UC Berkeley
UC Berkeley Electronic Theses and Dissertations

**Title**
Compilation Techniques for Embedded Data Parallel Languages

**Permalink**
https://escholarship.org/uc/item/6c02679n

**Author**
Catanzaro, Bryan Christopher

**Publication Date**
2011

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

Compilation Techniques for Embedded Data Parallel Languages

by

Bryan Christopher Catanzaro

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Engineering - Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Kurt W. Keutzer, Chair
Professor David A. Patterson
Professor Sara McMains

Spring 2011

Compilation Techniques for Embedded Data Parallel Languages

Copyright 2011
by
Bryan Christopher Catanzaro

Abstract

Compilation Techniques for Embedded Data Parallel Languages

by

Bryan Christopher Catanzaro

Doctor of Philosophy in Engineering - Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Kurt W. Keutzer, Chair

Contemporary parallel microprocessors exploit Chip Multiprocessing along with Single Instruction, Multiple Data parallelism to deliver high performance on applications that expose substantial fine-grained data parallelism. Although data parallelism is widely available in many computations, implementing data parallel algorithms in low-level efficiency languages such as C++ is often a difficult task, since the programmer is burdened with mapping data parallelism from an application onto the hardware structures designed to execute it. Languages specifically designed for data parallelism, such as NESL, aim to improve programmer productivity by allowing the programmer to express computation as a composition of data parallel primitives, such as map, reduce and scan. However, efficiently compiling nested data parallel computations to contemporary parallel microprocessors is challenging.

Additionally, the rise of productivity languages, such as Ruby and Python, has facilitated the construction of domain-specific embedded languages. These embedded languages employ familiar syntactic constructs, which eases the task of learning a new programming environment, while also retaining the full capabilities of the host language. This work capitalizes on the productivity of domain-specific embedded languages as well as the nested data parallel abstraction to create a programming environment that is both productive and efficient on contemporary parallel microprocessors. We describe Copperhead, a high-level data parallel language embedded in Python. The Copperhead programmer writes parallel computations via composition of familiar data parallel primitives supporting both flat and nested data parallel computation on arrays of data. Copperhead programs are expressed in a subset of the widely used Python programming language and interoperate with standard Python modules, including libraries for numeric computation, data visualization, and analysis.

Compiling data parallel computations requires analyses and transformations to schedule data parallel operations onto a target platform. We discuss the language, compiler, and runtime features that enable Copperhead to efficiently do so. We define the restricted subset of Python that Copperhead supports and introduce the program analysis techniques and transformations necessary for compiling Copperhead code into efficient low-level implementations. We demonstrate that indiscriminate use of the flattening or vectorization transform, common to data parallel compilers, is inefficient on contemporary microprocessors, and we advocate a more direct mapping of nested data parallel operations onto the natural parallelism hierarchy

provided by today's parallel platforms. We show how this direct mapping allows a data parallel compiler to capitalize on hierarchical on-chip memory structures, as well as perform data parallel primitive fusion in order to gain efficiency. We outline the runtime support that allows Copperhead programs to efficiently interoperate with standard Python modules.

We demonstrate the effectiveness of our techniques with several examples targeting the CUDA platform for parallel programming on graphics processors. Copperhead code is concise, on average requiring 3.6 times fewer lines of code than CUDA, and the compiler generates efficient low-level implementations, yielding 45-100% of the performance of hand-crafted, well optimized CUDA code. Copperhead provides an efficient and productive way to program parallel microprocessors.

To Jena

# Contents

**List of Figures** v

**List of Tables** viii

**1 Introduction** 1
- 1.1 The Implementation Gap . . . 3
- 1.2 Copperhead . . . 5
- 1.3 Contributions . . . 6
- 1.4 Outline . . . 7

**2 Background** 9
- 2.1 Data Parallelism . . . 9
  - 2.1.1 Data Parallel Architectures . . . 10
  - 2.1.2 Contemporary Processors . . . 11
  - 2.1.3 Parallelism Hierarchy . . . 11
- 2.2 Embedded Languages . . . 14
- 2.3 Data Parallel Programming Models . . . 16
- 2.4 Summary . . . 17

**3 The Copperhead Language** 18
- 3.1 Restricted Subset of Python . . . 19
  - 3.1.1 Expressions . . . 20
  - 3.1.2 Statements . . . 21
- 3.2 Type System . . . 21
  - 3.2.1 Definition . . . 22
- 3.3 Side Effects . . . 24
  - 3.3.1 Side Effects in Host Code . . . 26
  - 3.3.2 Loops . . . 27
- 3.4 Scoping and Ordering . . . 27
- 3.5 Closures . . . 28
- 3.6 Data-Parallel Primitives . . . 29
  - 3.6.1 `map` . . . 29
  - 3.6.2 `zip` . . . 30

3.6.3 `reduce` . . . 30
3.6.4 `scan` . . . 31
3.6.5 `gather` . . . 31
3.6.6 `scatter` . . . 31
3.6.7 `permute` . . . 32
3.6.8 `indices` . . . 32
3.7 Example programs . . . 32
3.7.1 Compressed Sparse Row Sparse Matrix Vector Multiplication . . . 32
3.7.2 Radix Sort . . . 33
3.8 Conclusion . . . 33

**4 Compiling Data Parallel Languages** 37
4.1 Source to source compilation . . . 37
4.2 Normalized Form . . . 38
4.2.1 Closure Conversion . . . 39
4.2.2 Single Assignment Conversion . . . 40
4.2.3 Procedure Flattening . . . 40
4.2.4 Expression Flattening . . . 41
4.2.5 Inlining . . . 41
4.2.6 Final Result . . . 43
4.3 Shape Analysis . . . 43
4.4 Data Parallel Primitive Scheduling . . . 45
4.4.1 Data Parallel Primitive Fusion . . . 46
4.5 The Flattening Transform . . . 49
4.6 Quantifying the Flattening Transform . . . 51
4.6.1 Load Balancing . . . 53
4.6.2 SIMD Effects . . . 56
4.6.3 Summary . . . 61
4.7 Scheduling Methodology . . . 61
4.8 Phase Analysis and Scheduling . . . 62
4.8.1 Phase Analysis . . . 62
4.8.2 Phase Scheduling . . . 66
4.8.3 Phase Analysis and Scheduling Example . . . 67
4.8.4 Limitations . . . 69
4.9 Using On-chip Memories . . . 69
4.10 Structures of Arrays . . . 71
4.11 Conclusion . . . 72

**5 The Copperhead Runtime** 73
5.1 CUDA C++ Back End . . . 73
5.2 Runtime . . . 74
5.2.1 Runtime Static Compilation . . . 76
5.2.2 Places . . . 78

5.3 Data Structures . . . 79
5.3.1 Arbitrarily Nested Sequences . . . 79
5.3.2 Uniformly Nested Sequences . . . 80
5.4 Foreign Function Interface . . . 81
5.5 Runtime Overheads . . . 82
5.6 Systems without compilers . . . 83
5.7 Conclusion . . . 84

**6 Results** **85**
6.1 Sparse Matrix Vector Multiplication . . . 86
6.2 Preconditioned Conjugate Gradient Linear Solver . . . 89
6.3 Lanczos Eigensolver . . . 92
6.4 Quadratic Programming: Nonlinear Support Vector Machine Training . . . 97
6.5 Productivity . . . 102
6.6 Conclusion . . . 104

**7 Conclusions** **105**
7.1 Contributions . . . 105
7.1.1 Direct Mapping of Nested Data Parallelism . . . 105
7.1.2 Phase Analysis and Scheduling . . . 106
7.1.3 Runtime Static Compilation . . . 107
7.2 Availability . . . 108
7.3 Future Work . . . 108
7.3.1 Alternative Backends . . . 108
7.3.2 Autotuning . . . 108
7.3.3 Aspect Oriented Debugging . . . 109
7.3.4 Usability Studies . . . 109
7.4 Conclusion . . . 109

**Bibliography** **111**

# List of Figures


1.1 Historical Clock Speeds of Intel Microprocessors. Data from [44] . . . . . 2
1.2 The Implementation Gap . . . . . 4

2.1 Two representative processors . . . . . 12
2.2 High level parameters of two representative processors . . . . . 12
2.3 Abstract Architecture . . . . . 13
2.4 Parallel Hierarchy . . . . . 13

3.1 Copperhead is an Embedded Subset of Python, designed to interoperate with standard Python libraries for numeric computing . . . . . 19
3.2 Grammar for Copperhead Expressions . . . . . 20
3.3 Grammar for Copperhead Statements . . . . . 21
3.4 Well typed and ill-typed programs . . . . . 22
3.5 A grammar for the Copperhead type system . . . . . 22
3.6 Describing Copperhead Types . . . . . 24
3.7 Example code with and without side effects . . . . . 24
3.8 Python code with side effects . . . . . 26
3.9 Referencing procedure identifiers defined outside a Copperhead procedure . 27
3.10 Referencing data identifiers defined outside a Python procedure . . . . . 27
3.11 Dynamic scoping in Python . . . . . 28
3.12 Illustrating Closures . . . . . 28
3.13 Procedure for computing $Ax$ for a matrix $A$ in CSR form and a dense vector $x$. Underlined operations indicate potential sources of parallel execution. . . . 32
3.14 Radix sort in Copperhead . . . . . 34
3.15 Illustrating Radix Sort . . . . . 35

4.1 Copperhead Compiler Flow . . . . . 38
4.2 Scaled vector addition before Closure Conversion . . . . . 39
4.3 Scaled vector addition after Closure Conversion . . . . . 39
4.4 A procedure before Single Assignment Conversion . . . . . 40
4.5 A procedure after Single Assignment Conversion . . . . . 40
4.6 Scaled vector addition before procedure flattening . . . . . 41
4.7 Scaled vector addition after procedure flattening . . . . . 41
4.8 Scaled vector addition before expression flattening . . . . . 42

4.9 Scaled vector addition after expression flattening . . . 42
4.10 Example procedure before inlining . . . 42
4.11 Example procedure after inlining . . . 42
4.12 SpMV procedure from Figure 3.13 after transformation by the front end compiler. . . . 43
4.13 A program with indeterminate shape . . . 44
4.14 Fusion Example Code . . . 46
4.15 Fusion Example Code after Normalization . . . 47
4.16 The performance impact of data parallel primitive fusion . . . 48
4.17 A nested sum operation . . . 49
4.18 Executing a Segmented Reduction . . . 50
4.19 Copperhead code for `multi_norm` . . . 52
4.20 Copperhead-like code for the flattened version of `multi_norm` . . . 52
4.21 Performance comparison of flattened versus direct mapped scheduling . . . 55
4.22 Performance comparison of nested parallelism mapping strategies, $\psi = 1$ . . . 57
4.23 Performance comparison of nested parallelism mapping strategies, $\psi = 10$ . . . 58
4.24 Performance comparison of nested parallelism mapping strategies, $\psi = 100$ . . . 58
4.25 Performance comparison of nested parallelism mapping strategies, $\psi = 1000$ 59
4.26 Performance comparison of nested parallelism mapping strategies, $\psi = 10000$ 59
4.27 Performance comparison of nested parallelism mapping strategies, $\psi = 100000$ 60
4.28 Simple flat data parallel example . . . 67
4.29 Data Dependence graph for code in figure 4.28 . . . 68
4.30 Analyzed and Scheduled data dependence graph for code in figure 4.28 . . . 68
4.31 Unfused primitives . . . 69
4.32 Closed over data may be intensively reused . . . 70
4.33 Creating an Array of Structures . . . 71

