkos, the Kokkos to SYCL translation presented more challenges than OpenMP to Kokkos. Many of these are due to the class structure used to implement BabelStream and heavy reliance on lambdas. Translation was done primarily with three sets of matchers and replacers. The first, not shown for brevity, translates the instantiation of a Kokkos view to a

SYCL accessor. Ideally, this translation would include setting the requested access privileges (read, write, or read-write), but since this is intended to be a generic matcher and replacer we leave that as an optimization the user can make once the translation is finished.

```
template <class T>
[[clang::matcher("kokkos")]]
auto kokkos_m() {
  [[clang::rewrite_setup]]
    int array_size;

  [[clang::matcher_block]] {
    Kokkos::parallel_for(array_size, KOKKOS_LAMBDA (const T
   idx) {
      clang_rewrite::loop_body();
    });
  }
}

template <class T>
[[clang::replace("kokkos")]]
auto kokkos_r() {
  [[clang::rewrite_setup]]
    size_t array_size;
  [[clang::rewrite_setup]]
    std::unique_ptr<sycl::queue> queue;

  [[clang::matcher_block]] {
    queue->submit([&] (sycl::handler &cgh) {
      cgh.parallel_for(sycl::range<1>{array_size},
   [=](sycl::id<1> idx) {
        clang_rewrite::loop_body();
      });
    });

  }
}
```

Listing 6.13 BabelStream: Kokkos to SYCL, kernel translation.

The second matcher/replacer pair, shown in Listing 6.13, and the source of most challenges we encountered, translates a Kokkos `parallel_for()` kernel

into a SYCL kernel. Since we currently have no support for matching a series of statements (we can only match on one top-level statement), we were unable to match on the Kokkos view instantiations *and* Kokkos kernel call to create SYCL accessors inside the SYCL kernel. Hence, we have separate matchers for turning Kokkos views into SYCL accessors (see above) and translating the kernels. We also could not rewrite the names of Kokkos views accessed in the loop body to their accessor equivalents. All of this had to be fixed by hand, but is comparatively trivial compared to generating the kernel itself.

The final matcher/replacer pair, not shown for brevity, simply translates calls to `Kokkos::fence()` to SYCL's `queue->wait()`.

6.5.1.1 Hand Edits Required. Similarly to Kokkos, a number of trivial edits were required, such as changing the type of header file declarations, adding SYCL includes, and initializing the SYCL queue. As mentioned above, SYCL also required further, more significant, hand editing to produce working code. We had to manually move the SYCL accessor declarations inside the kernel submission, add handlers to the accessor declarations, and rename array accesses inside kernels to their accessor equivalent. Once again, the `read_arrays()` kernel was an outlier that required hand-translation, since it is unique among the others.

6.5.1.2 Summary of BabelStream Translation. MARTINI was able to successfully translate 9 out of 10 OpenMP kernels and functions to Kokkos, and 7 out of 10 Kokkos kernels and functions to SYCL. While some hand edits were required in both cases, particularly for SYCL, MARTINI did the bulk of the translation automatically, with only two or three lines of code needing editing per kernel for SYCL, and less than that for Kokkos.

6.6 Evaluation

This section will evaluate MARTINI and the resulting code for the previous sections in this chapter.

6.6.1 HIPIFY. Our evaluation machine has two 14-core, hyperthreaded Intel Xeon(R) E5-2680 v4 CPUs running at 2.40GHz, 128 GB of RAM, and two AMD Instinct MI100 GPUs. HIP codes were compiled using `hipcc` 4.4.21432-f9dccde4 based on AMD Clang 13.0.0 and ROCm 4.5.2 and the same version of `hipify-perl`. We used `hipify-clang` with git hash `61241a4` compiled using gcc 9.3.0 and the same LLVM version as MARTINI, which is hash `4c2b57ae` from LLVM's main branch.

We compare translating a simple gravitational N-body simulation code[1] from CUDA to HIP with MARTINI-HIPIFY and AMD's HIPIFY. It features four variants: unoptimized (nbody-orig), struct-of-arrays (SOA) data layout (nbody-soa), cache blocked (nbody-block), and unrolled loops (nbody-unroll).

6.6.1.1 Performance. Since we cannot compile and run both CUDA and HIP on the same device on our testing machine, it would be unfair to compare the performance obtained by the original CUDA and translated HIP codes. Therefore, we only compare the performance of the automatically translated HIP codes created by `hipify-perl`, `hipify-clang`, and MARTINI.

The performance we obtained for each translation is given in Table 2. Each version was run for ten iterations, and the average and standard deviation run times per iteration are presented in ms. As expected, all three translators generate very similar code with very similar performance for both medium and larger-size problems, regardless of application version.

[1] `https://github.com/harrism/mini-nbody`

Interestingly, when we generated code that multiplied the number of threads by two ("#Threads x2" columns in Table 2), as done in Fig. 12, performance greatly improved on the larger problem size for all versions of the application except nbody-orig (which is a naive implementation where little performance gain is expected). Performance also improved on the smaller problem size for nbody-block and nbody-unroll. These numbers are bolded in the table. This is due to the wider thread waves on AMD GPUs compared to thread warps on NVIDIA GPUs. `hipify-perl` and `hipify-clang` are unable to make these kinds of changes easily, as we will discuss in Sec. 6.6.1.2.

		MARTINI-HIPIFY				AMD HIPIFY			
	Number	Unmodified		#Threads x2		hipify-perl		hipify-clang	
Benchmark	Particles	Mean	Stddev	Mean	Stddev	Mean	Stddev	Mean	Stddev
nbody-block	30000	319.65	5.42	**104.62**	0.11	319.62	3.44	318.73	6.34
	300000	457.65	1.35	**210.60**	0.90	453.55	1.63	449.20	2.71
nbody-orig	30000	171.99	0.47	172.43	0.35	171.63	0.78	172.63	0.89
	300000	415.97	2.11	418.35	0.75	414.98	2.72	416.91	2.38
nbody-soa	30000	198.45	1.66	197.19	1.78	205.60	1.19	205.38	2.40
	300000	426.05	0.42	**363.84**	2.67	429.44	0.54	428.72	2.76
nbody-unroll	30000	332.87	2.08	**180.71**	0.61	334.45	1.98	335.24	2.36
	300000	470.70	0.95	**229.27**	0.64	471.06	0.51	469.65	1.59

Table 2. Execution time in ms of the HIP output code for the N-body benchmark.

6.6.1.2 Usability.

Both MARTINI-HIPIFY and AMD's `hipify-clang` are command line compiler tools, but while the core of `hipify-clang`[2] is approximately 1,000 lines (excluding comments and newlines) of AST matchers and C++ making heavy use of Clang internals, MARTINI-HIPIFY is 5,672 lines of simple CUDA/C++ (again excluding comments and newlines), the vast majority of which was automatically generated using tables in `hipify-clang` that convert CUDA names to HIP names. Of the 712 matchers and replacers generated, 212 were for kernel launches with varying numbers of

[2]HipifyAction.cpp/.h and main.cpp.

launch parameters and arguments (this will be reduced by at least an order of magnitude once optional arguments are implemented), and the remaining 500 were for CUDA runtime functions and types. Of those 500, only 46 needed to be fixed by hand due to problems our generator script had getting the correct types from the CUDA headers (this will hopefully be fixed in later versions, as our support for types improves). As a rough comparison of code complexity, all of the matchers and replacers in our `HIPIFY` have a McCabe cyclomatic complexity (McCabe (1976)) of 1, while AMD's `hipify-clang` has an average cyclomatic complexity of 6.7 (calculated with `pmccabe`).

For a more concrete comparison, consider a user who wants to make a simple modification to the translation of CUDA kernel calls into HIP: multiplying the number of threads by two to improve performance on AMD devices, which generally have wider threading than NVIDIA devices. To do so with `hipify-clang`, that user would have to 1) determine that HipifyAction.cpp is where most of the translation is done, 2) find the function `HipifyAction::cudaLaunchKernel()`, 3) analyze the 73 lines of code in that function to find where the kernel configuration is handled, and 4) determine where in the relevant string manipulation code (shown in Listing 6.1) to insert their `* 2`. Without knowledge of the Clang AST and Clang's source manipulation libraries this is incredibly difficult, time-consuming, and highly dependent on the (in-source) documentation of `hipify-clang`.

To do the same thing with MARTINI-HIPIFY, the user would only have to modify the kernel call replacers similarly to what is shown in Fig. 12b. The replacers are easy to find by searching for the HIP kernel launch function name, and the modification could be done with a traditional search-and-replace tool. No

understanding of Clang internals is necessary to modify MARTINI-HIPIFY. (It was a simple matter for us to modify the script that generated MARTINI-HIPIFY so all kernel call replacers looked like the one in Fig. 12b.) Other modifications, for example, printing the size of all arrays allocated on the device, are similarly intuitive.

6.6.2 Inserting OpenMP Pragmas. We compiled both the original and transformed matrix multiplication code with gcc's `-O3` optimization level. We ran all our experiments on a machine featuring two 16-core, 32-thread Intel(R) Xeon(R) CPU E5-2697 v4 CPUs running at 2.30GHz and 128 GB of memory. All the code was compiled using gcc 12.2.0 on a Linux RHEL running a kernel v. 4.18. We measured time using either `clock_gettime`, or reading the hardware timestamp counter. All the measurements were taken 10 times, and we present the mean and standard deviation on the plots shown in this section.

