Supercharging Programming through Compiler Technology

by

William S. Moses

M.Eng., Massachusetts Institute of Technology (2017)
S.B., Massachusetts Institute of Technology (2017)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2023

Authored by: William S. Moses
Department of Electrical Engineering and Computer Science
May 19, 2023

Certified by: Charles E. Leiserson
Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by: Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Supercharging Programming through Compiler Technology

by

William S. Moses

Submitted to the Department of Electrical Engineering and Computer Science
on May 19, 2023, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

The decline of Moore's law and an increasing reliance on computation has led to an explosion of specialized software packages and hardware architectures. While this diversity enables unprecedented flexibility, it also requires domain-experts to learn how to customize programs to efficiently leverage the latest platform-specific API's and data structures, instead of working on their intended problem. For example, a researcher hoping to use machine learning on climate code must write a corresponding derivative simulation, understand and implement linear algebra routines, and performance engineer their simulation to run on multiple cores and nodes. Rather than forcing each user to bear this burden, I propose building high-level abstractions within general-purpose compilers that enable fast, portable, and composable programs to be automatically generated.

This thesis will demonstrate this approach through several real-world and composable compilers that I built for a variety of domains including parallelism, automatic differentiation, scheduling, portability, program search, and tensor arithmetic. These domains are critical to both scientific computing and machine learning. Individually, integration of domain knowledge into each of these compilers enable (often asymptotic) performance and usability benefits. Operating on a common compiler representation, however, enables these benefits to compound and provide greater performance than any domain-specific optimization in isolation.

This research in this thesis contains joint work with Charles E. Leiserson, Tao B. Schardl, Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, Zachary, Sven Verdoolaege, Andrew Adams, Albert Cohen, Qijing (Jenny) Huang, Ameer Haj-Ali, John Xiang, Ion Stoica, Krste Asanovic, John Wawrzynek, Valentin Churavy, Lorenzo Chelini, Ruizhe Zhao, Ludger Paehler, Jan Hückelheim, Sri Hari Krishna Narayanan, Michel Schanen, Johannes Doerfert, Paul Hovland, Ivan R. Ivanov, Jens Domke, and Toshio Endo.

Thesis Supervisor: Charles E. Leiserson
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

First and foremost I want to thank my family. I particularly want to thank my sister Sophia Moses, father John Moses, mother Marina Moses, and my pappou Panayoti Stefanidis. Your unwavering love and support have been instrumental to my time at MIT in undergraduate and graduate school.

I'd like to my advisor, Charles E. Leiserson, who has provided constant research and life advice since even before I came to MIT as an undergraduate. I'd like to thank the other members of my committee, Alan Edelman, Albert Cohen, and Saman Amarasinghe for providing advice not just on this thesis, but research and academia at large. I'd like to thank Srini Devadas for showing me how to be a good mentor – a role model I am to emulate as I enter academia. I'd like to thank Jason Ku for tireless work to make teaching fun.

Without a doubt, there are no colleagues of mine who have provided more support, advice (and fun conference excursions) than Valentin Churavy, Alex Zinenko, and Johannes Doerfert. I'd also like to thank my numerous research colleagues, without whom the work in this thesis would not be possible, including: Jed Brown, Hal Finkel, Marco Foco, Leila Gharaffi, Laurent Hascoet, Patrick Heimbach, Paul Hovland, Jan Hueckelheim, Mike Innes, Tim Kaler, Charles Leiserson, Yingbo Ma, Ludger Paehler, Chris Rackauckas, TB Schardl, Lizhou Sha, Yo Shavit, Vassil Vassilev, Sarah Williamson, Pat McCormick, George Stelle, Stephen Olivier, Joanna Balme, Eric Brown-Dymkosky, Victor Guerrero, Stephen Jones, Andre Kessler, Adam Lichtl, Kevin Lung, Ken Museth, Nathan Robertson, Youseef Marzouk, Jesse Michel, Kevin Kwok, Douglas Kogut, Jiahao Li, Bojan Serafimov, Carl Guo, Sanath Govindarajan, Walden Yan, Sage Simhon, Chuyang Chen, Shakil Ahmed, Abhishek Vu, Chris Hill, AJ Root, Teo Collins, Nicolas Vasilache, Zach Devito, Andrew Adams, Lorenzo Chelini, Ruiche Zhao, Sven VErdoolaege, Tim Gymnich, Pratush Das, Manuel Drewalrd, Theodoros Theodoridis, Priya Goyal, Ivan R. Ivanov, Jens Domke, Toshio Endo, John Wawrzynek, Krste Asanovic, Ion Stoica, Jenny Huang, Ameer Haj-Ali, Daniel Dunbar, Neil Thompson, Bill Kuszmaul, Charith Mendis, the Supertech research group, JuliaLab research group, COMMIT resaerch group, the FutureTech research group,

and more.

There are so many other friends and colleagues that I would like to thank, that I would quickly use up the remaining ink on the printer. I'd like to thank my lifelong friends I've made from primary school, Joey Cimento, Jeff Saulnier, Mike Timpane, and Joseph Orttung, who remind me to always have fun alongside work. I'd like to thank numerous friends from TJ including Andrew Coffee, Daniel Stiffler, Nalini Singh, Alex Atanasov, Matt Levonian, Will Qian Abi Gopal, Will Bradbury, James Bradbury, and Ryan Jian. I'd like to thank Mark Hannum, Charles Clancy, and John Dell for getting me into research. I'd like to thank my Maseeh pals: Arezu Esmaili, Andreea Martin, Alex Cabrales, Austin Clark, Brook Eyob, Elizabeth Bianchini, Henry Love, Juan Angulo, Mellisa Gianello, Rob Bugliarellia, Rodrigo Ruiz, Allison Tam, Ananya Nandy, Anna Bueno, Claudia Wu, Daphne Lin, Darius Bopp, Elena Albertia, Emily Sheng, Grace Yin, Jason Priest, Joseph Murphy, Kristen Fromback, Niyati Desai, Nola Mulugeta, Rogers Epstein, Ron Dentinger, Travis Hank, and Kevin Sabo. I'd like to thank Andrea Carney for making my time at Sidpac a constant joy. I'd like to thank many other folks including: Bob Knighton, Sofi Peterson, Emma Batson, Tasha Schoenstein, Burhan Azeem, Anne Hunter, Rachel (Cummings) Shavit, Tesla Wells, Crystal Wang, Erica Yuen, Predrag Gruveski Victoria Xia, Shayna Ahtek Botong Ma, Anish Athalye, Kelsey Becker, Lizhou Sha, Alex Chernyakovsky, Angel Alvarez, Alan Sadun, Jamie Voros, Logan Engstrom, Kevin Kwok, Cat Zeng, Zachary Pitcher, Tomas Villalon, Piper Lim, and Chris Peterson.

The research contained within was financially supported in part by numerous organizations. This includes a DOE Computational Sciences Graduate Fellowship DE-SC0019323; NSF Cyberinfrastructure for Sustained Scientific Innovation (CSSI) award numbers: 2104068, 2103942, and 2103804; the Defense Advanced Research Projects Agency (DARPA) under Agreement No. HR0011-20-9-0016; NSF Grant OAC-1835443; Los Alamos National Laboratories grant 531711. Research was sponsored in part by the United States Air Force Research Laboratory and was accomplished under Cooperative Agreement Number FA8750-19-2-1000. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the United States Air Force or the U.S. Government. The U.S. Government is

Contents

List of Figures

Chapter 1

Introduction

The end of Moore's law and the increasing reliance on computation has led to an explosion of complex software packages and hardware architectures in various domains. While this diversity enables an unprecedented level of flexibility in building applications, it also requires rewriting applications to efficiently support each combination of software paradigms (e.g. differentiable, encrypted) and hardware targets. The need for performance and portability has forced domain-experts to spend their time learning about and implementing concepts like GPU reductions or the TensorFlow API, rather than working on their intended problem.

Previous solutions have been proposed and include language extensions (e.g., OpenCL [102], OpenACC [169]), parallel programming frameworks (e.g., Kokkos [63]), and domain-specific languages (e.g., SPIRAL [314], Halide [319]). All of these approaches still require legacy applications to be ported, and sometimes entirely rewritten, due to differences in the language, or the underlying programming model.

I propose creating high-level abstractions and transformations once within a common multi-purpose compiler that can take existing code and generate optimized code for each software/hardware paradigm, rather than burdening the programmer. In contrast to the mythical "sufficiently smart compiler", my research focuses on explicit domains where preserving information across the library, language, and compiler boundary can be applied to real programs. This design is composable by definition and enables tool-builders to write platform-specific code once, rather than the user writing and maintaining

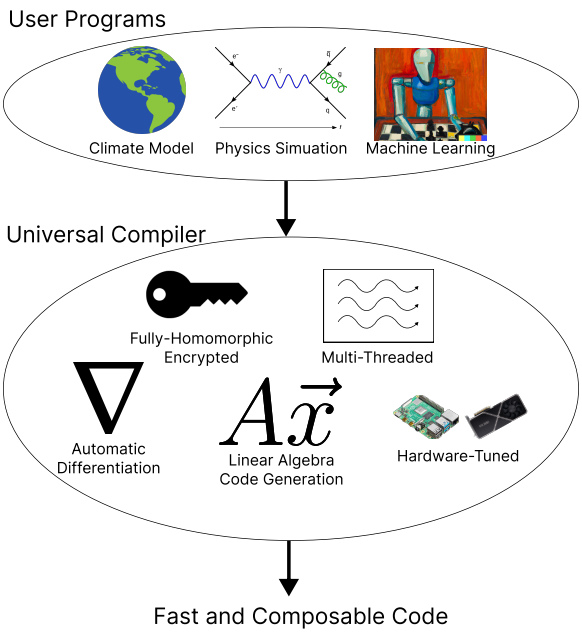


Figure 1-1: Building abstractions once within the compiler enables fast and composable code to be automatically generated for all user programs.

multiple versions. Further, bringing these semantics together in a single tool enables the compiler to automatically apply cross-package optimizations that would be difficult for even expert programmers to write by hand.

I demonstrate the feasibility of this approach by building several real-world and composable systems in several domains (described below). In addition to each of these compiler abstractions being individually beneficial for performance and usability, the fact that they co-exist in a common compiler framework means that they mutually benefit each other. As an example, both the Tapir representation of parallel programs (Chapter 2) and the Enzyme compiler for differentiating functions (Chapter 5) each individually provide performance benefits. Operating on a common compiler representation for both parallelism and differentiation, however, enables both seamless differentiation of parallel programs and composition of optimizations (Chapter 8). This combined and composable set enables greater performance of parallel differentiation than either parallel-specific optimizations, or differentiation-specific optimizations could provide individually.

Many chapters of this thesis will share variations of a common vector normalization example code (Figures 2-1, 4-1, 4-2, 4-3, 5-1, 9-1, 10-1, 10-2). Individually, each of these

will demonstrate how a system will run (often asymptotically) slower due to its lack of compiler-based domain knowledge. The fact that the same example has a significant performance gap in many different domains (differentiation, parallelism, accelerators, phase ordering, etc) indicates how critical compiler representations are – and also how tricky they are to get right.[1] Besides a stylistic choice to use a similar example throughout the thesis to ease comprehension, there is nothing fundamental about that particular example code that makes it benefit so extensively from a variety of domain optimizations. Instead, I argue that this example indicates that applications in general will tend to benefit from a variety of domain-specific optimizations.

Through the lens of compilers, this thesis will explore fundamental principles to building and efficiently executing modern scientific computing and machine learning applications. Among other topics, thesis will explore compiler representations and transformations for:

- *Parallelism:* leveraging multiple processing units, on the CPU (Chapters 2, 6, 9, 8), GPU/accelerators (Chapters 3, 9, 7), or distributed machines (Chapter 8);
- *Automatic differentiation:* computing the derivative of program functions (Chapters 5, 7, 8);
- *Scheduling:* finding the fastest versions of programs by optimizing for hardware specifics, like caches, bandwidth, or thread count (Chapters 3, 6);
- *Portability:* enabling programs to run efficiently on different hardware or systems (Chapters 3, 9);
- *Program search:* methods and bottlenecks for automatically finding faster versions of programs (Chapters 4, 10, 3); and
- *Tensor Arithmetic:* efficiently computing fast scalar, vector, matrix, and vector operations (Chapters 3, 6, 9).

[1] It is for this reason that the author lovingly refers to this thesis by a secondary title "10 ways to normalize a vector, number 7 will surprise you."

Chapter 2

Tapir: A compiler representation for fork-join parallelism

2.1 Introduction

Mainstream compilers, such as GCC [366], ICC [193], and LLVM [230] provide linguistic extensions for frameworks such as Cilk Plus[191] and OpenMP [21, 293] that allow programmers to write fork-join parallel programs. Typically in such frameworks, one can specify parallelism at a high level by denoting tasks or loops iterations that may be executed concurrently.

Although these mainstream compilers support fork-join parallelism, they struggle to optimize programs when they encounter such linguistic constructs. Paradoxically this can even mean that programs you'd expect to show large parallel speedups, are slower than the equivalant serial code. Consider, for example, the parallel `cilk_for` loop on lines 7–8 in Figure 2-1a, which indicates that iterations of the loop are free to execute in parallel. In a serial version of this loop, where the `cilk_for` keyword is replaced by an ordinary `for` keyword, each of the compilers GCC 5.3.0, ICC 16.0.3, and Cilk Plus/LLVM 3.9.0 observes that the call to `mag` on line 8 produces the same value in every iteration of the loop, and they optimize the loop by computing this value only once before the loop executes. This optimization dramatically reduces the total time to execute `normalize` from $\Theta(n^2)$ to $\Theta(n)$.

a

```
01 __attribute__((const))
02 double mag(const double *A, int n);
03
04 void normalize(double *restrict out,
05                const double *restrict in, int n) {
06
07   cilk_for (int i = 0; i < n; ++i)
08     out[i] = in[i] / mag(in, n);
09 }
```

b

```
10 __attribute__((const))
11 double mag(const double *A, int n);
12
13 void normalize(double *restrict out,
14                const double *restrict in, int n) {
15   #pragma omp parallel for
16   for (int i = 0; i < n; ++i)
17     out[i] = in[i] / mag(in, n);
18 }
```

Figure 2-1: A function that GCC, ICC, and Cilk Plus/LLVM all fail to optimize effectively. **a** A Cilk version of the code. The `cilk_for` loop on lines 7–8 allows each iteration of the loop to execute in parallel. The `mag` function computes the norm of a vector in $\Theta(n)$ time. The call to `mag` on line 8 can be safely moved outside of the loop, but none of these three mainstream compilers perform this code motion, even though they all do so when the `cilk_for` keyword is replaced with an ordinary `for` keyword. **b** The corresponding OpenMP code.

Although this same optimization can, in principle, be performed on the actual parallel loop in the figure, no mainstream compiler performs this code-motion optimization. The same is true when the parallel loop is written using OpenMP, as shown in Figure 2-1b.

This failure to optimize stems from how these compilers for serial languages implement parallel linguistic constructs. The compiler for a serial language, such as C [209] or C++ [373], can be viewed as consisting of three phases: a front end, a middle end, and a back end. The front end parses and type-checks the input program and translates it to an ***intermediate representation (IR)***, which represents the control flow of the program as a more-or-less language-independent ***control-flow graph (CFG)*** [9, Sec. 8.4.3]. The middle end consists of optimization passes that transform the IR into a more-efficient form. These optimizations tend to be independent of the instruction-set architecture of the target computer. The back end translates the optimized IR into machine code, performing low-level machine-dependent optimizations.

GCC, ICC, and Cilk Plus/LLVM all ***lower*** the parallel constructs — transform the parallel constructs to a more-primitive representation — in the front end. To compile the code in Figure 2-1a, for example, the front-end translates the parallel loop in lines 7–8 into IR in two steps. (The OpenMP code in Figure 2-1b is handled similarly.) First, the loop body (line 8) is lifted into a helper function. Next, the loop itself is replaced with a call to a library function implemented by the Cilk Plus runtime system, which takes as arguments

the loop bounds and helper function, and handles the spawning of the loop iterations for parallel execution. Since this process occurs in the front end, it renders the parallel loop unrecognizable to middle-end loop-optimization passes, such as code motion. In short, these compilers treat parallel constructs as syntactic sugar for opaque runtime calls, which confounds the many middle-end analyses and optimizations.

2.1.1 Previous approaches

This thesis aims to enable middle-end optimizations involving fork-join control flow by embedding parallelism directly into the compiler IR, an endeavor that has historically been challenging [243, 242]. For example, it is well documented [262] that traditional compiler transformations for serial programs can jeopardize the correctness of parallel programs. In general, four types of approaches have been proposed to embed parallelism in a mainstream compiler IR.

First, the compiler can use metadata to delineate logical parallelism. LLVM's parallel loop metadata [82], for example, is attached to memory accesses in a loop to indicate that they have no dependence on other iterations of the same loop. LLVM can only conclude that a loop is parallel if all its memory accesses are labeled with this metadata. Unfortunately, encoding parallel loops in this way is fragile, since a compiler transformation that moves code into a parallel loop risks serializing the loop from LLVM's perspective.

Second, the compiler can use intrinsic functions to demark parallel tasks. (For examples, see [428, 306, 244].) Often, either existing serial analyses and optimizations must be shut down when code contains these intrinsics, or the intrinsics offer minimal opportunities for compiler optimization.

Third, the compiler can use a separate IR to encode logical parallelism in the program. The HPIR [428, 27], SPIRE [211], and INSPIRE [199] representations, for instance, model parallel constructs using an alternative IR, such as one based on the program's abstract syntax tree [9, Sec. 2.5.1]. Such an IR can support optimizations involving parallel constructs without requiring changes to existing middle-end optimizations. But adopting a separate IR into a mainstream compiler has historically been criticized [245] as requiring consider-

able effort to engineer, develop, and maintain the additional IR to the same standards as the compiler's existing serial IR.

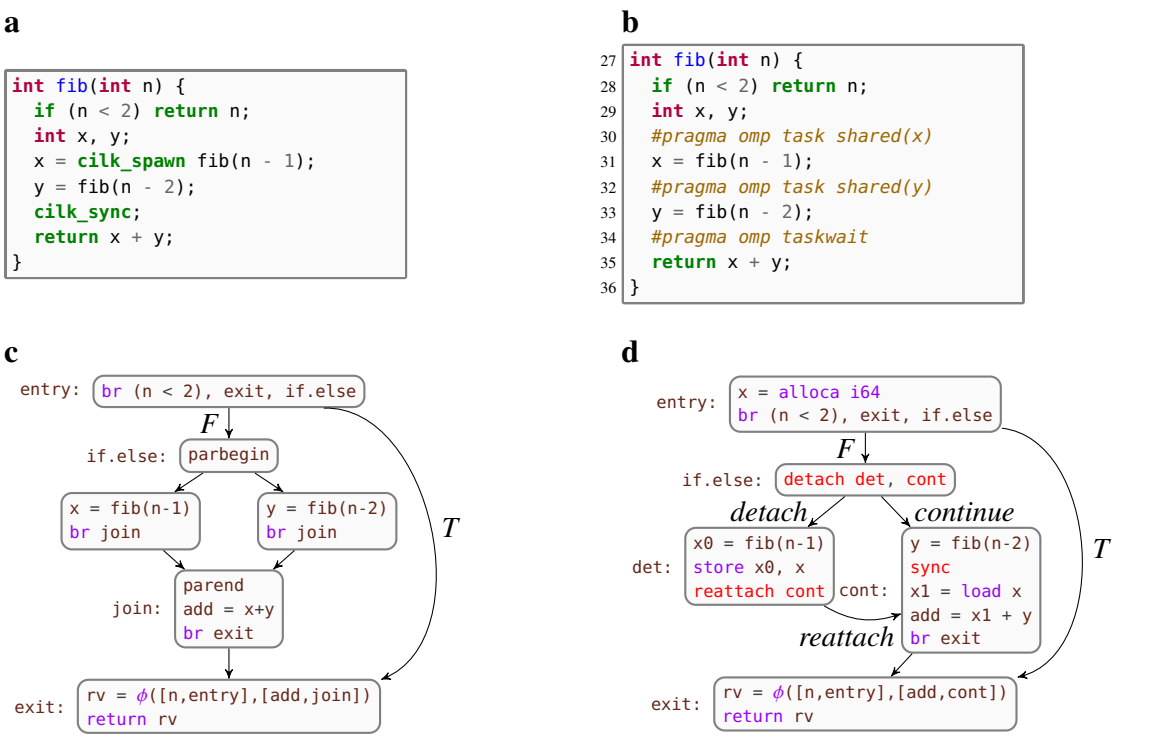


Figure 2-2: Comparison between a traditional CFG with symmetric parallelism and Tapir's CFG with asymmetric parallelism. **a** The Cilk function `fib` computes Fibonacci numbers. The `cilk_spawn` on line 22 allows the two recursive calls to `fib` to execute in parallel, and the `cilk_sync` on line 24 waits for the spawned call to return. A serial execution of `fib` executes `fib(n-1)` before `fib(n-2)`. **b** A comparable implementation of `fib` using OpenMP task parallelism. **c** A CFG for `fib` that encodes parallelism symmetrically. Rectangles denote basic blocks, which contain C-like pseudocode for `fib`. Edges denote control flow between basic blocks. The `parbegin` and `parend` statements create and synchronize the parallel calls to `fib`. The `br` instruction encodes either an unconditional or a conditional branch. True and false edges from a conditional branch are labeled *T* and *F*, respectively. The ϕ instruction, used to support a static-single-assignment (SSA) form of the program (see Section 2.2), takes as its arguments pairs that associate a value with each predecessor basic block of the current block. At runtime the ϕ instruction returns the value associated with the predecessor basic block that executed immediately before the current block. **d** The Tapir CFG for `fib`, which encodes parallelism asymmetrically. The `alloca` instruction allocates shared-memory storage on the call stack for a local variable. Section 2.2 defines the `detach`, `reattach`, and `sync` instructions and the *detach*, *reattach*, and *continue* edge types.

Fourth, the compiler can augment its existing IR to encode logical parallelism, which is the approach that Tapir follows. Unlike Tapir, all prior research on parallel precedence

graphs [364, 363], parallel flow graphs [362, 153], concurrent control-flow graphs [233, 288], and parallel program graphs [338, 337] represent parallel tasks as symmetric entities in a CFG. For the parallel `fib` function in Figures 2-2a and 2-2b, for example, the parallel flow graph in Figure 2-2c illustrates how forked subcomputations might be represented symmetrically. Some of these approaches struggle to represent common parallel constructs, such as parallel loops [233, 211], while others exhibit problems when subjected to standard compiler analyses and transformations for serial programs [233, 337, 153, 217, 364, 363, 331]. Existing serial-program analyses in LLVM, for example, assume that a basic block with multiple predecessors can observe the variables of only one predecessor at runtime. For the parallel flow graph in Figure 2-2c, however, instructions in the `join` block must observe the values of `x` and `y` from both of its predecessors, as has been observed by [233]. Parallel loops exacerbate this problem by allowing a dynamic number of tasks to join at the same basic block. Previous research [331, 6] has proposed solutions to these problems, including additional representations of the program and augmented analyses that account for interleavings of parallel instructions, but adopting these techniques into a mainstream compiler seems to require extensive changes to the existing codebase.

2.1.2 The Tapir approach

This thesis introduces Tapir, a compiler IR that represents logical fork-join parallelism asymmetrically in the program's CFG. The asymmetry corresponds to the assumption of ***serial semantics*** [123], which means it is always semantically correct to execute parallel tasks in the same order as an ordinary serial execution.

Tapir adds three instructions — `detach`, `reattach`, and `sync` — to the IR of an ordinary serial compiler to express fork-join parallel programs with serial semantics. Figure 2-2d illustrates the Tapir CFG for the `fib` function. As with the symmetric parallel flow graph in Figure 2-2c, Tapir places the logically parallel recursive calls to `fib` in separate basic blocks. But these blocks do not join at a synchronization point symmetrically. Instead, one block connects to the other, reflecting the serial execution order of the program.

The Tapir approach provides five advantages:

1. Introducing fork-join parallelism into the compiler is relatively easy.

2. The IR is expressive and can represent fork-join control constructs from different parallel-language extensions.
3. Tapir parallel constructs harmonize with the invariants associated with existing representations of serial code.
4. Standard serial optimizations work on parallel code with few modifications.
5. The optimizations enabled by Tapir's parallelism constructs are effective in practice.

I discuss each of these advantages in turn.

2.1.3 Ease of implementation

Tapir's asymmetric representation of logically parallel tasks makes it relatively simple to integrate Tapir into an existing compiler's intermediate representation such as LLVM IR [82]. Figure 2-3 documents the lines of code added, modified, or deleted to implement a prototype of Tapir in LLVM. As Figure 2-3 shows, Tapir/LLVM was implemented with about 6000 lines, compared to LLVM's roughly 4-million-line codebase. Moreover, fewer than 2000 lines of code were needed to adapt LLVM's existing compiler analyses and transformations to accommodate Tapir.

Compiler Component	*LLVM 4.0svn*	*Tapir/LLVM*	
Instructions	105,995	943	1,768
Memory Behavior	21,788	445	
Optimizations	152,229	380	
Parallelism Lowering	0	3,782	
Other	3,803,831	460	
Total	4,083,843	6,010	

Figure 2-3: Breakdown of the lines of code added, modified, or deleted in LLVM to implement the Tapir/LLVM prototype.

The breakdown of lines is as follows. The lines for "Instructions" add Tapir's instructions to LLVM IR and adapt LLVM's routines for reading and writing LLVM IR and bitcode files. Conceptually, these changes allow LLVM to correctly compile a Tapir program to a serial executable with no optimizations. The lines for "Memory Behavior" control how Tapir instructions may interact with memory operations, preventing the compiler from creating any races. The lines for "Optimizations" perform any adjustments required for LLVM

analyses and transformations to compile a Tapir program at optimization level `-O3`. Most of these modifications are not necessary for creating a correct executable but are added to allow the compiler to perform additional optimizations, such as parallel tail-recursion elimination (described in Section 2.4). The lines for "Parallelism Lowering" translate Tapir instructions into Cilk Plus runtime calls and allow the code to be race-detected with a provably good race detector [116]. The lines for "Other" address a bug in LLVM's implementation of `setjmp` and implement useful features for our development environment.

2.1.4 Expressiveness of Tapir

Tapir can express logical fork-join parallelism in parallel programs that have serial semantics. For example, Figure 2-2 illustrates how Tapir can express the parallelism encoded by the `cilk_spawn` and `cilk_sync` linguistics from Cilk++ [234] and Cilk Plus [191], as well as the parallelism encoded by OpenMP `task` and `taskwait` clauses [21]. Similarly, Tapir can express the parallelism encoded by OpenMP parallel sections [293] and Habanero's `async` and `finish` constructs [64]. Tapir can also express parallel loops, including `cilk_for` loops and OpenMP parallel loops that have serial semantics (described in Section 2.2). Other parallel constructs, such as those proposed in the C++17 parallelism extensions, can be represented as well. However, parallel operations that cannot be expressed in terms of fork-join parallelism, such as OpenMP's `ordered` clause, cannot be represented directly using Tapir's `detach`, `reattach`, and `sync` instructions.

Tapir makes minimal assumptions about the consistency [312, 49] of concurrent memory accesses. Tapir assumes that memory is shared among parallel tasks and that virtual-register state is local to each task. Parallel instructions in Tapir can exhibit a ***determinacy race***[1] [116] if they access the same memory location concurrently and at least one instruction writes to that location. Tapir itself does not fully define the possible outcomes of a determinacy race, and instead defers to existing compiler mechanisms, such as LLVM's atomic memory-ordering constraints [82], to define whichever memory model they choose. For any targeted runtime system, Tapir relies on a correct implementation of lowering in or-

[1]Determinacy races are also called general races [285] and are distinct from data races, which involve nonatomic accesses to critical regions.

der to implement the necessary synchronization, but Tapir is oblivious to how that runtime system implements the synchronization.

2.1.5 Serial semantics

By grounding its model of parallelism in serial semantics, Tapir enables common compiler optimizations for serial code to work on parallel code. Intuitively, because Tapir always allows parallel tasks to execute in their ordinary serial execution order, the compiler can to optimize parallel code in any manner that preserves the serial semantics of the program and does not introduce new determinacy races. These mild constraints support common optimizations on parallel code, such as sequentialization, which can be invalid under models of parallelism without serial semantics [393].

2.1.6 Optimizations

In practice, the Tapir team has found that Tapir enables a wide variety of standard compiler optimizations to work with parallel code. The prototype implementation of Tapir/LLVM, for example, successfully moves the call to `mag` in Figure 2-1 outside of the loop, just as it would for a serial `for` loop. As Section 2.4 discusses, Tapir enables other optimizations, including common-subexpression elimination [275, Sec. 12.2], loop-invariant-code motion [275, Sec. 13.2], and tail-recursion elimination [275, Sec. 15.1], to work on parallel code. Tapir also enables new optimizations on parallel control flow.

2.1.7 Evaluation of Tapir/LLVM

The compiler optimizations that Tapir enables are effective in practice. We evaluated the Tapir approach by measuring the performance of 20 Cilk application benchmarks compiled using Tapir/LLVM. We compared the performance of these executables to those produced by a comparable reference compiler, called Reference. Conceptually, Reference lowers parallel linguistic constructs directly into runtime calls, as mainstream compilers do today, but otherwise performs the same set of optimization passes as Tapir/LLVM. Section 2.6 describes our experimental setup in detail, including the design of Reference.

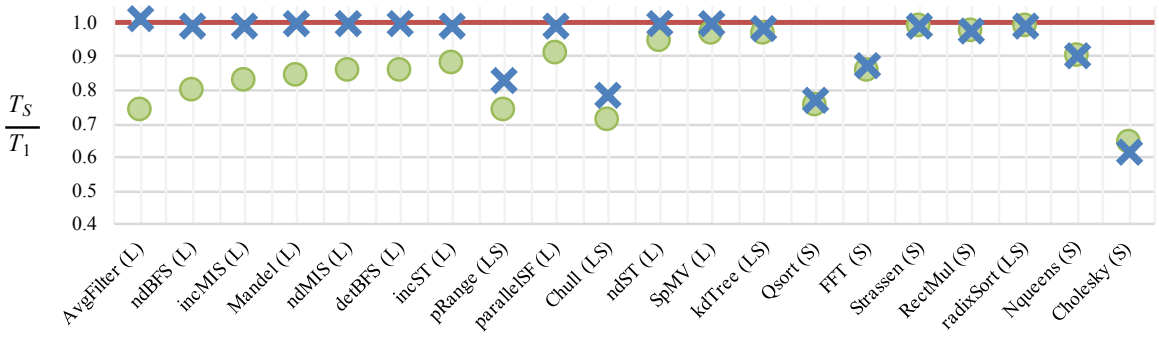


Figure 2-4: Comparison of the work efficiency of 20 parallel application benchmarks compiled using Tapir/LLVM (X's) and the comparable Reference compiler (O's), described in Section 2.6, which lowers parallelism in the compiler front end. Each point plots the work efficiency T_S/T_1 of a compiled benchmark, where T_1 is the work of the benchmark and T_s is the running time of the serial elision of the benchmark. Higher values indicate better work efficiency. The horizontal line at 1.0 plots the theoretically maximum work efficiency $T_S/T_1 = 1$. Benchmarks are sorted by decreasing difference in work efficiency between Tapir/LLVM- and Reference-compiled executables. Benchmarks marked with an "L" use parallel loops, and benchmarks marked with an "S" use `cilk_spawn`.

Figure 2-4 presents the results of comparing Tapir/LLVM and Reference in terms of the "work efficiency" of the compiled benchmarks. To perform this comparison, We compiled each benchmark using each compiler and then ran the executable on a single processing core of a multicore machine to measure its ***work***, the 1-core running time, denoted T_1. We also used each compiler to compile, run, and measure the 1-core running time of the ***serial elision*** [123] of each benchmark, denoted T_S, in which the benchmark is converted into a corresponding serial program by replacing all parallel linguistic constructs with their serial equivalents. We then computed the ***work efficiency*** of each compiled benchmark, which is the ratio T_S/T_1 of the running time T_S of the benchmark's serial elision divided by the work $T_!$ of the benchmark. In theory, the maximum possible work efficiency is $T_S/T_1 = 1$, but in practice, quirky behaviors of the compiler and multicore architecture can occasionally produce work efficiencies greater than 1. As Figure 2-4 shows, for most benchmarks, the executables compiled using Tapir/LLVM achieve equal or higher work efficiency than those compiled using Reference. Moreover, for many benchmarks, and particularly those implemented using parallel loops, Tapir/LLVM produces executables that achieve nearly

optimal work efficiency. Section 2.6 elaborates on these experiments.

2.1.8 Contributions

This chapter makes the following research contributions:

- The design of a compiler IR that represents fork-join parallelism asymmetrically, which enables existing serial optimizations to operate on parallel code and which also enables parallel optimizations.
- The implementation of Tapir/LLVM in the LLVM compiler by modifying about 6000 source lines of code (0.15% of the 4-million-line LLVM codebase).
- The implementation of parallel optimizations such as unnecessary synchronization elimination and parallel-loop scheduling.
- Experiments that demonstrate the advantage of embedding fork-join parallelism into a compiler's IR, as opposed to dealing with parallelism only in the compiler's front end.

2.1.9 Outline

The remainder of this thesis is organized as follows. Section 2.2 describes Tapir's representation and properties. Section 2.3 discusses how analysis passes can be adapted to operate on Tapir programs. Section 2.4 describes various optimizations on parallel control flow that Tapir enables. Section 2.5 describes auxiliary software we developed to exercise and test Tapir/LLVM. Section 2.6 discusses our evaluation of the effectiveness of Tapir. Section 2.7 discusses related work. Section 2.8 provides some concluding remarks. An appendix describes how to set up Tapir/LLVM and how to download and run our suite of application benchmarks.

2.2 Tapir

This chapter describes how Tapir represents logically parallel tasks asymmetrically in the CFG of a program. I define Tapir's three new instructions and how they interact with

LLVM's static single-assignment (SSA) form [9, Sec. 6.2.4]. Although I describe Tapir as an extension to LLVM IR [82], the Tapir team sees no reason why other compilers cannot gain similar advantages from Tapir-like instructions.

Like LLVM IR, Tapir treats a program function as a CFG $G = (V, E, v_0)$, where

- the set V of vertices represents the function's ***basic blocks***: sequences of LLVM instructions, where control flow can only enter through the first instruction and leave from the last instruction;
- the set E of edges denote control flow between (basic) blocks; and
- the designated vertex $v_0 \in V$ represents the ***entry point*** of the function.

2.2.1 Tapir instructions

Tapir extends LLVM IR with three instructions: `detach`, `reattach`, and `sync`. The `detach` and `reattach` instructions together delineate logically parallel tasks, and the `sync` instruction imposes synchronization on parallel tasks. The three instructions have the following syntax, where $b, c \in V$:

```
detach label b, label c
reattach label c
sync
```

The `label` keywords indicate that b and c are (labels of) basic blocks in V.

The `detach` and `reattach` instructions together delineate a parallel task as follows. A `detach` instruction terminates the block a that contains it and takes a ***detached*** block b and a ***continuation*** block c as its arguments. The `detach` instruction ***spawns*** the task starting at block b, allowing that task to execute in parallel with block c. The control-flow edge $(a, b) \in E$ is a ***detach*** edge, and the edge $(a, c) \in E$ is a ***continue*** edge. A `reattach` instruction, meanwhile, terminates the block a' that contains it and takes a single ***continuation*** block c as its argument, inducing a ***reattach*** edge $(a', c) \in E$ in the CFG. The `reattach` terminates the task spawned by a preceding `detach` instruction with the same continuation block. Together, a `detach` instruction and associated `reattach` instructions

demark the start and end of a parallel task and indicate that that task can execute in parallel with their common continuation block.

For the example in Figure 2-2d, the `detach` in the `if.else` block and the `reattach` in the `det` block share the same continuation block `cont`. Together, this `detach` and this `reattach` indicate that the `det` block is a parallel task which can execute in parallel with the `cont` block. In general, a parallel task delineated by `detach` and `reattach` can consist of many basic blocks in a single-entry subgraph.

The `detach` and `reattach` instructions in a CFG obey several structural properties. A `reattach` instruction j ***reattaches*** a `detach` instruction i if i and j share a common continuation block and there is a path from the detached block of i to j. Tapir assumes that every CFG $G = (V, E, v_0)$ obeys the following invariants on every `detach` instruction i and `reattach` instruction j in G:

1. A `reattach` instruction reattaches exactly one `detach` instruction.
2. If j reattaches i, then every path from v_0 to the block terminated by j passes through the detach edge of i, that is, the detach edge of i ***dominates*** j.
3. Every path starting from the detached block of i must reach a block terminated by a `reattach` instruction that reattaches i.
4. If j reattaches i and a path from i to j passes through the detach edge of another `detach` instruction i', then it must also pass through a `reattach` instruction j' that reattaches i'.
5. Every cycle containing a `detach` instruction i must pass through a `reattach` instruction that reattaches i.
6. The continuation block of j cannot contain any ϕ instructions [9, Sec 6.2.4].

These invariants imply that, at runtime, a `detach` instruction i with detached block b and continuation block c spawns the execution of a ***detached sub-CFG***, which is the single entry sub-CFG starting at b induced by all blocks on paths from b to a `reattach` instruction that reattaches i.

The dynamic execution of the program organizes memory as a tree of ***parallel contexts***. A new parallel context is created as a child of the current context when control enters a

function or follows a detach edge. When control executes a `reattach` instruction or leaves a function, the context is destroyed and the parent's context becomes the current context. An `alloca` instruction allocates shared memory in the current context.

The `sync` instruction synchronizes tasks spawned within its parallel context. At runtime, a `sync` instruction dynamically waits for the set of sub-CFG's detached in the same parallel context or any of its descendant parallel contexts to reach a `reattach` instruction. In the Tapir CFG illustrated in Figure 2-2d, for example, the `sync` instruction in the `cont` block simply waits for the execution of the `det` block to complete. Unlike `reattach` instructions, `sync` instructions are not explicitly associated with `detach` instructions, and they, in fact, can be executed within conditionals. A `sync` instruction j ***syncs*** a `detach` instruction i if i and j belong to the same parallel context and the CFG detached by i cannot be guaranteed to have completed when j executes.

2.2.2 Static single-assignment form

LLVM's static single-assignment (SSA) form [9, Sec. 6.2.4] must be adapted for Tapir programs. SSA form ensures that each virtual register is set at most once in a function. LLVM IR employs the ϕ ***instruction*** [9, Sec 6.2.4] to combine definitions of a variable from different predecessors of a basic block. In adapting SSA to Tapir, one concern is that a ϕ instruction might allow registers defined in the detached sub-CFG to be used in the continuation. A basic block containing a ϕ instruction must avoid inheriting register definitions from predecessors that are connected by reattach edges. Otherwise, a register in the detached sub-CFG might not have been computed by the time the continuation executes.

We implemented this constraint by simply forbidding reattach edges from going into basic blocks with ϕ instructions. But what if the continuation c of a `detach` instruction begins with a ϕ instruction? In this case, Tapir creates a new basic block c' containing only a branch instruction to c. Tapir reroutes the reattach and continuation edges originally going to c so that they go instead to c'. All other edges going to c are left in place.

The reason this solution works is as follows. No reattach edges in the resulting CFG go to blocks containing ϕ instructions. Because a detached sub-CFG does not dominate

any outside block, registers in the detached CFG can only be used in ϕ instructions of the immediate successors of the detached sub-CFG. Since the continuation is the only immediate successor of the detached sub-CFG and it contains no ϕ instructions, no registers from the detached sub-CFG may be accessed in the continuation.

2.2.3 Asymmetry in Tapir

The `detach` and `reattach` instructions express parallel tasks asymmetrically both syntactically in the structure of the CFG and semantically in the way memory state is managed. Both asymmetries are illustrated in Figure 2-2d.

First, the CFG detached by a `detach` instruction is connected by a reattach edge to the continuation block of that instruction, even though they can execute in parallel. For example, the reattach edge between `det` and `cont` in Figure 2-2d breaks the symmetry between them. Reattach edges reflect the serial semantics of a Tapir program, which dictates that a serial execution of the program executes the detached CFG to completion before starting to execute the continuation block. In fact, the parallel task delineated by a `detach` and a `reattach` instruction can be ***serialized*** by replacing the `detach` instruction with an unconditional branch to its detached block and replacing the `reattach` with an unconditional branch to its continuation block. In contrast, parallel flow graphs and similar previously explored representations join logically parallel tasks in the CFG at a synchronization point. By supporting separate `reattach` and `sync` instructions, Tapir decouples the termination of a parallel task from its synchronization.

Second, although memory state is shared among all parallel tasks in Tapir, a virtual register defined in a detached sub-CFG is not accessible in its parent parallel context. For example, the continuation block `cont` in Figure 2-2d cannot assume that the register value `x0` returned by `fib(n-1)` in block `det` is accessible, because the two basic blocks belong to different parallel contexts. Thus, `cont` must load it again after the `sync` instruction.

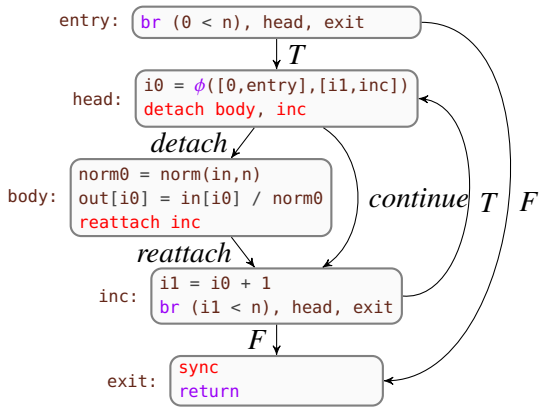


Figure 2-5: Tapir CFG for the parallel loops in Figure 2-1.

2.2.4 Parallel loops in Tapir

Figure 2-5 illustrates Tapir's default representation of the parallel loops from Figure 2-1. As Figure 2-5 shows, Tapir can represent a parallel loop in the CFG as an ordinary loop, where the `head` block repeatedly spawns the `body` block, and the `exit` block syncs the detached CFG's. Section 2.4 describes how this representation of parallel loops allows existing compiler loop optimizations to operate on Tapir parallel loops with only minor modifications. Although this loop structure can exhibit poor parallel performance when the loop body is small, separate optimization passes in Tapir/LLVM (see Section 2.4) transform this parallel-loop representation into a divide-and-conquer form that exhibits good performance.

2.3 Analysis passes

This chapter describes how LLVM's analysis passes can be adapted to operate on Tapir programs. I first discuss constraints on how Tapir programs can be safely transformed. Implementing these contraints on LLVM optimization passes primarily involves adapting standard compiler analyses — specifically alias analysis [9, Ch. 12], dominator analysis [9, Ch. 9], and data-flow analysis [9, Ch. 9] — to accomodate Tapir's instructions. I describe how each of these analyses was minimally modified to support Tapir.

2.3.1 Constraints on transformations

To be correct, a code transformation on a Tapir program must preserve the program's serial semantics, and it must not introduce any new behaviors into the program's set of behaviors. A program can exhibit more than one behavior if it contains a determinacy race. In general, the result of a determinacy race can vary nondeterministically from run to run depending on the order in which the participating instructions access the memory location. To avoid introducing new behaviors, code transformations must not create determinacy races, although they can eliminate determinacy races. Many existing serial optimizations can be adapted to respect these properties by adapting the standard compiler analyses they rely on. I now describe how LLVM's alias, dominator, and data-flow analyses were adapted for Tapir.

2.3.2 Alias analysis

LLVM uses alias analysis [9, Ch. 12] to determine whether different instructions might reference the same locations in memory, and in particular, to restrict the reordering of instructions that access the same memory. Tapir/LLVM modifies LLVM's alias analysis to prevent optimizations that move code around from introducing determinacy races. In particular, Tapir adapts LLVM's alias analysis to treat the instructions as if they access memory. For example, consider an instruction k that performs a load or a store. There are four cases to consider when moving k around either a `detach` instruction i or a `sync` instruction j:

1. The instruction k moves from before i to after i.
2. The instruction k moves from after i to before i.
3. The instruction k moves from before j to after j.
4. The instruction k moves from after j to before j.

Neither Case 2 nor Case 3 can introduce a determinacy race, because both motions serialize the execution of k with respect to the sub-CFG detached by i. Cases 1 and 4 might introduce a determinacy race, however, if k loads or stores a memory location that is also accessed by the CFG detached by i. To handle Case 1, i is treated as if it were a function call that

accesses all memory locations accessed in the CFG detached by i. Similarly, for Case 4, j is treated as if it were a function call that accesses all memory locations accessed by all instructions that j might sync. A `reattach` instruction is treated as a compiler fence that prevents instructions from moving across it. With these modifications, existing rules in LLVM that restrict reordering of loads and stores properly restrict memory reordering around Tapir's instructions.

2.3.3 Dominator analysis

Optimization passes determine what values are available to an instruction in part by using dominator analysis [9, Ch. 9], which deduces the dominance relation between all basic blocks and edges in a CFG. To handle Tapir programs correctly, optimization passes must not mistakenly cause instructions to use virtual registers that are defined in logically parallel tasks. If instruction i dominates instruction j, than an optimization pass might assume that the value produced by i is always available when j executes.

The asymmetry of Tapir's representation allows LLVM's dominator analysis to analyze Tapir programs correctly without any changes. Ignoring the names of edges, the difference between the CFG $G = (V, E, v_0)$ of a Tapir program and the CFG $G' = (V, E', v_0)$ of its serial elision is the set $E - E'$ of continue edges, each of which connects a `detach` instruction to its continuation. A continue edge short-cuts a detached sub-CFG, changing the continuation's immediate dominator from the detached sub-CFG to the block containing the `detach` instruction itself. This configuration of detach, reattach, and continue edges looks much like an ordinary `if` construct in which the detached sub-CFG is conditionally executed. As a result, dominator analysis never concludes that an instruction in a detached sub-CFG can execute before the corresponding continuation block.

2.3.4 Data-flow analysis

A wide class of code transformations, including those that might move instructions across a reattach edge, rely on data-flow analysis [9, Ch. 9] to examine the propagation of values along different paths through a CFG $G = (V, E, v_0)$. Fundamental to data-flow analysis is an

understanding of the set of possible program states at the beginning and end of each basic block $b \in V$, denoted $\textsc{in}(b)$ and $\textsc{out}(b)$, respectively.

To illustrate how LLVM's data-flow analyses were adapted to Tapir, let us examine the particular case of forward data-flow analysis. (Backward data-flow analysis is similar.) In an ordinary serial CFG, forward data-flow analysis evaluates $\textsc{in}(b)$ as the union of $\textsc{out}(a)$ for each predecessor block a of b:

$$\textsc{in}(b) = \bigcup_{(a,b)\in E} \textsc{out}(a) \ .$$

To handle Tapir CFG's, data-flow analyses must be adapted specifically to handle reattach edges. Because Tapir's asymmetric representation propagates virtual registers and memory state differently across a reattach edge, the modifications to LLVM data analyses consider registers and memory separately.

For variables stored in shared memory, the standard data-flow equations remain unchanged. Thus, LLVM need not be modified to handle them for Tapir.

For register variables, however, LLVM's data-flow analyses must be modified to exclude the values in registers from an immediate predecessor a of a basic block b if the edge $(a, b) \in E$ is a reattach edge. Denote the set of reattach edges in E by E_R. For a Tapir CFG, forward data-flow analyses define $\textsc{in}(b)$ for register variables as

$$\textsc{in}(b) = \bigcup_{(a,b)\in E-E_R} \textsc{out}(a) \ ,$$

that is, they ignore predecessors across a reattach edge. With this change, Tapir/LLVM correctly propagates register variables through the CFG, never allowing register values in a basic block to use register values set in a logically parallel detached sub-CFG.

2.4 Optimization passes

Tapir enables LLVM's existing optimization passes [248] to work across parallel control flow. It also enables new optimization passes that specifically target Tapir's fork-join

a

```
37 void search(int low, int high) {
38   if (low == high) search_base(low);
39   else {
40     cilk_spawn search(low, (low+high)/2);
41     search((low+high)/2 + 1, high);
42     cilk_sync;
43   }
44 }
```

b

```
45 void search(int low, int high) {
46   if (low == high) search_base(low);
47   else {
48     int mid = (low+high)/2;
49     cilk_spawn search(low, mid);
50     search(mid + 1, high);
51     cilk_sync;
52   }
53 }
```

Figure 2-6: Example of common-subexpression elimination on a Cilk program. **a** The function `search`, which uses parallel divide-and-conquer to apply the function `search_base` to every integer in the closed interval [`low`, `high`]. **b** An optimized version of `search`, where the common subexpression `(low+high)/2` in lines 40 and 41 of the original version is computed only once and stored in the variable `mid` in line 48 of the optimized version.

parallel constructs. This chapter discusses four representative optimizations. Common-subexpression elimination [275, Sec. 12.2] illustrates an optimization pass that "just works" with the additional Tapir instructions. Loop-invariant code motion [275, Sec. 13.2], and tail-recursion elimination [275, Sec. 15.1] were the only two out of LLVM's roughly 80 optimization passes that required any modification to work effectively on parallel code. Parallel-loop scheduling serves as an example of a new optimization pass.

2.4.1 Common-subexpression elimination

The common-subexpression elimination (CSE) optimization identifies redundant calculations and transforms the code so that they are only computed once. For example, the expression `(low+high)/2` in Figure 2-6a is computed in both line 40 and line 41. Tapir/LLVM performs CSE on this code, producing code equivalent to that in Figure 2-6b. Existing mainstream compilers that support fork-join parallelism do not eliminate this common subexpression, however, and they compute `(low+high)/2` twice. Tapir/LLVM can perform CSE across either a continue edge, as in the example, or a detach edge. Like the vast majority of optimization passes in Tapir/LLVM, CSE "just works" on Tapir code without any modifications to LLVM's CSE pass.

2.4.2 Loop-invariant code motion

The loop-invariant code motion (LICM) optimization [275, Sec. 13.2] aims to move computations out of loop bodies if they compute the same value on every iteration of the loop. LICM is responsible, for example, for moving the call to `norm` in the parallel loop in Figure 2-1a outside of the loop, as described in Section 2.1. By adapting LICM to handle parallel loops, Tapir/LLVM reduces the asymptotic serial running time of this parallel loop from $\Theta(n^2)$ to $\Theta(n)$.

Tapir/LLVM requires a minor change to LLVM's LICM pass to handle parallel loops. Consider the CFG illustrated in Figure 2-5, which models the parallel loops in Figure 2-1. For the serial elision of the loop, which would have a similar graph structure except with the continue edge missing, LLVM attempts to find candidate computations to move outside the loop by looking for instructions in the basic blocks of the loop body that dominate the exit block of the loop, such as the block `inc` in Figure 2-5. (The block labeled `exit` is the exit of the function, not the loop exit.) For a parallel loop, however, this analysis fails to identify any code to move due to the existence of the continue edge. As Figure 2-5 shows, with the continue edge, blocks in the loop body can never dominate the exit block `inc` as they could for the serial elision.

Tapir/LLVM modifies LLVM's LICM pass to handle a parallel loop by analyzing the serial elision of the loop, which essentially means ignoring continue edges. For simple parallel loop structures with a single continue edge, such as that shown in Figure 2-5, this modification is implemented by finding blocks in the loop body that dominate the predecessors of the loop exit. The modification required changing only 25 lines of LLVM's LICM pass.

2.4.3 Tail-recursion elimination

Tail-recursion elimination (TRE) [275, Sec. 15.1] aims to replace a recursive call at the end of a function with a branch to the start of the function. By eliminating these recursive tail calls, TRE can avoid function-call overheads and reduce the stack space they consume. This optimization can especially benefit fork-join parallel programs, as many parallel run-

a

```
54 void pqsort(int* start, int* end) {
55   if (begin == end) return;
56   int* mid = partition(start, end);
57   swap(end, mid);
58   cilk_spawn pqsort(begin, mid);
59   pqsort(mid+1, end);
60   cilk_sync;
61   return;
62 }
```

b

```
75 void pqsort(int* start, int* end) {
76   if (begin == end) return;
77   int* mid = partition(start, end);
78   swap(end, mid);
79   cilk_spawn pqsort(begin, mid);
80
81   start = mid+1;
82   // Begin inlined code
83   if (begin == end) goto join;
84   mid = partition(start, end);
85   swap(end, mid);
86   cilk_spawn pqsort(begin, mid);
87   pqsort(mid+1, end);
88   cilk_sync;
89   // End inlined code
90
91 join:
92   cilk_sync;
93   return;
94 }
```

c

```
63 void pqsort(int* start, int* end) {
64 pqsort_start:
65   if (begin == end) {
66     cilk_sync;
67     return;
68   }
69   int* mid = partition(start, end);
70   swap(end, mid);
71   cilk_spawn pqsort(begin, mid);
72   start = mid+1;
73   goto pqsort_start;
74 }
```

Figure 2-7: Example of tail-recursion elimination on a parallel quicksort program. **a** The Cilk function `pqsort` sorts an array of integers in the range specified by the `start` and `end` pointers. **b** A version of `pqsort` where the recursive tail call on line 59 has been replaced by one round of inlining. **c** A version of `pqsort` where tail-recursion elimination has removed the recursive tail call on line 59.

time systems impose additional setup and cleanup overhead on a spawned function.

LLVM's existing TRE pass can perform the TRE optimization on Tapir programs with just a minor modification. Specifically, the modified TRE pass ignores `sync` instructions after the tail-recursive call. Further, if TRE is applied and ignores a `sync` instruction, it must then insert a `sync` instruction before any remaining returns. This modification to LLVM's TRE pass required changing only 68 lines.

To see why these `sync` instructions can be safely ignored, consider Figure 2-7, which illustrates how Tapir/LLVM's TRE pass operates on the `pqsort` function, a parallel version of Hoare's quicksort algorithm [170]. The original tail-recursive code is shown in Figure 2-7a. Figure 2-7b illustrates the result of simply inlining the tail-recursive call. For the inlined code, all `return` statements are replaced with branches to the `join` label. Because there is a `cilk_sync` at the start of `join`, the `cilk_sync` on line 88 can be eliminated. call an arbitrary number of times, TRE can safely ignore a `cilk_sync` instruction after the final

tail-recursive call, assuming that it inserts a `cilk_sync` instruction before all remaining returns.

2.4.4 Parallel-loop scheduling and lowering

As discussed above and in Section 2.2, Tapir effectively represents a parallel loop as a serial loop over a body that is spawned every iteration. Depending on the number of iterations of the loop and the amount of work inside each loop, however, statically scheduling loop iterations in this way may be inefficient. For a parallel loop with a large number of iterations, for instance, it is faster to schedule the iterations in a recursive divide-and-conquer fashion, which produces more parallelism (see [258, Sec. 8.3]. For parallel loops with few iterations, however, the additional function calls required to perform the parallel divide-and-conquer can make the loop run slower than simply spawning off the iterations.

Tapir/LLVM implements a parallel optimization pass that schedules the iterations of a parallel loop using recursive divide-and-conquer, but only if that loop contains sufficiently many iterations. This pass is implemented as part of Tapir/LLVM's 3800-line lowering pass, which translates `detach`, `reattach`, and `sync` instructions into appropriate Cilk Plus runtime calls [190]. In particular, Tapir/LLVM uses the Cilk Plus runtime calls for `cilk_for` loops [190, Sec 10.7] to schedule parallel loops. Although we could have separated parallel-loop scheduling from lowering, we chose to combine these two passes so that we could perform fair comparisons between Tapir/LLVM and compilers that lower parallel constructs in their front end. We plan to separate the parallel-loop-scheduling and lowering passes in a future version of Tapir/LLVM.

2.4.5 Other optimization passes

Tapir/LLVM implements two minor parallel optimization passes: unnecessary-synchronization elimination and puny-task elimination. ***Unnecessary-synchronization elimination*** identifies and eliminates `sync` instructions that could not possibly sync a detached sub-CFG. ***Puny-task elimination*** serializes detached sub-CFG's that perform little or no work. If the runtime overhead of creating a parallel task outweighs the work in the task, the task might

as well be run serially. Both of these optimization passes were implemented in 52 lines of code by augmenting LLVM's SimplifyCFG pass.

2.5 Auxiliary software

This chapter describes auxiliary software that the Tapir team developed to exercise and test Tapir/LLVM. Although our research focuses on the middle end of the compiler, we implemented a front end for Cilk Plus. In addition, we developed compiler instrumentation that allows the compiler to interface to a race detector to verify the correctness of the Tapir/LLVM implementation.

To create the front end, the Tapir team created a modification of the Clang front end called PClang, which translates Cilk Plus codes to Tapir. We also created a version of Clang that can handle some OpenMP codes. PClang handles most of the fork-join control constructs specified by the Cilk Plus programming model, and specifically, enough to run all the benchmarks described in Section 2.6.

We augmented Tapir/LLVM in two ways to test the correctness of the implementation. First, we modified LLVM's internal verification pass to check that Tapir's invariants are also maintained. Second, we added an instrumentation pass to Tapir/LLVM to allow parallel executables to be tested for determinacy races using a provably good determinacy race detector. This race detector, based on the SP-bags algorithm [116], is guaranteed to find a determinacy race if an only if one exists in the program execution. The verification pass and race detector helped us locate and fix bugs in Tapir/LLVM, both within our code and within the underlying LLVM codebase. Tapir/LLVM now passes all tests in LLVM's regression test suites and correctly compiles our own suite of parallel test programs.

The instrumentation pass has proved useful for supporting other dynamic-analysis tools based on Tapir/LLVM. Genghis Chau of MIT adapted the Cilkprof scalability profiler [341] to use Tapir/LLVM and this instrumentation in order to build an integrated development environment with always-on race detection and scalability profiling facilities.

2.6 Evaluation

To evaluate the effectiveness of the approach, the Tapir team evaluated Tapir/LLVM on 20 benchmarks. The experiments support the contention that Tapir's approach of embedding parallelism in the IR is superior to lowering parallelism in the compiler front end. We could not simply run Tapir/LLVM against another compiler, such as Cilk Plus/LLVM [192], which lowers parallelism in the front end, because Cilk Plus/LLVM and Tapir/LLVM differ in more ways than just where they lower parallel constructs. Consequently, to perform an apples-to-apples comparison of these two approaches, we implemented a compiler called "Reference," which is as close to identical to Tapir/LLVM as we could muster, except for where lowering occurs. Figure 2-8 illustrates the compilation pipelines for Clang/LLVM, Tapir/LLVM, and Reference.

The first pipeline, Clang/LLVM, has the traditional three-phase structure. The Clang front-end takes serial C/C++ code and emits LLVM IR. The `-O3` middle-end optimizes the IR, and the CodeGen back-end lowers LLVM IR to machine code for a particular hardware platform.

The second pipeline shows how Tapir/LLVM is organized. The PClang front end takes parallel Cilk Plus code as input and emits Tapir. The middle-end now consists of three steps: `-O3` optimization, a Lower pass to lower Tapir to LLVM IR, and another pass at `-O3` optimization. The first `-O3` pass performs optimizations on the Tapir representation, the lowering pass translates all the Tapir-specific constructs to LLVM IR, and the second `-O3` pass performs optimizations on the LLVM IR. Finally, the CodeGen back end lowers LLVM IR to machine code.

The third pipeline, called Reference, models how mainstream compilers work today, where parallel constructs are transformed into runtime calls before any optimization can take place. The only difference between Reference and Tapir/LLVM is that the Tapir code emitted by the PClang front end is immediately lowered to LLVM IR before the rest of the Tapir pipeline is invoked. (The second Lower pass in the Reference pipeline therefore has no effect.) Although Reference lowers the parallel constructs early, two iterations of `-O3` are included to ensure that the Tapir/LLVM gains no advantage from optimizing twice.

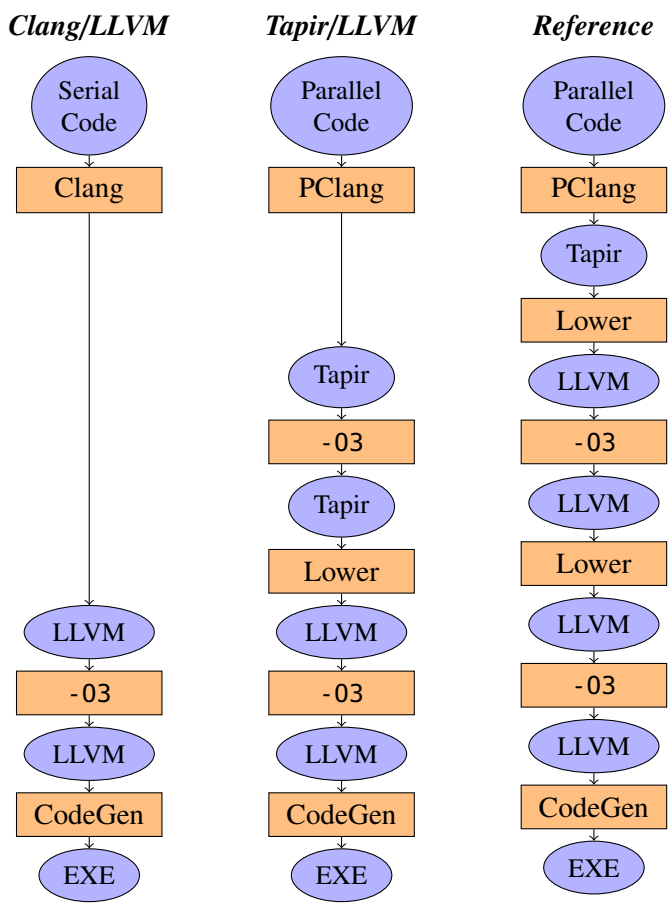


Figure 2-8: The compilation pipelines for Clang/LLVM, Tapir/LLVM, and Reference. Each block represents a compiler transformation, and each oval designates the format of the code at that point in the pipeline.

Although one might think that a second pass of `-O3` would be redundant, it is not. For example, a simple matrix-multiplication code runs 13% faster after two rounds of optimization compared to just one. And although most benchmarks run faster after two `-O3` passes, some actually run slower. Thus, we implemented Reference with the same passes as Tapir/LLVM, except for the initial Lower pass in Reference. This difference only affects parallel code. Serial code passes through both pipelines identically.

Suite	*Benchmark*	*Description*
Cilk	Cholesky	Cholesky decomposition
	FFT	Fast Fourier transform
	NQueens	*n*-Queens solver
	QSort	Hoare quicksort
	RectMul	Rectangular matrix multiplication
	Strassen	Strassen matrix multiplication
Intel	AvgFilter	Averaging filter on an image
	Mandel	Mandelbrot set computation
PBBS	CHull	Convex hull
	detBFS	BFS, deterministic algorithm
	incMIS	MIS, incremental algorithm
	incST	Spanning tree, incremental algorithm
	kdTree	Performance test of a parallel *k*-d tree
	ndBFS	BFS, nondeterministic algorithm
	ndMIS	MIS, nondeterministic algorithm
	ndST	Spanning tree, nondeterministic algorithm
	parallelSF	Spanning-forest computation
	pRange	Compute ranges on a parallel suffix array
	radixSort	Radix sort
	SpMV	Sparse matrix-vector multiplication

Figure 2-9: Descriptions of the 20 benchmarks used to evaluate Tapir/LLVM. These benchmarks were taken from the MIT Cilk benchmark suite [123], Intel Cilk Plus example programs [194], and the CMU Problem-Based Benchmark Suite [356]. "MIS" denotes the computation of a maximal independent set of a graph. "BFS" denotes the breadth-first search of a graph.

2.6.1 Benchmarking

To benchmark the compiler pipelines, we assembled a collection of benchmark programs taken from the MIT Cilk benchmark suite [123], Intel Cilk code samples [194], and the CMU Problem-Based Benchmark Suite [356]. From these collections, we selected stable programs that tend to exhibit little performance difference when the number or order of optimization passes is changed. Figure 2-9 describes the suite of benchmarks tested.

We compiled each program in our benchmark suite with both Tapir/LLVM and Reference, and we ran them on both 1 and 18 cores of our test machine. Additionally, we compiled the serial elision of each benchmark with each compiler. Each running time is the minimum of 10 runs on an Amazon AWS `c4.8xlarge` spot instance, which is a dual-

socket Intel Xeon E5-2666 v3 system with a total of 60 GiB of memory. Each Xeon is a 2.9 GHz 18-core CPU with a shared 25 MiB L3-cache. Each core has a 32 KiB private L1-data-cache and a 256 KiB private L2-cache. The system was "quiesced" to permit careful measurements by turning off Turbo Boost, dvfs, hyperthreading, extraneous interrupts, etc.

2.6.2 Overall performance

The results of our tests are given in Figure 2-10. For the first pair of rows, Reference and Tapir/LLVM produce essentially identical executables when compiling the serial elision of a benchmark. Differences in running times in these rows are due to system noise. The second pair of rows shows that Tapir/LLVM produces executables with better work than Reference on 15 of the benchmarks. Of the remaining 5 benchmarks, 4 demonstrate less than a 1% difference between their work relative to Tapir/LLVM or Reference. The fourth pair of rows elaborates on the results in the second pair to show that Tapir/LLVM produces executables with nearly optimal work efficiency (within 1%) on 12 of the benchmarks, whereas Reference does so on only 2. The third and fifth pairs of row show that Tapir/LLVM generally produces executables with similar or better parallel speedups than those produced by Reference.

The biggest slowdown created from Tapir/LLVM's compilation occurs on Cholesky, for which the executable produced by Tapir/LLVM has 4% more work than that produced by Reference. In investigating this benchmark, we found that LLVM runs a handful of optimizations on each function before the middle-end optimization and lowering passes in either Tapir/LLVM or Reference. Although these early optimizations have little effect on most programs, they reduce the work of the Reference-compiled Cholesky executable by approximately 20%. Although we experimented with several ways to implement lowering in Reference before these early optimizations, the resulting compilers consistently exhibited bugs on other benchmarks in the suite. In our final design for Reference, we placed the initial lowering pass as early as we could muster while still ensuring that Reference could compile all benchmarks correctly.

2.7 Related work

Various prior art explores compiler optimizations on unstructured parallel threads. For example, some researchers have explored how to find and remove unnecessary synchronization in Java programs [12, 330]. Joisha *et al*. [198] present a technique to detect instructions that are unaffected by parallel threads and can be safely optimized across unstructured parallel control flow. In contrast, our work on Tapir focuses on compiler optimizations for structured parallelism, namely fork-join parallel programs with serial semantics. Although fork-join parallelism may be more restricted than unstructured parallel threads, Tapir demonstrates that many of the optimizations for serial code easily extend to fork-join parallelism. Enabling similar optimizations for unstructured parallel threads appears to be a much harder problem.

Some previous work on compiler optimizations for fork-join parallel programs evaluate which instructions can safely execute in parallel [6] based on concurrency mechanisms supported by a particular memory model. For example, Barik *et al.* [26, 27] use interprocedural analysis to perform various optimizations affecting critical sections of X10 and Habanero-Java programs. Rather than dealing with the complexities of general concurrency mechanisms, Tapir enables compiler optimizations for an easy-to-understand situation: when the optimization respects the serial semantics of the program and does not introduce determinacy races. Compared with general concurrency mechanisms, well-structured parallelism seems to offer a less onerous path to performance.

Khaldi *et al*. [212] modify LLVM IR to support OpenSHMEM parallel programs with the aim of achieving performance in modern network interconnects that support efficient data transfers for partitioned global address spaces (PGAS). Based on the SPIRE methodology [211] for representing parallel code, they augment functions, basic blocks, instructions, identifiers, and types in LLVM IR with execution, synchronization, scheduling, and memory-layout information. In contrast, Tapir models fork-join parallelism for shared-memory multicores, a conceptually simpler context than PGAS systems, and extends LLVM IR minimally using only three instructions. Once again, the Tapir's strong assumption of a fork-join programming model with serial semantics that compiles to a flexible multicore ar-

chitecture seems to provide both performance and simplicity, albeit at the cost of scalability to huge cluster-based supercomputers that lack strong memory-consistency guarantees.

In contrast with much of the work referenced above, Chatarasi *et al*. [67] focus, as Tapir does, on fork-join programs with serial semantics. Specifically, they examine polyhedral optimizations on OpenMP programs with serial semantics. By combining dependency and happens-before analyses, they manage to enable traditional polyhedral optimizers to work on parallel loops, much as Tapir enables common middle-end compiler optimizations to work on parallel code.

2.8 Conclusion

To conclude, I would like to leave the reader with three interesting considerations regarding the nature of asymmetry in parallelism, the future of parallel optimizations, and extensions of Tapir-like systems to other models of parallel programming.

Reasoning about logically parallel tasks asymmetrically based on serial semantics can sometimes simplify the understanding of a parallel program's behavior. When a task is spawned to execute in parallel with another, it is natural to reason about the logically parallel tasks as symmetric, because their instructions can execute in any relative order. For parallel programs with serial semantics, however, it is always valid to execute the program on a single processor, which asymmetrically executes one parallel task to completion before starting the other. Serial semantics encourage an asymmetric representation of parallel control flow that is similar enough to its serial elision that most common analyses and transformations for serial programs work on parallel constructs with little or no modification. In particular, serial semantics enables common optimizations on parallel code that can be invalid under other models of parallelism [393].

One of the great benefits of Tapir is that its strategy for representing parallelism makes it easy to write optimization passes specifically for parallel code. Section 2.4 briefly mentioned some parallel optimization passes we implemented, including parallel-loop scheduling and unnecessary-sync elimination. In addition to helping close the performance gap between serial and parallel versions of code, we hope that the introduction of Tapir will en-

courage the development and implementation of many more parallel-optimization passes.

Finally, Tapir allows fork-join parallel programs to benefit from both serial and parallel optimizations. Moving forwards, it is natural to wonder whether other models of parallelism, such as pipeline parallelism [232, 284, 103] or data-graph computations [252, 251, 256, 286, 287, 355, 357], can take advantage of the Tapir approach.