5.1 Normalized, scheduled output of Mid-end compiler . . . 74
5.2 Sample CUDA C++ code generated for `spmv_csr`. Ellipses (...) indicate incidental type and argument information elided for brevity. . . . 75
5.3 Runtime Compilation . . . 77
5.4 Using Execution Places . . . 78
5.5 Implementing an Arbitrarily Nested Sequence . . . 79

6.1 SpMV procedure for sparse matrices in ELL format. . . . 87
6.2 Sparse Matrix and its Representation in ELL Format . . . 87
6.3 Average Double Precision Sparse Matrix Vector Multiply Performance . . . 89
6.4 Left: closeup of video frame. Right: gradient vector field for optical flow . . . 90
6.5 Structure of the matrix from the optical flow problem . . . 90
6.6 Forming and applying the block Jacobi preconditioner . . . 92
6.7 Initializing the Conjugate Gradient Iterations . . . 93
6.8 Preconditioned Conjugate Gradient Iteration . . . 94
6.9 Performance on Preconditioned Conjugate Gradient Solver . . . 95
6.10 Lanczos eigensolver iteration . . . 96

6.11 RBF Kernel evaluation . . . . . 99
6.12 Reduction operator for computing the Arg Extrema of a vector . . . . . 100
6.13 SVM Training iteration . . . . . 101
6.14 Standardized Lines of Code for Copperhead and C++ Programs . . . . . 103

# List of Tables


3.1 Basic monotypes in the Copperhead type system . . . . . 23

4.1 Selected Partitionings and Performance . . . . . 54
4.2 The parallel hierarchy $\mathcal{P}$ provided by OpenCL and CUDA . . . . . 63
4.3 Completion and Directionality types for selected data parallel primitives at the Distributed Level . . . . . 65
4.4 Completion and Directionality types for selected data parallel primitives at the Sequential Level . . . . . 65

5.1 Runtime Overheads . . . . . 83

6.1 Double Precision Sparse Matrix Vector Multiplication Performance in GFLOP/s . . . . . 88
6.2 Lanczos performance results . . . . . 97
6.3 Support Vector Machine Training Performance . . . . . 100
6.4 Number of Lines of Code for Example Programs . . . . . 102

# Acknowledgments

I'd like to thank my advisor, Kurt Keutzer, for his support, guidance and insight over the course of my time at Berkeley. Kurt went far beyond my expectations to make sure I felt comfortable and worked productively, and his advice has been instrumental in guiding this work, as well as preparing me for the future.

I'd also like to thank Michael Garland from NVIDIA Research, who made significant contributions to the design and implementation of Copperhead.

My work has been shaped by eight industrial internships; I'd like to thank Tom Fletcher for giving me my first exposure to the computer industry, as well as Kumar Anshumali, Herman Schmit, Pradeep Dubey, Jatin Chhugani, Sanjeev Kumar, Michael Garland and David Luebke for their mentorship.

Several academic instructors and advisors have broadened my horizons. I'd particularly like to thank Brent Nelson, Travis Oliphant, Tom Sederberg, Richard Frost, Raissa Solovyova, Rosalind Hall, Anna Muza and Karen Brotherson. They have enriched my understanding and challenged me to work harder, understand more deeply, and believe in myself. I'd also like to thank David Patterson and Sara McMains for serving on my dissertation committee and significantly improving this manuscript through their careful reading.

While at Berkeley, I've been privileged to collaborate with a stellar group of researchers, including Narayanan Sundaram, Bor-Yiing Su, Mark Murphy, Jike Chong, Yunsup Lee, Will Plishker, Kaushik Ravindran, NR Satish, Ekaterina Gonina, Chao-Yue Lai, Michael Anderson, and David Sheffield, as well as Armando Fox, David Patterson, Krste Asanović, James Demmel, and Kathy Yelick. Special thanks go to Henry Cook and Tyler Mecke, who contributed to the Copperhead project. Other graduate students in the department have made my experience at Berkeley lively and interesting, including Bryan Brady, Matt Moskewicz, Scott Weber, Andrew Mihal, Abhijit Davare, Elaine Cheong, Nathan Kitchen, Donald Chai, Christopher Batten, David Chinnery, and Doug Densmore, among others. Thanks to all for making my time here stimulating and productive.

Most importantly, I acknowledge my family: my wife Jena, my children Hyrum, Evelyn and Claire, and my parents Paul and Belinda. The sacrifices they have made in support of this work have made it possible, and their presence in my life gives me purpose.

# 1 Introduction

Driven by the capabilities and limitations of modern semiconductor manufacturing, the computing industry is currently undergoing a massive shift towards parallel computing. For more than three decades, the increased integration capabilities provided by successive generations of semiconductor manufacturing yielded ever more capable sequential processors. During the 1990s alone, sequential processor performance improved by two orders of magnitude. These increases in processing power served as the foundation for advances in computing applications that have profoundly changed the way we live, learn, conduct business, and entertain ourselves.

However, during the 2000s, sequential processor performance improvements slackened [2]. This was due primarily to power consumption constraints – as processors became more and more complex, their power consumption increased superlinearly until it was no longer feasible to supply power and cooling. Figure 1.1 illustrates this trend by graphing the clock speed of Intel microprocessors as a function of time. Clock speeds increased at a rapid pace until around 2004, when the Prescott Pentium 4 processor was introduced at 3.8 GHz. After that point, clock speeds have remained stagnant for the past seven years. For the first time in the forty year history of the microprocessor, we find ourselves in an era where processor frequency has virtually ceased scaling. Of course, processor performance is only correlated with frequency; other architectural properties also influence single-threaded performance, and in fact, we have seen single-threaded performance improve, albeit at a reduced rate. However, it is clear that we are now in a new era, one that requires parallel software in order to take advantage of parallel hardware.

Instead of continuing to focus on sequential performance, microprocessor architects have turned their attention on power efficiency and single chip multiprocessing. By dialing back sequential performance by a few percent, architects have reaped large power savings per core, and then integrated larger numbers of cores into multicore and manycore processors. The aggregate performance of each of these assemblages of cores can be much higher than the power-limited performance that could have been reached by single-threaded processors alone – but taking advantage of parallel processors requires parallel software.

Ideally, we would be transitioning to parallel processing because breakthroughs in parallel programming models and applications had proven that parallel processing is successful and profitable. Instead, the shift towards parallel computing is actually a retreat from even more daunting problems in silicon manufacturing and computer architecture [2]. As a result, com-

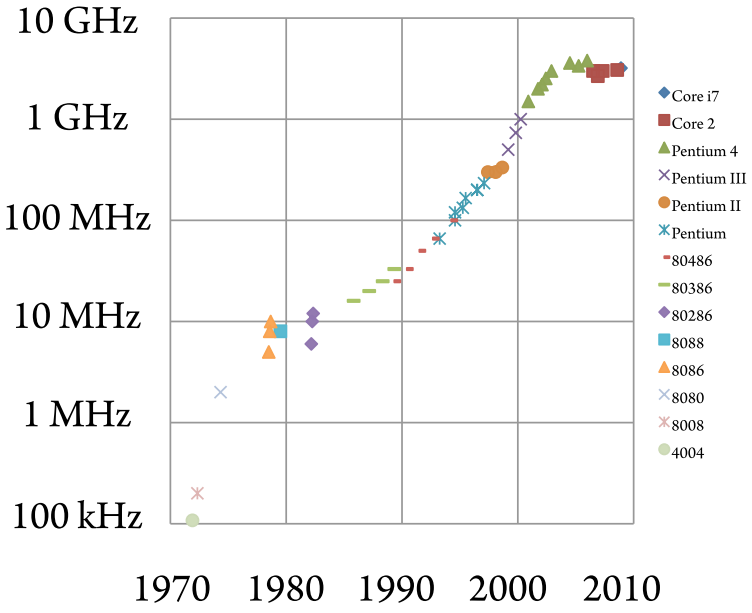



Figure 1.1: Historical Clock Speeds of Intel Microprocessors. Data from [44]

putationally intensive applications must now be rewritten to be scalable and efficient on parallel platforms. Consequently, one of the preeminent challenges of the current computing era is to make parallel programming mainstream and successful.

Although parallel programming has a long and somewhat challenged history, the computing landscape is different this time, and so parallelism is much more likely to succeed.

First, thanks to the integration capabilities of modern semiconductor manufacturing, the nature of parallel implementation has changed. Previous efforts at parallel programming focused on clusters of multi-socket, multi-node systems. With such systems, the overhead of communication and synchronization often dominate the improved performance obtained through parallelism, leading to poor application scalability: increasing the number of cores in a system did not improve performance. Now that parallel processors are integrated into monolithic silicon devices that share the same off-chip memory, more algorithms and programming models can effectively exploit parallelism, since the cost of communication and synchronization has been reduced by orders of magnitude.

Second, the economics behind parallel processing have changed. Parallel processing used to require a large investment: if you wanted a machine with twice the parallelism, you needed to pay at least twice as much money to pay for the extra processors. Consequently, in order for parallelism to be considered successful, applications needed to show linear or near-linear scalability in order to justify hardware investments. The advent of on-chip multiprocessor architectures have changed this calculus, since they translate the increased integration capabilities provided by Moore's law directly into parallelism. Pollock and Gelsinger famously observed [35] that doubling the number of transistors in a sequential processor results in only about a

40% performance improvement. Accordingly, to stay on the same performance trend that we enjoyed during the sequential processing era, if we double the number of cores on the chip, we need demonstrate only 40% performance improvement to justify the investment. As a result, sub-linear performance scaling now makes sense, and so nowadays, virtually every processor on the market is parallel, including processors for mobile devices, such as smartphones and tablets. Parallelism is coming along essentially for free, it is ubiquitous, and does not require large investment or perfect scalability to be profitable.

Finally, this time around, we have no alternative. Earlier attempts to capitalize on parallel processing were less successful because sequential processors were improving so rapidly. If an application wasn't performing quickly enough, one could just wait for a year until a faster generation of sequential processors came out. Now that sequential processor performance increases have drastically slowed, we don't have the option of waiting for sequential processors to catch up with our performance needs. Instead, it's time to examine our applications for parallelism and find efficient parallel implementations.

The good news is that parallelism can actually be fairly easy to find and extract, if one uses the correct tools and mental models. The push to parallel software, albeit a significant change, is far from an impossible task. Parallelism is abundant, often derived from the properties of the computations we're trying to perform, which model, analyze, and synthesize large data sets consisting of many objects.

## 1.1 The Implementation Gap

Computer architects are now creating highly parallel architectures, the challenge is exploiting them efficiently and productively. Manual implementation of parallel programs using efficiency languages such as C and C++ can yield high performance, but at a large cost in programmer productivity. Some case studies show that productivity-oriented languages such as Ruby and Python are two to five times more productive than efficiency languages [46, 75]. Additionally, parallel programming in efficiency languages is often viewed as an esoteric endeavor, to be undertaken by experts only. Without higher-level abstractions that enable easier parallel programming, parallel processing will not be widely utilized.

Conversely, application developers do not have the means to capitalize on fine-grained, highly parallel microprocessors, which limits their ability to implement computationally intensive applications. Application developers require high productivity from programming environments, to enable algorithmic design space exploration and rapid application prototyping. Consequently, they often use highly productive programming languages and environments, such as Ruby, Python and MATLAB, despite the inefficient way these environments exploit computing resources.