As presented in Section 6.3, we implemented a matrix-matrix multiplication (with loops in the correct order) and we generated an OpenMP version that parallelizes the outer loop. A comparison of the performance is shown Figure 13, executed using 32 threads. As expected, the OpenMP version outpaced the original version, demonstrating both the power of parallelism and how useful our tool could be for a first attempt at porting to OpenMP.

6.6.3 TeaLeaf. We evaluated the performance of the transformed code on a machine equipped with two Intel Xeon CPU E5-2697 v4 processors running at 2.30GHz and 128 GB of RAM. We used Kokkos cloned from the Git repository (commit `1a3ea28f6`) and compiled with gcc 12.2.0. In order to get fair comparisons between the CPU OpenMP code and the Kokkos versions, we used the OpenMP backend of Kokkos.

Figure 14 shows a comparison between the execution times of Jacobi and Chebychev kernels, using the OpenMP 3.1, OpenMP 4, Kokkos, and OpenMP-to-Kokkos versions, using the OpenMP backend for the Kokkos implementations. We ran every experiment 50 times, and plots show the mean and standard deviation of each vlaue.

TeaLeaf's documentation recommends using `OMP_PROC_BIND=spread` and `OMP_PLACES=threads` with their OpenMP 4 kernel and `OMP_PROC_BIND=true` with OpenMP 3.1. We noticed this was actually harming performance (the execution time was multiplied by 5 on some kernels), so we used the default configuration for all the kernels. On Figure 14b, we did not run the OpenMP 4 version since it was producing a numerical error. The OpenMP 4 implementation also gives very different performance on the Chebychev kernel (Figure 14a).

For these reasons, we can focus our observation on the OpenMP 3.1, Kokkos and OpenMP-to-Kokkos translated versions. We can see that their performance is very close to each other, and the translated code is slightly slower. We have tried to isolate this difference with microbenchmarks from the BabelStream suite, which are presented in the next section.

6.6.4 BabelStream. Figure 15 shows a comparison between the performance obtained by the translated Kokkos code and the original OpenMP code. We can see that the Kokkos code is slower: this can be explained by the fact that the translation creates all the Kokkos views for the three main variables, even though some of the kernels only use 2 out of 3. It adds some latency, which can be observed consistently on all four kernels.

Translating the Kokkos code is fast and simple, but since we are switching between parallel models, some approximations are being made and the generated

code requires some manual optimizations to reach optimal performance. An optimization here would be to remove the view creations that are not needed. MARTINI translations may not always be optimal, but can provide a good starting point and significant effort reduction for porting an application to a new model.

```
__global__ void kern(int a1 =
    0,
     int a2 = 0, /* more args
    */) {}

template<int nblocks, int
    nthreads,
           int shmem, int a1,
    int a2>
[[clang::matcher("launch2a3p")]]
auto launch_2_3_matcher() {
 [[clang::matcher_block]]
 kern<<<nblocks, nthreads,
    shmem>>>(a1, a2);
}
```

(a) Matcher for a CUDA kernel launch with three launch parameters and two kernel arguments. As our prototype is a work in progress, we opted to generate the matchers for varying numbers of launch parameters and arguments explicitly with a script. A matcher has to be valid in the source language, here CUDA, to allow (an unmodified) Clang to generate an AST. Depending on the situation, the user also needs to provide additional declarations, e.g., the `kern` (dummy) function, for the same reason.

```
template<int nblocks, int
    nthreads,
           int shmem, int a1,
    int a2>
[[clang::replace("launch2a3p")]]
auto launch_2_3_replacer() {
  [[clang::matcher_block]] {
  bool nthreadsIs1D =
   numeric_limits
      <decltype(nthreads)>
      ::is_integer;
  auto nthreads3D =
   nthreadsIs1D
      ? dim3(nthreads * 2)
      : nthreads;
  hipLaunchKernelGGL(kern,
   nblocks,
      nthreads3D, shmem, 0,
      a1, a2);
}}
```

(b) HIP kernel launch replacement code for the matcher in Fig. 12a. Replacers and matchers are linked by the name, here `"launch2a3p"`. Non-literal variables that are used in both act like capture groups in regular expressions. The expression in the source code that is bound to them by the matcher is substituted into the end result at use locations in the replacer.

Figure 12. Matcher/replacer pair for CUDA kernel launches with two kernel arguments and three launch parameters.

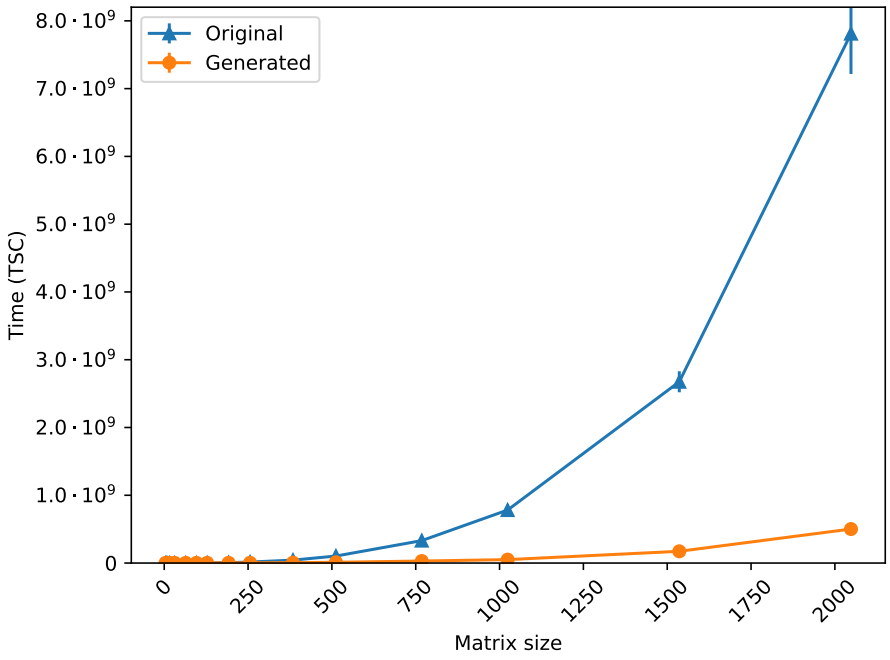
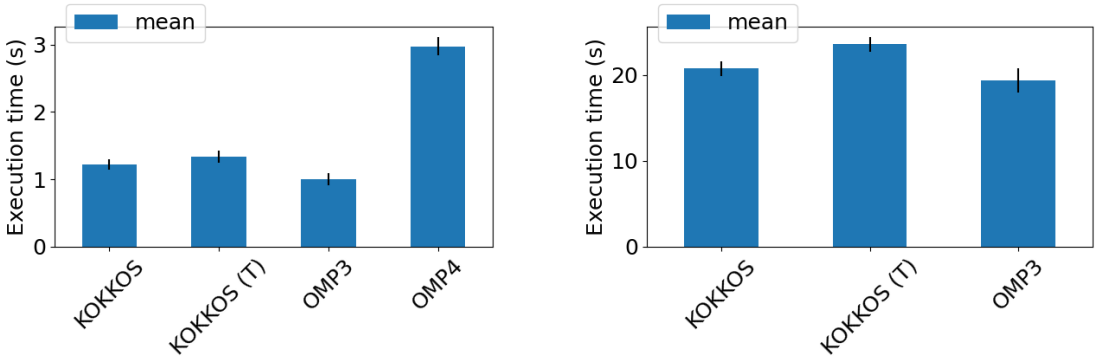


Figure 13. OpenMP pragma insertion: performance obtained by the original code and the generated code.



(a) Chebychev (b) Jacobi

Figure 14. Performance comparison on TeaLeaf kernels between Kokkos, OpenMP 3.1 and OpenMP 4 implementations, and our OpenMP-to-Kokkos translation (reported as KOKKOS(T)).

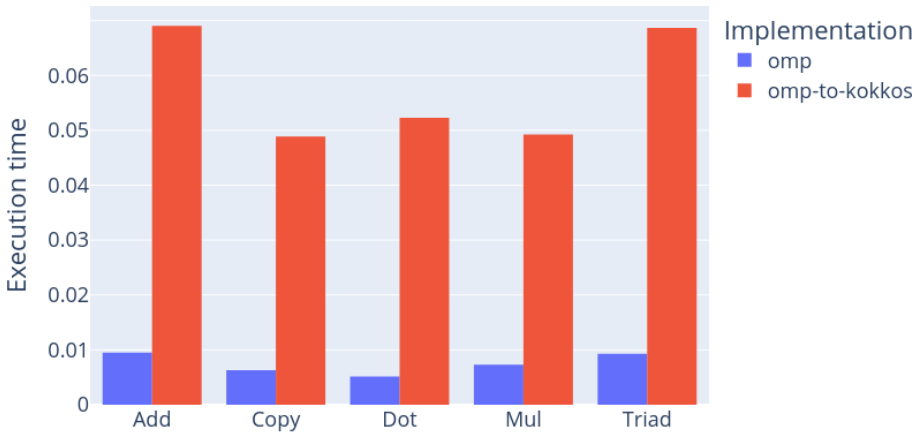


Figure 15. BabelStream benchmark.