		Cholesky	FFT	NQueens	QSort	RectMul	Strasssen	AvgFilter
T_S	Ref.	2.935	10.304	3.084	4.983	10.207	10.105	1.751
	Tapir	2.933	10.271	3.083	4.984	10.207	10.119	1.750
T_1	Ref.	6.581	10.413	10.196	2.355	30.520	1.316	6.596
	Tapir	6.461	10.415	10.196	1.730	25.774	1.187	5.673
T_{18}	Ref.	0.648	0.609	1.106	0.708	1.847	0.124	0.517
	Tapir	0.709	0.611	1.124	0.615	1.559	0.120	0.467
$\frac{T_S}{T_1}$	Ref.	0.757	0.980	0.991	0.743	0.845	0.710	0.801
	Tapir	0.771	0.980	0.991	1.012	1.000	0.788	0.992
$\frac{T_S}{T_{18}}$	Ref.	7.690	16.760	9.137	2.472	13.957	7.540	9.518
	Tapir	7.028	16.705	8.990	2.846	16.536	7.792	10.942
		Mandel	CHull	detBFS	incMIS	incST	kdTree	ndBFS
T_S	Ref.	25.779	0.938	5.670	4.993	4.190	5.473	3.950
	Tapir	25.780	0.935	5.666	5.006	4.173	5.466	3.956
T_1	Ref.	4.572	11.919	3.409	6.030	4.733	5.640	4.930
	Tapir	4.739	11.733	3.419	5.043	4.203	5.546	3.980
T_{18}	Ref.	0.387	0.788	0.196	0.559	0.352	0.342	0.415
	Tapir	0.396	0.774	0.197	0.527	0.329	0.339	0.361
$\frac{T_S}{T_1}$	Ref.	0.642	0.862	0.904	0.828	0.882	0.969	0.801
	Tapir	0.619	0.875	0.902	0.990	0.993	0.986	0.992
$\frac{T_S}{T_{18}}$	Ref.	7.579	13.034	15.730	8.932	11.855	15.982	9.518
	Tapir	7.407	13.270	15.650	9.474	12.684	16.124	10.942
		ndBFS	ndMIS	ndST	parallelSF	pRange	radixSort	SpMV
T_S	Ref.	3.950	9.210	4.069	5.136	2.564	3.775	1.780
	Tapir	3.956	9.253	4.053	5.136	2.559	3.775	1.783
T_1	Ref.	4.930	10.760	4.286	5.646	3.438	3.795	1.836
	Tapir	3.980	9.246	4.063	5.183	3.083	3.800	1.786
T_{18}	Ref.	0.415	0.774	1.925	0.414	0.348	0.284	0.118
	Tapir	0.361	0.701	1.692	0.392	0.330	0.285	0.112
$\frac{T_S}{T_1}$	Ref.	0.801	0.856	0.946	0.910	0.744	0.995	0.969
	Tapir	0.992	0.996	0.998	0.991	0.830	0.993	0.997
$\frac{T_S}{T_{18}}$	Ref.	9.518	11.899	2.105	12.406	7.353	13.292	15.085
	Tapir	10.942	13.138	2.395	13.102	7.755	13.246	15.893

Figure 2-10: Comparison between executables compiled using Reference and using Tapir/LLVM. Each column refers to a different parallel benchmark described in Figure 2-9. Rows labeled "Ref." describe executables compiled using Reference, and rows labeled "Tapir" describe executables compiled using Tapir/LLVM. Each measured running time is the minimum over 10 executions, measured in seconds. The pair of rows labeled T_S gives the running time of the executable compiled from the serial elision of each benchmark. The pair of rows labeled T_1 gives the work of each benchmark. The pair of rows labeled T_{18} gives the 18-core running time of each benchmark. The pair of rows labeled T_S/T_1 gives the work efficiency of each compiled benchmark, derived from the first and second pairs of rows. The pair of rows labeled T_S/T_{18} gives the parallel speedup of each compiled executable on 18 cores, derived from the first and third pairs of rows.

Chapter 3

Tensor Comprehensions

3.1 Introduction

Deep neural networks trained with back-propagation learning [231] are a method of choice to solve complex problems with sufficient data. Popular graph computation engines [380, 80, 73, 2, 297] offer high-level abstractions for optimizing and executing deep neural networks expressed as graphs of tensor operations. These frameworks make transparent use of heterogeneous computing systems, leveraging highly-optimized routines for individual operators. While these operators are sufficient for many applications, they fall short in a number of instances. Developing a novel type of layer or network architecture incurs high engineering cost or performance penalty. Even if a new layer may be expressed in terms of existing library primitives, performance is often far from peak for two reasons: missed optimizations across operators, and no tuning for its specific size, shape and data flow [395]. Our work aims at addressing this *productivity gap*.

In parallel to the software problem, a hardware race has begun, fueled by the needs for energy-efficient computing. With Google's TPU [201] and Microsoft's Brainwave project [261] on the bleeding edge, many large tech companies are pursuing their own hardware. At Google I/O 2018, Turing-award recipient John Hennessy called for fully rethinking our hardware, compilers and language support for domain-specific properties [168],

citing orders of magnitude speedup opportunities and power constraints caused by the advent of dark silicon [109].

With the increasing problem complexity and hardware limitations, growing the size of manually optimized libraries will not scale to future demands. To address these challenges, we present a novel *domain-specific flow* capable of generating highly-optimized kernels for tensor expressions. It leverages optimizations across operators and takes into account the size and shape of data. The polyhedral framework of compilation emerged as a natural candidate to design a versatile optimization flow satisfying the needs of the domain and target hardware. It has demonstrated strong results in domain-specific optimization [277, 32, 22, 107], expert-driven meta-programming [136, 71, 24], embedding of third-party library code [221], and automatic generation of efficient code for heterogeneous targets [28, 260, 310, 402, 22, 430]. We attempt to take the best of both worlds, defining a domain-specific language rich enough to capture full sub-graphs of modern Machine Learning (ML) models, while enabling aggressive compilation competitive to native libraries. In doing so, we may temporarily sacrifice some of the performance of über-optimized large matrix multiplications (e.g., compared to the recent Diesel polyhedral compiler [107]) while providing full automation and ML framework integration. Note that there is no fundamental difficulty in combining both approaches, recognizing and linking external library kernels when appropriate, as illustrated in subsection 3.3.7.

Our contributions are the following:

1. the Tensor Comprehensions (TC) Domain-Specific Language (DSL) with a tensor notation close to the mathematics of deep learning, with an emphasis on improving productivity while maintaining a direct lowering path to the intermediate representation of a parallelizing compiler for GPU acceleration;

2. an intermediate representation and Just-In-Time optimizing compiler based on the polyhedral framework, enabling complex program transformations and levels of automation unmatched by any other compiler for the acceleration of computational sub-graphs of neural networks;

3. coordinated optimization algorithms with integrated functional correctness, prof-

itability modeling, domain and target specialization; we propose a layered approach, relying on integer linear programming and other polyhedral algorithms to address the core program optimization and synthesis challenges, while resorting to evolutionary algorithms as a higher level of control, to select high level strategies and fine-tune transformation parameters;

4. the transparent integration of our flow into PyTorch [297] and Caffe2 [141], providing the fully automatic synthesis of high-performance GPU kernels from simple tensor algebra.

The TC flow is also portable to other ML frameworks with a few lines of code. While our initial implementation focuses on Nvidia GPUs, the core technology applies to other types of accelerators with shared or partitioned memory [260, 310, 402, 425]; these include vector and SIMD accelerators, and also the generation of computational patterns suitable for ASICs with systolic designs and efficient storage management involving non-volatile memory technologies.

3.2 Tensor Comprehensions

Tensor Comprehensions (TC) are an algorithmic notation for computing on multi-dimensional arrays. It borrows from the Einstein notation, a.k.a. summation convention: (1) index variables are defined implicitly and their range is inferred from what they index; (2) indices that only appear on the right hand side of a statement are assumed to be reduction dimensions; (3) the evaluation order of points in the iteration space does not affect the output.

A *tensor comprehension function*, or tensor comprehension for short, defines output tensors from *pointwise* and *reduction* operations over input tensors. These operations are defined declaratively as a sequence of pointwise equations or reductions, called *tensor comprehensions statements*, or statements for short.

Let us consider matrix-vector product as a simple example of a tensor comprehension with two statements:

```
def mv(float(M,K) A, float(K) x) → (C) {
  C(i)  = 0
  C(i) += A(i,k) * x(k)
}
```

This defines the function `mv` with `A` and `x` as input tensors and `C` as an output. The shapes of `A` and `X` are of size (M, K) and (K), respectively. The shape of `C` is inferred automatically. The statements introduce two indices '`i`' and '`k`'. Variables not defined in the function signature implicitly become indices. Their range is inferred based on how they are used in indexing (see Section 3.3.1); here we will discover `i` $\in [0, M)$, and `k` $\in [0, K)$. Because `k` only appears on the right-hand side, stores into `C` will *reduce* over `k` with the reduction operator `+`.

Intuitively, a tensor comprehension may be thought of as the *body* of a loop whose control flow is inferred from context. The equivalent C-style pseudo-code is:

```
tensor C({M}).zero(); // 0-filled single-dim tensor
parallel for (int i = 0; i < M; i++)
  reduction for (int k = 0; k < K; k++)
    C(i) += A(i,k) * x(k);
```

Importantly, the nesting order (`i` then `k`) is arbitrary: the semantics of a tensor comprehension is always invariant to loop permutation.[1] TC allows in-place updates while preserving a functional semantics that is atomic on full tensors: *RHS expressions are read in full before assigning any element on the LHS*. This specification is important in case the LHS tensor also occurs in the RHS [121]: the compiler is responsible for checking the causality of in-place updates on element-wise dependences, currently allowing only pointwise updates. Also, to enable in-place updates across TC functions, outputs of a TC statement can also be used as inputs.

We provide a short-cut for an *initializing reduction*, where the result is initialized to the operator's neutral element before reduction by appending '`!`' to the operator, e.g., '`+=!`' instead of '`+=`'. A one-line definition of the matrix-vector product **mv** is given below; and common ML kernels can be written in just a few lines, such as the **sgemm** function from

[1] Nested reductions over multiple variables are supported as long as they involve a single reduction operator, as commutation does not hold across reduction operators, e.g., $\min(\max(f(.))) \neq \max(\min(f(.)))$.

BLAS:

```
def mv(float(M,K) A, float(K) x) -> (C) {
  C(i) +=! A(i,k) * x(k)
}
```

```
def sgemm(float a, float b, float(N,M) A, float(M,K) B) -> (C) {
  C(i,j)  = b * C(i,j)           # initialization
  C(i,j) += a * A(i,k) * B(k,j)  # accumulation
}
```

Expressing general tensor contractions is equally easy. A fully connected layer followed by a rectified linear unit takes the form of a transposed matrix multiplication initialized to a broadcast bias term followed by pointwise clamping (applying the builtin scalar function `fmaxf` with 0):

```
def fcrelu(float(B,I) in, float(O,I) weight, float(O) bias) -> (out) {
  out(b,o)  = bias(o) where b in 0:B
  out(b,o) += in(b,i) * weight(o,i)
  out(b,o)  = fmaxf(out(b,o), 0)
}
```

The `where` annotation informs the inference algorithm of the intended index variable ranges when they cannot be unambiguously inferred. In this case, '`b`' indexes only '`out`' whose size *also* needs to be inferred. Unlike tensor kernel libraries with predefined layout conventions, notice that TC lets the user control data layout through the order of tensor indexing dimensions. Here we chose to reuse the `out` tensor across all comprehensions, indicating the absence of temporary storage.

Similarly, the `where` clause serves to indicate ranges of `kh` and `kw` in the max pooling layer, which would otherwise be under-constrained:

```
def maxpool2x2(float(B,C,H,W) in) -> (out) {
  out(b,c,i,j) max=! in(b,c, 2 * i + kh, 2 * j + kw) where kh in 0:2, kw in 0:2
}
```

A 2-D convolution is also simple. Its reduction is initialized to 0 (note the use of `+=!`) with

reduction dimensions kh, kw:

```
def conv2d(float(B,IP,H,W) in, float(OP,IP,KH,KW) weight) → (out) {
  out(b,op,h,w) +=! in(b,ip, h + kh, w + kw) * weight(op,ip,kh,kw)
}
```

Subscript expressions can be any affine function of iterators, or subscript-of-subscript expressions (a tensor element indexing another), and combinations thereof. The latter capture data-dependent accesses such as a gather operation:

```
def gather(float(N) X, int(A,B) I) → (Z) {
  Z(i,j) = X(I(i,j))
}
```

TC algorithmic notation differs from today's prominent frameworks where most operators are defined as black-box functions. The design of TC makes it easy to experiment with small layer variations while preserving a concise, in-place expression. Thus, a strided convolution is easily created as a tweak on convolution, e.g., strided by 2 along h and 3 along w is:

```
def sconv2d(float(N,C,H,W) I, float(F,C,KH,KW) W, float(F) B) → (O) {
  O(n,f,h,w) +=! I(n,c, 2 * h + kh, 3 * w + kw) * W(f,c,kh,kw)
  O(n,f,h,w) +=  B(f)
}
```

Figure 3-1 shows the grammar of the Tensor Comprehension language in EBNF notation.

3.2.1 Data Layout

TC makes data layout explicit and easy to reason about. It supports generalized tensor transpositions (i.e., applying an n-D permutation matrix where $n > 2$), and data tiling can be achieved by reshaping tensors and adjusting the index expressions. Range inference and checking guarantees such reshaping will always be consistent throughout the statements of a tensor comprehension. For instance, *NCHW* convolution operates on an explicit in-

```
num ::= <decimal number literal>
id ::= <C identifier>
binop ::= '+' | '-' | '*' | '/' | '=' | '≠' | ...
exp ::= num
  | ( '-' | '!' ) exp
  | exp binop exp
  | exp '?' exp ':' exp
  | id '.' num    # range of num-th dimension of id
  | id '(' exp_list ')'     # call or tensor access

reduction ::= '+='  | '*='  | 'min='  | 'max='
            | '+=!' | '*=!' | 'min=!' | 'max=!'

range_constraint ::= id '=' exp '..' exp
                   | id '=' exp

stmt ::= id '(' id_list ')' ( '=' | reduction )
         [ 'where' range_constraint_list ]
  | id_list = id '('id_list ')' # TC function call
```

```
arg ::= type id
return ::= id # inferred return type and range

scalar_type ::= 'double' | 'float' | 'half'
              | 'int' | 'byte' | 'uint32' | ...
type ::= scalar_type [ '(' id_list ')' ]

func ::= # TC function definition
  'def' id '(' arg_list ')' '↪' '(' return_list ')' '{'
    stmt_list
  '}'

id_list ::= <comma separated id list>
exp_list ::= <comma separated exp list>
arg_list ::= <comma separated arg list>
stmt_list ::= <whitespace separated stmt list>
return_list ::= <comma separated return list>
range_constraint_list ::= <non-empty comma separated
                           range_constraint list>
```

Figure 3-1: Simplified EBNF syntax for core TC. Parentheses denote inline alternatives, brackets denote optional clauses, angle brackets contain textual descriptions used for simplicity.

put, declared as `float I(N,C,H,W)`, with the layout matching the expected row-major semantics.

In addition, the TC compiler may transparently apply layout transformations, e.g., when mapping tensor tiles to GPU shared memory.

3.2.2 Automatic Differentiation

While TC does not natively deal with automatic differentiation (AD), backward passes can readily be implemented in TC as a few lines of code. Below is the backward pass of matrix multiplication. Native automatic differentiation support for TC can be enabled by integrating the Enzyme AD [271, 272, 273] tool into the TC pipeline. See Chapters 5, 7, and 8 for more details.

```
def matmul_grad(float(M,N) A, float(N,K) B, float(M,K) d_O) →
(d_A,d_B) {
  d_A(m,n) +=! d_O(m,r_k) * B(n,r_k)
  d_B(n,k) +=! d_O(r_m,k) * A(r_m,n)
}
```

3.3 Tensor Comprehensions Workflow

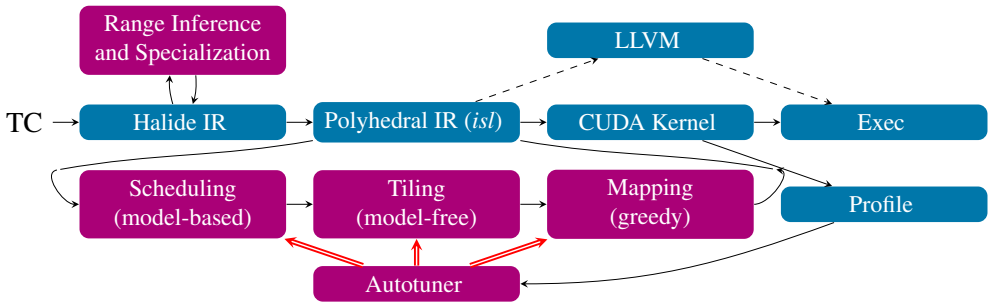


Figure 3-2: The JIT compilation flow lowers TC to Halide-IR, then to Polyhedral-IR, followed by optimization, code generation and execution

The Tensor Comprehensions workflow consists of several stages, progressively lowering the level of abstraction (Figure 3-2). Given a TC with specialized tensor sizes and strides,[2] we lower it to a parametric Halide-IR expression, which is further lowered to a polyhedral representation where most transformations are applied. The output of the polyhedral flow is CUDA code that can be further JIT-compiled with NVRTC and executed. Complementing this flow, an autotuner and serializable compilation engine interacts with scheduling and mapping strategies to search the optimization space.

Much of TC's versatility and effectiveness resides in its embedding of a polyhedral compiler as the main optimization engine. The polyhedral framework is an algebraic representation of "sufficiently regular" program parts, covering arithmetic expressions on arrays surrounded by static control flow [113]. It has been a cornerstone of loop optimization in the last three decades [195, 112, 15, 29, 52, 402] and is integrated into production compilers [388, 260, 149, 51]. Despite its deceiving apparent simplicity, it covers a large class of computationally-intensive kernels. It is parametric on loop bounds and array sizes, and captures more transformations of the control and data flow than domain-specific representations such as Halide [319] or TVM [74]. The use of the polyhedral model by TC is derived from that of PPCG [402] and this section only provides a general overview. Our transformation engine comprises the following, specially adapted or algorithmically novel components:

[2]Our toolchain supports parametric specifications, yet we have found early specialization to be beneficial in driving profitability decisions during polyhedral scheduling.

1. Range inference and lowering from high-level TC abstraction to the polyhedral representation;

2. Core affine scheduling adapted from *isl* which automatically optimizes for (outer) loop parallelism and locality, tuned towards folding a complete TC function into a single GPU kernel;

3. The schedule is further tiled to facilitate the mapping and temporal reuse on the deep parallelism and memory hierarchy of GPUs [405];

4. Mapping to GPUs borrows from PPCG [402] with extensions to support the more complex and imperfectly nested control structures of ML kernels;

5. Memory promotion deals with explicit data transfers to and from shared and private memory.

This work demonstrates that the polyhedral framework is particularly well suited for deep neural networks, featuring large and deeply nested loops with long dependence chains and non-uniform or all-to-all patterns—arising from fully connected layers and tensor contractions, and transpositions. These features push the optimization problem into a different heuristic space than Halide's for image processing, and a wider space than linear algebra alone.

3.3.1 Range Inference

TC loops are implicit and output tensor sizes are inferred from index ranges, which themselves may also be inferred. Our algorithm infers the largest rectangular ranges that avoid out-of-bounds reads on inputs. A `where` clause allows for disambiguation if multiple such ranges exist.

Consider the `conv2d` kernel on page 68. The sizes of the input tensors, `in` and `weight`, are known from the function signature. The algorithm needs to infer the ranges of the iterators, and the size of the output tensor `out`. The iterators `b`, `op`, `kh`, `kw` appear only once on the RHS and their ranges are therefore $[0, \texttt{B})$, $[0, \texttt{OP})$, $[0, \texttt{KH})$, $[0, \texttt{KW})$ so that they index the input tensors maximally. The iterator `ip` appears twice, but indexes the dimension of

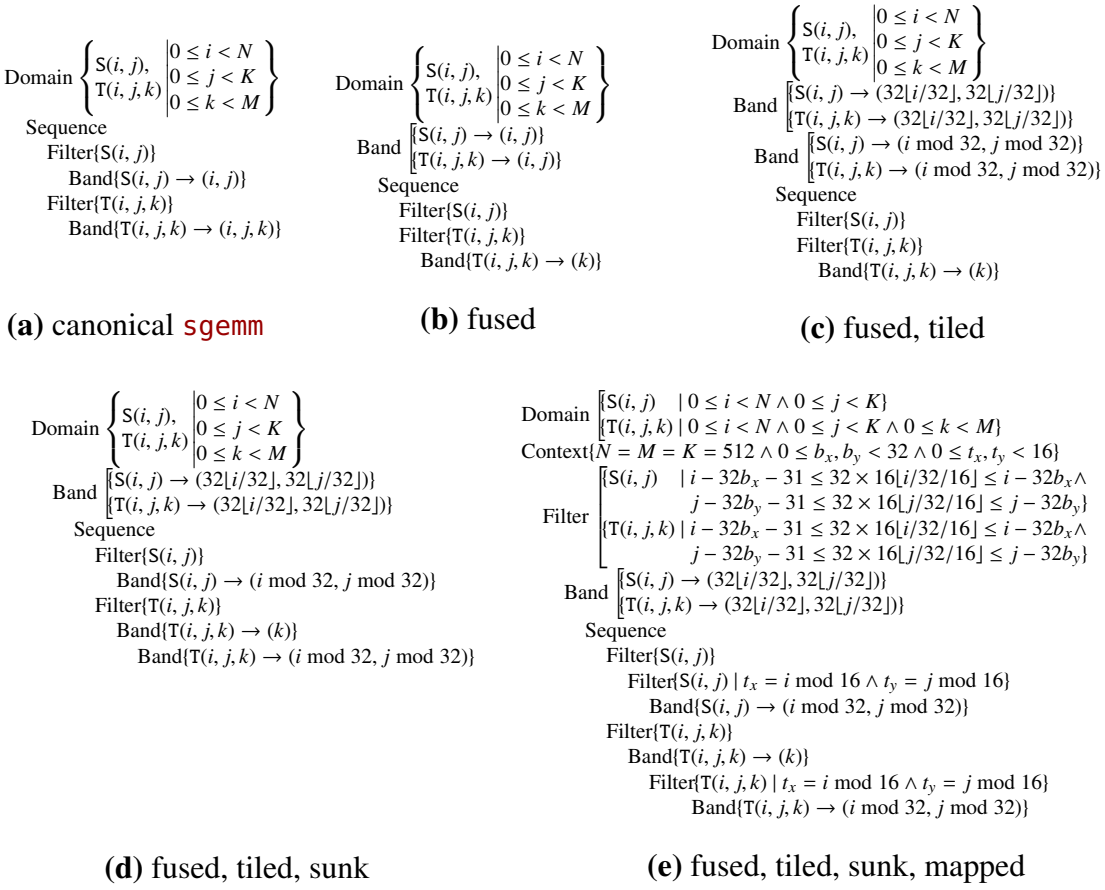
the same size, so its range is $[0, \texttt{IP})$. Had it been indexing dimensions of different sizes, its range would have been the intersection of all size-imposed ranges. Once the ranges of `kh` and `kw` are known, it is possible to infer those of `h` and `w`: we require $\texttt{h} + \texttt{kh} \leq \texttt{H}$ and $\texttt{w} + \texttt{kw} \leq \texttt{W}$, which leads to the maximal ranges of $[0, \texttt{H} - \texttt{KH})$ and $[0, \texttt{W} - \texttt{KW})$ respectively. Finally, the size of `out` can be inferred given the ranges of the iterators that index it, yielding `float(B,OP,H-KW,W-KW)`. The user of TC is able to inspect the symbolic sizes inferred for the output tensors using a command-line flag.

Consider now a typical stencil operation `A(i) += B(i + k) * K(k)`: there are multiple ways to maximize the ranges of `i` and `k`. To disambiguate without annotations, range inference proceeds in rounds. It maintains a set of index variables whose ranges are not yet resolved. Initially, it contains all variables not in any `where` clause. Each step considers argument expressions that contain a single unresolved variable and constructs a boolean condition stating the accesses are within bounds. Using Halide [319] mechanisms, range inference computes the maximal range that satisfies this condition given the already known ranges of other variables. If different ranges are computed for the same variable, they are then intersected. For the stencil above, in the first round we ignore the expression `B(i + k)` because it contains multiple unresolved variables. We use `K(k)` to deduce a range for `k`. In the second round, `B(i + k)` contains a single unresolved variable, and we use the already-inferred range of `k` to deduce a maximal range for `i`.

3.3.2 Lowering to the Polyhedral Representation

The role of lowering is to bridge the impedance mismatch between the logical layout of high level tensor operations (dimension ordering) and the data format the polyhedral code generator expects (C-style row-major arrays). It ensures the absence of aliasing and performs range inference for output tensors. Based on range inference, TC differs from NumPy-style implicit "broadcast" semantics (non-trivial tensor dimensionality extension) adopted by XLA, PyTorch and MXNet.

Our representation derives from schedule trees [404], implemented in the *isl* library [400], and uses a set of node types. Each TC-statement corresponds to multiple runtime statement



(a) canonical `sgemm` **(b)** fused **(c)** fused, tiled

(d) fused, tiled, sunk **(e)** fused, tiled, sunk, mapped

Figure 3-3: Optimization steps for `sgemm`

instances, one for every valuation of the index variables. The root *domain node* defines the set of statement instances to be executed. Due to the nature of the TC-language, the constraints on the index variables are always affine, resulting in an exact representation of the set of operations. A *band node* defines a *partial* execution order through one or multiple piecewise affine functions defined over iteration domains. The name refers to the notion of a *permutable schedule band*, a tuple of one-dimensional schedule functions that can be freely interchanged while preserving the semantics of the program. A *filter node* partitions the iteration space, binding its sub-tree to a subset of the iteration domain. It can be arranged into *set or sequence nodes* depending on whether or not the order of execution must be serialized. *Context nodes* provide additional information on the parameters, e.g., tensor extents or GPU grid/block sizes. Finally, *extension nodes* introduce auxiliary

computations that are not part of the original iteration domain, which is useful for, e.g., introducing data-copy statements.

A *canonical* schedule tree for a TC is defined by an outer *sequence* node, followed by *filter* nodes for each TC statement. Inside each filtered branch, *band* nodes define an identity schedule with as many one-dimensional schedule functions as loop iterators for the statement. The implicit loops form a permutable band as per TC semantics.

In addition to the schedule tree, our representation includes tensor access functions, which map the index variables to the subscripts of tensors they access. These subscripts are not necessarily affine, in which case over-approximations are used [38]: a non-affine access is assumed to potentially access *all* values along the given dimension. After the polyhedral representation is constructed, dependence analysis can be used to ensure the absence of out-of-bounds accesses [313].

Additional lowering steps include forward substitution of convolution expressions (storage/computation trade-off), padding, mirroring and clipping. The process is analogous to Halide's [319].

Example Figure 3-3(a) shows the canonical schedule tree for unions of relations where tuples of iterators are guarded with syntactic identifiers [313].[3] for the `sgemm` TC defined on Page 67. One recognizes a 2-D nest from the initialization statement followed by a 3-D nest for the update statement. The schedule can be either parametric in input sizes, or have extra context information on the tensor sizes. In cases where *band* nodes do not define an injective schedule, the statement instances are scheduled following the lexicographical order of their domain coordinates.

3.3.3 Tunable Polyhedral Scheduling

Program transformation in the polyhedral model involves defining a different schedule, which corresponds to a different (partial or total) order of traversing the iteration domain. The instances of all statements are scheduled completely automatically [52] using one of several scheduling strategies with which we extended the *isl* scheduler [405].

[3] We use the *named relation notation* of `iscc` [401]. The declaration of parameters $(N, M, K) \rightarrow \{\dots\}$ is omitted hereinafter for brevity.

The *isl* scheduler iteratively solves integer linear programming problems to compute piece-wise affine functions that form new schedule *band* nodes. Internally, it operates on a data dependence graph where nodes correspond to statements and edges express dependences between them. It introduces the *affine clustering* technique that is based on computing the schedule bands separately for individual strongly-connected components of the dependence graph and then clustering these components iteratively and scheduling them with respect to each other. Clustering not only decreases the size of the linear problems the scheduler has to solve, but also serves as a basis for *isl*'s loop fusion heuristic.

We extended *isl* to provide finer-grained control over the scheduling process. For affine transformations, the user can set additional scheduling options. For clustering, the user can supply a decision function for pairwise dependence graph component combination, after this combination was demonstrated to be valid by the scheduler. These configuration points serve as a basis for both fixed scheduling choices made by TC and *scheduling strategies*. In particular, TC tells the scheduler to produce schedules with only non-negative coefficients and without any skewing. Clustering decisions allow TC to control the conventional minimum and maximum fusion targets, and additionally, maximum fusion that preserves at least three nested parallel loops (to be mapped to CUDA blocks and threads). With the scheduling strategies one may optionally enable point band rescheduling (i.e., scheduling the inner dimensions after tiling). In particular, two fusion strategies can be specified, one for the global schedule and one for the point band. If these fusion strategies are different, then the point band (along with all its descendants) is rescheduled after tiling, preserving only the outer tile band of the original schedule. Scheduling strategies can be selected through the autotuning process. In all cases, we enforce that a single GPU kernel is generated.

Example Observing that the `C` tensor in `sgemm` (see Page 67) is reused between two nests, the scheduler constructs the tree in Figure 3-3(b) to leverage access locality and improve performance. This tree features an outer band node with `i` and `j` loops that became common to both statements, which corresponds to *loop fusion*. The sequence node ensures that instances of `S` are executed before respective instances of `T` enabling proper initialization. The second band is only applicable to `T` and corresponds to the innermost (reduction) loop `k`.

Overall, the tuning process is greatly simplified compared to Halide and TVM. Relying on a heavy-duty, well-understood analytical optimization framework based on integer linear programming, TC exposes a small, dedicated search space of high-level strategies and block size parameters. Beyond guaranteeing the validity of the transformation, dependences can be used to explore parallelization opportunities (independent instances can be executed in parallel), to improve data access locality (dependent instances executed close in time) or to automate vectorization [405, 52, 430, 396, 308].

3.3.4 Imperfectly Nested Loop Tiling

Let us first describe the general setting for loop tiling on schedule trees, before developing the TC-specific specialization and extensions.

Tiling permutable bands Pluto has been very successful at decoupling the actual implementation of loop tiling from the preparation of an affine schedule exposing permutable loops amenable to tiling [52]. This design allows exploring locality and parallelization tradeoffs without bloating the schedule representation with complex quasi-affine forms capturing the precise distribution of iterations into tile and point loops. Schedule trees ease the implementation of such a decoupled design, capturing tiling as the conversion of a permutable schedule band into a chain of two bands, with the outer band containing tile loops and the inner band containing point loops with fixed trip count. This can be seen as a conventional strip-mine and sink transformation.

In addition to conventional loop tiling, the schedule tree representation allows tiling imperfectly nested loops. The technique is based on the following observation: if a loop does not carry dependences, it can be sunk below any other loop. In valid schedules, all dependences are carried (or satisfied) by some loop, along which they feature a positive distance. A dependence is only violated if it has a negative distance along some loop *before* it is carried by another loop [208]. Parallel loops do not carry dependences by definition and therefore do not affect dependence satisfaction or violation. Therefore, imperfectly nested tiling may be implemented by first tiling bands in isolation and then sinking parallel point loops in the tree. During this process, the point band is replicated in each sub-tree

below a sequence (or set) node and its schedule is restricted to only map the relevant points in the iteration domain. Such an extension is particularly helpful in Pluto, where bands of permutable loops are rediscovered through a post-pass traversal of the affine schedule.

Parallelism and locality trade-offs TC applies two tiling schemes with complementary purposes.

The first one takes place immediately after affine scheduling. It aims at exposing a sufficient number of parallel dimensions, some of which amenable to memory coalescing, and some better suited to block-level parallelism. It also aims at exploiting data locality within thread blocks (through shared memory) and individual threads (through register reuse). This tiling scheme is influenced by the strong emphasis on loop fusion in the affine scheduling heuristic (to enforce that the generated code runs as a single GPU kernel). In this context, conventional loop nest tiling—considering a single band at a time—appears to be sufficient. This is the hypothesis we make in this chapter.[4]

The second tiling scheme takes place in the block and thread mapping algorithm, which is the topic of the next sub-section.

Example Figure 3-3(c) shows the schedule tree for the fused and tiled `sgemm`. It purposely has two imperfectly nested bands. Dependence analysis shows that loops `i` and `j` are parallel. Therefore, we can tile them and sink the point loops below the band of the reduction `k` loop, resulting in the schedule tree in Figure 3-3(d). Innermost nested bands with point loops can be joined together into a single band after checking for permutability. As indicated earlier, TC implements the fusion and tiling scheme of Figure 3-3(c) but not the sunk, imperfect scheme of Figure 3-3(d).

3.3.5 Mapping to Blocks and Threads

A schedule tree can also be used to represent the *mapping* to an accelerator, in particular a GPU with multiple blocks and threads. This operation is performed by associating certain schedule band members, and the corresponding loops, to thread or block indices. The polyhedral code generator then omits the loops, if possible, and rewrites the index expressions

[4]The TC implementation supporting our experiments does not implement imperfect loop tiling after affine scheduling.

accordingly. Building on PPCG, our mapping approach is decoupled from tiling for data locality: grid and block sizes are specified independently from tile sizes and are exposed as tunable parameters. Due to the semantics of blocks and threads, only parallel loops that belong to a permutable schedule band can be mapped. If point loops are mapped to threads, the ratio between tile sizes and blocks sizes controls the number of iterations executed by each thread. Note that tile sizes smaller than the block sizes lead to some threads not performing any computation.

Contrary to PPCG, which may generate multiple kernels for a given input program, our mapping approach handles imperfectly nested loops in a way that generates a single kernel as expected by ML frameworks. We require the schedule tree to have at least an outermost band with outer parallel dimensions. The parallel dimensions of the (single) outermost band are mapped to GPU blocks. In each schedule tree branch, the innermost permutable band, typically consisting of point loops, is mapped to GPU threads with the following restrictions: the number of mapped dimensions must be equal across branches, and on each branch, there must be exactly one band mapped to threads. The mapping is performed bottom-up, first attempting to map the leaf bands to threads, before moving to a parent band only if none of the children could be mapped to threads.

Thread mapping can be extended to imperfectly nested loops, following the same principle as imperfect loop tiling. Within a given thread block, one may sink parallel point loops so that multiple bands in a sequence (or set) may be equalized in depth and mapped together. However, TC currently does not perform any such sinking.

Example Our mapping strategy produces the schedule tree in Figure 3-3(e). We introduced a context node in the schedule tree to indicate the effective sizes of the parameters as well as the grid and block sizes (denoted as b_x, b_y and t_x, t_y, respectively, standing for the values eventually taken by `blockIdx.x`, `blockIdx.x` and `threadIdx.x`, `threadIdx.y`). This insertion is performed just-in-time, when the effective tensor sizes are known. Also notice the Filter nodes referring to the b_x, b_y, t_x and t_y parameters: these nodes express the *mapping* to the GPU.

3.3.6 Memory Promotion

We are interested in promoting parts of tensors into shared or private GPU memory. While the promotion decision is taken by a heuristic and the corresponding imperative code is generated at a later stage, schedule trees offer a convenient interface for attaching memory-related information. Memory promotion is based on the notion of an *array tile*, a form of data tiling for software-controlled local memories. It is a constant-size potentially strided block in the array that covers all elements accessed by within a given (schedule) tile. We build upon and extend PPCG's support for memory promotion [402, 405] and expose the promotion to shared and private memory as boolean options for the autotuner.

Promotion of Indirectly Accessed Arrays. Memory promotion is also applicable to indirectly accessed arrays. These frequently occur when modeling variable length data through *embedding layers* such as word embeddings in natural language processing. This is particularly important in the case of latency-bound benchmarks where there is little computational or additional data processing work to hide global memory latency. Indirect arrays used to be promoted in the initial TC implementation based on PPCG. When implementing parallel reductions, working towards the first released version of TC, we realized that parallelizing reductions was sufficient to deliver comparable or higher speedups in our word embedding benchmarks. For this reason, indirect array promotion was dropped from the publicly available version of TC. We still report on the design for it remains interesting to describe how the polyhedral TC flow may optimize non-affine data flow.

Without loss of generality, consider the access `O[l+Idx[i][j]][k]`. We refer to `O` as the outer array and to `Idx` as the index array. In case of nested indirections, outer/index pairs are processed iteratively from innermost to outermost. While the values taken by the first index expression of the outer array are unknown statically, we can still cache them locally as `shared_O[l][i][j][k] = O[l + Idx[i][j]][k]`. Because some values can be duplicated, indirect promotion is only possible if both the outer and the index arrays are only read, since writing to them could result in different values that cannot be trivially merged. In general, we require the index array to have an array tile, i.e., only a fixed-sized block of it is accessed. When computing the array tile for the outer array, we ignore the

indirect parts of the subscript (affine parts are treated as usual). We then introduce as many additional index expressions in the promoted outer array as are associated to the index array. Extents of the array along these new dimensions correspond exactly to the array tile sizes of the index array. Hence an element of the promoted array contains a copy of the global array element that would be accessed with the given index array. Indirect subscripts are only used when copying from global memory while all other accesses are rewritten through code generation. In presence of multiple indirect index expressions that share sub-expressions and have equal tile sizes along the corresponding dimensions, it is sufficient to introduce a single index expression in the promoted array for all identical sub-expressions.

Promotion Heuristics. Directly accessed arrays are promoted to shared memory if there exists an array tile of fixed size, if individual elements are accessed more than once and if at least one of the accesses does not feature memory coalescing. The latter is visible from the access relation with the schedule applied to the domain: the last access dimension should be aligned with the schedule dimension mapped to `x` threads.

For indirect arrays, the coalescing requirement may be dropped because of the presence of additional long memory dependences that these cases entail. The total amount of shared memory being fixed, one may follow a simple greedy heuristic, refusing promotion if the required amount of shared memory would outgrow the available resources.

3.3.7 Matching Library Calls

While TC aims at generating code for any computational kernel expressible in the DSL, if (part of) a kernel happens to match a pattern that is heavily optimized by some library, then it may as well be handled by that library. In particular, and as a proof of concept, TC looks for opportunities for letting CUB handle specific forms of reductions [323]. It is currently restricted to single-dimensional addition reductions.

A reduction is represented in TC by a binary relation between updated tensor elements and the statement instances that perform the corresponding updates.[5] Right before the mapping to threads, each permutable band with a sufficient number of parallel members

[5]This description is based on commit TC commit `8cfdd5764`, which is slightly ahead of the commit used in the experiments, but is easier to explain.

is checked for reductions. In particular, the band should have at least one non-parallel member and the number of parallel members plus one (corresponding to the non-parallel member) should be greater than or equal to the number of dimensions that will be mapped to threads. If the band schedules instances of exactly one reduction statement and if the instances of any other statement scheduled by the band can be moved before or after the reduction instances, taking into account the active dependence at (the top of) the band, then the remaining band (involving only reduction statement instances) will be considered for replacement by a library call during thread mapping.

When a band marked for replacement is considered during thread mapping, full/partial tile separation is applied—using the block size tuning parameter—since only the full tiles can be handled directly by CUB. Furthermore, the condition separating full tiles from partial tiles should be simple enough as otherwise the cost of determining when to invoke CUB would outweigh any possible benefit obtained from the invocation. If the condition is too complicated, the separation is discarded and the band is treated in the same way as bands that were not marked for replacement. Otherwise, the collection of full tiles is tiled along the parallel dimensions since a single scalar variable is used to hold the result of the reduction mapped to CUB. Synchronization and a special marking is then inserted around the point band of this tiling, which is later used during code generation to replace each full tile by a call to CUB. Finally, since CUB uses some shared memory, its consumpion is taken into account during the downstream memory promotion step.

3.3.8 Autotuning and Caching

While the polyhedral core of TC is capable of optimizing and generating code for any TC function, it is well known that the state of the art linear optimization heuristics are not sufficient to account for all performance anomalies and interactions with downstream program transformations [430, 220]. Different kernels need different, target-specific optimization trade-offs. We thus complement our flow by an autotuner that varies the options of the polyhedral JIT compiler marked as *tunable* in the previous section. These options can be stored and reused for similar operations/kernels (similar shapes, target architecture) since

autotuning may require significantly more time than compilation.

The tuning session is defined by a list of parameters to tune and their admissible values, initial values, and the search strategy. We currently implement a genetic search strategy [138]. It runs for multiple steps, each one evaluating multiple candidate values. Each candidate is assigned a fitness value inversely proportional to its runtime. The pool is updated on each generation by cross-breeding three candidates, chosen from the pool at random, with fitter candidates having a higher chance of being chosen, such that the each candidate's value is inherited from one of its parents. A subsequent mutation phase can change the candidate's values at random with some low probability. Much of the autotuning effort resides in tile size selection, for which no linear objective functions exist in polyhedral compilers. Genetic approaches have been used successfully to explore such spaces, performing better than random search due to the strong coupling of optimization decisions—including tile sizes bound by the limits of the memory hierarchy—[308, 75].

Autotuning evaluates 100s to 1000s versions for each kernel. We devise a generic multi-threaded, multi-GPU autotuner. It maintains a queue of candidates to compile with the polyhedral flow, and a queue of compiled kernels ready to be profiled on the GPU (see Figure 3-4). Candidates or kernels are picked up by available worker threads and compiled or profiled concurrently. Profiling results are accumulated in the tuning database and used for setting up successive search steps.

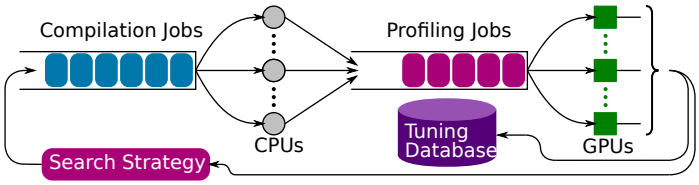


Figure 3-4: Multithreaded autotuning pipeline for kernels

Each generated version is "warmed up" by a few executions before being profiled. Without any performance guarantees, autotuning needs to quickly prune poor candidates. Because CUDA kernels cannot be stopped once launched, we rely on the following pruning heuristics to decrease the autotuning time by an order of magnitude. (1) Parameter specialization allows the exact number of active threads and blocks to be computed beforehand. Kernels with fewer threads than some configurable threshold (e.g., 256) are not launched.

(2) If during the first run, a kernel is more than 100× slower than the best version so far, or it is 5× slower after warmup, it is pruned immediately.

While autotuning time may become significant, compilation and autotuning time is not a fundamental limit to TC's applicability. In training scenarios, a significant amount of time is spent on computing the same kernel repeatedly over different data during the (stochastic) gradient descent. In inference scenarios, the network is optimized ahead of time. As a result, although TC operates as a JIT compiler, it only marginally hits the typical compilation/run-time trade-offs of JIT compilers. Autotuning time may become an issue in specific training scenarios where hyper-parameters would need to be frequently updated, but in such a case one may leverage TC's intrinsic handling of dynamic shapes and generate a single version of each operator or fused operators to handle all hyper-parameter configurations.

3.4 Integration with ML Frameworks

TC is designed to optimize individual layers or small subgraphs of an ML model. Considering the entire model is not only computationally expensive, but often leads to most transformations being hindered by a large number of data dependences. Furthermore, ML frameworks perform work distribution and placement at the model level, treating a layer as a unit of work; extremely large layers could interfere with the framework operation.

Unlike XLA or Glow, TC supports completely custom layers. In TC, *layer fusion* is merely pasting the code that constitutes the layers into a single function, or inlining TC functions at the AST level. Unlike Halide and TVM, the polyhedral backbone of TC includes instance-wise dependence analysis, capturing dependences and tensor access relations at the level of individual loop iterations and tensor elements. This allows TC to fuse operations without introducing redundant computation, and to combine fusion with enabling transformations such as shifting (for convolutions) or scaling (for pooling layers). TC's polyhedral representation also enables it to automatically infer sizes, and to discover parallelism and locality-parallelism trade-offs beyond a predefined collection of map/reduce/scan combinators.

Let us now describe the transparent integration into a ML framework, from a user perspective. Until now, such levels of integration had only been demonstrated on operator graph compilers such as XLA [140] and Glow [329], starting from a lower level of abstraction than TC, and missing the genericity and high reusability of a polyhedral framework as well as feedback-directed autotuning.

We opted for an "in process" implementation, streamlining the interaction with computation graph engines and ML applications built on top of them, a unique feature for a fully-automated scheduling and mapping flow. TC is integrated into any ML framework as follows. We provide a thin API that translates the specific tensor object model to our own, see Figure 3-5 and Figure 3-6. Operator definitions are overridden to generate TC rather than the framework's backend implementation, as well as provide users the ability to write their own TC. A single TC may correspond to a DAG of operators in the ML framework. The tensor comprehensions are then JIT-compiled as shown in Figure 3-2. DAG partitioning, matching and rewriting (like, e.g., TensorRT [290]) is currently not part of the flow, although this would make an interesting future combination, with feedback from the compiler.

```
string tc = R"TC(some_tc_for_conv)TC";
auto I = makeATenTensor<CudaBackend>({N, C, H, W});
auto W = makeATenTensor<CudaBackend>({F, C, KH, KW});
ATenAutotuner<CudaBackend> tuner(tc);
auto best = tuner.tune("conv", {I, W});
auto pExecutor =
    compile<CudaBackend>(tc, "conv", {I, W}, best[0]);
auto out = prepareOutputs(tc, "conv", {I, W});
auto times = profile(*pExecutor, {I, W}, outs);
```

Figure 3-5: Example of embedded usage in C++/ATen.

3.5 Performance Results

We evaluate our framework on 2 systems: (1) Nvidia Pascal nodes with 2 socket, 14 core Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz, with 2 Quadro P100-12GB; and (2) Nvidia Volta nodes with 2 socket, 20 core Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20GHz, with 8 Tesla V100-SXM2-16GB. Both systems use CUDA 9.0 and cuDNN 7.0.

```
import torch
import tensor_comprehensions as tc
tcdef = """...some_tc_for_conv..."""
T_I = torch.randn(N, C, H, W).cuda()
T_W = torch.randn(F, C, KH, KW).cuda()
# register the TC string
conv = tc.define(tcdef, name="conv")
# autotune the kernel
best = conv.autotune(T_I, T_W)
# run with best option and cache the binary
T_O = conv(T_I, T_W, options=best)
```

Figure 3-6: Example of embedded usage in PyTorch.

We report results for 9 TC functions ranging from a simple matrix multiplication kernel to a full WaveNet cell [394]. The individual benchmarks are described below: Figure 3-7 and Figure 3-8 show the complete source code. The matrix multiplication and convolution kernels were selected for their dominance of the training and inference time of the most classical networks [19, 418]. The other kernels bring interesting computation patterns to enable expressiveness and performance comparisons in more diverse network architectures.

These results are all based on TC commit `2e1a0dc54850` available at `https://github.com/nicolasvasilache/TensorComprehensions`.

Running the autotuner for 25 generations of 100 candidates, the (parallel) autotuning process takes up to 1h on the longest running kernels, and 6h in total.[6]

The relative performance of kernels automatically generated with TC compared to Caffe2 is shown in Figure 3-9 and Figure 3-10.[7] Caffe2 provides a very strong baseline by wrapping tuned implementations, which originate from either hand-tuned libraries or other high-performance code generators.[8] *We chose to compare against Caffe2 rather than against other optimization flows due to expressivity and automation limitations*: XLA or Glow do not support custom layers, and Halide or TVM lack range inference and automatic parallelism discovery, which significantly complicates the expression of new layers such as KRU and WaveNet. The common set of comparable layers would be limited to

[6]Classical strategies exist to accelerate autotuning, such as predictive modeling and search space pruning [4], but this was not the focus of this chapter.

[7]We compile Caffe2 and PyTorch from source (commit `6223bfdb1d32`) and integrate it in the TC testing flow for proper benchmarking.

[8]A recent unification effort [349] made Caffe2 the backend for PyTorch 1.0.

```
def tmm(float(M,K) A, float(N,K) B) -> (C) { C(m,n) +=! A(m,r_k) * B(n,r_k) }
def tbmm(float(B,N,M) X, float(B,K,M) Y) -> (Z) { Z(b,n,k) +=! X(b,n,r_m) * Y(b,k,r_m) }

def 1LUT(float(E1,D) LUT1, int(B,L1) I1) -> (O1) { O1(i,j) +=! LUT1(I1(i,r_k),j) }
def 2LUT(float(E1,D) LUT1, int(B,L1) I1, float(E1,D) LUT1, int(B,L1) I1) -> (O1, O2) {
  O1(i,j) +=! LUT1(I1(i,r_k),j)
  O2(i,j) +=! LUT2(I2(i,r_k),j)
}

def MLP3(float(B,M) I, float(O,N) W2, float(O) B2, float(P,O) W3, float(P) B3, float(Q,P) W4,
   float(Q) B4) -> (O1,O2,O3,O4) {
  O2(b,o)  = B2(o)
  O2(b,o) += O1(b,r_n) * W2(o,r_n)
  O2(b,o)  = fmaxf(O2(b,o), 0)
  O3(b,p)  = B3(p)
  O3(b,p) += O2(b,r_o) * W3(p,r_o)
  O3(b,p)  = fmaxf(O3(b,p), 0)
  O4(b,q)  = B4(q)
  O4(b,q) += O3(b,r_p) * W4(q,r_p)
  O4(b,q)  = fmaxf(O4(b,q), 0)
}

def kronecker3(float(D0,N0) W0, float(D1,N1) W1, float(D2,N2) W2, float(M,N0,N1,N2) X) -> (Y,XW1,XW2) {
  XW2(m,n0,n1,d2) +=!   X(m,n0,n1,r_n2) * W2(d2,r_n2)
  XW1(m,n0,d1,d2) +=! XW2(m,n0,r_n1,d2) * W1(d1,r_n1)
    Y(m,d0,d1,d2) +=! XW1(m,r_n0,d1,d2) * W0(d0,r_n0)
}

def group_convolution(float(N,G,C,H,W) I, float(G,F,C,KH,KW) W1, float(M) B) -> (O) {
  O(n,g,o,h,w)  = O(n,g,o,h,w) + B(m)
  O(n,g,o,h,w) += I(n,g,r_i, h + r_kh, w + r_kw) * W1(g,o,r_i,r_kh,r_kw)
}

def moments2_2d_1D(float(N,K) I) -> (mean,var) {
# var = E(x^2) - mean^2
  mean(n) +=! I(n,r_k)
  var(n)  +=! I(n,r_k) * I(n,r_k)
  mean(n)  =  mean(n) / K
   var(n)  =  var(n) / K - mean(n) * mean(n)
}

def group_normalization(float(N,G,D,H,W) I, float(G,D) gamma, float(G,D) beta, float(N,G) mean, float(N,G) var) ->
(O,mean,var) {
  O(n,g,d,h,w) = gamma(g,d) * (I(n,g,d,h,w) - mean(n,g)) * rsqrt(var(n,g) - mean(n,g) * mean(n,g) + 1e-5) + beta(g,d)
}

def group_normalization_single_kernel(float(N,G,D,H,W) I, float(G,D) gamma, float(G,D) beta) -> (O,mean,var) {
  mean(n,g) +=! I(n,g,r_d,r_h,r_w)
  var(n,g)  +=! I(n,g,r_d,r_h,r_w) * I(n,g,r_d,r_h,r_w)
  O(n,g,d,h,w) = gamma(g,d) * (I(n,g,d,h,w) - mean(n,g) / (D * H * W)) * rsqrt(var(n,g)/(D * H * W)
                - mean(n,g)/(D * H * W) * mean(n,g)/(D * H * W) + 1e-5) + beta(g,d)
}
```

Figure 3-7: TC Benchmarks used in the experiments. Evaluated sizes are available in Table 3.1.

matrix multiplications and convolutions, while one of the main contributions of TC is to enable exploration of new unconventional layers *before super-optimized implementations are available*.

```
def wavenet1(float(B, RESIDUAL_C, RECEPTIVE_FIELD) Data,
             float(DILATION_C, RESIDUAL_C, 2) FilterWeight, float(DILATION_C) FilterBias,
             float(DILATION_C, RESIDUAL_C, 2) GateWeight, float(DILATION_C) GateBias,
             float(RESIDUAL_C, DILATION_C) ResWeight, float(RESIDUAL_C) ResBias,
             float(SKIP_C, DILATION_C) SkipWeight, float(SKIP_C) SkipBias,
             float(DILATION_FACTOR) Dilation) -> (FilterOut, GateOut, NonLin, Res, Skip) {
  FilterOut(b, dilation_c, rf)  = FilterBias(dilation_c)
    where b in 0:B, dilation_c in 0:DILATION_C, rf in 0:RECEPTIVE_FIELD
  FilterOut(b, dilation_c, rf) += Data(b, r_residual_c, rf) * FilterWeight(dilation_c, r_residual_c, 1)
      + ((rf - DILATION_FACTOR >= 0)
        ? Data(b, r_residual_c, rf - DILATION_FACTOR) * FilterWeight(dilation_c, r_residual_c, 0) : float(0))
      where rf in 0:RECEPTIVE_FIELD

  GateOut(b, dilation_c, rf)  = GateBias(dilation_c)
    where b in 0:B, dilation_c in 0:DILATION_C, rf in 0:RECEPTIVE_FIELD
  GateOut(b, dilation_c, rf) += Data(b, r_residual_c, rf) * GateWeight(dilation_c, r_residual_c, 1)
    + ((rf - DILATION_FACTOR >= 0)
        ? Data(b, r_residual_c, rf - DILATION_FACTOR) * GateWeight(dilation_c, r_residual_c, 0) : float(0))
      where rf in 0:RECEPTIVE_FIELD

  NonLin(b, dilation_c, rf)  = tanh(FilterOut(b, dilation_c, rf)) where rf in 0:RECEPTIVE_FIELD
  NonLin(b, dilation_c, rf) *= 1 / (1 + exp(-GateOut(b, dilation_c, rf))) where rf in 0:RECEPTIVE_FIELD

  Res(b, residual_c, rf)  = Data(b, residual_c, rf) + ResBias(residual_c)
  Res(b, residual_c, rf) += NonLin(b, r_dilation_c, rf) * ResWeight(residual_c, r_dilation_c)

  Skip(b, skip, rf) +=! NonLin(b, r_dilation_c, rf) * SkipWeight(skip, r_dilation_c) where rf in 0:RECEPTIVE_FIELD
  Skip(b, skip, rf)  =  Skip(b, skip, rf) + SkipBias(skip) where rf in 0:RECEPTIVE_FIELD
}
```

Figure 3-8: Source of one full WaveNet cell.

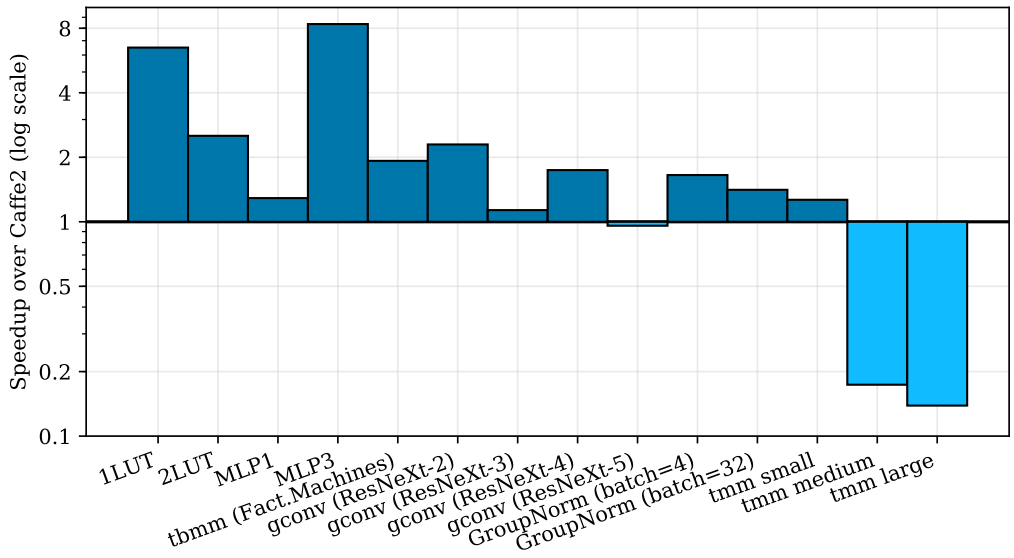


Figure 3-9: Speedup of TC-generated kernels over Caffe2 hand-tuned kernels on Quadro P100-12GB

In addition, Figure 3-11 brings together the performance of TC-compiled kernels on both GPU systems, normalized to Caffe2 on P100. This consolidated graph conveys 3

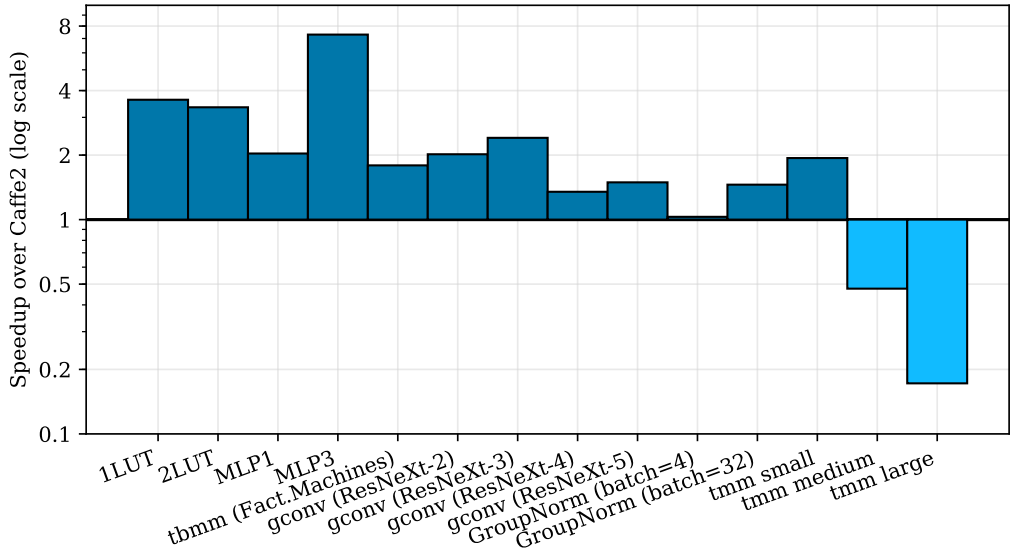


Figure 3-10: Speedup of TC-generated kernels over Caffe2 hand-tuned kernels on Tesla V100-SXM2-16GB

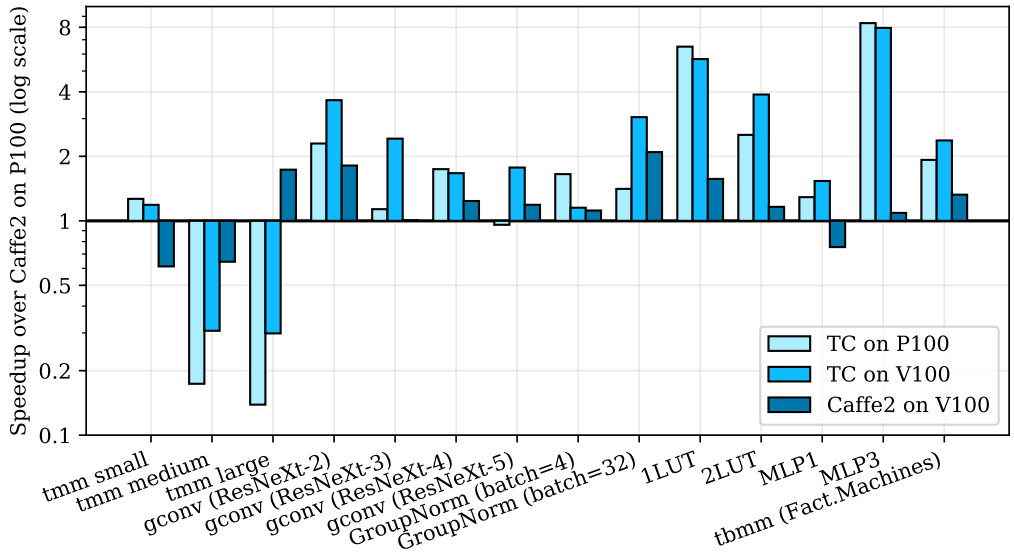


Figure 3-11: Relative performance: baseline is Caffe2 performance on a P100 GPU

classes of information in a common context: (1) speedup of Caffe2 V100 over Caffe2 P100 to illustrate the out-of-the-box benefits (or lack thereof) of a faster GPU; (2) speedup of TC over Caffe2 on P100 (main comparison); and (3) speedup of TC V100 over Caffe2 P100. The last choice may seem surprising, but presented in the context of the other two, allows for relative comparisons: the height of the Caffe2 V100 and TC V100 captures the raw speedups of TC on V100. *We aim at compactly illustrating that TC provides a path to*

performance portability, improving on state of the art frameworks and library primitives.

Table 3.1 provides absolute runtime running TC and Caffe2; all values are reported in μs.

		Pascal			Volta		
1LUT		p0	p50	p90	p0	p50	p90
$B = 128, D = 64, E1 = 10^7, L1 = 50$	TC	13	14	14	15	16	17
	Caffe2	85	91	95	56	58	63
2LUT		p0	p50	p90	p0	p50	p90
$B = 128, D = 64, E1, E2 = 10^7, L1, L2 = 50$	TC	52	54	57	35	35	37
	Caffe2	132	136	144	115	117	124
MLP1		p0	p50	p90	p0	p50	p90
$B = 128, M = 2000, N = 128$	TC	68	69	71	57	58	59
	Caffe2	87	89	91	116	118	123
MLP3		p0	p50	p90	p0	p50	p90
$B = 128, N = 128, O = 64, P = 32, Q = 2$	TC	18	19	19	20	20	21
	Caffe2	157	159	169	144	146	164
tbmm		p0	p50	p90	p0	p50	p90
$B = 500, K = 26, M = 72, N = 26$	TC	52	53	54	42	43	43
	Caffe2	94	102	103	76	77	78
Group Convolution		p0	p50	p90	p0	p50	p90
$C, F = 4, G, N = 32, H = 56, KH, KW = 3, W = 56$	TC	696	701	704	435	440	443
	Caffe2	1590	1609	1621	879	888	896
$C, F = 8, G, N = 32, H = 28, KH, KW = 3, W = 28$	TC	574	576	578	269	270	272
	Caffe2	640	653	692	613	650	660
$C, F = 16, G, N = 32, H = 14, KH, KW = 3, W = 14$	TC	265	272	276	274	284	287
	Caffe2	440	474	510	377	383	397
$C, F = 32, G, N = 32, H = 7, KH, KW = 3, W = 7$	TC	463	481	491	259	260	264
	Caffe2	456	461	469	367	388	394
Group Normalization		p0	p50	p90	p0	p50	p90
$C = 512, G = 32, H = 12, N = 4, W = 12$	TC	22	23	24	32	33	35
	Caffe2	37	38	40	33	34	35
$C = 512, G = 32, H = 48, N = 32, W = 48$	TC	1285	1290	1294	593	597	601
	Caffe2	1814	1819	1823	865	869	871
tmm		p0	p50	p90	p0	p50	p90
$K = 32, M = 128, N = 256$	TC	15	15	15	15	16	17
	Caffe2	18	19	20	31	31	32
$K = 1024, M = 128, N = 1024$	TC	318	334	344	181	189	192
	Caffe2	55	58	64	89	90	91
$K = 4096, M = 128, N = 16384$	TC	17168	17209	17270	7937	8004	8096
	Caffe2	2254	2388	2590	1360	1378	1419

Table 3.1: Absolute run time in μs

TMM: Transposed Matrix-Multiplication On matrix multiplications of shapes and sizes relevant to deep learning workloads (i.e., small $128 \times 32 \times 256$, medium $128 \times 1024 \times 1024$ and large $128 \times 4096 \times 16384$), TC does not perform competitively, except in the low-latency small case. This is due to: (1) the lack of a target-specific register blocking optimization, making kernels bound by shared memory bandwidth which is an order of

magnitude slower than register bandwidth; (2) the lack of target-specific, basic-block level optimizations including careful register allocation and instruction scheduling. Matrix multiplication is the most tuned computation kernel in history: the missing optimizations are all well known, and may be found in use cases and open source implementations such like CUTLASS [210]. Alternatively, polyhedral compilation has been shown to match or outperform cuBLAS, provided sufficient target- and operator-specific information has been captured in the optimization heuristic and code generator [107]. While our scientific focus was on covering a wide range of layers with TC, a production release would need to embed such operator-specific strategies as well. One strategy would be to follow the classification and heuristic steering of Kong et al. [220]. Also, TC does not replace all layers: it only acts as a custom operation in a graph; one may use TC concurrently with numerical libraries as well as custom implementations provided through TVM.

Group Convolution Group convolution is expressible with 2 lines of TC. We report comparisons for sizes relevant to the ResNext model [418]. Despite not using either register optimizations, Fourier or Winograd domain convolutions, TC produces faster kernels than the cuDNN ones, with running times between 250μs and 750μs. To check how TC fares w.r.t. recent advances in optimizing group convolutions, we performed an additional comparison with the PyTorch nightly package `py36_cuda9.0.176_cudnn7.1.2_1` with `torch.backends.cudnn.benchmark=True`. TC speedups range from -2% to $8\times$. We also observe PyTorch performance on V100 to be worse than on P100 while TC achieves performance portability.

Group Normalization Group Normalization was recently proposed as a way to overcome limitations of Batch Normalization at smaller batch sizes and increase parallelism [417]. In TC, group normalization is a 5-line function. TC performance is roughly 30% better than the hand-tuned Caffe2 implementation. Whereas Caffe2 uses 4 *handwritten* CUDA kernels, we chose to write the TC version as two separately compiled TC functions for better reuse and overall performance. We also experimented with writing a single fused TC but performance degraded. This is mostly due to kernels requiring substantially different grid configurations, which makes their fusion unprofitable. A larger, graph-level compiler that decides on TC function granularity, informed by the TC mapper and the

autotuner is necessary to automate this decision process, but is left for future work.

Production Model The kernels `1LUT`, `2LUT`, `MLP1` and `MLP3` are the backbone of a low-latency production model used at scale in a large company and correspond to (1) reductions over a large lookup table embedding (10M rows); (2) fused reduction over 2 large lookup table embeddings (10M rows); (3) small size Multi-Layer Perceptron (fully-connected, bias, ReLU); and (4) very small size, 3 consecutive Multi-Layer Perceptrons. Despite LUT sizes, this model is essentially latency-bound. Existing libraries are often not tuned for low-latency regimes and tend to perform poorly.

On these examples, the need for reuse and instruction-level parallelism is dwarfed by the need to quickly load data from the memory into registers. TC is able to adapt to the problem size, leveraging reduction parallelism to hide memory latency. This results in large speedups over Caffe2 with cuBLAS 9.0.

Transposed Batch MatMul This kernel is meant as a case study to characterize performance benefits and losses in the current flow, compared with reference libraries. For the sizes relevant to Factorization Machines [322], $(500 \times 26 \times 72 \times 26)$, Nvidia Profiler reports the TC autotuned kernel taking 56μs on the Nvidia Quadro P6000 GPU (Pascal) while both Pytorch and Caffe2 resort to the specialized cuBLAS function `maxwell_sgemm_128x64_nn` that takes 87μs. Beyond architecture mismatch indicated in the function name, a detailed performance comparison demonstrates that TC executes 500 blocks of $26 \times 13 = 338$ threads, compared to 500 blocks of 128 threads for cuBLAS, reaching 81.8% occupancy instead of 23.6%. Additionally, the cuBLAS kernel shows a large number of predicated-off instructions due to the block size not matching the problem size. Occupancy is limited by the number of registers in both cases (11264 vs. 15360), but the TC version can be distributed over 5 blocks instead of 4.[9] TC promotes all tensors to shared memory, saturating its bandwidth, whereas arithmetic instructions are the performance limiter for cuBLAS. Given the large occupancy metric, performance can be further increased by promoting one tensor to registers instead, trading off lower occupancy for reduced pressure on memory bandwidth.

[9]Mapping to blocks of 32×13 threads to obtain full warps results in 60μs execution time and only 4 blocks due to the higher number of registers per block.

Kronecker Recurrent Units These have been recently proposed as a solution to drastically reduce model sizes by replacing the weights matrix of a linear layer by a Kronecker product of much smaller matrices [200]. In TC, a Kronecker product of 3 matrices is easily written as shown in the `kronecker3` function in Figure 3-7. The following table shows the running time in μs—or out of memory (OOM)—of a large matrix multiplication in Caffe2 and the equivalent Kronecker product of 3 matrices. Note that *the performance difference mostly comes form a using a different algorithm*. While no specialized GPU library primitives exist for Kronecker recurrent units, TC's automatic flow enabled rapid exploration and reaching unprecedented levels of performance, as shown in Table 3.2. Clearly, this benchmark deserves a deeper discussion of the space of possible TC derivations, including memory/computation/parallelism trade-offs falling outside the scope of this chapter. The `kronecker3` function is one such possible implementations that performed well for the three selected matrix shapes; it avoids redundant computation at the expense of storage (two tensors for intermediate computations).

WaveNet WaveNet [394] is a popular model that enables generation of realistic sounding voices as highlighted at Google I/O 2018. We encoded a full WaveNet cell using a single TC function and compared our generated kernel with a WaveNet layer from PyTorch. This experiment uses a batch size of 1, residual and dilation channels of 32, and 256 skip channels. With TC, we observe performance improvements up to 4× on Volta, as shown on Table 3.2.

Algorithmic exploration of Kronecker Recurrent Units		**Pascal** p0	p50	p90	**Volta** p0	p50	p90
$256 \times 16^3 \times 32^3$	TC Kronecker	272	280	285	200	206	212
	Caffe2 MatMul	7714	8158	8216	4946	5065	5466
$256 \times 16^3 \times 64^3$	TC Kronecker	1334	1349	1365	998	1004	1011
	Caffe2 MatMul	64499	64765	65659	38280	39307	39327
$256 \times 16^3 \times 64 \times 128^2$	TC Kronecker	4408	4447	4472	4794	4815	5106
	Caffe2 MatMul	OOM	OOM	OOM	OOM	OOM	OOM
WaveNet Cell		p0	p50	p90	p0	p50	p90
receptive field = $4K$, dilation = 1	TC	457	466	477	253	255	257
receptive field = $4K$, dilation = 1	PyTorch	549	576	790	563	571	594
receptive field = $4K$, dilation = 32	TC	353	365	375	138	139	140
receptive field = $4K$, dilation = 32	PyTorch	551	574	630	562	569	585

Table 3.2: Algorithmic exploration of Kronecker Recurrent Units and optimization of a WaveNet cell

3.6 Related Work

Despite decades of progress in optimizing and parallelizing compilation, programmers of computationally intensive applications complain about the poor performance of optimizing compilers, often missing the machine peak by orders of magnitude. Among the reasons for this state of the affairs, one may cite the complexity and dynamic behavior of modern processors, domain knowledge required to prove optimizations' validity or profitability being unavailable to the compiler, program transformations whose profitability is difficult to assess, and the intrinsic difficulty of composing complex transformations, particularly in the case of computationally intensive loop nests [136, 24].

Several contributions have successfully addressed this issue, not by improving a general-purpose compiler, but through the design of application-specific program generators, a.k.a. active libraries [399]. Such generators often rely on feedback-directed optimization to select the best generation schema [358], as popularized by ATLAS [414] for dense matrix operations (and more recently BTO [36]) and FFTW [122] for the fast Fourier transform. Most of these generators use transformations previously proposed for traditional compilers, which fail to apply them for the aforementioned reasons. The SPIRAL project [314] made a quantum leap over these active libraries, operating on a domain-specific language (DSL) of digital signal processing formulas. Compilers for DSLs typically rely on domain-specific constructs to capture the intrinsic parallelism and locality of the application. Using such an approach, DSL compilers such as Halide [319] for image processing show impressive results. Its inputs are images defined on an infinite range while TC sets a fixed size for each dimension using range inference. This is better suited to ML applications, dominated by fixed size tensors with higher temporal locality than 2D-images; it is also less verbose in the case of reductions and does not carry the syntactic burden of anticipating the declaration of stage names and free variables (Halide needs this as a C++ embedded DSL). OoLaLa [253] takes a similar approach for linear algebra, and TACO [214] and Simit [215] use a similar notation as TC but generate sparse matrix code for numerical solvers.

Following this trend in the context of deep neural networks, we not only design yet another DSL and compiler but propose a more generic code generation and optimization

framework bringing together decades of research in loop nest optimization and parallelization for high-performance computing. We also design the domain language to cover a variety of existing and emerging machine learning models. Our framework automates a combination of affine transformations involving hierarchical tiling, mapping, shifting, fusion, distribution, interchange, on either parametric or fully instantiated problems, that are not accessible to Halide [319, 276], Latte [389] or XLA's [140] representations of tensor operations.