Figure 1.2 illustrates this problem, which we call the implementation gap [82, 73, 25]. Closing the implementation gap requires creating software infrastructure that provides higher level abstractions to application developers that enable them to productively program their applications, but in a way that can be mapped to efficient parallel implementations. We argue that higher level data parallel abstractions can serve as a productive substrate for efficient parallel

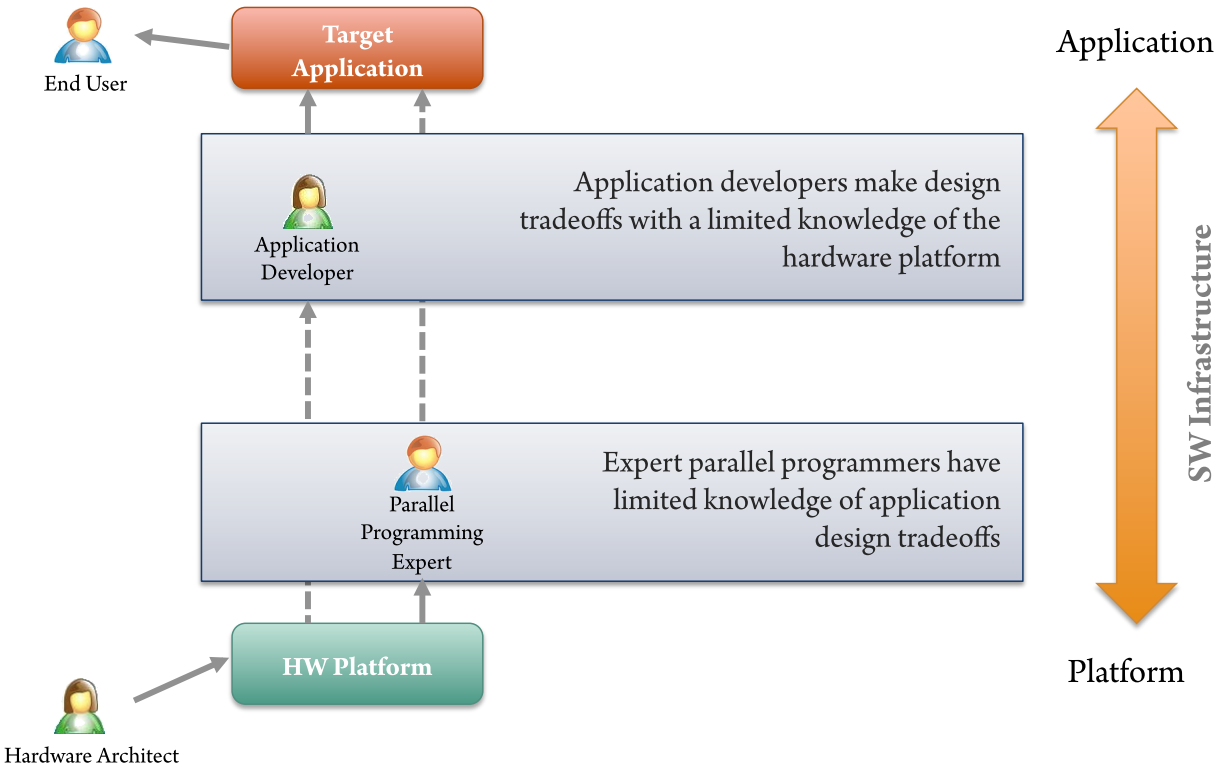



Figure 1.2: The Implementation Gap

programming, enabling the creation of software infrastructure that bridges the implementation gap. We create such software infrastructure in the form of a compiler and runtime, embedded in the popular Python language. The programmer expresses computations at a much higher level of abstraction, and our compiler and runtime efficiently execute them on a parallel platform. Although computations expressed using our infrastructure are much more abstract than a low-level efficiency language, we can achieve performance within a factor of two of optimized code written in an efficiency language. This represents a step towards closing the implementation gap for parallel programming.

## 1.2 Copperhead

Although the need for higher-level parallel abstractions seems clear, perhaps less so are the abstractions that should be provided to bridge the implementation gap, since there are many potential abstractions that could be presented to productivity programmers. In our view, *nested data parallelism*, as introduced by languages such as NESL [10], is particularly interesting. In data parallel languages, programmers express computation by using explicitly parallel operators that apply a computation element-wise across large sequences. Communication is expressed through data parallel primitives, such as reductions, scans, sorts, and compactions, which are also explicitly parallel. Although flat data parallel programming can be both high-performance and concise, its expressibility is limited. Many computations and data structures have a hierarchical structure, which is more naturally expressed by composing data parallel operations in a hierarchical fashion, operating on hierarchical data structures. For example, for sparse matrix vector multiplication, the matrix can be represented as a sequence of rows, each of which contains a sequence of non-zero elements. The computation involves applying a sparse vector dot product operation to each row of the matrix. The dot product operation itself can be expressed using data parallel primitives. Nested data parallelism is a natural way of expressing computation.

Nested data parallelism is abundant in many computationally intensive algorithms. It can be mapped efficiently to parallel microprocessors, which prominently feature hardware support for data parallelism. For example, mainstream x86 processors from Intel and AMD are adopting 8-wide vector instructions, Intel's Larrabee processor used 16-wide vector instructions, and modern Graphics Processing Units (GPUs) from NVIDIA and AMD use wider SIMD widths of 32 and 64, respectively. Consequently, programs that don't take advantage of data parallelism relinquish substantial performance.

Additionally, nested data parallelism as an abstraction explicitly exposes parallelism. In contrast to traditional auto-parallelizing compilers that must analyze and prove which operations can be parallelized and which can not, the compiler of a data-parallel language needs only to decide which operations to perform sequentially, and which to perform in parallel. Making these decisions involves mapping the nested data parallel operations onto the hierarchical structure of the parallel platform being targeted. The mechanisms by which we perform this mapping are crucial to achieving high performance, and are the central task of a data-parallel compiler.

Importantly, nested data parallel programs have a valid sequential interpretation, and are thus easier to understand and debug than parallel programs that expose race conditions and other complications of parallelism.

Motivated by these observations, Copperhead provides a set of nested data parallel abstractions, expressed in a restricted subset of the widely used Python programming language. Copperhead follows the ideas of Selective, Embedded, Just-in-Time Specialization (SEJITS)[13]. Instead of creating an entirely new programming language for nested data parallelism, we repurpose existing constructs in Python, such as `map` and `reduce`. Embedding Copperhead in Python provides several important benefits. For those who are already familiar with Python, learning Copperhead is more similar to learning how to use a Python package than it is to learning a new language. There is no need to learn any new syntax, instead the programmer must learn only what subset of Python is supported by the Copperhead language and runtime. Copperhead programs are clearly delineated from surrounding Python code using standard Python syntax, so that the selected parts of the application that conform to Copperhead's restricted syntax and semantics are obvious. The Copperhead runtime, which invokes compilation in limited ways, is implemented as a standard Python extension. Copperhead programs are invoked through the Python interpreter, allowing them to interoperate with the wide variety of Python libraries already extant for numeric computation, file manipulation, data visualization, and analysis. The ability to interoperate with the broad array of Python libraries makes Copperhead a productive environment for prototyping, debugging, and implementing entire applications, not just their computationally intense kernels.

Of course, without being able to efficiently compile nested data parallel programs, Copperhead would not be of much use. Previous work on compilers for nested data parallel programming languages has focused on *flattening transforms* to target flat arrays of processing units. As we will discuss in Chapter 4, flattening transforms are a useful tool, but applying them indiscriminately sacrifices factors of two to five in performance on contemporary parallel microprocessors, because flattening transforms do not take advantage of the MIMD capabilities of on-chip multiprocessors.

Contemporary parallel microprocessors support a hierarchy of parallelism, with independent cores containing tightly coupled processing elements. Accordingly, the Copperhead compiler maps nested data parallel programs to a nested parallelism hierarchy, forgoing the use of flattening transforms, which we find to substantially improve performance. This is the central task of the Copperhead compiler, to which we will devote much attention in this work.

The current Copperhead compiler targets CUDA C++, running on manycore Graphics Processors from NVIDIA. We generate efficient, scalable parallel programs, performing within 45-100% of well optimized, hand-tuned CUDA code. Our initial implementation focus on CUDA does not limit us to a particular hardware platform, as the techniques we have developed for compiling data-parallel code are widely applicable to other platforms as well.

## 1.3 Contributions

This work includes the following main contributions:

- A direct mapping strategy for nested data parallel operations onto the parallel hierarchy provided by a target parallel platform. We detail how the flattening or vectorization transform common to many other data parallel compilers in inefficient on contemporary microprocessors, with a performance loss of $3 - 5\times$ compared to direct mappings of nested data parallel operations onto a parallel hierarchy. The direct mapping strategy allows us to use the control flow structures avaliable in contemporary multiprocessors, which is crucial to attaining efficient performance comparable with hand-written efficiency code.

- Supporting this strategy, we propose a simple *phase analysis* technique that analyzes dataflow graphs to discover synchronization points, as well as *phase scheduling*, which fuses data parallel operations into phases that proceed without synchronization. Fusing data parallel operations is also crucial in order to achieve efficiency: for simple examples, we show that not fusing data parallel operations can result in a performance loss of $9\times$, due to increased memory bandwidth usage as well as extraneous synchronization, which limits the effective parallelism the platform can sustain. The decision of which operations to fuse can be a complicated optimization, and so we choose to use a simple maximalist heuristic that works well in many cases.

- Finally, we show how *runtime static compilation* enables efficiently embedding compiled languages into productivity environments. Naive embeddings of compiled languages can incur large runtime compilation costs, which can be on the order of tens of seconds, thus overwhelming efficiencies from a compiled binary that may only run for milliseconds. Runtime static compilation provides productivity programmers with a familiar workflow, while mitigating runtime overheads. Consequently, more sophisticated code generation techniques may be employed, such as autotuning and design space exploration. Additionally, the results of our compiler can be reused as a library in other programs, and applications built using runtime static compilation can be deployed on targets that disallow dynamic code generation, whether due to security constraints or the fact that many clients may not have compilers installed.

## 1.4 Outline

This thesis follows the following outline:

- Chapter 2 gives the background necessary to understand this thesis, including a discussion of related work.

- Chapter 3 describes the Copperhead language, along with its type system, the data parallel primitives it provides, and its relationship to Python.

- Chapter 4 details the compilation techniques by which Copperhead programs are efficiently compiled to efficiency language implementations. Special attention is paid to mapping nested data parallel operations directly onto a parallelism hierarchy.

- Chapter 5 discusses the runtime by which Copperhead programs are embedded in Python programs, describing the runtime static compilation approach we employ.
- Chapter 6 provides experimental results to show that Copperhead programs are both concise and efficient.
- Chapter 7 concludes with a discussion of the contributions of this work.

The next chapter explains the importance of data parallelism, the history of data parallel programming models and architectures, as well as trends in embedded domain specific languages, all of which influence this work.

# 2 Background

In this chapter, we discuss some important background and related work.

## 2.1 Data Parallelism

As mentioned in the introduction, the shift to parallelism has been motivated by power efficiency considerations. On parallelized workloads, multiple slower cores running in parallel can deliver the performance of a fast monolithic core, but at much lower power consumption. However, in order to keep performance scaling at historical rates, the number of cores on a chip must continue to scale along with Moore's law. This, in turn, requires programmers to write parallel software that can take advantage of all these cores. The Berkeley View [2] argued that, once the genie of parallelism had been released from its bottle, programmers and hardware architects ought to embrace highly parallel architectures, which sacrifice single threaded performance in order to deliver higher aggregate performance on parallelizeable code, while improving overall power efficiency. This argument was motivated by the observation that an exponential increase in core count, enabled by Moore's law, will drive all architectures to highly parallel architectures in a relatively short timeframe. Accordingly, computationally intensive applications must be able to use very large amounts of parallelism. If a single application is to scale to utilize large amounts of parallelism, it stands to reason that it will be taking advantage of relatively small, fine-grained parallelism, or, in other words, data parallelism.