CHAPTER VII

REWRITING FOR PERFORMANCE MEASUREMENT

This chapter contains material originally published by Johnson et al. (2022). It also contains material that will be published by Huck et al. (n.d.). Alister Johnson did all development work on MARTINI and related writing, Camille Coti assisted with writing related to the Clang plugin.

This chapter will discuss how MARTINI can be used to instrument source code, in particular for performance measurement.

7.1 An Example: Basic Instrumentation

Profiling, for one reason or another, is something every application developer wants to do at some point. However, maintaining a program version that performs any form of logging is costly, especially given the varied kinds of logging one might want to perform. Most tools that add instrumentation to code do so naively, and instrument every function, whether or not that function is actually of interest. This can make it prohibitively slow to run the application, or overwhelm the developers with data. Even more advanced instrumentation tools that give developers more control, like PDT (Lindlan et al. (2000)), are still very rigid in *how* they instrument and don't allow for custom instrumentation, leading developers to insert their own instrumentation code. While abstractions like templates, macros, and `#ifdef` can help with this, applications making heavy use of them often redesign a multi-level DSL with severe implications for long-term maintainability and readability. Given easy to use, customizable, and fully automatic code rewriting, however, developers can instead create short-lived, special-purpose code versions on-demand while keeping the core application code clean.

While we do not provide a full-fledged instrumentation suite yet, we showcase the benefits of MARTINI for instrumentation tasks in Fig. 16. Through the two simple before-and-after code snippets shown in Fig. 16a, users can easily create a one-off program version that will log every call, including lambdas. As shown in Figs. 16b and 16c, function calls and lambda invocations are replaced by the macros `LOG_FN` and `LOG_LAMBDA`, respectively. Note that, to MARTINI, there is no semantic difference between declaring a variable as an argument to a matcher or replacer and declaring it outside a `[[clang::matcher_block]]`. We demonstrate the former in this example.

Though contrived, this example clearly shows how the power of semantic matching and the simplicity of example-based rewriting come together. Since the rules are reusable, easily customizable, and maintainable by non-expert users, instrumented code can be produced from the original application at any point. Thus, one-off rewriting effectively reduces the maintenance burden while offering more powerful capabilities than "baked-in" instrumentation solutions.

7.2 Instrumenting Functions

One of the most common and important parts of instrumenting applications is the ability to instrument functions. Due to choices made when initially designing MARTINI's matcher and replacer syntax, we had to develop special syntax to be able to match and replace inside function declarations. This syntax is shown in Figure 17a. The `[[clang::function_matcher]]` attribute denotes that this matcher is intended to work across the entire function, and the `function_body()` macro will match all statements in the function body. This syntax is limited to modifying only the function body, not the function signature. Syntax to modify the

function signature (for example, to add a new argument or change the return type) is under development, and a sample is shown in Fig. 17b.

7.3 Arbitrary Instrumentation

Instrumenting functions is very useful, but sometimes applications need more granularity in their data collection. MARTINI can also allow users to instrument arbitrary code structures, such as for loops, if statements, and function calls. In particular, instrumenting for loops is of interest, since most scientific applications spend the vast majority of their time in loops (sometimes multiple loops per function, so function-based profiling is too coarse grained), so knowing which loops are the performance bottlenecks can help developers better allocate their time. An example of instrumenting all loops with two or more nests is shown in Figure 18. Beyond simply inserting instrumentation at arbitrary points, however, MARTINI can insert arbitrary instrumentation code, so users are not limited in the data they can collect. The user may also add a conditional to the `function_body()` (or `loop_body()`) macro to only select functions (or loops) that have interesting features, such as math or conditionals.

7.4 Evaluation

This section will compare MARTINI's instrumentation capabilities to those of three other similar tools with varying degrees of sophistication.

7.4.1 `-finstrument-functions`. The `-finstrument-functions` option (Free Software Foundation, Inc. (n.d.)) is available in most mainstream compilers, including `gcc` and Clang. It will add instrumentation calls at the entries and exits of all functions that the user can then implement to, for example, collect timing data. The user may specify files and functions to ignore, but in general, this option will instrument everything, including libraries the user may not want to

collect data from. It also adds a great deal of overhead, since all functions now have additional calls at entry and exit. Specifying files and functions to ignore to minimize extraneous data and overhead is time consuming and imprecise. MARTINI takes the opposite approach, where users must specify functions or other code structures that have characteristics of interest (e.g., they contain math). This limits instrumentation to code the user cares about, greatly reducing overhead.

7.4.2 TAU Clang Plugin. The TAU Clang plugin (Coti, Denny, et al. (2020); Huck et al. (2024)) works similarly to `-finstrument-functions`, but offers a great deal more control to users in selecting which functions to instrument. Users provide a *selection file* to the plugin, which contains the names of files and functions to include or exclude, and supports wildcard characters, so that, for example, users could exclude all functions named `foo_#` (where `#` works as the wildcard, matching any substring), but *include* a function named `foo_bar`.

This plugin is implemented as a *function pass* over the LLVM IR, and has two modes of operation. The first is function definition instrumentation, which will insert instrumentation calls at the entries and exits of functions. These calls can be to any function, but the plugin currently only allows users to insert a single function call at a time. The second mode is callsite instrumentation, which will find all places functions of interest are called from and insert instrumentation around the calls. To avoid instrumenting very small functions (e.g., getters and setters, which can potentially greatly increase overhead when instrumented), there is also an environment variable to control the minimum number of instructions a function must contain to be instrumented. It is also possible to instrument for loops by selecting the file and line number, though this is somewhat fragile, as line numbers can change.

The main limitations of this plugin are that it cannot instrument arbitrary code regions or add arbitrary instrumentation code, both of which MARTINI can do. The only feature the plugin has that MARTINI does not is the ability to instrument specific line numbers, though as mentioned, this is fragile and susceptible to errors when the code is edited and line numbers change. However, if the user can create a matcher for the desired line of code, MARTINI can match it (and perhaps other, similar lines the user was unaware of) and still instrument it, emulating the plugin's functionality in a more robust manner.

7.4.3 PDT. The Performance Database Toolkit (PDT) (Lindlan et al. (2000)) is an instrumentation tool that also ships as part of TAU (Shende and Malony (2006)). Users can specify which code they would like to instrument with the same DSL-like interface that allows them to select files and functions for instrumentation. It has many of the same features as the Clang plugin and MARTINI, including selective function instrumentation and for loop instrumentation, but it is, again, limited in that it cannot instrument arbitrary code regions or add arbitrary instrumentation code. PDT is perhaps the most sophisticated of these three tools, with the Clang plugin as a very close second, but MARTINI is yet more flexible and powerful. The only feature PDT has that MARTINI does not is, once again, the ability to instrument a specific line of code, though the user can get around this in the same way as they can with the Clang plugin.

As mentioned in Chapter III, PDT was in fact the inspiration for MARTINI. While rewriting PDT to make use of the most recent advances in Clang and LLVM, the question of "what if we could insert arbitrary instrumentation code?" was raised. And thus MARTINI was created.

```
auto fn(auto);
auto LOG_FN(auto, auto) {}
auto LOG_LAMBDA(auto, auto) {}

[[clang::matcher("log_fn")]]
auto fn_call_matcher(int arg) {
  fn(arg);
}
[[clang::replace("log_fn")]]
auto fn_call_replacer(int arg) {
  LOG_FN(fn, arg);
}
[[clang::matcher("log_lambda")]]
auto lambda_call_matcher(int
   arg) {
  auto lambda = [&](int){};
  [[clang::matcher_block]]
  lambda(arg);
}
[[clang::replace("log_lambda")]]
auto lambda_call_replacer(int
   arg) {
  auto lambda = [&](int){};
  [[clang::matcher_block]]
  LOG_LAMBDA(lambda, arg);
}
```

(a) Matcher and replacer pairs for instrumenting a single argument function (`"log_fn"`) and a single argument lambda (`"log_lambda"`).

```
int g(int);
void test() {
  [&](int _){
    g(g(_));
  }(g(0));
}
```

(b) Non-trivial input example with nested calls and a lambda invocation which make it complex for text-based search-and-replace tools. Such tools would also struggle with newlines, comments, and (malicious) strings, such as: `f(g("\"),q(\""))`.

```
int g(int);
void test() {
  LOG_LAMBDA([&](int _){
      LOG_FN(g, LOG_FN(g,
   _));
    }, LOG_FN(g, 0));
}
```

(c) The input from 16b rewritten by MARTINI with the shown example-based rewrite rules. All three function calls are replaced by the `"log_fn"` rule while the lambda invocation was modified by the `"log_lambda"` rule.

Figure 16. Example of how MARTINI can effectively instrument a code base with simple example-based rewrite rules that are semantic context-aware.

```
[[clang::function_matcher("func")]]
int foo() {
    function_body();
}

[[clang::replace_in_body("func")]]
int foom() {
    [[clang::matcher_block]] {
        TRACE_CALL(&foo);
        function_body();
    }
}
```

(a) Matcher and replacer pair for inserting a call tracing macro into functions.

```
[[clang::function_matcher("func")]]
void foo() {
    function_body();
}

[[clang::function_replace("func")]]
int foo() {
    [[clang::matcher_block]] {
        function_body();
        return 42;
    }
}
```

(b) Example matcher and replacer pair to modify a function's signature. Note that further matchers and replacers may be needed to replace all the calls to a modified function.