The polyhedral framework is a powerful abstraction for the analysis and transformation of loop nests, and a number of tools and libraries have been developed to realize its benefits [112, 52, 402, 50, 430], including production compilers such as GCC (Graphite) and LLVM (Polly). Polyhedral techniques have also been tailored for domain-specific purposes. State of the art examples include the PolyMage [277] DSL for image processing pipelines and the PENCIL approach to the construction of parallelizing and compilers for DSLs [22, 32]. PolyMage is a clear illustration of the benefits of operating at a high level of abstraction, closer to the mathematics of the domain of interest: while GCC/Graphite and LLVM/Polly struggle to recover affine control and flow from low-level code, PolyMage natively captures patterns amenable to domain-specific optimization, such as stencil-specific overlapped tiling with or without recomputation, and cache-conscious fusion and tiling heuristics; it also offers a more productive programming experience for end users. Interestingly, some techniques derived from PolyMage crossed out of polyhedral representations into Halide's automatic scheduler [276]. Back to deep learning frameworks, TVM extends Halide with recurrent (parallel scan) operators, support for ML accelerators, and tight integration with ML frameworks [74]. It also provides autotuning capabilities [75] and shares several engineering goals of TC, such as transparent ML framework integration. Much like PolyMage, TC implements optimizations well suited to the long distance, non-uniform reuse patterns of deep learning models; these heuristics are not available in general-purpose compilers such as LLVM/Polly, Pluto or PPCG, or semi-automatic frameworks such as Halide and TVM.

None of the aforementioned frameworks offer the complete transparency of TC's end-to-end compilation flow. TVM involves some level of manual intervention and/or feedback-

directed optimization even for producing the most baseline GPU implementation, and it guarantees functional correctness for a subset of the scheduling primitives and tensor operations: e.g., convolutions can only be fused at the expense of introducing redundant computations, or involving lower level transformations that cannot be verified at compilation time. In addition, the balance between analytical objective functions (profitability heuristics) and feedback-directed autotuning is completely different: Halide and TVM auto-schedulers expose all scheduling decisions to the autotuner and infer most performance-related information from execution profiles, while TC's polyhedral flow reduces the autotuning space to a narrow set of optimization options and tile sizes.

TC also shares several motivations with Latte [389] and PlaidML [303], including a high level domain-specific language and an end-to-end flow. TC provides elementwise access that is just as expressive when implementing custom layers, but unlike Latte it is more concise thanks to type and shape inference, safer regarding static bound checking and graph connectivity, and more flexible by decoupling indexing from representation and layout choices. In addition, our framework implements more complex scheduling and mapping transformations than both Latte and PlaidML, some of which are essential to GPU targets with partitioned memory architectures. Unlike Latte, it is also designed as a JIT compilation library for seamless integration with deep learning frameworks. Unlike PlaidML, it is not limited to high level patterns and rewrite rules, but captures complex affine transformations resulting from analytical modeling and autotuning. As a consequence, the TC compilation process takes generally more time than PlaidML, a price to pay for the ability to implement a wider range of optimizations.

Like TC, XLA [140] provides automatic shape and size inference, it may operate "in process" as a JIT compilation library, and it integrates into a production deep learning framework (TensorFlow, Caffe2 [141]). XLA shares many motivations with Latte, with a focus on integration and completeness of functionality rather than on the complexity of the optimizations and mapping strategies. Glow [329] is a recent domain-specific, retargetable compiler for PyTorch/Caffe2. It shares many of the motivations and capabilities of XLA, while emphasizing retargetability (CPUs as well GPUs and ML accelerators from multiple vendors) and the ability to differentiate, optimize, lower operations and sub-graphs of op-

erations within its own hierarchy of intermediate representations. It can leverage blackbox numerical libraries as well as generate custom vector processing kernels relying on LLVM. Our compiler design and algorithmic contributions would naturally fit XLA, Latte or Glow, except for the following: TC remains independent from a specific computation graph while preserving tight integration with production frameworks; we did not use an embedded DSL approach—keeping C++ as an interface for implementing optimization strategies only—isolating the user from complexity and debugging hurdles of embedded DSLs, and we leverage polyhedral techniques to factor-out most of the optimization heavy-lifting, while XLA, Latte and Glow resort to operation-specific emitters/lowering, optimization schemas and heuristics.

Recently, R-Stream·TF [311] was presented as a proof-of-concept adaptation of the R-Stream polyhedral compiler to the automatic optimization of TensorFlow operators. Similarly to our approach, the generated code is wrapped as a custom operator of TensorFlow. The tool takes a computation graph as input and partitions it into sub-graphs amenable to tensor fusion, contraction and layout optimization. R-Stream·TF also leverages the broadcast semantics of TensorFlow to maximize the operator's polymorphism w.r.t. input tensor dimension and shapes. This makes R-Stream·TF very aggressive in terms of static memory management and kernel partitioning. We made the more pragmatic choice of leaving most of these decisions to the level of tensor algebra, allowing a domain-specific optimizer or ML expert to rewrite declarative comprehensions into capacity- and layout-optimized ones. On the other hand, TC is more ambitious in its domain-specialization of affine scheduling and mapping, aiming for the generation of a single accelerated kernel, with heuristics adapted to the high dimensional, non-uniform, long distance reuse patterns of neural networks. The lack of algorithmic detail in the R-Stream·TF paper prevents us from comparing those affine transformation heuristics.

3.7 Conclusion

We presented and evaluated the first fully automatic, end-to-end flow, mapping a high-level mathematical language to high-performance accelerated GPU kernels. TC resembles

the mathematical notation of a deep neural network and makes it easy to reason about, communicate, and to manually alter the computation and storage/computation trade-offs. Our flow leverages decades of progress in polyhedral compilation to implement the heavy-duty program transformations, analytical modeling of profitable optimizations, and code synthesis. It also implements domain-specific optimizations, code generation, autotuning with a compilation cache, and lightweight integration within Caffe2 and PyTorch. This unique combination differs from alternative proposals relying mainly on autotuning such as TVM [75], or pattern-based transformations such as PlaidML [303].

TC is capable of quickly synthesizing solid accelerated implementations that effectively lift bottlenecks in large training runs. In practice, such bottlenecks slow down ML research significantly, requiring substantial engineering efforts to be mobilized. Our contribution addresses this productivity gap; it brings more expressive power and control in the hands of domain experts, relieving ML frameworks' dependence on highly tuned vendor libraries, without compromising performance. TC automates boilerplate optimization that has been replicated over the numerous deep learning frameworks, and builds on a generic polyhedral intermediate representation and libraries shared with other domains (image processing, linear algebra) and general-purpose compilers (LLVM/Polly). Future work includes additional model-based domain-specific optimizations, CPU code generation, learning best mapping configurations automatically, automatic differentiation, interaction with the graph-level optimizer, and providing a path to emit a series of calls to a native library or hardware acceleration blocks.

Chapter 4

AutoPhase: Machine-Learning Assisted Optimization Ordering

4.1 Introduction

High-Level Synthesis (HLS) automates the process of creating digital hardware circuits from algorithms written in high-level languages. Modern HLS tools [419, 187, 61] use the same front-end as the traditional software compilers. They rely on traditional software compiler techniques to optimize the input program's intermediate representation (IR) and produce circuits in the form of RTL code. Thus, the quality of compiler front-end optimizations directly impacts the performance of HLS-generated circuit.

Program optimization is a notoriously difficult task. A program must be just in "the right form" for a compiler to recognize the optimization opportunities. This is a task a programmer might be able to perform easily, but is often difficult for a compiler. Despite a decade of research on developing sophisticated optimization algorithms, there is still a performance gap between the HLS generated code and the hand-optimized one produced by experts.

In this chapter, we build off the LLVM compiler [230]. However, our techniques, can be broadly applicable to any compiler that uses a series of optimization passes. In this case, the optimization of an HLS program consists of applying a sequence of analysis and

optimization phases, where each phase in this sequence consumes the output of the previous phase, and generates a modified version of the program for the next phase. Unfortunately, these phases are not commutative which makes the order in which these phases are applied critical to the performance of the output.

Consider the program in Figure 4-1, which normalizes a vector. Without any optimizations, the `norm` function will take $\Theta(n^2)$ to normalize a vector. However, a smart compiler will implement the *loop invariant code motion (LICM)* [275] optimization, which allows it to move the call to `mag` above the loop, resulting in the code on the left column in Figure 4-2. This optimization brings the runtime down to $\Theta(n)$—a big speedup improvement. Another optimization the compiler could perform is *(function) inlining* [275]. With inlining, a call to a function is simply replaced with the body of the function, reducing the overhead of the function call. Applying inlining to the code will result in the code in the right column of Figure 4-2.

```
__attribute__((const))
double mag(int n, const double *A) {
  double sum = 0;
  for(int i=0; i<n; i++){
    sum += A[i] * A[i];
  }
  return sqrt(sum);
}
void norm(int n, double *restrict out,
        const double *restrict in) {
  for(int i=0; i<n; i++) {
    out[i] = in[i] / mag(n, in);
  }
}
```

Figure 4-1: A simple program to normalize a vector.

Now, consider applying these optimization passes in the opposite order: first inlining then LICM. After inlining, we get the code on the left of Figure 4-3. Once again we get a modest speedup, having eliminated n function calls, though our runtime is still $\Theta(n^2)$. If the compiler afterwards attempted to apply LICM, we would find the code on the right of Figure 4-3. LICM was able to successfully move the allocation of sum outside the loop. However, it was unable to move the instruction setting `sum=0` outside the loop, as doing so would mean that all iterations excluding the first one would end up with a garbage value

```
void norm(int n, double *restrict out,
          const double *restrict in) {
  double precompute = mag(n, in);




  for(int i=0; i<n; i++) {
    out[i] = in[i] / precompute;
  }
}
```

```
void norm(int n, double *restrict out,
          const double *restrict in) {
  double precompute, sum = 0;
  for(int i=0; i<n; i++){
    sum += A[i] * A[i];
  }
  precompute = sqrt(sum);
  for(int i=0; i<n; i++) {
    out[i] = in[i] / precompute;
  }
}
```

Figure 4-2: Progressively applying LICM (left) then inlining (right) to the code in Figure 4-1.

```
void norm(int n, double *restrict out,
          const double *restrict in) {

    for(int i=0; i<n; i++) {
        double sum = 0;
        for(int j=0; j<n; j++){
            sum += A[j] * A[j];
        }
        out[i] = in[i] / sqrt(sum);
    }
}
```

```
void norm(int n, double *restrict out,
          const double *restrict in) {
    double sum;
    for(int i=0; i<n; i++) {
        sum = 0;
        for(int j=0; j<n; j++){
            sum += A[j] * A[j];
        }
        out[i] = in[i] / sqrt(sum);
    }
}
```

Figure 4-3: Progressively applying inlining (left) then LICM (right) to the code in Figure 4-1.

for sum. Thus, the internal loop will not be moved out.

As this simple example illustrates, the order in which the optimization phases are applied can be the difference between the program running in $\Theta(n^2)$ versus $\Theta(n)$. It is thus crucial to determine the optimal phase ordering to maximize the circuit speeds. Unfortunately, not only is this a difficult task, but the optimal phase ordering may vary from program to program. Furthermore, it turns out that finding the optimal sequence of optimization phases is an NP-hard problem, and exhaustively evaluating all possible sequences is infeasible in practice. In this work, for example, the search space extends to more than 2^{247} phase orderings.

The goal of this chapter is to provide a mechanism for automatically determining good phase orderings for HLS programs to optimize for the circuit speed. To this end, we aim to leverage recent advancements in deep reinforcement learning (RL) [378, 157] to address the phase ordering problem. With RL, a software agent continuously interacts with

the environment by taking actions. Each action can change the state of the environment and generate a "reward". The goal of RL is to learn a policy—that is, a mapping between the observed states of the environment and a set of actions—to maximize the cumulative reward. An RL algorithm that uses a deep neural network to approximate the policy is referred to as a deep RL algorithm. In our case, the observation from the environment could be the program and/or the optimization passes applied so far. The action is the optimization pass to apply next, and the reward is the improvement in the circuit performance after applying this pass. The particular framing of the problem as an RL problem has a significant impact on the solution's effectiveness. Significant challenges exist in understanding how to formulate the phase ordering optimization problem in an RL framework.

In this chapter, we consider three approaches to represent the environment's state. The first approach is to directly use salient features from the program. The second approach is to derive the features from the sequence of optimizations we applied while ignoring the program's features. The third approach combines the first two approaches. We evaluate these approaches by implementing a framework that takes a group of programs as input and quickly finds a phase ordering that competes with state-of-the-art solutions. Our main contributions are:

- Leveraging deep RL to address the phase-ordering problem.

- An importance analysis on the features using random forests to significantly reduce the state and action spaces.

- AutoPhase: a framework that integrates the current HLS compiler infrastructure with the deep RL algorithms.

- A demonstration that AutoPhase gets a 28% improvement over `-O3` for nine real benchmarks. Unlike all state-of-the-art approaches, deep RL demonstrates the potential to generalize to thousands of different programs after training on a hundred programs.

4.2 Background

4.2.1 Compiler Phase-ordering

Compilers execute optimization passes to transform programs into more efficient forms to run on various hardware targets. For ease, roups of optimizations are often packaged into "optimization levels" , such as `-O0` (no optimization), `-O1` (some optimization) `-O2` (more optimization), and `-O3` (most optimization). While these optimization levels offer a simple set of choices for developers, they are handpicked by the compiler-designers and often most benefit certain groups of benchmark programs. The compiler community has attempted to address the issue by selecting a particular set of compiler optimizations on a per-program or per-target basis for software [387, 13, 294, 17].

Since the search space of phase-ordering is too large for an exhaustive search, many heuristics have been proposed to explore the space by using machine learning. Huang *et al.* tried to address this challenge for HLS applications by using modified greedy algorithms [176, 177]. It achieved 16% improvement vs -O3 on the CHstone benchmarks [161], which we used in this chapter. In [5] both independent and Markov models were applied to automatically target an optimized search space for iterative methods to improve the search results. In [368], genetic algorithms were used to tune heuristic priority functions for three compiler optimization passes. Milepost GCC [126] used machine learning to determine the set of passes to apply to a given program, based on a static analysis of its features. It achieved an 11% execution time improvement over -O3, for the ARC reconfigurable processor on the MiBench program suite1. In [225] the challenge was formulated as a Markov process and supervised learning was used to predict the next optimization, based on the current program state. OpenTuner [17] autotunes a program using an AUC-Bandit-meta-technique-directed ensemble selection of algorithms. Its current mechanism for selecting the compiler optimization passes does not consider the order or support repeated optimizations. Wang *et al.* [412], provided a survey for using machine learning in compiler optimization where they also described that using program features might be helpful. NeuroVectorizer [156, 155] used deep RL for automatically tuning compiler pragmas such as vectorization and interleaving factors. NeuroVectorizer achieves 97% of the oracle perfor-

mance (brute-force search) on a wide range of benchmarks.

4.2.2 Reinforcement Learning Algorithms

Reinforcement learning (RL) is a machine learning approach in which an agent continually interacts with the environment [203]. In particular, the agent observes the state of the environment, and based on this observation takes an action. The goal of the RL agent is then to compute a policy–a mapping between the environment states and actions–that maximizes a long term reward.

RL can be viewed as a stochastic optimization solution for solving Markov Decision Processes (MDPs) [35], when the MDP is not known. An MDP is defined by a tuple with four elements: $S, A, P(s,a), r(s,a)$ where S is the set of states of the environment, A describes the set of actions or transitions between states, $s'{\sim}P(s,a)$ describes the probability distribution of next states given the current state and action and $r(s,a) : S \times A \rightarrow R$ is the reward of taking action a in state s. Given an MDP, the goal of the agent is to gain the largest possible aggregate reward. The objective of an RL algorithm associated with an MDP is to find a decision policy $\pi^*(a|s) : s \rightarrow A$ that achieves this goal for that MDP:

$$\pi^* = \arg\max_{\pi} \mathbb{E}_{\tau\sim\pi(\tau)} \left[\sum_t r(s_t, a_t) \right] = \arg\max_{\pi} \sum_{t=1}^{T} \mathbb{E}_{(s_t,a_t)\sim\pi(s_t,a_t)} \left[r(s_t, a_t) \right]. \quad (4.1)$$

Deep RL leverages a neural network to learn the policy (and sometimes the reward function). Policy Gradient (PG) [379], for example, updates the policy directly by differentiating the aggregate reward $\mathbb{E}$ in Equation 4.1:

$$\nabla_\theta J = \frac{1}{N} \sum_{i=1}^{N} \left[(\sum_t \nabla_\theta log \pi_\theta(a_{i,t}|s_{i,t}))(\sum_t r(s_{i,t}, a_{i,t})) \right] \quad (4.2)$$

and updating the network parameters (weights) in the direction of the gradient:

$$\theta \leftarrow \theta + \alpha \nabla_\theta J, \quad (4.3)$$

Note that PG is an on-policy method in that it uses decisions made directly by the current

policy to compute the new policy.

Over the past couple of years, a plethora of new deep RL techniques have been proposed [265, 328]. In this chapter, we mainly focus on Proximal Policy Optimization (PPO) [350], Asynchronous Advantage Actor-critic (A3C) [265].

PPO is a variant of PG that enables multiple epochs of minibatch updates to improve the sample complexity. Vanilla PG performs one gradient update per data sample while PPO uses a novel surrogate objective function to enable multiple epochs of minibatch updates. It alternates between sampling data through interaction with the environment and optimizing the surrogate objective function using stochastic gradient ascent. It performs updates that maximizes the reward function while ensuring the deviation from the previous policy is small by using a surrogate objective function. The loss function of PPO is defined as:

$$L^{CLIP}(\theta) = \hat{E}_t[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\varepsilon, 1+\varepsilon)\hat{A}_t)] \tag{4.4}$$

where $r_t(\theta)$ is defined as a probability ratio $\frac{\pi_\theta(\mathbf{a_t}|\mathbf{s_t})}{\pi_{\theta_{old}}(\mathbf{a_t}|\mathbf{s_t})}$ so $r(\theta_{old}) = 1$. This term penalizes policy update that move $r_t(\theta)$ from $r(\theta_{old})$. $\hat{A}_t$ denotes the estimated advantage that approximates how good $\mathbf{a_t}$ is compared to the average. The second term in the *min* function acts as a disincentive for moving r_t outside of $[1-\varepsilon, 1+\varepsilon]$ where ε is a hyperparameter.

A3C uses an actor (usually a neural network) that interacts with the critic, which is another network that evaluates the action by computing the value function. The critic tells the actor how good its action was and how it should adjust. The update performed by the algorithm can be seen as $\nabla_\theta log\pi_\theta(a_{i,t}|s_{i,t})\hat{A}_t$.

4.2.3 Evolutionary Algorithms

Evolutionary algorithms are another technique that can be used to search for the best compiler pass ordering. It contains a family of population-based meta-heuristic optimization algorithms inspired by natural selection. The main idea of these algorithms is to sample a population of solutions and use the good ones to direct the distribution of future generations. Two commonly used Evolutionary Algorithms are Genetic Algorithms (GA) [139] and Evolution Strategies (ES) [81].

GA generally requires a genetic representation of the search space where the solutions are coded as integer vectors. The algorithm starts with a pool of candidates, then iteratively evolves the pool to include solutions with higher fitness by the three following strategies: selection, crossover, and mutation. Selection keeps a subset of solutions with the highest fitness values. These selected solutions act as parents for the next generation. Crossover merges pairs from the parent solutions to produce new offsprings. Mutation perturbs the offspring solutions with a low probability. The process repeats until a solution that reaches the goal fitness is found or after a certain number of generations.

ES works similarly to GA. However, the solutions are coded as real numbers in ES. In addition, ES is self-adapting. The hyperparameters, such as the step size or the mutation probability, are different for different solutions. They are encoded in each solution, so good settings get to the next generation with good solutions. Recent work [334] has used ES to update policy weights for RL and showed it is a good alternative for gradient-based methods.

4.3 AutoPhase Framework for Automatic Phase Ordering

We leverage an existing open-source HLS framework called LegUp [61] that compiles a C program into a hardware RTL design. In [176], an approach is devised to quickly determine the number of hardware execution cycles without requiring time-consuming logic simulation. We develop our RL simulator environment based on the existing harness provided by LegUp and validate our final results by going through the time-consuming logic simulation. AutoPhase takes a program (or multiple programs) and intelligently explores the space of possible passes to figure out an optimal pass sequence to apply. Table 4.1 lists all the passes used in AutoPhase. The workflow of AutoPhase is illustrated in Figure 4-4.

4.3.1 HLS Compiler

AutoPhase takes a set of programs as input and compiles them to a hardware-independent intermediate representation (IR) using the Clang front-end of the LLVM compiler. Optimization and analysis passes act as transformations on the IR, taking a program as input

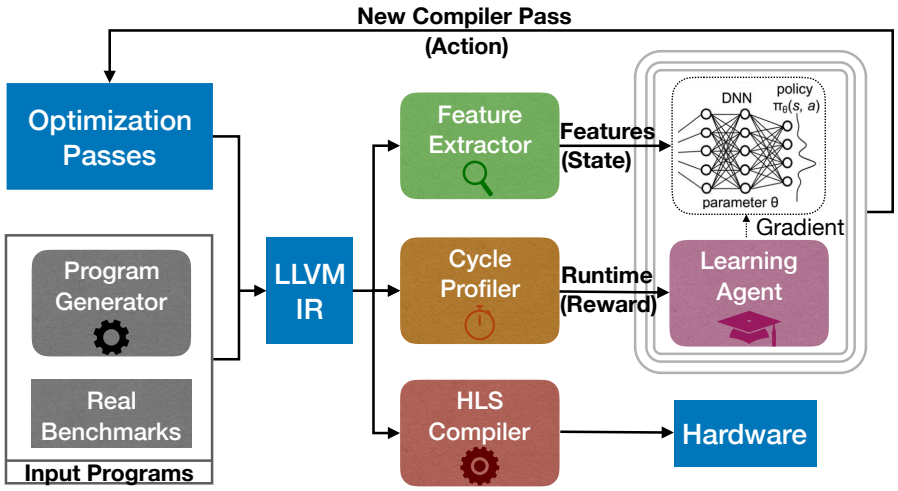


Figure 4-4: The block diagram of AutoPhase. The input programs are compiled to an LLVM IR using Clang/LLVM. The feature extractor and clock-cycle profiler are used to generate the input features (state) and the runtime improvement (reward), respectively from the IR. The input features and runtime improvement are fed to the deep RL agent as in input data to train on. The RL agent predicts the next best optimization passes to apply. After convergence, the HLS compiler is used to compile the LLVM IR to hardware RTL.

and emitting a new IR as output. The HLS tool LegUp is invoked after the compiler optimization as a back-end pass, which transforms LLVM IR into hardware modules.

4.3.2 Clock-cycle Profiler

Once the hardware RTL is generated, one could run a hardware simulation to gather the cycle count results of the synthesized circuit. This process is quite time-consuming, hindering RL and all other optimization approaches. Therefore, we approximate cycle count using the profiler in LegUp [176], which leverages the software traces and runs 20× faster than hardware simulation. In LegUp, the frequency of the generated circuits is set as a compiler constraint that directs the HLS scheduling algorithm. In other words, HLS tool will always try to generate hardware that can run at a certain frequency. In our experiment setting, without loss of generality, we set the target frequency of all generated hardware to 200MHz. We experimented with lower frequencies too; the improvements were similar but the cycle counts the different algorithms achieved were better as more logic could be fitted in a single cycle.

4.3.3 IR Feature Extractor

Wang *et al.* [412] proposed to convert a program into an observation by extracting all the features from the program. Similarly, in addition to the LegUp backend tools, we developed analysis passes to extract 56 static features from the program, such as the number of basic blocks, branches, and instructions of various types. We use these features as partially observable states for the RL learning and hope the neural network can capture the correlation of certain combinations of these features and certain optimizations. Table 4.2 lists all the features used.

4.3.4 Random Program Generator

As a data-driven approach, RL generalizes better if we train the agent on more programs. However, there are a limited number of open-source HLS examples online. Therefore, we expand our training set by automatically generating synthetic HLS benchmarks. We first generate standard C programs using CSmith [420], a random C program generator, which is originally designed to generate test cases for finding compiler bugs. Then, we develop scripts to filter out programs that take more than five minutes to run on CPU or fail the HLS compilation.

4.3.5 Overall Flow of AutoPhase

We integrate the compilation utilities into a simulation environment in Python with APIs similar to an OpenAI gym [55]. The overall flow works as follows:

1. The input program is compiled into LLVM IR using the Clang/LLVM.
2. The IR Feature Extractor is run to extract salient program features.
3. LegUp compiles the LLVM IR into hardware RTL.
4. The Clock-cycle Profiler estimates a clock-cycle count for the generated circuit.
5. The RL agent takes the program features or the histogram of previously applied passes and the improvement in clock-cycle count as input data to train on.
6. The RL agent predicts the next best optimization passes to apply.

Table 4.1: LLVM Transform Passes.

0	1	2	3	4	5	6	7
-correlated-propagation	-scalarrepl	-lowerinvoke	-strip	-strip-nondebug	-sccp	-globalopt	-gvn
8	9	10	11	12	13	14	
-jump-threading	-globaldce	-loop-unswitch	-scalarrepl-ssa	-loop-reduce	-break-crit-edges	-loop-deletion	
15	16	17	18	19	20	21	22
-reassociate	-lcssa	-codegenprepare	-memcpyopt	-functionattrs	-loop-idiom	-lowerswitch	-constmerge
23	24	25	26	27	28	29	30	31
-loop-rotate	-partial-inliner	-inline	-early-cse	-indvars	-adce	-loop-simplify	-instcombine	-simplifycfg
32	33	34	35	36	37	38	39	40	41
-dse	-loop-unroll	-lower-expect	-tailcallelim	-licm	-sink	-mem2reg	-prune-eh	-functionattrs	-ipsccp
			42	43	44	45			
			-deadargelim	-sroa	-loweratomic	-terminate			

Table 4.2: Program Features.

0	Number of BB where total args for phi nodes >5	28	Number of And insts
1	Number of BB where total args for phi nodes is [1,5]	29	Number of BB's with instructions between [15,500]
2	Number of BB's with 1 predecessor	30	Number of BB's with less than 15 instructions
3	Number of BB's with 1 predecessor and 1 successor	31	Number of BitCast insts
4	Number of BB's with 1 predecessor and 2 successors	32	Number of Br insts
5	Number of BB's with 1 successor	33	Number of Call insts
6	Number of BB's with 2 predecessors	34	Number of GetElementPtr insts
7	Number of BB's with 2 predecessors and 1 successor	35	Number of ICmp insts
8	Number of BB's with 2 predecessors and successors	36	Number of LShr insts
9	Number of BB's with 2 successors	37	Number of Load insts
10	Number of BB's with >2 predecessors	38	Number of Mul insts
11	Number of BB's with Phi node # in range (0,3]	39	Number of Or insts
12	Number of BB's with more than 3 Phi nodes	40	Number of PHI insts
13	Number of BB's with no Phi nodes	41	Number of Ret insts
14	Number of Phi-nodes at beginning of BB	42	Number of SExt insts
15	Number of branches	43	Number of Select insts
16	Number of calls that return an int	44	Number of Shl insts
17	Number of critical edges	45	Number of Store insts
18	Number of edges	46	Number of Sub insts
19	Number of occurrences of 32-bit integer constants	47	Number of Trunc insts
20	Number of occurrences of 64-bit integer constants	48	Number of Xor insts
21	Number of occurrences of constant 0	49	Number of ZExt insts
22	Number of occurrences of constant 1	50	Number of basic blocks
23	Number of unconditional branches	51	Number of instructions (of all types)
24	Number of Binary operations with a constant operand	52	Number of memory instructions
25	Number of AShr insts	53	Number of non-external functions
26	Number of Add insts	54	Total arguments to Phi nodes
27	Number of Alloca insts	55	Number of Unary operations

7.New LLVM IR is generated after the new optimization sequence is applied.

8.The machine learning algorithm iterates through steps (2)–(7) until convergence.

Note that AutoPhase uses the LLVM compiler and the passes used are listed in Table 4.2. However, adding support for any compiler or optimization passes in AutoPhase is very easy and straightforward. The action and state definitions must be specified again.

4.4 Correlation of Passes and Program Features

Similar to the case with many deep learning approaches, explainability is one of the major challenges we face when applying deep RL to the phase-ordering challenge. To analyze and understand the correlation of passes and program features, we use random forests [54] to learn the importance of different features. Random forest is an ensemble of multiple decision trees. The prediction made by each tree could be explained by tracing the decisions made at each node and calculating the importance of different features on making the decisions at each node. This helps us to identify the effective features and passes to use and show whether our algorithms learn informative patterns on data.

For each pass, we build two random forests to predict whether applying it would improve the circuit performance. The first forest takes the program features as inputs while the second takes a histogram of previously applied passes. To gather the training data for the forests, we run PPO with high exploration parameter on 100 randomly generated programs to generate feature–action–reward tuples. The algorithm assigns higher importance to the input features that affect the final prediction more.

4.4.1 Importance of Program Features

The heat map in Figure 4-5 shows the importance of different features on whether a pass should be applied. The higher the value is, the more important the feature is (the sum of the values in each row is one). The random forest is trained with 150,000 samples generated from the random programs. The index mapping of features and passes can be found in Tables 4.1 and 4.2. For example, the yellow pixel corresponding to feature index 17 and pass index 23 reflects that *number-of-critical-edges* affects the decision on whether to apply *-loop-rotate* greatly. A critical edge in control flow graph is an edge that is neither the only edge leaving its source block, nor the only edge entering its destination block. The critical edges can be commonly seen in a loop as a back edge so the number of critical edges might roughly represent the number of loops in a program. The transform pass *-loop-rotate* detects a loop and transforms a while loop to a do-while loop to eliminate one branch instruction in the loop body. Applying the pass results in better circuit performance

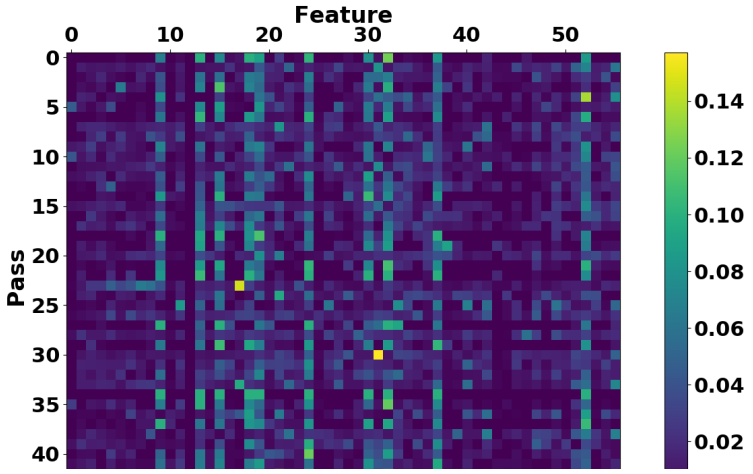


Figure 4-5: Heat map illustrating the importance of feature and pass indices.

as it reduces the total number of FSM states in a loop.

Other expected behaviors are also observed in this figure. For instance, the correlation between *number of branches* and the transform passes *-loop-simplify*, *-tailcallelism* (which transforms calls of the current function *i.e.*, self recursion, followed by a return instruction with a branch to the entry of the function, creating a loop), *-lowerswitch* (which rewrites switch instructions with a sequence of branches). Other interesting behaviors are also captured. For example, in the correlation between *binary operations with a constant operand* and *-functionattrs*, which marks different operands of a function as read-only (constant). Some correlations are harder to explain, for example, *number of BitCast instructions* and *-instcombine*, which combines instructions into fewer simpler instructions. This is actually a result of *-instcombine* reducing the loads and stores that call bitcast instructions for casting pointer types. Another example is *number of memory instructions* and *-sink*, where *-sink* basically moves memory instructions into successor blocks and delays the execution of memory until needed. Intuitively, whether to apply *-sink* should be dependent on whether there is any memory instruction in the program. Our last example to show is *number of occurrences of constant 0* and *-deadargelim*, where *-deadargelim* helped eliminate dead/unused constant zero arguments.

Overall, we observe that all the passes are correlated to some features and are able to affect the final circuit performance. We also observe that multiple features are not effective at directing decisions and training with them could increase the variance that would result

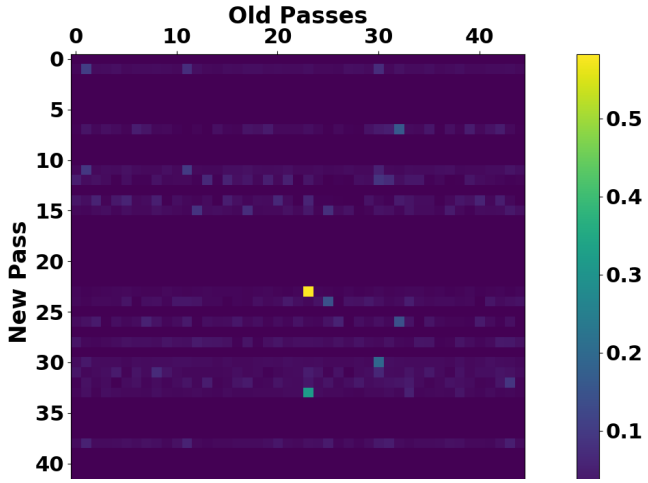


Figure 4-6: Heat map illustrating the importance of indices of previously applied passes and the new pass to apply.

in lower prediction accuracy of our results. For example, the total number of instructions did not give a direct indication of whether applying a pass would be helpful or not. This is because sometimes more instructions could improve the performance (for example, due to loop unrolling) and eliminating unnecessary code could also improve the performance. In addition, the importance of features varies among different benchmarks depending on the tasks they perform.

4.4.2 Importance of Previously Applied Passes

Figure 4-6 illustrates the impact of previously applied passes on the new pass to apply. The higher the value is, the more important having the old pass is. From this figure, we learn that for the programs we trained on passes *-scalarrepl, -gvn, -scalarrepl-ssa, -loop-reduce, -loop-deletion, -reassociate, -loop-rotate, -partial-inliner, -early-cse, -adce, -instcombine, -simplifycfg, -dse, -loop-unroll, -mem2reg, and -sroa*, are more impactful on the performance compared to the rest of the passes regardless of their order in the trajectory. Point (23,23) has the highest importance in which implies that pass *-loop-rotate* is very helpful and should be included if not applied before. By examining thousands of the programs, we find that *-loop-rotate* indeed reduces the cycle count significantly. Interestingly, applying this pass twice is not harmful if the passes were given consecutively. However, giving this pass twice with some other passes between them is sometimes very harmful. Another

interesting behavior our heat map captured is the fact that applying pass 33 (*-loop-unroll*) after (not necessarily consecutive) pass 23 (*-loop-rotate*) was much more useful compared to applying these two passes in the opposite order.

4.5 Problem Formulation

4.5.1 The RL Environment Definition

Assume the optimal number of passes to apply is N and there are K transform passes to select from in total, our search space $\mathcal{S}$ for the phase-ordering problem is $[0, K^N)$. Given M program features and the history of already applied passes, the goal of deep RL is to learn the next best optimization pass a to apply that minimizes the long term cycle count of the generated hardware circuit. Note that the optimization state s is partially observable in this case as the M program features cannot fully capture all the properties of a program.

Action Space – we define our action space $\mathcal{A}$ as $\{a \in \mathbb{Z} : a \in [0, K)\}$ where K is the total number of transform passes.

Observation Space – two types of input features were considered in our evaluation: ① **program features** $\mathbf{o_f} \in \mathbb{Z}^M$ listed in Table 4.2 and ② **action history** which is a histogram of previously applied passes $\mathbf{o_a} \in \mathbb{Z}^K$. After each RL step where the pass i is applied, we call the feature extractor in our environment to return new $\mathbf{o_f}$, and update the action histogram element o_{a_i} to $o_{a_i} + 1$.

Reward – the cycle count of the generated circuit is reported by the clock-cycle profiler at each RL iteration. Our reward is defined as $R = c_{prev} - c_{cur}$, where c_{prev} and c_{cur} represent the previous and the current cycle count of the generated circuit respectively. It is possible to define a different reward for different objectives. For example, the reward could be defined as the negative of the area and thus the RL agent will optimize for the area. It is also possible to co-optimize multiple objectives (e.g., area, execution time, power, etc.) by defining a combination of different objectives.

4.5.2 Applying Multiple Passes per Action

An alternative to the action formulation above is to evaluate a complete sequence of passes with length N instead of a single action a at each RL iteration. Upon the start of training a new episode, the RL agent resets all pass indices $\mathbf{p} \in \mathbb{Z}^N$ to the index value $\frac{K}{2}$. For pass p_i at index i, the next action to take is either to change to a new pass or not. By allowing positive and negative index update for each p, we reduced the total steps required to traverse all possible pass indices. The sub-action space a_i for each pass is thus defined as $[-1, 0, 1]$. The total action space $\mathcal{A}$ is defined as $[-1, 0, 1]^N$. At each step, the RL agent predicts the updates $[a_1, a_2, ..., a_N]$ to N passes, and the current optimization sequence $[p_1, p_2, ..., p_N]$ is updated to $[p_1 + a_1, p_2 + a_2, ..., p_N + a_N]$.

4.5.3 Normalization Techniques

In order for the trained RL agent to work on new programs, we need to properly normalize the program features and rewards so they represent a meaningful state among different programs. In this work, we experiment with two techniques: ① taking the logarithm of program features or rewards and, ② normalizing to a parameter from the original input program that roughly depicts the problem size. For technique ①, note that taking the logarithm of the program features not only reduces their magnitude, it also correlates them in a different manner in the neural network. Since, $w_1 \log(o_{f_1}) + w_2 \log(o_{f_2}) = log(o_{f_1}^{w_1} o_{f_2}^{w_2})$, the neural network is learning to correlate the products of features instead of a linear combination of them. For technique ②, we normalize the program features to the total number of instructions in the input program ($\mathbf{o_{f\_norm}} = \frac{\mathbf{o_f}}{o_{f_{51}}}$), which is feature #51 in Table 4.2.

4.6 Evaluation

To run our deep RL algorithms we use RLlib [239], an open-source library for reinforcement learning that offers both high scalability and a unified API for a variety of applications. RLlib is built on top of Ray [269], a high-performance distributed execution framework targeted at large-scale machine learning and reinforcement learning applications. We ran

Table 4.3: The observation and action spaces used in the different deep RL algorithms.

	RL-PPO1	RL-PPO2	RL-PPO3	RL-A3C	RL-ES
Deep RL Algorithm	PPO	PPO	PPO	A3C	ES
Observation Space	Program Features	Action History	Action History + Program Features	Program Features	Program Features
Action Space	Single-Action	Single-Action	Multiple-Action	Single-Action	Single-Action

the framework on a four-core Intel i7-4765T CPUwith a Tesla K20c GPUfor training and inference.

We set our frequency constraint in HLS to 200MHz and use the number of clock cycles reported by the HLS profiler as the circuit performance metric. In [176], results showed a one-to-one correspondence between the clock cycle count and the actual hardware execution time under certain frequency constraint. Therefore, better clock cycle count will lead to better hardware performance.

4.6.1 Performance

To evaluate the effectiveness of various algorithms for tackling the phase-ordering problem, we run them on nine real HLS benchmarks and compare the results based on the final HLS circuit performance and the sample efficiency against state-of-the-art approaches for overcoming the phase ordering, which include random search, Greedy Algorithms [176], OpenTuner [17], and Genetic Algorithms [119]. These benchmarks are adapted from CHStone [161] and LegUp examples. They are: *adpcm*, *aes*, *blowfish*, *dhrystone*, *gsm*, *matmul*, *mpeg2*, *qsort*, and *sha*. For this evaluation, the input features/rewards were not normalized, the pass length was set to 45, and each algorithm was run on a per-program basis. Table 4.3 lists the action and observation spaces used in all the deep RL algorithms.

The bar chart in Figure 4-7 shows the percentage improvement of the circuit performance compared to -O3 results on the nine real benchmarks from CHStone. The dots on the blue line in Figure 4-7 show the total number of samples for each program, which is the number of times the algorithm calls the simulator to gather the cycle count. `-O0` and `-O3` are the default compiler optimization levels. `RL-PPO1` is a PPO explorer where we set all the rewards to 0 to test if the rewards are meaningful. `RL-PPO2` is the PPO agent that learns the next pass based on a histogram of applied passes. `RL-A3C` is the A3C agent

that learns based on the program features. `Greedy` performs the greedy algorithm, which always inserts the pass that achieves the highest speedup at the best position (out of all possible positions it can be inserted to) in the current sequence. `RL-PPO3` uses a PPO agent and the program features but with the action space described in Section 4.5.2. explained in Section 4.5.2. `OpenTuner` runs an ensemble of six algorithms, which includes two families of algorithms: particle swarm optimization [207] and GA, each with three different crossover settings. `RL-ES` is similar to A3C agent that learns based on the program features, but updates the policy network using the evolution strategy instead of backpropagation. `Genetic-DEAP` [119] is a genetic algorithm implementation. `random` randomly generates a sequence of 45 passes at once instead of sampling them one-by-one.

From `Greedy`, we see that always adding the pass in the current sequence that achieves the highest reward leads to sub-optimal circuit performance. `RL-PPO2` achieves higher performance than `RL-PPO1`, which shows that the deep RL captures useful information during training. Using the histogram of applied passes results in better sample efficiency, but using the program features with more samples results in a slightly higher speedup. `RL-PPO2`, for example, at the minor cost of 4% lower speedup, achieves 50× more sample efficiency than `OpenTuner`. Using ES to update the policy is supposed to be more sample efficient for problems with sparse rewards like ours, however, our experiments did not benefit from that. Furthermore, `RL-PPO3` with multiple action updates achieves a higher speedup than the other deep RL algorithms with a single action. One reason for that is the ability of `RL-PPO3` to explore more passes per compilation as it applies multiple passes simultaneously in between every compilation. On the other hand, the other deep RL algorithms apply a single pass at a time.

4.6.2 Generalization

With deep RL, the search should benefit from prior knowledge learned from other different programs. This knowledge should be transferable from one program to another. For example, as discussed in section 4.4 applying pass *-loop-rotate* is always beneficial, and *-loop-unroll* should be applied after *-loop-rotate*. Note that the black-box search algorithms,

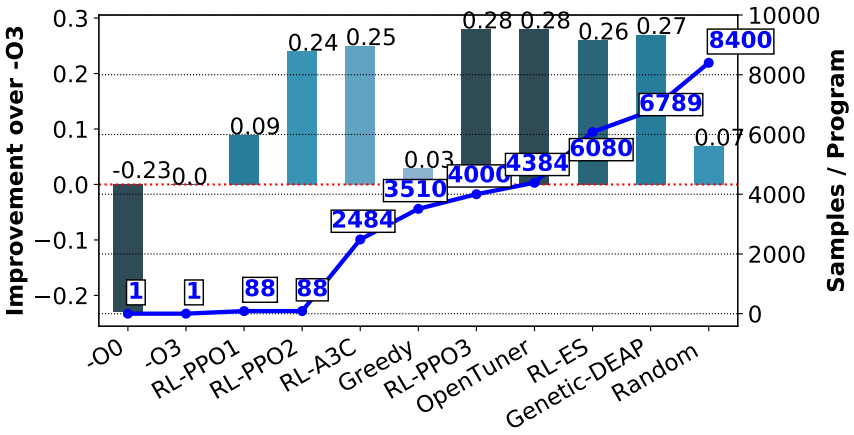


Figure 4-7: Circuit Speedup and Sample Size Comparison.

such as OpenTuner, GA, and greedy algorithms, cannot generalize. For these algorithms, rerunning a new search with many compilations is necessary for every new program, as they do not learn any patterns from the programs to direct the search and can be viewed as a smart random search.

To evaluate how generalizable deep RL could be with different programs and whether any prior knowledge could be useful, we train on 100 randomly-generated programs using PPO. Random programs are used for transfer learning due to lack of sufficient benchmarks and because it is the worst-case scenario, *i.e.*, they are very different from the programs that we use for inference. The improvement can be higher if we train on programs that are similar to the ones we inference on. We train a network with 256×256 fully connected layers and use the histogram of previously applied passes concatenated to the program features as the observation and passes as actions.

As described in Section 4.5.3, we experiment with two normalization techniques for the program features: ① taking the logarithm of all the program features and ② normalizing the program features to the total number of instructions in the program. In each pass sequence, the intermediate reward was defined as the logarithm of the improvement in cycle count after applying each pass. The logarithm was chosen so that the RL agent will not give much larger weights to big rewards from programs with longer execution time. Three approaches were evaluated: `filtered-norm1` uses the filtered (based on the analysis in Section 4.4 where we only keep the important features and passes) program features and passes from Section with normalization technique ①, `original-norm2` uses all the pro-

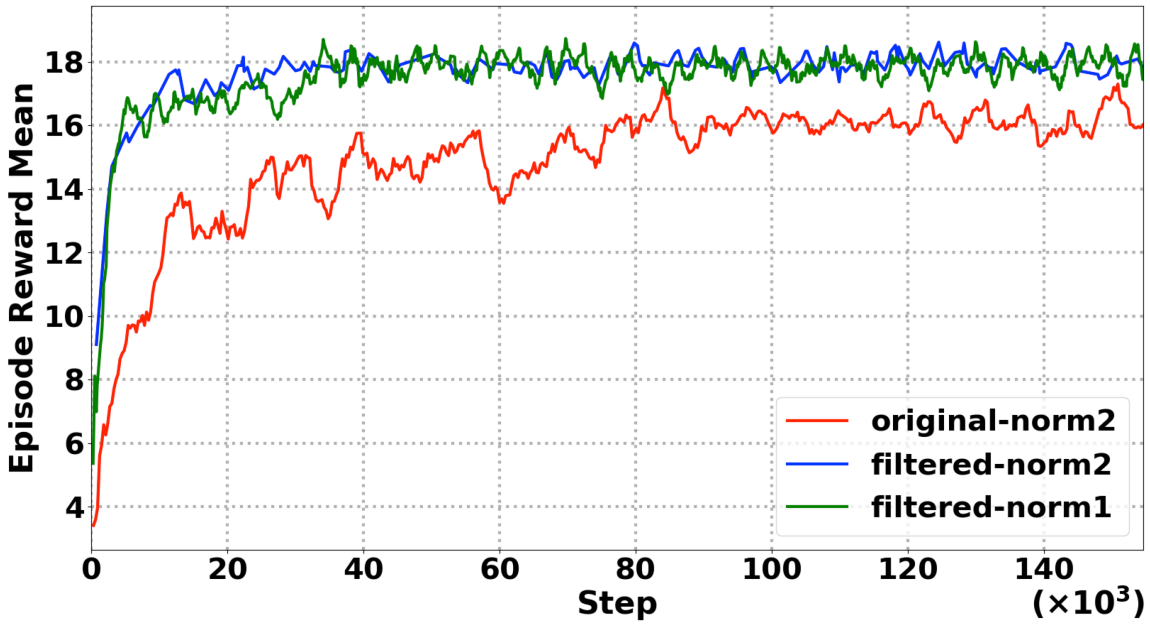


Figure 4-8: Episode reward mean as a function of step for the original approach where we use all the program features and passes and for the filtered approach where we filter the passes and features (with different normalization techniques). Higher values indicate faster circuit speed.

gram features and passes with normalization technique ②, and `filtered-norm2` uses the filtered program features and passes from Section 4.4 with normalization technique ②. Filtering the features and passes might not be ideal, especially when different programs have different feature characteristics and impactful passes. However, reducing the number of features and passes helps to reduce variance among all programs and significantly narrow the search space.

Figure 4-8 shows the episode reward mean as a function of the step for the three approaches. We observe that `filtered-norm2` and `filtered-norm1` converge much faster and achieve a higher episode reward mean than `original-norm2`, which uses all the features and passes. At roughly 8,000 steps the `filtered-norm2` and `filter-norm1` already achieve a very high episode reward mean, with minor improvements in later steps. Furthermore, the episode reward mean of the filtered approaches is still higher than that of `original-norm2` even when we allowed it to train for 20 times more steps (*i.e.*, 160,000 steps). This indicates that filtering the features and passes significantly improved the learning process. All three approaches learned to always apply pass *-loop-rotate*, and *-loop-*

unroll after *-loop-rotate*. Another useful pass that the three approaches learned to apply is *-loop-simplify*, which performs several transformations to transform natural loops into a simpler form that enables subsequent analyses and transformations.

We now compare the generalization results of `filtered-norm2` and `filtered-norm1` with the other black-box algorithms. We use 100 randomly-generated programs as the training set and nine real benchmarks from CHStone as the testing set for the deep RL-based methods. With the state-of-the-art black-box algorithms, we first search for the best pass sequences that achieved the lowest aggregated hardware cycle counts for the 100 random programs and then directly apply them to the nine test set programs. In Figure 4-9, the bar chart shows the percentage improvement of the circuit performance compared to -O3 on the nine real benchmarks, the dots on the blue line show the total number of samples each inference takes for one new program.

This evaluation shows that the deep RL-based inference achieves higher speedup than the predetermined sequences produced by the state-of-the-art black-box algorithms for new programs. The predetermined sequences that are overfitted to the random programs can cause poor performance in unseen programs (*e.g.,* -24% for `Genetic-DEAP`). Besides, normalization technique ② works better compared to normalization technique① for deep RL generalization (4% vs 3% speedup). This indicates that normalizing the different instructions to the total number of instructions *i.e.*, the distribution of the different instructions in Technique ② represents more universal characteristics across different programs, while taking the log in Technique ① only suppresses the value ranges of different program features. Furthermore, when we use other 12,874 randomly generated programs as the testing set with `filtered-norm2`, the speedup is 6% compared to -O3.

4.7 Conclusions

In this chapter, we propose an approach based on deep RL to improve the performance of HLS designs by optimizing the order in which the compiler applies optimization phases. We use random forests to analyze the relationship between program features and optimization passes. We then leverage this relationship to reduce the search space by identifying

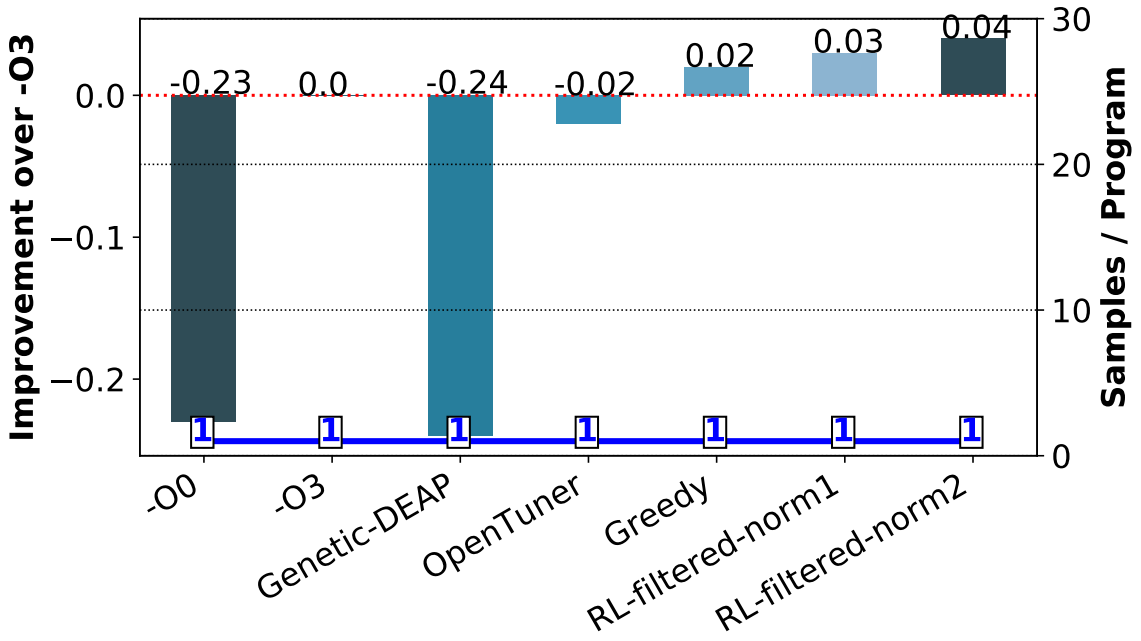


Figure 4-9: Circuit Speedup and Sample Size Comparison for deep RL Generalization.

the most likely optimization phases to improve the performance, given the program features. Our RL based approach achieves 28% better performance than compiling with the -O3 flag after training for a few minutes, and a 24% improvement after training for less than a minute. Furthermore, we show that unlike prior work, our solution shows potential to generalize to a variety of programs. While in this chapter we have applied deep RL to HLS, we believe that the same approach can be successfully applied to software compilation and optimization. Going forward, we envision using deep RL techniques to optimize a wide range of programs and systems.

Chapter 5

Enzyme: Compiler-based Automatic Differentiation

5.1 Introduction

Machine learning (ML) frameworks such as PyTorch [297] and TensorFlow [2] have become widespread as the primary workhorses of the modern ML community. Computing gradients necessary for algorithms such as backpropagation [166], Bayesian inference, uncertainty quantification [410], and probabilistic programming [87] requires all of the code being differentiated to be written in these frameworks. This is problematic for applying ML to new domains as existing tools like physics simulators [117, 56, 90, 91, 174], game engines, and climate models [369] are not written in the domain specific languages (DSL's) of ML frameworks. The rewriting required has been identified as the quintessential challenge of applying ML to scientific computing [25]. As stated by Rackauckas [317] "this is [the key challenge of scientific ML] because, if there is just one part of your loss function that isn't AD-compatible, then the whole network won't train."

To remedy this issue, the trend has been to either create new DSL's [174, 90, 237] that make the rewriting process easier or to add differentiation as a first-class construct in programming languages [255, 53, 413, 185]. This results in efficient gradients, but still requires rewriting in either the DSL or the differentiable programming language. Developers

may want to use code foreign to a ML framework to either re-use existing tools or write loss functions in a language with an easier abstraction for their use case. While there exist reverse-mode automatic differentiation (AD) frameworks for various languages, using them automatically on foreign code for an ML framework is difficult as they still require rewriting and have limited support for cross-language AD and libraries[413, 171, 163, 184]. The two primary approaches to computing gradients are as follows.

Operator-overloading computes derivatives by providing differentiable versions of existing language constructs. Examples include Adept [171]/ADOL-C [145], C++ libraries providing differentiable types; and JAX [53]/Autograd [255], Python libraries providing derivatives of NumPy-style functions. These approaches, however, require rewriting programs to use differentiable operators in place of standard language utilities. This prevents differentiation of many libraries and code in other languages.

Source-rewriting [147] analyzes the source code of programs and emits source code defining the gradient. Examples of tools include Tapenade [163, 295] for C and Fortran; ADIC [281] for C and C++; and Zygote [184, 186, 185] for Julia. Users must provide all code being differentiated to the tool ahead-of-time and must write programs in a specific subset of the language. This makes source-rewriting hard to use with header-only libraries and impossible to use with precompiled libraries.

Both operator-overloading and source-rewriting AD systems differentiate programs before optimization. Performing AD on unoptimized programs, however, may result in complicated gradients that cannot be simplified by future optimization. As an example, the gradient of `norm` in Figure 5-1 runs in $O(N)$ if optimization is run before AD and $O(N^2)$ if optimization is run after AD.

Traditional AD systems have not operated on optimized intermediate representation (IR) as doing so requires either re-implementing all of the optimizations or working at a low-level after which optimization has already been performed. Conventional wisdom says that producing efficient gradients for low-level IR is difficult as it lacks high-level information many tools rely upon: "AD is more effective in high-level compiled languages (e.g. Julia, Swift, Rust, Nim) than traditional ones such as C/C++, Fortran and LLVM IR [...]" – Innes [184]. This chapter challenges that wisdom by creating an efficient AD tool

```
float mag(const float* x);// Compute magnitude in O(N)
void norm(float* out, float* in) {
    // LICM optimization can move outside the loop
    // float res = mag(in);
    for(int i=0; i<N; i++) {
      out[i] = in[i]/mag(in);
    }
}
```

```
// LICM, then AD, O(N)
void ∇norm(float* out, float* d_out,
           float*  in, float* d_in) {
  float res = mag(in);
  for (int i=0; i<N; i++) {
    out[i] = in[i]/res;
  }
  float d_res = 0;
  for (int i=N-1; i>=0; i--) {
    d_res += -in[i]*in[i]/res*d_out[i];
    d_in[i] += d_out[i]/res;
  }
  ∇mag(in, d_in, d_res);
}
```

```
// AD, then LICM O(N^2)
void ∇norm(float* out, float* d_out,
           float*  in, float* d_in) {
  float res = mag(in);
  for (int i=0; i<N; i++) {
    out[i] = in[i]/res;
  }
  for (int i=N-1; i>=0; i--) {
    float d_res = -in[i]*in[i]/res \
                        * d_out[i];
    d_in[i] += d_out[i]/res;
    ∇mag(in, d_in, d_res);
  }
}
```

Figure 5-1: ***Top:*** An $O(N^2)$ function `norm` which normalizes a vector. Running loop-invariant-code-motion (LICM) [275, Sec. 13.2] moves the $O(N)$ call to `mag` outside the loop, reducing `norm`'s runtime to $O(N)$. ***Left:*** An $O(N)$ `∇norm` resulting from running LICM before AD. Both `mag` and its adjoint `∇mag` are outside the loop. ***Right:*** An $O(N^2)$ `∇norm` resulting from running LICM after AD. `∇mag` remains inside the loop as it uses a value computed inside the loop, making LICM illegal.

for LLVM [230], a low-level IR and set of optimizations used by many compilers.

This chapter presents Enzyme, an efficient cross-platform compiler plugin for automatic differentiation that operates on LLVM IR [230] and makes the following contributions:

- Enzyme, a compiler plugin for LLVM that can synthesize fast gradients of statically analyzable LLVM IR, including IR generated by compiler frontends for C, C++, Fortran, Rust, Swift, etc.
- PyTorch-Enzyme/TensorFlow-Enzyme, a foreign-function interface that allows machine learning researchers to use foreign code written in LLVM-compiled languages in PyTorch and TensorFlow.
- Enzyme.jl, a Julia package that uses Enzyme to synthesize gradients of code written in a dynamic high-level language using only low-level information.
- Multisource AD and static library support by leveraging link-time optimization (LTO) [230, 197].
- A study demonstrating that running AD after optimization results in significant performance gains on a standard machine learning benchmark suite [361] and achieves state-of-the-art performance.

Related work Clad is a plugin to the Clang compiler that implements forward mode automatic differentiation on a subset of C/C++ with reverse mode in development [398]. Chen et al. [72] present an end-to-end differentiable model for protein structure prediction. DiffTaichi [174] implements a differentiable DSL for physics and robotics simulation. de Avila Belbute-Peres et al. [90] also provide a differentiable physics framework. Halide is a differentiable DSL for image processing [237]. Swift implements first class automatic differentiation [413]. Elliot [108] present a compiler plugin to provide differentiable programming in Haskell. Enzyme differs from the related work by running on generic low-level IR and post-optimization. This gives Enzyme several performance and compatibility benefits that don't exist in current systems.

5.2 Design

Enzyme is composed of three stages: ***type analysis*** determines the underlying types of

```
void f(void* dst, void* src) { memcpy(dst, src, 8); }
```

```
// Gradient memcpy for double inputs
void ∇f(double* dst, double* ddst,
        double* src, double* dsrc) {
  // Forward pass
  memcpy(dst, src, 8);
  // Reverse pass
  dsrc[0] += ddst[0];
  ddst[0] = 0;


}
```

```
// Gradient memcpy for float inputs
void ∇f(float* dst, float* ddst,
        float* src, float* dsrc) {
  // Forward pass
  memcpy(dst, src, 8);
  // Reverse pass
  dsrc[0] += ddst[0];
  ddst[0] = 0;
  dsrc[1] += ddst[1];
  ddst[1] = 0;
}
```

Figure 5-2: ***Top:*** Call to `memcpy` for an unknown 8-byte object. ***Left:*** Gradient for a `memcpy` of 8 bytes of double data. ***Right:*** Gradient for a `memcpy` of 8 bytes of float data.

values, ***activity analysis*** determines what instructions and values can impact the gradient calculation, and ***synthesis*** creates the necessary functions to compute the gradient. A core design goal of Enzyme is to operate upon optimized IR. As seen in Figure 5-1 this can result in significant benefits such as simpler and more optimized gradients, though it requires working on a low-level representation. Gradients synthesized by Enzyme contain two parts: a ***forward pass*** that mirrors the original code and a ***reverse pass*** that computes the gradient by inverting the instructions in the forward pass. Inverted instructions in the reverse pass are known as ***adjoint***s. For all differentiable instructions in LLVM, Enzyme defines an adjoint to describe how gradients propagate through each instruction.

Type Analysis One challenge of performing AD on LLVM IR (and even C/C++) is that LLVM types do not necessarily represent the type of the underlying data. For example, the `memcpy` function copies data between generic pointers without types (`void*`). Creating a correct gradient for `memcpy`, however, requires knowing the type of the memory being copied. As shown in Figure 5-2, copying 8 bytes of double data requires performing one double (8-byte) addition in the reverse pass, whereas copying 8 bytes of float data requires two float (4-byte) additions. These operations are incompatible, resulting in an incorrect gradient if the wrong one is used.

Since Enzyme works on a low-level representation, Enzyme must use a new interprocedural fixed-point analysis rather than relying on types prescribed by the language. Every

value in a function is given a ***type tree*** that describes the known type at any given byte offset in the value. If the type at a particular offset is a pointer type, we have a new type tree that represents the types inside that offset. An example type tree is shown in Figure 5-3.

Type analysis initializes the type trees of all values to empty and uses type-based alias analysis (TBAA) metadata to initialize the type trees of loads, stores, and `memcpy` operations. TBAA allows us to make assumptions about the underlying type because of strict aliasing [93, 82]. For every kind of instruction, Enzyme implements a type propagation rule that specifies how types flow through the instruction. As an example, if the result of a load is known to be type `T`, then the pointer loaded must be a pointer to `T` at offset 0. Type analysis then runs all of the type propagation rules until a fixed point is reached. This is an application of abstract interpretation [85].

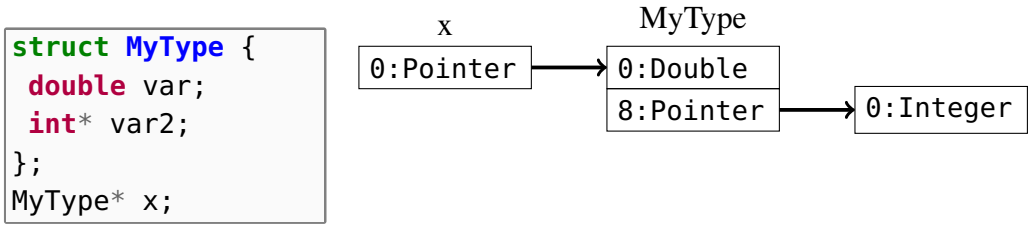


Figure 5-3: An example TypeTree used by Type Analysis. The variable `x` (declared on the left) is a pointer type, which points to a struct MyType, which contains a double at byte 0, and then a pointer at byte 8. That nested pointer points to an integer.

Sometimes Type Analysis cannot deduce all the necessary information statically (e.g. if bithacks to modify a floating-point). Rather than produce incorrect code, Enzyme will emit a compile-time error if it is unable to perform an analysis needed by AD. This enables programmers to provide this information to the compiler in the form of additional attributes, a custom derivative, or other means.

Activity Analysis Activity analysis determines what instructions could impact the gradient computation and is common in automatic differentiation systems to avoid performing unnecessary adjoints [353, 43]. Enzyme also uses activity analysis to avoid taking gradients of "undifferentiable" instructions such as the `cpuid` instruction. An instruction is active if and only if it can propagate a differential value to its return or another memory location. For example, a function that counts the length of a an active input array would not be active. In our implementation of activity analysis, we leverage LLVM's alias analysis [9, Ch. 12]

```
double sum(double* x) {
  double total = 0;
  for(int i=0; i<10; i++)
    total += read() * x[i];
  return total;
}

void ∇sum(double* x,
          double* d_x) {
  double* readCache = malloc(10*8);
  for(int i=0; i<10; i++)
    readCache[i] = read();
  // reverse
  for(int i=10-1; i>=0; i--)
    d_x[i] += readCache[i];
  }
  free(readCache);
}
```

```
double g(double* x) { return *x * *x; }
void f(double* x) { *x = g(x); }

{/*return val*/double,/*cache*/double}
augmented_g(double* x) {
  return {x[0]*x[0], x[0]};
}

void rev_g(double* x, double* d_x,
           double d_ret, double cache) {
  d_x[0] += 2 * cache * d_ret;
}

void ∇f(double* x, double* d_x) {
  {call, cache} = augmented_g(x);
  *x = call;
  double d_ret = *d_x;
  *d_x = 0;
  rev_g(x, d_x, d_ret, cache);
}
```

Figure 5-4: ***Left:*** Caching the result of `read` for the reverse pass. ***Right:*** Creating an augmented forward pass for a function to ensure requisite values are cached for the reverse.

and type analysis to help prove that instructions are inactive. As an example, any read-only function that returns an integer must be inactive since it cannot propagate differential values through the return or any memory location. This is true because the differential value of any integer value must be zero and while the instruction can read active memory it cannot propagate it anywhere.

Shadow Memory Shadow memory is common in AD systems as a way to store gradients of values. Consider the gradient of `sum` in the left of Figure 5-4. The gradient function `∇sum` takes in both `x` as an argument as well as the shadow `d_x`, where it will store the result. Enzyme's scheme is designed to be amenable to optimizations in LLVM while maintaining sufficient flexibility to represent arbitrary programs. For every active value in the forward pass, Enzyme creates and zeros a shadow version of that value. Similarly, any data structures (including function arguments) need to be duplicated. For any data structures computed inside the function being differentiated, Enzyme will create a shadow data structure automatically. This involves duplicating any memory instructions such as `malloc`, `new`, and stores of pointers, with equivalent shadow memory operations.

Finally, Enzyme delays all deallocations until the memory is not needed by the gradient calculation. Shadow memory is used to compute the adjoint of instructions like `load` in the reverse pass, which propagates the gradient of the load to the shadow of the pointer operand. Given shadow versions of all arguments and active globals, the shadow version of any value can be computed by duplicating the instruction that created the original value, replacing operands with their shadow. For calls to functions, we return the shadow pointer along with the original pointer.

Synthesis Given the results of type and activity analysis, Enzyme can now perform synthesis, the creation of the gradient function. Enzyme initializes all the shadow values as described above. For every basic block `BB` in the original program, Enzyme creates a corresponding reverse block `reverse_BB`. Enzyme then emits the adjoint of all instructions from `BB` into `reverse_BB` in reverse order. Enzyme then branches to the reverse of `BB`'s predecessor, returning if `BB` was the entry block. Finally, Enzyme replaces any return instruction in the forward pass with a branch to its reverse block. An example of this procedure is shown in Figure 5-5.

Cache Computing adjoints of certain instructions requires values computed in the forward pass. By default, Enzyme will attempt to recompute these in the reverse pass. However, it may be impossible or less efficient to recompute certain instructions. The question of whether and how to cache is known as the well-studied "checkpointing" problem in the literature [146, 224]. Checkpointing in Enzyme adds additional complexity with the inclusion of potentially-aliasing memory, a cost model for LLVM instructions (many of which are cost-free), and the impact of checkpointing on future optimization.

Consider the calls to `read` on the left of Figure 5-4, which cannot be recomputed. Enzyme provides a cache (often referred to as a tape in other AD systems) that provides forward-pass values to the reverse pass. In this example, Enzyme allocates memory (in this case an array of 10 doubles) to store the values needed by the reverse pass. If Enzyme can statically bound the number of values needing to be cached (e.g. a loop of fixed size), it will perform a single allocation to cache that instruction. If not, Enzyme will dynamically reallocate memory. For function calls, Enzyme may need to augment a call in the forward pass as shown in the right of Figure 5-4 to save values needed to compute the adjoint of the

```
define double @relu3(double %x)
entry:
  ; Shadow values for reverse
  ; alloca %d_x = 0.0
  ; alloca %d_call = 0.0
  ; alloca %d_result = 0.0
  br (%x > 0), if.true, if.end
if.true:
  %call = @pow(%x, 3)
  br cond.end
if.end:
  %res = phi[%call, %if.true],
 ↪  [0, %entry]
  ret %res
;
```

```
reverse_if.end:
  ; adjoint of return
  store %d_res = 1.0
  ; adjoint of %res phi node
  %d_call += if %x > 0, (load %d_res), else 0
  store %d_res = 0.0
  br %cmp, %reverse_if.true, %reverse_entry
reverse_if.true:
  ; adjoint of %call
  %df = 3 * @pow(%x, 2)
  %d_x += %df * (load %d_call)
  store %d_call = 0.0
  br %reverse_entry
reverse_entry:
  %0 = load %d_x
  ret %0
```

Figure 5-5: Example gradient synthesis for `relu(pow(x,3))`. The left hand side shows the LLVM IR for the original computation. In the comments on the left we show the shadow allocations of active variables that would be added to the forward pass. The right hand side shows the reverse pass that Enzyme would generate. The full synthesized gradient function would combine these (with shadow allocations added), replacing the return in `if.end` with a branch to `reverse_if.end`.

call.

To maximize performance, it is often desirable to reduce the number of values cached and Enzyme contains optimizations to reduce the number of values that need caching. Enzyme greatly benefits from LLVM's alias analysis and function attributes by proving that it is legal to recompute certain instructions. Enzyme also runs a differential-use analysis to determine which values are not necessary for computing the gradient and avoids caching them. This analysis is sometimes referred to as "to be recorded analysis" in other systems [164]. Additionally, if Enzyme already cached an equivalent value (e.g. a load to the same location which couldn't have since been written to), Enzyme simply reuses the existing cache for that value. Finally, if a cached value A is only used to recompute a single value B in the reverse pass, Enzyme will choose to cache the value B instead of the value A, minimizing the amount of work in the reverse pass.

Function Calls It is desirable to compute both the forward and reverse pass in the same function. This allows for optimization between the forward and reverse pass, and

can reduce memory usage. Enzyme detects whether it is legal to move the forward pass instructions of a function into the adjoint computation. If so, the forward pass call is erased and the combined function is used as the adjoint.

An ***indirect function call*** is a call to an anonymous function pointer which is not known at compile time. Like all other active pointers in a function, there exists a shadow version of the function pointer being called. Whenever a function pointer is used outside of a static call, we create a new global variable containing a pair of functions, namely the augmented forward and reverse pass. This global is then used as the shadow pointer for the original function. Thus, whenever Enzyme needs to perform an adjoint of an active indirect function call, it extracts the augmented forward and gradient functions from the shadow of the indirect callee, then uses those functions in the adjoint. Like the rest of shadow memory, this is handled automatically by Enzyme for all objects created inside functions being differentiated. If you want Enzyme to differentiate a function with a virtual C++ class as an argument, however, you need to pass in a modified virtual method table in the shadow that conforms with Enzyme's calling convention.

Limitations Enzyme needs access to the IR for any function being differentiated to create adjoints. This prevents Enzyme from differentiating functions loaded or created at runtime like a shared library or self-modifying code. Enzyme also must be able to deduce the types of active memory operations and phi nodes. Practically, this means enabling TBAA for your language and limiting yourself to programs with statically-analyzable types (no unions of differing types nor copies of undefined memory). Enzyme presently does not implement adjoints of exception-handling instructions so exceptions should be disabled (e.g. with `-fno-exceptions` for a C++ compiler).

5.3 Usage

Enzyme is designed to simplify both importing foreign code into machine-learning workflows and providing native AD for LLVM-based languages. Enzyme is implemented as an LLVM compiler plug-in, allowing it to be easily used in existing tools without need to build and maintain custom forks of LLVM, PyTorch, or TensorFlow.

```
__attribute__((
  enzyme("augment", augment_f),
  enzyme("gradient", gradient_f)
))
double f(double in);
double func(double* x, double* y) {
    return f(*x) + f(*y);
}
```

```
double dfunc(double* x, double *d_x,
             double* y) {
    __enzyme_autodiff(func,
       // The variable x is active
       enzyme_dup, x, d_x,
       // The variable y is constant
       enzyme_const, y);
}
```

Figure 5-6: ***Left:*** Specifying a custom forward and reverse pass for `f`. ***Right:*** Creating a gradient for func with x as an active variable and y as a constant.