Data parallelism has been widely utilized for many years, both in hardware architectures as well as programming models, thanks to its excellent parallel scalability. In Our Pattern Language (OPL), data parallelism is a key algorithmic strategy pattern, which describes how an algorithm is expressed [52], [62]. Quoting [52], data parallelism is summarized as

> An algorithm is organized as operations applied concurrently to the elements of a set of data structures. The concurrency is in the data.

The beauty of data parallelism is that it is highly scalable, since parallelism is not determined by potentially concurrent tasks performing different operations, but rather by the parallelism naturally present in the data. As long as the data set is large, the available parallelism is also large.

### 2.1.1 Data Parallel Architectures

Although the connection between computer architecture and programming models can at times be obscured, our interest in data parallel languages is motivated by contemporary computer architecture, and our desire to achieve efficient performance necessitates a good understanding of the architectures we target. The potential of data parallelism to provide scalability to parallel algorithms has been known for many years, and various architectures have been proposed and implemented. We cite a few important exemplars that have led to the creation of contemporary data parallel architectures.

One of the very first architectures designed for data parallelism was the Westinghouse SOLOMON computer from the early 1960s. SOLOMON was organized as an array of Processing Elements (PEs) that executed identical programs in lock step, controlled by a sequencer processor, where each PE could communicate only with its neighbors in a 2D mesh topology [83]. The SOLOMON computer pioneered the idea of having multiple processors execute the same program in lockstep. This concept proved to be a good fit between the needs of computationally intensive codes, which are often expressed in a data parallel fashion, as well as the needs of efficient computer architecture, since Single Instruction Multiple Data execution brings compelling hardware efficiencies. Extending the ideas introduced by SOLOMON, the ILLIAC IV computer was constructed as a set of several 64-processor arrays, each modeled after SOLOMON [4].

Two decades later, the Connection Machines, such as the CM-2 and CM-5 [88], [39], [41], continued the lineage of data parallel architectures, organized around large, flat SIMD processor arrays. The Maspar MP-1 [6], [66] followed a similar philosophy, but used improved semiconductor processes to increase on-chip parallelism and integration.

Vector architectures also made significant use of data parallelism for increased scalability and efficiency. Vector architectures encourage the expression of data parallel operations at the instruction set level, which are then time-multiplexed onto SIMD hardware. Some important vector architectures include the Cray 1 [77], XMP , and YMP-C90 [69]. The Floating Point Systems array processors [22] employed commodity hardware with software pipelining to efficiently implement vector operations. More recently, the Vector IRAM project [57] integrated a vector processor with on-chip eDRAM to address the memory bandwidth and latency concerns that arise in modern computer architectures due to the so-called "memory wall" [2]. Stream processors, such as Imagine [53], are closely related to vector processors and are also oriented towards data parallelism.

Additionally, even traditional CPUs, such as those from Intel, ARM, and IBM, have adopted SIMD extensions. Intel's recent Sandy Bridge processors introduce the AVX instruction set, which provides 8-wide SIMD instructions. Interestingly, the first implementation of Sandy Bridge offers 8-wide SIMD instructions on a four-core architecture, meaning that the parallelism available from SIMD operations is greater than the parallelism provided by the multicore architecture. In other words, traditional CPUs are becoming more reliant on data parallelism for performance and efficiency reasons. The Larrabee [80], Knight's Ferry, and Knight's Corner projects from Intel integrate large numbers of simple CPU cores, each with a 32-wide SIMD vector unit. Clearly, architectural support for data parallelism is widespread

and abundant, even among traditional multicore architectures.

Programmable Graphics Processing Units (GPUs)from AMD and NVIDIA are also architected to exploit large amounts of data parallelism [61], [71]. They feature several highly multithreaded processing elements, each with their own private memories, which operate on wide SIMD vectors, from 8-64 elements long, and are optimized for streaming workloads, which compensate for long memory latencies of several hundred clock cycles by switching between many instruction contexts.

### 2.1.2 Contemporary Processors

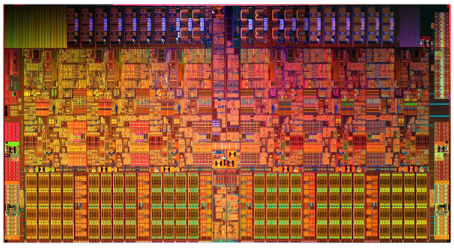
As noted earlier, we are motivated by the need for parallelism within a single socket. Since monolithic processors have been replaced with integrated on-chip multiprocessors, the need intensifies for parallel programming models that allow single socket processors to take advantage of parallelism. There are many parallel programming models and parallel architectures that focus on issues that arise during programming of clustered systems. However, we have chosen to restrict our attention to single socket parallelism, since the single socket case is of pressing importance. Although the techniques we describe could be extended to target multi-socket parallel systems, we have focused on the single-socket case, where all processors on the chip share a single offchip memory pool.

To further sharpen our discussion of contemporary processors, we consider two processors shown in Figure 2.1 and detailed in Table 2.2: Intel's Westmere-EP and NVIDIA's GF110.

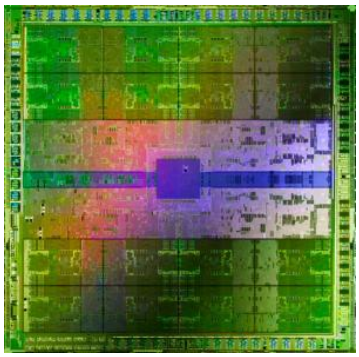
Both of these processors are highly reliant on parallelism to attain maximum performance. Both of them are composed of multiple independent processing cores, sharing a last level cache. However, the degree of parallelism required to fully utilize these processors differs substantially. This can be seen when comparing the number of resident SIMD lanes in each processor, or equivalently, the number of resident thread contexts that can simultaneously reside in dedicated state on the processor, multiplied by the SIMD width per thread. The Westmere-EP processor can hold 48 lanes in context, while the GF110 holds 24576. This means that programming models for the GF110 processor must express and exploit considerable fine-grained parallelism in order to utilize the processor. In order to hold the contexts for so many SIMD lanes, the GF110 has an inverted memory hierarchy in comparison to the Westmere-EP: whereas conventional processors have the register file as the smallest pool of memory on the chip, and the last level cache as the largest, the GF110 register file is actually the largest amount of state on the processor, and subsequent levels of the memory hierarchy are progressively smaller.

### 2.1.3 Parallelism Hierarchy

Now that we have examined contemporary parallel architectures, we present the abstraction that we will target with our compiler. Figure 2.3 shows a simple model for a single socket processor, comprised of multiple cores, each of which have their own private on-chip memories. Each core is multithreaded, with register file state for multiple SIMD vectors. Each core also contains SIMD vector units. Corresponding to this abstract architecture, we have a parallelism



Westmere-EP



GF110

Figure 2.1: Two representative processors

| | Intel Westmere-EP | NVIDIA GF110 |
|---|---|---|
| Processing Elements | 6 cores | 16 cores |
| Physical SIMD | 4 way SIMD | 16 way SIMD |
| Logical SIMD | 4 way SIMD | 32 way SIMD |
| SIMD Issue Rate | 2 SIMD/cycle | 2 SIMD/cycle |
| Multithreading | 2 threads/core | 48 threads/core |
| Maximum Resident SIMD Lanes | 48 | 24576 |
| Register File | 6 kB | 2048 kB |
| Local Store/L1 Data Cache | 192 kB | 1024 kB |
| L2 Data Cache | 1536 kB | 768 kB |
| L3 Data Cache | 12 MB | — |
| Transistor Count | 1.17 B | 3.0 B |
| Semiconductor Process | 32 nm | 40 nm |
| Die Size | 248 $mm^2$ | 520 $mm^2$ |
| Clock Frequency | 3.46 GHz | 1.54 GHz |
| Power Dissipation (TDP) | 95 W | 220 W (processor only) |
| SP GFLOP/s | 166 | 1577 |
| DP GFLOP/s | 83 | 788 |
| DRAM Bandwidth | 32 GB/s | 192 GB/s |

Figure 2.2: High level parameters of two representative processors

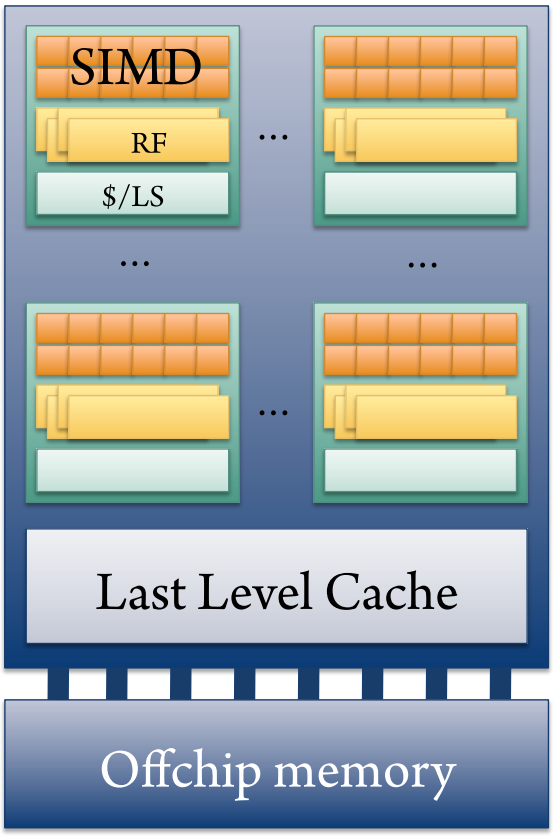



Figure 2.3: Abstract Architecture

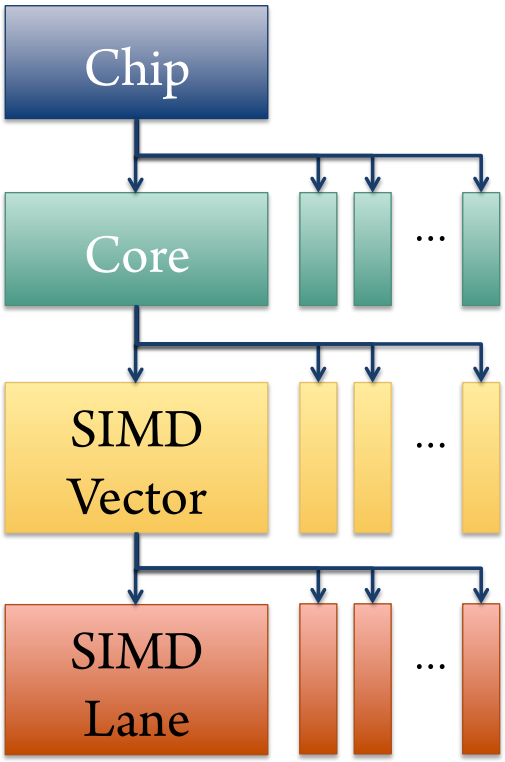



Figure 2.4: Parallel Hierarchy

hierarchy. Figure 2.4 illustrates this hierarchy: the root of the hierarchy is the chip, since we are targeting single-socket parallelism. Each chip has multiple cores, each core stores execution contexts for multiple SIMD vectors via multithreading, and each SIMD vector has multiple SIMD lanes. Targeting this model, our compiler gains high efficiency, because the code we emit maps naturally to the hardware structures provided by the processor.

## 2.2 Embedded Languages

With the growing interest in computational science, more programming is done by experts in each application domain instead of by expert programmers. Such domain experts, such as science professionals like physicists, biologists, and medical researchers, as well as engineers in various fields and computer scientists studying particular algorithmic domains, like computer vision or machine learning, constitute an important category of customers for high performance and parallel processing for two reasons. First, their respective domains are growing in importance. Second, their application domains can greatly benefit from improved computational capability. These domain experts increasingly turn to scripting languages such as Python and MATLAB, which emphasize programmer productivity over hardware efficiency. Besides offering abstractions tailored to the domains, these *productivity-level languages* (PLLs) often provide excellent facilities for debugging and visualization. While we are not yet aware of large-scale longitudinal studies on the productivity of such languages compared to traditional imperative languages such as C, C++, and Java, individual case studies have found that such languages allow programmers to express the same programs in 3–10× fewer lines of code and in 1/5 to 1/3 the development time [74, 24, 46].