Figure 17. Matcher and replacer examples for modifying functions, both for instrumentation and general purpose.

```
[[clang::matcher("forloop")]]
void foo_m() {
    [[clang::rewrite_setup]]
        int n, m, N, M;
    [[clang::matcher_block]] {
        for (int i = n; i < N; i++) {
            for (int j = m; j < M; j++) {
                loop_body();
            }
        }
    }
}

[[clang::replace("forloop")]]
void foo_r() {
    [[clang::rewrite_setup]]
        int n, m, N, M;
    [[clang::matcher_block]] {
        START_INST();
        for (int i = n; i < N; i++) {
            for (int j = m; j < M; j++) {
                loop_body();
            }
        }
        STOP_INST();
    }
}
```

Figure 18. Matcher and replacer pair for inserting instrumentation around nested for loops.

CHAPTER VIII

SUMMARY OF RESULTS

This section will summarize the results from this dissertation and conclude.

8.1 Summary

In Chapter I, we described three goals for our code rewriting tool, and now we shall see how MARTINI measures up. Goal (1) was minimizing the complexity of our user interface. While a full usability study is outside the scope of this work, as can be seen in Chapters IV, V, VI, and VII, MARTINI's syntax is minimal and much simpler than the syntax of most other similar tools. It is also purely in the source language, so users do not have to learn anything new to use MARTINI. MARTINI does not limit what users can and cannot express, beyond the limitations imposed by the information available in the AST, which helps MARTINI meet Goal (2), to maximize what we can express. MARTINI allows for both custom and arbitrary rewrites. Goal (3) was to allow users to automate a wide variety of jobs. To this end, MARTINI is source-to-source and makes both the original and rewritten code available so users can test their rewrites and/or produce one-off code versions for special purposes. It can also fully automate edits, so bulk editing is simple. These results and comparisons to other rewriting tools are shown in Figure 19.

Though we have discovered much can be automated, we have also determined some things are difficult to automate. On the simpler end of the scale, many optimizations and instrumentation can be automated. On the other end of the difficulty scale, at a minimum, basic ports can be automated, though to get the best performance, hand edits may be needed. We have plans to add features to MARTINI to ameliorate some of these difficulties, but so far, we see no way

around the need for human developers to look for special cases and performance opportunities. Once those opportunities have been found, however, MARTINI can greatly increase developer productivity in exploiting them.

But how does MARTINI serve performance portability? At the end of Chapter II, we listed several goals and pieces of advice for performance portability systems. MARTINI, as demonstrated in Chapters IV, V, VI, and VII, can help applications and the programming models they use achieve many of these goals. For programming models that have a complicated front end, like SYCL and, to some extent, Kokkos, as shown in Chapter VI, MARTINI can provide an alternative to directly adopting them by allowing users to rewrite their code from a simpler programming model, acting in many ways as a performance portability layer (see Sec. 2.5.2). Similarly, MARTINI can provide another layer of abstraction on top of an existing model.

Where MARTINI truly shines, however, is in allowing the separation of concerns between configuring code and (optimizing) mapping parallelism to hardware. As shown in Chapters V and VI, MARTINI allows developers to maintain the "textbook" version of their application, and then store (parameterized) optimizations elsewhere to produce machine-specific versions. This enables performance portability by allowing users to keep cleaner code that can automatically be ported and optimized for any architecture. Furthermore, it can help users automate porting from one programming model to another so they aren't locked in as strongly if a model ends up not meeting their needs. And so they can verify that the model is giving them the performance they want, MARTINI furthermore allows them to easily produce an instrumented version of their code, as shown in Chapter VII.

8.2 Conclusion

To conclude: this dissertation has presented MARTINI, an automatic code rewriting tool, and provided several studies of its usefulness. MARTINI can be used to maintain, optimize, port, and profile a wide variety of C and C++ applications, but is particularly well suited to larger applications due to its automation capabilities. There is much work yet to be done, but MARTINI has been demonstrated to out-perform other similar rewriting tools and be a boon for performance portability.

	Friendly UI	Custom rewrites	Bulk edits	Original code preserved	Arbitrary rewrites	Rewritten code visible	UI not another language
sed/awk	×	✓	✓	✓	×	✓	×
LLVM pass	×	✓	✓	✓	✓	×	×
Polly and polyhedral model	×	×/varies	✓	✓	×	×/varies	×
Transformer library and ClangMR	×	✓	✓	✓	✓	✓	×
Omni	×	✓	✓	✓	✓	✓	×
ROSE	×	✓	✓	✓	✓	✓	×
Xevolver	×	✓	✓	✓	✓	✓	×
Bones	×	×/tricky	✓	✓	×	✓	✓
Cetus	×	✓	✓	✓	✓	✓	×
Stratego/XT	✓	✓	✓	✓	✓	✓	×
CHiLL	×	✓	✓	✓	✓	✓	×
Coccinelle	✓	✓	✓	✓	✓	✓	×
Orio	✓	✓/tricky	×/slow	✓	✓	✓	×
Nobrainer	✓	✓	✓	✓	×	✓	✓
Lift	×	✓/tricky	✓	✓	×	✓	×
Sydit	✓	✓	×/slow	×	✓	✓	✓
Lase	✓	✓	×/slow	×	✓	✓	✓
Haskell	✓	✓/tricky	✓	✓	×/tricky	×	✓
MARTINI	✓	✓	✓	✓	✓	✓	✓

Figure 19. A comparison of MARTINI and several other code transformation tools, as described in Section 2.8.6. Boxes with an extra note can be read as "generally yes/no, but it's tricky/slow/varies."

CHAPTER IX

FUTURE DIRECTIONS

This chapter will discuss work that remains to be done and some directions future work could go.

9.1 Remaining Development Work

While MARTINI can do a great deal with the portions of the C++ standard it currently supports, there is still much of the standard that is not supported, most notably classes and structs. The syntax around matching these constructs needs to be developed and implemented. It is likely further AST matchers will need to be implemented to support this as well. Support for matching types also needs to be improved. Although having MARTINI be fully polymorphic is currently a strength, there should be support for matching on specific types (and perhaps categories of types, e.g., floating point vs integer vs pointer) as well, and the syntax and semantics surrounding that need to be developed.

The `insert_before` and `insert_after` transformations also require some work, mostly in the area of safety. For example, if the user wishes to insert some code before variable declarations, we don't want to insert code into the bounds of a for loop, which may contain the declaration of the iteration variable. Some work has been done here, but there are certainly more cases we have yet to cover.

Support for matching on and transforming functions, while serviceable, is also still fairly rudimentary. There is much do be done, including developing syntax for changing a function's signature (and then changing all or some of the corresponding calls) and adding an "on exit" transformation attribute that could modify, for example, all the returns or other code at a function's exit points. This will be particularly useful for function instrumentation.

Finally, MARTINI is currently limited by its inability to match sequences of statements. (For example, we can implement loop fission by matching a single, top level for loop, but not loop fusion – we cannot match two loops in a sequence.) The basic algorithm to do so exists in the form of pseudocode, but needs to be implemented and verified to be correct. There are also various hazards, such as intervening statements inside a list of matched statements, that need to be handled sensibly. The syntax to handle sequences of statements (and potential hazards) has been partially developed, but also needs to be completed. Being able to match sequences of statements will unlock many new and interesting forms of transformations, including reordering statements, fusing statements (e.g., loop fusion), and other forms of refactoring.

9.2 New Features

This section will describe some of the features we wish to add to MARTINI.

9.2.1 Custom Directives. As discussed in Sec. 2.7.3, custom directives are one path to performance portability. Though the Clang compiler does not currently support adding custom/arbitrary directives to the AST so they could be matched, this is a feature that could be added in the future, and is one we are *very* interested in pursuing. As other tools that support custom directives have done, applications using MARTINI could have custom directives to give developers more control over where and when to apply certain transformations, and/or to provide MARTINI with parameters for transformations.

9.2.2 Control Structures. Some control structures have been added to MARTINI's syntax, such as checking whether a loop contains a certain code structure, but there is much more that can be done here. For example, what if the user only wishes to apply a transformation if some condition (e.g., more than 5

statements inside a loop body) is met? There is currently no way for them to do so. We would like to add more control structures to MARTINI, such as conditionals and extra parameters (e.g., number of times to unroll a loop).

9.2.3 Statistics Reporting. One common feature request we have seen for code analysis tools is the ability to report on various code statistics, such as the average depth of for loop nesting, conditionals inside of loops, and number of statements in functions. MARTINI's ability to translate an archetypal example of these statistics into an AST matcher makes it a clear choice for collecting these kinds of statistics, but currently it has no way to do so. We plan to implement a non-transformation attribute that will collect and report code statistics like these, so users can see exactly what their code looks like.

9.2.4 Transformation Order and Priority. Currently, MARTINI has no way to compose transformations, except by running the tool multiple times with different specification files. We wish to allow users to compose transformations in a single run, but this would require some notion of ordering between transformations. Syntax to provide MARTINI with ordering information has been developed, but needs to implemented.

We also need to implement a priority system for transformations, so that when a code snippet fits multiple matchers and replacers, but only *one* transformation should be applied, it is clear which one that is. This is a problem we ran into in Secs. 6.4 and 6.5, when one kernel needed special treatment but we had no way to distinguish it from the others, which had very similar structures.

9.3 Future Case Studies

There are many case studies we would like to perform with MARTINI. This section will present a selection of them.