Static Languages Using gradients inside LLVM-based languages simply requires calling an external `__enzyme_autodiff` function as shown on the right in Figure 5-6. For added control, users may specify whether a variable is active by including either an Enzyme-specific variable or metadata as part of the function call. Enzyme requires the IR for all functions it may need to differentiate to be available when the pass is run. For single-source programs, all the IR is simply available. Codebases with multiple source files or those using external libraries require an additional step. Enzyme makes use of Link-Time Optimization (LTO) [197, 230], a compiler technique for whole-program optimization that preserves IR from all source files until link time where a final set of interprocedural optimizations may run. To use Enzyme on multi-source codebases, a user enables LTO and runs Enzyme on the merged IR for all the sources. Static libraries are handled by compiling them with the `-fembed-bitcode` command that ensures that bitcode is included in the library as well. This allows Enzyme to perform AD on a program linking against a static library, by extracting the bitcode in the static library and then running Enzyme on the original program with the IR of the static library.

Programmers can use custom forward and backward passes in Enzyme by specifying them as metadata on the function to be differentiated, even if the definition of that function is not available during AD. In a separate Clang C/C++ frontend extension, we allow users to specify this directly with function attributes as on the left in Figure 5-6. Internally, one can also specify the type propagation, activity analysis, and adjoint rules for custom foreign functions. To minimize the amount of work for users, we provide these rules for common functions in the C/C++ standard and math libraries.

```julia
function f(x)
  sum = zero(x)
  for i = 1:10^7
    sum += x^i / i
  end
  return sum
end
```

Tool	Runtime (s)
Enzyme.jl	0.810
Zygote.jl	24.638
AutoGrad.jl	609.256

```julia
using Zygote, Enzyme

Zygote.@adjoint f(x),
  Enzyme.pullback(f, x)

Zygote.gradient(f, 0.5)
```

Figure 5-7: ***Left:*** A simple scalar function computing a Taylor expansion. ***Center:*** The runtime of the gradient as computed by Enzyme.jl and two common Julia AD frameworks. ***Right:*** How Enzyme can be embedded in existing AD frameworks to use Enzyme's efficient implementation of scalars.

Dynamic Languages Dynamic languages such as Julia require more consideration. Julia uses LLVM to perform native code generation for functions as a Just-In-Time compiler. The IR for all code needed by Enzyme is not immediately available since Julia's execution engine uses caching aggressively. We use the infrastructure developed for Julia's GPU code generator [40, 39] to collect all the function definitions reachable by the function to be differentiated. Julia implements its own version of common math functions like `sin` with custom implementations that are not amenable to type analysis, or resolves them to indirect function calls through opaque pointers into `libm`. `Enzyme.jl` uses Enzyme-specific LLVM metadata to mark these functions as behaving "sin-like". The Enzyme plugin is loaded and the Enzyme pass directly executed over the collected IR.

Zygote [184, 186, 185] is a popular automatic-differentiation framework for Julia used in probabilistic programming [128] and scientific machine learning [318]. Zygote performs source-to-source AD on high-level Julia code with optimizations for matrix programs. As shown in Figure 5-7, however, it can perform poorly on scalar programs. By embedding Enzyme inside Zygote as shown in the right of Figure 5-7, Julia is able to perform AD with both high-level knowledge and low-level optimizations. By utilizing embedded bitcode, `Enzyme.jl` provides the ability to take derivatives of foreign functions.

ML Frameworks Having demonstrated the ability to synthesize gradients of functions in a variety of languages compiled by LLVM, we will demonstrate how to leverage this ability to embed foreign code into a machine learning framework. After specifying the desired gradient by calling `__enzyme_autodiff` as shown in Figure 5-8, users can follow

```
// Input tensor + size, and output tensor
void f(float* inp, size_t n, float* out);
void diffef(float* inp, float* d_inp, size_t n, float* d_out) {
  // enzyme_dupnoneed specifies not recomputing the output
  __enzyme_autodiff(f, enzyme_dup, inp, d_inp,
                       n,
                       enzyme_dupnoneed, (float*)0, d_out);
}
```

```
import torch
from torch_enzyme import enzyme
# Create some initial tensor
inp = ...
# Apply foreign function to tensor
out = enzyme("test.c", "f").apply(inp)
# Derive gradient
out.backward()
print(inp.grad)
```

```
import tensorflow as tf
from tf_enzyme import enzyme

inp = tf.Variable(...)
# Use external C code as a TF op
out = enzyme(inp, filename="test.c",
                  function="f")
# Results is a TF tensor
out = tf.sigmoid(out)
```

Figure 5-8: ***Top:*** Sample glue code for using Enzyme to produce a custom operator for an ML framework. ***Left & Right:*** Sample code of using Enzyme to provide gradients of foreign code in PyTorch and TensorFlow, respectively.

the tutorials for creating a custom operator in PyTorch [84] or TensorFlow [83] and compiling the custom operator with Enzyme as described above. To simplify this workflow for machine learning researchers, we also created a simple package for PyTorch and TensorFlow in Figure 5-8 that exposes this functionality in Python without needing to compile a custom operator.

5.4 Evaluation

We evaluate the Enzyme approach by measuring the run time of seven benchmarks: the three reverse-mode automatic differentiation benchmarks from Microsoft's machine learning-focused ADBench suite [361], and four additional tests that are technically interesting or represent potential uses of Enzyme in practice. The ADBench suite includes bundle analysis (BA), a long short term memory model (LSTM), and a gaussian mixture model (GMM). We also differentiate two integrators (Euler, RK4) from the Odeint header-only ODE solver library [8]; a simple Fast Fourier Transform (FFT); and a finite difference discretized sim-

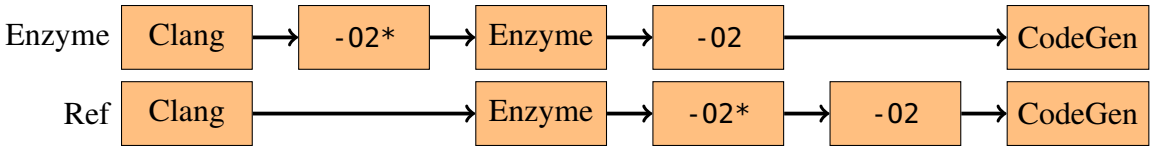


Figure 5-9: The pipelines Enzyme and Ref, which run optimizations before and after AD, respectively. The goal of running optimizations prior to AD is to reduce work and simplify the code. The first round of optimizations (`-O2*`) disables scheduling passes such as vectorization or unrolling that make heuristic decisions based on the current code size and machine attributes. Scheduling optimizations are included in the second round of optimizations (`-O2`) when the entire code (including gradient) is available.

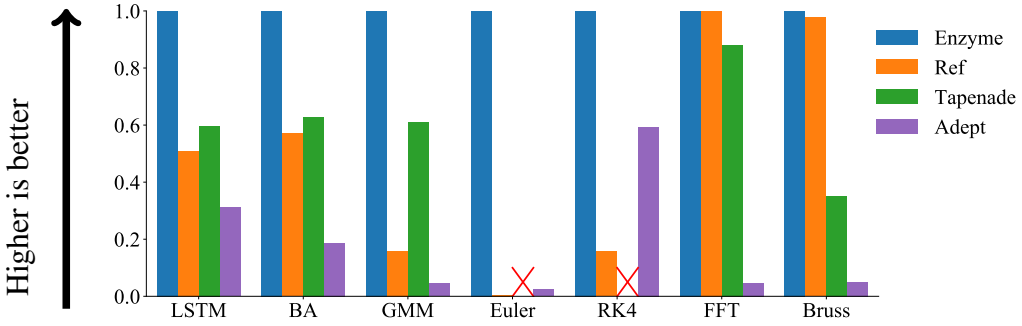


Figure 5-10: Relative speedup of different AD systems on the benchmark suite, higher is better. A red X is used to denote a system not being compatible with the benchmark (Tapenade only supports C and not C++ programs). For each benchmark, we take the geometric mean of the run time for all test cases, normalizing to the victor. A value of 1.0 denotes the fastest AD system tested for that benchmark, whereas a value of 0.5 denotes that an AD system produced a gradient which took twice as long.

ulation of the 2-dimensional Brusselator system (Bruss) [114, 423].

The two integrators test indirect function calls, complicated C++ headers, and foreign ODE solvers. The FFT test demonstrates AD of recursive functions. The Brusselator test demonstrates the utility in adjoint sensitivity analysis for ordinary differential equations, a widely applicable method with applications to PDE-constrained optimization [42, 236], control theory [302], and scientific machine learning like neural ODEs [318, 72].

We ran our experiments on a "quiesced" AWS c4.8xlarge instance with hyperthreading and Turbo Boost disabled. For all benchmarks, we took the geometric mean across all inputs. We ran all 92 inputs from ADBench, removing the 21 inputs where Adept exhausted system memory or a tool ran in under 0.01 seconds. For the integrator and FFT tests, we ran a total of 36 different inputs, with the number iterations or the input size increasing

	Enzyme	Ref	Tapenade	Adept
LSTM	**2.408**	4.727	4.033	7.722
BA	**0.256**	0.450	0.408	1.380
GMM	**0.076**	0.480	0.125	1.677
Euler	**0.165**	29.453	N/A	6.954
RK4	**3.936**	25.015	N/A	6.632
FFT	**0.122**	**0.122**	0.139	2.632
Bruss	**0.180**	**0.184**	0.513	3.546

Table 5.1: Geometric mean runtime of benchmark suite in seconds. Tapenade compiles only C and not C++. N/A denotes a system incompatible with the benchmark (Tapenade only supports C and not C++ programs).

exponentially. For Bruss, we ran a total of 10 trials.

To evaluate the effectiveness of AD on optimized IR, we construct two pipelines shown in Figure 5-9. The Enzyme pipeline consists of running optimizations before Enzyme AD, followed by a second round of optimizations. The Reference (Ref) pipeline is identical to the Enzyme pipeline, except that AD is performed before the first round of optimization. This allows us to effectively evaluate the importance of optimization on AD without considering additional confounding factors (such as differing tape implementations) between Enzyme and existing source AD systems. Taking the geometric mean across all benchmarks and inputs, Enzyme outperforms Ref by a factor of 4.2.

We also compare against the two fastest C/C++ AD tools evaluated in ADBench, Tapenade and Adept[1]. These results are presented in Figure 5-10. Enzyme demonstrates state-of-the-art performance in all benchmarks. Enzyme's advantage in the BA, LSTM, Euler, and RK4 tests appears to stem from running optimizations before AD. Enzyme uses a different tape structure than Tapenade (using a recursive set of allocations rather than a stack), which explains their differences on the GMM and Bruss benchmarks. Enzyme does not need to store as much on its tape as Adept (such as not needing to store which statements were executed), explaining Enzyme's superior performance on FFT and Bruss.

[1]For ADBench benchmarks, Tapenade and Adept had their ADBench implementations were evaluated directly. Tapenade and Adept versions of benchmarks outside ADBench were generated via Tapenade's web interface or replacing programs with Adepts differentiable types, using Vector and Matrix extensions where relevant. All benchmarks are available on Github.

5.5 Conclusion

Enzyme demonstrates the feasibility of performing efficient AD on low-level programs, opening up the door for language-independent AD and AD after optimization. This transforms the existing workflow machine learning researchers use to bring ML to foreign code. Instead of rewriting foreign code for machine learning, they can automatically synthesize fast gradients! This allows researchers to apply ML to a vast array of new use cases without the substantial effort of a rewrite or new DSL.

Building Enzyme as part of the LLVM compiler creates many avenues for future research. Exploring new AD-specific optimizations in LLVM may yield additional performance benefits. One could use LLVM's existing GPU or parallel code generators on programs generated by Enzyme [152, 154]. Enzyme could be extended to differentiate GPU and CPU-parallel programs by using existing representations for these programs in LLVM [326, 172, 343, 95]. Enzyme could also be extended to support forward-mode AD, mixed-mode AD [324], and the checkpointing problem beyond a simple heuristic. Fine-tuning the location of Enzyme in LLVM's optimization pass pipeline remains an open question. Enzyme opens up opportunities for cross-language AD. There are also opportunities to use Enzyme to port various physics engines and other codebases to ML frameworks.

Chapter 6

Polygeist: Improving Polyhedral Scheduling Via High-Level Structure And Low-Level Optimization

6.1 Introduction

Improving the efficiency of computation has always been one of the prime goals of computing. Program performance can be improved significantly by reaping the benefits of parallelism, temporal and spatial locality, and other performance sources. Relevant program transformations are particularly tedious and challenging when targeting modern multicore CPUs and GPUs with deep memory hierarchies and parallelism, and are often performed automatically by optimizing compilers.

The polyhedral model enables precise analyses and a relatively easy specification of transformations (loop restructuring, automatic parallelization, etc.) that take advantage of hardware performance sources. As a result, there is growing evidence that the polyhedral model is one of the best frameworks for efficient transformation of compute-intensive programs [397, 99, 277], and for programming accelerator architectures [407, 406, 100]. Consequently, the compiler community has focused on building tools that identify and optimize parts of the program that can be represented within the polyhedral model (commonly

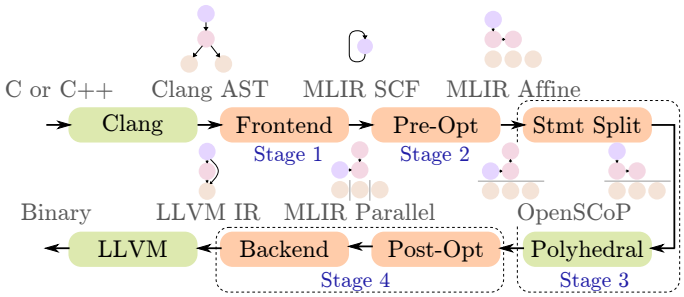


Figure 6-1: Polygeist flow consists of 4 stages. The frontend traverses Clang AST to emit MLIR SCF dialect (Section 6.3.1), which is raised to the Affine dialect and pre-optimized (Section 6.3.2). The IR is then processed by a polyhedral scheduler (Sections 6.3.3,6.3.4) before post-optimization and parallelization (Section 6.3.5). Finally, it is translated to LLVM IR for further optimization and binary generation by LLVM.

referred to as static-control parts, or SCoP's). Such tools tend to fall into two categories.

Compiler-based tools like Polly [149] and Graphite [307] detect and transform SCoPs in compiler intermediate representations (IRs). While this offers seamless integration with rest of the compiler, the lack of high-level structure and information hinders the tools' ability to perform analyses and transformations. This structure needs to be recovered from optimized IR, often imperfectly or at a significant cost [150]. Moreover, common compiler optimizations such as LICM may interfere with the process [223]. Finally, low-level IRs often lack constructs for, e.g., parallelism or reductions, produced by the transformation, which makes the flow more complex.

Source-to-source compilers such as Pluto [52], PoCC [304] and PPCG [406] operate directly on C or C++ code. While this can effectively leverage the high-level information from source code, the effectiveness of such tools is often reduced by the lack of enabling optimizations such as those converting hazardous memory loads into single-assignment virtual registers. Furthermore, the transformation results must be expressed in C, which is known to be complex [29, 151] and is also missing constructs for, e.g., reduction loops or register values not backed by memory storage.

This chapter proposes and evaluates the benefits of a polyhedral compilation flow, Polygeist (Figure 6-1), that can leverage both the high-level structure available in source code and the fine-grained control of compiler optimization provided by low-level IRs. It builds on the recent MLIR compiler infrastructure that allows the interplay of multiple

abstraction levels within the same representation, during the same transformations [229]. Intermixable MLIR abstractions, or *dialects*, include high-level constructs such as loops, parallel and reduction patterns; low-level representations fully covering LLVM IR [230]; and a polyhedral-inspired representation featuring loops and memory accesses annotated with affine expressions. Moreover, by combining the best of source-level and IR-level tools in an end-to-end polyhedral flow, Polygeist preserves high-level information and leverages them to perform new or improved optimizations, such as statement splitting and loop-carried value detection, on a *lower*-level abstraction as well as to influence downstream optimizations.

We make the following contributions:

- a C and C++ frontend for MLIR that preserves high-level loop structure from the original source code;
- an end-to-end flow with raising to and lowering from the polyhedral model, leveraging our abstraction to perform more optimizations than both source- and IR-level tools, including reduction parallelization;
- an exploration of new transformation opportunities created by Polygeist, in particular, statement splitting;
- and an end-to-end comparison between Polygeist and state-of-the-art source- and IR-based tools (Pluto [52] and Polly [151]) along with optimization case studies.

6.2 The MLIR Framework

6.2.1 Overview

MLIR is an optimizing compiler infrastructure inspired by LLVM [230] with a focus on extensibility and modularity [229]. Its main novelty is the IR supporting a fully extensible set of instructions (called *operations*) and types. Practically, MLIR combines SSA with nested regions, allowing one to express as first-class operations the concepts ranging from machine instructions such as floating-point addition to structured control flow such as loops,

```
%result = "dialect.operation"(%operand, %operand)
          {attribute = #dialect<"value">} ({
// Inside a nested region.
^basic_block(%block_argument: !dialect.type):
  "another.operation"() : () -> ()
}) : (!dialect.type) -> !dialect.result_type
```

Figure 6-2: Generic MLIR syntax for an operation with two operands, one result, one attribute and a single-block region.

from hardware circuitry [78] to large machine learning graphs. Operations define runtime semantics of a program and process immutable values. Compile-time information about values is expressed in *types*, and information about operations is expressed in *attributes*. Operations can have attached regions, which in turn contain (basic) blocks of further operations. The generic syntax, accepted by all operations, illustrates the structure of MLIR in Figure 6-2. Additionally, MLIR supports user-defined custom syntax.

Attributes, operations and types are organized in *dialects*, which can be thought of as modular libraries. MLIR provides a handful of dialects that define common operations such as modules, functions, loops, memory or arithmetic instructions, and ubiquitous types such as integers and floats. We discuss the dialects relevant to Polygeist in the following sections.

6.2.2 Affine and MemRef Dialects

The *Affine* dialect [264] aims at representing SCoP's with explicit polyhedral-friendly loop and conditional constructs. The core of its representation is the following classification of value categories:

- *Symbols*—integer values that are known to be loop-invariant but unknown at compile-time, also referred to as program *parameters* in polyhedral literature, typically array dimensions or function arguments. In MLIR, symbols are values defined in the top-level region of an operation with "affine scope" semantics, e.g., functions; or array dimensions, constants, and affine map (see below) application results regardless of their definition point.

- *Dimensions*—are an extension of symbols that also accepts induction variables of

affine loops.

- *Non-affine*—any other values.

Symbols and dimensions have `index` type, which is a platform-specific integer that fits a pointer (`intptr_t` in C).

MLIR provides two attributes relevant for the Affine dialect:

- *Affine maps* are multi-dimensional (quasi-)linear functions that map a list of dimension and symbol arguments to a list of results. For example, $(d_0, d_1, d_2, s_0) \rightarrow (d_0 + d_1, s_0 \cdot d_2)$ is a two-dimensional quasi-affine map, which can be expressed in MLIR as `affine_map<(d0,d1,d2)[s0] -> (d0+d1, s0*d2)>`. Dimensions and symbols are separated to allow quasi-linear expressions: symbols are treated as constants, which can therefore be multiplied with dimensions, whereas a product of two dimensions is invalid.

- *Integer sets* are collections of integer tuples constrained by conjunctions of (quasi-)linear expressions. For example, a "triangular" set $\{(d_0, d_1) : 0 \leq d_0 < s_0 \wedge 0 \leq d_1 \leq d_0\}$ is represented as `affine_set<(d0,d1)[s0]: (d0 >= 0, s0-d0-1 >= 0, d1 >= 0, d0-d1 >= 0)>`.

The Affine dialect makes use of the concepts above to define a set of operations. An `affine.for` is a "for" loop with loop-invariant lower and upper bounds expressed as affine maps with a constant step. An `affine.parallel` is a "multifor" loop nest, iterations of which may be executed concurrently. Both kinds of loops support reductions via loop-carried values as well as max(min) expression lower(upper) bounds. An `affine.if` is a conditional construct, with an optional `else` region, and a condition defined as inclusion of the given values into an integer set. Finally, `affine.load` and `affine.store` express memory accesses where the address computation is expressed as an affine map.

Figure 6-3 illustrates the Affine dialect for a polynomial multiplication, `C[i+j] += A[i] * B[j]`. This simple example highlights the fact that MLIR supports, and encourages, IRs from different dialects to be used together.

A core MLIR type—`memref`, which stands for **mem**ory **ref**erence—and the corresponding `memref` dialect are also featured in Figure 6-3. The `memref` type describes a

```
%c0 = constant 0 : index
%0 = memref.dim %A, %c0 : memref<?xf32>
%1 = memref.dim %B, %c0 : memref<?xf32>
affine.for %i = 0 to affine_map<()[s0] -> (s0)>()[%0] {
  affine.for %j = 0 to affine_map<()[s0] -> (s0)>()[%1] {
    %2 = affine.load %A[%i] : memref<?xf32>
    %3 = affine.load %B[%j] : memref<?xf32>
    %4 = mulf %2, %3 : f32
    %5 = affine.load %C[%i + %j] : memref<?xf32>
    %6 = addf %4, %5 : f32
    affine.store %6, %C[%i + %j] : memref<?xf32>
  }
}
```

Figure 6-3: Polynomial multiplication in MLIR using Affine and Standard dialects.

structured multi-index pointer into memory, e.g., `memref<?xf32>` denotes a 1-d array of floating-point elements; and the `memref` dialect provides memory and type manipulation operations, e.g., `memref.dim` retrieves the dimensionality of a `memref` object. `memref` does not allow internal aliasing, i.e., different indices always point to different addresses. This effectively defines away the delinearization problem that hinders the application of polyhedral techniques at the LLVM IR level [150]. Throughout this paper, we only consider `memrefs` with the default *layout* that corresponds to contiguous row-major storage compatible with C ABI. In practice, `memrefs` support arbitrary layouts expressible as affine maps, but these are not necessary in Polygeist context.

6.2.3 Other Relevant Core Dialects

MLIR provides several dozen dialects. Out of those, only a handful are relevant for our discussion:

- The *Structured Control Flow* (`scf`) dialect defines the control flow operations such as loops and conditionals that are not constrained by affine categorization rules. For example, the `scf.for` loop accepts any integer value as loop bounds, which are not necessarily affine expressions.

- The *Standard* (`std`) dialect contains common operations such as integer and float arithmetic, which is used as a common lowering point from higher-level dialects

before fanning out into multiple target dialects and can be seen as a generalization of LLVM IR [230].

- The *LLVM* dialect directly maps from LLVM IR instructions and types to MLIR, primarily to simplify the translation between them.

- The *OpenMP* dialect provides a dialect- and platform-agnostic representation of OpenMP directives such as "parallel" and "workshare loop", which can be used to transform OpenMP constructs or emit LLVM IR that interacts with the OpenMP runtime.

- The *Math* dialect groups together mathematical operations on integer and floating type beyond simple arithmetic, e.g., `math.pow` or `math.sqrt`.

6.3 An (Affine) MLIR Compilation Pipeline

The Polygeist pipeline consists of 4 components (Figure 6-1):

1. a frontend that allows entering MLIR at the SCF loops level from C or C++ code (Section 6.3.1);

2. a preprocessing step within MLIR that raises to the Affine dialect (Section 6.3.2);

3. a polyhedral scheduler of the Affine parts of the program *via* a round-trip to and from OpenSCoP (Section 6.3.3) and running Pluto transformations, controlled by the new statement splitting heuristic (Section 6.3.4);

4. a backend that runs postprocessing MLIR optimizations (section 6.3.5) and final lowering to an executable.

6.3.1 Frontend

Polygeist relies on the Clang AST to emit MLIR IRs. It thus avoids reimplementing parsing and language-level semantic analysis and handles modern C and C++ features. As is typical for compiler frontends, Polygeist creates a recursive symbol table data structure to

C type	LLVM IR type	MLIR type
`int`	`i32 (on machine X)`	`i32 (on machine X)`
`intNN_t`	`iNN`	`iNN`
`uintNN_t`	`iNN`	`uiNN`
`float`	`float`	`f32`
`double`	`double`	`f64`
`ty *`	`ty *`	`memref<?x ty>`
`ty &`	`ty *`	`memref<1x ty>`
`ty **`	`ty **`	`memref<memref<?x ty>>`
`ty[N][M]`	`[N x [M x ty]]*`	`memref<NxMx ty>`

Figure 6-4: Type correspondence between C, LLVM IR and MLIR types.

look up the correct variable for a given scope. Polygeist lazily registers all global variables and functions found in the AST to its symbol table before generating any code. Polygeist then traverses the call graph from a given entry function (`main` by default), creating and defining MLIR functions as necessary.

Control Flow & High Level Information

In contrast to traditional compiler pipelines, targeting a branch-based IR, Polygeist leverages the high-level MLIR operations such as `scf.while` (a looping construct) and `scf.if` (a conditional construct) within the SCF dialect to preserve the control flow structure of the source code. C-level `continue` and `break` constructs are handled by introducing signal variables and checking them before each operation that follows original constructs. Furthermore, within a `#pragma scop`, Polygeist assumes that the program is affine and uses an `affine.for` to represent loops directly.

Types & Polygeist ABI

While emitting operations, Polygeist must decide how to represent C or C++ types within MLIR. For primitive types such as `int` or `float`, Polygeist emits an MLIR variant of that type with the same width as would be used within LLVM/Clang. This allows Polygeist to keep the same Application Binary Interface (ABI) as code compiled by a normal C or C++ compiler when calling a function with only primitive types. On the other hand, for pointer, reference and array types, Polygeist uses `memref` type (Figure 6-4). This allows Polygeist to preserve more of the structure available within the original program (e.g.,

multi-dimensional arrays) and enables interaction with MLIR's high-level memory operations.

This represents a breaking change to the C ABI for any functions with pointer arguments. Polygeist addresses this by providing an attribute for function arguments and allocations to use a C-compatible pointer type rather than `memref`, applied by default to external functions such as `strcmp` and `scanf`. When calling a pointer-ABI function with a `memref`-ABI argument, Polygeist generates wrapper code that recovers the C ABI-compatible pointer from `memref` and ensures the correct result. Figure 6-5 shows an example demonstrating how the Polygeist and C ABI may interact for a small program.

When allocating and deallocating memory, this difference in ABI becomes significant. This is because allocating several bytes of an array with `malloc` then casting to a `memref` will not result in legal code (as `memref` itself may not be implemented with a raw pointer). Thus, Polygeist identifies calls to allocation and deallocation functions and replaces them with legal equivalents for `memref`.

Functions and global variables are emitted using the same name used by the C or C++ ABI. This ensures that all external values are loaded correctly, and multi-versioned functions (such as those generated by `C++` templates or overloading) have distinct names and definitions.

Instruction Generation

For most instructions, Polygeist directly emits an MLIR operation corresponding to the equivalent C operation (`addi` for integer add, `call` for function call, etc.). For some special instructions such as a call to `pow`, Polygeist chooses to emit a specific MLIR operation in the Math dialect, instead of a call to an external function (defined in libm). This permits such instructions to be better analyzed and optimized within MLIR.

Operations that involve memory or pointer arithmetic require additional handling. MLIR does not have a generic pointer arithmetic instruction; instead, it requires that `load` and `store` operations contain all of the indices being looked up. This presents issues for operations that perform pointer arithmetic. To remedy this, we use a temporary `subindex` operation for `memref`'s that curry the index. A subsequent optimization pass within Polygeist,

forwards the indices in a `subindex` to any `load` or `store` which uses them.

Local Variables

Local variables are handled by allocating a `memref` on stack at the top of a function. This permits the desired semantics of C or C++ to be implemented with relative ease. However, as many local variables and arguments contain `memref` types, this immediately results in a `memref` of a `memref`—a hindrance for most MLIR optimizations as it is illegal outside of Polygeist. As a remedy, we implement a heavyweight memory-to-register (mem2reg) transformation pass that eliminates unnecessary loads, stores, and allocations within MLIR constructs. Empirically this eliminates all `memref`s of `memref` in the Polybench suite.

6.3.2 Raising to Affine

The translation from C or C++ to MLIR directly preserves high-level information about loop structure and n-D arrays, but does not generate other Affine operations. Polygeist subsequently raises memory, conditional, and looping operations into their Affine dialect counterparts if it can prove them to be legal affine operations. If the corresponding frontend code was enclosed within `#pragma scop`, Polygeist assumes it is legal to raise all operations within that region. Any operations which are not proven or assumed to be affine remain untouched. We perform simplifications on affine maps to remove loops with zero or one iteration and drop branches of a conditional with a condition known at compile time.

Memory operations and loop bounds

To convert an operation, Polygeist replaces its bound and subscript operands with identity affine maps (`affine_map<() [s0]->(s0)>[%bound]`). It then folds the operations computing the map operands, e.g., `addi`, `muli`, into the map itself. Values that are transitively derived from loop induction variables become map dimensions and other values become symbols. For example, `affine_map< ()[s0]->(s0)>[%bound]` with `%bound = addi %N, %i`, where `%i` is an induction variable, is folded into `affine_map<(d0)[s0] ->(s0 + d0)>(%i)[%N]`. The process terminates when no operations can be folded or

when Affine value categorization rules are satisfied.

Conditionals

Conditional operations are emitted by the frontend for two input code patterns: `if` conditions and ternary expressions. The condition is transformed by introducing an integer set and by folding the operands into it similarly to the affine maps, with in addition `and` operations separating set constraints and `not` operations inverting them (`affine.if` only accepts ≥ 0 and $= 0$ constraints). Polygeist processes nested conditionals with C-style short-circuit semantics, in which the subsequent conditions are checked within the body of the preceding conditionals, by hoisting conditions outside the outermost conditional when legal and replacing them with a boolean operation or a `select`. This is always legal within `#pragma scop`.

Conditionals emitted for ternary expressions often involve memory loads in their regions, which prevent hoisting due to side effects. We reuse our mem2reg pass to replace those to equivalent earlier loads when possible to enable hoisting. Empirically, this is sufficient to process all ternary expressions in the Polybench/C suite [309]. Otherwise, ternary expressions would need to be packed into a single statement by the downstream polyhedral pass.

6.3.3 Connecting MLIR to Polyhedral Tools

Regions of the input program expressed using MLIR Affine dialect are amenable to the polyhedral model. Existing tools, however, cannot directly consume MLIR. We chose to implement a bi-directional conversion to and from OpenScop [30], an exchange format readily consumable by numerous polyhedral tools, including Pluto [52], and further convertible to `isl` [400] representation. This allows Polygeist to seamlessly connect with tools created in polyhedral compilation research without having to amend those tools to support MLIR.

Most polyhedral tools are designed to operate on C or Fortran inputs build around *statements*, which do not have a direct equivalent in MLIR. Therefore, we design a mecha-

```
void setArray(int N, double val, double* array) {
  ...
}
int main(int argc, char** argv) {
  ...
  cmp = strcmp(str1, str2)
  ...
  double array[10];
  set_array(10, array)
}
```

⇓

```
func @setArray(%N: i32, %val: f64, %array: memref<?xf64>) {
  %0 = index_cast %N : i32 to index
  affine.for %i = 0 to %0 {
    affine.store %val, %array[%i] : memref<?xf64>
  }
  return
}

func @main(%argc: i32,
           %argv: !llvm.ptr<ptr<i8>>) -> i32 {
  ...
  %cmp = llvm.call @strcmp(%str1, %str2) :
           (!llvm.ptr<i8>, !llvm.ptr<i8>) -> !llvm.i32
  ...
  %array = memref.alloca() : memref<10xf64>
  %arraycst = memref.cast %array : memref<10xf64> to
    memref<?xf64>
  call @setArray(%N, %val, %arraycst) :
           (i32, f64, memref<?xf64>) -> ()
}
```

Figure 6-5: Example demonstrating Polygeist ABI. For functions expected to be compiled with Polygeist such as `setArray`, pointer arguments are replaced with `memref`'s. For functions that require external calling conventions (such as `main`/`strcmp`), we fall back to using `llvm.ptr` and generating conversion code where appropriate.

nism to create statement-like structure from chains of MLIR operations. We further demonstrate that this gives Polygeist an ability to favorably affect the behavior of the polyhedral scheduler by controlling statement granularity (Section 6.3.4).

Simple Statement Formation

Observing that C statements amenable to the polyhedral model are (mostly) variable assignments, we can derive a mechanism to identify statements from chains of MLIR operations. A `store` into memory is the last operation of the statement. The backward slice of this operation, i.e., the operations transitively computing its operands, belong to the statement. The slice extension stops at operations producing a value categorized as affine dimension or symbol, directly usable in affine expressions. Such values are loop induction variables or loop-invariant constants.

Some operations may end up in multiple statements if the value is used more than once. However, we need the mapping between operations and statements to be bidirectional in order to emit MLIR after the scheduler has restructured the program without considering SSA value visibility rules. If an operation with multiple uses is side effect free, Polygeist simply duplicates it. For operations whose duplication is illegal, Polygeist stores their results in stack-allocated `memref`'s and replaces all further uses with memory loads. Figure 6-6 illustrates the transformation for value `%0` used in operation `%20`. This creates a new statement.

Region-Spanning Dependencies

In some cases, a statement may consist of MLIR operations across different (nested) loops, e.g., a load from memory into an SSA register happens in an outer loop while it is used in inner loops. The location of such a statement in the loop hierarchy is unclear. More importantly, it cannot be communicated to the polyhedral scheduler. Polygeist resolves this by storing the value in a stack-allocated `memref` in the defining region and loading it back in the user regions. Figure 6-6 illustrates this transformation for value `%0` used in operation `%10`. Similarly to the basic case, this creates a new statement in the outer loop that can be scheduled independently.

```
affine.for %i = ... {
  %0 = affine.load %A[%i]
  affine.store %other, %A[%i] // motion-barrier
  affine.for %j = ... {
    %1 = affine.load %B[%j]
    %10 = mulf %0, %1 : f64   // use-1
    store %10, %res[%i, %j]
  %20 = addf %0, %0 : f64     // use-2
}
```

⇓

```
%tmp = memref.alloca() : memref<1xf64>
affine.for %i = ... {
  %0 = affine.load %A[%i]
  affine.store %0, %tmp[0]    // store to scratchpad
  affine.store %other, %A[%i] // motion-barrier
  affine.for %j = ... {
    %1 = affine.load %B[%j]
    %2 = affine.load %tmp[0]  // load back for use-1
    %10 = mulf %2, %1 : f64   // use-1 (%2 instead of %0)
    affine.store %10, %res[%i, %j]
  }
  %19 = affine.load %tmp[0]   // load back for use-2
  %20 = addf %19, %19 : f64     // use-2 (%19 instead of %0)
  // ...
}
```

Figure 6-6: Polygeist breaks region-spanning use-def chains and handles multi-use values by introducing scratchpad storage when operation duplication is illegal. In absence of `motion-barrier` statement, the `%0` load would be duplicated and sunk. Pseudo-MLIR with types and braces omitted for brevity.

This approach can be seen as a reg2mem conversion, the inverse of mem2reg performed in the frontend. It only applies to a subset of values, and may be undone after polyhedral scheduling has completed. Furthermore, to decrease the number of dependencies and memory footprint, Polygeist performs a simple value analysis and avoids creating stack-allocated buffers if the same value is already available in another memory location and can be read from there.

SCoP Formation

To define a SCoP, we outline individual statements into functions so that they can be represented as opaque calls with known memory footprints, similarly to Pencil [22]. This process also makes the inter-statement SSA dependencies clear. These dependencies exist

```
func @S1(%A: memref<?xf64>, %tmp: memref<1xf64>, %i: index) {
  %0 = affine.load %A[%i]
  affine.store %0, %tmp[0]   // store to scratchpad
}

func @S2(%A: memref<?xf64>, %other: f64, %i: index) {
  affine.store %other, %A[%i]
}

func @S3(%B: memref<?x?xf64>, %tmp: memref<1xf64>,
         %res: memref<?x?xf64>, %i: index, %j: index) {
  %1 = affine.load %B[%j]
  %2 = affine.load %tmp[0]   // load back for use-1
  %10 = mulf %2, %1 : f64    // use-1
  affine.store %10, %res[%i, %j]
}

func @S4(%tmp: memref<1xf64>, ...) {
  %19 = affine.load %tmp[0]  // load back for use-2
  %20 = addf %19, %19 : f64    // use-2
  // ...
}

%tmp = memref.alloca() : memref<1xf64>
affine.for %i = ... {
  call @S1(%A, %tmp, %i)
  call @S2(%A, %other, %i)
  affine.for %j = ... {
    call @S3(%B, %tmp, %res, %i, %j)
  }
  call @S4(%tmp)
}
```

Figure 6-7: Outlining makes polyhedral “statements” visible in code from Fig. 6-6.

between calls that *use* the same SSA value since all use-def chains (except for induction variables that are processed separately) have been encapsulated into statements. We lift all local stack allocations and place them at the entry block of the surrounding function in order to keep them visible after loop restructuring. Figure 6-7 demonstrates the resulting IR.

The remaining components of the polyhedral representation are derived as follows: the domain of the statement is defined to be the iteration space of its enclosing loops, constrained by their respective lower and upper bounds, and intersected with any "if" conditions. This process leverages the fact that MLIR expresses bounds and conditions directly as affine constructs. The access relations for each statement are obtained as unions of affine maps of the `affine.load` (read) and `affine.store` (must-write) operations, with RHS of the relation annotated by an "array" that corresponds to the SSA value of the accessed `memref`. Initial schedules are assigned using the $(2d + 1)$ formalism, with odd dimensions representing the lexical order of loops in the input program and even dimensions being equal to loop induction variables. Affine constructs in OpenScop are represented as lists of linear equality ($= 0$) or inequality (≥ 0) coefficients, which matches exactly the internal representation in MLIR, making the conversion straightforward.

Code Generation Back to MLIR

The Pluto scheduler produces new schedules in OpenScop as a result. Generating loop structure back from affine schedules is a solved, albeit daunting, problem [29, 151]. Polygeist relies on CLooG [29] to generate an initial loop-level AST, which it then converts to Affine dialect loops and conditionals. There is no need to simplify affine expressions at code generation since MLIR accepts them directly and can simplify them at a later stage. Statements are introduced as function calls with rewritten operands and then inlined.

6.3.4 Controlling Statement Granularity

Recall that Polygeist reconstructs "statements" from sequences of primitive operations (Section 6.3.3). We initially designed an approach that recovers the statement structure

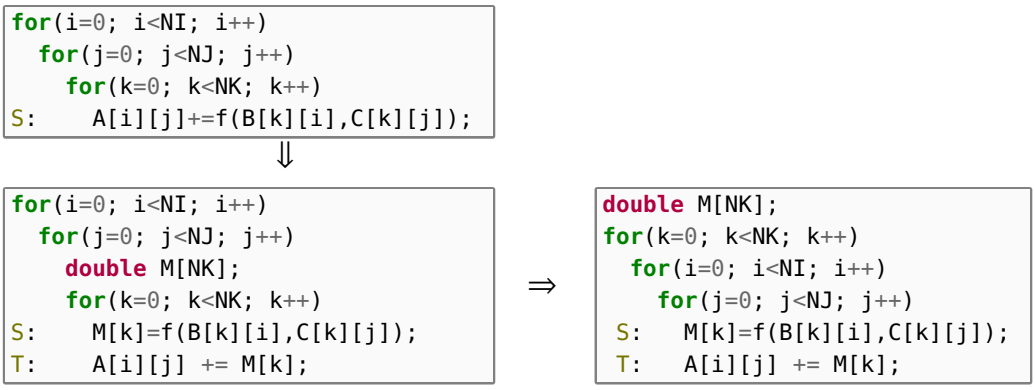


Figure 6-8: Splitting a nested reduction statement (top) into a fully parallel compute statement and a trivial reduction statement (bottom left) makes Pluto generate different schedules (bottom right). Further scratchpad array expansion may enable loop fission and give scheduler even more liberty.

similar to that in the C input, but this is not a requirement. Instead, statements can be formed from any subsets of MLIR operations as long as they can be organized into loops and sorted topologically (i.e., there are no use-def loops between statements). To expose the dependencies between such statements to the affine scheduler, we reuse the idea of going through scratchpad memory: each statement writes the values required by other statements to dedicated memory locations, and the following statements read from those. The scratchpads are subject to partial array expansion [110] to minimize their effect on the affine scheduler as single-element scratchpad arrays create artificial scalar dependencies. This change in *statement granularity* gives the affine scheduler unprecedented flexibility allowing it to chose different schedules for different *parts* of the same C statement.

Consider, for example, the statement `S` in Figure 6-8(top) surrounded by three loops iterating over `i`, `j` and `k`. Such contraction patterns are common in computational programs (this particular example can be found in the `correlation` benchmark with `B≡C`, see Section 6.5.5). The loop order that best exploits the locality is (`k`, `i`, `j`), which results in temporal locality for reads from `B` (the value is reused in all iterations of the now-innermost `j` loop) and in spatial locality for reads from `C` (consecutive values are read by consecutive iterations, increasing the likelihood of L1 cache hits). Yet, Pluto never proposes such an order because of a reduction dependency along the `k` dimension due to repeated read/write access to `A[i][j]` as Pluto tends to pick loops with fewer dependencies as outermost.

While the dependency itself is inevitable, it can be moved into a separate statement `T` in Figure 6-8(bottom left). This approach provides scheduler with more freedom of choice for the first statement at a lesser memory cost than expanding the entire `A` array. It also factors out the reduction into a "canonical" statement that is easier to process for the downstream passes, e.g., vectorization.

Implementing this transformation at the C level would require manipulating C AST and reasoning about C (or even C++) semantics. This is typically out of reach for source-to-source polyhedral optimizers such as Pluto that treat statements as black boxes. While it is possible to implement this transformation at the LLVM IR level, e.g., in Polly, where statements are also reconstructed and injection of temporary allocations is easy, the heuristic driving the transformation is based on the loop structure and multi-dimensional access patterns which are difficult to recover at such a low level [150].

The space of potential splittings is huge—each MLIR operation can potentially become a statement. Therefore, we devise a heuristic to address the contraction cases similar to Figure 6-8. Reduction statement splitting applies to statements:

- surrounded by at least 3 loops;
- with LHS≠RHS, and using all loops but the innermost;
- with two or more different access patterns on the RHS.

This covers statements that could have locality improved by a different loop order and with low risk of undesired fission. This heuristic merely serves as an illustration of the kind of new transformations Polygeist can enable.

6.3.5 Post-Transformations and Backend

Polygeist allows one to operate on both quasi-syntactic and SSA level, enabling analyses and optimizations that are extremely difficult, if not impossible, to perform at either level in isolation. In addition to statement splitting, we propose two techniques that demonstrate the potential of Polygeist.

Transforming Loops with Carried Values (Reductions)

Polygeist leverages MLIR's first-class support for loop-carried values to detect, express and transform reduction-like loops. This support does not require source code annotations, unlike source-level tools [321] that use annotations to enable detection, nor complex modifications for parallel code emission, unlike Polly [98], which suffers from LLVM missing first-class parallel constructs. We do not modify the polyhedral scheduler either, relying on post-processing for reduction parallelization, including outermost parallel reduction loops.

The overall approach follows the definition proposed in [202] with adaptations to MLIR's region-based IR, and is illustrated in Figure 6-9. Polygeist identifies memory locations modified on each iteration, i.e. `load`/`store` pairs with loop-invariant subscripts and no interleaving aliasing `store`s, by scanning the single-block body of the loop. These are transformed into *loop-carried values* or secondary induction variables, with the `load`/`store` pair lifted out of the loop and repurposed for reading the initial and storing the final value. Loop-carried values may be updated by a chain of side effect-free operations in the loop body. If this chain is known to be associative and commutative, the loop is a *reduction*. Loop-carried values are detected even in absence of reduction-compatible operations. Loops with such values contribute to mem2reg, decreasing memory footprint, but are not subject to parallelization.

Late Parallelization

Rather than relying on the dependence distance information obtained by the affine scheduler, Polygeist performs a separate polyhedral analysis to detect loop parallelism in the generated code. The analysis itself is a classical polyhedral dependence analysis [111, 106] implemented on top of MLIR region structure. Performing it after SSA-based optimizations, in particular mem2reg and reduction detection, allows parallelizing more loops. In particular, reduction loops and loops with variables whose value is only relevant within a single iteration similar to live-range reordering [403] but without expensive additional polyhedral analyses (live-range of an SSA value defined in a loop never extends beyond the loop).

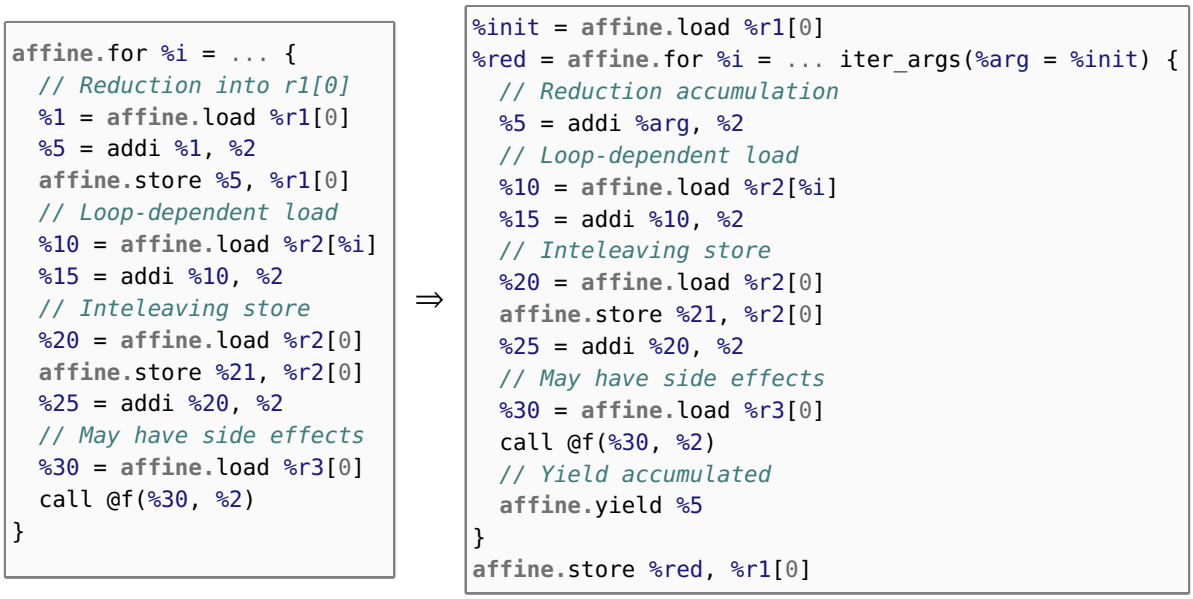


Figure 6-9: Polygeist detects memory locations accessed in all loop iterations, e.g. reduction accumulators such as `%r1[0]` and transforms them to loop-carried values (secondary induction variables), except when computed with side-effects, interleaved stores or by non-associative/commutative operations.

6.4 Evaluation

Our evaluation has two goals. 1) We want to demonstrate that the code produced by Polygeist without additional optimization does not have any inexplicable performance differences than a state-of-the-art compiler like Clang. 2) We explore how Polygeist's internal representation can support a mix of affine and SSA-based transformation in the same compilation flow, and evaluate the potential benefits compared to existing source and compiler-based polyhedral tools.

6.4.1 Experimental Setup

We ran our experiments on an AWS `c5.metal` instance with hyper-threading and Turbo Boost disabled. The system is Ubuntu 20.04 running on an Intel Xeon Platinum 8275CL CPU at 3.0 GHz with 1.5, 48, 71.5 MB L1, L2, L3 cache, respectively, and 256 GB RAM. We ran all 30 benchmarks from PolyBench [309], using the "EXTRALARGE" dataset. Pluto is unable to extract SCoP from the `adi` benchmark. We ran a total of 5 trials for each benchmark, taking the execution time reported by PolyBench; the median result is taken

unless stated otherwise. Every measurement or result reported in the following sections refers to double-precision data. All experiments were run on cores 1-8, which ensured that all threads were on the same socket and did not potentially conflict with processes scheduled on core 0.

In all cases, we use two-stage compilation: (i) using `clang` at `-O3` excluding unrolling and vectorization; or Polygeist to emit LLVM IR from C; (ii) using `clang` at `-O3` to emit the final binary. As several optimizations are not idempotent, a second round of optimization can potentially significantly boost (and rarely, hinder) performance. This is why we chose to only perform vectorization and unrolling at the last optimization stage. Since Polygeist applies some optimizations at the MLIR level (e.g., mem2reg), we compare against the two-stage compilation pipeline as a more fair baseline (CLANG). We also evaluate a single-stage compilation to assess the effect of the two-stage flow (CLANGSING).

6.4.2 Baseline Performance

Polygeist must generate code with runtime *as close as possible* to that of existing compilation flows to establish a solid baseline. In other words, Polygeist should *not introduce overhead nor speedup* unless explicitly instructed otherwise, to allow for measuring the effects of additional optimizations. We evaluate this by comparing the runtime of programs produced by Polygeist with those produced by Clang at the same commit (Apr 2021)[1]. Figure 6-10 summarizes the results with the following flows:

- CLANG: A compilation of the program using Clang, when running two stages of optimization;
- CLANGSING: A compilation of the program using Clang, when running one stage of optimization;
- MLIR-CLANG: A compilation flow using the Polygeist frontend and preprocessing optimizations within MLIR, but not running polyhedral scheduling nor postprocessing.

[1] LLVM commit `20d5c42e0ef5d252b434bcb610b04f1cb79fe771`.

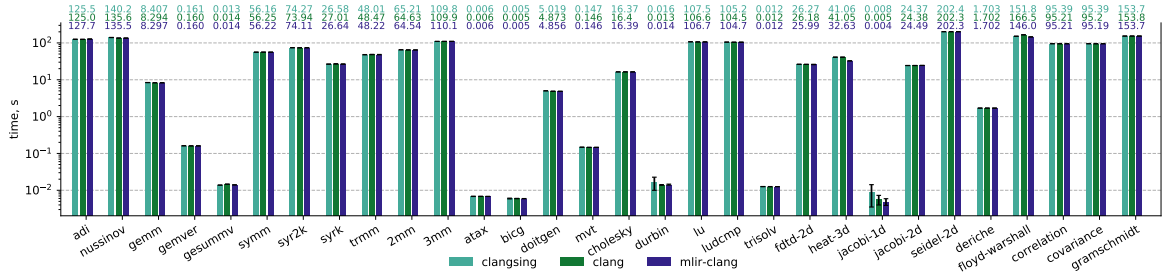


Figure 6-10: Mean and 95% confidence intervals (log scale) of program run time across 5 runs of Polybench in CLANG, CLANGSING and MLIR-CLANG configurations, lower is better. The run times of code produced by Polygeist without optimization is comparable to that of Clang. No significant variation is observed between single and double optimization. Short-running `jacobi-1d` shows high intra-group variation.

6.4.3 Compilation Flows

We compare Polygeist with a source-level and an IR-level optimizer (Pluto and Polly) in the following configurations:

- PLUTO: Pluto compiler auto-transformation [52] using `polycc`[2] with `-noparallel` and `-tile` flags;
- PLUTOPAR: Same as above but with `-parallel` flag;
- POLLY: Polly [149] LLVM passes with affine scheduling and tiling, and no pattern-based optimizations [127];
- POLLYPAR: Same as above with auto-parallelization;
- POLYGEIST: Our flow with Pluto and extra transforms;
- POLYGEISTPAR: Same as above but with `-parallel` Pluto schedule, Polygeist parallelization and reductions.

Running between source and LLVM IR levels, we expect Polygeist to benefit from both worlds, thus getting code that is on par or better than competitors. When using Pluto, both standalone and within Polygeist, we disable the emission of vectorization hints and loop unrolling to make sure both transformations are fully controlled by the LLVM optimizer, which also runs in Polly flows. We run Polly in the latest stage of Clang compilation, using

[2]Pluto commit `dae26e77b94b2624a540c08ec7128f20cd7b7985`

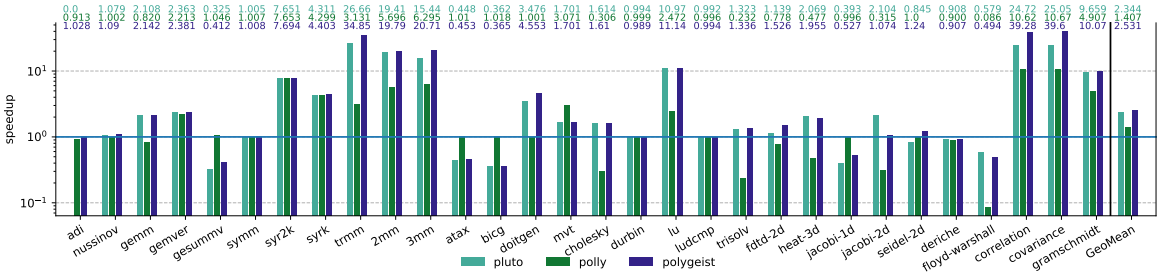


Figure 6-11: Median speedup over Clang for sequential configurations (log scale), higher is better. Polygeist outperforms (2.53× geomean speedup) both Pluto (2.34×) and Polly (1.41×) on average. Pluto can't process `adi`, which is therefore excluded from summary statistics.

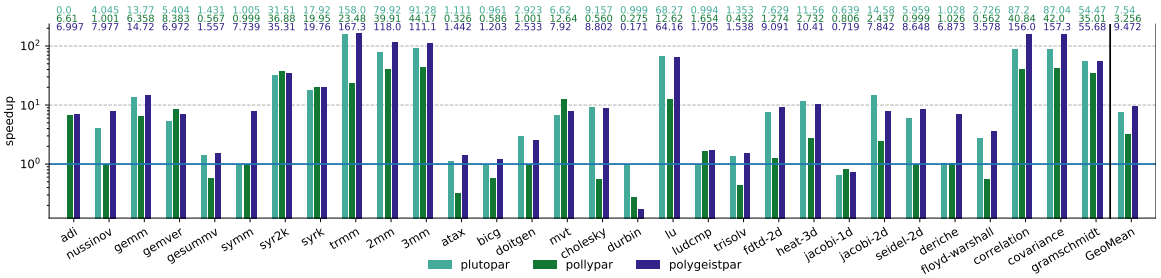


Figure 6-12: Median speedup over Clang for parallel configurations (log scale), higher is better. Polygeist outperforms (9.47× geomean speedup) both Pluto (7.54×) and Polly (3.26×) on average. Pluto can't process `adi`, which is therefore excluded from summary statistics.

`-mllvm -polly` and additional flags to enable affine scheduling, tiling and parallelization as required. Polly is taken at the same LLVM commit as Clang. We disable pattern-based optimizations [127] that are not available elsewhere. Figures 6-11 and 6-12 summarize the results for sequential and parallel flows, respectively.

6.5 Performance Analysis

6.5.1 Benchmarking

The transformation of reduction loops, in particular parallelization, may result in a different order of partial result accumulation. This is not allowed under IEEE 754 semantics, but is supported by compilers with `-ffast-math` option.

We found that Polybench allocation function hinders Clang/LLVM alias analysis, neg-

atively affecting performance in, e.g., `adi`. Therefore, we modified all benchmarks to use `malloc` that is known to produce non-aliasing pointers.

6.5.2 Baseline Comparison

We did not observe a significant difference between the runtimes of CLANG and CLANGSING configurations, with a geometric mean of 0.43% symmetric difference[3] across benchmarks. Therefore, we only consider CLANG as baseline throughout the remainder of this paper. We did not observe a significant difference between the runtimes of CLANG and MLIR-CLANG configurations either, with a geometric mean of 0.24% symmetric difference.

We found a variation in runtimes of short-running benchmarks, in particular `jacobi-1d`. This can be attributed to the interaction with the data initialization and benchmarking code, and with other OS processes. Excluding the benchmarks running in under 0.05s (`jacobi-1d`, `gesummv`, `atax`, `bicg`) from the analysis, we obtain 0.32% and 0.17% geomean symmetric differences respectively for the two comparisons above. These results suggest that our flow has no unexplained (dis)advantages over the baseline.

6.5.3 Performance Differences in Sequential Code

Overall, Polygeist leads to larger speedups, with 2.53× geometric mean, than both Pluto (2.34×) and Polly (1.41×), although improvements are not systematic. Some difference between Polygeist and Polly is due to the employed polyhedral schedulers, e.g., in `lu` and `mvt`. Polygeist produces code faster tha both Pluto and Polly in `2mm`, `3mm` and others thanks to statement splitting, see Section 6.5.5.

Given identical statements and schedules, codegen-level optimization accounts for other performance difference. `seidel-2d` is the clearest example: Pluto executes $2.7 \cdot 10^{11}$ more integer instructions than Polygeist. Assuming these to be index/address computations, a mix of `add` (throughput 1/2 or 1/4) and `imul/shl` (thoughput 1), we can expect a $\approx$ 59s difference at 3GHz, consistent with experimental observations. Polygeist optimizes away a part of those in its post-optimization phase and emits homogeneous address computa-

[3] Symmetric difference is computed as $2 \cdot |a - b|/(a + b)$.

tion from `memref` with proper machine size type, enabling more aggressive bound analysis and simplification in the downstream compiler. Conversely, `jacobi-2d` has poorer performance because Polygeist gives up on simplifying CLooG code, with up to 75 statement copies in 40 branches, for compiler performance reasons, as opposed to Clang that takes up to 5s to process it but results in better vectorization. Further work is necessary to address this issue by emitting vector instructions directly from Polygeist.

6.5.4 Performance Differences In Parallel Code

Similarly to sequential code, some performance differences are due to different schedulers. For example, in `cholesky` and `lu`, both Pluto and Polygeist outperform Polly, and the remaining gap can be attributed to codegen-level differences. Conversely, in `gemver` and `mvt` Polly has a benefit over both Pluto and Polygeist. On `ludcmp` and `syr(2)k`, SSA-level optimizations let Polygeist produce code which is faster than Pluto and at least as fast as Polly. These results demonstrate that Polygeist indeed leverages the benefits of both the affine and SSA-based optimizations.

Polygeist is the only flow that obtains speedup on `deriche` (6.9×) and `symm` (7.7×). Examining the output code, we observe that only Polygeist manages to parallelize these two benchmarks. Considering the input code in Figure 6-13, one can observe that the `i` loop reuses the `ym1` variable, which is interpreted as parallelism-preventing loop-carried dependency by polyhedral schedulers. Polygeist performs its own parallelism analysis after promoting `ym1` to an SSA register (carried by the `j` loop) whose use-def range does not prevent parallelization.

Similarly, the Polygeist parallelizer identifies two benchmarks with parallel reduction loops that are not contained in other parallel loops: `gramschmidt` and `durbin`. `gramschmidt` benefits from a 56× speedup with Polygeist, compared to 34× with Polly and 54× with Pluto. `durbin` sees a 6× slowdown since the new parallel loop has relatively few iterations and is nested inside a sequential loop, leading to synchronization costs that outweigh the parallelism benefit. Section 6.5.6 explores the `durbin` benchmark in more detail. Polybench is a collection of codes (mostly) known to be parallel and, as such, has little need for

```
for (i=0; i<_PB_W; i++) {
  ym1 = SCALAR_VAL(0.0);
  // ...
  for (j=0; j<_PB_H; j++) {
    ym1 = y1[i][j];
    /*...*/
  }
}
```

```
%z = constant 0.0 : f64
affine.parallel %i = ... {
  affine.for %j = ... iter_args(%ym1=%z) -> f64 {
    %0=affine.load %y1[%i,%j]
    // ...
    affine.yield %0
  }
}
```

Figure 6-13: Excerpt from the `deriche` benchmark. The outer loop reuses `ym1` which makes it appear non-parallel to affine schedulers (left). Polygeist detects parallelism thanks to its mem2reg optimization, reduction-like loop-carried `%ym1` value detection and late parallelization (right).

reduction parallelization on CPU where one degree of parallelism is sufficient. When targeting inherently target architectures as GPUs, however, exploiting reduction parallelism could be vital for achieving peak performance [228, 321].

6.5.5 Case Study: Statement Splitting

We identified 5 benchmarks where the statement splitting heuristic applied: `2mm`, `3mm`, `correlation`, `covariance` and `trmm`. To assess the effect of the transformation, we executed these benchmarks with statement splitting disabled, suffixed with `-nosplit` in Figure 6-14. In sequential versions, `2mm` is 4.1% slower (3.13s vs 3.26s), but the other benchmarks see speedups of 25%, 50%, 51% and 27%, respectively. For parallel versions, the speedups are of 36%, 20%, 44%, 40% and −9% respectively.

Examination of polyhedral scheduler outputs demonstrates that it indeed produced the desired schedules. For example, in the `correlation` benchmark which had the statement `A[i][j] += B[k][i]*B[k][j]` Polygeist was able to find the (k, i, j) loop order after splitting. Using hardware performance counters on sequential code we confirm that the overall cache miss ratio has indeed decreased by 75%, 50%, 20%, 27%, and −26%, respectively. However, the memory traffic estimated by the number of bus cycles has *increased* by 9% for `2mm`, and decreased by 18%, 32%, 32%, and 21% for the other benchmarks. This metric strongly correlates with the observed performance difference in the same run (r=0.99, $p = 3 \cdot 10^{-11}$). This behavior is likely due to the scheduler producing a different fusion structure, e.g., not fusing outermost loops in `2mm`, which also affects locality. Sim-

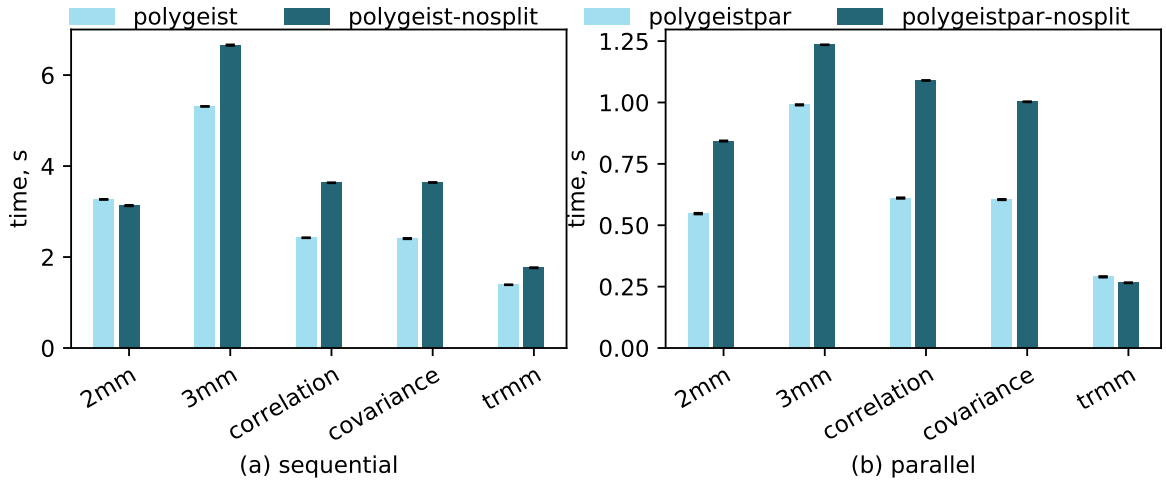


Figure 6-14: Mean and 95% confidence intervals of run time across 5 runs of Polybench where statement splitting is applicable (Section 6.3.4), lower is better. It results in faster run time (geomean 1.28× sequential, 1.39× parallel speedup) except for sequential `2mm` (−4%) and parallel `trmm` (−9%).

ilar results can be observed for parallel code. Further research is necessary to exploit the statement splitting opportunities, created by Polygeist, and interplay with fusion.

6.5.6 Case Study: Reduction Parallelization in `durbin`

In this benchmark, Polygeist uses its reduction optimization to create a parallel loop that other tools cannot. For the relatively small input run by default, $N = 4000$ iterations inside another sequential loop with N iterations, the overall performance decreases. We hypothesize that the cost of creating parallel threads and synchronizing them outweighs the benefit of the additional parallelism and test our hypothesis by increasing N. Considering the results in Figure 6-15, one observes that Polygeist starts yielding speedups (> 1) for $N \geq 16000$ whereas Polly only does so at $N \geq 224000$, and to a much lesser extent: 6.62× vs 1.01×. Without reduction parallelization, Polygeist follows the same trajectory as Polly. Pluto fails to parallelize any innermost loop and shows no speedup. This evidences in favor of our hypothesis and highlights the importance of being able to parallelize reductions.

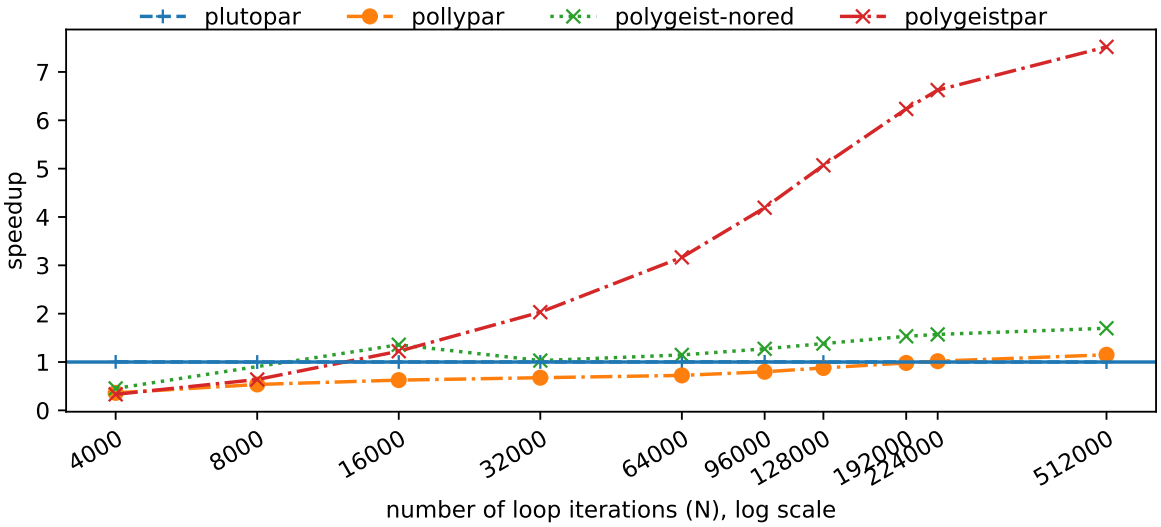


Figure 6-15: Reduction parallelization allows PolygeistPar to produce larger speedups and at smaller sizes than PollyPar and PolygeistPar without reduction support. PlutoPar fails to parallelize leading to no speedup.

6.6 Related Work

MLIR Frontends

Since the adoption of MLIR under the LLVM umbrella, several frontends have been created for generating MLIR from domain-specific languages. Teckyl [99] connects the productivity-oriented Tensor Comprehensions [397] notation to MLIR's Linalg dialect. Flang—the LLVM's Fortran frontend—models Fortran-specific constructs using the FIR dialect [351]. COMET, a domain-specific compiler for chemistry, introduces an MLIR-targeting domain-specific frontend from a tensor-based language [280]. NPComp aims at providing the necessary infrastructure to compile numerical Python and PyTorch programs taking advantage of the MLIR infrastructure [289]. PET-to-MLIR converts a subset of polyhedral C code to MLIR's Affine dialect by parsing `pet`'s internal representation. In addition to currently not handling specific constructs (`if`s, symbolic bounds, and external function calls), parsing `pet`'s representation limits the frontend's usability as it cannot interface with non-polyhedral code such as initialization, verification, or printing routines [219]. In contrast, Polygeist generates MLIR from non-polyhedral code (though not necessarily in the Affine dialect). CIRCT is a new project under the LLVM umbrella that aims to apply MLIR development methodology to the electronic design automation industry [78]. Stripe uses

MLIR Affine dialect as a substrate for loop transformations in machine learning models, including tiling and vectorization, and accepts a custom DSL as input [426].

Combining "Classical" and Polyhedral Flows

Few papers have focused on combining "classical", mostly AST-level, and polyhedral transformations. PolyAST pioneered the approach by combining an affine scheduler with AST-level heuristics for fusion and tiling [354], although similar results were demonstrated with only polyhedral transformations [430]. An analogous approach was experimented in CUDA-CHiLL [427]. Arguably, many automated polyhedral flows perform loop fusion and/or tiling as a separate step that can be assimilated to classical transformations. Pluto [52] uses several "syntactic" postprocessing passes to exploit spatial locality and parallelism in stencils [430]. Several tools have been proposed to drive polyhedral loop transformations with scripts using classical loop transformations such as fusion and permutation as operations, including URUK [136], CHiLL [71] and Clay [24]. Polygeist differs from all of these because it preserves the results of such transformations in its IR *along with* polyhedral constructs and enables interaction between different levels of abstraction.

Additional (Post-)Polyhedral Transformations

Support for handling reduction loops was proposed in Polly [98], but the code generation is not implemented. At the syntactic level, reduction support was added to PET via manual annotation with Pencil directives [321]. R-Stream reportedly uses a variant of statement splitting to affect scheduler's behavior and optimize memory consumption [260]. PolySIMD uses variable renaming around PPCG polyhedral flow to improve vectorization [66]. Polygeist automates these leveraging both SSA and polyhedral information.

Integration of Polyhedral Optimizers into Compilers

Polyhedral optimization passes are available in production (GCC [307], LLVM [149], IBM XL [51]) and research (R-Stream [260], ROSE [316]) compilers. In most cases, the polyhedral abstraction must be extracted from a lower-level representation before being transformed and lowered in a dedicated code generation step [29, 151]. This extraction process is not guaranteed and may fail to recover high-level information available at the source level [150]. Furthermore, common compiler optimizations such as LICM are known to interfere with it [223]. Polygeist maintains a sufficient amount of high-level information,

in particular loop and n-D array structure, to circumvent these problems by design.

Source-to-source polyhedral compilers such as Pluto [52] and PPCG [406] operate on a C or C++ level. They lack interaction with other compiler optimizations and a global vision of the code, which prevents, e.g., constant propagation and inlining that could improve the results of polyhedral optimization. Being positioned between the AST and LLVM IR levels, Polygeist enables the interaction between higher- and lower-level abstractions that is otherwise reduced to compiler pragmas, i.e. mere optimization hints. Furthermore, Polygeist can rely on MLIR's progressive raising [69] to target abstractions higher level than C code with less effort than polyhedral frameworks [70].

6.7 Discussion

6.7.1 Limitations

Frontend While Polygeist could technically accept any valid C or C++ thanks to building off Clang, it has the following limitations. Only `structs` with values of the same type or are used within specific functions (such as `FILE` within `fprintf`) are supported due to the lack of a struct-type in high-level MLIR dialects. All functions that allocate memory must be compiled with Polygeist and not a C++ compiler to ensure that a `memref` is emitted rather than a pointer.

Optimizer The limitations of the optimizer are inherited from those of the tools involved. In particular, the MLIR affine value categorization results in all-or-nothing modeling, degrading any loop to non-affine if it contains even one non-affine access or a negative step. Running Polygeist's backend on code not generated by Polygeist's frontend, which reverses loops with negative steps, is limited to loops with positive indices. Finally, MLIR does not yet provide extensive support for non-convex sets (typically expressed as unions). Work is ongoing within MLIR to address such issues.

Experiments While our experiments clearly demonstrate the benefits of the techniques implemented in Polygeist— statement splitting and late (reduction) parallelization — non-negligible effects are due to scheduler difference: Pluto in Polygeist and `isl` in Polly. The

version of Polly using Pluto[4] is not compatible with modern LLVM necessary to leverage MLIR. Connecting `isl` scheduler to Polygeist may have yielded results closer to Polly, but still not comparable more directly because of the interplay between SCoP detection, statement formation and affine scheduling.