Although PLLs support rapid development of initial working code, they typically make inefficient use of underlying hardware, performing 100 to 1000× slower than sequential efficiency code on many problems. The recent move towards parallel processing amplifies this performance gap, since the flexibility that makes PLLs efficient also impedes parallel implementation. For example, Python and Ruby do not support concurrently executing threads, since it is not possible in these languages to prove that two operations are independent, given the extremely dynamic dispatch and data structures that underpin their respective interpreters.

In contrast, contemporary multicore CPUs and manycore graphics processors require careful low-level orchestration to attain reasonable efficiency. Consequently, many applications are eventually rewritten in *efficiency-level languages* (ELLs), such as C or C++ with parallel extensions like Cilk, OpenMP, and CUDA. Because ELLs expose hardware-supported programming models directly, they can achieve orders of magnitude higher performance than PLLs on emerging parallel hardware. However, the performance comes at high cost: the abstractions provided by ELLs are a poor match to those used by domain experts. Additionally, programs written in ELLs are not very portable between parallel platforms, making ELLs a poor medium for exploratory work, debugging, and prototyping.

Ideally, domain experts could use high-productivity domain-appropriate abstractions *and* achieve high performance in a single language, without rewriting their code. However, the *implementation gap* discussed earlier makes it difficult to program both productively and ef-

ficiently, since the abstractions needed to program productively do not map well to parallel hardware. This implementation gap is already a problem, but is further widening. Domains are specializing into sub-disciplines, and available target hardware is becoming more heterogeneous, with multithreaded multicore, manycore GPUs, and message-passing systems all exposing radically different programming models.

The need for higher productivity is not new, which is why domain experts have long turned to higher level PLLs. However, the increasing difficulty of creating efficient code for contemporary parallel architectures means that it is increasingly important for higher level programs to successfully be compiled efficiently on parallel hardware. We achieve both higher productivity and higher performance by embedding a data parallel language in a PLL, utilizing the metaprogramming and introspection facilities in modern scripting languages such as Python and Ruby to compile high level data parallel computations to data parallel hardware.

Much related work exists, describing how to embed domain specific languages into higher-level scripting environments [45]. Such approaches avoid the need for users to learn a new language – instead they simply learn the interface of a library embedded in a higher-level language, which can then be interpreted according to the semantics the library provides. We adopt this approach by embedding a data parallel language in Python.

We chose Python as the substrate language for our data parallel language because it is already in widespread use amongst the scientific community. Popular Python packages such as Numpy [70], Scipy [49], and Matplotlib [47], provide widely used libraries for numeric computation and visualization. We wish to interoperate with these libraries, in order to create a productive environment for developing full applications, not just for computationally intensive kernels.

There have been other attempts to compile various subsets of Python. The Cython compiler [29] compiles a largely Python-like language into sequential C code. Clyther [26] takes a similar approach to generating OpenCL kernels. In both cases, the programmer writes a sequential Python program that is transliterated into sequential C. Rather than transliterating Python programs, PyCUDA [56] provides a metaprogramming facility for textually generating CUDA kernel programs, as well as bindings for low-level operations needed to manage GPU memory. This allows the program to parametrically unroll loops and vary grid dimensions, among other things. We use the facilities PyCUDA provides as part of the Copperhead runtime.

Theano [87] provides an expression tree facility for numerical expressions on numerical arrays. Garg and Amaral [33] recently described a technique for compiling Python loop structures and array operations into GPU-targeted code. These projects all provide useful ways of interacting with parallel hardware and generating efficiency language code, but the programmer still expresses parallel computations isomorphically to the equivalent efficiency language code. For example, the programmer must still encode decisions about which operations should be performed in parallel and which should be performed sequentially by using substantially different language constructs, the programmer must use on-chip memory manually, and the resulting program is still explicitly tied to a particular parallel platform, because the mapping decisions the programmer makes are deeply and pervasively expressed throughout the code. Copperhead aims to solve a fairly different problem, namely compiling a program from

a higher level of abstraction into efficiently executable code.

## 2.3 Data Parallel Programming Models

There is an extensive literature investigating many alternative methods for parallel programming. Perhaps the first data parallel programming model came with APL [48], which allowed programmers to apply operations across datasets, rather than the more imperative, loop-oriented approach common in languages like Fortran. Paralation languages, such as Paralation Lisp, introduced the concept of the flattening or vectorization transform, which enabled the whole-program transformation of nested data parallel programs into flat data parallel programs [78]. Other data parallel approaches [40, 7, 9] have been proposed, historically often in close association with SIMD and vector machines, such as the the CM-2 and Cray C90, respectively.

The NESL language extended the application of the flattening transform to show that nested data parallel languages can achieve both efficiency and portability [10]. The flattening transform was further extended to fully higher-order functional languages in Data Parallel Haskell [38],[21]. In contrast to these methods, we attempt to schedule nested data parallel operations directly onto the hierarchical structure of the machines we target. Our scheduling technique allows programs generated by our compiler to execute efficiently, on par with hand-written efficiency level code.

SISAL [34] showed how data parallel languages can be compiled efficiently to clustered machines, notably making use of in-place transforms to reduce extraneous copies during program execution.

The CUDA platform [68, 79, 65] defines a blocked Single Program, Multiple Data (SPMD) programming model for executing parallel computations on GPUs. OpenCL [84] defines a similar programming model that can be applied across a variety of hardware platforms. Although CUDA and OpenCL are not strictly nested data parallel languages, their programming model depends on data parallelism, and they serve as low-level efficiency languages for data parallel substrates.

The Thrust [43] and CUDPP [27] libraries provide a collection of flat data parallel primitives for use in CUDA programs. Programming with these flat data parallel libraries provides significantly higher productivity than manual programming using CUDA or OpenCL. However, these flat data parallel approaches cannot deal with nested data parallelism, and require the programmer to manually flatten their programs to allow them to be expressed using these libraries. Copperhead uses selected Thrust primitives in the code it generates, when flat data parallel operators are required.

Systems for compiling flat data parallel programs for GPU targets have been built in a number of languages, including C# [86], C++ [64, 63], and Haskell [59]. Such systems typically define special data parallel array types and use operator overloading and metaprogramming techniques to build expression trees describing the computation to be performed on these arrays. The Ct [36] library adopts a similar model for programming more traditional multicore processors. However, these systems have not provided a means to automatically map nested data parallel programs to a hierarchically nested parallel platform.

Rather than providing data parallel libraries, others have explored techniques that mark up sequential loops to be parallelized into CUDA kernels [60, 90, 37]. In these models, the programmer writes an explicitly sequential program consisting of loops that are parallelizable. The loop markups, written in the form of C *pragma* preprocessor directives, indicate to the compiler that loops can be parallelized into CUDA kernels.

Although there have been many data parallel programming models over the years, none of them can be efficiently compiled to contemporary parallel processors. The parallelism hierarchy provided by today's parallel microarchitectures encourages the expression of nested data parallelism, which can be mapped neatly onto hardware structures. Expressing computations with flat data parallel programming models is inefficient. Manually flattening computation yields implementations that do not use memory bandwidth efficiently, since intermediate data structures must be fully realized, and also introduces extraneous synchronization, which limits the amount of parallelism that can be realistically sustained. Programming models that require the use of pragmas are difficult to use, since semantics of the program are entangled in auxiliary syntax that does not match the remainder of the program. Low-level models such as CUDA and OpenCL allow high performance, but at a significant productivity cost. Data parallel compilers of the past targeted significantly different hardware, before the advent of on-chip multiprocessing, which encouraged them to use compilation techniques that are deleterious to efficiency on contemporary parallel hardware, as we will detail in Chapter 4.

Copperhead aims to solve these problems by making use of nested data parallel abstractions and compiling efficiently to contemporary hardware, endeavoring to bridge the gap between high-level descriptions of a computation and efficient low-level implementation. We detail how this is done in the following chapters.

## 2.4 Summary

In summary, data parallelism is a fundamental pattern in parallel computing. There have been many architectures conceived to support data parallelism over the years, as well as many data parallel programming models. Our work aims to translate higher-level representations, comprised of nested data parallel computations, onto contemporary parallel architectures. To do this, we embed a data parallel language in Python, which is defined in the next chapter.

# 3 The Copperhead Language

As mentioned in the previous chapter, preexisting high-level data parallel languages either cannot be compiled to contemporary parallel hardware, or their compilation approach cannot yield efficient performance. To overcome these issues, we introduce Copperhead, a data parallel language embedded in Python. A Copperhead program is a Python program that imports the Copperhead language environment:

```
from copperhead import *
```

A Copperhead program is executed, like any other Python program, by executing the sequence of its top-level statements.

Selected procedure definitions within the body of the program may be marked with the Copperhead decorator, as in the following:

```
@cu
def add_vectors(x, y):
    return map(lambda xi,yi: xi+yi, x, y)
```

This `@cu` decorator declares the associated procedure to be a Copperhead procedure. These procedures must conform to the requirements of the Copperhead language, and they may be compiled for and executed on any of the parallel platforms supported by the Copperhead compiler. Once defined, Copperhead procedures may be called just like any other Python procedure, both within the program body or, as shown below, from an interactive command line.

```
>>> add_vectors(range(10), [2]*10)
[2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

The `@cu` decorator interposes a wrapper object around the native Python function object that is responsible for compiling and executing procedures on a target parallel platform.

Copperhead is fundamentally a data-parallel language. It provides language constructs, specifically `map`, and a library of primitives such as `reduce`, `gather`, and `scatter` that all have intrinsically parallel semantics. They operate on one dimensional arrays of data that we refer to as *sequences*.

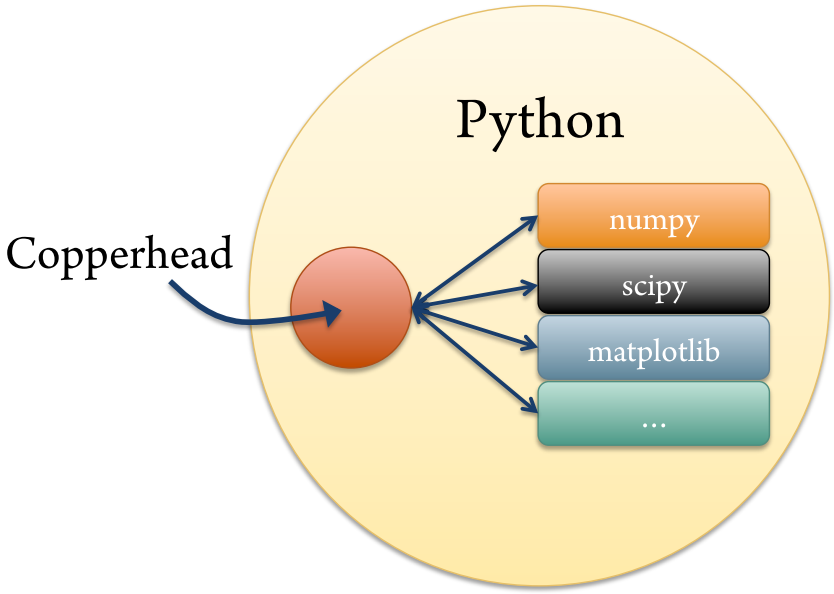



Figure 3.1: Copperhead is an Embedded Subset of Python, designed to interoperate with standard Python libraries for numeric computing

## 3.1 Restricted Subset of Python

The Copperhead language is a restricted subset of the Python 2.6 language [76]. Every valid Copperhead procedure must also be a valid Python procedure. Portions of the program outside procedures marked with the `@cu` decorator are unaffected by these restrictions, as they are normal Python programs that will be executed by the standard Python interpreter. The Copperhead language supports a very restricted subset of Python in order to enable efficient compilation. Figure 3.1 represents the relationship between Copperhead and Python.

Copperhead is designed as an embedded language for three reasons. First, for programmers learning Copperhead, the experience is more similar to learning the Application Programming Interface of a Python package, rather than learning a new language. There is no need to learn a new set of syntactic constructs or deal with an unfamiliar compilation flow. Modern productivity languages like Python are very flexible, allowing for the widespread proliferation of embedded domain specific languages. We expect this to be a common mechanism for exploiting parallelism in domain specific ways [20, 11].

Second, embedding Copperhead in Python allows Copperhead to interact with the wide array of Python software for numeric and scientific computing. For example, Copperhead programs can interoperate with the popular `numpy`, `scipy`, and `matplotlib` packages, which provide useful data visualization and serialization capabilities. The goal of Copperhead is to be a productive environment for writing entire programs, not just their computationally intensive data parallel kernels. Embedding Copperhead in Python allows programmers to use familiar,