9.3.1 Multiple Precision.

Many applications have begun adopting different floating point precision arithmetic in the hopes of seeing performance gains, particularly on GPUs, which often show much better performance on 32-bit and 16-bit floating point representations, compared to the 64-bit representations common in scientific applications. However, lowering the precision calculations are done in can have ramifications for correctness, which developers would like to avoid. MARTINI could allow users to test using multiple precision both by helping them programmatically change the types in their code and by inserting correctness checks for crucial variables. With MARTINI, the original code could be preserved so any changes could easily be removed. Alternatively, if changes prove beneficial, they could easily be applied throughout the application with minimal extra work.

9.3.2 Reducing Floating Point Errors.

Sometimes applications will encounter errors in floating point arithmetic, simply due to the fact that not all numbers can be represented by existing floating point formats. For example, if a very large number is added to many very small numbers, the small numbers may just disappear. However, if the ordering of the sum is changed, and the small numbers are added together first, that partial sum may be large enough to not disappear when added to the large number. There exist algorithms to minimize the chance of such errors, and some applications have taken to profiling the values of variables in their arithmetic so that said arithmetic can be reordered to reduce errors (Job et al. (2020)). Modifying codes to introduce these algorithms is non-trivial, though highly programmatic, so MARTINI could easily automate it. These algorithms also can obscure the true purpose of a line of code, so it may be best to keep them out of the main source code and as a transformation to be applied just before compilation. Also, in cases where the optimal ordering of some arithmetic is

not constant, MARTINI could insert different variants of the ordering, depending on the magnitudes of the variables involved.

9.3.3 More Porting. Though Chapter VI discussed several efforts to port between different programming models, there are still several such porting efforts we would like to investigate. In particular, porting between OpenACC and OpenMP is of interest, given the models are so similar. Porting between CUDA or OpenCL and a higher level model, such as Kokkos or SYCL, is also of interest, given the models are so *dis*similar, and moving from a non-portable model to a portable model is the goal of many application teams.

9.4 Future Integrations

This section will discuss other applications we would like to integrate MARTINI into, or would like to integrate into MARTINI.

9.4.1 Build Systems. Ideally, MARTINI would become part of application build systems, applying transformations as a last step before compilation. These transformations could be anything, including per-platform performance optimizations, instrumentation, debugging statements, algorithmic variations, or parallelism. All of these transformations, however, can obfuscate and clutter user code, so they are all better kept *not* as part of the main codebase, but separately, to be applied only when needed. Having MARTINI as part of their build system would help user productivity by allowing them to write cleaner, better, and faster code.

9.4.2 MLIR. The MLIR project (Lattner et al. (2020)) aims to provide a single, unified compiler IR ecosystem, along with solving many problems facing the compiler community, such as the difficulty of writing high-quality DSL compilers and many mainstream compilers defining their own IRs on top

of LLVM. Instead of implementing many better compilers, it seems easier to instead implement a better compiler building infrastructure.[1] MLIR is designed to standardize SSA IRs and includes built in support for documentation generation, debugging infrastructure, and parsing logic, among other things. MLIR users can define their own operations, types, rewrite patterns (similar to Haskell's rewrite rules), and optimization passes on top of MLIR's infrastructure, while keeping their IR extensible for the future.

MARTINI's methods are not exclusive to the Clang AST, though they are greatly helped along by Clang's AST matchers. Porting MARTINI to MLIR, with its better-defined IR structures, relations, and dialects, would allow for even more powerful transformations, with even more languages available as front ends.

9.4.2.1 Flang. One of the most common questions we hear is, "can it work with Fortran?" Once Flang, the new LLVM- and MLIR-based Fortran compiler's tooling infrastructure matures, we don't see why not. All we require to port MARTINI is a parser and something resembling Clang's AST matcher interface. Once Flang implements AST matchers, we would very much like to add Fortran support to MARTINI.

[1]The authors of Tapir (Stelle, Moses, Olivier, and McCormick (2017)) agree that it would be helpful to move the HPC community towards a single, standardized IR. By having standard, common IRs, we avoid duplicating effort and can have multiple programming models work together easily.

REFERENCES CITED

Amini, M., Creusillet, B., Even, S., Keryell, R., Goubier, O., Guelton, S., ... Villalon, P. (2012, January). Par4All: From convex array regions to heterogeneous computing. In *IMPACT 2012: Second International Workshop on Polyhedral Compilation Techniques HiPEAC 2012.* Paris, France. Retrieved from `https://hal-mines-paristech.archives-ouvertes.fr/hal-00744733` (2 pages)

Aurora exascale supercomputer. (2023). Argonne National Laboratory. Retrieved from `https://www.anl.gov/aurora`

Bailey, D. (1991, August). Twelve ways to fool the masses when giving performance results on parallel computers. *Supercomputing Review*, 54–55.

Bell, N., & Hoberock, J. (2012). Thrust: A productivity-oriented library for CUDA. In *GPU computing gems Jade edition* (pp. 359–371). Elsevier.

Bravenboer, M., Kalleberg, K. T., Vermaas, R., & Visser, E. (2008). Stratego/XT 0.17. a language and toolset for program transformation. *Science of Computer Programming*, *72*(1), 52-70. Retrieved from `https://www.sciencedirect.com/science/article/pii/S0167642308000452` (Special Issue on Second issue of experimental software and toolkits (EST)) doi: https://doi.org/10.1016/j.scico.2007.11.003

Breuer, S., Steuwer, M., & Gorlatch, S. (2014). Extending the SkelCL skeleton library for stencil computations on multi-GPU systems. In *Proceedings of the 1st international workshop on high-performance stencil computations* (pp. 15–21).

Brown, C., Abdelfattah, A., Tomov, S., & Dongarra, J. J. (2020). Design, optimization, and benchmarking of dense linear algebra algorithms on AMD GPUs. In *High performance extreme computing conference, HPEC.* doi: 10.1109/HPEC43674.2020.9286214

Chamberlain, B. L., Callahan, D., & Zima, H. P. (2007). Parallel programmability and the Chapel language. *The International Journal of High Performance Computing Applications*, *21*(3), 291–312.

Chen, C., Chame, J., & Hall, M. (2008). *CHiLL: A framework for composing high-level loop transformations* (Tech. Rep.). Citeseer.

Clang Developers. (n.d.). *clang::tooling::Transformer class reference.* Retrieved from `https://clang.llvm.org/doxygen/classclang_1_1tooling_1_1Transformer.html` (`https://clang.llvm.org/doxygen/classclang_1_1tooling_1_1Transformer.html`. Accessed March 8 2023)

Clement, V., Ferrachat, S., Fuhrer, O., Lapillonne, X., Osuna, C. E., Pincus, R., ... Sawyer, W. (2018, July). The CLAW DSL: Abstractions for performance portable weather and climate models. In *Proceedings of the platform for advanced scientific computing conference* (pp. 2:1–2:10). New York, NY, USA: ACM. Retrieved from `http://doi.acm.org/10.1145/3218176.3218226` doi: 10.1145/3218176.3218226

Coti, C., Denny, J. E., Huck, K., Lee, S., Malony, A. D., Shende, S., & Vetter, J. S. (2020). OpenACC profiling support for Clang and LLVM using Clacc and TAU. In *2020 IEEE/ACM International Workshop on HPC User Support Tools (HUST) and Workshop on Programming and Performance Visualization Tools (ProTools)* (p. 38-48). doi: 10.1109/HUSTProtools51951.2020.00012

Coti, C., Falcou, J., & Matei, B. (2020). *High-performance implementation of wavelet transforms using SIMD.* SIAM PP 20 poster session.

Custers, P. (2012). Algorithmic species: Classifying program code for parallel computing. *Master's thesis, Eindhoven University of Technology*.

Daniel, D., & Panetta, J. (2019, November). On applying performance portability metrics. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)* (pp. 50–59).

Dave, C., Bae, H., Min, S.-J., Lee, S., Eigenmann, R., & Midkiff, S. (2009). Cetus: A source-to-source compiler infrastructure for multicores. *Computer*, *42*(12), 36-42. doi: 10.1109/MC.2009.385

de Supinski, B. R., Scogland, T. R. W., Duran, A., Klemm, M., Bellido, S. M., Olivier, S. L., ... Mattson, T. G. (2018, November). The ongoing evolution of OpenMP. *Proceedings of the IEEE*, *106*(11), 2004-2019. doi: 10.1109/JPROC.2018.2853600

Deakin, T., McIntosh-Smith, S., Price, J., Poenaru, A., Atkinson, P., Popa, C., & Salmon, J. (2019, November). Performance portability across diverse computer architectures. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)* (pp. 1–13).

Deakin, T., Poenaru, A., Lin, T., & Mcintosh-Smith, S. (2020, November). Tracking performance portability on the yellow brick road to exascale. In *Proceedings of P3HPC 2020* (pp. 1–13). United States: Institute of Electrical and Electronics Engineers (IEEE). doi: 10.1109/P3HPC51967.2020.00006

Denny, J. E., Lee, S., & Vetter, J. S. (2018, November). CLACC: Translating OpenACC to OpenMP in Clang. In *2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)* (pp. 18–29). doi: 10.1109/LLVM-HPC.2018.8639349

Dolbeau, R., Bihan, S., & Bodin, F. (2007). HMPP: A hybrid multi-core parallel programming environment. In *Workshop on general purpose processing on graphics processing units (GPGPU 2007)* (Vol. 28).