6.7.2 Opportunities and Future Work

Connecting MLIR to existing polyhedral flows opens numerous avenues for compiler optimization research, connecting polyhedral and conventional SSA-based compiler transformations. This gives polyhedral schedulers access to important analyses such as aliasing and useful information such as precise data layout and target machine description. Arguably, this information is already leveraged by Polly, but the representational mismatch between LLVM IR and affine loops makes it difficult to exploit them efficiently. MLIR exposes similar information at a sufficiently high level to make it usable in affine transformations.

By mixing abstractions in a single module, MLIR provides finer-grain control over the entire transformation process. An extension of Polygeist can, e.g., ensure loop vectorization by directly emitting vector instructions instead of relying on pragmas, which are often merely a recommendation for the compiler. The flow can also control lower-level mechanisms like prefetching or emit specialized hardware instructions. Conversely, polyhedral analyses can guarantee downstream passes that, e.g., address computation never produces out-of-bounds accesses and other information.

Future work is necessary on controlling statement granularity made possible by Polygeist. Beyond affecting affine schedules, this technique enables easy rematerialization and local transposition buffers, crucial on GPUs [376], as well as software pipelining; all without having to produce C source which is known to be complex [420]. On the other hand, this may have an effect on the compilation time as the number of statements is an important factor in the complexity bound of the dependence analysis and scheduling algorithms.

[4] http://pluto-compiler.sourceforge.net/#libpluto

6.7.3 Alternatives

Instead of allowing polyhedral tools to parse and generate MLIR, one could emit C (or C++) code from MLIR[5] and use C-based polyhedral tools on the C source, but this approach decreases the expressiveness of the flow. Some MLIR constructs, such as parallel reduction loops, can be directly expressed in the polyhedral model, whereas they would require a non-trivial and non-guaranteed raising step in C. Some other constructs, such as prevectorized affine memory operations, cannot be expressed in C at all. Polygeist enables transparent handling of such constructs in MLIR-to-MLIR flows, but we leave the details of such handling for future work.

The Polygeist flow can be similarly connected to other polyhedral formats, in particular `isl`. We choose OpenScop for this work because it is supported by a wider variety of tools. `isl` uses schedule trees [404] to represent the initial and transformed program schedule. Schedule trees are sufficiently close to the nested-operation IR model making the conversion straightforward: "for" loops correspond to band nodes (one loop per band dimension), "if" conditionals correspond to filter nodes, function-level constants can be included into the context node. The tree structure remains the same as that of MLIR regions. The inverse conversion can be obtained using `isl`'s AST generation facility [151].

6.8 Conclusion

We present Polygeist, a compilation workflow for importing existing C or C++ code into MLIR and allows polyhedral tools, such as Pluto, to optimize MLIR programs. This enables MLIR to benefit from decades of research in polyhedral compilation. We demonstrate that the code generated by Polygeist has comparable performance with Clang, enabling unbiased comparisons between transformations built for MLIR and existing polyhedral frameworks. Finally, we demonstrate the optimization opportunities enabled by Polygeist considering two complementary transformations: statement splitting and reduction parallelization. In both cases, Polygeist achieves better performance than state-of-the-art polyhedral compiler and source-to-source optimizer.

[5] https://github.com/marbre/mlir-emitc

Chapter 7

Fast Automatic Differentiation of GPU Kernels via Compiler Optimization

7.1 Introduction

Automatic differentiation (AD) provides an accurate way of computing derivatives of mathematical functions that are implemented in computer programs. *Gradients* (or *adjoints*), a special case of derivatives for functions with one output and many inputs, have applications in optimization [104], uncertainty quantification [130], inverse design, stability analysis,and machine learning [254]. Reverse-mode AD has been the tool of choice to compute these gradients for large applications with many input parameters.

As the research community has been continuously pushing the boundary of the size of problems they want to solve, large-scale applications have had to leverage the latest in high-performance computing including distributed computation, parallelism, and accelerators. For many machine learning and scientific computing applications, this means relying on kernels that are highly optimized for graphics processing units (GPUs).

While considerable effort has been expended to compute gradients of MPI and OpenMP programs (see Sec 7.2 for related work), no AD tool has been presented to date that can compute gradients of CUDA or ROCm (AMD) kernels. The cause rests with both the GPU's parallelism and its complex performance characteristics. The biggest issue for both

```
void init(double* ar, int N, double val) {
  parallel_for(int i=0; i<N ; i++)
    // Concurrent reads of val
    ar[i] = val;
}

double ∇init(double* ar, double* d_ar, int N, double val) {
  double d_val = 0.0;
  parallel_for(int i=0; i<N ; i++)
    ar[i] = val;
  parallel_for(int i=0; i<N ; i++) {
    // Concurrent writes to d_val
    d_val += d_ar[i];                    ↯ race ↯
    d_ar[i] = 0.0;
  }
  return d_val;
}
```

Figure 7-1: A parallel initialize function (top) with a naive reverse mode AD gradient function (bottom) that does not take the parallelism into account. Consequently, the concurrent read of the variable `val` causes a race in the reverse-mode gradient computation.

performance and correctness is due to the implied computational flow reversal of reverse-mode AD; every read becomes a write in the adjoint computation and vice versa. Consider the program at the top of Figure 7-1. It contains a simple parallel for loop that reads from the same variable `val` in all threads and sets each index of the output array `ar` to that value. Since all threads read the same value, there is a *concurrent read access* on `val`, which does not impact the final result. Computing the gradient function of this program (i.e. the derivative of the input `val`), one must accumulate all of the partial derivatives of `val` generated by uses in the outputs. Such an action unfortunately leads to a *write race* on the gradient `d_val`, which may be updated by multiple threads at the same time. Special care must be taken to avoid undefined behavior and ensure the correctness of the gradient computation while preserving as much of its parallelism as possible.

GPUs often have relatively small amounts of memory per thread. Moreover, the memory of GPUs commonly has complex performance characteristics, with global memory being slow but large, shared memory being fast but small, and the use of certain types of memory preventing the simultaneous use of large thread counts. Potential remedies involve

either sacrificing generality, by rewriting HPC applications in a differentiable domain-specific language (DSL), or resorting to approaches such as numerical differentiation.

To leverage the potential performance benefits of reverse-mode AD without sacrificing generality, one requires a tool that is capable of both handling the complex performance characteristics of GPU architectures and generating code that maintains the correctness of the gradient without sacrificing the inherent parallelism of the original program. Unlike many other tools, Enzyme[1] [271] performs AD alongside the traditional optimization pipeline by performing differentiation within the LLVM compiler [230]. Thus, we can leverage existing code transformation infrastructure to build the requisite analyses and transformations for maintaining the correctness and performance of the corresponding gradient kernel. Furthermore, LLVM provides frontends for most commonly used languages including C/C++, Fortran, Julia, Rust, Swift, and TensorFlow and backends for different hardware architectures including CPU, NVIDIA GPUs [326, 172, 415], and AMD GPUs, allowing us to build reverse-mode AD for multiple languages and architectures.

Overall, this chapter makes the following contributions:

- An algorithm for correctly generating gradients of GPU kernels and a corresponding proof sketch of correctness

- An extension to the Enzyme AD engine for LLVM that can generate gradients for GPU kernels written in either CUDA (NVIDIA) or ROCm (AMD)

- A collection of optimization passes for Enzyme/LLVM that allow generated GPU gradients to run efficiently on modern hardware

- A study demonstrating, for the first time, the feasibility of reverse-mode automatic differentiation of GPU kernels through the use of GPU and AD-specific optimizations (caching and recomputation).

[1] https://enzyme.mit.edu

7.2 Related Work

AD tools that differentiate programs at runtime are often relatively straightforward to develop using, for example, operator overloading in C++ [145, 249, 409, 34, 171, 332]. Unfortunately, in reverse mode, they generally produce a large *tape* to store operations and intermediate values for subsequent reverse differentiation, which causes challenges with their memory footprint in real-world applications.

A more efficient, but more challenging type of AD uses a compile-time transformation to translate the source code for a given function evaluation into the derivative function evaluation. Several such tools have been developed for Fortran and C including ADIFOR [45], Tapenade [163], TAF [132], OpenAD [392], ADIC [47], and ADIC2 [281]. Unlike these tools, Enzyme is based on the LLVM compiler instead of an AD-specific framework and emits gradient programs in LLVM IR instead of the original source language. This approach allows Enzyme to benefit from the language support, optimizations, and maturity of the LLVM platform.

For differentiating codes running in distributed-memory environments, libraries such as the *Adjoinable MPI* library have been developed that reverse the nonblocking communication patterns in the original code [391, 62]. Other studies have presented reverse-mode AD for OpenMP codes [44, 59, 58, 118, 180, 183] or hybrid OpenMP/MPI codes [133]. Some studies [133, 44] have identified that reverse-mode AD creates potential write races on multicore CPU programs and suggest atomic updates or privatization as solutions.

Derivatives can be computed on GPUs for programs written in certain domain-specific languages (DSLs) such as PyTorch [297], Halide [319], TensorFlow [1], or JAX [53]. The AD approach used in these languages uses the structure of, and high-level knowledge about, programs that can be written in those DSLs and does not easily generalize to arbitrary programs written in a general-purpose language such as C or CUDA. Previous works have discussed AD or symbolic differentiation for programs that call CUDA kernels [142, 144]. Such works, however, do not present differentiation of the kernels themselves or else use the forward mode of AD [48, 324].

7.3 Automatic Differentiation

This section provides a brief summary of automatic differentiation concepts that are relevant to this work. For a more thorough introduction, we refer to [147, 282, 271].

AD takes as its input a computer program P that implements a mathematical function and produces a new program that computes the derivative, or gradient, of that function. AD tools are able to produce such a derivative by examining the individual instructions of P (such as add or mul) and generating the corresponding partial derivatives of the instructions. By applying the chain rule of calculus, they then compute the derivative of the entire program by accumulating the partial derivatives all instructions of P. Any order of accumulating these derivatives is correct, but the order affects the efficiency, ease of implementation, and memory usage. Two particular strategies have become popular.

Forward or tangent mode combines the derivatives of instructions in the order in which the original instructions are evaluated, resulting in the propagation of derivatives from an instruction's input(s) to its output. Consider the instruction $v = f(w, u)$. The derivative of its output, $\dot{v}$, can be evaluated by computing

$$\dot{v} = \frac{\partial f}{\partial w} \dot{w} + \frac{\partial f}{\partial u} \dot{u}.$$

For the overall program, the derivative of all outputs $z_0, \ldots, z_m$ with respect to one of its inputs x can thus be computed by setting $\dot{x} = 1$ at the start of the program, and reading the final value of the differential or ***shadow*** $\dot{z}_0 \ldots \dot{z}_m$ at the end of the program. Computing the derivative with respect to multiple inputs requires a forward mode evaluation for every input. This is also true for numeric differentiation or finite differences, where a separate evaluation with a small perturbation for each input variable is required. Numeric differentiation has the added disadvantage of being less accurate, and requiring the choice of a step size.

Reverse or adjoint mode combines the derivative of instructions in a ***reverse pass***, which computes the derivative or ***adjoint*** of the instructions in the reverse order of the original program, and propagates them from an instruction's outputs to its inputs. Considering the same instruction $v = f(w, u)$, the derivative of inputs $\bar{w}, \bar{u}$ can be evaluated by

computing[2]

$$\bar{w} += \frac{\partial f}{\partial w}\ \bar{v}; \quad \bar{u} += \frac{\partial f}{\partial u}\ \bar{v}; \quad \bar{v} = 0.$$

The derivative of output z with respect to any input x can then be computed by setting $\bar{z} = 1$ prior to evaluating all the partial derivatives, then reading the final value of the shadow input $\bar{x}$. This allows a single evaluation of reverse mode to compute the gradient (derivative of output with respect to all inputs) in a single evaluation. Evaluating the derivative with respect to multiple outputs, however, requires an evaluation per output. In practice, programs with a large number of inputs, but few outputs (e.g. a loss function) dominate both scientific and machine learning use cases. Since reverse mode can compute derivatives in this case asymptotically faster than other methods, our work focuses entirely on reverse-mode AD.

Despite its attractiveness for practical applications, reverse mode AD is not without challenges, two of which are particularly relevant for this work. First, for a nonlinear instruction (such as x^2), one requires the original input to compute the derivative (in this case $2x$). While this is true for both forward and reverse modes, it is a challenge during the reverse pass. To provide the necessary inputs, the AD tool must evaluate all original instructions in an ***augmented forward pass*** and cache the required intermediate values (potentially causing a large memory footprint), or store only selected intermediate variables from which others can be recomputed (trading some memory for additional computation). Our work addresses analysis strategies to reduce the amount of storage needed, but does not address recomputation strategies, which are an active research subject on their own [146, 411, 18] and are beyond the scope of this work. Second, since the derivative evaluation occurs in a different order than the original program, parallelization strategies that are correct for the original program may not be correct for the derivative program, and special care needs to be taken to avoid data races. This and other challenges are addressed in Section 7.4.

[2]In reverse mode, the derivative adds to the shadow value $\bar{w}$ rather than setting it directly. This ensure the derivatives from all uses of w are taken into account. The total derivative of w is finalized when all partial derivatives have been accumulated. This is guaranteed to occur before the reverse of the instruction that defines w as all users of w must occur after w in the original program and thus all adjoints that update w must occur prior the reverse of w's defining instruction. Since we are adding to the shadow, we must also initialize the shadow to zero. This is primarily done in the forward pass when creating the primal variable. To accommodate variables which are redefined (e.g. when in a loop), the shadow is again zero initialized after its value is propagated to the shadow inputs.

Memory load

```
%res = load %ptr
```

Reverse memory load

```
%tmp = load %d_res
store %d_res = 0
atomic %d_ptr += %tmp
```

Memory store

```
store %ptr = %val
```

Reverse memory store

```
%tmp = load %d_ptr
store %d_ptr = 0
load/store %d_val += %tmp
```

Figure 7-2: Rules for memory operations. Shadow registers `d_res` and `d_val` are thread-local since they shadow thread-local registers. There is no risk of racing on thread-local data and no special handling required. Both `ptr` and shadow `d_ptr` might be raced on and require atomics in the adjoint of the load. If `ptr` (and consequently `d_ptr`) is proven to be thread-local or have constant memory, the atomic update can be replaced with a serial update or reduction, respectively.

The reverse mode of automatic differentiation is closely related to the backpropagation algorithm for neural networks, and both have been implemented in DSLs such as PyTorch [297], TensorFlow [2], and others [174, 348, 218, 90]. These DSLs do not differentiate compute kernels directly, but expose high-level operations such as matrix multiply, and provide existing superoptimized GPU kernels for both the original function and its derivative. This approach is very effective for programs that can be written within these DSLs. For existing HPC applications or those that do not easily map to a DSL, this is unfortunately not an option. For this reason, there continues to be a need for AD tools such as Enzyme that can differentiate programs written in general purpose languages.

7.4 Reverse-Mode AD for GPU Kernels

Enzyme performs reverse-mode automatic differentiation over the LLVM intermediate representation (LLVM-IR). Since Enzyme is tightly integrated within the LLVM pipeline, it can differentiate any programming language with an LLVM frontend and can target any architecture that has an LLVM backend. Most importantly, this alleviates the need for DSLs or language restrictions to apply AD to code. Prior to this work, GPU kernels could not be differentiated in reverse mode without being rewritten in an explicitly differentiable DSL (e.g., PyTorch). To differentiate GPU kernels, we extend Enzyme to handle shared-memory accesses, avoid data races in the presence of concurrent reads in the primal, differentiate

parallel control flow (e.g., `sync_threads`), and differentiate GPU-specific intrinsic functions (e.g., the LLVM-IR representation of the CUDA thread identifier `threadIdx.x`).

Enzyme first performs an ***activity analysis*** [43], which deduces what instructions and values in the function could impact the resulting gradient computation. For every active value, Enzyme creates a corresponding ***shadow memory*** location, which is used to store intermediate derivative values. For active function arguments, Enzyme expects the callee of the gradient function to pass in the shadow of each argument (see Section 7.4.4). We refer to prior work [271] for a more detailed explanation of Enzyme on serial programs. Here, we will focus on our contribution of synthesizing gradient functions for GPU kernels and the necessary changes and improvements to Enzyme.

7.4.1 GPU Memory-Aware Gradient Synthesis

Case 1

```
A:
  store %ptr
  barrier
B:
  store %ptr
```

Gradient Case 1

```
∇B:
  load %d_ptr
  store %d_ptr = 0
  barrier
∇A:
  load %d_ptr
  store %d_ptr = 0
```

Case 2

```
A:
  store %ptr
  barrier
B:
  load %ptr
```

Gradient Case 2

```
∇B:
  atomicAdd d_ptr

  barrier
∇A:
  load %d_ptr
  store %d_ptr = 0
```

Case 3

```
A:
  load %ptr
  barrier
B:
  store %ptr
```

Gradient Case 3

```
∇B:
  load %d_ptr
  store %d_ptr = 0
  barrier
∇A:
  atomicAdd %d_ptr
```

Figure 7-3: Illustrations for the case analysis of the `barrier` instruction adjoint definition.

The most challenging aspect of generating fast and correct gradient code from parallel code is reasoning about memory operations, especially on the different memory types of the GPU. Both NVIDIA and AMD GPUs have thread-local, shared (block-local), and global memory, as well as constant memory that cannot change during the execution of a kernel. We define rules for synthesizing correct gradients according to which kind of memory is accessed. We define the shadow of constant memory to be global memory, to ensure that the reverse pass is able to write the corresponding gradient to the shadow. Our approach

requires that the primal code is determinacy race-free. Thus, we assume the appropriate use of atomic accesses and barriers (see Section 7.4.2).

Memory that is known to be thread-local cannot be accessed concurrently by multiple threads and is therefore equivalent to memory in serial AD. The gradient computation can access and update non-atomically without introducing a race.

In contrast, global and shared memory can be accessed concurrently by multiple threads in the primal. In the gradient computation this can cause concurrent write accesses, and thus races, if the updates are performed non-atomically (see Figure 7-1). The generic solution is to perform all accesses and updates in the reverse pass atomically. Such an approach, however, has severe performance downsides. Instead, we translate loads and stores of global- and shared-memory locations according to the rules displayed in Figure 7-2. That is, locations that are accessible by other threads are accessed atomically, while thread-local locations such as the thread-local shadow locations are accessed non-atomically. Further, we identify the special case where all threads in a block load from the same memory location in shared memory. In this case we employ an efficient block-level reduction computation that uses synchronous warp shuffle operations instead of atomic accesses.

7.4.2 Adjoints of Barriers

In GPU programming, barriers (e.g. `sync_threads` in CUDA) can synchronize the execution of threads within a warp or block. This is especially important in the presence of shared memory because it allows threads to communicate efficiently without memory races. We define the adjoint of barrier calls to be another barrier at the corresponding location in the reverse pass and show that this is sufficient by case analysis.

Given two consecutive code blocks *A* and *B*, separated by a barrier, that write or read the same memory location, the barrier provides four distinct memory guarantees:

1. All stores in A must complete prior to a store in B.

2. All stores in A must complete prior to a load in B.

3. All loads in A must complete prior to a store in B.

4. All loads in A must complete prior to a load in B.

Figure 7-3 shows minimal examples for cases 1–3; all four cases are discussed in the following.

Case 1: Store, Barrier, Store In the primal, the store in `B` will clobber the store in `A`, causing subsequent loads to see the value stored in `B`. As a result, we must ensure that the gradient will increment only the derivative of the value stored in `B` and not the value stored in `A`. The barrier in the reverse pass ensures that only ∇`B` could read a nonzero adjoint from `d_ptr`, as desired.

Case 2: Store, Barrier, Load For the reverse code to be correct we require the load of `d_ptr`, which is the adjoint of the primal load, to happen after all `atomicAdd` operations, which are the results of the primal store. The barrier in the reverse pass is sufficient to guarantee that ordering.

Case 3: Load, Barrier, Store We require that all of the stores of `d_ptr`, which are caused by the primal load, complete prior to any `atomicAdd`, which is the adjoint of the primal store. The barrier in the reverse pass will ensure this. Note that there cannot be a race in ∇`B` because that would require a preexisting race in `B`, which is violating our precondition.

Case 4: Load, Barrier, Load In the case of a barrier between two loads, the barrier operation is superfluous and can be removed with no change in semantics. Therefore, no extra considerations are needed.

7.4.3 GPU Intrinsics and Shared-Memory Allocations

The gradient is independent of most GPU-specific built-ins and intrinsics (e.g., threadIdx.x) since they are known to LLVM to be pure, that is, independent of memory. Furthermore, most intrinisics are *inactive* and can consequently be recomputed without special handling by Enzyme. Exceptions include barriers and special memory accesses (e.g., tensor core or atomic memory operations). The former is described in Section 7.4.2, and the latter can be implemented in a manner similar to traditional memory operations.

Shared-memory allocations require explicit handling to provide adjoint locations, also allocated in shared memory, that act as shadows. In LLVM-IR, a shared-memory allocation is represented as a global value with an explicit address space that is effectively uninitialized at kernel launch time. Therefore, in addition to the shadow allocation, we generate initialization code that is executed at the very beginning of differentiated kernels.

7.4.4 Usage

Enzyme is available as a plugin for the LLVM "core" compiler component. When Enzyme is loaded into compilers such as Clang, an optimization pass is enabled that acts on calls to the `__enzyme_autodiff` function.[3] The first argument to this function is the primal that is differentiated, followed by the primal arguments interleaved with shadow locations for pointers. For usage within CUDA, one calls `__enzyme_autodiff` from inside a device kernel that is launched through the normal CUDA API. Figure 7-4 shows how the GPU function `inner` is differentiated and how the synthesized gradient, `∇inner`, looks conceptually[4].

7.5 Optimizations

GPU architectures feature multiple kinds of memory that differ in their access latency, visibility, and size. While registers and shared memory are much faster than global memory, they are limited resources on GPUs and are allocated for a kernel at launch time. If a kernel requests a large number of registers or a large shared-memory allocation, the effective available parallelism (occupancy) of the kernel is lowered to fulfill the request. This can become a bottleneck for applications since a major benefit of using GPUs is their high throughput offered by plentiful parallelism. To achieve good performance, Enzyme must consequently consider trade-offs between using slower global memory or increasing the use of registers and shared memory, which may result in fewer kernel instances being run

[3] As Julia is JIT compiled, Enzyme.jl can explicitly call Enzyme's ABI for creating derivatives, rather than loading Enzyme into an existing optimization pipeline.

[4] Note that while we show CUDA code for readability, Enzyme acts on the lower level LLVM-IR that can be targeted by various languages and parallel programming models.

```
__device__ void __enzyme_autodiff(void*, ...);

__device__ void inner(float* a, float* x, float* y) {
  y[threadIdx.x] = a[0] * x[threadIdx.x];
}

__global__ void ∇kernel(float* a, float* da,
                        float* x, float* dx,
                        float* y, float* dy) {
  __enzyme_autodiff((void*)inner, a, da, x, dx, y, dy);
}
```

```
// Synthesized by Enzyme on the LLVM-IR level
// from the definition of the inner function.
__device__ void ∇inner(float* a, float* da,
                       float* x, float* dx,
                       float* y, float* dy) {
  y[threadIdx.x] = a[0] * x[threadIdx.x];

  float dy_tmp = dy[threadIdx.x];
  dy[threadIdx.x] = 0.0f;

  float dx_tmp = a[0] * dy_tmp;
  atomic { dx[threadIdx.x] += dx_tmp; }

  float da_tmp = x[threadIdx.x] * dy_tmp;
  atomic { da[0] += da_tmp; }
}
```

Figure 7-4: A simple GPU function, `inner`, that is differentiated by Enzyme within the CUDA kernel `∇kernel` (top). A high-level representation of the synthesized gradient Enzyme would generate is shown as `∇inner` (bottom). The call to `__enzyme_autodiff` is replaced by a call to the newly generated derivative function.

(a)

```
for (int i=0; i<N; i++) {
  for (int j=0; j<M; j++) {
    use(array[j]);
  }
}
overwrite(array);
```

(b)

```
double *cache = new double[N*M];
for (int i=0; i<N; i++) {
  for (int j=0; j<M; j++) {
    cache[i*M+j] = array[j];
    use(array[j]);
  }
}
overwrite(array);
diffe_overwrite(array);
for (int i=N-1; i>=0; i--) {
  for (int j=M-1; j>=0; j--) {
    diffe_use(cache[i*M+j]);
  }
}
delete[] cache;
```

(c)

```
double *cache = new double[M];
memcpy(cache, array, M*sizeof(double));
for (int i=0; i<N; i++) {
  for (int j=0; j<M; j++) {
    use(array[j]);
  }
}
overwrite(array);
diffe_overwrite(array);
for (int i=N-1; i>=0; i--) {
  for (int j=M-1; j>=0; j--) {
    diffe_use(cache[j]);
  }
}
delete[] cache;
```

Figure 7-5: In (a), there is a sample program that uses values of an array in a loop nest. The loads of the array cannot be hoisted by LICM. The array is overwritten outside of the loop nest. Enzyme would require caching a value for every execution of the load instruction, as shown in (b) and using $\Theta(NM)$ memory. Using the cache LICM optimization, the cache could be hoisted outside the loop as shown in (c), requiring only $\Theta(M)$ memory.

simultaneously.

Like all reverse-mode AD tools, Enzyme may need to preserve values generated in the forward pass for use in the reverse. If a value is available in the reverse pass, for example, if the memory that holds it was not overwritten, Enzyme will simply use it. When a memory location holding a value required for the reverse pass is modified, however, Enzyme must ensure that the value is preserved, or cached, an action that inevitably requires additional storage.

While it is generally beneficial to reduce the amount of memory used to cache values, doing so is especially important for GPU execution. In general, the number of memory locations that need to be cached is not known at compile time. Consequently, Enzyme has to cache values in thread-local storage, allocated through the dynamic allocation function `malloc`. In CUDA, `malloc` is backed by global memory and cached in the L1 cache. Global memory is substantially slower to access than registers or shared memory, which is why cache use can dramatically increase the kernel runtime. Moreover, excessive caching can require more than the available GPU heap memory and prevent the program from being run at all. Since memory size and bandwidth are the primary bottlenecks, most of our optimizations aim to minimize global memory accesses. Our experimental results in Section 7.6 demonstrate that significant GPU-specific and AD-specific optimizations are

necessary to run the reverse pass in a reasonable time. Below, we briefly explain the most important optimizations that we use for this work.

Alias Analysis Alias analysis [9, Ch. 12] is fundamental to Enzyme's ability to determine whether an instruction can be recomputed or must be cached. Instructions that do not access memory are trivially recomputable. For instructions that read memory, Enzyme uses LLVM's alias analysis pipeline to determine whether the value is overwritten before it is required in the reverse pass. Depending on the quality of available alias information, for example, from types and restrict qualifiers, this can reduce the number of cached values significantly. However, if there are potentially aliasing pointers (e.g. two plain pointer arguments), Enzyme is required to assume that writes to one might modify any element read through the other. In the worst case, this uncertainty can force Enzyme to cache all read accesses of a constant input array.

In our analysis, we found that common math functions, such as `cos`, are seen as being able to write to *any* global memory and thus potentially overwrite most memory locations. LLVM models `libm` implementations of these functions as `writeonly` because they can set the global `errno` variable, assuming the user does not explicitly disable this potential side effect. The situation is different for CUDA code since there is no `libm` available. Instead, Clang will effectively map all available math functions onto respective CUDA builtin functions, for example, `__nv_cos`. Since the LLVM analyses and optimizations are not aware of these CUDA-specific functions, they are conservatively assumed to read and write arbitrary memory. For the sake of Enzyme's cache, we allow alias analysis to assume that common math functions do not act as barriers to recomputation.

Another significant barrier to performance is the aliasing behavior of `sync_threads`. In order to ensure correctness for multithreaded GPU programs, LLVM's aliasing properties of architecture-specific `barrier` intrinsics assume that `barrier` can read and write to most memory locations. For the same reasons as above, this assumption forces Enzyme to unnecessarily cache values. We extend Enzyme to define a `barrier` instruction `S` as having the aliasing behavior of all instructions that precede `S` until it reaches another `barrier` or the start of the kernel being differentiated.

Loop-Invariant Cache Enzyme caches the results of individual instructions rather than

(a)

```
use(x[0] + y[0]);
overwrite(x, y);
```

(b)

```
double x_cache = x[0];
double y_cache = y[0];
use(x[0] + y[0]);
overwrite(x, y);
diffe_overwrite(x, y);
diffe_use(x_cache[i] + y_cache[i]);
```

(c)

```
double sum_cache = x[0] + y[0];
use(x[0] + y[0]);
overwrite(x, y);
diffe_overwrite(x, y);
diffe_use(sum_cache);
```

Figure 7-6: (a) A sample program that loads two variables x and y and then perform some computation with the result. These variables are subsequently overwritten and thus would require caching to be available in the reverse pass. A naive cache algorithm would produce the code in (b) in which both overwritten memory locations x and y are cached. As shown in (c), one could instead cache the sum since neither x nor y is individually necessary to compute the gradient.

memory ranges. This approach can be more efficient for general programs, especially if memory access patterns are sparse. This can be problematic, however, in cases where many instructions load from the same piece of memory that must be cached. Enzyme relies on LLVM-based optimizations such as common sub-expression elimination (CSE) and loop-invariant-code-motion (LICM) [275, Sec. 13.2] to remove such equivalent accesses in the original program and subsequently prevent unnecessary caching. In several cases, however, the LLVM optimizations may not be legal, or even beneficial for the original code, but would otherwise result in a large amount of unnecessary caching.

For example, consider the program shown in Figure 7-5(a). The load cannot be optimized by LICM since it depends on the innermost iteration variable `j`. If the load is required for a reverse-pass computation, Enzyme must cache every result of the load as shown in Figure 7-5(b), resulting in an $\Theta(NM)$ cache. However, we notice that the array is only potentially overwritten outside of the loop nest, and we could have instead chosen to simply cache the total size of the memory used ($\Theta(M)$) as in Figure 7-5(c). This cache optimization detects scenarios where it is legal and profitable to cache loads from a parent loop nest, thereby reducing the total cache.

Equivalent Load Cache Similar to how the loop-invariant cache optimization remedies issues where LICM may not optimize the initial code to reduce the cache, we also present a cache-variant of common sub-expression elimination. Consider two loops that both load from an array. Because the loops are not fused, these loads cannot be dedupli-

cated by common sub-expression elimination. Consequently, Enzyme would have to create two separate caches. However, since both of these load from the same memory without a potential write in between, we can instead cache the array once and use it during the reverse pass in both places.

Cache Forwarding GPU programs commonly use shared memory as a cache for global memory when it may be used by many threads. This is highly beneficial because accesses from shared memory are much faster than loads from global memory. If that shared memory is overwritten, however, it may need to be cached for the reverse pass. The original global memory it is derived from, however, may not have been overwritten. In this case, instead of allocating a cache to preserve the overwritten values in shared memory, we can simply reload the underlying memory the shared memory is acting as a cache for, preventing an unnecessary allocation of global memory for the cache. An additional though yet unimplemented extension to this optimization is to reuse the faster shared memory as a cache for the reverse pass rather than having to load from the slower global memory.

PHI Unwrapping In addition to load and call instructions that may not be recomputable, Enzyme may also have to cache PHI instructions. PHI instructions occur when the current basic block has multiple potential predecessors. The PHI instruction forwards a value from the actual predecessor that just branched to the current block, preventing recomputation and requiring caching.

This optimization aims to compute an equivalent value to the PHI by determining a condition `C` that determines the actual predecessor of the basic block. The PHI node can then be recomputed by recomputing the condition `C` and selecting the corresponding value the PHI node would have when coming from the predecessor corresponding to `C`. Computing `C` can be done by traversing the function's control-flow graph and attempting to identify a chain of conditions to branch instructions that lead to the PHI node from a given predecessor. This cannot always be done at compile-time but nevertheless allows Enzyme to avoid caching many PHI instructions in unnecessary allocations.

Allocation Optimizations Enzyme performs most cache allocations on the heap, backed by global memory. By running the heap-to-stack optimization pass, we can lower a heap allocation into a stack allocation and subsequently open the possibility of promoting the

stack allocation to individual registers. Additionally, Enzyme may make several separate allocations for different instruction caches. A function call (such as a call to `malloc` or `free`) is expensive on the GPU. We provide a further optimization that coalesces several individual allocations into a larger allocation, thereby reducing the overhead of allocating cache memory.

Recompute versus Cache Heuristics When Enzyme deduces that a value `V` is required in the reverse pass, Enzyme explicitly caches all loads, calls, and PHI instructions necessary to compute `V`. We extend Enzyme with a heuristic to instead directly cache the value being recomputed, rather than the loads necessary to recompute it, if we predict that this will result in a smaller amount of cached memory as shown in Figure 7-6. We also extend this heuristic to find the minimal set of values to cache by determining a minimum branch cut between values that must be cached and instructions that require values from the forward pass. In general, solving for the *optimal* cache size is difficult to do at compile time because many relevant parameters such as loop bounds may not be known.

Loop Bound Calculation Enzyme frequently computes the bounds of loops, for example, to determine the size of cache space allocations or to index into the cache. Enzyme piggybacks on top of LLVM's existing scalar evolution analysis to attempt to statically deduce the size of loops. This allows Enzyme to allocate the required cache memory in advance. However, not all loops have statically known bounds. For these dynamically sized loops, Enzyme must continuously reallocate the cache inside the loop to ensure sufficient memory exists to contain the values from all iterations. When the total number of iterations is not statically analyzable, Enzyme adds a variable to cache the count for use in index computations.

Consequently, it is desirable for Enzyme to statically deduce the bounds of loops. However, LLVM's analysis passes must be conservative and account for behavior like potential integer wraparound, causing hard-to-analyze bounds on seemingly simple loops. Enzyme extends LLVM's scalar evolution to take advantage of a key fact: if one is indeed evaluating code in the reverse pass, none of the forward-pass loops could have been infinite loops. When computing bounds for cache sizes, we can consequently add the extra fact that the loop is not infinite, allowing Enzyme to statically compute bounds of additional loops.

Register Locality In contrast to virtual instruction sets like LLVM, physical architectures have a fixed set of registers available for computation. To map a computation onto a physical instruction set such as that used by a GPU, one must perform register allocation to map the virtual registers used by LLVM to a fixed set of physical registers. When there are insufficient registers available to represent all virtual registers, the compiler must spill the instruction into a stack allocation (which on NVIDIA GPUs spills to the L1 cache and subsequently global memory). Therefore, it is crucial for Enzyme to maximize the locality of virtual register uses to avoid spilling. By default, Enzyme reuses a value from the forward pass if it dominates its potential use in the reverse pass, because it will always be available without an explicit allocation. This scheme is problematic for the GPU, however, because it may increase the lifetime of registers, leading to spilling and increased global memory use.

To remedy this situation, Enzyme will choose to recompute loads from shared memory if there is register pressure. While a load from shared memory is certainly slower than reusing a register, it is still faster than a load from global memory in a potential spill.

Inlining Choosing to inline or call a function can have substantial performance implications. Inlining a function may be beneficial because it may allow Enzyme to combine loads or otherwise reduce redundant cache allocations through the loop-invariant cache or equivalent load cache optimizations. On the other hand, by calling a function rather than inlining it, Enzyme will explicitly recompute data structures generated by the function being called in the reverse pass. This action can increase register locality and may require fewer instructions to recompute PHI nodes since there are fewer potential predecessors.

7.6 Evaluation

We evaluate our approach on five established GPU-based HPC proxy applications:

- CUDA-based RSBench [384] and XSBench [385], two implementations of Monte Carlo neutron transport algorithms

- An extended version of the CUDA lattice-Boltzmann method (LBM) solver from the

Parboil benchmark suite [371], with applications in computational fluid dynamics

- CUDA-based Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH) code [205], a proxy application for computational fluid dynamics solvers

- A discontinuous-Galerkin (DG) volume integral[5][3] kernel as used in the pure Julia [41] climate code ClimateMachine.jl[6] [347] and implemented for both CUDA and AMD GPUs

7.6.1 Setup

For each application, we time just the evaluation of the code being differentiated, excluding time taken for device memory initialization and transfer or other calling code. For CUDA kernels, we explicitly increase the size of the device heap to 1 GB. RSBench, XSBench, and the CUDA.jl version of DG were evaluated on an NVIDIA 2080 Super. LBM was evaluated on an NVIDIA V100. LULESH was evaluated on an NVIDIA RTX A6000. The AMDGPU.jl version of DG was evaluated on an AMD Vega 64. Benchmarks were tested with LLVM main at commit `8dab25954b0acb53731c4aa73e9a7f4f98263030`, Julia 1.6, and Enzyme at commit `ec75831a8cb0`. The benchmark suite is available at `https://github.com/wsmoses/Enzyme-GPU-Tests`.

All benchmarks were evaluated a minimum of five times, taking the geometric mean as the final result. For each benchmark we evaluated the original kernel and the combined forward/reverse pass generated by Enzyme (Figure 7-7); the combined forward and reverse pass with various optimizations described in Section 7.5 disabled (Figure 7-10); the compile times of the benchmarks (Figure 7-14); and the scalability of the gradients compared to the original code (Figures 7-11 and 7-12).

With the exception of the LBM benchmark (see below), modifying a benchmark to enable differentiation simply required allocating and initializing shadow arrays (to store the output gradients), and creating a kernel which calls `__enzyme_autodiff` on the kernel to be differentiated, as demonstrated in Figure 7-4.

[5] `https://github.com/lcw/Heptapus.jl`
[6] `https://github.com/CliMA/ClimateMachine.jl/`

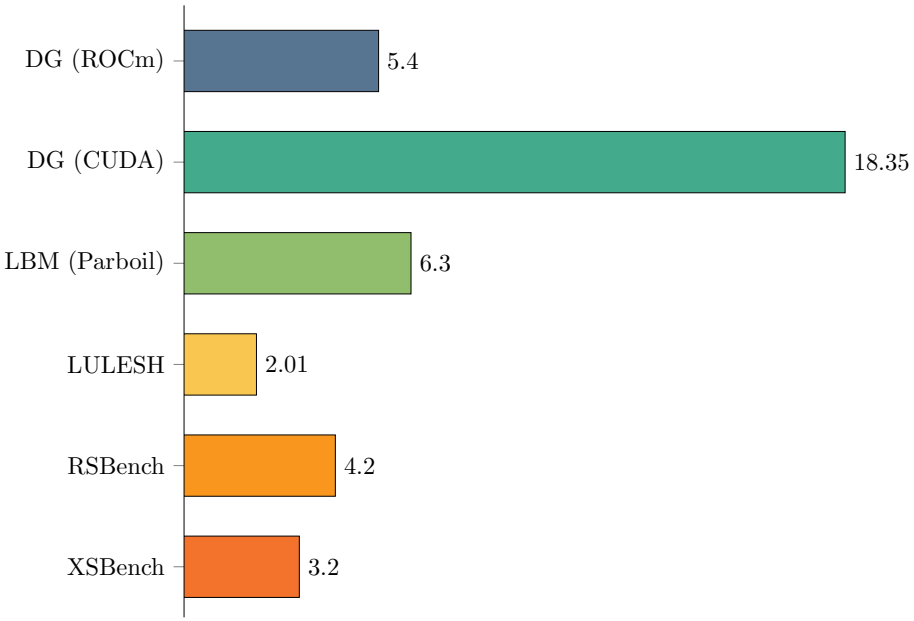


Figure 7-7: AD overhead of the benchmark applications, as compared with a single evaluation of the forward pass. An overhead of N can be read as saying that collecting the gradients of all inputs (as well as running the original code) is equivalent to running the original code N times.

```
void kern(float* src, float* dst) {
    streamCollide<<<...>>>(src, dst);
}

void lbm(int nTimeSteps, float* src, float* dst) {
    for (unsigned int i=0; i<nTimeSteps/2; i++) {
        kern(src, dst);
        kern(dst, src);
    }
}
```

Figure 7-8: Simplified version of the computation within LBM. The `kern` function calls a GPU kernel that iterates the simulation one timestep forward in time, storing the result in `dst`. The `lbm` CPU function calls the GPU kernel until all iterations have completed. The iteration must happen outside the kernel to ensure that all threads from one timestep have completed prior to performing another timestep.

(a)

```
// CPU Code
void aug_kern(float* src, float *dsrc,
              float* dst, float* ddst) {
  void* tape = Allocator.allocate(...);
  aug_streamCollide<<<...>>>(src, dsrc,
                             dst, ddst, tape);
}
void rev_kern(float* src, float *dsrc,
              float* dst, float* ddst, void* tape) {
  rev_streamCollide<<<...>>>(src, dsrc,
                             dst, ddst, tape);
  Allocator.free(tape);
}
__attribute__((enzyme(aug_kern, rev_kern)))
void kern(float* src, float* dst);

void ∇lbm(int nTimeSteps, float* src, float* dsrc,
          float* dst, float* ddst) {
  __enzyme_autodiff(lbm, nTimeSteps, src, dsrc,
                    dst, ddst);
}
```

(b)

```
// GPU Code
__global__
void aug_streamCollide(float* src, float* dsrc,
                       float* dst, float* ddst,
                       void** tape) {
  size_t idx = threadIdx.x + ...;
  tape[idx] = __enzyme_augmentfwd(streamCollide,
                                  src, dsrc,
                                  dst, ddst);
}

__global__
void rev_streamCollide(float* src, float* dsrc,
                       float* dst, float* ddst,
                       void** tape) {
  size_t idx = threadIdx.x + ...;
  __enzyme_reverse(streamCollide, src, dsrc,
                   dst, ddst, tape[idx]);
}
```

Figure 7-9: Differentiation of the combined CPU+GPU computation in LBM. The code in (a) represents host code, which differentiates the overall function `lbm`, defined in Figure 7-8. The `kern` function is annotated with custom forward and reverse passes `aug_kern` and `rev_kern`. These functions allocate a tape and call the `aug_streamCollide` and `rev_streamCollide` kernels, which are generated by Enzyme in (b).

The correctness of the generated gradients was verified by comparing with numeric differentiation. Since our benchmarks have too many parameters to use numeric differentiation effectively, only a few inputs per benchmarks were tested.

7.6.2 Benchmark Descriptions

RSBench and XSBench RSBench and XSBench are U.S. Department of Energy proxy applications that represent the core computation of Monte Carlo simulations within particle transport algorithms such as in OpenMC [327]. The majority of the runtime of XSBench is spent in memory operations with a semi-random access pattern. By calculating neutron cross-sections with the multipole method, RSBench trades off several magnitudes of memory in exchange for a significant amount of computation to unpack the data. Together, RSBench and XSBench allow us to differentiate both compute-bound and memory-bound applications, respectively.

Lattice Boltzmann Method (LBM) LBM is a particle-based fluid dynamics simulation method. It works by modeling fluid density on a lattice (grid) and in each time step

performing a streaming step (allowing fluid to flow into adjacent grid cells) and a collision step (which models the interaction of fluids flowing into a particular cell from neighboring cells). This so-called stream-collide sequence is responsible for the majority of the computational cost of typical LBM solvers and is implemented in the CUDA version of Parboil LBM in a method called `performStreamCollide_kernel`. CPU driver code calls this kernel in a loop to advance the simulation by several timesteps, as shown in Figure 7-8.

Unlike the other benchmarks tested, where the entire function being differentiated was on the GPU, differentiating LBM requires the differentiation of heterogeneous programs. Since LLVM does not yet support modules which contain both CPU and GPU code, we perform differentiation in two steps. First, we use Enzyme to generate an augmented forward and reverse pass for the GPU kernel. The forward pass is equivalent to the original function, saving any data that is required for the reverse pass and may be overwritten. The forward and reverse pass of the GPU kernels can then be imported into the CPU code by using Enzyme's support for custom derivatives. The heterogeneous AD setup is demonstrated in Figure 7-9. Note that while we demonstrate this shim layer for clarity, in practice this can be simplified for end users through the use of advanced compiler transformations or macros.

LULESH LULESH [205] is an unstructured explicit shock hydrodynamics solver, which was initially introduced as a proxy application for computational fluid dynamics on high-performance computing systems and has since been employed as a proxy application for complex fluid dynamics codes. LULESH emulates complex hydrodynamic solvers by splitting the computational domain into volumetric elements on an unstructured mesh. This allows LULESH to mimic the complex data movement characteristics of unstructured data structures. All measurements were analyzed with NVIDIA NSight Compute to discern the individual measurements of the gradient *ApplyMaterialPropertiesAndUpdateVolume* kernel from the general application runtime.

Discontinuous Galerkin (DG) The discontinuous-Galerkin volume integral[3] kernel is part of a fluid dynamics simulation model. It is written in Julia, and we use `CUDA.jl` [40] and `AMDGPU.jl` [336] in combination with Enzyme.jl [271] to synthesize and execute the kernel and its derivative. The code features GPU-specific features, such as shared memory,

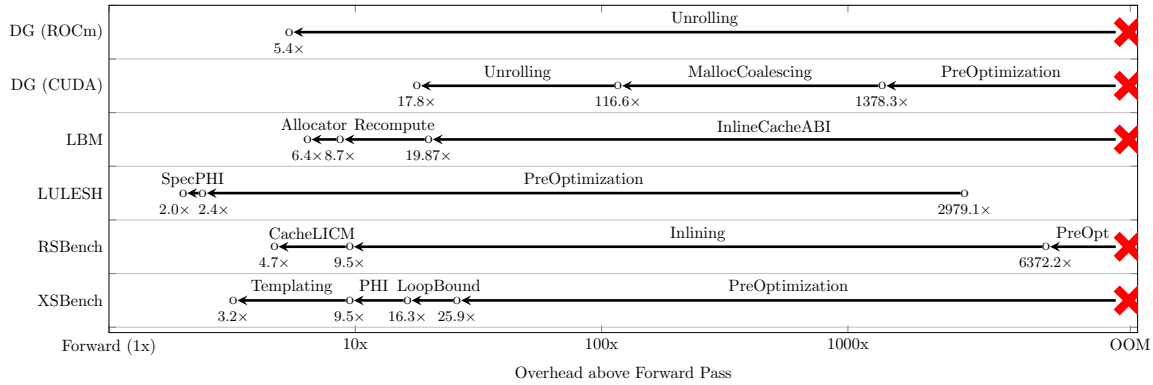


Figure 7-10: Overhead of selectively disabling AD and GPU-specific optimizations described in Section 7.5. OOM indicates running out of memory or an indefinite runtime. Each dot represents the overhead of AD compared to the forward pass alone.

and is memory bound. We modified the original code to use noncoherent memory loads in the case of `CUDA.jl` and constant memory loads in the case of `AMDGPU.jl`.

7.6.3 Results

As demonstrated by the original Enzyme work (Chapter 5) [271], embedding AD within the compiler allows one to perform AD after optimization, which is on average 4.2× faster than AD before optimization. Since prior tools perform AD at a source level, they must perform AD prior to any compiler optimizations. Although there exist no tools that we can compare against that perform reverse-mode AD on GPU kernels, we attempted to perform a similar ablation analysis here to see what a tool not implemented within a compiler might be able to achieve, if one were to be written. Without applying standard LLVM optimizations prior to AD, RSBench and XSBench take an indefinite amount of time to run. LULESH has an overhead of 2979.1× without preprocessing optimizations. LBM is able to be differentiated without preprocessing optimizations for two iterations, but exhausts GPU memory on anything larger (scaling tests use 50-600 iterations). In order to legalize Julia code for the GPU (such as the ROCm and CUDA DG codes), it is necessary to run the LLVM optimization pipeline, along with Julia's custom optimization passes. We therefore conclude that the ability to run optimizations alongside AD is in fact a precondition of successful reverse-mode AD of general GPU programs.

Overall, the combined forward and gradient generated by Enzyme have a reasonable

overhead when compared with that of the forward pass (Figure 7-7). RSBench and XS-Bench have a $3-4\times$ overhead due to the need to cache intermediate computations from the forward pass. Similarly, LBM must cache the current state variables every iteration leading to an overhead of 6.3×. The kernel evaluated in LULESH does not need to cache additional values, and as a result the 2.01× overhead is spent performing the corresponding gradient computations. The DG benchmark has a 5.4× overhead when run on AMD, primarily from the additional computation, whereas it has a 18× overhead on CUDA as it quickly exhausts the amount of available registers and the CUDA assembler decides to spill a large number of registers into global memory.

AD and GPU-Specific Optimizations To evaluate the effectiveness of the optimizations described in Section 7.5, we evaluated all benchmarks with several AD and GPU-specific optimizations being successively disabled. Not all benchmarks benefit from the same optimizations, and the order in which compiler optimizations are applied can dramatically impact performance [175]. For each benchmark, we visualize a path through the exponentially large optimization space that attempts to enable each optimization when it will have the largest impact on performance. The results of this analysis are shown in Figure 7-10. An end user trying to maximize their performance wouldn't explore all optimization combinations/paths, instead simply enabling all optimizations. As disabling optimizations quickly blows up the runtime of the program, the ablation analysis of benchmarks was run at a smaller test size to ensure the computation completed in a reasonable time where necessary.

For the ROCm DG kernel, an unrolling optimization was necessary to allow Enzyme to create the gradient without caching any additional values. Without unrolling, the GPU was unable to allocate sufficient device memory to succeed.

For the CUDA DG kernel, simply applying the standard Julia+LLVM optimization pipeline enabled the gradient to run, though at a 1378.3× overhead. Running an optimization that coalesced multiple allocations into a single `malloc` call reduced this runtime to 116.6×. Like in the ROCm case, applying unrolling eliminates any need to cache values, reducing the overhead to 17.8×.

For ablation analysis, we ran the LBM kernel for 150 iterations. The use of an efficient

CPU to GPU calling convention for caching values was necessary for the gradient to run on a problem of this size. Applying the improved recompute vs cache heuristic allowed Enzyme to detect that it could cache a double which representing a sum, rather than the individual of overwritten values. This analysis reduced the size of the cache from 80 bytes per thread to 20 bytes per thread. As a result, the AD overhead was reduced from 19.87× to 8.7×. Finally, using a LIFO allocator rather than `cudaMalloc` to allocate cache memory brought the AD overhead down to 6.4×.

For ablation analysis, LULESH was run on a computational domain size of 90^3. Applying just LLVM optimizations prior to AD brought the LULESH gradient overhead down to 2.4× from 2979.1×. As the LULESH kernel was particularly branch heavy, enabling speculative execution of ϕ predecessors when recomputing values in the reverse pass reduced the AD overhead down to 2.01×.

Running RSBench on a problem size of 10,200, LLVM optimizations alone resulted in an overhead of 6374×. By applying additional inlining, this overhead was reduced to 9.5× as LLVM could optimize between functions, enabling Enzyme to eliminate redundant values being cached, as well as use a more efficient intraprocedural caching infrastructure. Enabling the loop invariant cache and equivalent load cache optimizations reduced the overhead down to 4.7×.

Running XSBench on a problem size of 17,000,000 with LLVM optimizations, the overhead was 25.9×. Allowing Enzyme to avoid caching loop bounds when it can prove that all its instructions are inactive, drops the overhead to 16.3×. Performing PHI restructuring reduces the overhead to 9.5×. Passing the mode of simulation through a C++ template eliminates code generation of helper routines from different simulation modes and reduces the overhead to 3.2×. This leads to fewer branches in the forward pass and allows Enzyme to avoid analogous branches in the reverse pass.

Scalability We compare the scalability of our approach in two ways. First, we consider applications where increasing the problem size increases the number of threads, while maintaining constant work per thread. We plot the overhead as a function of problem size for DG and LULESH, XSBench, and RSBench in Figure 7-11. DG on CUDA, LULESH, XSBench, and RSBench maintain a constant overhead as the problem size increases. DG

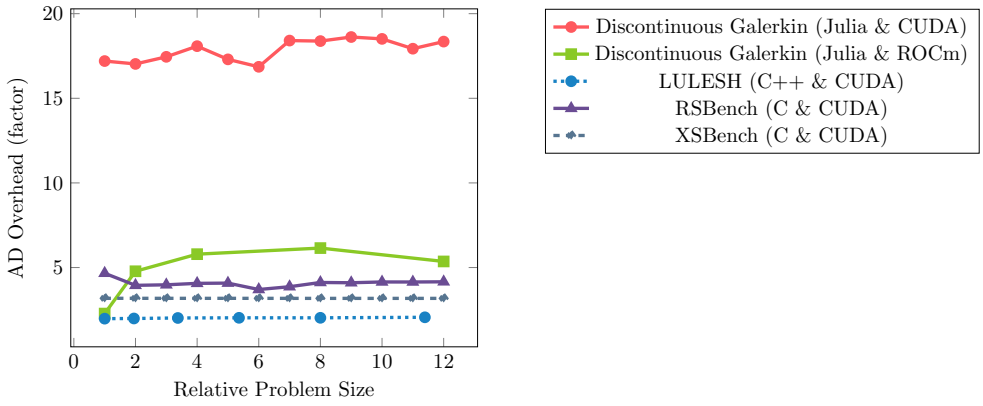


Figure 7-11: Overhead of Enzyme compared with the forward pass as the problem size and number of threads increase, with constant work per thread.

on AMD's overhead increases at the start but quickly asymptotes. When the problem size is increased in the LBM benchmark, the amount of work and number of kernel calls increase without increasing the number of threads. As demonstrated by Figure 7-12, the overhead quickly asymptotes as the additional setup required by Enzyme gets amortized across a larger number of iterations.

LULESH Case Study Automatic differentiation of LULESH's compute kernels is a prime example of the importance of running optimizations prior to reverse-mode automatic differentiation on GPUs. While the generated gradient has a 2979.1× overhead without any LLVM optimizations prior to AD, this is reduced to 2.4× by simply running LLVM's standard optimization pipeline. This does not require deep changes to LULESH or manual tuning. Using all the optimizations described in Section 7.5 resulted in a reduction of the AD overhead to ~ 2.01×. Because of the effectiveness of the optimizations and low overhead, we looked at the memory access patterns in depth to understand the impact Enzyme and its optimizations had on the memory system of the GPU.

In Figure 7-13, we analyze the memory characteristics of the *ApplyMaterialPropertiesAndUpdateVolume* kernel, using NVIDIA's NSight-Compute analyzer. We use the memory workload analysis as a guide to evaluate the performance of the synthesized gradient kernel and judge whether there are potentially missed optimizations, or common access patterns within the gradient kernel. For this kernel, the profile shows that there is an ~ 50% increase in memory traffic when performing gradient calculations. If excessive caching or

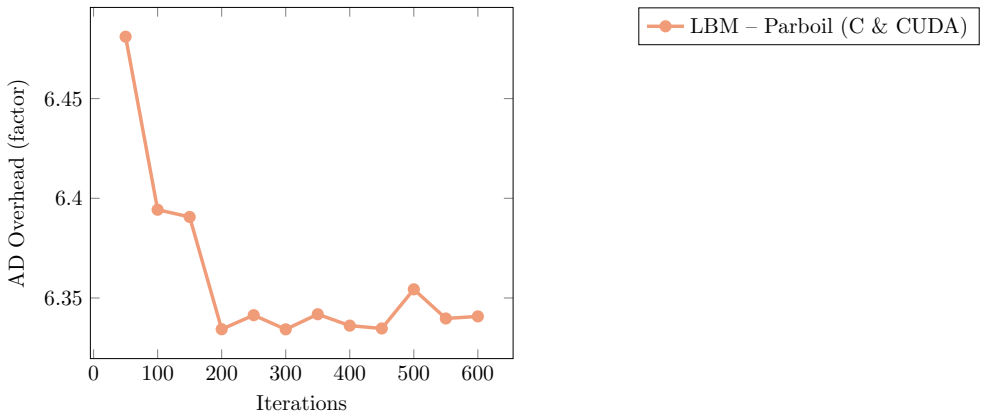


Figure 7-12: Overhead of Enzyme compared with the forward pass where work is increased while maintaining a constant number of threads.

register spilling occurred, we would have seen an increase in *Local* memory traffic. This performance report is typical of an efficient gradient kernel, which is reflected in the low AD overhead of 2.01×.

Discontinuous Galerkin (DG) Case Study We evaluated the DG kernel on both AMD and NVIDIA GPUs. The NVIDIA variant shows an overhead of 18× versus an overhead of 5.4× for the AMD variant. Performance analysis of the NVIDIA implementation unveiled two bottlenecks in the gradient kernel. The first bottleneck was caused by a large number of values reused from the forward pass. This created excessive register spilling and correspondingly increased global memory traffic. Second, some atomic increment operations on shared memory were heavily contended. Surprisingly, the AMD implementation performs much better. We hypothesize that AMD is faster because the AMDGPU LLVM backend directly optimizes for the target architecture and can perform optimizations such as target register allocation. In contrast, the NVIDIA LLVM backend targets NVIDIA's virtual instruction set architecture NVPTX and leaves register allocation to `ptxas`.

Enzyme allows the user to specify whether the gradient should be calculated with respect to an argument. We used the DG kernel to verify that applying Enzyme with all arguments set to be constant (not differentiated), would not incur any overhead.

Compile Time We compare the time spent to compile kernels with and without the gradient generated by Enzyme. In practice, when running large simulations the compile time is negligible compared with the runtime. Nevertheless, it is useful to verify that also

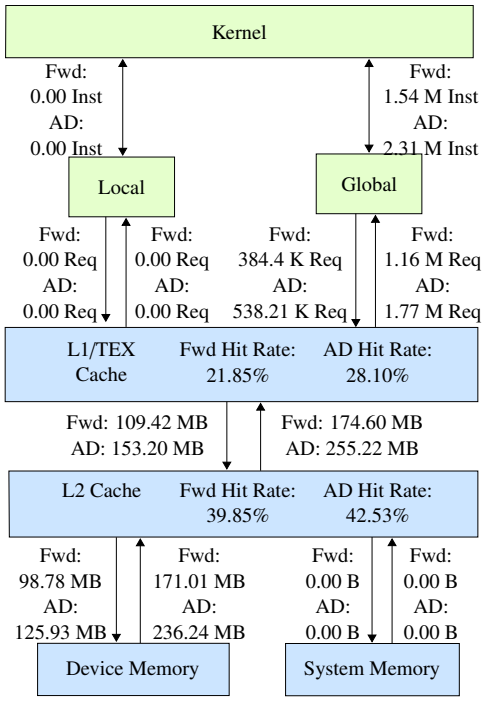


Figure 7-13: Memory workload analysis for LULESH at size 135^3 comparing the original code (Fwd) to the gradient (AD).

compiling the derivatives does not substantially change the program's overall compile time. For the four C/C++ benchmarks (LBM, LULESH, RSBench, and XSBench), we measured just the compile time of the file that contained the kernel being differentiated. This is then compared with compiling the same source file, but also generating all the requisite derivative information. This involves creating additional functions, running a second round of optimizations, and running the backend code generator for the additional kernel(s). For codes that just compile the combined forward and reverse passes (LULESH, RSBench, and XSBench), we would expect a $\sim 3\times$ overhead as in addition to the original kernel, there is now a second kernel which is twice the size (containing the forward and reverse pass). For codes in which a forward and reverse pass are requested separately, we would expect a $\sim 4\times$ overhead to account for the additional augmented forward pass, and the split reverse pass (which contains its own forward and reverse pass). These compile times are all within expectation.

The two Julia codes must be analyzed separately. As Julia is a JIT, Enzyme.jl works by running its own additional compilation within Julia's runtime and performing foreign

Test	Forward	AD	Overhead
LBM	1.54	5.65	4.32×
LULESH	8.34	23.82	2.86×
RSBench	14.99	33.29	2.22×
XSBench	15.9	23.5	1.48×
Julia DG CUDA	0.50	3.41	6.82×
Julia DG AMD	1.12	2.56	2.29×

Figure 7-14: Compile time in seconds of the source file with and without derivatives.

function calls into Enzyme loaded as a dynamic library As a result, a direct comparison is not meaningful. Nevertheless we demonstrate that the "forward" time, taken to compile the original kernel, is comparable with the "AD" time to perform a foreign function call to `libEnzyme.so`, which generates the derivative runtime function.

7.7 Conclusion

By extending Enzyme, an AD tool for LLVM, we have created the first AD tool capable of generating gradients of GPU kernels without rewriting entire applications with a differentiable DSL. Reverse-mode differentiation of GPU kernels adds several challenges including potential data races caused by the GPU's parallelism and the GPU's complex performance characteristics. We demonstrate an algorithm for differentiating GPU-based parallel control flow and other intrinsics that ensures the correctness of the resultant gradients. To maximize performance of the generated gradients, we introduce several novel AD and GPU-specific optimizations. Through various ablation analyses, we show how without these optimizations reverse-mode GPU AD is intractable in practice. We demonstrate reasonable performance and scalability on several applications relevant to the HPC community.

There exist several avenues for future work. Many of the optimizations described in Section 7.5, especially those involving caching, could make better use of shared memory, when available. For example, with rare exception, Enzyme currently maintains the GPU schedule described in the forward pass for use in the reverse. One could imagine allowing Enzyme to reschedule a kernel in such a way that minimizes potential races and therefore

allows better performance. Moreover, Enzyme currently identifies constant shared-memory indices as the only scenarios where it can perform a reduction rather than falling back to an atomic increment. Extending Enzyme to more aggressively identify locations where it can perform a reduction rather than atomics can result in additional performance boosts, especially in kernels that, like the DG kernel, make heavy use of shared memory (see Section 7.6.3). Extending Enzyme to support Forward and Mixed-Mode [40] AD may provide potential performance boosts by allowing Enzyme to choose the differentiation algorithm expected to perform fastest for a particular workload. Moreover, support for parallelism demonstrated here in the context of GPUs can be extended to support both CPU parallelism and distributed frameworks such as MPI to allow Enzyme to efficiently differentiate a wider variety of HPC applications.

Chapter 8

Scalable Automatic Differentiation of Multiple Parallel Paradigms

8.1 Introduction

Derivatives are at the core of many modern applications in science and engineering, such as machine learning [31], gradient-based optimization [135, 257], inverse problems [131], and computer graphics [238]. Automatic differentiation (AD) is a method for the automatic generation of derivatives of mathematical functions implemented in computer programs. AD is able to compute derivatives accurately to machine precision, unlike finite difference approaches.

Parallel computation, using a variety of frameworks, has become the de facto standard for large-scale computing and machine learning applications. This commonly involves using parallel dialects and frameworks such as the Message Passing Interface (MPI) [148] to provide distributed parallelism, or OpenMP [65] and Julia tasks [41] for shared-memory parallelism, as well as higher-level frameworks such as RAJA [33].

In addition to being difficult to create any derivatives of parallel programs, it is desirable to preserve the original program's parallelism for the accumulation of derivatives. This is not always straightforward, particularly in the so-called reverse-mode AD or the closely related back-propagation [59, 58, 44, 118, 180, 183], which will be briefly explained in

Section 8.3.

This chapter demonstrates how using a common low-level compiler infrastructure to synthesize adjoints of parallel codes enables differentiation across a wide variety of parallel models and source languages. To this end, we extend the Enzyme automatic differentiation framework [271], which already supports synthesizing adjoints of GPU kernels [272] to arbitrary parallel frameworks representable as a directed acyclic graph (DAG) of dependencies. To showcase the generality of our approach, we differentiate MPI (distributed parallelism), OpenMP (multicore parallelism), Julia Tasks (multicore parallelism within a JIT), and describe how additional frameworks can be supported by simply marking the parallelism.

By enabling support for the underlying programming models within the compiler, we are able to differentiate any parallel framework built on top of them such as RAJA (running atop OpenMP and MPI) and MPI.jl (Julia bindings for MPI). Moreover, we demonstrate that differentiating low-level parallelism concepts such as shared and thread-local memory automatically yields support for higher-level primitives such as reductions or `firstprivate` variables. Finally, we showcase how jointly supporting these parallelism models in one tool naturally enables differentiation of hybrid parallel programs, and that deep integration of AD into the compiler enables performance optimizations usually only available in domain-specific/functional programming languages. Overall, this chapter makes the following contributions:

- An extension to the theory of reverse-mode differentiation of single-static-assignment (SSA) intermediate representations to handle parallel execution of instructions, and thus differentiation of parallel languages and constructs that lower to such a representation.

- A demonstration of how implementing this model within the Enzyme AD engine enables end-to-end, automated reverse-mode differentiation of parallel constructs (OpenMP, MPI, RAJA, Julia Tasks, etc) written in an LLVM-compatible language (C/C++, Julia, Fortran, Swift, Rust, Python, etc).

- Experimental results for codes from the LULESH [205] benchmark suite written in C++/OpenMP, C++/MPI,

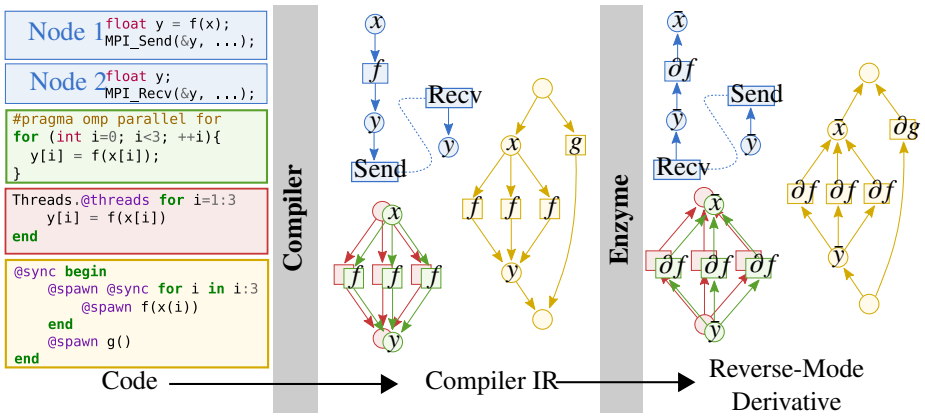


Figure 8-1: The compiler lowers the various parallel programming languages (left) into a common representation (center). Some constructs such as Julia tasks and OpenMP work-sharing loops may result in an almost identical representation. The automatic differentiation rules in Enzyme can be written for the intermediate representation, greatly simplifying the generation of reverse-mode derivatives (right) for the input languages and constructs, and further enabling compiler-optimizations.

C++/MPI+OpenMP, C++/RAJA, and Julia/MPI.jl and parallel variants of the miniBUDE mini-app [305] written in C++/OpenMP and Julia/Tasks.

8.2 Related Work

AD tools including TAF [283] and Tapenade [182] have offered differing levels of support for OpenMP. Both tools perform source-to-source transformation, have to express the gradient of OpenMP/MPI programs using valid OpenMP/MPI, and have to hence implement rules for many different OpenMP clauses. By working on LLVM IR, we avoid having to explicitly handle e.g. firstprivate/lastprivate variables, which are expressed in LLVM IR using standard assignment instructions at appropriate locations. Moreover, embedding within the compiler enables Enzyme to differentiate after (parallel) optimization, including the ability to hoist parallel code out of loops and providing better aliasing information. Special treatment of MPI reversal was implemented in adjoint MPI libraries [340], integrated into various AD tools such as CoDiPack [332, 333], ADOL-C [409], Tapenade [163], TAF [134] and dco [250], and used in applications such as the computational fluid dynamics (CFD) solvers SU2 [11], OpenFOAM [383] and STAMPS [279], and the NASA Ice Sheet System Model [227]. The published solutions for MPI require modifications to the original code,

including the use of special MPI function signatures. Adjoint MPI library extensions are developed separately from the AD tools used for the remaining program, and can interfere with certain program analyses like activity analysis [374]. Instead, Enzyme covers MPI in a transparent and seamless manner without manual intervention.

Reverse-mode differentiation of parallel read access to shared memory may result in concurrent increment access and thus require special treatment to avoid data races. Enzyme uses atomic updates whenever the analysis can not otherwise guarantee safe access. For the special case of stencil loops, PerforAD [181] instead provide a Python-based DSL that uses loop transformations to avoid concurrent increments during the reverse sweep. The functional programming language Futhark [345] differentiates high-level parallel routines and the authors discuss the use of generalized histograms, while other work considers generalized reductions [179] for the same task. Enzyme [271, 272] previously introduced support for race-free GPU-parallel programs (CUDA, ROCm) with support for different memory types, and block-level synchronization, as well as relevant AD and GPU-specific optimizations. This chapter extends the work in Chapter 7 to differentiate any DAG-based parallel framework in a single tool, and alongside novel generic parallel optimizations.

8.3 AD Background

Differentiation of programs is performed by augmenting each individual instruction with auxiliary instructions to compute its partial derivative, and augmenting each individual variable with an auxiliary variable to hold derivative values. The derivatives are accumulated following the chain rule of calculus to obtain the derivatives of the overall program. The order in which individual derivatives are accumulated does not change the overall result, but does affect the run time and memory consumption.

Many tools implement AD capabilities using a variety of strategies and supporting input languages including C [47, 163], C++ [145, 171, 332, 398], Fortran [163], Julia [325, 186], or MATLAB [46], while machine learning frameworks such as TensorFlow[2], PyTorch [297], JAX [53], and DEX [298] support AD natively.

Two strategies are common: The *forward mode* accumulates derivatives in the order of

the original computation, and is efficient for programs with few differentiable inputs and an arbitrary number of differentiable outputs. In contrast, the *reverse* or *adjoint mode* and the closely related *back propagation* are efficient for programs with an arbitrary number of differentiable inputs and few differentiable outputs. This is a common situation in machine learning, engineering and science, where functions with millions of input parameters are commonly optimized subject to a scalar loss function.

Reverse mode accumulates derivatives in the inverse order of the original computation, requiring data flow reversal and special handling for overwritten values that must be preserved or recomputed for the derivative computation of nonlinear instructions. One common approach is to trace the computation at run time using operator-overloading. Another approach is to use source-rewriting before compilation, which significantly reduces the performance overhead of differentiation, at the cost of more complex tool development. The Enzyme approach is closely related to source-rewriting, but has unique advantages due to its deep integration into the LLVM compiler. We refer to [271, 272] and [147, 282] for detailed discussions of Enzyme and AD, respectively.

8.4 Differentiation model

Enzyme uses reverse mode AD by default, and the remaining discussion will be exclusively about this mode. We will refer to an instruction and its auxiliary derivative instruction as *primal* and *adjoint*, and we will refer to auxiliary variables as *shadow*. We will identify the shadow of the output of an instruction `I` with `shadow(I)`, which represents the derivative of `I`. Within Enzyme, taking the derivative of an instruction `I` involves four steps:

1. Load and zero `shadow(I)`.
2. Compute the partial derivative of `I` w.r.t. its inputs.
3. Multiply the result of (2) with the shadow retrieved in (1).
4. Increment the shadow of `I`'s input with the result of (3).

Evaluating the adjoint of all instructions in reverse order ensures that when evaluating the adjoint of `I`, `shadow(I)` contains the total derivative, or sum of all partial derivatives from

its uses. This holds because the adjoints for all uses of `I`, which add the corresponding partial derivatives to `shadow(I)`, occur before the adjoint of `I` when run in reverse order since all uses of `I` must occur after `I` when run forward.

8.4.1 Differentiating parallel tasks

Instructions within fork-join parallel programs do not have a defined order in which they are run. Instead, instructions form a directed acyclic graph (DAG) of permissible orderings. A node in the DAG with multiple children represents a spawn, and a node with multiple successors represents a sync. Reverse mode AD of such a program requires reversal of that DAG. The above differentiation model has to be extended for this situation. Since `I` must still dominate all the uses of `I` in the original (primal) program, all adjoints of the uses are guaranteed to have been executed prior to the adjoint of `I`. The adjoint of those uses, however, may occur in parallel. When operating on a parallel program, Enzyme will perform an atomic addition or reduction when incrementing the shadow of an instruction that is not thread-local. This ensures that the total derivative is available and the computation is correct.