```
E : x | (E1, ..., En) | A[E]
    | True | False | integer | floatnumber

    | E1 + E2 | E1 < E2 | not E | ...
    | E1 and E2 | E1 or E2 | E1 if Ep else E2

    | F(E1, ..., En) | lambda x1, ..., xn: E

    | map(F, A1, ..., An)
    | [E for x in A]
    | [E for x1, ..., xn in zip(A1, ..., An)]
```

Figure 3.2: Grammar for Copperhead Expressions

highly productive software to create their programs.

Third, embedding Copperhead in Python has several practical advantages. For example, since Copperhead procedures are all syntactically valid Python procedures, we do not need to write a parser for Copperhead programs, but can instead use the standard Python `ast` module. Additionally, the task of writing a compiler is simplified by Python's concise and object-oriented style.

Copperhead adopts the lexical and grammatical rules of Python. In summarizing its restricted language, we denote expressions by $E$ and statements by $S$. We use lower-case letters to indicate identifiers and $F$ and $A$ to indicate function-valued and array-valued expressions, respectively.

### 3.1.1 Expressions

Figure 3.2 shows the grammar for Copperhead expressions. We explain this grammar, group by group.

The most basic expressions are literal values, identifier references, tuple constructions, and accesses to array elements.

The basic logical and arithmetic operators are allowed, as are Python's `and`, `or`, and conditional expressions.

Expressions that call and define functions must use only positional arguments. Optional and keyword arguments are not allowed. This restriction is not fundamental to the design of the language, but is rather an implementation detail that may change in the future.

Copperhead relies heavily on `map`, which applies a function element-wise to a sequence or set of sequences. In Copperhead, `map` is the fundamental source of parallelism and is elevated from a built-in function (as in Python) to a special form in the grammar in order to conform to our type system. Copperhead also supports a limited form of Python's list comprehension syntax, which is transliterated into equivalent `map` constructions during parsing.

```
S : return E
    | x_1, ..., x_n = E
    | if E: suite else: suite
    | def f(x_1, ..., x_n): suite
```

Figure 3.3: Grammar for Copperhead Statements

### 3.1.2 Statements

Figure 3.3 shows the grammar for Copperhead statements. The body of a Copperhead procedure consists of a *suite* of statements: one or more statements $S$, which are nested by indentation level according to Python's normal syntax. Each statement $S$ of a suite must be of the following form: Copperhead further requires that every execution path within a suite must return a value, and all returned values must be of the same type. This restriction is necessary since every Copperhead procedure must have a well-defined static type.

## 3.2 Type System

In addition to these grammatical restrictions, Copperhead expressions must be *statically well-typed*. We employ a minimalistic Hindley-Milner style polymorphic type system [42][30], which we overlay on top of Python's standard type system. Python programs have a very weak typing, which is often referred to as *duck typing*, meaning that if it quacks like a duck, it is a duck (and can therefore be used as a duck). In practice, this means that even the simplest of operations requires dynamic function dispatch, performed after inspecting the arguments for compatible functions.

For example, to implement the addition operator, Python objects can define the `__add__(self, other)` method. When the interpreter sees the operation `c = a + b`, it first examines `a` for the presence of an `__add__` method. The correspondence between the + operator and the `__add__` method is built into the Python language and interpreter, and documented such that programmers wishing to write new data types that can be operated on by the + operator need simply to implement a `__add__` method in their data type. If an `__add__` method is found, it is called as follows: `c = a.__add__(b)`. If this call raises a TypeError exception, which can occur if the `__add__(self, other)` method does not know how to add objects of `b`'s type to `a`, then the interpreter examines `b` for the presence of a `__radd__(self, other)` method, and if found, calls it: `c = b.__radd__(a)`. The presence of the `__radd__` method allows for the case where `a` does not know how to add itself to `b`, but `b` does know how to add itself to `a`. Falling back to `b.__add__(a)` in case of a TypeError exception when trying to execute `a+b` would not be permissible in general, since the + operator need not be commutative. The sequence in which the interpreter inspects `a` for `__add__`, and then if necessary inspects `b` for `__radd__` is defined by the language, although most Python programmers do not concern themselves with such details, even though they endow the language with tremendous flexibility. If neither `a` nor `b` contains a compatible method, an exception is raised.

```
@cu
def well_typed(a):
    if a:
        return 0
    else:
        return 1
```

```
def ill_typed(a):
    if a:
        return 0
    else:
        return True
```

Figure 3.4: Well typed and ill-typed programs

$T$ : $S$ | $P$
$S$ : $M$ | $V$
$M$ : type_constructor($S_1$, ... $S_n$)
$P$ : $\forall$ [$V_1$, ... $V_n$] $S$
$V$: identifier

Figure 3.5: A grammar for the Copperhead type system

This very dynamic dispatch brings Python tremendous flexibility and abstraction. However, as should be obvious from the description, it also incurs heavy runtime overhead penalties due to the series of costly introspections, which are performed on every object before executing every operation. Dynamic dispatch is one of the major reasons that Python code tends to perform on the order of 100$\times$ slower than compiled efficiency level code.

Static typing both provides richer information to the compiler that it can leverage during code generation as well as avoids the run-time overhead of dynamic type dispatch. Since we are designing Copperhead to achieve performance competitive with hand-written efficiency code, this is a key restriction. Future versions of the Copperhead language may expand the type system to allow for the use of type classes, similar to Haskell.

Programs that do not comply with our type system are rejected. Figure 3.4 shows two programs, one of which is well typed in the Copperhead type system, while the other is not. In this case, the ill-typed program is rejected, since both branches of the conditional do not return a value of the same type.

### 3.2.1 Definition

Figure 3.5 shows the Copperhead type system.

In Figure 3.5, $T$ is all types representable by the type system, $S$ is the set of monomorphic types representable by the type system, $M$ is the set of monotypes, $P$ is the set of polymorphic types, and $V$ are type variables used in polymorphic type construction. The set of Monotypes $M$ has several elements, with associated type constructors as shown in Table 3.1.

The type system is used with a Hindley-Milner style type inference engine [42][30] that

| Monotype | `type_constructor` | Description |
|---|---|---|
| `Fn` | `Fn(`$S_1$`,`$S_2$`)` | Function, with input $S_1$ and output $S_2$ |
| `Seq` | `Seq(`$S_1$`)` | Sequence, with element type $S_1$ |
| `Tuple` | `Tuple(`$S_1$`, ..., `$S_n$`)` | $n$-ary tuple, with element types $S_1, ..., S_n$ |
| `Int` | `Int()` | 32-bit signed integer |
| `Long` | `Long()` | 64-bit signed integer |
| `Float` | `Float()` | 32-bit floating-point number |
| `Double` | `Double()` | 64-bit floating-point number |
| `Bool` | `Bool()` | Boolean |
| `Void` | `Void()` | Void type is the type of a statement |

Table 3.1: Basic monotypes in the Copperhead type system

infers types for all data and functions referred to in a Copperhead program. Informally, the type inference system derives symbolic types for all identifiers in the program, based on how they interact with each other through known type signatures. For example, the type signature of the + operator is `∀ A Fn(Tuple(A, A), A)`, meaning that the + operator is a function that takes two arguments, both of some arbitary type `A` and returns another argument, also of the same type `A`. Accordingly, when we observe `a+b`, although we may not know what concrete types `a` and `b` are instantiated with, we know that they both must be of the same type. The type inference system first assigns all identifiers a unique type variable, and then proceeds to *unify* type variables together, given observations on how the variables interact with one another, as well as type signatures for some built-in procedures such as the + operator. After type inference, each identifier in the program, including procedure identifiers, are annotated with type information, which is often polymorphic. The Copperhead compiler treats all polymorphic types as C++ template variables, and then instantiates them in C++ code with concrete C++ types, such as `float` or `double` in a wrapper function. This instantiation creates a complete program specialized for the particular types that the Copperhead function was called with, but still keeps the majority of Copperhead generated code templatized to operate on many types, which facilitates reuse.

Since we use a Hindley-Milner type inference system, programmers are not required to annotate their code with type information. We chose to use type inference as opposed to required type annotations for two reasons: first, Python programs themselves are written without type information, and we want Copperhead code to look and feel like Python code; and second, avoiding type annotations is more productive, since the programmer does not have to specify information the compiler can derive itself.

There are situations where the type system cannot derive type information, for example, when interfacing external libraries written in efficiency languages to Copperhead, or when specifying the built-in Copperhead *prelude* functions, described in Section 3.6. Although most Copperhead programmers will never run into these situations, we have implemented a simple embedded domain specific language to specify types to the Copperhead compiler. For example, Figure 3.6 shows how the type of `reduce`, namely `∀a : Fn(Tuple(Fn(Tuple(a, a), a), Seq(a), a), a)` can be written in this language.