Dreuning, H., Heirman, R., & Varbanescu, A. L. (2018). A beginner's guide to estimating and improving performance portability. In R. Yokota, M. Weiland, J. Shalf, & S. Alam (Eds.), *High performance computing* (pp. 724–742). Cham: Springer International Publishing.

Dufek, A. S., Gayatri, R., Mehta, N. A., Doerfler, D., Cook, B., Ghadar, Y., & DeTar, C. (2021). Case study of using Kokkos and SYCL as performance-portable frameworks for Milc-Dslash benchmark on NVIDIA, AMD and Intel GPUs. In *Workshop on Performance, Portability and Productivity in HPC, P3HPC@SC.* doi: 10.1109/P3HPC54578.2021.00009

Edwards, H. C., & Sunderland, D. (2012, February). Kokkos array performance-portable manycore programming model. In *Proceedings of the 2012 international workshop on programming models and applications for multicores and manycores* (pp. 1–10). New York, NY, USA: ACM.

Edwards, H. C., Sunderland, D., Amsler, C., & Mish, S. (2011, Sep.). Multicore/GPGPU portable computational kernels via multidimensional arrays. In *2011 IEEE International Conference on Cluster Computing* (p. 363-370). doi: 10.1109/CLUSTER.2011.47

Edwards, H. C., Trott, C. R., & Sunderland, D. (2014). Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, *74*(12), 3202 – 3216. (Domain-Specific Languages and High-Level Frameworks for High-Performance Computing)

Ernsting, S., & Kuchen, H. (2014, December). A scalable farm skeleton for hybrid parallel and distributed programming. *International Journal of Parallel Programming*, *42*(6), 968-987.

Free Software Foundation, Inc. (n.d.). *Program instrumentation options.* Retrieved from https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html#index-finstrument-functions

Frontier supercomputer debuts as world's fastest, breaking exascale barrier. (2022, May). Oak Ridge National Laboratory. Retrieved from https://www.ornl.gov/news/frontier-supercomputer-debuts-worlds-fastest-breaking-exascale-barrier

GHC Team. (n.d.). *The glorious Glasgow Haskell compilation system user's guide, version 7.0.1.* Retrieved from "https://downloads.haskell.org/~ghc/7.0.1/docs/html/users_guide/rewrite-rules.html" (https://downloads.haskell.org/~ghc/7.0.1/docs/html/users_guide/rewrite-rules.html. Accessed March 8, 2023)

The GNU awk user's guide. (n.d.). Retrieved from "https://www.gnu.org/software/gawk/manual/gawk.html" (https://www.gnu.org/software/gawk/manual/gawk.html. Accessed March 7, 2023)

Green500 list. (2023, June). Top500. Retrieved from https://www.top500.org/lists/green500/2023/06/

Griebl, M., Lengauer, C., & Wetzel, S. (1998). Code generation in the polytope model. In *In ieee pact* (pp. 106–111). IEEE Computer Society Press.

Grosser, T., Groesslinger, A., & Lengauer, C. (2012). Polly — performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, *22*(04). Retrieved from https://doi.org/10.1142/S0129626412500107 doi: 10.1142/S0129626412500107

Gustafson, J. (1991, June). *Twelve ways to fool the masses when giving performance results on traditional vector computers.*

Han, T. D., & Abdelrahman, T. S. (2009). hiCUDA: A high-level directive-based language for GPU programming. In *Proceedings of 2nd workshop on general purpose processing on graphics processing units* (pp. 52–61). New York, NY, USA: ACM.

Harrell, S. L., Kitson, J., Bird, R., Pennycook, S. J., Sewall, J., Jacobsen, D., ... Robey, R. (2018, November). Effective performance portability. In *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)* (p. 24-36). doi: 10.1109/P3HPC.2018.00006

Harris, D. (2021, May). *Need for speed: Researchers switch on world's fastest ai supercomputer.* Nvidia. Retrieved from `https://blogs.nvidia.com/blog/2021/05/27/nersc-perlmutter-ai-supercomputer/`

Hartono, A., Norris, B., & Sadayappan, P. (2009). Annotation-based empirical performance tuning using Orio. In *2009 IEEE International Symposium on Parallel & Distributed Processing* (p. 1-11). doi: 10.1109/IPDPS.2009.5161004

Hennessy, J., & Patterson, D. (2019). *Computer architecture: A quantitative approach* (6th ed.). Elsevier Science.

Hollman, D., Lelbach, B., Edwards, H. C., Hoemmen, M., Sunderland, D., & Trott, C. (2019, November). mdspan in C++: A case study in the integration of performance portable features into international language standards. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)* (pp. 60–70).

Holmen, J., Peterson, B., & Berzins, M. (2019, November). An approach for indirectly adopting a performance portability layer in large legacy codes. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)* (pp. 36–49).

Hornung, R., & Keasler, J. (2013). *A case for improved C++ compiler support to enable performance portability in large physics simulation codes* (Tech. Rep.). Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States).

Huck, K., Coti, C., Johnson, A., & Malony, A. D. (n.d.). *Source alteration for profiling.* (Work in progress.)

Huck, K., Shende, S., Malony, A., Coti, C., Spear, W., Alcaraz, J., ... Beekman, I. (2024). Preparing the TAU performance system for exascale and beyond. *International Journal on High Performance Computing Applications*. (In submission)

Intel Corp. (2020, May). *Intel oneAPI programming guide (beta).* Retrieved from `https://software.intel.com/content/www/us/en/develop/documentation/oneapi-programming-guide/top.html`

Job, V., Grové, T., Fogerty, S., Mauney, C., Neuman, B., Monroe, L., & Robey, R. W. (2020). Order matters: A case study on reducing floating point error in sums via ordering and grouping. In *2020 IEEE/ACM 4th International Workshop on Software Correctness for HPC Applications (Correctness)* (p. 10-19). doi: 10.1109/Correctness51934.2020.00007

Johnson, A. (2020). Area exam: General-purpose performance portable programming models for productive exascale computing. *University of Oregon, Eugene, OR, USA. Area Exam Report*.

Johnson, A., Coti, C., Malony, A. D., & Doerfert, J. (2022). Martini: The little match and replace tool for automatic application rewriting with code examples. In J. Cano & P. Trinder (Eds.), *Euro-par 2022: Parallel processing* (pp. 19–34). Cham: Springer International Publishing.

Johnson, A., Coti, C., Malony, A. D., & Hueckelheim, J. (n.d.). *User-assisted automatic code translation with MARTINI.* (To be submitted to EuroPar '24.)

Joubert, W., Archibald, R., Berrill, M., Michael Brown, W., Eisenbach, M., Grout, R., ... Turner, J. (2015, August). Accelerated application development: The ORNL Titan experience. *Comput. Electr. Eng.*, *46*(C), 123–138.

Kats, L. C., & Visser, E. (2010). The Spoofax language workbench: Rules for declarative specification of languages and IDEs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (p. 444–463). New York, NY, USA: Association for Computing Machinery. Retrieved from `https://doi.org/10.1145/1869459.1869497` doi: 10.1145/1869459.1869497

Khronos OpenCL Working Group. (2023, April). *The OpenCL specification.* Khronos Group. Retrieved from `https://registry.khronos.org/OpenCL/specs/3.0-unified/html/OpenCL_API.html`

Kindratenko, V., & Trancoso, P. (2011, May). Trends in high-performance computing. *Computing in Science Engineering*, *13*(3), 92-95. doi: 10.1109/MCSE.2011.52

Kogge, P., & Shalf, J. (2013, November). Exascale computing trends: Adjusting to the "new normal" for computer architecture. *Computing in Science Engineering*, *15*(6), 16-26. doi: 10.1109/MCSE.2013.95

Komatsu, K., Egawa, R., Hirasawa, S., Takizawa, H., Itakura, K., & Kobayashi, H. (2016). Translation of large-scale simulation codes for an OpenACC platform using the Xevolver framework. *International Journal of Networking and Computing*, *6*(2), 167–180.

Lambert, J., Lee, S., Kim, J., Vetter, J. S., & Malony, A. D. (2018, June). Directive-based, high-level programming and optimizations for high-performance computing with FPGAs. In *Proceedings of the 2018 international conference on supercomputing* (pp. 160–171). New York, NY, USA: ACM.

Larkin, J. (2016, April). Performance portability through descriptive parallelism. *DoE CoE Performance Portability Workshop*.

Lattner, C., & Adve, V. (2004). LLVM: a compilation framework for lifelong program analysis and transformation. In *International symposium on code generation and optimization, 2004. cgo 2004.* (pp. 75–86).

Lattner, C., Pienaar, J., Amini, M., Bondhugula, U., Riddle, R., Cohen, A., ... Zinenko, O. (2020). MLIR: A compiler infrastructure for the end of Moore's law. *ArXiv*, *abs/2002.11054*.

Lee, S., & Eigenmann, R. (2010, November). OpenMPC: Extended OpenMP programming and tuning for GPUs. In *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis* (p. 1-11). doi: 10.1109/SC.2010.36

Lee, S., Min, S.-J., & Eigenmann, R. (2009, February). OpenMP to GPGPU: A compiler framework for automatic translation and optimization. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (pp. 101–110). New York, NY, USA: ACM.