Differentiation of a parallel for loop and `spawn`/`sync` pair can now be shown to correctly implement the reversal of the DAG. The `sync` in the primal is transformed into `spawn` in the adjoint, and a `spawn` in the primal is transformed into a `sync`. A parallel for loop spawns off several tasks in parallel that are subsequently synchronized. Differentiating a parallel for loop results in a parallel for loop of adjoints at the corresponding location in the reversed DAG. This is equivalent to the `sync` of the primal parallel for being transformed into a `spawn` of all tasks constituting the loop. See Figure 8-2 for an illustration. For a thorough example and proof of how to differentiate parallel control flow in the specific context of a GPU-style barrier in Enzyme (as opposed to any general parallelism, described here), see [272].

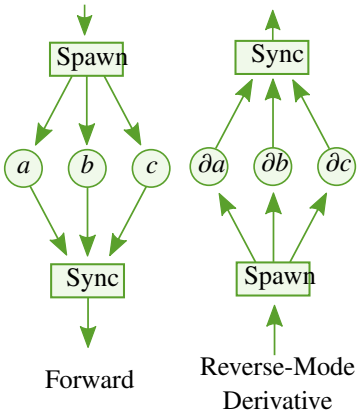


Figure 8-2: Illustration of Correctness for Parallel AD: The control and data flow is reversed, hence inverting the data- and control flow. In the forward pass (left) the control flow goes from spawn to sync. In the reverse-mode derivative (right), the locations of spawn and sync are reversed.

8.4.2 Differentiating message passing

MPI's model can be thought of as an implicit parallel for loop across the entire program, with distinct address spaces, focusing on explicit data management as opposed to execution management. MPI communication routines do not expose the implicit parallel for construct but instead expose only explicit data management using function calls with inputs and outputs. MPI's nonblocking communication (e.g., `MPI_Isend`, `MPI_Irecv`, `MPI_Wait`) can be treated as parallel task constructs, where `MPI_Isend` and `MPI_Irecv` dispatch a task synchronized at the corresponding `MPI_Wait`.[1] Differentiation of MPI in this fashion is quite efficient and results in twice the number of MPI calls, for both the primal and derivative values (which may be able to be fused during optimization), and at most thrice the amount of MPI-related memory (the original buffer to send/receive, the derivative buffer to send/receive, and potentially a temporary buffer for derivative accumulation).

8.4.3 Caching of intermediate results

The adjoints of instructions often require the arguments of the original value. For example, when computing the adjoint of x^2, one needs to preserve the original value of x to compute

[1] Depending on MPI implementation, and parameters this may only specify that the task on the current node has completed (e.g. the `MPI_Isend`) and not necessarily that the corresponding task on the partner node has completed (e.g. the `MPI_Irecv`).

```
void square(double* data) {
  #pragma omp parallel
  {
    int tid = threadid();
    data[tid] = data[tid] * data[tid];
  }
}
```

```
void outlined(double*& data) {
    int tid = threadid();
    data[tid] = data[tid] * data[tid];
}
void square(double* out, int start, end end) {
  __kmpc_fork(outlined, out, start, end);
}
```

Figure 8-3: ***Left:*** An OpenMP function which squares an element of an array on each thread. ***Right:*** The compiler lowers this construct to a closure `outlined` of the body and a call to the `__kmpc_fork` runtime call which runs the closure on each thread.

the adjoint $2x$. If the instruction that computed x cannot be rerun and the value is not otherwise available in the reverse pass, the value needs to be cached. Using a minimum-cut recompute vs cache analysis [272], Enzyme determines a minimal set of values that must be preserved in order to satisfy the dependencies of the reverse pass. Enzyme allocates caches in one of three ways:

1. Allocate a stack variable if that variable is guaranteed to be alive for the entire duration of the differentiation

2. Allocate an array prior to the loop and store the value in a slot indexed by the loop variable if the value is computed within a loop with known size

3. Dynamically reallocate an array within the loop if the loop does not have a known size

8.5 Compiler-Integrated Differentiation

In contrast to existing differentiation approaches, Enzyme performs differentiation inside of the compiler. This enables Enzyme to access and modify the program at a variety of stages during the compilation pipeline. Deeply integrating AD within the entire compiler stack provides Enzyme with a large amount of flexibility. For example, Enzyme can run additional optimizations both prior to and after differentiation, identify source-line information from metadata, rewrite and modify library calls, and even run a JIT compiler. In addition to enabling new optimization opportunities, these capabilities thereby allow Enzyme to handle a wide array of parallel constructs in a robust and concise manner by writing a few core

routines that can be generally applied to parallel methods.

Enabling support for parallelism within our extension to Enzyme requires three steps:

1. Identifying a runtime call or construct which enables parallelism. This enables Enzyme to produce parallel-safe derivative accumulation.

2. Telling Enzyme how to call the parallel runtime call, which it will call with an automatically generated derivative of the body of that construct.

3. Marking any information which is required to compute the derivative of the parallel construct as needing to be preserved by Enzyme's caching infrastructure.

Given all this, Enzyme empowers the user to differentiate parallel constructs without rewriting their original code.

8.5.1 Identifying Parallel Constructs

The easiest way for Enzyme to identify a parallel construct is by identifying a corresponding runtime call. Enzyme provides several utilities for recognizing function calls that match a certain pattern. For example, when compiling with Clang (the C/C++ compiler frontend for LLVM), Flang (the Fortran compiler for LLVM), or MLIR (a higher level intermediate representation), a program with OpenMP parallelism will call the `__kmpc_fork_call` function to run a given closure on all threads (see Figure 8-3). When writing programs with MPI for parallelism, the LLVM IR will contain calls to functions like `MPI_Isend`, `MPI_Irecv`, and `MPI_Wait` which asynchronously send data to another process, receive data from another process, and wait for a given operation to finish, respectively. Identifying the parallel constructs from Julia is somewhat more difficult because some of its parallel runtime calls do not have a unique ABI and instead create just-in-time compiled functions. Instead of identifying these calls from the (potentially inlined) assembly structure of the function, Enzyme can explicitly mark a source-level Julia method (such as `Base.threads_for`) as matching a parallel pattern (e.g. a call to task creation). In particular, the ability to leverage the compiler to mark arbitrary functions enables Enzyme to be invariant to the randomized names generated by Julia's JIT. This not only permits Enzyme to recognize the individual task creation mechanisms, but alternatively can be used to

```
void ∇square(float* data, float* d_data) {
  // Allocate an array to cache the values of data[i]
  // from the reverse pass, so the original value can
  // be used to compute the reverse pass.
  float* cache = new float[numthreads()];

  // Run the forward pass on every thread.
  __kmpc_fork(aug_outlined, data, d_data, cache);
  // Run the reverse pass on every thread.
  __kmpc_fork(rev_outlined, data, d_data, cache);
  // Free the cache
  delete[] cache;
}
```

```
// Bodies automatically generated by Enzyme when
// informed about the closure.
void aug_outlined(float*& data, float*& d_data,
                  float*& cache) {
  int tid = threadid();
  cache[tid] = data[tid];
  data[tid] = data[tid] * data[tid];
}
void rev_outlined(float*& data, float*& d_data,
                  float*& cache) {
   int tid = threadid();
   d_data[tid] *= 2 * cache[tid];
}
```

Figure 8-4: ***Left:*** Gradient of square (ref. Figure 8-3). This calls the OpenMP parallel runtime call twice, once for the forward pass, and once for the reverse pass. ***Right:*** The forward and reverse passes of the outlined OpenMP parallel body.

identify an entire parallel for-loop construct directly, instead of the underlying tasks which implement it.

8.5.2 Differentiating Parallel Constructs

Now that Enzyme can identify a program's parallelism, Enzyme must be taught how to call these parallel constructs in order to fill in the derivative information. Some parallel constructs, such as OpenMP, or Julia Tasks take a function closure. Differentiating functions which call a closure requires telling Enzyme to differentiate the closure body and potentially wrapping the auto-generated derivative of the closure to match the expected ABI and calling convention. See Figure 8-4 for an example of Enzyme-generated derivative closures for the OpenMP program in Figure 8-3. Finally, one needs to tell Enzyme the corresponding adjoint of the function, just like any other functions or LLVM instructions. As an example, differentiating `MPI_ISend` results in a `MPI_Wait` and differentiating a Julia spawn task (`Base.enq_work`) in a corresponding Julia task wait (`Base.wait`).

8.5.3 Data caching

The final step to enable differentiation is to inform Enzyme about any information which must be preserved to compute the adjoint of a parallel construct. As an example, consider an OpenMP worksharing loop construct (`#pragma omp parallel for` which divides an iteration space of arbitrary size to be run efficiently on the available system threads. The

```
int send(double* data, double* d_data, int dst) {
  // The original and shadow requests.
  MPI_Request req, d_req;

  // The forward pass of Isend, which also stores what
  // type of instruction (and its buffer) for use in a
  // reverse wait.
  MPI_Isend(data, MPI_REAL, dst, req);
  d_req = {ISend, d_data, ... };
  ... // Code for corresponding Irecv
  MPI_Wait(&req);

  // Derivative of MPI_Wait:
  //  If the origin task was an Isend, perform Irecv
  if (d_req.type == ISend)
    MPI_Irecv(...);
  ... // Derivate code for corresponding Irecv
  // Derivative of MPI_Isend
  MPI_Wait(...)
}
```

Figure 8-5: An asynchronous MPI send request and its corresponding derivative. Since the derivative of the wait must know what type of instruction it synchronized in order to spawn of its corresponding adjoint in the reverse, the request type is stored in the shadow request in the forward pass. A full MPI program would also need to call an analagous `recv` and derivative on the destination node.

bounds of the loop must be preserved for the adjoint construct to execute a corresponding worksharing loop across the same number of iterations. Caching data for a `MPI_Wait`, however is more difficult. The derivative of a wait on task `t` is to spawn the corresponding derivative task `shadow(t)`. However, MPI has multiple types of tasks (send, receive) which may be synchronized by the same `MPI_Wait`. This can be resolved through the use of shadow variables. We can define the shadow variable of the original request to store what task was being waited upon. Therefore when computing the adjoint of `MPI_Wait` in the reverse pass, Enzyme can look inside the shadow request to identify whether it should create an `MPI_Isend` or `MPI_Irecv`. This is demonstrated in Figure 8-5.

8.5.4 General Applicability

Existing tools to differentiate parallel programs must understand every parallel construct in the language they are designed to differentiate. In contrast, by operating in the compiler, we can choose to instead differentiate parallel programs with many constructs (e.g. pri-

vate memory, reductions) after they have been lowered into simpler operations (e.g. load and store operations). This enables Enzyme to differentiate these constructs without any explicit support being required, as Enzyme already knows how to differentiate memory operations. Similarly, higher level parallel languages such as RAJA, which internally implement parallelism using lower level frameworks (e.g. OpenMP) do not need any explicit support to be handled, since we can choose to differentiate after it has been lowered to OpenMP. This flexibility extends across languages. Adding the corresponding handler for an MPI call in Enzyme differentiates MPI code regardless of whether it was written directly in C++, or using the MPI.jl [60] wrapper inside Julia. Of course, this does not mean that differentiation needs to be applied at the lowest level, but that differentiating a single lower level construct enables differentiation of several higher-level routines and languages. For example, even though Enzyme differentiates Julia tasks, which are used to implement a "parallel for" in Julia, we still also provide an explicit derivative for the Julia "parallel for" for performance. In contrast, differentiating certain memory constructs (like private memory), or any code able to be optimized may be faster when differentiated at a lower-level since reducing the work of the original code can make an outsized impact in the reverse program [271, 272].

8.5.5 Optimization and Differentiation

Running optimizations prior to differentiation was found to provide a significant speedup in the original Enzyme work (Chapter 5). This effect occurs for two primary reasons. First, the additional optimizations result in simplified code which has improved analysis properties (e.g. aliasing, readonly, etc). Secondly, the additional optimizations reduce the work done by the function being differentiated, which in turn, enables the corresponding generated derivative to perform less work in both the forward and backwards passes. The need to optimize parallel programs has been well studied in a variety of works such as Tapir applying optimizations for Cilk [343] and OpenMPOpt [95]. This need is accentuated in the context of differentiation where improving aliasing properties can enable Enzyme to avoid unnecessarily caching variables for use in the reverse pass. For example, without op-

```
void fp(double* out, double in) {
  #pragma omp parallel for firstprivate(in)
  for (int i=0; i<N; i++) {
    out[i] = in;
    in = 0;
  }
}
```

```
void fp(double* out, double in) {
  #pragma omp parallel
  {
    double in_local = in;
    #pragma omp for
    for (int i=0; i<N; i++) {
      out[i] = in_local;
      in_local = 0;
    }
  }
}
```

```
double ∇fp(double* out, double* d_out, double in) {
  double d_in = 0;
  #pragma omp parallel for firstprivate(in)
  for (int i = 0; i < N; i++) {
    out[i] = in;
    in = 0;
  }
  // Run the reverse pass
  __omp_parallel(rev_outlined, out, d_out, in, d_in);
  return d_in;
}
void rev_outlined(int tid, double*& out, double*& d_out,
                           double& in, double& d_in) {
  double d_in_local = 0;
  int lb = 0, ub = N;
  __omp_for_loop(tid, &lb, &ub);
  for (int i=ub-1; i>=0; i--) {
    d_in_local = 0; // adjoint of in_local = 0
    // adjoint of out[i] = in_local
    d_in_local += d_out[i];
    d_out[i] = 0;
  }
  // Fall back to atomic if not proven thread local.
  atomic { d_in += d_in_local; }
}
```

Figure 8-6: ***Top Left:*** An OpenMP function that uses firstprivate memory to set the first iteration handled by each thread to `in`, the remainder to 0. ***Bottom Left:*** An explicit version of the code on the top left, with firstprivate being replaced with an equivalent thread-local `in_local`. ***Right:*** C code representing the gradient generated by Enzyme.

timization an OpenMP closure function captures all of the surrounding variables by value, and can potentially alias any memory. Moreover, applying parallel optimization after differentiation may also help. For example, such an optimization may be able to merge the two parallel fork calls made in Figure 8-4. We evaluate the impact of running OpenMPOpt in the context of differentiation in Section 8.7.

8.6 Other Parallel Constructs

This section gives an overview of how supporting parallel control flow (parallel for, task create/wait) and memory can enable support for other common parallel constructs.

8.6.1 Memory

Local vs Shared Memory

Enzyme is designed to have common caching and adjoint increment routines that can be used to implement the adjoint of any instruction of a function call with relative ease. Intro-

ducing a new parallel model does not require a modification to the caching infrastructure besides informing Enzyme about what calls are parallel.

The common adjoint increment routine begins by performing analysis to detect whether the shadow memory location being modified is thread- (or node-) local. This analysis builds of alias analysis to deduce if any allocation, or more specifically, offset into memory, could be used on another thread. As an example, an allocation defined within a thread which is not captured must be thread-local. If the memory location is thread-local, Enzyme performs an efficient serial load, add, and store. If this cannot be proven, Enzyme will next attempt to prove that the given memory location is the same for all threads within a parallel for loop (for all containing parallel loops). If this is the case, Enzyme will look in its catalog of reductions to see whether a reduction implementation for that style of thread exists; if so, Enzyme then will use it to sum the contribution for all threads. If none of these situations apply, Enzyme will perform an atomic add. Besides optionally registering a new reduction, a parallel framework designer adding Enzyme support can inform Enzyme that a given location is thread-local. It is legal to fall back and mark every location as being shared among threads (resulting in many atomics/reductions), but doing so may not be desirable for performance. Enzyme provides several helper methods for marking thread-local properties. The shadow of function-local registers and allocations can be marked as thread-local (this is the case for OpenMP, MPI, and CUDA but not for pthreads, Julia tasks, or Cilk tasks). Enzyme supports a differentiation configuration which assumes that the generated derivative function will itself be called in parallel and that any derivative memory location passed as an argument may be accumulated in parallel. While we found these options sufficient for OpenMP and MPI, additional thread-local settings can be implemented (and thus made available to any parallel model that chooses to apply them).

Private Memory

OpenMP and other parallel frameworks have a variety of different memory clauses. For example, OpenMP private (and its cousins firstprivate / lastprivate) specifies that a variable has a separate copy per thread (with first private initializing the thread-local value to the

```
double min_per_thread[num_threads()];
#pragma omp parallel
{
  double min_value = 0;
  #pragma omp for
  for(int i = 0; i < N; i++)
    min_value = min(data[i], min_value);
  min_per_thread[omp_get_thread_num()] = min_value;
}
double final_val = 0;
for(int i = 1; i < omp_get_num_threads(); i++)
  final_val = min(final_val, min_per_thread[i]);
```

Figure 8-7: A manual user-written min reduction, as simplified from its use from the `CalcCourantConstraintForElems` and `CalcHydroConstraintForElems` functions in LULESH. While this could be rewritten to use higher-level reduction routines which can be handled by both Enzyme and other tools, differentiating it "as-is" requires correct handling of a variety of OpenMP constructs.

value outside the loop and the lastprivate specifying that after the loop completes, the final iteration's thread-local value should be copied to the variable outside the loop). These constructs are already lowered to allocations and stores (as required) at the semantically correct location. Therefore, no additional work is required to handle these.

Consider the program at the top left of Figure 8-6. Since the variable `in` is marked firstprivate, a thread-local copy of `in` will be created, initialized to the argument, as is made explicit on the bottom left of Figure 8-6 with `in_local`. When executed, the first iteration handled by each thread will set `out[i]` to `in`, whereas all other iterations will set `out[i]` to zero.

Differentiating this with Enzyme will produce the code to the right in Figure 8-6. In the reverse pass, the reverse for loop will set the derivative of `in_local` to zero at the start of an iteration (adjoint of `in_local = 0`), then increment the derivative of `in_local` by the derivative of `out[i]` (adjoint of `out[i] = in_local`). This approach simplifies to merely setting the derivative of `in_local` to the derivative of the last iteration when run in reverse, or equivalently the first iteration when run in the original program. Since the primal code set the first iteration of each thread equal to `in`, the correct adjoint is indeed the sum of the derivatives of all the indices that were set to `in`. This case would be especially challenging for any source-to-source AD transformation tool since OpenMP has no "for" construct that

will subdivide the loop and then reverse the order of each per-thread chunk. In contrast, not only is this possible to do on the LLVM level with Enzyme, but is automatically handled by handling the parallel and memory primitives alone.

Reductions

Proper handling of private and shared memory allows Enzyme to handle higher-level parallel constructs built on top of memory, regardless of implementation. For example, the C++ version of LULESH implements a custom reduction, shown in Figure 8-7[2]. In contrast, RAJA provides a custom reduction operation/template for later use. Both reduction styles are automatically handled by Enzyme.

8.6.2 Concurrent Caching

Instructions and allocations computed within a parallel region create thread-local values or registers. Special care must be taken to ensure that the same values created within each thread are available and mapped to the corresponding thread in the reverse pass. Enzyme caches such values by preallocating memory for each thread and storing each value at an index corresponding to the current thread ID. If the same threads used for the forward-pass are also available at the corresponding time in the reverse pass (this includes Julia threads, the LLVM OpenMP runtime[3]), the corresponding reverse computation will access the corresponding caches indexed by their thread ID. If a different number or set of threads may be available, the parallel framework must inform Enzyme how to remap the threads.

Caches of values computed within a worksharing parallel for loop which does not specify how the loop's iterations map to threads, however, can be stored in a location indexed by the iteration of the for loop. This approach provides flexibility in how iterations are distributed among threads and even permits a different mapping of threads to iterations in the reverse pass.

[2]Depending on the size and parallel overhead, it may be more efficient to implement parallel min as a divide-and-conquer. The example in Figure 8-7 is used by LULESH, and the divide-and-conquer style version is also to be handled by Enzyme.

[3]This is stronger than the current OpenMP specification. However, as Enzyme exists within LLVM, this can be assumed. If the LLVM OpenMP runtime is changed to no longer have this property, Enzyme can check the LLVM version it was built against and select a different cache mapping.

8.6.3 High-Level Language Constructs

Foreign Library Calls

Unlike statically compiled languages (e.g., Fortran, C++, Rust, Swift) that call into libraries such as MPI by linking to the appropriate symbol, just-in-time compiled languages must dynamically load the symbol at runtime for use. This requirement presents additional challenges for Enzyme because the compiler will not be able to recognize a call to `MPI_Send`, since all it will see is a call to a specific integer address. We remedy this within Enzyme.jl (Julia's bindings for Enzyme) by performing an additional processing step on the LLVM IR of the function to be differentiated by Enzyme. In that pass, Enzyme.jl will look for calls to an integer address and identify the name of the function being called by looking through Julia's symbol table. This then allows Enzyme to identify the function being called and generate the corresponding derivative code.[4]

Garbage Collection

Support for special garbage collection (GC) intrinsics must also be handled within Enzyme in order to differentiate parallel code that uses a library such as MPI.jl. Allocation of garbage-collected variables is straightforward and is handled by registering the garbage collection allocation function to Enzyme's allocation handler. Julia contains special macros `GC.preserve` which specify that a given variable must be preserved within the given scope, even if there are no uses known by Julia. This macro is necessary when making foreign function calls, which may not appear to Julia as a use of the memory. The macro is lowered into the function call `gc_preserve_begin` and `gc_preserve_end` runtime calls, which takes a list of variables to be preserved.In addition to preserving the original variables as specified, Enzyme must modify the call to also preserve the shadow of any variable being preserved, since they may also be modified in a way not known to Julia. Enzyme will also add a corresponding GC preservation in the reverse pass. This informs Julia's garbage collector to similarly preserve variables when computing the adjoint of that region. For

[4]This process is done for all foreign library calls and thus remedies similar issues that may occur when calling other foreign libraries.

example, if a `memcpy` had its arguments marked for preservation in the original code, the derivative of the `memcpy` (containing stores and loads to the shadow) would now also have its arguments marked for preservation. There may be instances when preservation is not needed, but this overly conservative approach is still correct and may be further optimized later.

8.6.4 Non-determinism

Non-determinism in parallel programs can arise from undefined behavior in the program. As an example in a parallel program one can create a *write-race* where several threads write distinct values to a single memory location at the same time. In Enzyme, like in other serial and parallel AD tools, differentiating a program with undefined behavior may result in a gradient calculation with undefined behavior. Take a primal function which reads from undefined memory, this will result in a gradient function which reads from a corresponding location of undefined shadow (derivative) memory.

A further source of non-determinism is that it is possible to have a program with an undefined parallel execution order that yields a deterministic result (through synchronization, reductions, atomics, or simply distinct memory locations per thread). In the case of synchronization, the reversal of the dependency DAG described in Section 8.4.1 and derivative accumulation described in Section 8.6.1 enable correct handling deterministic, but racy programs. In the context of a barrier or locked/atomic region in the forward pass, this will result in a program semantically equivalent to another barrier or atomic region in the reverse pass. For the locked/atomic region, the now serialized parallel tasks must be executed in the reverse order, which is performed by caching the actual execution order. For atomic instructions this is simpler as atomic instructions within LLVM return the previous value of memory – which is precisely what would require caching. Some simpler atomic instructions like add do neither require caching the execution order, nor an additional value.

8.7 Evaluation

To illustrate the composability of Enzyme's differentiation of parallel frameworks, we apply it to several distinct parallel variations of LULESH [204, 205], and miniBUDE [305]. In these results, *forward* denotes the time it takes to run the original program and *gradient* denotes the time it takes to both run the original program and compute the derivative of all the inputs, and *overhead* denotes the ratio of the gradient runtime to the forward runtime.

LULESH is a 5000-line hydrodynamics proxy application developed by Lawrence Livermore National Laboratory[5]. As an unstructured explicit shock hydrodynamics solver, it emulates the behavior of complex solvers by splitting the computational domain into volumetric elements on an unstructured mesh, hence mimicking the complex data movement characteristics of unstructured data structures. We designed our evaluation to test how effectively a single low-level implementation of parallelism within an automatic differentiation tool can enable a diverse set of parallelism models. We evaluate LULESH variations that use MPI, OpenMP, hybrid MPI+OpenMP, MPI.jl, and the RAJA portable parallel programming framework, written in C++ and Julia. To compare our performance against the automatic differentiation performance to the CoDiPack-differentiated LULESH of Hück et al. [178].

Mini-BUDE is a 200-line mini-app. developed by the University of Bristol emulating the main computational kernels of the heavily compute-bound molecular docking engine BUDE [259]. BUDE predicts the binding energy of two molecules using molecular mechanics, in order to evaluate the ability of test molecules to bind with a target molecule. Each potential pose of the molecules needs to be evaluated for its free energy, hence resulting in hundreds of thousands pose-evaluations for each molecule. Our evaluation on miniBUDE was designed to validate our automatic differentiation performance claims on LULESH on a second, distinct application, as well as testing Enzyme's ability to automatically differentiate Julia's shared-memory parallelism. We evaluate an OpenMP version in C++, and a Julia-version utilizing tasks.

[5]https://asc.llnl.gov/codes/proxy-apps/lulesh

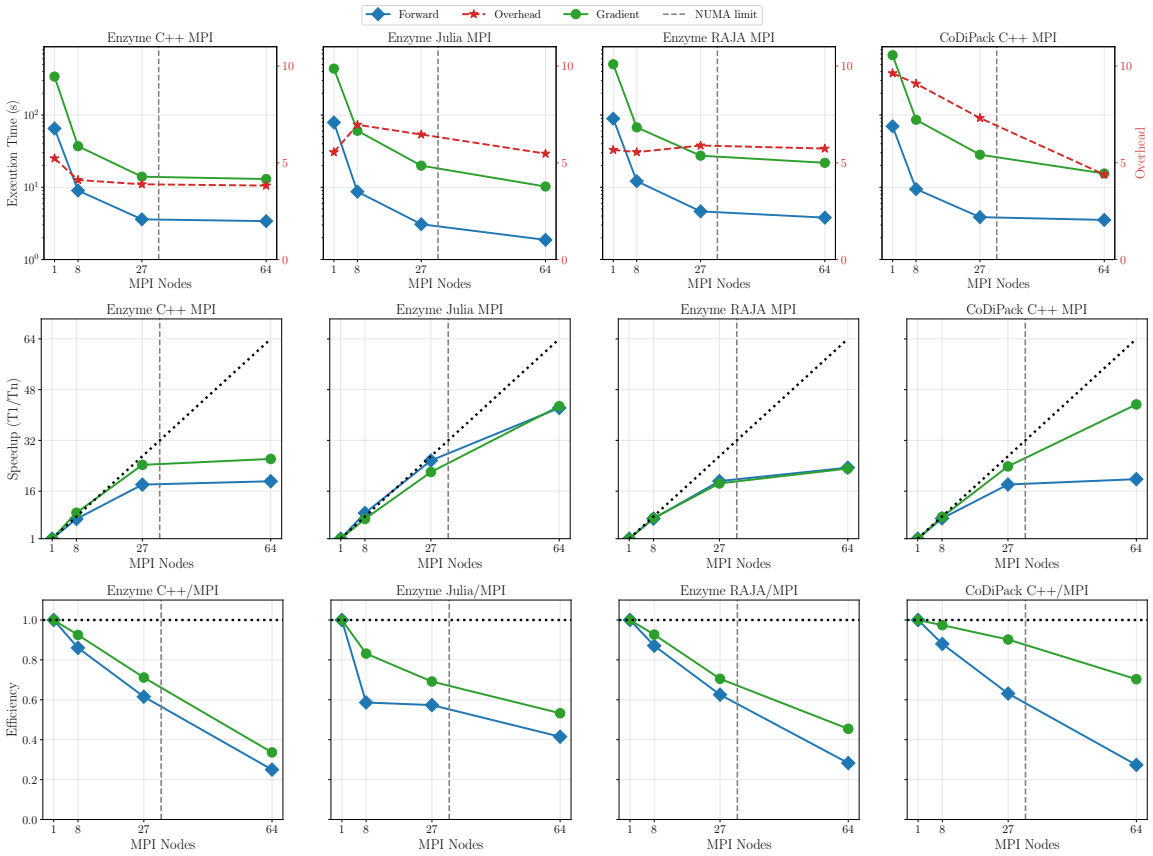


Figure 8-8: **Top Row:** Runtime for 10 iterations of the LULESH proxy-benchmark for the different implementations. The number of processors is increased, while the overall problem size stays fixed. We used the following task count-block size combinations: 1:192, 8:96, 27:64, and 64:48. **Middle Row:** Strong scaling behavior. **Bottom Row:** Weak scaling behavior. The number of processors is increased, while the per-processor problem size stays fixed. The block size used was 48.

8.7.1 Benchmark Implementation Details

C++ The C++ code is based on the official 2.0 release of LULESH[6]. We modified the code by creating a second shadow domain to store the derivative result and added an option to switch between primal and derivative computation. The only other change made was to pass member functions to the communication subroutines at compile time rather than as an array. This is not required for differentiation or correctness, and member functions are passed as an array in the RAJA version. For the OpenMP-version of miniBUDE[7] we created a shadow domain for the computational kernel and added an option to switch betweeen

[6] `https://github.com/LLNL/LULESH`
[7] `https://github.com/UoB-HPC/miniBUDE`

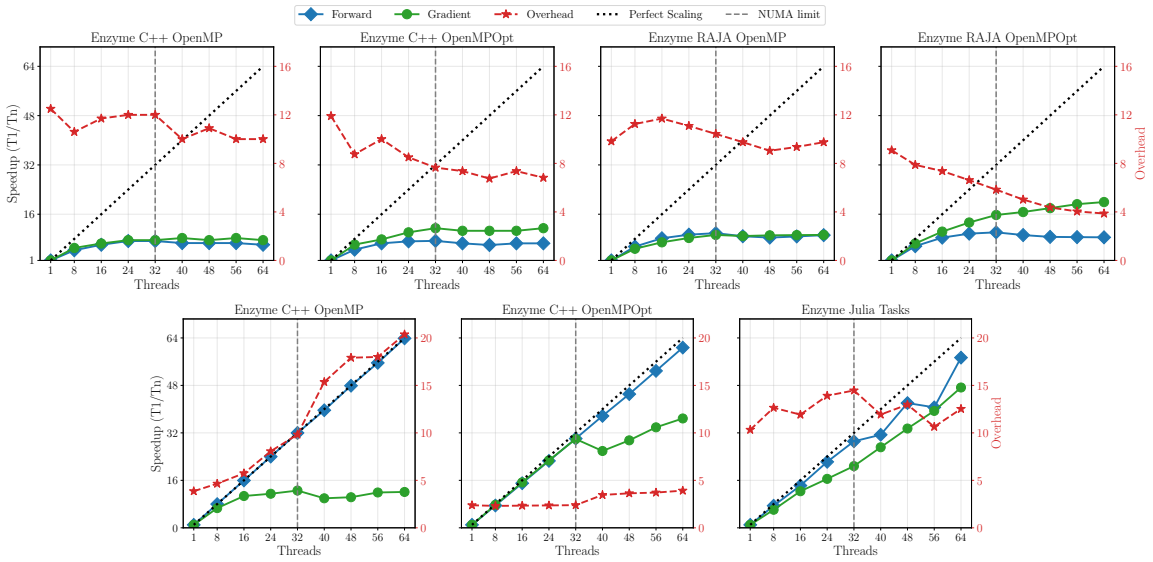


Figure 8-9: Thread parallelism strong scaling on the LULESH **(Top Row)** and BUDE **(Bottow Row)** proxy-benchmarks for the different evaluated implementations. The number of available processors is increased, while the overall problem size stays fixed. The block size used for LULESH was 96 and the default number of poses was used for BUDE.

primal and derivative computation.

RAJA The RAJA code is based off LULESH 2.0 from the official RAJA benchmark repository[8]. We added a second domain to store derivatives and added a flag to enable differentiation. The only changes we made were to use a standard allocator, rather than the custom allocator in the repository, and to free memory at the end of each iteration (calling `std::vector::shrink_to_fit` in addition to the `std::vector::clear`). The RAJA version was seemingly identical to the vanilla C++ version with the exception of using C++ `std::vector` instead of bare pointers, RAJA looping constructs instead of regular serial or OpenMP parallel for loops, and the runtime member function passing.

Julia Since no official or unofficial version of LULESH exists in Julia, we built a new version from scratch based on the official C++ version, and LLNL's unverified FORTRAN version of LULESH 1.0. We elected to port LULESH's MPI communication to Julia through the use of MPI.jl [60]. While the code attempts to remain faithful to the official C++ code, some differences include the use of garbage-collected arrays and minor changes to better match standard Julia design paradigms. The code's correctness was verified against LULESH's correctness checks of [205]. For the Julia-version of miniBUDE

[8] `https://github.com/LLNL/RAJAProxies`

we created a shadow domain for the computational kernel, no-inlined the core kernel, and added the option to switch between the primal and derivative computation for evaluation purposes. **CoDiPack** CoDiPack [332] is an existing operator overloading automatic differentiation tool for C++ with an extension to differentiate through MPI code. We use a version of LULESH modified by the CoDiPack authors [178] which rewrites variables and communication within the application to use CoDiPack-specific variants. We use CoDiPack LULESH as a performance baseline for existing state-of-the art tools. Like Enzyme, we use CoDiPack in reverse-mode for the tests.

Setup Experiments were run on an AWS c6i.metal instance with hyper-threading and Turbo Boost disabled, running Ubuntu 20.04 running on a dual-socket Intel Xeon Platinum 8375C CPU at 2.9 GHz with 32 cores each and 256 GB RAM.

All C++ codes are compiled using LLVM 14, and the Julia codes use Julia version 1.7.1. C++ MPI-codes are run with OpenMPI 4.0.3, and Julia's MPI code is run with MPICH 4.0.1. Experiments with OpenMP were benchmarked with LLVM 14's OpenMP implementation. We measured the time taken to execute the forward and differentiated versions of LULESH, and miniBUDE using different types of parallelism. For LULESH MPI strong scaling we report runtimes from 10 consecutive iterations from one run. For LULESH C++ and RAJA, MPI weak scaling, thread scaling, and MPI task and thread strong scaling we report times for 100 iterations. All remaining runs of LULESH experiments use 10 iterations. For the C++ and Julia versions of miniBUDE, we report times for the default number of iterations (100 and 8 iterations respectively). We studied the parallel scaling of the forward and differentiated code with increasing MPI rank and OpenMP thread counts.

8.7.2 Gradient verification

For realistic applications it is rarely feasible to perform tests for all relevant input values, nor is it generally feasible to compute the entire gradient or Jacobian matrix for applications with many active inputs or outputs using both the forward and reverse mode. It is therefore common practice to limit tests to certain inputs, and in the case of AD, to further limit tests

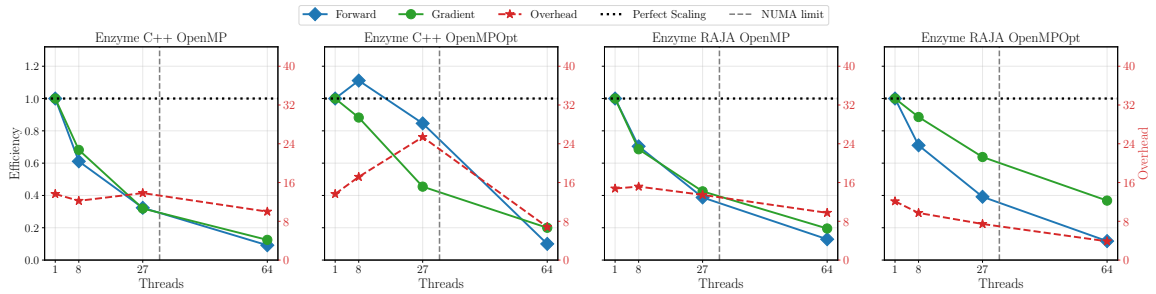


Figure 8-10: Thread parallelism weak scaling on the LULESH proxy-benchmarks for the different evaluated implementations. The number of available processors is increased, while the problem size per processor stays fixed. We used the following thread count-block size combinations: 1:24, 8:48, 27:72, and 64:96.

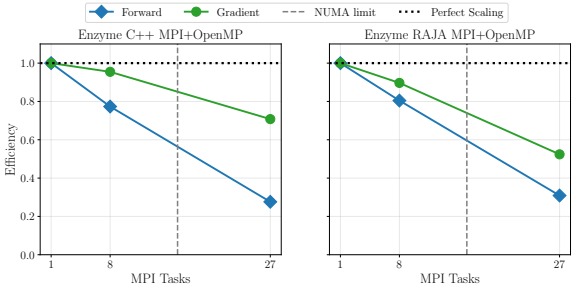


Figure 8-11: Efficiency of LULESH when running with 1 8, and 27 MPI ranks, 2 OpenMP threads, and a block size of 48.

to certain projections of the Jacobian matrix that can be efficiently computed with multiple approaches for comparison.

In order to verify the gradients computed by Enzyme, we selected a projection that can be efficiently computed using the reverse mode, while also being easy to approximate using finite differences. Using reverse mode, this projection can be computed by initializing all shadow variables to 1, and summing the computed shadow variables. When using finite differences, the same projection can be computed by perturbing all input variables at once by the same small value and summing the resulting derivatives of all output variables as approximated by the corresponding finite difference formula (we use central differences for the perturbations and derivative approximations). Both projections should yield the same scalar value, up to round-off and truncation errors. We note that this is similar to the "fast mode" gradient check [315] implemented in PyTorch.

8.8 Results

For each of the benchmarks (LULESH, miniBUDE), and parallel frameworks (OpenMP, MPI, OpenMP+MPI, RAJA, Julia Threads, MPI.jl) we evaluated the scalability of the original code, Enzyme-generated derivatives, and baseline CoDiPack-generated derivatives, if available.

LULESH requires the number of MPI ranks to be a perfect cube, MPI scaling tests hence ran on 1, 8, 27, and 64 ranks. The strong scaling of the benchmarks is shown in Figure 8-8 **(middle row)**. Here, we plot the speedup (time to run on one rank divided by time to run on N ranks) as the total amount of work is fixed while the number of ranks is increased. We find the scaling behavior of the derivative computation to be better than that of the primal in the C++, RAJA and CoDiPack cases. It matches the primal for the Julia implementation. It can be observed that the speedup of all cases reduces after 27 ranks, because of non-uniform memory access (NUMA). Each socket on the AWS instance can support 32 threads, beyond which threads must access non-local memory resulting in increased memory latency and consequently reduced speedup.

In both strong and weak scaling experiments, all versions of MPI-based LULESH differentiated with Enzyme (C++, Julia, RAJA), the differentiated code scales similarly to that of the original function. For the C++ and RAJA tests the decreased weak scaling of both the original LULESH benchmark and its derivatives can be explained by NUMA effects that occur when one needs to access data on more than one socket. We attribute the performance difference between LULESH.jl, and other LULESH implementations to the used MPI versions.

As the CoDiPack code is a modification of the C++ LULESH codebase with CoDiPack primitives, we can roughly compare the run times of Enzyme on the C++ LULESH against CoDiPack LULESH. While the CoDiPack gradient appears to scale better than Enzyme, this is because CoDiPack has a large gradient overhead (additional instructions required to compute the derivative of a single instruction in the reverse pass) for serial instructions unrelated to MPI. This causes the overall gradient overhead (considering all MPI and serial instructions) for CoDiPack to be quite high at 1 rank (see Figure 8-8 **(top row)**). The

scaling tests perform fewer serial instructions per rank at higher node, causing the total overhead to be composed of fewer serial instructions in proportion to an MPI call. As a result, CoDiPack's apparently improved scalability is an artifact of the higher serial differentiation overhead being called proportionally fewer times, rather than scalability of its MPI differentiation.

We also evaluated strong scaling performance of the OpenMP C++ and RAJA versions of LULESH in Figure 8-9 (**top row**) (CoDiPack cannot differentiate OpenMP LULESH, and LULESH.jl does not use threads). To evaluate the effectiveness of parallel optimization we ran versions of the OpenMP LULESH with and without OpenMPOpt enabled. We extended the OpenMPOpt in LLVM 14 to also handle hoisting loads out of parallel regions. We again find that the scaling behavior of the derivative matches that of the original function.

We find that LULESH OpenMP has a relatively flat gradient overhead. The overhead drops when OpenMPOpt is enabled due largely to the fact that OpenMPOpt moves a pointer indirection out of a loop, improving alias analysis and allowing Enzyme to avoid caching as much data.

We evaluated the strong scaling performance of the OpenMP C++, OpenMPOpt C++, and Julia Task versions of miniBUDE (OpenMPOpt does not apply to Julia tasks). With regular OpenMP, the gradient overhead worsens as threads increase but does not grow with OpenMPOpt. This is again due to parallel load hoisting moving data outside a loop, which in this test enables Enzyme to avoid having to cache any data at all, electing instead to recompute temporaries. There is a slight decrease in scalability for the gradient at 32 threads due to using both sockets at that point and needing to update data from both CPU's. Notably, the gradient continues to scale on multiple sockets after the initial performance loss. miniBUDE.jl's overhead is higher, but again scales well. The higher overhead is because Julia arrays have an extra level of pointer indirection that causes alias analysis to conclude that several values need be cached. The amount of data being cached is still moderate due to Enzyme's ability to rematerialize temporary allocations.

The weak scaling performance of the OpenMP C++ and RAJA versions of LULESH can be found in Figure 8-10. We again find that scaling of the LULESH OpenMP and Open-

MPOpt gradient matches that of the primal. The C++ OpenMPOpt displays anomalous behavior because OpenMPOpt did not optimize for a single thread as effectively. Finally, Figure 8-11 shows the scaling behavior of LULESH using both MPI task and OpenMP thread parallelism.

Overall, for all types of parallelism, the differentiated code scales similarly to the forward code. Since Enzyme can cache data without contention or a shared data structure, only gradient accumulation may add contention to the program. The use of analyses which detect what pointers and registers are thread local and thus can be accumulated serially helps preserve the parallel scaling properties.

8.9 Conclusion

We have introduced a composable and generic LLVM-based mechanism to differentiate a variety of parallel programming models. In addition to simplifying the ability to handle high-level parallelism constructs, this marks the first time that an automatic differentiation tool can handle multiple parallelism models and multiple languages with a single implementation. We showcase the potential of this approach on the proxy apps LULESH, and miniBUDE, demonstrating Enzyme's practical use in real-world scientific simulation codes with nontrivial parallelization patterns. The overhead of the differentiated code is well inside the expected runtime of overheads of other state-of-the-art differentiation tools, is even comparable to the overhead of sequential programs[278], and observes the same scaling behavior of the differentiated code when compared with the original code. At the same time Enzyme does not require its users to utilize custom Adjoint-MPI libraries, and rewrite their application, making its AD of parallel programs much more accessible for users.

Chapter 9

High-Performance GPU-to-CPU Transpilation and Optimization via High-Level Parallel Constructs

9.1 Introduction

Despite x86 CPUs and NVidia GPUs remaining primary platforms for computation, customized and emerging architectures play an important role in the computing landscape. A custom version of an ARM CPU, A64FX, is even used in one of the top supercomputers Fugaku [339] where its high-bandwidth memory is expected to compete with that of GPUs. However, these architectures are often overlooked by efficiency-oriented frameworks and libraries. For example, PyTorch [297] targeting Intel's oneDNN [188] backend expectedly underperforms on ARM due to architecture differences and even Fujitsu's customized oneDNN [125] does not yield competitive performance on some kernels. Such situations call for performance portability.

Many non-library approaches for performance portability have been proposed and include language extensions (e.g., OpenCL [102], OpenACC [169]), parallel programming frameworks (e.g., Kokkos [63]), domain-specific languages (e.g., SPIRAL [314], Halide [319] or Tensor Comprehensions [397]). All of these approaches still require legacy applications

to ported, and sometimes entirely rewritten, due to differences in the language, or the underlying programming model.

We explore an alternative approach based on a fully automated compiler that takes code in one programming model (CUDA) and produces a binary targeting another one (CPU threads). While GPU-to-CPU translation has been explored in the past [372, 92, 160], it was rarely able to produce efficient code. In fact, optimizations for CPUs and even generic compiler transforms, such as common sub-expression elimination or loop-invariant code motion, are hindered by the lack of analyzable representations of parallel constructs inside the compiler [274]. As representations of parallelism within a mainstream compiler have only recently begun to be explored [344, 222, 367, 96, 94], existing transformations are limited and tend to apply to simple CPU codes only.

We propose a compiler model for most common GPU constructs: multi-level parallelism, level-wide synchronization, and level-local memory. In contrast to source and AST-level approaches, which operate before the optimization pipeline, and existing compiler approaches, which model synchronization as a black-box optimization barrier, we model synchronization from memory semantics. This allows synchronization-based code to interoperate with existing optimizations and enables novel parallel-specific optimizations.

Our model is implemented using MLIR [229] and LLVM [230] and leverages MLIR's nested-module approach for GPU [154]. We extended the Polygeist [270] C/C++ frontend to support CUDA and to produce MLIR which preserves high-level parallel structure. Our prototype compiler is capable of compiling PyTorch CUDA kernels, as well as other compute-intensive benchmarks, to any CPU architecture supported by LLVM. In addition to transformations accounting for the differences in the execution model, we also exploit parallelism on the CPU via OpenMP. Finally, our MocCUDA PyTorch integration allows us to compile and execute CUDA kernels in absence of a GPU while substituting unsupported calls.

We evaluate our compiler on Rodinia CUDA benchmarks [68] and PyTorch CUDA kernels. When targeting a commodity CPU, our OpenMP-accelerated CUDA code yields comparable performance with the reference OpenMP implementations from the Rodinia suite, as well as improved scalability. When using our framework to run PyTorch on the

CPU-only Fugaku Supercomputer, we achieve roughly twice the images processed per second of a Resnet-50 [165] training run compared to existing PyTorch CPU backends.

Overall, this chapter makes the following contributions:

- A common high-level and platform-agnostic representation of SIMT-style parallelism backed by a semantic definition of barrier synchronization that ensures correctness through memory semantics, and thus transparent application of existing optimizations.

- Novel parallel-specific optimizations which can exploit our high-level parallel semantics to optimize programs.

- An extension to the Polygeist C/C++ MLIR frontend capable of directly mapping GPU and CPU parallel constructs into our high-level parallelism primitives.

- An end-to-end transpilation[1] of CUDA to CPU for a subset of the Rodinia [68] benchmark suite and the internal CUDA kernels in PyTorch [297] necessary to run Resnet-50 on the CPU-only Fugaku supercomputer.

9.2 Background

Mainstream compilers like Clang and GCC lack a unified high-level representation of parallelism. Compiling parallel constructs in frameworks like CUDA, OpenMP, or SYCL, forces the body of a parallel region to exist within a separate (closure) function which is invoked by a parallel runtime. Concepts such as thread index or synchronization are then represented separately, often through opaque intrinsic calls. As the compiler historically lacked information about parallelism and effects of the involved runtimes, any parallel construct also inadvertently acted as a barrier to optimization. While there have been attempts [96, 94, 381, 274, 343, 222, 367] in recent years to improve representations for CPU parallel constructs, accelerator programming comes with additional challenges. The unique

[1]We use the term *transpilation* to refer to taking a program in one programming model and emitting code for another, similar to source-to-source CUDA-to-C transpilers though now on IR. This procedure also *cross-compiles* the code. which refers to emitting non-native instructions.

programming model and complex memory hierarchy have left high-level representations of GPU parallelism within mainstream compilers under-explored.

```
__device__ float sum(float* data, int n) { ... }
__global__
void normalize(float *out, float* in, int n) {
  int tid = blockIdx.x + blockDim.x * threadIdx.x;
  // Optimization: Compute the sum once per block.
  // __shared__ int val;
  // if (threadIdx.x == 0) val = sum(in, n);
  // __syncthreads;
  float val = sum(in, n);
  if (tid < n)
    out[tid] = in[tid] / val;
}
void launch(int *d_out, int* d_in, int n) {
  normalize<<<(n+31)/32, 32>>>(d_out, d_in, n);
}
```

Figure 9-1: A sample CUDA program `normalize`, which normalizes a vector and the CPU function `launch` launching the kernel. Each GPU threads calls `sum`, resulting in $O(N^2)$ work. Using shared memory (commented) reduces the work to $O(N^2/B)$ at extra resource cost. Computing `sum` before the kernel reduces work to $O(N)$.

9.2.1 GPU Compilation

Consider the CUDA program in Figure 9-1, which normalizes a vector. When compiled using Clang, the GPU program is a separate compilation unit. This prevents any optimization between the GPU kernel and the CPU calling code. In the case of Figure 9-1, the total work of the program in a traditional compiler is $O(N^2)$, due to the $O(N)$ call to `sum` being performed for each thread. However, if the call to `sum` is performed only once prior to the kernel call, e.g., by performing loop-invariant code motion (LICM), the work would reduce to $O(N)$. A less effective variant of this optimization could reduce the work to $O(\frac{N^2}{B})$ through the use of shared memory. MLIR provides a nested-module representation for GPU programs that supports host/device code motion [154], but parallel code motion has not been implemented. In GPU to CPU code motion, LICM out of a parallel loop is always legal as any former device memory is also available on the host.

```
// Kernel launch is available within the calling
// function, enabling optimizations across the
// GPU/CPU boundary.
func @launch(%h_out : memref<?xf32>,
             %h_in  : memref<?xf32>, %n : i64) {
  // Parallel for across all blocks in a grid.
  parallel.for (%gx, %gy, %gz) = (0, 0, 0)
            to (grid.x, grid.y, grid.z) {
    // Shared memory = stack allocation in a block.
    %shared_val = memref.alloca : memref<f32>
    // Parallel for across all threads in a block.
    parallel.for (%tx, %ty, %tz) = (0, 0, 0)
              to (blk.x, blk.y, blk.z) {
      // Control-flow is directly preserved.
      if %tx == 0 {
        %sum = func.call @sum(%d_in, %n)
        memref.store %sum, %shared_val[] : memref<f32>
      }
      // Syncronization via explicit operation.
      polygeist.barrier(%tx, %ty, %tz)
      %tid = %gx + grid.x * %tx
      if %tid < %n {
        %res = ...
        store %res, %d_out[%tid] : memref<?xf32>
      }
    }
  }
}
```

Figure 9-2: Polygeist/MLIR equivalent of `launch`/`normalize` code from Figure 9-1. The kernel call is available directly in the host code which calls it. The parallelism is explicit with parallel for loops across the blocks and threads. Shared memory is placed within the block parallel for, allowing access from any thread in the same block, but not a different block.

9.2.2 MLIR Infrastructure

MLIR is a recent compiler infrastructure designed for reuse and extensibility [229]. Rather than providing a predefined set of instructions and types, MLIR operates on collections of *dialects* containing interoperable user-defined operations, attributes and types. Operations are a generalization of IR instructions that can be arbitrarily complex, in particular, contain regions with more IR thus creating a nested representation. Operations define and use values that obey single static assignment (SSA) [88]. For example, MLIR dialects may model entire instruction sets such as NVVM (virtual IR for NVidia GPUs), other IRs such as LLVM IR [230], control flow such as loops, parallel programming models such as OpenMP and OpenACC, machine learning graphs, etc.

MLIR supports GPU thanks to the eponymous dialect, which defines the high-level SIMT programming model, host/device communication, and a set of platform-specific dialects: NVVM (CUDA), ROCDL (ROCm) and SPIR-V. MLIR's approach to GPU programming benefits from a *unified* code representation. Since an MLIR module may contain other modules, the "host" translation unit may embed the "device" translation unit as IR rather than file reference or binary blob. This approach provides host/device optimization opportunities unavailable to other compilers, in particular to move code between host and device [154].

9.2.3 Polygeist

Polygeist is a C/ C++ frontend for MLIR based on Clang [270]. It is capable of translating a broad range of C++ programs into a mix of MLIR dialects that preserve elements of the high-level structure of the program. Specifically, Polygeist preserves structured control flow (loops and conditionals) as MLIR SCF dialect operations and simplifies analyses by preserving multi-dimensional array constructs whenever possible by relying on the MLIR's multi-dimensional memory reference (memref) type. Finally, Poylgeist is able to identify parts of the program suitable for polyhedral optimization [113] and represent them using the Affine dialect.

```
__global__ f() {
  codeA();
  barrier();
  codeB();
}
```

```
__global__ f() {     // 0<=t.x< blockDim.x
  A[threadIdx.x] = ...; //  W A[i]: i==t.x
  barrier();            // RW A[i]: i!=t.x
  ... = A[threadIdx.x]; // R  A[i]: i==t.x
}
```

Figure 9-3: **Left:** A program containing a barrier between two arbitrary instructions. **Right:** Barrier semantics can be refined memory addresses accessed by operations above/-below it in all threads *except* the current one.

9.3 Approach

We extended the Polygeist compiler [270] to directly emit parallel MLIR from CUDA. This leverages the unified CPU/GPU representation to allow the optimizer to understand host/device execution, and to enable optimization across kernel boundary. The use of existing MLIR's first-class parallel constructs (`scf.parallel`, `affine.parallel`) enables us to target existing CPU and GPU backends. Finally, MLIR's extensible operation set allows us to define custom instructions, with relevant properties and custom optimizations.

We define the representation of a GPU kernel launch as follows (illustrated in Figure 9-2):

- A 3D parallel for-loop over all blocks in the grid.
- A stack allocation for any shared memory, scoped to be unique per block.
- A 3D parallel for-loop over all threads in a block.
- A custom Polygeist barrier operation that provides equivalent semantics to a CUDA synchronization.

This procedure enables us to represent any GPU program in a form that preserves the desired semantics. It is fully understood by the compiler and is thus amenable to compiler optimization. Moreover, by representing GPU programs with general parallelism, allocation, and synchronization constructs, we are not only able to optimize the original program, but also retarget it for a different architecture.

9.3.1 Barrier Semantics

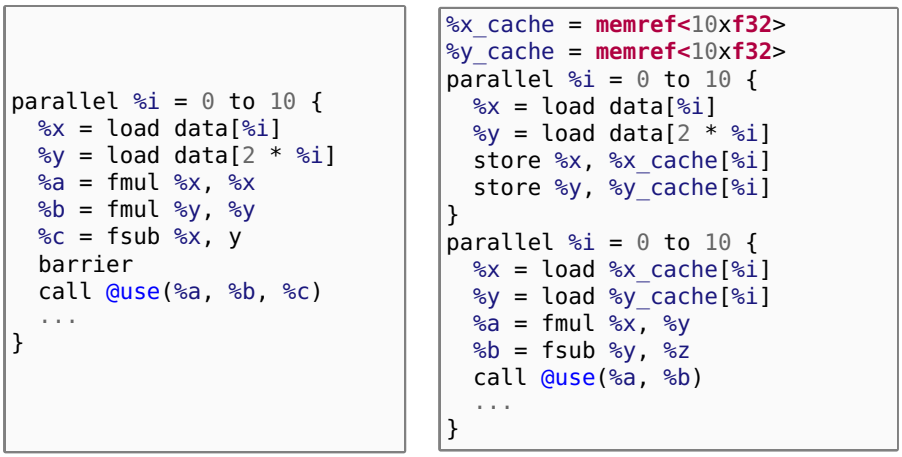
A CUDA `__syncthreads` function guarantees that all threads in a block have finished executing all instructions prior to the function call, before any threads executes any instruction after the call. Traditionally, compilers represent such functions as opaque optimization barriers that could touch all memory, and forbid any transformation involving them.

In our system, we chose to represent thread-level synchronization through a new `polygeist.barrier` operation. Unlike other approaches, `polygeist.barrier` (hence referred to as simply `barrier`) aims to only prevent transformations that would change externally visible behavior. Rather than disallowing any code motion across a `barrier`, we can successfully achieve the desired semantics by defining `barrier` to have specific memory properties, represented as a collection of memory locations (including unknown), and memory effect type (read, write, allocate, free), as is standard within MLIR. Consider the simple program in Figure 9-3(left). The impact of the synchronization can only be observed if `codeA` and `codeB` access the same memory. Moreover, if both only read the same memory location, the synchronization is also unnecessary. We can enumerate the remaining cases: (1) `codeA` writes, `codeB` loads; (2) `codeA` loads, `codeB` writes; (3) `codeA` writes, `codeB` writes.

The barrier having the write behavior of `codeA` would ensure correctness of (1): the load in `codeB` could not be hoisted above the barrier, as it would appear to read a different value. Symmetrically, the barrier having the write behavior of `codeB` ensures the correctness of (2). Thus, the union of the writing behaviors of `codeA` and `codeB` is sufficient to prevent illegal movement of loads across the barrier.

However, this does not prevent writes from being moved. For example, `codeB` could be duplicated above the barrier in (3), and it would appear to have the same final memory state since the extraneous write before the barrier would never be read. Thus, we also define the barrier to have the reading behavior of `codeA` and `codeB`.

This model can be extended to include memory effects of all operations in the parallel loop which may have been executed before, or after, a given barrier. On a control flow graph with explicit branches, this requires exploring the operations within predecessors or successors, respectively. However, operating on MLIR's structured control flow level, with



```
parallel %i = 0 to 10 {
  %x = load data[%i]
  %y = load data[2 * %i]
  %a = fmul %x, %x
  %b = fmul %y, %y
  %c = fsub %x, y
  barrier
  call @use(%a, %b, %c)
  ...
}
```

```
%x_cache = memref<10xf32>
%y_cache = memref<10xf32>
parallel %i = 0 to 10 {
  %x = load data[%i]
  %y = load data[2 * %i]
  store %x, %x_cache[%i]
  store %y, %y_cache[%i]
}
parallel %i = 0 to 10 {
  %x = load %x_cache[%i]
  %y = load %y_cache[%i]
  %a = fmul %x, %y
  %b = fsub %y, %z
  call @use(%a, %b)
  ...
}
```

Figure 9-4: Parallel loop splitting around a barrier: the code above the barrier is placed in a separate parallel "for" loop from the code following the barrier. This transformation eliminates the barrier, while preserving the semantics. The min-cut algorithm stores `%x` and `%y`, which are then used to recompute `%a`, `%b`, and `%c` in the second loop.

explicit operations for loops and conditionals, simplifies the analysis. Furthermore, if more than one barrier is present in the same block, it is unnecessary to look past it.

Given a sufficiently expressive side effect model, the memory semantics of the barrier can be further expanded. While barriers enforce ordering reads/writes to the same location from *different* threads, the natural execution order is sufficient within one thread. Therefore, barriers need not capture the memory effects of operations where the address is an *injective function* of the thread identifier. We implement the refinement for *affine* forms of access expressions leveraging the polyhedral framework in MLIR/Polygiest. For each memory access, we define an integer relation between a set of possible thread id values and the set of accessed array subscripts, $\mathcal{R} : T \to A$. We then compose direct and inverse relations for relevant operations to obtain a relation between thread indices accessing the same subscript, $\mathcal{D} = \mathcal{R}^{-1} \circ \mathcal{R} : T \to T'$. Finally, we subtract the identity relation $\mathcal{D} \setminus \mathcal{I} : T \to T'$. If non-empty, $\mathcal{D} \neq \emptyset$, different threads may access the same address and the barrier is required. Given a non-affine access or non-static control flow, we conservatively assume an access of the entire array dimension. In practice, this is rarely necessary on GPU code, whose loops typically have parametric/static bounds. Aliasing guarantees must be checked when more than one base address is involved.

Consider the code in Figure 9-3(right). Since the sets of accessed addresses do not

```
parallel for %id=0 to N {
  for %j = 5 to 0 {
    if (%id < 2^%j)
      A[%id] +=  A[%id + 2^%j]
    barrier
  }
}
```

```
for %j = 5 to 0 {
  parallel for %id=0 to N {
    if (%id < 2^%j)
      A[%id]+=A[%id + 2^%j]
    barrier
  }
}
```

Figure 9-5: ***Left:*** A shared memory addition, which consists of a kernel call which contains for loop with a barrier inside. ***Right:*** Same but with the barrier directly in the parallel loop after a parallel/serial loop interchange.

overlap, $\mathcal{A}_o \cap \mathcal{A}_b = \varnothing$, code motion across the barrier is allowed. In contrast, if the load or store to `A` were offset by 1, the barrier would be necessary as the data loaded after the barrier would be stored by a different thread.

9.3.2 Barrier Lowering

To enable GPU programs to run on a CPU, we must efficiently emulate the synchronization behavior of GPU programs. Whereas the memory semantics in Section 9.3.1 enable us to preserve the correctness of barriers during optimization, this section discusses how to implement the barrier on a CPU.

CPU architectures have no notion of thread blocks, nor the barrier instruction which waits on this conceptual grouping of threads. Instead, we use regular CPU threads and work sharing to distribute the thread-block loop iterations across them. Conceptually, this differs from the GPU execution model in which threads execute one iteration each. Work sharing requires each thread to execute multiple iterations sequentially, making it impossible to synchronize in the middle of iterations, but only at the end of the loop.

To address this, we developed a new barrier elimination technique for our MLIR representation. Our approach is an extension of loop fission (see Section 9.7) combining two transformations: *parallel loop splitting* and *interchange*.

Parallel Loop Splitting

Suppose a barrier has the kernel function (or, in our representation, parallel `for` loop) as its direct parent. It can be eliminated by splitting the loop around the barrier into two

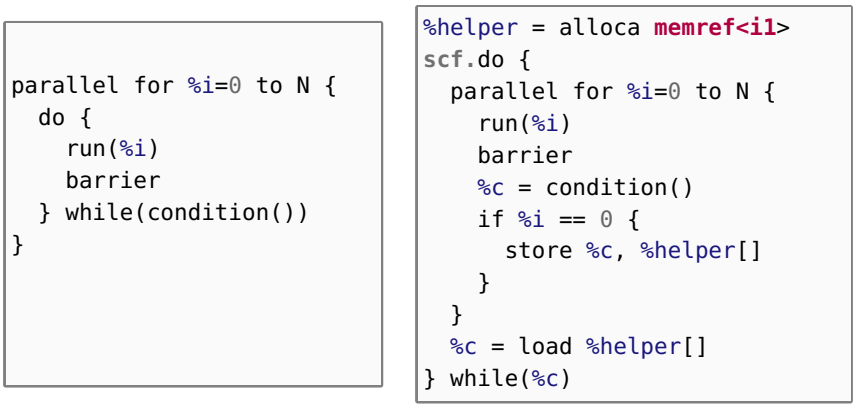


Figure 9-6: Parallel interchange around a `while` loop. As the `condition()` function call must be executed on each thread to preserve correctness, a helper variable is used which holds the value of the call on the first thread.

parallel `for` loops that run the code before and after the barrier, respectively. If the code before the barrier created SSA values that were used after it, these must be either stored or recomputed in the second parallel loop. We use the technique similar to one in [272] to determine the minimum amount of data that needs to be stored. Specifically, we create a graph of all SSA values. We then mark each value definition that cannot be recomputed (e.g. loads from overwritten memory) before the `barrier` as source, and values used after the `barrier` as sinks. We derive the minimum amount of data needing to be stored by performing a minimum branch cut on this graph.

Parallel Loop Interchange

Not all barrier operations have a parallel `for` as their immediate parent, some may be nested in other control flow operations. We created a model that specifies what instructions may run in parallel. With the sole exception of `barrier`, our representation does not require any specific ordering or concurrency to the program. Therefore it is legal (though potentially a reduction in parallelism) to add additional barriers. We can use this property to implement barrier lowering for control flow.

Consider a control-flow construct `C` containing a barrier and nested in a parallel `for`. Adding barriers immediately around `C` will result in parallel loop splitting directly above and below `C`. As a result, the operations above and below `C` will be separated into their own

parallel `for` and `C` will be the sole operation in the middle loop. We can then apply one of the following techniques to interchange `C` with the parallel `for`, thus making the `barrier`'s parent a parallel `for`.

Consider the case of a serial `for` loop containing a barrier, Figure 9-5. This pattern is common in GPU code, e.g., to implement a reduction across threads [162]. As `barrier` must wait for all threads, each thread must execute the same number of `barrier`s. Therefore, the number of iterations of the inner loop is the same for all threads, allowing for loop interchange.

While an `if` statement can be considered a loop with zero or one iteration, directly interchanging it with the surrounding parallel `for` when necessary is more efficient.

Whereas `for` loops in MLIR have a fixed trip count, `while` loops support dynamic exit conditions, like in Figure 9-6. Since correctness requires executing `condition()` in every thread, a direct interchange would not be legal. However, GPU synchronization semantics require the trip count to be the same in all threads. Therefore, one can still perform an interchange using a helper variable to store the result of the condition.

This illustrates one of the advantages of building off of MLIR/Polygeist. By preserving high level program structures, we can use more efficient patterns to remove barriers.

9.4 Parallel Optimization

The high-level representation of both parallelism and GPU programs provided by Polygeist/MLIR enables a variety of optimizations. These include general optimizations that would apply to any parallel program as well as specific optimizations in the context of GPU to CPU conversion.

9.4.1 Barrier Elimination & Motion

As GPU-style barriers have to be specially transformed to support CPU architectures, eliminating or simplifying any barriers can have dramatic effects. Moreover, even when running GPU code on the GPU, barrier elimination is highly useful as any synchronization reduces parallelism. Much of the infrastructure for barrier elimination/simplification comes directly

```
__global__ void bpnn_layerforward(...) {
  __shared__ float node[HEIGHT];
  __shared__ float weights[HEIGHT][WIDTH];
  if ( tx == 0 ) node[ty] = input[index_in] ;
  // Unnecessary Barrier #1
  __syncthreads();
  // Unnecessary Store #1
  weights[ty][tx] = hidden[index];
  __syncthreads();

  // Unnecessary Load #1
  weights[ty][tx] = weights[ty][tx] * node[ty];
  __syncthreads();

  for ( int i = 1 ; i <= log2(HEIGHT) ; i++){
    if( ty % pow(2, i) == 0 )
      weights[ty][tx] += weights[ty+pow(2, i-1)][tx];
    __syncthreads();
  }

  hidden[index] = weights[ty][tx];
  // Unnecessary Barrier #2
  __syncthreads();

  if ( tx == 0 ) out[by * hid + ty] = weights[tx][ty];
}
```

Figure 9-7: An example CUDA kernel from the Rodinia backprop test that contains unnecessary synchronization and unnecessary use of shared memory.

from its memory behavior defined in Section 9.3.1. Let $M_B^{\uparrow}$ ($M_B^{\downarrow}$) be the union of memory effects before (after) a barrier `B` until the edge of the parallel region. Let $M_B^{\bullet\dagger}$ be the subset of $M_B^{\bullet}$ with effects until the first barrier rather than region edge. Given a barrier `B`, if there are no memory effects to the same location across the barrier other than a read-after-read (RAR), i.e. $M_B^{\uparrow\dagger} \cap M_B^{\downarrow} = \varnothing$, `B` has its behavior subsumed by the previous barrier. Symmetrically $M_B^{\uparrow} \cap M_B^{\downarrow\dagger} = \varnothing$ means the barrier is subsumed by the following one. A specific case of a removable barrier is one that has no memory effects at all.

For example, consider the code in Figure 9-7, which comes from the `backprop` Rodinia benchmark [68]. The first and last `__syncthreads` instructions are unnecessary. This can be proven from our memory-based barrier elimination algorithm above as follows. For the first barrier, $M^{\uparrow}$ (going all the way to the start) contains only a write to `node` and a read from `input`. $M^{\downarrow\dagger}$ (going to the second `__syncthreads`) contains a write to `weights` and a read from `hidden`. None of these conflict if, given the calling context, the pointers are known not to alias. Thus, it is safe to eliminate the barrier.

The same memory analysis can also be applied to perform barrier motion. One simply needs to place a fictitious barrier at the intended location and check if the previous memory analysis would deduce that the current barrier is unnecessary, thereby permitting barrier motion.

9.4.2 Memory-to-register promotion across barriers

One of the goals of defining `barrier`'s semantics from its memory behavior is to enable memory optimizations to operate correctly and effectively in code that contains barriers. As described in Section 9.3.1, barriers have the memory behavior of the code above and below them with the notable exception of an access from the current thread. This hole is important as it enables memory-to-register promotion (mem2reg) to operate on thread-local memory such as local variables. This optimization can replace slow memory reads with fast registers. For example, consider again the code in Figure 9-7. Consider the load and store to `weights[ty][tx]` labeled "Unnecessary Store #1" and "Unnecessary Load #1", and the sync in between the two. The only value that can be loaded at that point is the

same value which was stored earlier, a register containing the value loaded from `hidden`. As that same location is overwritten before anyone else could read from `weights`, the first store also can be safely eliminated once the load is removed. During mem2reg, Polygeist can derive this forwarding property, since the hole in the memory properties described in Section 9.3.1 allows it to deduce that the barrier operation does not overwrite the store for the current thread. As a result, traditional load and store forwarding correctly operates on the barrier code.

9.4.3 Parallel loop-invariant code motion

The traditional loop-invariant code motion optimization aims to move an instruction `I` outside serial "for" loops, reducing the number of times `I` is executed. If `I` may access memory, or has other side effects, in addition to checking that the operands of `I` are themselves loop invariant, the compiler must check that no other code within the "for" loop conflicts with the memory access performed by `I`.

On present compilers, while it is possible to apply loop-invariant code motion to serial for loops within GPU kernels, it is not possible to apply loop-invariant code motion to hoist instructions outside of a kernel call. This is in part due to the fact that GPU kernels are kept in a separate module from the CPU code which calls them, as well as a lack of understanding of parallelism (see Figure 9-1).

Counter-intuitively, with the right semantics we can apply loop-invariant code motion to parallel for loops even if we would not be able to apply it to an equivalent serial loop. We will rely on the fact that semantics of our program permits us to arbitrarily interleave iterations of a parallel "for" loop as long as we maintain the orderings required by barriers. As such, it is legal, though not necessarily fast, to run the program in lock-step. In other words, if a parallel for loop had 10 instructions, each thread can execute instruction 1 before any thread executed instruction 2, and so on. As a consequence, it is now legal to hoist an instruction so long as its operands are invariant and no *prior* instruction in the parallel for loop conflicts with `I`.

```
omp.parallel {
  omp.wsloop %i= 1 to 10 {
    codeA(%i)
  }
}
omp.parallel {
  omp.wsloop %i= 1 to 10 {
    codeA(%i)
  }
}
```

```
omp.parallel {
  omp.wsloop %i=1 to 10 {
    codeA(%i)
  }
  omp.barrier
  omp.wsloop %i=1 to 10 {
    codeA(%i)
  }
}
```

Figure 9-8: Example of OpenMP parallel region fusion. Fuse two adjacent OpenMP parallel regions by inserting a barrier to allow the threads to be initialized once instead of twice.

```
for (i=0; i<N; i++) {
  #pragma omp parallel for
  for (j=0; j<10; j++) {
    body(i, j);
  }
}
```

```
#pragma omp parallel
for (i=0; i<N; i++) {
  #pragma omp for
  for (j=0; j<10; j++) {
    body(i, j);
  }
  #pragma omp barrier
}
```

Figure 9-9: Example of OpenMP parallel region hoisting. This can be seen as an extension of parallel region fusion across “regions” corresponding to each iteration of the outer loop.

9.4.4 Block Parallelism Optimizations

OpenMP is our primary target for parallel execution on the CPU. It implements parallel "for" loops as two constructs. First, the loop is outlined into a function which is called once per thread, representing OpenMP’s "parallel" construct. Then, within the outlined function, the iteration space is distributed across threads, representing OpenMP’s "worksharing loop" construct. OpenMP also has a "barrier" construct, but with semantics *different* than that of a GPU barrier.

When multiple parallel loops are executed in a row, e.g., following the barrier lowering from Section 9.3.2, the overhead of thread management can be reduced by fusing adjacent OpenMP "parallel" constructs [95] *without* fusing the worksharing loops (see Figure 9-8), thus not undoing the barrier lowering. This can be extended to moving the OpenMP parallel region outside the surrounding “for” in Figure 9-9, initializing threads once rather than N times. Applying these to control flow constructs enables all of the “for” loops generated by performing parallel loop fission on a block to have their OpenMP “parallel” (but not work

sharing loops) fused.