```
@cutype("((a, a)->a, [a], a) -> a")
def reduce(f, x, i):
    pass
```

Figure 3.6: Describing Copperhead Types

```
def xpy_bad(x, y):
    for i in indices(y)
        y[i] = x[i] + y[i]
```

```
@cu
def xpy_good(x, y):
    def duad(xi, yi):
        return xi + yi
    return map(duad, x, y)
```

Figure 3.7: Example code with and without side effects

The type derived from this description interacts with the Copperhead system as any automatically derived type would. This ability facilitates extending Copperhead to interface with external libraries, which is very important. We do not envision a world where all computationally intensive software is written in Copperhead – part of Python's strength as a programming environment is its ability to interact efficiently with software written in other ways, and we aim to provide similar capabilities for Copperhead.

## 3.3 Side Effects

All data in a Copperhead procedure are considered immutable values. Thus, statements of the form `x = E` bind the value of the expression `E` to the identifier `x`; they *do not* assign a new value to an existing variable. All identifiers are strictly lexically scoped.

In general, Python programs make some use of side effects for computation. Some data types, such as primitives like integers, as well as compound tuple data types, are immutable, meaning that they can not be operated on using side effects. However, others, such as lists and dictionaries, as well as most custom classes, can be mutated, and so side effects are commonly used in Python.

Copperhead does not allow side effects because they signficantly complicate compilation. To make this discussion concrete, see Figure 3.7, which shows two procedures that implement vector addition. The `xpy_bad` procedure 3.7 implements vector addition by destructively modifying one of its inputs. This program is illegal in Copperhead. In contrast, the `xpy_good` procedure is side effect free, and is legal in Copperhead.

In this simple example, parallelizing the `for` loop would require knowledge about `indices`: specifically, that `indices` returns a sequence of values with no repeated elements. The compiler would also need to prove that there are no loop-carried dependences in order to parallelize

iterations of the loop. Considerable research has been devoted to parallelizing compilers that perform such analyses. However, such analyses are not required for languages that do not allow side effects.

Instead of using side effects, the computation can be written in Copperhead as shown in the `xpy_good` procedure shown in Figure 3.7. The `for` loop from `xpy_bad` has been replaced with an explicitly unordered `map`, which can be interpreted as a parallel function invocation and executed in whatever ordering is most convenient for a particular parallel platform.

Programming via side effects is widely used in general, including in parallel computing. For example, the widely used Basic Linear Algebra Subroutine (BLAS) package works entirely through destructive updates. Destructive updates can improve performance and minimize memory usage in some circumstances. For example, if the programmer wishes to update two elements in a large vector containing, for the sake of example, one billion elements, this operation can be performed using side-effects by destructively updating the two elements in the vector. Programming approaches that forgo side effects must transform the old vector into the new updated vector, which in practice results in copying the old vector into a new result. In our example, this would result in an extra memory allocation of one billion elements, as well as the memory traffic necessary to copy the one billion elements from the old vector to the new vector. If memory space is constrained, this might not even be feasible. Programming with explicit side effects allows the programmer to express these operations directly.

However, as we mentioned earlier, it also significantly impedes parallelization, since the presence of side effects forces parallelizing compilers to prove that any side effects in the program are localized with respect to the parallelization being performed. Constructing this proof is not always possible, since it can involve making use of information that may be obvious to the programmer, but not to the compiler. Additionally, the use of side effects allows programmers to create programs that are so encumbered by data dependences that they are not parallelizeable at all.

Although Copperhead requires programmers to describe computations without the use of side effects, the compiler has the freedom to *implement* Copperhead programs using side effects. For example, if the Copperhead compiler can prove that a particular identifier is never reused, it is free to do in-place destructive updates internally, using the same storage to represent multiple textual identifiers in the program. This ability has been used with notable success in previous data parallel languages that also forbade side effects, such as SISAL [34].

Furthermore, the decision to embed Copperhead in an existing productivity language provides additional motivation for forbidding side effects. In many productivity languages that allow side effects, such as Python and Ruby, collective operations such as `map` have sequential semantics. The presence of side effects requires that these collective operations be executed in the same sequential order as defined in the productivity language. Figure 3.8 shows a simple Python program that operates using side effects, along with its output.

The result of the program shown in Figure 3.8 is only defined because Python's `map` construct has sequential semantics – any attempt to parallelize the `map` operation in this example would lead to the same problems that arise when parallelizing traditional loops. Therefore, any relaxed semantics that could enable parallelism would also conflict with the established sequential semantics of collective operations in the host productivity language, were we to allow

```
def example(c):
    a = []
    def side_effect(b):
        a.append(b)
        return a
    map(side_effect, c)
    return a

» print example([1,2,3,4])
» [1,2,3,4]
```

Figure 3.8: Python code with side effects

side effects. Since we disallow side effects in Copperhead, we prevent semantic mismatches between Python and Copperhead due to our relaxed ordering rules, which we discuss in the next section.

Finally, since Copperhead programs are embedded in Python, and Python allows for side effects, parts of the program that naturally need to use side effects can be written as traditional Python programs. Using Copperhead does not preclude the programmer from ever using side effects, instead we simply require that the side effects be expressed in the enclosing Python program. Of course, the Copperhead compiler does not attempt to parallelize the enclosing Python program, which runs in the standard Python interpreter as any other program might.

Although our decision to forbid side effects in Copperhead programs does impose certain restrictions on the programmer, we feel these restrictions are well justified due to the flexibility they afford the compiler, which enables efficient compilation of high level data parallel computations to parallel hardware.

### 3.3.1 Side Effects in Host Code

We acknowledge the importance of programming with side effects, and Copperhead is designed to facilitate programming with side effects for certain cases where it is more convenient or more efficient. More specifically, Copperhead comes with a set of routines that allow the programmer to mutate Copperhead data structures directly from the enclosing host code written in Python. It is also straightforward to execute sequential iterations, such as those that guide iterative solvers, by writing loops in the host Python program. During each iteration of the loop, side effect free Copperhead procedures can be called to perform the bulk of the computation, while Python code manages sequential iteration and mutation of Copperhead data structures.

```
@cu
def foo():
    return True

@cu
def bar(x):
    return map(foo, x)
```

Figure 3.9: Referencing procedure identifiers defined outside a Copperhead procedure

```
a = [1,2,3,4]

def baz(x):
    return map(op_add, x, a)
```

Figure 3.10: Referencing data identifiers defined outside a Python procedure

### 3.3.2 Loops

Data parallel languages traditionally do not provide explicit loop constructs. The reason for this is straightforward: standard loops encode a particular execution ordering and operate via mutation of an index variable. Typically, data parallel languages guide the programmer to express computations in terms of unordered operations on data structures in an applicative, functional style without side effects. Unordered operations can be parallelized arbitrarily, and as we have pointed out, the lack of side effects enables a wide range of compiler transformations that are essential to high performance. Accordingly, we disallow loops in Copperhead. If sequential, ordered iteration is required, the programmer can use tail recursion for a similar effect: the Copperhead compiler will convert tail recursion into stateful loops. Alternatively, the programmer can express the sequential loop in Python, as we previously discussed.

## 3.4 Scoping and Ordering

Copperhead applies strict lexical scoping to all variables encountered in a Copperhead program. Importantly, Copperhead procedures may reference identifiers defined outside of procedure scope only if those identifiers refer to functions. For example, Figure 3.9 is valid Copperhead code. In procedure `bar`, the only identifier referenced that is defined outside the scope of `bar` is the function `foo`:

In contrast, Figure 3.10 shows Python code that would not be valid Copperhead code, since procedure `baz` references a non-function identifier defined outside the scope of `baz`.

Additionally, Python employs dynamic scoping, while Copperhead uses strict lexical scoping. For example, Figure 3.11 shows a Python program that relies on dynamic scoping; such

```
def dynamic_scoping(x):
    if x:
        y = 1
    else:
        y = 2
    return y
```

Figure 3.11: Dynamic scoping in Python

```
@cu
def axpy(a, x, y):
    def triad(xi, yi):
        return a * xi + yi
    return map(triad, x, y)
```

Figure 3.12: Illustrating Closures

programs are rejected by the Copperhead type system. In practice, Copperhead's scoping rules are enforced by the restriction that Copperhead programs are written without loops, with the constraint that all branches of a conditional must return a value.

Copperhead does not guarantee any particular order of evaluation, other than the partial ordering imposed by data dependencies in the program. Python, in contrast, always evaluates statements from top to bottom and expressions from left to right. By definition, a Copperhead program must be valid regardless of the order of evaluations, and thus Python's mandated ordering is one valid ordering of the program.

We use this flexibility, enabled by our choice to forbid mutable assignment, to reorder and transform procedures in a number of ways, which greatly improves the efficiency of generated code from the Copperhead compiler.

## 3.5 Closures

Closures, or functions with free variables bound in the lexical environment [89], are commonly used in Python and other efficiency languages. Copperhead supports closures in a limited fashion. For example, in Figure 3.12, the identifier `a` is closed over by the procedure `triad`. The use of `a` in `triad` performs a broadcast of that identifier to each of the invocations of `triad` that are created by the `map`. Since `map` performs element-wise operations on a set of input sequences, closures provide the mechanism for auxiliary data to be used in a non-element-wise access pattern during `map` operations. Closures are also used for determining the data that should be kept on chip; this process is explained in Section 4.9.

Copperhead does not support unlimited higher order operations. For example, closures must remain limited to the scope in which they are defined; they cannot be returned from a

function as an ordinary variable. Many productivity languages allow flexible use of closures, they can be a mechanism to encapsulate data with procedures, to create coroutines and continuations, for example. Our usage of closures is more restrictive, which enables us to remove indirection and create efficient binaries.

## 3.6 Data-Parallel Primitives

Copperhead is a data-parallel language. Programs manipulate data sequences by applying aggregate operations, such as `map` or `reduce`. The semantics of these primitives are implicitly parallel: they may always be performed by some parallel computation, but may also be performed sequentially. Copperhead provides a *prelude* of built-in standard data-parallel aggregate operations, which are composed by the programmer to form a computation and imported into the Python scoping environment when the programmer writes a Copperhead program. We now define selected Copperhead primitives.

### 3.6.1 `map`

In Copperhead, `map` is the workhorse of data parallelism. It is used to describe parallel function invocation. Python itself provides a builtin `map` construct, defined as `map`(*function, iterable, ...*), which applies *function* to every item of *iterable*. If additional *iterable* arguments are provided, the *function* is given an element from each *iterable*. Equivalently,

```
map(f, x1, ..., xn) = [f(x1[0], ..., xn[0]), f(x1[1], ..., xn[1]), ...]
```

Copperhead's `map` mirrors this syntax. In fact, in order to comply with Copperhead's type system, `map` is elevated to a special syntax tree node, since it processes a variable number of inputs, which is not allowed for arbitrary Copperhead functions.

As in Python, when called on nested data structures, `map` treats each subsequence as an item of the overall sequence.

There are three differences between Copperhead `map` and Python `map`. First, Python `map` allows the *iterable* arguments it consumes to be of different lengths. Arguments that are shorter than the others are extended with `None` items. In contrast, Copperhead requires that arguments to `map` be of identical lengths, and calling `map` on arguments with different lengths generates a runtime error. Second, Copperhead `map` returns a Copperhead sequence, whereas Python `map` returns a Python list in Python version 2, or a Python iterator in version 3. Although Copperhead sequences are designed to be compatible with Python lists and can return iterators, they are not identical, for various technical reasons. In future versions of Copperhead that operate in Python 3, it would be possible to make Copperhead data structures iterators and comply fully with the semantics of Python's `map`. Finally, Python `map` can operate with a nullary *function*: if the special Python identifier `None` is passed in as the function, `map` will return a copy of the input arguments, as if the appropriate arity identity function has been provided. Copperhead does not allow this, since we do not have the special Python identifier `None`. Equivalent functionality, if desired, can be obtained by providing an identity function, or simply using an assignment or `zip` statement.

### 3.6.2 `zip`

Python provides a builtin `zip` construct, defined as `zip`([*iterable, ...*]) that returns a list of tuples, gathered from each of the *iterable* arguments. Copperhead also provides `zip`, although with the caveat that the user must provide exactly two sequences, and the result is a sequence of pairs.

Zip is useful for constructing sequences of tuples from flat sequences. For example, to perform an `arg max` operation, one might zip an index sequence together with a data sequence, to create a sequence of labeled data, which then could be operated on with a `reduce` primitive to find the maximum element along with its position in the sequence. This corresponds to notionally converting a Structure of Arrays into an Array of Structures. However, the Copperhead compiler is free to implement data structures in arbitrary ways, and temporary data structures created by the Copperhead compiler are kept in Structure of Arrays format for maximum performance, even though the program treats them as Arrays of Structures. More details about this are in Section 4.10.

More explicitly,