Lee, S., & Vetter, J. S. (2014a, November). OpenARC: Extensible OpenACC compiler framework for directive-based accelerator programming study. In *2014 first workshop on accelerator programming using directives* (p. 1-11). doi: 10.1109/WACCPD.2014.7

Lee, S., & Vetter, J. S. (2014b, June). OpenARC: Open accelerator research compiler for directive-based, efficient heterogeneous computing. In *Proceedings of the 23rd international symposium on high-performance parallel and distributed computing* (pp. 115–120). New York, NY, USA: ACM.

Lee, S.-I., Johnson, T. A., & Eigenmann, R. (2003). Cetus-an extensible compiler infrastructure for source-to-source transformation. In *LCPC* (pp. 539–553).

Leung, A., Vasilache, N., Meister, B., Baskaran, M., Wohlford, D., Bastoul, C., & Lethin, R. (2010, March). A mapping path for multi-GPGPU accelerated computers from a portable high level programming abstraction. In *Proceedings of the 3rd workshop on general-purpose computation on graphics processing units* (pp. 51–61). New York, NY, USA: ACM.

Liao, C., Lin, P.-H., Schordan, M., & Karlin, I. (2018). A semantics-driven approach to improving DataRaceBench's OpenMP standard coverage. In B. R. de Supinski, P. Valero-Lara, X. Martorell, S. Mateo Bellido, & J. Labarta (Eds.), *Evolving OpenMP for Evolving Architectures* (pp. 189–202). Cham: Springer International Publishing.

Lidman, J., Quinlan, D. J., Liao, C., & McKee, S. A. (2012). ROSE::FTTransform – a source-to-source translation framework for exascale fault-tolerance research. In *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN 2012)* (p. 1-6).

Lindlan, K., Cuny, J., Malony, A., Shende, S., Mohr, B., Rivenburgh, R., & Rasmussen, C. (2000). A tool framework for static and dynamic analysis of object-oriented software with templates. In *SC '00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing* (p. 49-49). doi: 10.1109/SC.2000.10052

The LINPACK benchmark. (2023). Top500. Retrieved from `https://www.top500.org/project/linpack/`

The LLVM compiler infrastructure. (n.d.). Retrieved from `"https://llvm.org/"` (`https://llvm.org/`. Accessed March 8, 2023)

Lucas, R., Ang, J., Bergman, K., Borkar, S., Carlson, W., Carrington, L., ... Laros III, J. (2014, February). *DOE Advanced Scientific Computing Advisory Subcommittee (ASCAC) report: Top ten exascale research challenges* (Tech. Rep.). US Department of Energy. doi: 10.2172/1222713

Luk, C., Hong, S., & Kim, H. (2009, December). Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (p. 45-55).

Majeed, M., Dastgeer, U., & Kessler, C. W. (2013). Cluster-SkePU: A multi-backend skeleton programming library for GPU clusters. In *PDPTA 2013.*

Martineau, M., & McIntosh-Smith, S. (2017). The productivity, portability and performance of OpenMP 4.5 for scientific applications targeting Intel CPUs, IBM CPUs, and NVIDIA GPUs. In B. R. de Supinski, S. L. Olivier, C. Terboven, B. M. Chapman, & M. S. Müller (Eds.), *Scaling OpenMP for Exascale Performance and Portability (IWOMP '17)* (pp. 185–200). Cham: Springer International Publishing.

Martone, M., & Lawall, J. (2021). Refactoring for performance with semantic patching: Case study with recipes. In H. Jagode, H. Anzt, H. Ltaief, & P. Luszczek (Eds.), *High performance computing* (pp. 226–232). Cham: Springer International Publishing.

McCabe, T. (1976). A complexity measure. *Transactions on Software Engineering*, *SE-2*(4). doi: 10.1109/TSE.1976.233837

McCalpin, J. (n.d.). *STREAM: Sustainable memory bandwidth in high performance computers.* Retrieved from `https://www.cs.virginia.edu/stream/`

McIntosh-Smith, S., Boulton, M., Curran, D., & Price, J. (2014). On the performance portability of structured grid codes on many-core computer architectures. In J. M. Kunkel, T. Ludwig, & H. W. Meuer (Eds.), *Supercomputing* (pp. 53–75). Cham: Springer International Publishing.

McIntosh-Smith, S., Martineau, M., Deakin, T., Pawelczak, G., Gaudin, W., Garrett, P., ... Beckingsale, D. (2017). TeaLeaf: A mini-application to enable design-space explorations for iterative sparse linear solvers. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)* (p. 842-849). doi: 10.1109/CLUSTER.2017.105

Meister, B., Leung, A., Vasilache, N., Wohlford, D., Bastoul, C., & Lethin, R. (2009). Productivity via automatic code generation for PGAS platforms with the R-Stream compiler. In *Workshop on Asynchrony in the PGAS Programming Model.*

Mendonça, G., Guimarães, B., Alves, P., Pereira, M., Araújo, G., & Pereira, F. M. Q. a. (2017, May). DawnCC: Automatic annotation for data parallelism and offloading. *ACM Trans. Archit. Code Optim.*, *14*(2). Retrieved from `https://doi.org/10.1145/3084540` doi: 10.1145/3084540

Meng, N., Kim, M., & McKinley, K. S. (2011). Sydit: Creating and applying a program transformation from an example. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (p. 440–443). New York, NY, USA: Association for Computing Machinery. Retrieved from `https://doi.org/10.1145/2025113.2025185` doi: 10.1145/2025113.2025185

Meng, N., Kim, M., & McKinley, K. S. (2013). Lase: Locating and applying systematic edits by learning from examples. In *2013 35th International Conference on Software Engineering (ICSE)* (p. 502-511). doi: 10.1109/ICSE.2013.6606596

Michalakes, J., Loft, R., & Bourgeois, A. (2001). *Performance-portability and the weather research and forecast model.*

Mo, Z.-y. (2018, October). Extreme-scale parallel computing: bottlenecks and strategies. *Frontiers of Information Technology & Electronic Engineering*, *19*(10), 1251–1260.

MPI Forum. (2015, June). *MPI: A message-passing interface standard.* MPI Forum.

Murai, H., Sato, M., Nakao, M., & Lee, J. (2018, November). Metaprogramming framework for existing HPC languages based on the Omni compiler infrastructure. In *2018 Sixth International Symposium on Computing and Networking Workshops (CANDARW)* (pp. 250–256). doi: 10.1109/CANDARW.2018.00054

Nakao, M., Lee, J., Boku, T., & Sato, M. (2012, May). Productivity and performance of global-view programming with XcalableMP PGAS language. In *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)* (p. 402-409). doi: 10.1109/CCGrid.2012.118

Nakao, M., Murai, H., Shimosaka, T., Tabuchi, A., Hanawa, T., Kodama, Y., ... Sato, M. (2014, November). XcalableACC: Extension of XcalableMP PGAS language using OpenACC for accelerator clusters. In *2014 first workshop on accelerator programming using directives* (p. 27-36). doi: 10.1109/WACCPD.2014.6

Nugteren, C., & Corporaal, H. (2014, December). Bones: An automatic skeleton-based C-to-CUDA compiler for GPUs. *ACM Trans. Archit. Code Optim.*, *11*(4), 35:1–35:25.

Nugteren, C., Corvino, R., & Corporaal, H. (2013). Algorithmic species revisited: A program code classification based on array references. In *2013 IEEE 6th International Workshop on Multi-/Many-core Computing Systems (MuCoCoS)* (p. 1-8). doi: 10.1109/MuCoCoS.2013.6633604

Nugteren, C., Custers, P., & Corporaal, H. (2013). Automatic skeleton-based compilation through integration with an algorithm classification. In C. Wu & A. Cohen (Eds.), *Advanced parallel processing technologies* (pp. 184–198). Berlin, Heidelberg: Springer Berlin Heidelberg.

Nvidia. (2023, October). *CUDA Toolkit documentation.* Nvidia. Retrieved from `https://docs.nvidia.com/cuda/`

Ofenbeck, G., Steinmann, R., Caparros, V., Spampinato, D. G., & Püschel, M. (2014, March). Applying the roofline model. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)* (p. 76-85). doi: 10.1109/ISPASS.2014.6844463

OpenACC Group. (n.d.). *OpenACC.* Retrieved from `https://www.openacc.org/`

OpenACC Standards Group. (2011, November). *The OpenACC application program interface, version 1.0.* OpenACC Group.

OpenACC Standards Group. (2018, November). *The OpenACC application program interface, version 2.7.* OpenACC Group.

OpenACC Standards Group. (2019, November). *The OpenACC application program interface, version 3.0.* OpenACC Group.

OpenMP Architecture Review Board. (2011, July). *OpenMP application program interface, version 3.1.* OpenMP ARB.

OpenMP Architecture Review Board. (2015, November). *OpenMP application program interface, version 4.5.* OpenMP ARB.

OpenMP Architecture Review Board. (2018, November). *OpenMP application program interface, version 5.0.* OpenMP ARB.

OpenSHMEM Contributors Committee. (2017, December). *OpenSHMEM application programming interface.* Open Source Software Solutions, Inc.

Padioleau, Y., Lawall, J., Hansen, R. R., & Muller, G. (2008). Documenting and automating collateral evolutions in Linux device drivers. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008* (p. 247–260). New York, NY, USA: Association for Computing Machinery. Retrieved from `https://doi.org/10.1145/1352592.1352618` doi: 10.1145/1352592.1352618

Pakin, S. (2011, December). *Ten ways to fool the masses when giving performance results on GPUs.* HPCWire.