As GPU programs tend to be written with high parallelism in mind, the parallelism provided by the different blocks may already saturate the number of available cores alone. If there is no use of shared memory, the block and thread parallelism can be collapsed into a single OpenMP parallel `for`, which will evenly divide the total iteration space in a single parallel region. However, if there is shared memory, our tool will generate nested parallel regions to represent the shared memory allocation. In this case, the additional overhead from the nested OpenMP parallel regions may outweigh the potential added parallelism. In addition, parallelizing the inner loops may lead to adverse memory effects such as false sharing, further penalizing performance [430, 396]. As such, we also support an optimization for serializing any nested OpenMP parallel regions. Performing such serialization may leverage memory locality to improve performance.

9.5 MocCUDA: Integration into PyTorch

One of our goals is to support execution of originally GPU codes on a CPU-only supercomputer such as Fugaku [339]. We focus on PyTorch [297] that has not been ported to the A64FX architecture and therefore uses naive fallback CPU kernels. Observing that CPUs with high-bandwidth memory are likely to benefit from GPU-style optimization, we implement MocCUDA, a mock GPU backend for PyTorch that redirects the calls to CUDA runtime and libraries to our implementations or A64FX-specific math libraries [125]. We collect statistics of library calls and may optionally substitute them with CPU versions transpiled by Polygeist.

9.6 Evaluation

We demonstrate the advantages and applicability of our approach on two well-known GPU benchmark suites: a subset of the GPU Rodinia benchmark suite [68] and a PyTorch implementation of a Resnet-50 neural network. These benchmarks were chosen to 1) provide a rough performance comparison of our GPU to CPU compilation on a benchmark

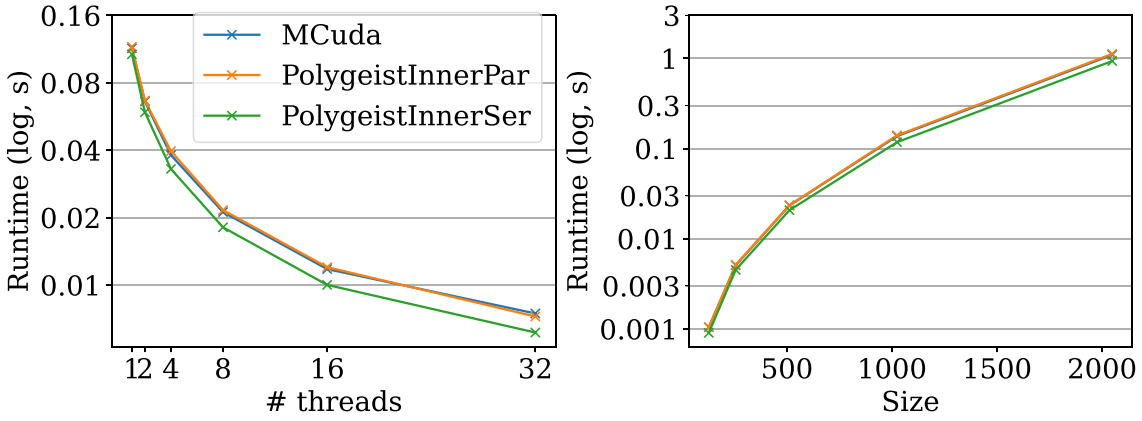


Figure 9-10: PolygeistInnerPar performs similarly to MCUDA; PolygeistInnerSer outperforms MCUDA. PolygeistInnerSer disables inner loop parallelization similaly to MCUDA, whereas PolygeistInnerPar keeps both the blocks and threads parallel. Left: Average runtime as a function of thread count (averaging over matrix sizes). Right: Average runtime as a function of matrix size (averaging over thread counts).

suite (Rodinia) that has hand-coded CPU versions and 2) demonstrate a successful end-to-end integration of our system into a useful and real application (PyTorch Resnet-50) on Supercomputer Fugaku, which does not have any GPUs. Additionally, we compare the performance of our approach to the existing MCUDA [372] tool on a CUDA matrix multiplication.

For Rodinia, we compare our translated CUDA to CPU code against OpenMP versions of the benchmarks, where they exist, as well as a run on a GPU. For the PyTorch Resnet-50, we compare against the "native" and oneDNN backends.

Polygeist[2] was compiled using LLVM 15 (git `00a1258`). For the PyTorch Resnet-50, we compile Pytorch v1.4.0 using NVidia's CUDA 11.6 SDK for Arm[3], LLVM 13, and Fujitsu's SSL2 v1.2.34 library. For the baseline PyTorch measuremets, we use Fujitsu's pre-installed PyTorch (v1.5.0).

We evaluate the Rodinia and matrix multiplication tests on an AWS c6i.metal instance (dual-socket Intel Xeon Platinum 8375C CPU at 2.9 GHz with 32 cores each and 256 GB RAM) running Ubuntu 20.04. Measurements were performed on the first socket, with hyperthreading and turbo boost disabled. Each number is the median of at least 5 repetitions.

[2]MocCUDA and Polygeist are available at `https://gitlab.com/domke/MocCUDA` and `https://github.com/llvm/Polygeist`.

[3]Even though we will run PyTorch on a GPU-less system, we must compile PyTorch on a CUDA-enabled system to ensure the correct code is emitted. We also prevented inlining of three Pytorch functions.

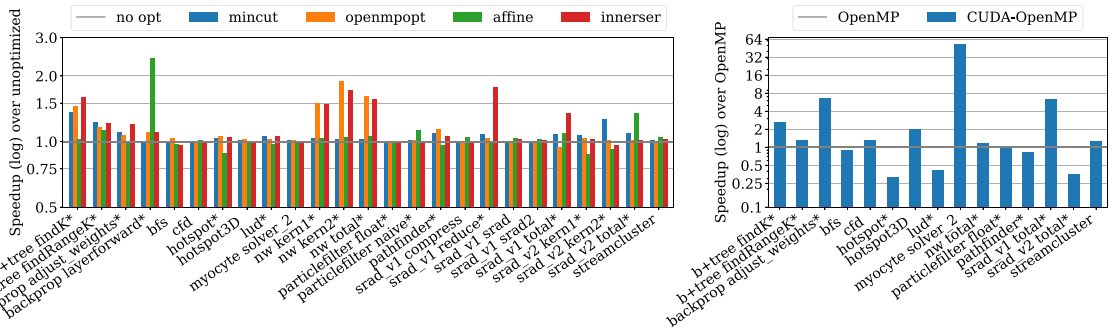


Figure 9-11: Left: Relative speedup (higher is better) applying parallel optimizations, proposed in Section 9.4, over our flow without optimization. Right: Speedup of transpiled CUDA-to-OpenMP compared against native OpenMP code (when available) running with 32 threads. Asterisks denote barriers within the benchmark.

9.6.1 Comparison to MCUDA

First, we compare with the previous work in MCUDA [372]. MCUDA is an AST-level tool which produces new CPU C/C++ as an output and uses loop fission to handle synchronization. As a source-to-source tool, MCUDA only handles a fraction of the input language, making it unable to run on Rodinia programs. Instead, we compare the runtimes of a matrix multiplication kernel across a range of threads (1–24) and matrix sizes (128×128 – 2048×2048) in Figure 9-10. Polygeist with all optimization excluding serialization of the inner loop (PolygeistInnerPar) produces code within 1.3% of MCUDA on average. PolygeistInnerPar has a 1.5% slowdown on 1 thread, and 3.2% speedup on 32 threads. This behavior is caused by OpenMP overhead in handling nested parallel constructs. In fact, MCUDA only parallelizes the outermost loop. When Polygeist also serializes the inner loops (PolygeistInnerSer), it achieves an overall 14.9% speedup over MCUDA, with a 4.5% speedup on 1 thread and 21.7% speedup on 32 threads.

9.6.2 Use case 1: Rodinia Benchmarks

We benchmarked the 14 benchmarks that are currently supported by Polygeist, and had a nontrivial runtime.[4] We verified correctness by comparing the program outputs produced

[4]The `hybridsort`, `kmeans`, `leukocyte`, `mummergpu` `huffman` and `heartwall` use unsupported C++ or CUDA features within Polygeist (virtual functions and texture memory). The `lavaMD` and `dwt2d` benchmarks use *ill-formed* C++ with undefined behavior due to reading from uninitialized memory. The `nn` and `gaussian` tests ran in ≤ 0.005 seconds.

by compiling with `nvcc` and executed on a GPU, and compiled by our flow and executed on a CPU. We also employed the use of CPU-based parallel and undefined behavior analysis tools, which via our tool, allowed us to successfully diagnose and repair one race bug and several undefined memory bugs in the original CUDA code. We inserted timing measurements across kernels and/or computational portions of the code that include kernels, in some cases multiple per benchmark. Where possible, we time equivalent portions of the OpenMP versions of the same benchmarks.

We compare the Rodinia CUDA benchmarks compiled for the CPU with the Rodinia OpenMP verions of the benchmark in Figure 9-11(right). While there is some variation from benchmark to benchmark, overall our approach is on par with the hand-coded versions of the benchmarks, and even nets a 58% geomean performance improvement, when the inner serialization optimization is enabled. Without inner serialization, we still see a geomean speedup of 34%. The speedup for `myocte` is largely due to fewer instruction and data cache misses on the transcompiled code, which comes from optimizations which specialize the (parallel) to kernel call context, as well as the CUDA version employing fewer branches. The speedup for `backprop` is partially due to parallel optimizations (see Figure 9-11(left)) and partially due to the CUDA code being implemented with a linear array, as required by CUDA, instead of the double-pointer used in the OpenMP code. The `srad_v1` benchmark benefits from a shared memory reduction in addition to parallel optimizations which eliminate most barriers and shared memory. In contrast, `hotspot` and `pathfinder` see a slowdown compared against native OpenMP code, due to duplicated computation in order to reduce synchronization and make better use of plentiful GPU parallelism. The slowdown for the transpiled CUDA version of `lud` is due to being written with a transposed loop ordering in contrast to the OpenMP code.

We test the scaling properties of our approach by comparing transpiled CUDA with native OpenMP kernels in Figure 9-12. Transpiled CUDA codes generally scale much better than the native OpenMP versions. As most CUDA programs are written with thousands of threads in mind, this indicates that our framework was able to preserve that parallelism as the GPU-specific constructs were being rewritten for CPU-compatible equivalents. On 32 threads without inner serialization, transpiled CUDA codes had a geomean speedup of

16.1× across all tests. As OpenMP versions of benchmarks do not exist for all tests, if we consider only CUDA codes for which there exists an OpenMP version, we find a geomean speedup of 14.0×, whereas OpenMP has only a speedup of 7.1×. Serializing the inner loop slightly reduces scalability, but still results in improved scalability over OpenMP, finding a geomean speedup of 14.9× over all tests with inner serialization enabled, and a 12.5× speedup on codes with OpenMP versions. That is, most of the speedup is due to transpliation and barrier optimization as illustrated in Fig. 9-14(right). Inner lop serialization was observed to be beneficial in presence of multiple outer loops for which the OpenMP model triggers barrier synchronization repeatedly after the inner loop.

We perform an ablation analysis to show how individual optimizations impact performance. The "mincut" series in Figure 9-11(left) shows performance with the optimization outlined in Section 9.3.2. This is only relevant for benchmarks containing barriers (marked by an asterisk in the Figure). When applicable, mincut provides a 5.8% geomean speedup. The "openmpopt" series in Figure 9-11(left) demonstrates the impact of OpenMP region merging and similar optimizations and results in a 10.5% geomean speedup. The "affine" series in Figure 9-11(left) shows the result of raising control flow to their affine variants and enabling simple serial and parallel loop optimizations (such as loop unrolling and reindexing). While this produces a geomean speedup of 5.4% across the board, it results in a 2.4× speedup for the backprop layerforward test as it results in a loop containing synchronization being fully unrolled and reduced to `if` statements.

9.6.3 Use case 2: Pytorch/Resnet50 Test

To evaluate the PyTorch Resnet-50, we execute a full node-parallel training run on one TofuD unit of the Fugaku FX1000 supercomputer, comparing against the native PyTorch CPU backend and the optimized oneDNN backend, as available. We *replaced* the functions related to computing log-likelihood with Polygeist-transpiled functions as their CUDA kernels use barriers and their CPU versions contain naive implementations, and dispatched other calls to relevant libraries.

We ran multiple forward and back propagation passes of Resnet-50 on 224×224 Im-

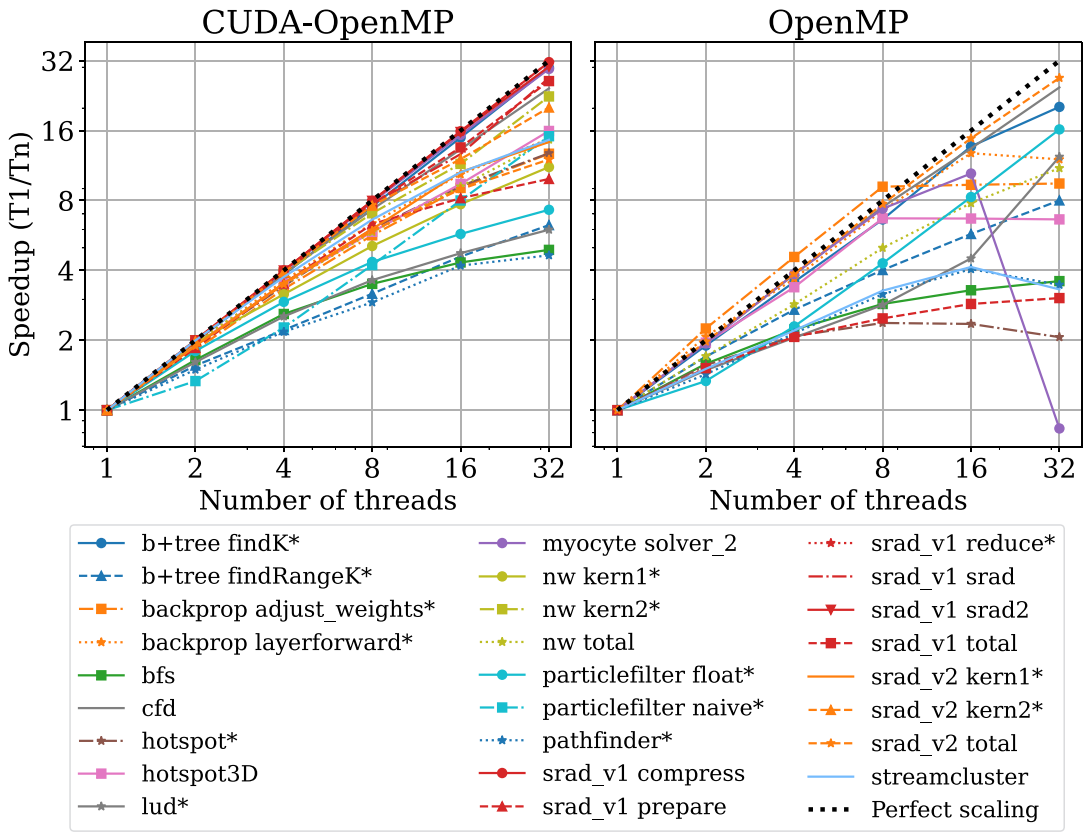


Figure 9-12: Scaling behavior behavior of CUDA Rodinia kernels, when run on the CPU with OpenMP, and OpenMP Rodinia kernels (where available), using 32 threads. Not all Rodinia CUDA kernels have OpenMP versions.

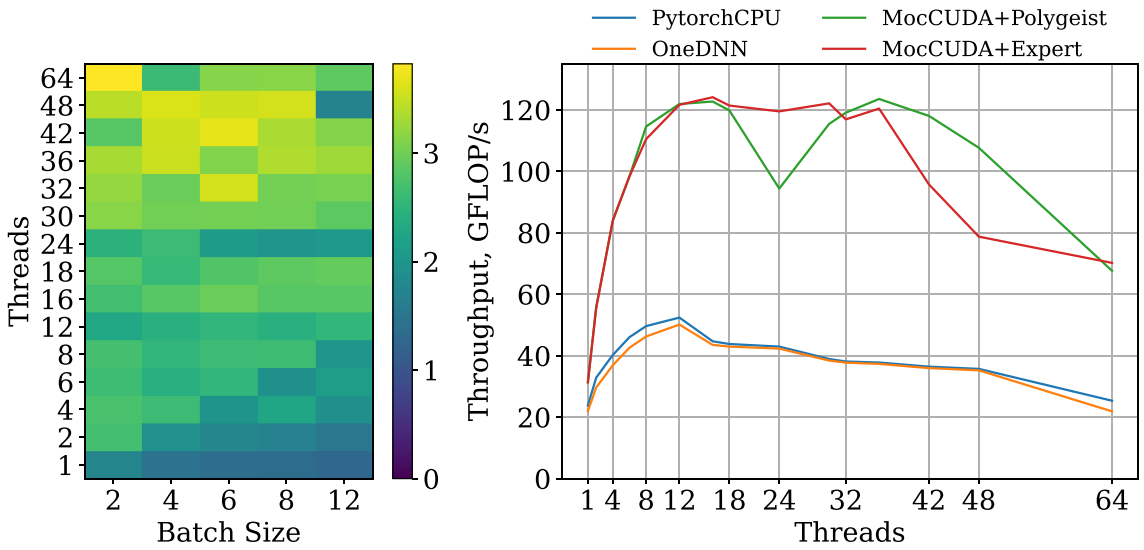


Figure 9-13: ResNet50 training on Fugaku node. Left: heatmap of relative throughput increase of "MocCUDA+Polygeist" over Fujitsu-*tuned* oneDNN, higher is better. Right: geomean throughput across batch sizes; "MocCUDA+Expert" uses an expert-written OpenMP kernel; "MocCUDA+Polygeist" uses the generated kernel, and PytorchCPU is Pytorch's native OpenMP backend.

ageNet in a data-parallel fashion. We employ Horovod's synthetic benchmarking script [352]. We build Horovod v0.19.5 with CUDA, LLVM, and Fujitsu's MPI library to enable multi-node, distributed deep learning on top of Pytorch. We assign one MPI rank per A64FX core memory group (CMG), emulating up to 4 GPUs per node, and scale the test from one node (2 ranks) to 12 nodes (48 ranks) in one TofuD unit (smallest 2×3×2 torus) while keeping the number of OpenMP threads fixed at 12 to accommodate one thread per core. We use Pytorch v1.4.0 for our approach, while the other backends depend on Pytorch v1.5.0.

Performance was measured in GFLOP/s by using perf, and Benchmarker [101], which sets up the neural network and test data and executes the layer. We run with batch sizes 1–228 on 1–64 threads, averaging across epochs, and we compare the different backends for batch sizes 1–12 where all backends ran successfully.

Peak performance for MocCUDA was achieved at batch size 168 with 42 threads at 943 GFLOP/s, which amounts to 14% of the theoretical peak of the A64FX processor [124]. MocCUDA systematically outperforms Fujitsu's tuned oneDNN across batch sizes and thread counts, yielding up to 4.5× throughput increase (geomean 2.7×, min 1.2×) as shown

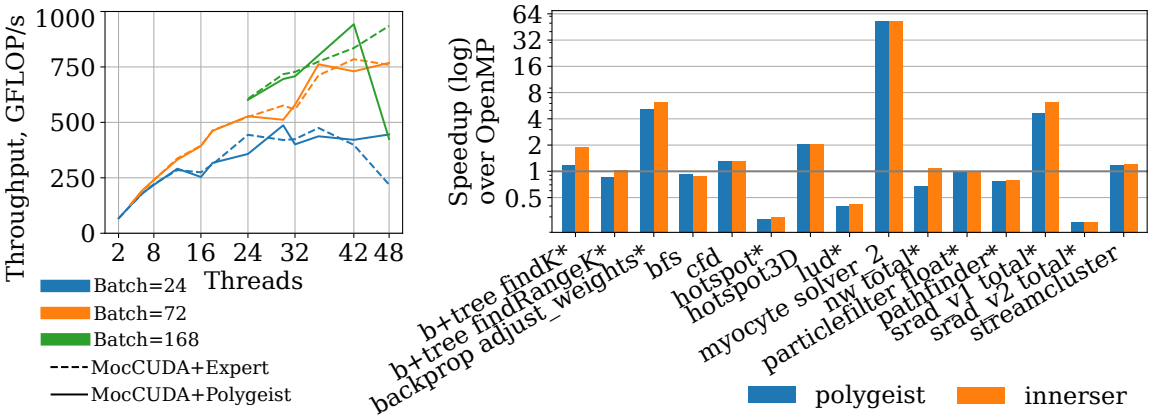


Figure 9-14: Left: ResNet throughput continues to scale for large batch sizes; large batches time out with few threads. Right: inner loop serialization contributes up to 30% speedup while most comes from barrier optimizations.

in Figure 9-13. MocCUDA with expert-written kernels is comparable to MocCUDA with Polygeist-generated kernels. Furthermore, the throughput of MocCUDA keeps increasing with the number of threads provided a sufficiently large batch size as shown in Fig. 9-14(left). For batch size 24, it plateaus at 24 threads while for batch size 168, it peaks at 42 threads.

The improvement can be explained by a combination of the PyTorch CPU design and performance characteristics of oneDNN. As Intel's oneDNN [188] does not account for HBM available on A64FX, it uses cache-friendly direct convolutions instead of GEMM-based convolutions, less efficient in presence of HBM for Arm CPUs. While the custom fork of oneDNN tuned by Fujitsu [125], improves upon Intel oneDNN's performance (though by a geomean of 6%), it still leaves room for performance improvements.

This demonstrates that our approach is capable of automatically deriving efficient versions of deep learning kernels (and potentially other applications) from their CUDA versions, thus addressing the limitations of missing or inefficient kernels for CPUs with high-bandwidth memory without the need for reverse or re-engineering the application.

9.7 Related Work

9.7.1 GPU to CPU Synchronization

One of the first tools for emulating GPUs on a CPU was provided directly by NVidia for debugging purposes and emulated each thread on the GPU with a distinct CPU thread. While functional, the large gap in the number of available threads makes the emulation inefficient.

MCUDA [372] (2008) performs an AST transformation of C GPU code to generate new C CPU code that calls a thread-independent parallel `for` routine. MCUDA pioneered the use of "deep fission" to handle synchronization, which splits parallel loops and other constructs at synchronization points in order to eliminate them. This fission technique is also applied in other tools: Ocelot [92] (2010), a binary-translation tool that parses PTX assembly into LLVM and just-in-time compiles kernel functions; POCL [196] (2015), a Clang/LLVM compiler pass for OpenCL; COX [160] (2021), another LLVM transformation pass for translation of CUDA that uses fission, and handles warp-level primitives; and even this work. While the intuition behind the fission approach is similar to that used here, we apply fission inside of a high-level compiler, rather than either source or a low-level IR. As demonstrated in Section 9.3.1, performing fission on structured programs enables more efficient code transformations. While applying fission at a source-level misses the opportunity to run optimizations before fission (like barrier elimination) and applying fission at a low-level requires attempting to reconstruct the high-level structure, operating within MLIR allows us to both apply optimization and preserve high-level structure. Moreover, source-level tools tend to be quite fragile as they must re-implement parsing and semantics or the target language (e.g. C++), and as a result only operate on a limited subset of the input language, requiring re-engineering effort to replace unsupported constructs (like pointer arithmetic).

Another approach uses continuation-passing to handle synchronization by creating state machine of all synchronization points (e.g. "microthreading") [370] (2010). Karrenberg and Hack [206] (2012) propose a continuation-passing approach in LLVM that includes an algorithm for detecting and reducing divergence in the control-flow-graph. Follow-up

work minimizes live values to reduce memory traffic [266].

VGPU [299] (2021) is similar to NVidia's virtual GPU, except using C++ thread and fence. Shared memory, implemented as a single global, is expanded by the number of blocks.

Prior work that operates at the low-level LLVM IR extends significant effort to reconstruct high-level constructs, such as loops and kernel configurations, required for either efficient fission or continuation passing. For example, POCL [196] runs canonicalizations and loop transformations to rewrite the control flow graph and attempt to recognize it as a specific form that can be handled. Prior work that operates at source/AST level (e.g. MCUDA), beyond still needing to recognize GPU-level concepts, cannot benefit from optimizations that simplify the code resulting in easier control flow.

In contrast, by operating on MLIR's mix-of-abstractions, we are able to simultaneously preserve source-level structure and perform program transformations such as loop unrolling or LICM that can, e.g., remove nested synchronization.

9.7.2 Parallel Portablity/IR, & OpenMP Optimizations

Several tools define new abstractions in the host language that are amenable to CPU or GPU execution. Examples include ISPC [301], RAJA [33], Kokkos [105], or MapCG [173] (limited to map-reduce code) in C++, Loo.py [216] in Python, and KernelAbstractions.jl [77] in Julia. These approaches provide performance portability for any new code written with them. However, existing code must be rewritten in said framework and may not compose with other frameworks/languages.

Several pieces of prior art discuss parallel intermediate representations, such as Tapir [344] for representing Cilk [123] in LLVM; OpenMPIR [367] for representing OpenMP in LLVM, PPIR [346] for pattern trees, and the MLIR OpenMP Dialect; as well as SDf3 [375] for visually representing concurrency as a control-flow graph. These works primarily focus on the *representation* for their particular style of parallelism (e.g. OpenMP tasks in OpenMPIR), which does not include GPU-style barriers, rather than on parallel *transformations* (such as barrier elimination) or optimizations, with the exception of consistency/race

checks or automatic parallelization [267, 291].

The use of OpenMP parallel region expansion is known to be beneficial [95]. Clang/LLVM optionally supports the transformation in a weaker form [241].

9.7.3 Barriers

Several pieces of prior work explored the semantics of barrier or synchronization instructions, including in relation to GPUs. Work has been done to verify the correctness of barriers [10]. [360] experimentally evaluates the forward progress, fairness models of various GPU vendors. [359] implements a GPU barrier that applies across work-groups, as opposed to just within a work group. [377] add Java memory barriers to programs to ensure weak and sequential consistency semantics. They find that without synchronization and delay set analysis, introducing consistency semantics has an average 26.5× slowdown, whereas when using these analyses to insert fewer synchronizations can achieve a 10% and 26% slowdown for weak and sequential consistency, respectively.

Barrier elimination was implemented in the SUIF compiler for SPMD with shared memory [390] and for software-distributed memory [159]. This relies on a purpose-built communication analysis across the barrier whereas our method leverages the memory effects of the barrier itself. On the other hand, it supports synchronization minimization, such as replacing a barrier with nearest-neighbor communication, which our flow currently does not. Several pieces of work have proposed code generation techniques or code transformations aimed at minimizing the amount of synchronization within SPMD programs [89] or imperfect loops [292]. These approaches are applied to a sequential program, or one without synchronization at all, while our approach is applied parallel CUDA programs.

Synchronization minimization was explored within the polyhedral framework [240]. PolyAST supported analysis and transformation of programs with OpenMP directives [67]. While our flow may benefit from the polyhedral representation, it may operate without it and supports a significantly larger set of input programs. Razanajato et.al. leveraged the framework to generate different OpenMP parallelism constructs [320], which is complementary to our code generation.

9.8 Conclusion

By extending Polygeist/MLIR, we developed an end-to-end system capable of representing, optimizing, and transpiling CPU and GPU parallel programs. A key component of our framework is the development of a high-level barrier operation, key to representing GPU programs, whose semantics can be fully defined by its memory behavior. Unlike prior representations of parallel barriers, our semantics enable direct integration of barriers within optimization. As efficacy validation, we demonstrated GPU to CPU transpilation of a subset of the Rodinia benchmark suite on a commodity CPU and transpile Resnet-50 from the PyTorch CUDA source to run on A64FX CPU. The Rodinia benchmarks achieve a 58% geomean speedup of the transpiled GPU code over handwritten OpenMP versions. Similarly, we observe a $\approx 2\times$ speedup of transpiled kernels over the native PyTorch CPU backend.

Currently, the transpiled GPU code keeps the same schedule when run on the CPU, except for the innermost loop serialization that improves performance. A fruitful avenue of future work may perform advanced rescheduling the code to better take advantage of CPU-style memory hierarchies.

Chapter 10

"Header-time" Optimization

```
double mag(double *A, uint N);

void norm(double *In, double *Out) {
  for (uint i = 0; i <= N; ++i)
    Out[i] = In[i] / mag(In,N);
}
```

Figure 10-1: A file with a function definition (`norm`) and a function declaration (`mag`). As the latter is opaque, it is illegal to move the call outside the loop, resulting in a runtime of $O(N^2)$.

10.1 Introduction

Writing fast code is difficult. Writing code which can automatically be made fast by a compiler can be even more difficult. The latest version of the LLVM compiler [230] has 1983 unique command line options, 166 optimization and analysis passes, and more than 100 attributes. As a result, developing optimizable code currently requires both expertise in performance engineering, as well as in compiler optimizations.

Consider the code in Figure 10-1 which defines a function `norm` which normalizes a vector by calling an $O(n)$-time external function `mag` to compute the magnitude in a loop. This causes the program to run in $O(n^2)$. While the programmer may intend for the result of each call to be the same, the function is declared externally preventing the compiler from

proving that it can hoist the call outside the loop. One could add additional information to the callsite like restrict to guarantee that the input won't be overwritten by a write to the output, but that isn't sufficient since the `mag` function could read or even modify a global variable. To ensure LLVM is able to successfully perform the optimization, one needs to mark `mag` as being `readonly` and `argmemonly` (meaning it only touches memory passed as arguments), two internal attributes that don't even have representations in C++! This is shown in Figure 10-1.

Compiling the definition of `mag` function will automatically derive this information for us. Sadly, this information is only available within a single file, preventing all other files from benefiting from its information. While theoretically one could combine all of the code of an application together in a single file either in source (e.g. a Unity build [263]) or object-form (e.g. link-time optimization (LTO) [197]), this significantly hinders the speed of building applications. A large codebase, such as the Chromium web browser [76] with more than 24 million lines of code, can take the better part of a day to compile sequentially while a parallel compilation can reduce that by a factor of 50× to ≈20 minutes [120].

```
__attribute__((fn_attr("readonly"), fn_attr("argmemonly")))
double mag(double *A, uint N);

void norm(double *restrict In, double *restrict Out) {
  double licm = mag(In,N);
  for (uint i = 0; i <= N; ++i) {
    Out[i] = In[i] / licm;
  }
}
```

Figure 10-2: A file with an annotated declaration of `mag`, allowing the compiler to successfully hoist the call outside the loop and reduce the runtime to $O(N)$.

In this work we propose a new approach which preserves compiler-derived information, without limiting build parallelism. Our approach, HTO or "Header"-time optimization, encodes the program analysis information available during compilation into automatically generated support files that augment existing declarations in subsequent compilations. Such additional information is especially useful as it restricts the potential effects of functions that are declared and used but defined in a separate translation unit. As a result, calls to ex-

ternal functions may no longer be an optimization barrier for the compiler and consequently allow more optimizations.

10.1.1 Contributions & Overview

This chapter makes the following contributions:

- The design of HTO, a novel method to reuse available knowledge to enable inter-translation unit analyses by providing annotations in the source code.
- An implementation of HTO in the Clang/LLVM compiler.
- An evaluation of traditional compilation, state-of-the-art LTO techniques, and HTO to evaluate the performance and compilation time advantages of sharing information across translation units.

10.2 Related Work

Inter-translation unit optimization, also referred to as whole-program optimization (WPO) or cross-module optimization (CMO) in the literature is commonly performed at link time because the entirety of the code's symbols are available.

An early example of link time optimization (LTO) was the HP-UX compiler [20] which effectively merged all sources into one unit, performing inter- and intra- procedural optimizations from there. Both GCC [365] and LLVM/Clang [230] support a similar technique, referred to as "LTO mode" in GCC and ("full"-)LTO in LLVM/Clang. This whole-program concatenation technique provides the greatest optimization opportunity as well as the largest cost. First, a single module with the entire source code of a large application, together with the memory requirement of the compiler itself, can easily overwhelm a single node. Notably, the HP-UX LTO implementation [20] avoids memory exhaustion by paging to disk, resulting in substantial slowdowns. Second, the single module prevents the embarrassingly parallel compilation of translation units. While intra-translation unit parallelization is possible, it is significantly harder to do as the module itself is inherently

a shared resource. In addition to compiler-based inter-translation unit optimizations there are source based schemes that can be used with any compiler. For these, usually referred to as unity-builds [263], all source code is copied, e.g., via `#include`, into a single source file that is then compiled. Our annotated header approach provides a new way to perform inter-translation unit optimizations. In contrast to the existing techniques we avoid any sequentialization as well as code duplication, e.g., via inter-translation unit copying or inlining. This allows us to strike a more beneficial trade-off between compilation time cost and execution time benefit. Furthermore, HTO does not require the source files to be distributed in a compiler intermediate format and provides (partial) interoperability between compilers. The former is especially useful for sensitive code that cannot be shared in non-binary form but for which we still want to allow inter-translation unit optimizations.

To reduce build time and memory usage, compiler-based schemes have introduced function summaries. Summary-based LTO is usually less effective than full LTO as not all inter-procedural optimizations are possible with only summary information. Summary schemes typically consist of a parallel summary generation phase, a serial summary aggregation and/or optimization phase, followed by a potentially parallel final optimization phase. SYZYGY [268] provides summary-based LTO for HP-UX, WHOPR [137] for GCC and thin-LTO [197] for LLVM/Clang. AMD's Open64 compiler also performs summary-based LTO [14]. A common goal is to reduce or eliminate the serial summary aggregation/optimization phase to better enable parallel or incremental compilation. Our approach is centered around attributes that provide semantic information for functions, parameters and return values. The annotated function declarations are a form of summary, though one that is (partially) interoperable between compilers and at the same time human readable (ref. section 10.3.2).

Inlining is usually a critical transformation during compilation. How and when to inline function definitions across translation unit boundaries was therefore researched extensively in the past. Hall [158] uses a whole program call graph to schedule inter-procedural optimizations and in particular examines the impact of inlining. Triantafyllis et al [386] describe the Procedure-Boundary Elimination framework which creates a whole-program control-flow graph with focus on individual regions to provide a more scalable approach

for inlining.

Profile-guided optimization (PGO), also known as Feedback-Directed optimization (FDO), is a technique that allows information from a prior program execution to better inform optimizations [79, 230]. PGO is related to our work (HTO) as feedback acquired via a special compilation is utilized in subsequent compilations. In contrast to our work, PGO collects information while running the program and HTO collects or preserves the information derived by compiling the program. Performing this addition compilation and profiling runs is often very beneficial, especially for large codebases such as Chromium [76]. LIPO [235] is an extension to GCC that combines LTO with PGO, creating dynamic link-time summaries construction during the profile executable run.

Doerfert, Homerding, and Finkel [97] have shown that function attributes on declarations, among other things, can substantially improve performance of LLVM/Clang on scientific proxy applications. However, their exploratory search for "valid" attributes is costly and designed for a one-off usage where the results are manually verified. HTO provides correct attributes by construction and the approach is lightweight enough to be used as part of continuous integration testing.

10.3 Header-Time Optimization

Information derived in the course of compilation is in the form of runtime data-structures such as aliasing sets or dominator trees accessible to optimization passes within a compilation unit. To facilitate persistent information, compilers allow annotations of their intermediate format. In header-time optimization, we leverage attribute annotations attached to function definitions in order to provide additional inter-translation unit optimization. The key idea is to lift existing compiler-derived information back into the source code level such that follow up compilations can benefit from the already computed results. Providing information in the source code allows HTO to define annotations that are interoparable between different compilers. The augmented source files can be generated separately and added for future use to existing installations without modifying it. Moreover, HTO is compatible with existing compiler technologies such as PGO and LTO.

10.3.1 Attributes

In the LLVM intermediate representation (LLVM-IR) [82] there are currently 69 target independent attributes. Some attributes are introduced by the frontend, e.g., `always_inline` is created for `__attribute__((always_inline))` in C/C++, while others can also be derived by analyses, e.g., `readonly` to indicate a function will only read memory or function parameter is only read and not written. We refer the reader to the attribute sections in the LLVM-IR language reference [82] for an up-to-date list. Prior work [97] provides a good description of how these attributes enable optimization, as well as their relationship to C/C++ attributes, if any.

LLVM-IR attributes can be attached to values in different positions. (a) Functions; the attribute property holds for the entire function body. (b) Parameters; the attribute property holds for the value of the parameter at the function entry. (c) Function returns; the attribute property holds for the value returned by the function. In addition, each position has a corresponding attribute for a call site that provides the same semantics, but limited to a single call site. In this work we utilize only the information deduced by the compiler for positions (a) to (c) as those are meaningful in the context of a function declaration. In LLVM 11, three passes are for the most part responsible to deduce attributes: 1) `InferFunctionAttrsPass`, which annotates known library functions with attributes based on the semantics given to them by a language standard; 2) `(PostOrder)FunctionAttrPass`, which infers ten LLVM-IR attributes based on the function definition and call sites; 3) `Attributor`, a recent and ongoing development to replace the `FunctionAttrPass` pass and other IPO passes in a composable manner which deduces 19 LLVM-IR attributes. While the `Attributor` is not enabled by default in the LLVM optimization pipeline, we enabled it for all our experiments, including the baseline.

10.3.2 Attribute Generation

To lift attributes from the compiler intermediate level we added an additional pass to the LLVM pass pipeline (-O1, -O2, -O3, etc.) that runs at the very end, just before the LLVM-IR level is left. For all externally visibly functions in the translation unit, this HTO pass

will extract the compiler-derived attributes at the attribute positions (a) to (c) described in subsection 10.3.1. The information is either presented to the user in form or source code remarks or used to create annotated function declarations in new support (header) files.

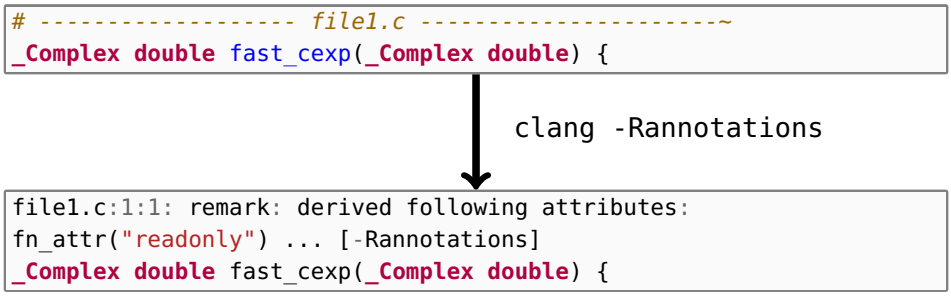


Figure 10-3: Remark Mode: We leverage existing functionality [246] to emit optimization remarks listing the attributes that were deduced for functions, parameters, and return values. Remarks can be printed for human consumption, dumped to machine readable formats, or viewed in graphical tools where remarks are displayed at the source locations.

Remark Mode

In remark mode, HTO summarized deduced attributes in compile-time remarks [246]. The existing infrastructure allows the user to inspect the information on the command line, as illustrated in Figure 10-3, or, alternatively, use existing tooling support, e.g., to visualize the remarks [129]. Remark mode allows users to selectively incorporate the deduced information manually into their program, e.g., via existing C/C++ attributes. It also provides a novel way to gain insights into the reasoning the compiler performed. As shown in Figure 10-4, the presence as well as absence of attributes can provide a deeper understanding of the program, the compiler capabilities, and bugs.

```
sum.c:1:1: remark: derived following attributes:
fn_attr("readnone") [-Rannotations]
double sum(double* array, int size);
```

Figure 10-4: HTO derived that the function `sum` is `readnone`, indicating that it doesn't read any memory. Given that `sum` should return the sum of all elements in an array, this is definitely a bug as it cannot produce a correct output without reading the array.

Header Mode

In header mode, HTO creates function declarations in the source with attributes that encode the information deduced during the compilation. We provide two distinct use cases for using HTO below. The annotated declarations are designed to be includable via existing compiler flags while the header generation is triggered by the new `-hto-dir` command line flag.

Single-App Header Mode In the first use case of header mode HTO, a program is initially compiled to generate annotated headers. These headers can either be standalone or attached to the existing ones. The augmented headers are then included in subsequent compilations to provide low-overhead inter-translation unit information. This is similar to the initial compilation of a program version that collects runtime profiles for profile-guided optimizations (PGO). The key difference here is that the "profile" for HTO was recorded during compilation of the program and not during the program run. Moreover, the initial compilation used to generate PGO information can also be used to generate HTO headers, and thus does not require an extra compilation. Note that subsequent compilation can update and improve the HTO headers as discussed further in subsection 10.3.3.

Figure 10-5 illustrates the workflow for the single-app header HTO mode. The code shown in the example is similar to Figure 10-2 but inspired by an early version of the RS-Bench proxy application [384]. Doerfert, Homerding, and Finkel [97] exposed the call hoisting opportunity shown here through their search of optimistic annotation opportunities. Adding a missing attribute to the program caused a $\approx 10\%$ speedup. As a result, the authors of the application since changed their code manually to benefit from the enabled optimization consistently. HTO provides developers with an automated mechanism to discover such missing information.

Header Mode: Libraries The second use case for header mode HTO is library deployment. All non-trivial programs use one or more libraries that are built at different times, using different compilers, and usually distributed only with binaries and a set of declarations to define the API (i.e. headers). When a library is deployed, most calls into the library will effectively act as optimization barriers. While LTO schemes can provide information

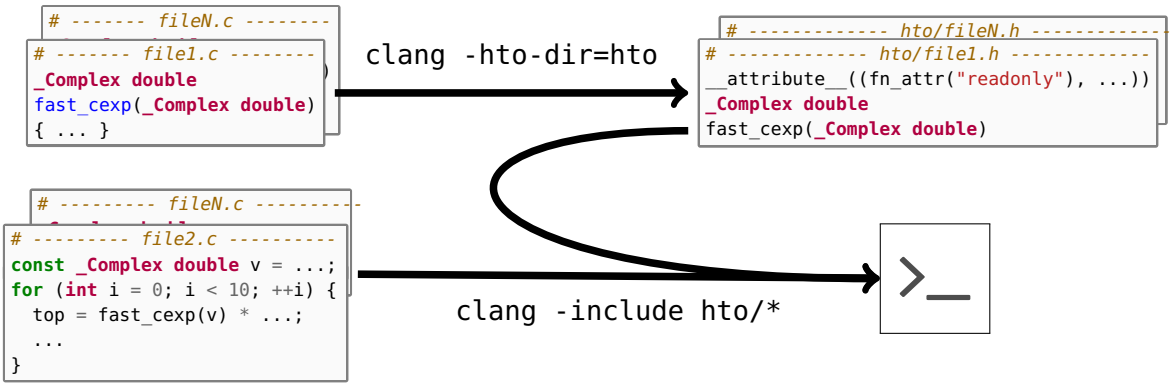


Figure 10-5: ***Header Mode: Single-App.*** Through the `-hto-dir` flag a header files for each source file is generated. The header contains an augmented function declaration for each externally visible function definition in the input. In subsequent compilations those headers can be easily included to expose optimization opportunities, e.g. to allow hoisting of `readonly` calls out of loops, similar to the motivation example in Figure 10-2.

when an application is linked against an "LTO-enabled" library, there are drawbacks. For one, LTO schemes are tied to a specific compiler and not commonly distributed for these portability reasons. Additionally, these schemes may require the library source code to be exposed to a non-trivial degree, a potential problem for proprietary code. The HTO approach allows library writers to create annotated headers with information extracted during the library compilation. This information can be easily restricted to a subset if need be, e.g., for intellectual property reasons. A key difference to HTO single-app mode is that there is no separate compilation necessary for the application to benefit from HTO. The header generation was performed once, during the library deployment, and users can continue to benefit with minimal involvement and overhead. The library headers can even be prepared to automatically include the HTO header if the compiler is capable of using the contained information, though we have not explored this option yet.

10.3.3 Inter-Translation-Unit Data Flow Analysis

For both single-app and library uses cases of HTO, one can run more than one round of HTO. In this way, headers created in a previous compilation are used to produce headers in a subsequent compilation.

Attribute deduction in LLVM, like many other compiler analyses, is generally per-

formed as a fixpoint data-flow analysis [213]. There are two ways to initialize the fixpoint equation system, “optimistically” or “pessimistically”. In “optimistic“ mode (the default used within a translation unit), the system has to iterate until a fixpoint is reached because the intermediate states are *not* sound approximations of the program behavior. However, the final fixpoint is, under some additional constraints, known to be reached and known to be optimal. In “pessimistic” mode, the system can use any intermediate state to optimize the program as it is known to be a sound approximation of the program behavior. The resulting fixpoint may not be optimal, however.

Fixpoint data flow analysis is a common technique used in both intra- and inter-procedural settings. With HTO, such analyses can be extended to the inter-translation unit scope. Using HTO and recompilation, existing attribute deduction becomes inter-procedural across translation units. Eventually a fixpoint is reached in this setting as well. As with the transition from intra- to inter-procedural analysis, inter-translation unit analysis can significantly improve the compiler’s ability to reason and optimize.

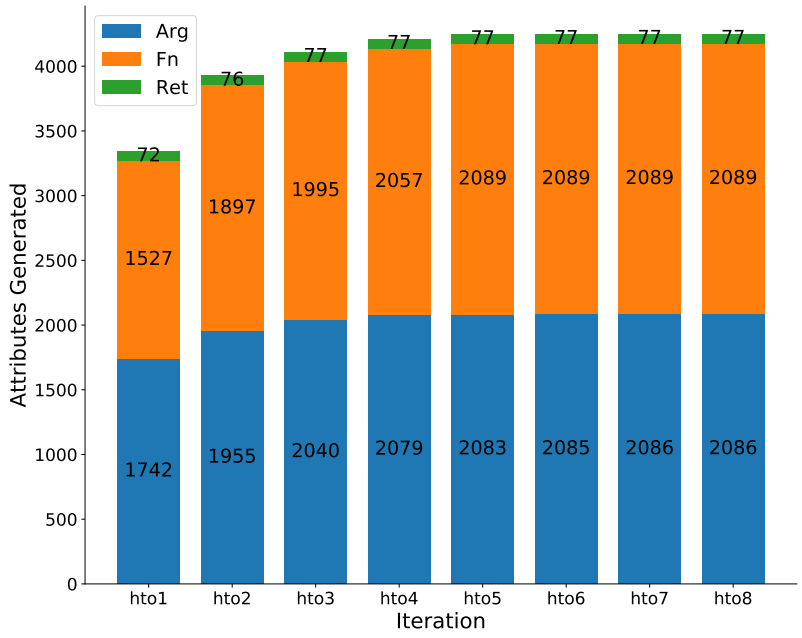


Figure 10-6: The y-axis shows the number and kind of attributes in HTO headers for the *musl* standard C library [115]. The x-axis show how recompilation using the existing HTO generated headers and producing new augmented headers changes these numbers until a fixpoint state is reached. Note that we exclude 1558 `unwind` and `strictfp` function attributes from these graphs as they are added to *all* functions due to the compilation configuration.

To determine the efficiency of inter-translation unit data flow analysis, we recompiled the *musl* standard C library [115] in HTO header mode such that it included and produced annotated headers until a fixpoint was reached. In Figure 10-6 the number and kind of attributes in the HTO generated headers after each iteration is shown. Note that we did not count two function attributes that were always present due to the compilation options (C language with strict floating point arithmetic). While many attributes are derived in the very first iteration (without including any HTO headers), performing additional rounds of HTO is able increase the number of attributes found. We see that after seven compilations a fixpoint state was reached. A selection of the attributes that were derived, together with their number of occurrences in the first and last iteration, is given in Figure 10-7. The left table shows that the iterations mostly increased `readonly`, `nofree`, and `nocapture` attributes for parameters. The `readonly` and `nocapture` attributes are especially helpful for code movement transformations that involve memory. The `nofree` attribute is not widely used yet but will allow to preloading memory locations, when beneficial. As shown in the right table, various function attributes that restrict potential memory effects (`writeonly` to `readnone`) have been added during the fixpoint iteration. Furthermore, the compiler was able to determine the recursion and return behavior of more functions.

10.4 Evaluation

We evaluated HTO on the multi-source portion of the LLVM test suite [247]. For all experiments we ran an initial compilation to generate annotated headers. Afterwards, we collected evaluation numbers for four distinct configurations: a reference release build (our baseline), a release build utilizing the annotated headers (HTO), a release build with thin inter-translation unit function summaries and selective code duplication (thin-LTO), and full (monolithic) link-time optimization (full-LTO). We found the performance of full-LTO to be similar to that of thin-LTO and for brevity, plot just the compile and execution time of thin-LTO.

attribute	hto1	hto8
`returned`	19	19
`align`	28	28
`nonnull`	28	28
`deref.`	28	28
`readnone`	109	113
`writeonly`	132	146
`readonly`	196	290
`nofree`	283	438
`nocapture`	466	543

attribute	hto1	hto8
`noreturn`	11	11
`writeonly`	32	38
`readonly`	42	98
`inaccmemonly`	53	105
`argmemonly`	113	159
`readnone`	161	179
`norecurse`	256	346
`nofree`	264	370
`willreturn`	285	354
`nosync`	300	418
`nounwind`*	1558	1558
`strictfp`*	1558	1558

Figure 10-7: The kind and number of LLVM-IR attributes [82] derived for the *musl* standard C library [115]. The hto1 column describes the first compilation with HTO, hence the attributes produced without any existing HTO headers. The hto8 column shows the fixpoint state after 7 recompilations in which prior HTO results have been used and new augmented HTO headers have been produced. The left table shows a selection of relevant parameter attributes and the right table lists selected function attributes. Note that `nounwind` and `strictfp` are listed here for reference while it is eliminated from Figure 10-6, all functions have that attribute by construction.

10.4.1 Setup

Each execution time and build time is the median of 10 runs on an Amazon AWS c5.metal instance, which is a dual-socket Intel Xeon 8275CL system with a total of 192 GiB of memory. Each Xeon is a 3.0 GHz 24-core CPU with a shared 35.75 MB L3-cache. We selected all tests that had a runtime of above 0.5 seconds to further reduce the impact of noise. Compilation of the tests was done serially to further reduce noise.

10.4.2 Speedup Potential

Of the programs tested in the suite, 24 tests showed a performance impact of above than 2% compared to the baseline. These tests are shown inside the box in Figure 10-8. Thirteen tests showed significant and similar speedups for both HTO and thin-LTO. These tests are shown within the blue oval. Eleven tests improved only with thin-LTO and not HTO.

While thin-LTO and HTO can derive similar speedups on many programs, they do so by different means. Thin-LTO is heuristic based and the duplication of function definitions

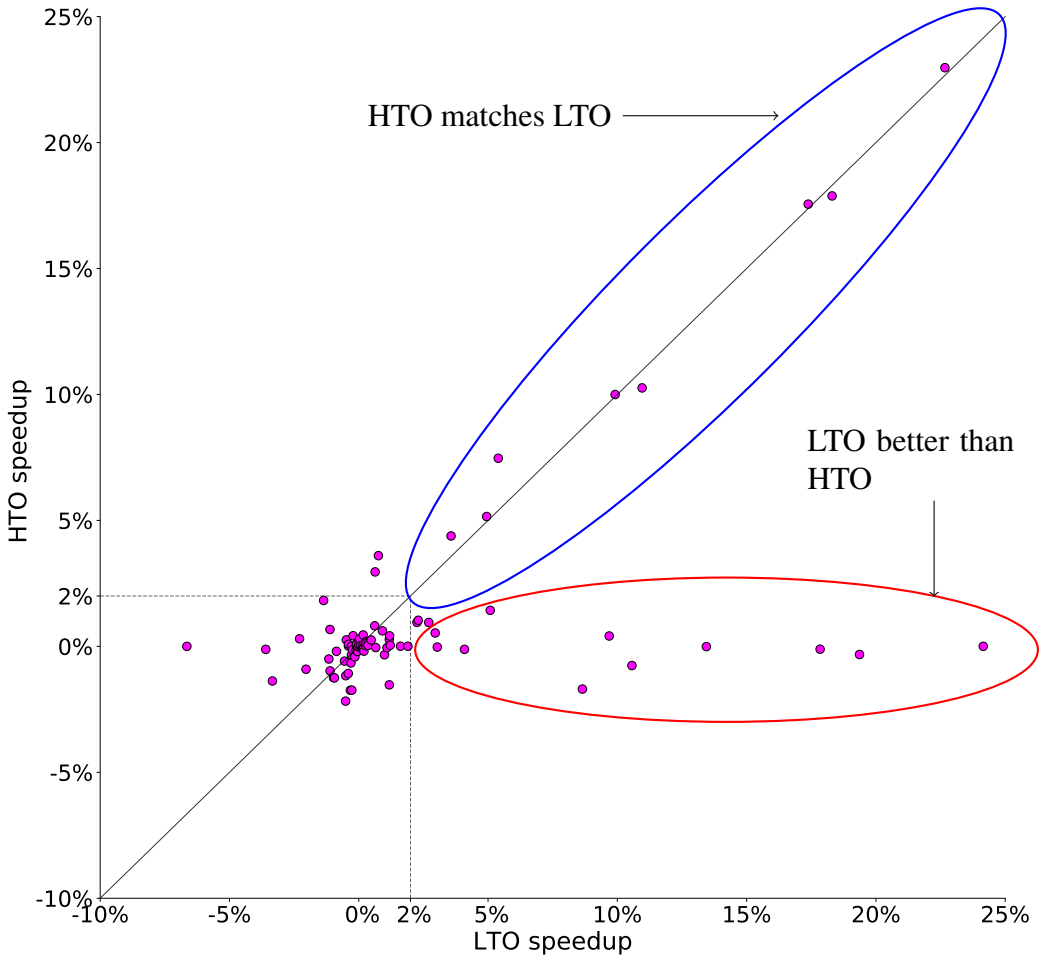


Figure 10-8: Speedup of HTO (y-axis) and thin-LTO (x-axis) against vanilla LLVM for the multi-source tests in the LLVM test suite. The box encloses all tests with less than 4% performance difference. The blue oval highlight tests that have a significant speedup for both HTO and thin-LTO. The red oval highlights tests where LTO achieves a speedup not found by HTO.

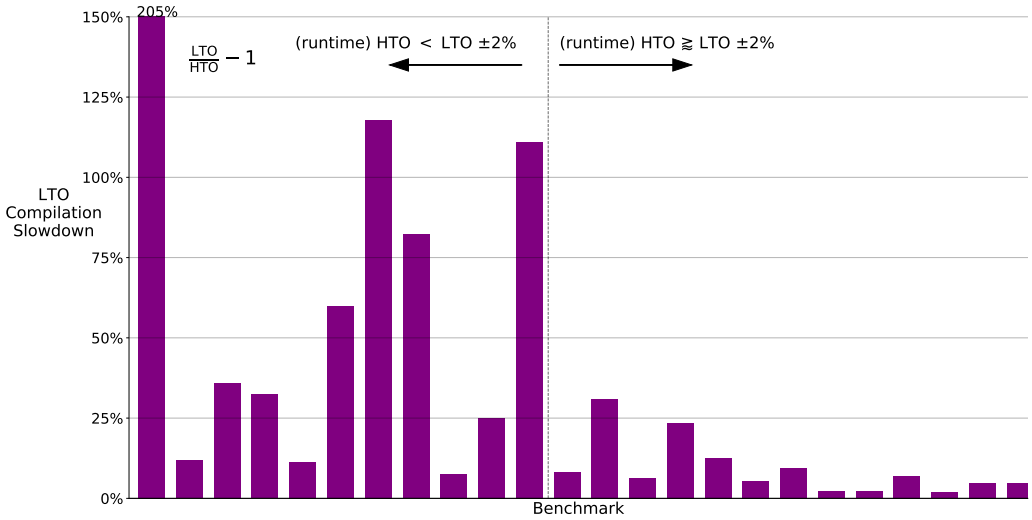


Figure 10-9: Compile-time overhead of LTO and HTO on codes where a speedup exists. Codes on the left have the greatest speedup on LTO and not HTO, whereas codes on the right have the greatest speedup on both LTO and HTO.

is driven by inlining. LLVM is historically better at inlining and intra-procedural optimization than inter-procedrual optimizations. Function body duplication and inlining both come at a cost, potentially with regards for code-size and most certainly with regards to duplicated optimizations. This forces thin-LTO to carefully choose whether to duplicate a function from one translation unit into anther one. If a declaration is not duplicated, no inter-translation unit reasoning is performed. HTO, however, provides attribute summaries for all functions definitions in the application. This allows HTO to achieve some speedup not found by thin-LTO if the thin-LTO heuristic decided against importing the function definition. At the same time, thin-LTO is able to achieve speedups that HTO cannot, especially if it is related to inlining or inter-translation unit code motion. The most significant improvements for thin-LTO in our experiments were caused by inlining an originally indirect function call; this is out-of-scope for HTO with the current set of LLVM-IR attributes.

10.4.3 Speedup Cost

In Figure 10-9, we compare the build time for all programs which had a significant speedup (e.g. those in the blue or red ovals from Figure 10-8). The programs in where HTO and LTO both had speedup (e.g. the blue oval) are on the right and the programs where LTO

had a speedup where HTO did not (e.g. the red oval) are on the left. For the programs where LTO found a speedup that HTO did not find, it required a significant additional cost in build time, presumably from various interprocedural optimization.

10.5 Conclusion

In this chapter, we present HTO, a novel method for inter-translation unit optimization by sharing compiler-derived attributes in source annotations. HTO is able to achieve half of the benefits of traditional summary-based thin-LTO for less than half the compile time cost. Furthermore, HTO allows compiler-agnostic, hence portable, optimization between libraries and applications that are compiled and distributed independently. We demonstrate that attribute summaries can already provide sufficient information in many cases where thin-LTO provides a benefit.

We also present several avenues of future work. Teaching the compiler to be selective about what generated headers to import may provide additional compile-time benefit. Additional integration of HTO into the compiler can lower the dependence on C/C++ language semantics and provide additional benefits in compile time, execution time, and ease of integration. Providing mechanisms that combine the attribute-based approach of HTO and inlining-based approach of traditional thin-LTO may be able to create additional benefits. This could happen in one of several ways: (1) Running both HTO and LTO to create annotated headers based of LTO builds. This is similar to the iterative HTO compilation described in subsection 10.3.3. (2) Inclusion of small function definitions in generated headers to allow inter-translation unit inlining with HTO. (3) Unconditional distribution of function attributes in the thin-LTO synchronization step to gain the HTO benefit in this compiler-specific mode.

Finally, as new attributes are added (10 of the 69 LLVM-IR attributes were added in the past 15 months), we expect both inter-procedural analyses and HTO to improve naturally.

Chapter 11

Concluding Thoughts

This thesis has discussed how to reduce the burden of programming by endowing a general purpose compiler with domain-specific information and transformations. Unlike traditional domain-specific languages or portability frameworks that require programs to be rewritten, this enables programmers to use such tools to be used without significant work. Moreover, by operating within a general-purpose compiler, these representations and optimizations are composable by definition leading to combined benefits which are far greater than any individual piece of domain-specific optimization in isolation.

In addition to its theoretical insights and benefits for each domain of interest, a key contribution of this thesis has been the construction of open source, real-world compilers that enable others to put these ideas to use. This open development not only provides validation of the concepts within, but also provides value to the broader scientific community. The Tapir compiler (Chapter 2)[1] for parallel programs has seen adoption as part of OpenCilk [342], the Swarm hardware architecture [422], the Department of Energy's Katsune project [367, 226] and inspired interest in parallel compiler optimizations like OpenMPOpt [94]. Tensor Comprehensions (Chapter 3)[2] was in use by Facebook and been part of revitalized interest in leveraging compilation for deep-learning systems in tools like MLIR [229], Pytorch 2.0 [416], JaX [53], Glow [329], Tiramisu [23], and others. Autophase (Chapter 4)[3] has seen use in the HLS and traditional compiler communities and has

[1] `https://github.com/wsmoses/Tapir-LLVM`
[2] `https://github.com/facebookresearch/TensorComprehensions`
[3] `https://github.com/ucb-bar/autophase`

been adopted by the CompilerGym [86] framework. Enzyme (Chapters 5, 7, 8)[4] has been adopted by a variety of communities for tasks ranging from exascale material science [408], differential rendering [424], molecular dynamics [143], black hole imaging [382], finite elements [16], climate modelling [167], solid mechanics [57], building design [296] and has been accepted as an incubator project of the LLVM foundation. Polygeist (Chapters 6, 9)[5] has also become an LLVM incubator project and been adopted by Xilinx/AMD [78], Imperial College London [429], EPFL [300], UIUC [421] and others for HLS; Intel/CodePlay for SYCL [189]; ETH Zurich for scheduling [37]; and several others as a C/C++ frontend [335, 7]. HTO (Chapter 10)[6], is being explored by the LLVM community and its use is being explored by Google.

The importance of domain-specific program representations and optimizations will only grow with time. This trend has been accelerating due to both the proliferation of new hardware architectures and the widespread adoption of software throughout the scientific community. To serve as case studies, this thesis chose to study key domains for both scientific computing and machine learning. The areas studied within this thesis do not exhaustively handle every potential domain, however, and instead explored topics which are of importance to broad communities. The ongoing advancement of science means that there will always be new domains requiring optimization, be it a niche research-group scale subdomain of physics to an entirely new field. In addition to creating compiler representations for the wide-reaching domains everyone will benefit from, I believe that an important area of future research will be extending these ideas to allow the compiler to understand the domain information within arbitrary libraries.

[4] https://github.com/EnzymeAD
[5] https://github.com/llvm/Polygeist
[6] https://github.com/wsmoses/LLVM-HTO

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[3] Daniel S Abdi, Lucas C Wilcox, Timothy C Warburton, and Francis X Giraldo. A GPU-accelerated continuous and discontinuous Galerkin non-hydrostatic atmospheric model. *The International Journal of High Performance Computing Applications*, 33(1):81–109, 2019.

[4] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. Using machine learning to focus iterative optimization. In *Proc. of the Intl. Symp. on Code Generation and Optimization*, CGO'06, pages 295–305, Washington, DC, 2006. IEEE Computer Society.

[5] Felix Agakov, Edwin Bonilla, John Cavazos, Björn Franke, Grigori Fursin, Michael FP O'Boyle, John Thomson, Marc Toussaint, and Christopher KI Williams. Using machine learning to focus iterative optimization. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 295–305. IEEE Computer Society, 2006.

[6] Shivali Agarwal, Rajkishore Barik, Vivek Sarkar, and Rudrapatna K. Shyamasundar. May-happen-in-parallel analysis of X10 programs. In *PPoPP*, pages 183–193, 2007.

[7] Nicolas Bohm Agostini, Serena Curzel, Vinay Amatya, Cheng Tan, Marco Minutoli, Vito Giovanni Castellana, Joseph Manzano, David Kaeli, and Antonino Tumeo.

An mlir-based compiler flow for system-level design and hardware acceleration. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '22, New York, NY, USA, 2022. Association for Computing Machinery.

[8] Karsten Ahnert and Mario Mulansky. Odeint – solving ordinary differential equations in C++. *AIP Conference Proceedings*, 1389(1):1586–1589, 2011.

[9] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, second edition, 2006.

[10] Alexander Aiken and David Gay. Barrier inference. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, page 342–354, New York, NY, USA, 1998. Association for Computing Machinery.

[11] Tim A Albring, Max Sagebaum, and Nicolas R Gauger. Efficient aerodynamic design using the discrete adjoint method in SU2. In *17th AIAA/ISSMO multidisciplinary analysis and optimization conference*, page 3518, 2016.

[12] Jonathan Aldrich, Craig Chambers, EminGun Sirer, and Susan Eggers. Static analyses for eliminating unnecessary synchronization from Java programs. In Agostino Cortesi and Gilberto Filé, editors, *Static Analysis*, volume 1694 of *Lecture Notes in Computer Science*, pages 19–38. Springer Berlin Heidelberg, 1999.

[13] Lelac Almagor, Keith D Cooper, Alexander Grosul, Timothy J Harvey, Steven W Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. Finding effective compilation sequences. *ACM SIGPLAN Notices*, 39(7):231–239, 2004.

[14] AMD. Using the x86 open64 compiler suite. `https://developer.amd.com/wordpress/media/2012/10/open64.pdf`, 2012.

[15] Corinne Ancourt and François Irigoin. Scanning polyhedra with DO loops. In *Proc. of the 37th ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 39–50. ACM, 1991.

[16] Julian Andrej. Differentiating large-scale finite element applications with mfem, 2023.

[17] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 303–316. ACM, 2014.

[18] Guillaume Aupy, Julien Herrmann, Paul Hovland, and Yves Robert. Optimal multistage algorithm for adjoint computation. *SIAM Journal on Scientific Computing*, 38(3):C232–C255, 2016.

[19] Ammar Ahmad Awan, Hari Subramoni, and Dhabaleswar K. Panda. An in-depth performance characterization of cpu- and gpu-based dnn training on modern architectures. In *Proc. of the Machine Learning on HPC Environments*, MLHPC'17, pages 8:1–8:8, New York, NY, 2017. ACM.

[20] Andrew Ayers, Stuart de Jong, John Peyton, and Richard Schooler. Scalable cross-module optimization. *ACM SIGPLAN Notices*, 33(5):301–312, 1998.

[21] E. Ayguade, N. Copty, A. Duran, J. Hoeflinger, Yuan Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and Guansong Zhang. The design of OpenMP tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20(3):404–418, 2009.

[22] Riyadh Baghdadi, Ulysse Beaugnon, Albert Cohen, Tobias Grosser, Michael Kruse, Chandan Reddy, Sven Verdoolaege, Adam Betts, Alastair F. Donaldson, Jeroen Ketema, Javed Absar, Sven Van Haastregt, Alexey Kravets, Anton Lokhmotov, Robert David, and Elnar Hajiyev. PENCIL: A Platform-Neutral Compute Intermediate Language for Accelerator Programming. In *2015 Intl. Conf. on Parallel Architecture and Compilation (PACT)*, pages 138–149, October 2015.

[23] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 193–205, 2019.

[24] Lénaïc Bagnères, Oleksandr Zinenko, Stéphane Huot, and Cédric Bastoul. Opening Polyhedral Compiler's Black Box. In *Proc. of the 2016 Intl. Symp. on Code Generation and Optimization*, CGO 2016, pages 128–138, New York, NY, 2016. ACM.

[25] Nathan Baker, Frank Alexander, Timo Bremer, Aric Hagberg, Yannis Kevrekidis, Habib Najm, Manish Parashar, Abani Patra, James Sethian, Stefan Wild, Karen Willcox, and Steven Lee. Workshop report on basic research needs for scientific machine learning: Core technologies for artificial intelligence. *USDOE Office of Science (SC)*, 2 2019.

[26] Rajkishore Barik and Vivek Sarkar. Interprocedural load elimination for dynamic optimization of parallel programs. In *PACT*, pages 41–52, 2009.

[27] Rajkishore Barik, Jisheng Zhao, and Vivek Sarkar. Interprocedural strength reduction of critical sections in explicitly-parallel programs. In *PACT*, pages 29–40, 2013.

[28] Muthu Manikandan Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. A compiler framework for optimization of affine loop nests for GPGPUs. In *Proc. of the 22nd Intl. Conf. on Supercomputing*, ICS'08, pages 225–234, New York, NY, 2008. ACM.

[29] Cédric Bastoul. Code Generation in the Polyhedral Model Is Easier Than You Think. In *Proc. of the 13th Intl. Conf. on Parallel Architectures and Compilation Techniques*, PACT'04, pages 7–16, Washington, DC, 2004. IEEE Computer Society.

[30] Cédric Bastoul. Openscop: A specification and a library for data exchange in polyhedral compilation tools. Technical report, Paris-Sud University, 2011.

[31] Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey, 2018.

[32] Ulysse Beaugnon, Alexey Kravets, Sven van Haastregt, Riyadh Baghdadi, David Tweed, Javed Absar, and Anton Lokhmotov. Vobla: A vehicle for optimized basic linear algebra. In *Proc. of the 2014 SIGPLAN/SIGBED Conf. on Languages, Compilers and Tools for Embedded Systems*, LCTES'14, pages 115–124, New York, NY, 2014. ACM.

[33] David Beckingsale, Richard Hornung, Tom Scogland, and Arturo Vargas. Performance portable C++ programming with RAJA. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, pages 455–456, 2019.

[34] Bradley M Bell. CppAD: a package for C++ algorithmic differentiation. *Computational Infrastructure for Operations Research*, 57(10), 2012.

[35] Richard Bellman. A markovian decision process. In *Journal of Mathematics and Mechanics*, pages 679–684, 1957.

[36] Geoffrey Belter, E. R. Jessup, Ian Karlin, and Jeremy G. Siek. Automating the generation of composed linear algebra kernels. In *Proc. of the Conf. on High Performance Computing Networking, Storage and Analysis*, SC'09, pages 59:1–59:12, New York, NY, 2009. ACM.

[37] Tal Ben-Nun, Berke Ates, Alexandru Calotoiu, and Torsten Hoefler. Bridging control-centric and data-centric optimization. In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization*, CGO 2023, page 173–185, New York, NY, USA, 2023. Association for Computing Machinery.

[38] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. The Polyhedral Model Is More Widely Applicable Than You Think. In Rajiv Gupta, editor, *Compiler Construction*, number 6011 in Lecture Notes in Computer Science, pages 283–303. Springer, March 2010.

[39] Tim Besard, Valentin Churavy, Alan Edelman, and Bjorn De Sutter. Rapid software prototyping for heterogeneous and distributed platforms. *Advances in engineering software*, 132:29–46, June 2019.

[40] Tim Besard, Christophe Foket, and Bjorn De Sutter. Effective extensible programming: Unleashing Julia on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 2018.

[41] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017.

[42] Lorenz T Biegler, Omar Ghattas, Matthias Heinkenschloss, and Bart van Bloemen Waanders. Large-scale pde-constrained optimization: an introduction. In *Large-Scale PDE-Constrained Optimization*, pages 3–13. Springer, 2003.

[43] Christian Bischof, Alan Carle, George Corliss, Andreas Griewank, and Paul Hovland. Adifor–generating derivative codes from fortran programs. *Scientific Programming*, 1(1):11–29, 1992.

[44] Christian Bischof, Niels Guertler, Andreas Kowarz, and Andrea Walther. Parallel reverse mode automatic differentiation for OpenMP programs with ADOL-C. In Christian H. Bischof, H. Martin Bücker, Paul D. Hovland, Uwe Naumann, and J. Utke, editors, *Advances in Automatic Differentiation*, pages 163–173. Springer, 2008.

[45] Christian Bischof, Peyvand Khademi, Andrew Mauer, and Alan Carle. ADIFOR 2.0: Automatic differentiation of Fortran 77 programs. *IEEE Computational Science and Engineering*, 3(3):18–32, 1996.

[46] Christian Bischof, Bruno Lang, and Andre Vehreschild. Automatic differentiation for MATLAB programs. In *PAMM: Proceedings in Applied Mathematics and Mechanics*, volume 2, pages 50–53. Wiley Online Library, 2003.

[47] Christian H Bischof, Lucas Roh, and Andrew J Mauer-Oats. ADIC: an extensible automatic differentiation tool for ANSI-C. *Software: Practice and Experience*, 27(12):1427–1456, 1997.

[48] Johannes Blühdorn, Nicolas R. Gauger, and Matthias Kabel. AutoMat — automatic differentiation for generalized standard materials on GPUs, 2020.

[49] Hans-J. Boehm and Sarita V. Adve. Foundations of the C++ concurrency memory model. In *PLDI*, pages 68–78, 2008.

[50] Uday Bondhugula, Aravind Acharya, and Albert Cohen. The Pluto+ Algorithm: A Practical Approach for Parallelization and Locality Optimization of Affine Loop Nests. *ACM Trans. on Programming Languages and Systems*, 38(3):12:1–12:32, April 2016.