Peccerillo, B., & Bartolini, S. (2017). PHAST library — enabling single-source and high performance code for GPUs and multi-cores. In *2017 International Conference on High Performance Computing Simulation (HPCS)* (p. 715-718).

Pennycook, S. J., Sewall, J. D., & Duran, A. (2018). Supporting function variants in OpenMP. In B. R. de Supinski, P. Valero-Lara, X. Martorell, S. Mateo Bellido, & J. Labarta (Eds.), *Evolving OpenMP for Evolving Architectures* (pp. 128–142). Cham: Springer International Publishing.

Pennycook, S. J., Sewall, J. D., & Hammond, J. R. (2018, November). Evaluating the impact of proposed OpenMP 5.0 features on performance, portability and productivity. In *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)* (pp. 37–46). doi: 10.1109/P3HPC.2018.00007

Pennycook, S. J., Sewall, J. D., & Lee, V. W. (2016). A metric for performance portability. In *Proceedings of the international workshop on performance modeling, benchmarking, and simulation.*

Pennycook, S. J., Sewall, J. D., & Lee, V. W. (2019). Implications of a metric for performance portability. *Future Generation Computer Systems*, *92*, 947 – 958.

Pino, S., Pollock, L., & Chandrasekaran, S. (2017, May). Exploring translation of OpenMP to OpenACC 2.5: lessons learned. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (p. 673-682). doi: 10.1109/IPDPSW.2017.84

Quinlan, D., & Liao, C. (2011). The ROSE source-to-source compiler infrastructure. In *Cetus users and compiler infrastructure workshop, in conjunction with PACT.*

Reinders, J., Ashbaugh, B., Brodman, J., Kinsner, M., Pennycook, J., & Tian, X. (2019). *Data Parallel C++: Mastering DPC++ for programming of heterogeneous systems using C++ and SYCL.* Apress. (Unedited advance preview of chapters 1-4)

ROCm Developers. (n.d.). *HIPIFY: Tools to translate CUDA source code into portable HIP C++ automatically.* Retrieved from `https://github.com/ROCm-Developer-Tools/HIPIFY`

Savchenko, V., Sorokin, K., Pankratenko, G., Markov, S., Spiridonov, A., Alexandrov, I., ... Sun, K. (2019). Nobrainer: An example-driven framework for C/C++ code transformations. In N. Bjørner, I. Virbitskaite, & A. Voronkov (Eds.), *Perspectives of system informatics.* Springer International Publishing. doi: 10.1007/978-3-030-37487-7_12

Savchenko, V. V., Sorokin, K. S., Bronshtein, I. E., Volkov, A. S., Kachanov, V. V., Pankratenko, G. A., ... Aleksandrov, I. V. (2020, 01). NOBRAINER: A tool for example-based transformation of C/C++ code. *Programming and Computer Software*, *46*(5). doi: 10.1134/S0361768820040052

Schweitz, E., Lethin, R., Leung, A., & Meister, B. (2006). R-stream: A parametric high level compiler. *Proceedings of HPEC*.

sed, a stream editor. (n.d.). Retrieved from "https://www.gnu.org/software/sed/manual/sed.html" (https://www.gnu.org/software/sed/manual/sed.html. Accessed March 7, 2023)

Sedova, A., Eblen, J. D., Budiardja, R., Tharrington, A., & Smith, J. C. (2018, November). High-performance molecular dynamics simulation for biological and materials sciences: Challenges of performance portability. In *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)* (p. 1-13). doi: 10.1109/P3HPC.2018.00004

Shalf, J., Dosanjh, S., & Morrison, J. (2011). Exascale computing technology challenges. In J. M. L. M. Palma, M. Daydé, O. Marques, & J. C. Lopes (Eds.), *High Performance Computing for Computational Science – VECPAR 2010* (pp. 1–25). Berlin, Heidelberg: Springer Berlin Heidelberg.

Shende, S. S., & Malony, A. D. (2006). The Tau parallel performance system. *The International Journal of High Performance Computing Applications*, *20*(2), 287-311. Retrieved from https://doi.org/10.1177/1094342006064482 doi: 10.1177/1094342006064482

Smith, J. E. (1988). Characterizing computer performance with a single number. *Communications of the ACM*, *31*(10), 1202–1206.

Stelle, G., Moses, W. S., Olivier, S. L., & McCormick, P. (2017, November). OpenMPIR: Implementing OpenMP tasks with Tapir. In *Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC* (pp. 3:1–3:12). New York, NY, USA: ACM.

Steuwer, M., & Gorlatch, S. (2013). SkelCL: Enhancing OpenCL for high-level programming of multi-GPU systems. In V. Malyshkin (Ed.), *Parallel computing technologies* (pp. 258–272). Berlin, Heidelberg: Springer Berlin Heidelberg.

Steuwer, M., Kegel, P., & Gorlatch, S. (2011, May). SkelCL - a portable skeleton library for high-level GPU programming. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum* (p. 1176-1182). doi: 10.1109/IPDPS.2011.269

Steuwer, M., Kegel, P., & Gorlatch, S. (2012, May). Towards high-level programming of multi-GPU systems using the SkelCL library. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum* (pp. 1858–1865).

Suda, R., Takizawa, H., & Hirasawa, S. (2016). Xevtgen: Fortran code transformer generator for high performance scientific codes. *International Journal of Networking and Computing*, *6*(2), 263-289.

Sultana, N., Calvert, A., Overbey, J. L., & Arnold, G. (2016, July). From OpenACC to OpenMP 4: Toward automatic translation. In *Proceedings of the xsede16 conference on diversity, big data, and science at scale* (pp. 44:1–44:8). New York, NY, USA: ACM.

Sun, Y., Mukherjee, S., Baruah, T., Dong, S., Gutierrez, J., Mohan, P., & Kaeli, D. R. (2018). Evaluating performance tradeoffs on the Radeon Open Compute platform. In *International symposium on performance analysis of systems and software, ISPASS 2018.* doi: 10.1109/ISPASS.2018.00034

Tabuchi, A., Nakao, M., Murai, H., Boku, T., & Sato, M. (2017, May). Implementation and evaluation of one-sided PGAS communication in XcalableACC for accelerated clusters. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)* (p. 625-634). doi: 10.1109/CCGRID.2017.81

Takizawa, H., Hirasawa, S., Hayashi, Y., Egawa, R., & Kobayashi, H. (2014, December). Xevolver: An XML-based code translation framework for supporting HPC application migration. In *2014 21st International Conference on High Performance Computing (HiPC)* (p. 1-11). doi: 10.1109/HiPC.2014.7116902

Thakur, R., Balaji, P., Buntinas, D., Goodell, D., Gropp, W., Hoefler, T., ... Larsson Träff, J. (2010, January). MPI at exascale. *Procceedings of SciDAC 2010*, *2*.

Top500 list. (2023, June). Top500. Retrieved from `https://www.top500.org/lists/top500/2023/06/`

Verdoolaege, S., Juega, J. C., Cohen, A., Gómez, J. I., Tenllado, C., & Catthoor, F. (2013, January). Polyhedral parallel code generation for CUDA. *ACM Trans. Archit. Code Optim.*, *9*(4), 54:1–54:23. doi: 10.1145/2400682.2400713

Wienke, S., Miller, J., Schulz, M., & Müller, M. S. (2016, November). Development effort estimation in HPC. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (p. 107-118). doi: 10.1109/SC.2016.9

Williams, S., Waterman, A., & Patterson, D. (2009, April). Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, *52*(4), 65–76.

Wolfe, M. (2010, March). Implementing the PGI Accelerator model. In *Proceedings of the 3rd workshop on general-purpose computation on graphics processing units* (pp. 43–50). New York, NY, USA: ACM.

Wolfe, M. (2011, April). *Compilers and more: Exascale programming requirements.* HPCwire.

Wolfe, M. (2016a, June). *Compilers and more: OpenACC to OpenMP (and back again).* HPCwire.

Wolfe, M. (2016b, April). *Compilers and more: What makes performance portable?* HPCwire.

Wolfe, M., Lee, S., Kim, J., Tian, X., Xu, R., Chandrasekaran, S., & Chapman, B. (2017, May). Implementing the OpenACC data model. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (pp. 662–672). doi: 10.1109/IPDPSW.2017.85

Wong, M., Richards, A., Rovatsou, M., & Reyes, R. (2016, February). *Khronos's opencl sycl to support heterogeneous devices for c++.*

Wright, H. K., Jasper, D., Klimek, M., Carruth, C., & Wan, Z. (2013). Large-scale automated refactoring using ClangMR. In *International conference on software maintenance.* doi: 10.1109/ICSM.2013.93

Yang, C., Gayatri, R., Kurth, T., Basu, P., Ronaghi, Z., Adetokunbo, A., ... Williams, S. (2018, November). An empirical roofline methodology for quantitatively assessing performance portability. In *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)* (p. 14-23). doi: 10.1109/P3HPC.2018.00005

Yang, G.-W., & Fu, H.-H. (2018, October). Application software beyond exascale: challenges and possible trends. *Frontiers of Information Technology & Electronic Engineering*, *19*(10), 1267–1272.

Zhu, W., Niu, Y., & Gao, G. R. (2007, June 01). Performance portability on EARTH: a case study across several parallel architectures. *Cluster Computing*, *10*(2), 115–126.