[51] Uday Bondhugula, Sanjeeb Dash, Oktay Gunluk, and Lakshminarayanan Renganarayanan. A model for fusion and code motion in an automatic parallelizing compiler. In *Parallel Architectures and Compilation Techniques (PACT), 2010 19th Intl. Conf. on*, pages 343–352. IEEE, 2010.

[52] Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *Proc. of the 29th ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 2008.

[53] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, and Skye Wanderman-Milne. JAX: composable transformations of Python+NumPy programs, 2018.

[54] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[55] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[56] Michael Broughton, Guillaume Verdon, Trevor McCourt, Antonio J Martinez, Jae Hyeon Yoo, Sergei V Isakov, Philip Massey, Murphy Yuezhen Niu, Ramin Halavati, Evan Peters, et al. TensorFlow quantum: A software framework for quantum machine learning. *arXiv preprint arXiv:2003.02989*, 2020.

[57] Jed Brown, Valeria Barra, Natalie Beams, Leila Ghaffari, Matthew Knepley, William Moses, Rezgar Shakeri, Karen Stengel, Jeremy L Thompson, and Junchao Zhang. Performance portable solid mechanics via matrix-free *p*-multigrid. *arXiv preprint arXiv:2204.01722*, 2022.

[58] H. M. Bücker, B. Lang, A. Rasch, C. H. Bischof, and D. an Mey. Explicit loop scheduling in OpenMP for parallel automatic differentiation. In J. N. Almhana and V. C. Bhavsar, editors, *Proceedings of the 16th Annual International Symposium on High Performance Computing Systems and Applications, Moncton, NB, Canada, June 16–19, 2002*, pages 121–126, Los Alamitos, CA, 2002. IEEE Computer Society Press.

[59] H. Martin Bücker, Bruno Lang, Dieter an Mey, and Christian H. Bischof. Bringing together automatic differentiation and OpenMP. In *Proceedings of the 15th ACM International Conference on Supercomputing, Sorrento, Italy, June 17–21, 2001*, pages 246–251, New York, 2001. ACM Press.

[60] Simon Byrne, Lucas C Wilcox, and Valentin Churavy. MPI.jl: Julia bindings for the Message Passing Interface. In *Proceedings of the JuliaCon Conferences*, volume 1, page 68, 2021.

[61] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Tomasz Czajkowski, Stephen D Brown, and Jason H Anderson. Legup: An open-source high-level synthesis tool for fpga-based processor/accelerator systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(2):24, 2013.

[62] J.I. Cardesa, L. Hascoët, and C. Airiau. Adjoint computations by algorithmic differentiation of a parallel solver for time-dependent PDEs. *Journal of Computational Science*, 45:101155, 2020.

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

[64] Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. Habanero-Java: the new adventures of old X10. In *PPPJ*, pages 51–61, 2011.

[65] Rohit Chandra, Leo Dagum, David Kohr, Ramesh Menon, Dror Maydan, and Jeff McDonald. *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[66] Prasanth Chatarasi, Jun Shirako, Albert Cohen, and Vivek Sarkar. A unified approach to variable renaming for enhanced vectorization. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 1–20. Springer, 2018.

[67] Prasanth Chatarasi, Jun Shirako, and Vivek Sarkar. Polyhedral optimizations of explicitly parallel programs. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 213–226, 2015.

[68] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, 2009.

[69] Lorenzo Chelini, Andi Drebes, Oleksandr Zinenko, Albert Cohen, Nicolas Vasilache, Tobias Grosser, and Henk Corporaal. Progressive raising in multi-level ir. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 15–26. IEEE, 2021.

[70] Lorenzo Chelini, Oleksandr Zinenko, Tobias Grosser, and Henk Corporaal. Declarative loop tactics for domain-specific optimization. *ACM Trans. Archit. Code Optim.*, 16(4), December 2019.

[71] Chun Chen, Jacqueline Chame, and Mary Hall. Chill: A framework for composing high-level loop transformations. Technical report, Technical Report 08-897, U. of Southern California, 2008.

[72] Tian Qi Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. Neural ordinary differential equations. In *Advances in neural information processing systems*, pages 6571–6583, 2018.

[73] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR*, abs/1512.01274, 2015.

[74] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symp. on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, Carlsbad, CA, 2018. USENIX Association.

[75] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 3389–3400. Curran Associates, Inc., 2018.

[76] Max Christoff. Chrome just got faster with profile guided optimization, 2020.

[77] Valentin Churavy, Dilum Aluthge, Lucas C Wilcox, Simon Byrne, Maciej Waruszewski, Ali Ramadhan, Meredith, Simeon Schaub, James Schloss, Julian Samaroo, Jake Bolewski, Charles Kawczynski, Jeremy E Kozdon, Jinguo Liu, Oliver Schulz, Oscar, Páll Haraldsson, Takafumi Arakaki, and Tim Besard. Juliagpu/kernelabstractions.jl: v0.8.0, March 2022.

[78] CIRCT Developers. CIRCT charter, 2020.

[79] Robert Cohn and P Geoffrey Lowney. Feedback directed optimization in compaq's compilation tools for alpha. In *2nd ACM Workshop on Feedback-Directed Optimization (FDO)*. Citeseer, 1999.

[80] R. Collobert, K. Kavukcuoglu, and C. Farabet. Implementing neural networks efficiently. In G. Montavon, G. Orr, and K-R. Muller, editors, *Neural Networks: Tricks of the Trade*. Springer, 2012.

[81] Edoardo Conti, Vashisht Madhavan, Felipe Petroski Such, Joel Lehman, Kenneth Stanley, and Jeff Clune. Improving exploration in evolution strategies for deep reinforcement learning via a population of novelty-seeking agents. In *Advances in Neural Information Processing Systems*, pages 5032–5043, 2018.

[82] LLVM Contributors. LLVM language reference manual, 2023.

[83] TensorFlow Contributors. Create an op : TensorFlow core, 2020.

[84] Torch Contributors. Extending torchscript with custom C operators, 2020.

[85] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, 1977.

[86] Chris Cummins, Bram Wasti, Jiadong Guo, Brandon Cui, Jason Ansel, Sahir Gomez, Somya Jain, Jia Liu, Olivier Teytaud, Benoit Steiner, et al. Compilergym: Robust, performant compiler optimization environments for ai research. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 92–105. IEEE, 2022.

[87] Marco F Cusumano-Towner, Feras A Saad, Alexander K Lew, and Vikash K Mansinghka. Gen: a general-purpose probabilistic programming system with programmable inference. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 221–236, 2019.

[88] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.

[89] Alain Darte and Robert Schreiber. A linear-time algorithm for optimal barrier placement. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, page 26–35, New York, NY, USA, 2005. Association for Computing Machinery.

[90] Filipe de Avila Belbute-Peres, Kevin Smith, Kelsey Allen, Josh Tenenbaum, and J. Zico Kolter. End-to-end differentiable physics for learning and control. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 7178–7189. Curran Associates, Inc., 2018.

[91] Jonas Degrave, Michiel Hermans, Joni Dambre, and Francis Wyffels. A differentiable physics engine for deep learning in robotics. *Frontiers in neurorobotics*, 13:6, 2019.

[92] Gregory Diamos, Andrew Kerr, Sudhakar Yalamanchili, and Nathan Clark. Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 353–364. IEEE, 2010.

[93] Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. Type-based alias analysis. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, page 106–117, New York, NY, USA, 1998. Association for Computing Machinery.

[94] Johannes Doerfert, Jose Manuel Monsalve Diaz, and Hal Finkel. The tregion interface and compiler optimizations for openmp target regions. In Xing Fan, Bronis R. de Supinski, Oliver Sinnen, and Nasser Giacaman, editors, *OpenMP: Conquering the Full Hardware Spectrum - 15th International Workshop on OpenMP, IWOMP 2019, Auckland, New Zealand, September 11-13, 2019, Proceedings*, volume 11718 of *Lecture Notes in Computer Science*, pages 153–167. Springer, 2019.

[95] Johannes Doerfert and Hal Finkel. Compiler optimizations for OpenMP. In *International Workshop on OpenMP*, pages 113–127. Springer, 2018.

[96] Johannes Doerfert and Hal Finkel. Compiler optimizations for parallel programs. In Mary W. Hall and Hari Sundar, editors, *Languages and Compilers for Parallel Computing - 31st International Workshop, LCPC 2018, Salt Lake City, UT, USA, October 9-11, 2018, Revised Selected Papers*, volume 11882 of *Lecture Notes in Computer Science*, pages 112–119. Springer, 2018.

[97] Johannes Doerfert, Brian Homerding, and Hal Finkel. Performance exploration through optimistic static program annotations. In Michèle Weiland, Guido Juckeland, Carsten Trinitis, and Ponnuswamy Sadayappan, editors, *High Performance Computing*, Cham, 2019. Springer International Publishing.

[98] Johannes Doerfert, Kevin Streit, Sebastian Hack, and Zino Benaissa. Polly's polyhedral scheduling in the presence of reductions. *CoRR*, abs/1505.07716, 2015.

[99] Andi Drebes. Teckyl: An MLIR frontend for tensor operations, 2020.

[100] Andi Drebes, Lorenzo Chelini, Oleksandr Zinenko, Albert Cohen, Henk Corporaal, Tobias Grosser, Kanishkan Vadivel, and Nicolas Vasilache. TC-CIM: Empowering tensor comprehensions for computing-in-memory. In *IMPACT 2020-10th International Workshop on Polyhedral Compilation Techniques*, 2020.

[101] Aleksandr Drozd. Benchmarker. Online GitHub repository: `https://github.com/undertherain/benchmarker/`, commit `e1f22da320b0c7384cbd2f4df50255c7c2fa6b9d`, 2021.

[102] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra. From CUDA to OpenCL: Towards a performance-portable solution for multi-platform gpu programming. *Parallel Computing*, 38(8):391–407, 2012.

[103] Wei Du, Renato Ferreira, and Gagan Agrawal. Compiler support for exploiting coarse-grained pipelined parallelism. In *SC*, pages 8–21, 2003.

[104] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7), 2011.

[105] H Carter Edwards, Christian R Trott, and Daniel Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202–3216, 2014.

[106] Christine Eisenbeis and Jean-Claude Sogno. A general algorithm for data dependence analysis. In *Proceedings of the 6th International Conference on Supercomputing*, ICS '92, page 292–302, New York, NY, USA, 1992. Association for Computing Machinery.

[107] Venmugil Elango, Norm Rubin, Mahesh Ravishankar, Hariharan Sandanagobalane, and Vinod Grover. Diesel: DSL for linear algebra and neural net computations on GPUs. In *Proc. of the 2nd ACM SIGPLAN Intl. Workshop on Machine Learning and Programming Languages*, MAPL 2018, pages 42–51, New York, NY, 2018. ACM.

[108] Conal Elliott. The simple essence of automatic differentiation. *Proc. ACM Program. Lang.*, 2(ICFP):1–29, July 2018.

[109] Hadi Esmaeilzadeh, Emily R. Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *38th Intl. Symp. on Computer Architecture (ISCA 2011), June 4-8, 2011, San Jose, CA*, pages 365–376, 2011.

[110] Paul Feautrier. Array expansion. In *ACM International Conference on Supercomputing 25th Anniversary Volume*, page 99–111, New York, NY, USA, 1988. Association for Computing Machinery.

[111] Paul Feautrier. Dataflow Analysis of Array and Scalar References. *Intl. Journal of Parallel Programming*, 20(1):23–53, 1991.

[112] Paul Feautrier. Some Efficient Solutions to the Affine Scheduling Problem. Part II. Multidimensional Time. *Intl. Journal of Parallel Programming*, 21(6):389–420, 1992.

[113] Paul Feautrier and Christian Lengauer. Polyhedron Model. In David Padua, editor, *Encyclopedia of Parallel Computing*, pages 1581–1592. Springer, 2011.

[114] Martin Feinberg. Chemical reaction network structure and the stability of complex isothermal reactors—i. the deficiency zero and deficiency one theorems. *Chemical engineering science*, 42(10):2229–2268, 1987.

[115] Rich Felker and MUSL Developers. Musl libc, 2020.

[116] Mingdong Feng and Charles E. Leiserson. Efficient detection of determinacy races in Cilk programs. *Theory of Computing Systems*, 32(3):301–326, 1999.

[117] Yu Feng, Man-Yat Chu, Uroš Seljak, and Patrick McDonald. Fastpm: a new scheme for fast simulations of dark matter and haloes. *Monthly Notices of the Royal Astronomical Society*, 463(3):2273–2286, 2016.

[118] Michael Förster. *Algorithmic Differentiation of Pragma-Defined Parallel Regions: Differentiating Computer Programs Containing OpenMP*. PhD thesis, RWTH Aachen, 2014.

[119] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research*, 13:2171–2175, jul 2012.

[120] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, page 475–488, USA, 2019. USENIX Association.

[121] Basilio B. Fraguela, Ganesh Bikshandi, Jia Guo, María J. Garzarán, David Padua, and Christoph von Praun. Optimization techniques for efficient HTA programs. *Parallel Computing*, 38(9):465 – 484, 2012.

[122] Matteo Frigo and Steven G Johnson. FFTW: An adaptive software architecture for the FFT. In *Acoustics, Speech and Signal Processing, 1998. Proc. of the 1998 IEEE Intl. Conf. on*, volume 3, pages 1381–1384. IEEE, 1998.

[123] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 212–223, 1998.

[124] Fujitsu, 2021.

[125] Fujitsu, 2022.

[126] Grigori Fursin, Yuriy Kashnikov, Abdul Wahid Memon, Zbigniew Chamski, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Bilha Mendelson, Ayal Zaks, Eric Courtois, et al. Milepost gcc: Machine learning enabled self-tuning compiler. *International journal of parallel programming*, 39(3):296–327, 2011.

[127] Roman Gareev, Tobias Grosser, and Michael Kruse. High-performance generalized tensor operations: A compiler-oriented approach. *ACM Trans. Archit. Code Optim.*, 15(3), September 2018.

[128] Hong Ge, Kai Xu, and Zoubin Ghahramani. Turing: a language for flexible probabilistic inference. In *International Conference on Artificial Intelligence and Statistics, AISTATS 2018, 9-11 April 2018, Playa Blanca, Lanzarote, Canary Islands, Spain*, pages 1682–1690, 2018.

[129] Giorgis Georgakoudis, Johannes Doerfert, Ignacio Laguna, and Thomas R. W. Scogland. Faros: A framework to analyze openmp compilation through benchmarking and compiler optimization analysis. In Kent Milfeld, Bronis R. de Supinski, Lars Koesterke, and Jannis Klinkenberg, editors, *OpenMP: Portable Multi-Level Parallelism on Modern Systems*, pages 3–17, Cham, 2020. Springer International Publishing.

[130] Roger Ghanem, David Higdon, and Houman Owhadi. *Handbook of uncertainty quantification*, volume 6. Springer, 2017.

[131] Omar Ghattas and Karen Willcox. Learning physics-based models from data: perspectives from inverse problems and model reduction. *Acta Numerica*, 30:445–554, 2021.

[132] R. Giering, T. Kaminski, and T. Slawig. Generating efficient derivative code with TAF: Adjoint and tangent linear Euler flow around an airfoil. *Future Generation Computer Systems*, 21(8):1345–1355, 2005.

[133] Ralf Giering, Thomas Kaminski, Ricardo Todling, Ronald Errico, Ronald Gelaro, and Nathan Winslow. Tangent linear and adjoint versions of NASA/GMAO's Fortran 90 global weather forecast model. In H. Martin Bücker, George F. Corliss, Paul D. Hovland, Uwe Naumann, and Boyana Norris, editors, *Automatic Differentiation: Applications, Theory, and Implementations*, pages 275–284. Springer, 2005.

[134] Ralf Giering, Thomas Kaminski, Ricardo Todling, Ronald Errico, Ronald Gelaro, and Nathan Winslow. Tangent linear and adjoint versions of NASA/GMAO's Fortran 90 global weather forecast model. In *Automatic Differentiation: Applications, Theory, and Implementations*, pages 275–284. Springer, 2006.

[135] Michael B Giles and Niles A Pierce. An introduction to the adjoint approach to design. *Flow, turbulence and combustion*, 65(3):393–415, 2000.

[136] Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. Semi-Automatic Composition of Loop Transformations for Deep Parallelism and Memory Hierarchies. *Intl. Journal of Parallel Programming*, 34(3):261–317, July 2006.

[137] T. Glek and Jan Hubicka. Optimizing real world applications with GCC link time optimization. *CoRR*, abs/1010.2196, 2010.

[138] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1st edition, 1989.

[139] David E Goldberg. *Genetic algorithms*. Pearson Education India, 2006.

[140] XLA: Domain-specific compiler for linear algebra to optimizes tensorflow computations. `https://www.tensorflow.org/performance/xla`, 2017. Commit 0f1b88a.

[141] Priya Goyal, Piotr Dollár, Ross B. Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch SGD: training ImageNet in 1 hour. *CoRR*, abs/1706.02677, 2017.

[142] Markus Grabner, Thomas Pock, Tobias Gross, and Bernhard Kainz. Automatic differentiation for GPU-accelerated 2D/3D registration. In Christian H. Bischof, H. Martin Bücker, Paul D. Hovland, Uwe Naumann, and J. Utke, editors, *Advances in Automatic Differentiation*, pages 259–269. Springer, 2008.

[143] Joseph Greener. Differential molecular simulation with molly.jl, 2023.

[144] Felix Gremse, Andreas Höfter, Lukas Razik, Fabian Kiessling, and Uwe Naumann. GPU-accelerated adjoint algorithmic differentiation. *Computer physics communications*, 200:300–311, 2016.

[145] Andreas Griewank, David Juedes, and Jean Utke. Algorithm 755: ADOL-C: a package for the automatic differentiation of algorithms written in C/C++. *ACM Transactions on Mathematical Software (TOMS)*, 22(2):131–167, 1996.

[146] Andreas Griewank and Andrea Walther. Algorithm 799: Revolve: An implementation of checkpointing for the reverse or adjoint mode of computational differentiation. *ACM Trans. Math. Softw.*, 26(1):19–45, March 2000.

[147] Andreas Griewank and Andrea Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Society for Industrial and Applied Mathematics, USA, second edition, 2008.

[148] William Gropp, William D Gropp, Ewing Lusk, Anthony Skjellum, and Argonne Distinguished Fellow Emeritus Ewing Lusk. *Using MPI: portable parallel programming with the message-passing interface*, volume 1. MIT Press, 1999.

[149] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. Polly-performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(04):1250010, 2012.

[150] Tobias Grosser, Jagannathan Ramanujam, Louis-Noel Pouchet, Ponnuswamy Sadayappan, and Sebastian Pop. Optimistic delinearization of parametrically sized arrays. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 351–360, 2015.

[151] Tobias Grosser, Sven Verdoolaege, and Albert Cohen. Polyhedral AST generation is more than scanning polyhedra. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 37(4):1–50, 2015.

[152] Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Größlinger, and Louis-Noël Pouchet. Polly-polyhedral optimization in LLVM. In *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*, volume 2011, page 1, 2011.

[153] Dirk Grunwald and Harini Srinivasan. Data flow equations for explicitly parallel programs. In *PPoPP*, pages 159–168, 1993.

[154] Tobias Gysi, Christoph Müller, Oleksandr Zinenko, Stephan Herhut, Eddie Davis, Tobias Wicky, Oliver Fuhrer, Torsten Hoefler, and Tobias Grosser. Domain-specific multi-level ir rewriting for gpu: The open earth compiler for gpu-accelerated climate simulation. *ACM Trans. Archit. Code Optim.*, 18(4), sep 2021.

[155] Ameer Haj-Ali, Nesreen K Ahmed, Ted Willke, Sophia Shao, Krste Asanovic, and Ion Stoica. Learning to vectorize using deep reinforcement learning. In *Workshop on ML for Systems at NeurIPS*, December 2019.

[156] Ameer Haj-Ali, Nesreen K Ahmed, Ted Willke, Sophia Shao, Krste Asanovic, and Ion Stoica. Neurovectorizer: End-to-end vectorization with deep reinforcement learning. *International Symposium on Code Generation and Optimization (CGO)*, February 2020.

[157] Ameer Haj-Ali, Nesreen K. Ahmed, Ted Willke, Joseph Gonzalez, Krste Asanovic, and Ion Stoica. A view on deep reinforcement learning in system optimization. *arXiv preprint arXiv:1908.01275*, 2019.

[158] Mary Wolcott Hall. *Managing interprocedural optimization*. PhD thesis, Rice University, 1991.

[159] Hwansoo Han, Chau-Wen Tseng, and Pete Keleher. Eliminating barrier synchronization for compiler-parallelized codes on software dsms. *International journal of parallel programming*, 26(5):591–612, 1998.

[160] Ruobing Han, Jaewon Lee, Jaewoong Sim, and Hyesoon Kim. COX: CUDA on X86 by exposing warp-level functions to CPUs. *ACM Trans. Archit. Code Optim.*, jul 2022.

[161] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, Hiroaki Takada, and Katsuya Ishii. CHstone: A benchmark program suite for practical c-based high-level synthesis. In *Circuits and Systems, 2008. ISCAS 2008. IEEE International Symposium on*, pages 1192–1195, 2008.

[162] Mark Harris et al. Optimizing parallel reduction in cuda. *Nvidia developer technology*, 2(4):70, 2007.

[163] L. Hascoët and V. Pascual. The Tapenade Automatic Differentiation tool: Principles, Model, and Specification. *ACM Transactions On Mathematical Software*, 39(3), 2013.

[164] Laurent Hascoët, Uwe Naumann, and Valérie Pascual. "to be recorded" analysis in reverse-mode automatic differentiation. *Future Generation Computer Systems*, 21(8):1401–1417, 2005.

[165] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[166] Robert Hecht-Nielsen. Theory of the backpropagation neural network. In *Neural networks for perception*, pages 65–93. Elsevier, 1992.

[167] Patrick Heimbach and Sarah Williamson. Dj4earth: Oceans, ice sheets, adjoints, and ad, 2023.

[168] John Hennessy. The future of computing. Google I/O presentation, `https://www.youtube.com/watch?v=Azt8Nc-mtKM`, May 2018.

[169] J. A. Herdman, W. P. Gaudin, O. Perks, D. A. Beckingsale, A. C. Mallinson, and S. A. Jarvis. Achieving portability and performance through OpenACC. In *2014 First Workshop on Accelerator Programming using Directives*, pages 19–26, 2014.

[170] C. A. R. Hoare. Algorithm 64: Quicksort. *CACM*, 4(7):321, 1961.

[171] Robin J Hogan. Fast reverse-mode automatic differentiation using expression templates in C++. *ACM Transactions on Mathematical Software (TOMS)*, 40(4):1–16, 2014.

[172] Justin Holewinski. PTX back-end: GPU programming with LLVM. In *The Ohio State University. LLVM Developer's Meeting. November*, volume 18, 2011.

[173] Chuntao Hong, Dehao Chen, Wenguang Chen, Weimin Zheng, and Haibo Lin. MapCG: Writing parallel program portable between CPU and GPU. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, page 217–226, New York, NY, USA, 2010. Association for Computing Machinery.

[174] Yuanming Hu, Luke Anderson, Tzu-Mao Li, Qi Sun, Nathan Carr, Jonathan Ragan-Kelley, and Frédo Durand. DiffTaichi: Differentiable programming for physical simulation. In *International Conference on Learning Representations*, 2020.

[175] Qijing Huang, Ameer Haj-Ali, William Moses, John Xiang, Ion Stoica, Krste Asanovic, and John Wawrzynek. Autophase: Compiler phase-ordering for HLS with deep reinforcement learning. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 308–308. IEEE, 2019.

[176] Qijing Huang, Ruolong Lian, Andrew Canis, Jongsok Choi, Ryan Xi, Stephen Brown, and Jason Anderson. The effect of compiler optimizations on high-level synthesis for fpgas. In *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on*, pages 89–96. IEEE, 2013.

[177] Qijing Huang, Ruolong Lian, Andrew Canis, Jongsok Choi, Ryan Xi, Nazanin Calagar, Stephen Brown, and Jason Anderson. The effect of compiler optimizations on high-level synthesis-generated hardware. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 8(3):14, 2015.

[178] Alexander Hück, Joachim Protze, Jan-Patrick Lehr, Christian Terboven, Christian Bischof, and Matthias S Müller. Towards compiler-aided correctness checking of adjoint mpi applications. In *2020 IEEE/ACM 4th International Workshop on Software Correctness for HPC Applications (Correctness)*, pages 40–48. IEEE, 2020.

[179] Jan Hückelheim and Johannes Doerfert. Spray: Sparse reductions of arrays in openmp. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 475–484. IEEE, 2021.

[180] Jan Hückelheim, Paul Hovland, Michelle Mills Strout, and Jens-Dominik Müller. Reverse-mode algorithmic differentiation of an OpenMP-parallel compressible flow solver. *The International Journal of High Performance Computing Applications*, 33(1):140–154, 2019.

[181] Jan Hückelheim, Navjot Kukreja, Sri Hari Krishna Narayanan, Fabio Luporini, Gerard Gorman, and Paul Hovland. Automatic differentiation for adjoint stencil loops. In *Proceedings of the 48th International Conference on Parallel Processing*, ICPP 2019, New York, NY, USA, 2019. Association for Computing Machinery.

[182] Jan Christian Hückelheim and Laurent Hascoët. Source-to-source automatic differentiation of OpenMP parallel loops. *In Review*, 2021.

[183] J.C. Hückelheim, P.D. Hovland, M.M. Strout, and J.-D. Müller. Parallelizable adjoint stencil computations using transposed forward-mode algorithmic differentiation. *Optimization Methods and Software*, 33(4-6):672–693, 2018.

[184] Michael Innes. Don't unroll adjoint: differentiating ssa-form programs. *arXiv preprint arXiv:1810.07951*, 2018.

[185] Mike Innes. Sense & sensitivities: The path to general-purpose algorithmic differentiation. In *Proceedings of Machine Learning and Systems 2020*, pages 58–69. ACM, 2020.

[186] Mike Innes, Alan Edelman, Keno Fischer, Chris Rackauckas, Elliot Saba, Viral B Shah, and Will Tebbutt. A differentiable programming system to bridge machine learning and scientific computing, 2019.

[187] Intel. Intel High-Level Synthesis Compiler, 2019.

[188] Intel. Oneapi deep neural network library (OneDNN), 2022.

[189] Intel, Inteon, and Codeplay. Sycl polygeist fork, 2023.

[190] Intel Corporation. *Intel Cilk Plus Application Binary Interface Specification*, 2010. Document Number: 324512-001US. Available from `https://software.intel.com/sites/products/cilk-plus/cilk_plus_abi.pdf`.

[191] Intel Corporation. *Intel Cilk Plus Language Specification*, 2010. Document Number: 324396-001US. Available from `http://software.intel.com/sites/products/cilk-plus/cilk_plus_language_specification.pdf`.

[192] Intel Corporation. Cilk Plus/LLVM. Available from `http://cilkplus.github.io/`, 2013.

[193] Intel Corporation. *Intel C++ Compiler 16.0 User and Reference Guide*, 2015.

[194] Intel Corporation. Intel Cilk Plus samples. Available from `https://software.intel.com/en-us/code-samples/intel-compiler/intel-compiler-features/intelcilkplus`, 2016.

[195] François Irigoin and Remi Triolet. Supernode partitioning. In *Proc. of the 15th ACM SIGPLAN-SIGACT symp. on Principles of programming languages*, pages 319–329. ACM, 1988.

[196] Pekka Jääskeläinen, Carlos Sánchez de La Lama, Erik Schnetter, Kalle Raiskila, Jarmo Takala, and Heikki Berg. pocl: A performance-portable OpenCL implementation. *International Journal of Parallel Programming*, 43(5):752–785, 2015.

[197] Teresa Johnson, Mehdi Amini, and Xinliang David Li. ThinLTO: scalable and incremental LTO. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO '17, pages 111–121. IEEE Press, February 2017.

[198] Pramod G. Joisha, Robert S. Schreiber, Prithviraj Banerjee, Hans J. Boehm, and Dhruva R. Chakrabarti. A technique for the effective and automatic reuse of classical compiler optimizations on multithreaded code. In *POPL*, pages 623–636, 2011.

[199] Herbert Jordan, Simone Pellegrini, Peter Thoman, Klaus Kofler, and Thomas Fahringer. INSPIRE: The Insieme parallel intermediate representation. In *PACT*, pages 7–18, 2013.

[200] Cijo Jose, Moustpaha Cisse, and François Fleuret. Kronecker recurrent units. *CoRR*, abs/1705.10142, 2017.

[201] Norman P. Jouppi et al. In-datacenter performance analysis of a tensor processing unit. In *Proc. of the 44th Intl. Symp. on Computer Architecture, ISCA 2017, Toronto, ON, Canada, June 24-28, 2017*, pages 1–12, 2017.

[202] Pierre Jouvelot and Babak Dehbonei. A unified semantic approach for the vectorization and parallelization of generalized reductions. In *Proceedings of the 3rd International Conference on Supercomputing*, ICS '89, page 186–194, New York, NY, USA, 1989. Association for Computing Machinery.

[203] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. In *Reinforcement learning: A survey*, volume 4, pages 237–285, 1996.

[204] Ian Karlin. Lulesh programming model and performance ports overview. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2012.

[205] Ian Karlin, Jeff Keasler, and Rob Neely. LULESH 2.0 Updates and Changes, number = LLNL-TR-641973. Technical report, Lawrence Livermore National Lab, August 2013.

[206] Ralf Karrenberg and Sebastian Hack. Improving performance of OpenCL on CPUs. In Michael O'Boyle, editor, *Compiler Construction*, pages 1–20, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[207] James Kennedy. Particle swarm optimization. *Encyclopedia of machine learning*, pages 760–766, 2010.

[208] Ken Kennedy and John R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, 2002.

[209] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, Inc., second edition, 1988.

[210] Andrew Kerr, Duane Merrill, Julien Demouth, and John Tran. CUTLASS: Fast linear algebra in CUDA C++. `https://devblogs.nvidia.com/cutlass-linear-algebra-cuda`, December 2017.

[211] D. Khaldi, P. Jouvelot, C. Ancourt, and F. Irigoin. SPIRE, a sequential to parallel intermediate representation extension. Technical report, Technical Report CRI/A-487, MINES ParisTech, 2012.

[212] Dounia Khaldi, Pierre Jouvelot, François Irigoin, Corinne Ancourt, and Barbara Chapman. LLVM parallel intermediate representation: Design and evaluation using OpenSHMEM communications. In *LLVM*, pages 2:1–2:8, 2015.

[213] Gary A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, pages 194–206, New York, NY, USA, 1973. ACM.

[214] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proc. ACM Program. Lang.*, 1(OOPSLA):77:1–77:29, October 2017.

[215] Fredrik Kjolstad, Shoaib Kamil, Jonathan Ragan-Kelley, David I. W. Levin, Shinjiro Sueda, Desai Chen, Etienne Vouga, Danny M. Kaufman, Gurtej Kanwar, Wojciech Matusik, and Saman Amarasinghe. Simit: A language for physical simulation. *ACM Trans. Graph.*, 35(2):20:1–20:21, March 2016.

[216] Andreas Klöckner. Loo.py: Transformation-based code generation for GPUs and CPUs. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming (ARRAY'14)*, page 82–87, New York, NY, USA, 2014. Association for Computing Machinery.

[217] Jens Knoop, Bernhard Steffen, and Jürgen Vollmer. Parallelism for free: Efficient and optimal bitvector analyses for parallel programs. *ACM TOPLAS*, pages 268–299, 1996.

[218] Dmitrii Kochkov, Jamie A. Smith, Ayya Alieva, Qing Wang, Michael P. Brenner, and Stephan Hoyer. Machine learning accelerated computational fluid dynamics, 2021.

[219] Konrad Komisarczyk, Lorenzo Chelini, Kanishkan Vadivel, Roel Jordans, and Henk Corporaal. PET-to-MLIR: A polyhedral front-end for mlir. In *2020 23rd Euromicro Conference on Digital System Design (DSD)*, pages 551–556. IEEE, 2020.

[220] Martin Kong and Louis-Noël Pouchet. A performance vocabulary for affine loop transformations. *CoRR*, abs/1811.06043, 2018.

[221] Martin Kong, Richard Veras, Kevin Stock, Franz Franchetti, Louis-Noël Pouchet, and P. Sadayappan. When polyhedral transformations meet SIMD code generation. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation, PLDI'13, Seattle, WA*, pages 127–138, June 2013.

[222] Maria Kotsifakou, Prakalp Srivastava, Matthew D. Sinclair, Rakesh Komuravelli, Vikram Adve, and Sarita Adve. HPVM: Heterogeneous parallel virtual machine. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '18, page 68–80, New York, NY, USA, 2018. Association for Computing Machinery.

[223] Michael Kruse and Tobias Grosser. DeLICM: scalar dependence removal at zero memory cost. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, pages 241–253, 2018.

[224] Navjot Kukreja, Jan Hückelheim, Michael Lange, Mathias Louboutin, Andrea Walther, Simon W Funke, and Gerard Gorman. High-level python abstractions for optimal checkpointing in inversion problems. *arXiv preprint arXiv:1802.02474*, 2018.

[225] Sameer Kulkarni and John Cavazos. Mitigating the compiler optimization phase-ordering problem using machine learning. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, 2012.

[226] Los Alamos National Laboratory. Kitsune compiler, 2023.

[227] E. Larour, J. Utke, A. Bovin, M. Morlighem, and G. Perez. An approach to computing discrete adjoints for MPI-parallelized models applied to ice sheet system model 4.11. *Geoscientific Model Development*, 9(11):3907–3918, 2016.

[228] Rasmus Wriedt Larsen and Troels Henriksen. Strategies for regular segmented reductions on GPU. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing*, pages 42–52, 2017.

[229] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14, 2021.

[230] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[231] Yann LeCun, Bernhard E. Boser, John S. Denker, Donnie Henderson, Richard E. Howard, Wayne E. Hubbard, and Lawrence D. Jackel. Handwritten digit recognition with a back-propagation network. In *Advances in Neural Information Processing Systems 2, [NIPS Conf., Denver, CO, November 27-30, 1989]*, pages 396–404, 1989.

[232] I-Ting Angelina Lee, Charles E. Leiserson, Tao B. Schardl, Zhunping Zhang, and Jim Sukha. On-the-fly pipeline parallelism. *ACM TOPC*, 2(3):17:1–17:42, 2015.

[233] Jaejin Lee, Samuel P. Midkiff, and David A. Padua. Concurrent static single assignment form and constant propagation for explicitly parallel programs. In *LCPC*, pages 114–130, 1997.

[234] Charles E. Leiserson. The Cilk++ concurrency platform. *Journal of Supercomputing*, 51(3):244–257, 2010.

[235] David Xinliang Li, Raksit Ashok, and Robert Hundt. Lightweight feedback-directed cross-module optimization. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, 2010.

[236] Shengtai Li and Linda Petzold. Adjoint sensitivity analysis for time-dependent partial differential equations with adaptive mesh refinement. *Journal of Computational Physics*, 198(1):310–325, 2004.

[237] Tzu-Mao Li, Michaël Gharbi, Andrew Adams, Frédo Durand, and Jonathan Ragan-Kelley. Differentiable programming for image processing and deep learning in Halide. *ACM Trans. Graph. (Proc. SIGGRAPH)*, 37(4):139:1–139:13, 2018.

[238] Tzu-Mao Li, Michal Lukáč, Michaël Gharbi, and Jonathan Ragan-Kelley. Differentiable vector graphics rasterization for editing and learning. *ACM Transactions on Graphics (TOG)*, 39(6):1–15, 2020.

[239] Eric Liang, Richard Liaw, Philipp Moritz, Robert Nishihara, Roy Fox, Ken Goldberg, Joseph E Gonzalez, Michael I Jordan, and Ion Stoica. Rllib: Abstractions for distributed reinforcement learning. *arXiv preprint arXiv:1712.09381*, 2017.

[240] Amy W. Lim and Monica S. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, page 201–214, New York, NY, USA, 1997. Association for Computing Machinery.

[241] LLVM Contributors. OpenMP-aware optimizations. Online: `https://openmp.llvm.org/optimizations/OpenMPOpt.html`, 2021.

[242] LLVM Developer List. LLVMdev discussions on Intel OpenMP proposal. Available from `http://lists.llvm.org/pipermail/llvm-dev/2012-September/053861.html`, September 2012.

[243] LLVM Developer List. LLVMdev discussions on OpenCL SPIR proposal. Available from `http://lists.llvm.org/pipermail/llvm-dev/2012-September/053293.html`, September 2012.

[244] LLVM Developer List. LLVMdev Parallelization metadata and intrinsics in LLVM (for OpenMP, etc.). Available from `http://lists.llvm.org/pipermail/llvm-dev/2012-September/053792.html`, September 2012.

[245] LLVM Developer List. LLVMdev discussions on parallel IR. Available from `http://lists.llvm.org/pipermail/llvm-dev/2015-March/083314.html`, March 2015.

[246] LLVM Developers. LLVM Remarks System, 2020.

[247] LLVM Developers. Test-Suite Guide, 2020.

[248] LLVM Project. *LLVM's Analysis and Transform Passes*, 2015. Available from `http://llvm.org/docs/Passes.html`.

[249] Johannes Lotz, Klaus Leppkes, and Uwe Naumann. dco/c++-derivative code by overloading in C++. *Aachener Informatik Berichte (AIB-2011-06)*, 2011.

[250] Johannes Lotz, Uwe Naumann, Max Sagebaum, and Michel Schanen. Discrete adjoints of PETSc through dco/c++ and adjoint MPI. In Felix Wolf, Bernd Mohr, and Dieter an Mey, editors, *Euro-Par 2013 Parallel Processing*, pages 497–507, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[251] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, pages 716–727, 2012.

[252] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. GraphLab: A new framework for parallel machine learning. In *UAI*, pages 340–349, 2010.

[253] Mikel Luján, T. L. Freeman, and John R. Gurd. Oolala: An object oriented analysis and design of numerical linear algebra. In *Proc. of the 15th ACM SIGPLAN Conf. on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA'00, pages 229–252, New York, NY, 2000. ACM.

[254] Dougal Maclaurin, David Duvenaud, and Ryan Adams. Gradient-based hyperparameter optimization through reversible learning. In *International conference on machine learning*, pages 2113–2122. PMLR, 2015.

[255] Dougal Maclaurin, David Duvenaud, and Ryan P Adams. Autograd: Effortless gradients in numpy. In *ICML 2015 AutoML Workshop*, volume 238, 2015.

[256] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, pages 135–146, 2010.

[257] Joaquim RRA Martins and Andrew B Lambe. Multidisciplinary design optimization: a survey of architectures. *AIAA journal*, 51(9):2049–2075, 2013.

[258] Michael McCool, Arch D. Robison, and James Reinders. *Structured Parallel Programming: Patterns for Efficient Computation*. Elsevier Science, 2012.

[259] Simon McIntosh-Smith, James Price, Richard B Sessions, and Amaurys A Ibarra. High performance in silico virtual drug screening on many-core processors. *The International Journal of High Performance Computing Applications*, 29(2):119–134, 2015. PMID: 25972727.

[260] Benoit Meister, Nicolas Vasilache, David Wohlford, Muthu Manikandan Baskaran, Allen Leung, and Richard Lethin. *R-Stream Compiler*, pages 1756–1765. Springer, Boston, MA, 2011.

[261] Microsoft unveils project brainwave for real-time AI. `https://www.microsoft.com/en-us/research/blog/microsoft-unveils-project-brainwave`, August 2017.

[262] Samuel P. Midkiff and David A. Padua. Issues in the optimization of parallel programs. In *ICPP*, pages 105–113, 1990.

[263] József Mihalicza. How# includes affect build time in large systems. In *Proceedings of the 8th international conference on applied informatics (ICAI 2010)*, volume 2, 2010.

[264] MLIR Developers. MLIR affine dialect, 2020.

[265] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937, 2016.

[266] Simon Moll, Johannes Doerfert, and Sebastian Hack. Input space splitting for OpenCL. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, page 251–260, New York, NY, USA, 2016. Association for Computing Machinery.

[267] Sungdo Moon and Mary W Hall. Evaluation of predicated array data-flow analysis for automatic parallelization. *ACM SIGPLAN Notices*, 34(8):84–95, 1999.

[268] Sungdo Moon, Xinliang D Li, Robert Hundt, Dhruva R Chakrabarti, Luis A Lozano, Uma Srinivasan, and S-M Liu. Syzygy-a framework for scalable cross-module ipo. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 65–74. IEEE, 2004.

[269] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. Ray: A distributed framework for emerging {AI} applications. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 561–577, 2018.

[270] William S. Moses, Lorenzo Chelini, Ruizhe Zhao, and Oleksandr Zinenko. Polygeist: Raising C to polyhedral MLIR. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 45–59, 2021.

[271] William S. Moses and Valentin Churavy. Instead of rewriting foreign code for machine learning, automatically synthesize fast gradients. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS'20, Red Hook, NY, USA, 2020. Curran Associates Inc.

[272] William S. Moses, Valentin Churavy, Ludger Paehler, Jan Hückelheim, Sri Hari Krishna Narayanan, Michel Schanen, and Johannes Doerfert. Reverse-mode automatic differentiation and optimization of GPU kernels via Enzyme. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery.

[273] William S Moses, Sri Hari Krishna Narayanan, Ludger Paehler, Jan Churavy, Valentinnand Hückelheim, Michel Schanen, Johannes Doerfert, and Paul Hovland. Scalable automatic differentiation of multiple parallel paradigms through compiler augmentation. In *SC '22: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, New York, NY, USA, 2022. Association for Computing Machinery.

[274] William Steven Moses. How should compilers represent fork-join parallelism? Master's thesis, Massachusetts Institute of Technology, 2017.

[275] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[276] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. Automatically scheduling Halide image processing pipelines. *ACM Trans. on Graphics (TOG)*, 35(4):83:1–83:11, July 2016.

[277] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. Polymage: Automatic optimization for image processing pipelines. In *Proc. of the Twentieth Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'15, pages 429–443, New York, NY, 2015. ACM.

[278] J-D Müller and P Cusdin. On the performance of discrete adjoint cfd codes using automatic differentiation. *International journal for numerical methods in fluids*, 47(8-9):939–945, 2005.

[279] Jens-Dominik Müller, Jan Hueckelheim, and Orest Mykhaskiv. STAMPS: a finite-volume solver framework for adjoint codes derived with source-transformation AD. In *2018 Multidisciplinary Analysis and Optimization Conference*, page 2928, 2018.

[280] Erdal Mutlu, Ruiqin Tian, Bin Ren, Sriram Krishnamoorthy, Roberto Gioiosa, Jacques Pienaar, and Gokcen Kestor. Comet: A domain-specific compilation of high-performance computational chemistry. In *The 33rd Workshop on Languages and Compilers for Parallel Computing*, 2020.

[281] Sri Hari Krishna Narayanan, Boyana Norris, and Beata Winnicka. ADIC2: Development of a component source transformation system for differentiating C and C++. *Procedia Computer Science*, 1(1):1845–1853, 2010. ICCS 2010.

[282] Uwe Naumann. *The art of differentiating computer programs: an introduction to algorithmic differentiation*. SIAM, 2011.

[283] Uwe Naumann, Laurent Hascoët, Chris Hill, Paul Hovland, Jan Riehme, and Jean Utke. A framework for proving correctness of adjoint message-passing programs. In Alexey Lastovetsky, Tahar Kechadi, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 316–321, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[284] Angeles Navarro, Rafael Asenjo, Siham Tabik, and Calin Cascaval. Analytical modeling of pipeline parallelism. In *PACT*, pages 281–290, 2009.

[285] Robert H. B. Netzer and Barton P. Miller. What are race conditions? *ACM Letters on Programming Languages and Systems*, 1(1):74–88, 1992.

[286] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *SOSP*, pages 456–471, 2013.

[287] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. Deterministic Galois: On-demand, portable and parameterless. In *ASPLOS*, pages 499–512, 2014.

[288] Diego Novillo, Ron Unrau, and Jonathan Schaeffer. Concurrent SSA form in the presence of mutual exclusion. In *ICPP*, pages 356–364, 1998.

[289] npcomp developers. MLIR npcomp, 2020.

[290] Deploying deep neural networks with Nvidia TensorRT. `https://devblogs.nvidia.com/parallelforall/deploying-deep-learning-nvidia-tensorrt`, April 2017.

[291] Cosmin E Oancea and Lawrence Rauchwerger. Logical inference techniques for loop parallelization. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 509–520, 2012.

[292] M. O'Boyle and E. Stohr. Compile time barrier synchronization minimization. *IEEE Transactions on Parallel and Distributed Systems*, 13(6):529–543, 2002.

[293] OpenMP Architecture Review Board. *OpenMP Application Program Interface, Version 4.0*, July 2013. Available from `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`.

[294] Zhelong Pan and Rudolf Eigenmann. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 319–332. IEEE Computer Society, 2006.

[295] Valérie Pascual and Laurent Hascoët. Mixed-language automatic differentiation. *Optimization Methods and Software*, 33(4-6):1192–1206, 2018.

[296] PassiveLogic. Using enzyme autodiff with swift. `https://medium.com/passivelogic/using-enzyme-autodiff-with-swift-afa1bc2dc102`, 2021.

[297] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. In *NIPS 2017 Autodiff Workshop: The Future of Gradient-based Machine Learning Software and Techniques*, Long Beach, CA, December 2017.

[298] Adam Paszke, Daniel Johnson, David Duvenaud, Dimitrios Vytiniotis, Alexey Radul, Matthew Johnson, Jonathan Ragan-Kelley, and Dougal Maclaurin. Getting to the point. index sets and parallelism-preserving autodiff for pointful array programming. *arXiv preprint arXiv:2104.05372*, 2021.

[299] Atmn Patel, Shilei Tian, Johannes Doerfert, and Barbara Chapman. A virtual GPU as developer-friendly OpenMP offload target. In *50th International Conference on Parallel Processing Workshop*, ICPP Workshops '21, New York, NY, USA, 2021. Association for Computing Machinery.

[300] Morten Borup Petersen. A dynamically scheduled hls flow in mlir. Master's thesis, École Polytechnique Fédérale de Lausanne, 2022.

[301] Matt Pharr and William R Mark. ispc: A SPMD compiler for high-performance CPU programming. In *2012 Innovative Parallel Computing (InPar)*, pages 1–13. IEEE, 2012.

[302] Michael Piasecki and Nikolaos D Katopodes. Control of contaminant releases in rivers. i: Adjoint sensitivity analysis. *Journal of hydraulic engineering*, 123(6):486–492, 1997.

[303] PlaidML. `https://www.intel.ai/plaidml/#gs.bBu0cF8W`, 2018.

[304] PoCC. The polyhedral compiler collection, 2020. Online; accessed on December 2020.

[305] Andrei Poenaru, Simon Mcintosh-Smith, and Tom Lin. A performance analysis of modern parallel programming models using a compute-bound application. In *High Performance Computing - 36th International Conference, ISC High Performance 2021, Proceedings*, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), pages 332–350. Springer, June 2021.

[306] Antoniu Pop and Albert Cohen. Preserving high-level semantics of parallel programming annotations through the compilation flow of optimizing compilers. In *CPC*, 2010.

[307] Sebastian Pop, Albert Cohen, Cédric Bastoul, Sylvain Girbal, Georges-André Silber, and Nicolas Vasilache. Graphite: Polyhedral analyses and optimizations for gcc. In *Proceedings of the 2006 GCC Developers Summit*, page 2006, 2006.

[308] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, P. Sadayappan, and Nicolas Vasilache. Loop transformations: Convexity, pruning and optimization. In *38th ACM Symp. on Principles of Programming Languages (POPL)*, Austin, Texas, January 2011.

[309] Louis-Noël Pouchet and Tomofumi Yuki. Polybench/c 4.2.1.

[310] Louis-Noel Pouchet, Peng Zhang, P. Sadayappan, and Jason Cong. Polyhedral-based data reuse optimization for configurable computing. In *Proc. of the ACM/SIGDA Intl. Symp. on Field Programmable Gate Arrays*, FPGA'13, pages 29–38, New York, NY, 2013. ACM.

[311] Benoit Pradelle, Benoit Meister, Muthu Baskaran, Jonathan Springer, and Richard Lethin. Polyhedral optimization of tensorflow computation graphs. In *6th Workshop on Extreme-scale Programming Tools (ESPT, associated with SC'17)*, November 2017.

[312] William Pugh. Fixing the Java memory model. In *JAVA*, pages 89–98, 1999.

[313] William Pugh and David Wonnacott. Static Analysis of Upper and Lower Bounds on Dependences and Parallelism. *ACM Trans. Program. Lang. Syst.*, 16(4):1248–1278, July 1994.

[314] Markus Püschel, José M. F. Moura, Bryan Singer, Jianxin Xiong, Jeremy Johnson, David Padua, Manuela Veloso, and Robert W. Johnson. Spiral: A generator for platform-adapted libraries of signal processing alogorithms. *The Intl. Journal of High Performance Computing Applications*, 18(1):21–45, 2004.

[315] PyTorch. Fast backward mode gradcheck. `https://pytorch.org/docs/stable/notes/gradcheck.html#fast-backward-mode-gradcheck`.

[316] Dan Quinlan. Rose: Compiler support for object-oriented frameworks. *Parallel processing letters*, 10(02n03):215–226, 2000.

[317] Christopher Rackauckas. The essential tools of scientific machine learning (scientific ml). *The Winnower*, 08 2019.

[318] Christopher Rackauckas, Yingbo Ma, Julius Martensen, Collin Warner, Kirill Zubov, Rohit Supekar, Dominic Skinner, and Ali Ramadhan. Universal differential equations for scientific machine learning. *arXiv preprint arXiv:2001.04385*, 2020.

[319] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Seattle, WA, June 2013.

[320] Harenome Razanajato, Cidric Bastoul, and Vincent Loechner. Lifting barriers using parallel polyhedral regions. In *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*, pages 338–347, 2017.

[321] Chandan Reddy, Michael Kruse, and Albert Cohen. Reduction drawing: Language constructs and polyhedral compilation for reductions on gpu. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, PACT '16, page 87–97, New York, NY, USA, 2016. Association for Computing Machinery.

[322] Steffen Rendle. Factorization machines. In *Proc. of the 2010 IEEE Intl. Conf. on Data Mining*, ICDM'10, pages 995–1000, Washington, DC, 2010. IEEE Computer Society.

[323] Nvidia Research. Cub documentation. `https://nvlabs.github.io/cub`. Version 1.8.0.

[324] Jarrett Revels, Tim Besard, Valentin Churavy, Bjorn De Sutter, and Juan Pablo Vielma. Dynamic automatic differentiation of GPU broadcast kernels. *CoRR*, abs/1810.08297, 2018.

[325] Jarrett Revels, Miles Lubin, and Theodore Papamarkou. Forward-mode automatic differentiation in Julia. *arXiv preprint arXiv:1607.07892*, 2016.

[326] Helge Rhodin. A PTX code generator for LLVM. *Oct*, 29:1–63, 2010.

[327] Paul K Romano and Benoit Forget. The OpenMC Monte Carlo particle transport code. *Annals of Nuclear Energy*, 51:274–281, 2013.

[328] Stéphane Ross, Geoffrey Gordon, and Drew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 627–635, 2011.

[329] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Summer Deng, Roman Dzhabarov, James Hegeman, Roman Levenstein, Bert Maher, Nadathur Satish, Jakob Olesen, Jongsoo Park, Artem Rakhov, and Misha Smelyanskiy. Glow: Graph lowering compiler techniques for neural networks. *CoRR*, abs/1805.00907, 2018.

[330] Erik Ruf. Effective synchronization removal for Java. In *PLDI*, pages 208–218, 2000.

[331] Radu Rugina and Martin C. Rinard. Pointer analysis for structured parallel programs. *TOPLAS*, pages 70–116, January 2003.

[332] Max Sagebaum, Tim Albring, and Nicolas R Gauger. High-performance derivative computations using codipack. *ACM Transactions on Mathematical Software (TOMS)*, 45(4):1–26, 2019.

[333] Max Sagebaum and Nicolas R Gauger. Medipack–message differentiation package, 2020.

[334] Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*, 2017.

[335] Caio Salvador Rohwedder, Nathan Henderson, João P. L. De Carvalho, Yufei Chen, and José Nelson Amaral. To pack or not to pack: A generalized packing analysis and transformation. In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization*, CGO 2023, page 14–27, New York, NY, USA, 2023. Association for Computing Machinery.

[336] Julian Samaroo, Valentin Churavy, Wiktor Phillips, Jason Barmparesos, Julia TagBot, Takafumi Arakaki, Stephan Antholzer, Alessandro, chriselrod, and Tim Besard. Juliagpu/amdgpu.jl: v0.2.6 for zenodo, April 2021.

[337] Vivek Sarkar. Analysis and optimization of explicitly parallel programs using the parallel program graph representation. In *LCPC*, pages 94–113, 1998.

[338] Vivek Sarkar and Barbara Simons. Parallel program graphs and their classification. In *LCPC*, pages 633–655, 1994.

[339] Mitsuhisa Sato, Yutaka Ishikawa, Hirofumi Tomita, Yuetsu Kodama, Tetsuya Odajima, Miwako Tsuji, Hisashi Yashiro, Masaki Aoki, Naoyuki Shida, Ikuo Miyoshi, Kouichi Hirai, Atsushi Furuya, Akira Asato, Kuniki Morita, and Toshiyuki Shimizu. Co-design for A64FX manycore processor and "Fugaku". In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2020.

[340] Michel Schanen, Uwe Naumann, Laurent Hascoët, and Jean Utke. Interpretative adjoints for numerical simulation codes using MPI. *Procedia Computer Science*, 1(1):1825–1833, 2010. ICCS 2010.

[341] Tao B. Schardl, Bradley C. Kuszmaul, I-Ting Angelina Lee, William M. Leiserson, and Charles E. Leiserson. The Cilkprof scalability profiler. In *SPAA*, pages 89–100, 2015.

[342] Tao B Schardl, I-Ting Angelina Lee, and Charles E Leiserson. Brief announcement: Open cilk. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, pages 351–353, 2018.

[343] Tao B. Schardl, William S. Moses, and Charles E. Leiserson. Tapir: Embedding fork-join parallelism into llvm's intermediate representation. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '17, pages 249–265, New York, NY, USA, 2017. ACM.

[344] Tao B Schardl, William S Moses, and Charles E Leiserson. Tapir: Embedding recursive fork-join parallelism into llvm's intermediate representation. *ACM Transactions on Parallel Computing (TOPC)*, 6(4):1–33, 2019.

[345] Robert Schenck, Ola Rønning, Troels Henriksen, and Cosmin E Oancea. Ad for an array language with nested parallelism. *arXiv preprint arXiv:2202.10297*, 2022.

[346] Adrian Schmitz, Julian Miller, Lukas Trümper, and Matthias S Müller. PPIR: Parallel pattern intermediate representation. In *2021 IEEE/ACM International Workshop on Hierarchical Parallelism for Exascale Computing (HiPar)*, pages 30–40. IEEE, 2021.

[347] Tapio Schneider, Shiwei Lan, Andrew Stuart, and João Teixeira. Earth System Modeling 2.0: A blueprint for models that learn from observations and targeted high-resolution simulations. *Geophysical Research Letters*, 44(24):12,396–12,417, 2017.

[348] Samuel Schoenholz and Ekin Dogus Cubuk. JAX md: a framework for differentiable physics. *Advances in Neural Information Processing Systems*, 33, 2020.

[349] Mike Schroepfer. Day 2 keynote. Facebook f8 presentation at McEnery Convention Center, San Jose, California, `https://developers.facebook.com/videos/f8-2018/f8-2018-day-2-keynote`, May 2018.

[350] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[351] Eric Schweitz. An MLIR dialect for high-level optimization of Fortran. In *2019 LLVM Developers Meeting*, 2019.

[352] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow, 2018.

[353] Jaewook Shin and Paul D Hovland. Comparison of two activity analyses for automatic differentiation: Context-sensitive flow-insensitive vs. context-insensitive flow-sensitive. In *Proceedings of the 2007 ACM symposium on Applied computing*, pages 1323–1329, 2007.

[354] Jun Shirako, Louis-Noël Pouchet, and Vivek Sarkar. Oil and water can mix: An integration of polyhedral and ast-based transformations. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 287–298. IEEE, 2014.

[355] Julian Shun and Guy E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *PPoPP*, pages 135–146, 2013.

[356] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. Brief announcement: the Problem Based Benchmark Suite. In *SPAA*, pages 68–70, 2012.

[357] Julian Shun, Laxman Dhulipala, and Guy E. Blelloch. Smaller and faster: Parallel processing of compressed graphs with Ligra+. In *DCC*, pages 403–412, 2015.

[358] Michael D. Smith. Overcoming the challenges to feedback-directed optimization (keynote talk). In *Proc. of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, DYNAMO'00, pages 1–11, New York, NY, 2000.

[359] Tyler Sorensen, Alastair F. Donaldson, Mark Batty, Ganesh Gopalakrishnan, and Zvonimir Rakamarić. Portable inter-workgroup barrier synchronisation for GPUs. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, page 39–58, New York, NY, USA, 2016. Association for Computing Machinery.

[360] Tyler Sorensen, Lucas F Salvador, Harmit Raval, Hugues Evrard, John Wickerson, Margaret Martonosi, and Alastair F Donaldson. Specifying and testing gpu workgroup progress models. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–30, 2021.

[361] Filip Srajer, Zuzana Kukelova, and Andrew Fitzgibbon. A benchmark of selected algorithmic differentiation tools on some problems in computer vision and machine learning. *Optimization Methods and Software*, 33(4-6):889–906, 2018.

[362] Harini Srinivasan and Dirk Grunwald. An efficient construction of parallel static single assignment form for structured parallel programs. Technical report, Technical Report CU-CS-564-91, University of Colorado at Boulder, 1991.

[363] Harini Srinivasan, James Hook, and Michael Wolfe. Static single assignment for explicitly parallel programs. In *POPL*, pages 260–272, 1993.

[364] Harini Srinivasan and Michael Wolfe. Analyzing programs with explicit parallelism. In *LCPC*, pages 405–419, 1991.

[365] Richard M Stallman et al. *Using and porting the GNU compiler collection*. Free Software Foundation, 1999.

[366] Richard M. Stallman and the GCC Developer Community. *Using the GNU Compiler Collection (for GCC version 6.1.0)*. Free Software Foundation, 2016.

[367] George Stelle, William S. Moses, Stephen L. Olivier, and Patrick McCormick. OpenMPIR: Implementing openmp tasks with tapir. In *Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC*, pages 3:1–3:12, New York, NY, USA, 2017. ACM.

[368] Mark Stephenson, Saman Amarasinghe, Martin Martin, and Una-May O'Reilly. Meta optimization: Improving compiler heuristics with machine learning. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, 2003.

[369] Rick Stevens, Valerie Taylor, Jeff Nichols, Arthur Barney Maccabe, Katherine Yelick, and David Brown. AI for science. Technical report, Argonne National Lab.(ANL), Argonne, IL (United States), 2020.

[370] John A. Stratton, Vinod Grover, Jaydeep Marathe, Bastiaan Aarts, Mike Murphy, Ziang Hu, and Wen-mei W. Hwu. Efficient compilation of fine-grained SPMD-threaded programs for multicore CPUs. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, page 111–119, New York, NY, USA, 2010. Association for Computing Machinery.

[371] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 127, 2012.

[372] John A. Stratton, Sam S. Stone, and Wen-mei W. Hwu. MCUDA: An efficient implementation of CUDA kernels for multi-core CPUs. In José Nelson Amaral, editor, *Languages and Compilers for Parallel Computing*, volume 5335, pages 16–30. Springer, Berlin, Heidelberg, 2008. Series Title: Lecture Notes in Computer Science.

[373] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 4th edition, 2013.

[374] Michelle Mills Strout, Barbara Kreaseck, and Paul D. Hovland. Data-flow analysis for MPI programs. In *2006 International Conference on Parallel Processing (ICPP'06)*, pages 175–184, 2006.

[375] Sander Stuijk, Marc Geilen, and Twan Basten. Sdfˆ 3: Sdf for free. In *Sixth International Conference on Application of Concurrency to System Design (ACSD'06)*, pages 276–278. IEEE, IEEE, 2006.

[376] I-Jui Sung, Juan Gómez-Luna, José María González-Linares, Nicolás Guil, and Wen-Mei W. Hwu. In-place transposition of rectangular matrices on accelerators. *SIGPLAN Notices*, 49(8):207–218, February 2014.

[377] Zehra Sura, Xing Fang, Chi-Leung Wong, Samuel P. Midkiff, Jaejin Lee, and David Padua. Compiler techniques for high performance sequentially consistent java programs. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, page 2–13, New York, NY, USA, 2005. Association for Computing Machinery.

[378] Richard S Sutton and Andrew G Barto. *Introduction to reinforcement learning*, volume 135. MIT press Cambridge, 1998.

[379] Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*, pages 1057–1063, 2000.

[380] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016.

[381] Xinmin Tian, Hideki Saito, Ernesto Su, Jin Lin, Satish Guggilla, Diego Caballero, Matt Masten, Andrew Savonichev, Michael Rice, Elena Demikhovsky, Ayal Zaks, Gil Rapaport, Abhinav Gaba, Vasileios Porpodas, and Eric N. Garcia. LLVM compiler implementation for explicit parallelization and SIMD vectorization. In *Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC*, pages 4:1–4:11, Denver, CO, USA, 2017. ACM.

[382] Pal Tiede. Accelerating black hole imaging with enzyme, 2023.

[383] Markus Towara, Michel Schanen, and Uwe Naumann. MPI-parallel discrete adjoint OpenFOAM. *Procedia Computer Science*, 51:19–28, 2015. International Conference On Computational Science, ICCS 2015.

[384] John R. Tramm, Andrew R. Siegel, Benoit Forget, and Colin Josey. Performance analysis of a reduced data movement algorithm for neutron cross section data in Monte Carlo simulations. In *EASC 2014 - Solving Software Challenges for Exascale*, Stockholm, 2014.

[385] John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. XSBench – the development and verification of a performance abstraction for Monte Carlo reactor analysis. In *PHYSOR 2014 - The Role of Reactor Physics toward a Sustainable Future*, Kyoto, 2014.

[386] Spyridon Triantafyllis, Matthew J Bridges, Easwaran Raman, Guilherme Ottoni, and David I August. A framework for unrestricted whole-program optimization. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 61–71, 2006.

[387] Spyridon Triantafyllis, Manish Vachharajani, Neil Vachharajani, and David I August. Compiler optimization-space exploration. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, pages 204–215. IEEE Computer Society, 2003.

[388] Konrad Trifunovic, Albert Cohen, David Edelsohn, Feng Li, Tobias Grosser, Harsha Jagasia, Razya Ladelsky, Sebastian Pop, Jan Sjödin, and Ramakrishna Upadrasta. GRAPHITE two years after: First lessons learned from real-world polyhedral compilation. In *GCC Research Opportunities Workshop (GROW'10)*, 2010.

[389] Leonard Truong, Rajkishore Barik, Ehsan Totoni, Hai Liu, Chick Markley, Armando Fox, and Tatiana Shpeisman. Latte: A language, compiler, and runtime for elegant and efficient deep neural networks. In *Proc. of the 37th ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI'16, pages 209–223, New York, NY, 2016. ACM.

[390] Chau-Wen Tseng. Compiler optimizations for eliminating barrier synchronization. *ACM SIGPLAN Notices*, 30(8):144–155, 1995.

[391] J. Utke, L. Hascoët, P. Heimbach, C. Hill, P. Hovland, and U. Naumann. Toward adjoinable MPI. In *2009 IEEE International Symposium on Parallel Distributed Processing*, pages 1–8, 2009.

[392] Jean Utke, Uwe Naumann, Mike Fagan, Nathan Tallent, Michelle Strout, Patrick Heimbach, Chris Hill, and Carl Wunsch. OpenAD/F: A modular, open-source tool for automatic differentiation of Fortran codes. *ACM Transactions on Mathematical Software*, 34(4):18:1–18:36, 2008.

[393] Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. Common compiler optimisations are invalid in the C11 memory model and what we can do about it. In *POPL*, pages 209–220, 2015.

[394] Aäron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew W. Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. *CoRR*, abs/1609.03499, 2016.

[395] Nicolas Vasilache, Jeff Johnson, Michaël Mathieu, Soumith Chintala, Serkan Piantino, and Yann LeCun. Fast convolutional nets with fbfft: A GPU performance evaluation. *CoRR*, abs/1412.7580, 2014.

[396] Nicolas Vasilache, Benoît Meister, Muthu Baskaran, and Richard Lethin. Joint scheduling and layout optimization to enable multi-level vectorization. In *IMPACT-2: 2nd Intl. Workshop on Polyhedral Compilation Techniques*, Paris, France, Jan 2012.

[397] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary Devito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. The next 700 accelerated layers: From mathematical expressions of network computation graphs to accelerated gpu kernels, automatically. *ACM Transactions on Architecture and Code Optimization (TACO)*, 16(4):1–26, 2019.

[398] V Vassilev, M Vassilev, A Penev, L Moneta, and V Ilieva. Clad — automatic differentiation using clang and LLVM. *Journal of Physics: Conference Series*, 608:012055, may 2015.

[399] T Veldhuizen and E Gannon. Active libraries: Rethinking the roles of compilers and libraries. In Michael E Henderson, Christopher R Anderson, and Stephen L Lyons, editors, *Proc. of the 1998 SIAM Workshop: Object Oriented Methods for Interoperable Scientific and Engineering Computing*, pages 286–295. SIAM Press, 1998.

[400] Sven Verdoolaege. Isl: An integer set library for the polyhedral model. In *Proc. of the Third Intl. Conf. on Mathematical Software*, ICMS'10, pages 299–302, Berlin, Heidelberg, 2010. Springer.

[401] Sven Verdoolaege. Counting Affine Calculator and Applications. In *First Intl. Workshop on Polyhedral Compilation Techniques (IMPACT'11)*, Chamonix, France, April 2011.

[402] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. Polyhedral Parallel Code Generation for CUDA. *ACM Trans. on Architecture and Code Optimization*, 9(4):54:1–54:23, January 2013.

[403] Sven Verdoolaege and Albert Cohen. Live-range reordering. In *International Workshop on Polyhedral Compilation Techniques*, 2016.

[404] Sven Verdoolaege, Serge Guelton, Tobias Grosser, and Albert Cohen. Schedule Trees. In *4th Workshop on Polyhedral Compilation Techniques (IMPACT, Associated with HiPEAC)*, Vienna, Austria, January 2014.

[405] Sven Verdoolaege and Gerda Janssens. Scheduling for PPCG. Report CW 706, Department of Computer Science, KU Leuven, Leuven, Belgium, June 2017.

[406] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. Polyhedral parallel code generation for cuda. *ACM Trans. Archit. Code Optim.*, 9(4):54:1–54:23, January 2013.

[407] Sven Verdoolaege, Manjunath Kudlur, Harinath Kamepalli, and Rob Schreiber. Generating SIMD instructions for cerebras cs-1 using polyhedral compilation techniques.

In *IMPACT 2020-10th International Workshop on Polyhedral Compilation Techniques*, 2020.

[408] Jordi Vila-Pérez, R Loek Van Heyningen, Ngoc-Cuong Nguyen, and Jaume Peraire. Exasim: Generating discontinuous galerkin codes for numerical solutions of partial differential equations on graphics processors. *SoftwareX*, 20:101212, 2022.

[409] A. Walther and A. Griewank. Getting started with ADOL-C. In U. Naumann and O. Schenk, editors, *Combinatorial Scientific Computing*, chapter 7, pages 181–202. Chapman-Hall CRC Computational Science, 2012.

[410] Mu Wang, Guang Lin, and Alex Pothen. Using automatic differentiation for compressive sensing in uncertainty quantification. *Optimization methods & software*, 33(4-6):799–812, November 2018.

[411] Qiqi Wang, Parviz Moin, and Gianluca Iaccarino. Minimal repetition dynamic checkpointing algorithm for unsteady adjoint calculation. *SIAM Journal on Scientific Computing*, 31(4):2549–2567, 2009.

[412] Z. Wang and M. OBoyle. Machine learning in compiler optimization. In *Machine Learning in Compiler Optimization*, volume 106, pages 1879–1901, Nov 2018.

[413] Richard Wei, Marc Rasi Dan Zheng, and Bart Chrzaszcz. Differentiable programming mega-proposal. `https://github.com/apple/swift/blob/master/docs/DifferentiableProgramming.md`, 2019. Accessed: 2019-09-25.

[414] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Proc. of the 1998 ACM/IEEE Conf. on Supercomputing*, SC'98, pages 1–27, Washington, DC, 1998. IEEE Computer Society.

[415] Jingyue Wu, Artem Belevich, Eli Bendersky, Mark Heffernan, Chris Leary, Jacques Pienaar, Bjarke Roune, Rob Springer, Xuetian Weng, and Robert Hundt. Gpucc: An open-source GPGPU compiler. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO '16, page 105–116, New York, NY, USA, 2016. Association for Computing Machinery.

[416] Peng Wu. Pytorch 2.0: The journey to bringing compiler technologies to the core of pytorch (keynote). In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization*, pages 1–1, 2023.

[417] Yuxin Wu and Kaiming He. Group normalization. *CoRR*, abs/1803.08494, 2018.

[418] Saining Xie, Ross B. Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks. *CoRR*, abs/1611.05431, 2016.

[419] Xilinx. Vivado High-Level Synthesis, 2019.

[420] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 283–294, 2011.

[421] Hanchen Ye, Hyegang Jun, Jin Yang, and Deming Chen. High-level synthesis for domain specific computing. In *Proceedings of the 2023 International Symposium on Physical Design*, pages 211–219, 2023.

[422] Victor A Ying, Mark C Jeffrey, and Daniel Sanchez. T4: Compiling sequential code for effective speculative parallelization in hardware. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 159–172. IEEE, 2020.

[423] Polly Y Yu and Gheorghe Craciun. Mathematical analysis of chemical reaction systems. *Israel Journal of Chemistry*, 58(6-7):733–741, 2018.

[424] Zihan Yu, Cheng Zhang, Derek Nowrouzezahrai, Zhao Dong, and Shuang Zhao. Efficient differentiation of pixel reconstruction filters for path-space differentiable rendering. *ACM Trans. Graph.*, 41(6), nov 2022.

[425] Tomofumi Yuki and Sanjay Rajopadhye. Parametrically tiled distributed memory parallelization of polyhedral programs. Technical Report CS13-105, Colorado State University, June 2013.

[426] Tim Zerrell and Jeremy Bruestle. Stripe: Tensor compilation via the nested polyhedral model. *CoRR*, abs/1903.06498, 2019.

[427] Huihui Zhang, Anand Venkat, Protonu Basu, and Mary Hall. Combining polyhedral and ast transformations in chill. In *Proceedings of the Sixth International Workshop on Polyhedral Compilation Techniques, IMPACT*, volume 16, 2016.

[428] Jisheng Zhao and Vivek Sarkar. Intermediate language extensions for parallelism. In *SPLASH*, pages 329–340, 2011.

[429] Ruizhe Zhao, Jianyi Cheng, Wayne Luk, and George A Constantinides. Polsca: Polyhedral high-level synthesis with compiler transformations. In *32nd International Conference on Field Programmable Logic and Applications (FPL'22)*, 2022.

[430] Oleksandr Zinenko, Sven Verdoolaege, Chandan Reddy, Jun Shirako, Tobias Grosser, Vivek Sarkar, and Albert Cohen. Modeling the conflicting demands of parallelism and temporal/spatial locality in affine scheduling. In *Proceedings of the 27th International Conference on Compiler Construction*, CC 2018, page 3–13, New York, NY, USA, 2018. Association for Computing Machinery.