```
zip(a, b) = [(a[0], b[0]), (a[1], b[1]), ...]
```

Copperhead only supports `zip` with two sequences because it makes `zip` have a well defined type in the type system. For zipping together more sequences, the Copperhead prelude contains several variants: `zip3(a, b, c)`, `zip4(a, b, c, d)`, etc.

Python allows the arguments to `zip` to be of different lengths. However, similarly to `map`, Copperhead requires them to be of the same length.

Copperhead also provides an `unzip`, which takes a list of tuples, and returns a tuple of lists. Python applies an overloaded * operator for this purpose, which we do not support, since we do not support overloaded operators in general.

### 3.6.3 `reduce`

Python provides a built-in `reduce` function, defined as `reduce`(*function, iterable*[*, initializer*]). `reduce` applies a binary *function* sequentially to reduce a sequence into a single value, starting with the value *initializer*, if present. Copperhead provides a similar facility:

`reduce(`$\oplus$`, x, z) =` $\left(\bigoplus_{x_i \in \mathtt{x}} x_i\right) \oplus \mathtt{z}$

However, Copperhead places a few additional restrictions on this operation. Firstly, the binary operator $\oplus$ must be both associative and commutative, which allows for the reduction to be parallelized. Secondly, the initializer is not optional. This is required in order to keep the operation well defined for the case when the sequence `x` is empty. Python raises a `TypeError` for the case when the *iterable* is empty and the *initializer* is not provided. We choose simply to require the initial value.

Parallel reductions can be complicated when invoked with pseudo-associative operators, since the order in which the reduction is performed can result in different answers. Unfortunately, floating-point addition, which may be the most common reduction operator, suffers from this problem. This problem is common to all parallel reductions. Copperhead has the

ability to generate fully repeatable reductions, which provide a unique total ordering for every reduction of a given length, thus ensuring repeatability of results, even for this case.

The most common reduction is a simple `sum` operation, where the operator is numerical addition, and the prefix is 0. We provide `sum(x)` as a Copperhead primitive.

### 3.6.4 `scan`

Closely related to `reduce` is the `scan` collective operation, or equivalently, prefix sum. Scan also requires a binary function $\oplus$, and applies it across a sequence `x`, but instead of returning a scalar value, it returns a sequence of partial sums. Equivalently,

`x[0]` $\oplus$ `x[1], ...,` $\bigoplus_{x_i \in \mathtt{x}} x_i$`]`

For example, `scan(op_add, [1, 2, 3, 4]) = [1, 3, 6, 10]`. This primitive is very important in data parallel programming because the computation of partial sums can be parallelized, given restrictions on the binary operator $\oplus$, even though the result appears to have been produced by a sequential loop. For more explanation about the utility of `scan` and variants, see [81]. In this case, similarly to `reduce`, we require the binary operator $\oplus$ to be both associative and commutative. No initializer is required for inclusive scan: if an empty sequence is provided to scan the result is defined to be an empty sequence.

The standard prefix sum operation in Copperhead is the *inclusive scan* operation, however we also provide *exclusive scan* operations (`exscan`) as well as reverse inclusive (`rscan`) and exclusive (`exrscan`) scan operations. Exclusive scan operations require an additional argument for the initializer, which must be of the same type as the elements of `x`.

### 3.6.5 `gather`

Copperhead provides a `gather` primitive to enable indirection, such as found in sparse data structures. Given two sequences, a data sequence `x`, and an index sequence `i`,

```
gather(x, i) = [x[i[0]], x[i[1]], ... ]
```

### 3.6.6 `scatter`

The mirror image of `gather` is `scatter`. Given three sequences: a data sequence `x`, an index sequence `indices`, and a destination sequence `y`, then `z=scatter(x, indices, y)` is defined:

$$\mathtt{z[i]} = \begin{cases} \mathtt{x[t]} & \mathtt{i} \in \mathtt{indices}, \mathtt{t} = \arg_{\mathtt{i}} \mathtt{indices} \\ \mathtt{y[i]} & \mathtt{i} \notin \mathtt{indices} \end{cases}$$

This operation can be conceptualized as making a copy of `y`, and then scattering elements of `x` into it according to the array of `indices`. Copperhead requires the `y` array so that `scatter` is well defined even if `indices` does not cover every element of the output.

The result of `scatter` has undefined semantics for the case where `indices` has repeated elements.

```
@cu
def spmv_csr(A_values, A_columns, x):
    def spvv(Ai, j):
        z = gather(x, j)
        return sum(map(lambda Aij, xj: Aij*xj, Ai, z))

    return map(spvv, A_values, A_columns)
```

Figure 3.13: Procedure for computing $Ax$ for a matrix $A$ in CSR form and a dense vector $x$. Underlined operations indicate potential sources of parallel execution.

### 3.6.7 `permute`

`permute` is a close relative of `scatter`, the only difference being that the programmer, by using `permute`, asserts that the `indices` sequence is a permutation of the indices of `x`. In this case, there is no need for the source array `y`, since we know that every element of the result will come from `x`.

If `z=permute(x, indices)`, then

`z[i] = x[t], t=arg`$_{\texttt{i}}$ `indices`

If the programmer supplies an `indices` sequence that is not a permutation of the indices of `x`, the results are undefined.

### 3.6.8 `indices`

`indices(x)` creates a sequence with the same shape as `x`, but with each element counting the indices of `x`. For a sequence `x` with length `n`:

```
indices(x) = [0, 1, ..., n-1, n]
```

## 3.7 Example programs

Copperhead programs are constructed via composition of data parallel primitives from the prelude. To illustrate how this is accomplished, we present two simple Copperhead programs.

### 3.7.1 Compressed Sparse Row Sparse Matrix Vector Multiplication

Figure 3.13 shows a simple Copperhead procedure for computing the sparse matrix-vector product (SpMV) $y = Ax$. Here we assume that $A$ is stored in Compressed Sparse Row (CSR) format—one of the most frequently used representation for sparse matrices—and that $x$ is a dense vector. The matrix representation simply records each row of the matrix as a sequence

containing its non-zero values along with a corresponding sequence recording the column index of each value. A simple example of this representation is:

$$A = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix}$$

```
vals = [[1,7],[2,8],[5,3,9],[6,4]]
cols = [[0,1],[1,2],[0,2,3],[1,3]]
```

The body of `spmv_csr` applies a sparse dot product procedure, `spvv`, to each row of the matrix using `map`. The sparse dot product itself produces the result $y_i$ for row $i$ by forming the products $A_{ij}x_j$ for each column $j$ containing a non-zero entry, and then summing these products together. It uses `gather` to fetch the necessary values $x_j$ from the dense vector $x$ and uses `sum`, a convenient special case of `reduce` where the operator is addition, to produce the result.

This simple example illustrates two important issues that are a central concern for our Copperhead compiler. First, it consists of a number of potentially parallel aggregate operations, which are underlined. Second, these aggregate operators are nested: those within `spvv` are executed within an enclosing `map` operation invoked by `spmv_csr`. One of the central tasks of our compiler is to decide how to schedule these operations for execution. Each of them may be performed in parallel or by sequential loops, and the nesting of operations must be mapped onto the hardware execution units in a suitable fashion.

### 3.7.2 Radix Sort

To further illustrate the use of data parallel primitives, we present a radix sort implementation using a scan based algorithm, as presented by [7]. Figure 3.14 shows the Copperhead code for this procedure, which sorts `A` using bits `[lsb, msb)` as the key. To illustrate how this sorting algorithm works, we show the execution of a very simple example in Figure 3.15. This procedure iterates over the bits of an integral-valued data type, from least significant bit to most significant bit, at each step separating the data by whether the bit is set. Data elements where the bit is not set are moved to the beginning of the array, thus sorting the data by that bit. Performing this separation entails computing the new index for every element, which depends on how many elements before and after it have their bit set. Prefix sum operations, `scan` and `rscan`, are used to derive the distance each data element must move, and then the data is `permuted`. Tail recursion performs iteration over the bits of the datatype being sorted.

## 3.8 Conclusion

This chapter has described the design of Copperhead, a data parallel programming language embedded in Python. Copperhead is designed to be as simple of a language as possible, both to ease the task of creating a compiler for Copperhead, as well as to make learning Copperhead as easy as possible. We presented the subset of Python that is supported by Copperhead, detailed its type system and limitations, and presented the Copperhead *prelude*, a set of data parallel

```
# This procedure takes three inputs:
#    A:   The sequence of data to be sorted
#    msb: Sort up to this bit
#    lsb: Sort starting from this bit
# For 32-bit data, you would call this procedure with:
#    lsb = 0, msb = 32

@cu
def radix_sort(A, msb, lsb):
    if lsb>=msb:
        # Done sorting
        return A
    else:
        # Sort the lsb'th bit
        # First, figure out which elements have the lsb'th bit set
        bits  = map(lambda x: (x>>lsb)&1, A)
        # Then count all the ones, starting from the front
        ones  = scan(op_add, bits)
        # Then count all the zeros, starting from the back
        zeros = rscan(op_add, [b^1 for b in bits])

        # Then compute offsets, which describe how far each element
        # should move
        offsets = map(delta, bits, ones, zeros)

        # Permute the input data to sort with respect to the lsb'th bit
        A = permute(A, map(op_add, indices(A), offsets))
        # Iterate
        return radix_sort(A, msb, lsb+1)

# This procedure operates on three scalars, and
# simply selects between them
@cu
def delta(bit, ones_before, zeros_after):
    if bit==0:  return -ones_before
    else:       return +zeros_after
```

Figure 3.14: Radix sort in Copperhead

```
A   = [0, 3, 2, 1]
msb = 2
lsb = 0

bits    = [0, 1, 0, 1]
ones    = [0, 1, 1, 2]
zeros   = [2, 1, 1, 0]
offsets = [0, 1, -1, 0]

A   = [0, 2, 3, 1]
msb = 2
lsb = 1

bits    = [0, 1, 1, 0]
ones    = [0, 1, 2, 2]
zeros   = [2, 1, 1, 1]
offsets = [0, 1, 1, -2]

A   = [0, 1, 2, 3]
```

Figure 3.15: Illustrating Radix Sort

primitives provided by Copperhead that form the foundation for expressing computation in Copperhead. We also showed how these elements contribute to the expression of a few simple example programs in Copperhead.

Now that we have defined a simple embedded data parallel language, the important task is to compile it efficiently to modern parallel hardware. The next chapter details how we accomplish this.

# 4 Compiling Data Parallel Languages

In this chapter, we discuss techniques for compiling data parallel languages onto modern parallel microprocessors, motivated by the Copperhead language.

## 4.1 Source to source compilation

The biggest problems in implementing data parallel computations are not in low-level sequential code generation, as typified by traditional code generation problems such as register allocation, constant propagation, and dead code elimination. Classical compiler optimizations for sequential code remain crucial to performance, but may be conducted as normal once the computation has been mapped onto a parallel platform. Instead, problems implementing data parallel computations come from higher-level considerations, such as how computations are scheduled onto the target platform, and how on-chip memory is utilized.

Accordingly, we do not consider the implementation of traditional low-level code generation problems. Instead, we utilize existing compilers to do these optimizations for us, by compiling data parallel computation into scheduled, imperative source code. In other words, we perform source-to-source compilation, which significantly reduces the scope of the compilation problem we must solve.

We have designed our compiler to support three basic usage patterns. First is what we refer to as Runtime Static Compilation, which we describe in more detail in Section 5.2.1. When the programmer invokes a `@cu`-decorated function either from the command line or from a Python code module, the Copperhead runtime may need to generate code for this procedure if none is already available. Second: batch compilation where the Copperhead compiler is asked to generate a set of C++ code modules for the specified procedures. This code may be subsequently used either within the Copperhead Python environment or linked directly with external C++ applications. The third common scenario is one where the compiler is asked to generate a collection of variant instantiations of a Copperhead procedure in tandem with an autotuning framework for exploring the performance landscape of a particular architecture.

In the following discussion, we assume that the compiler is given a single top level Copperhead function—referred to as the "entry point"—to compile. It may, for instance, be a function like `spmv_csr` that has been invoked at the Python interpreter prompt. For the CUDA platform, the compiler will take this procedure, along with any procedures it invokes, and produce a single sequential *host* procedure and one or more parallel *kernels* that will be invoked by

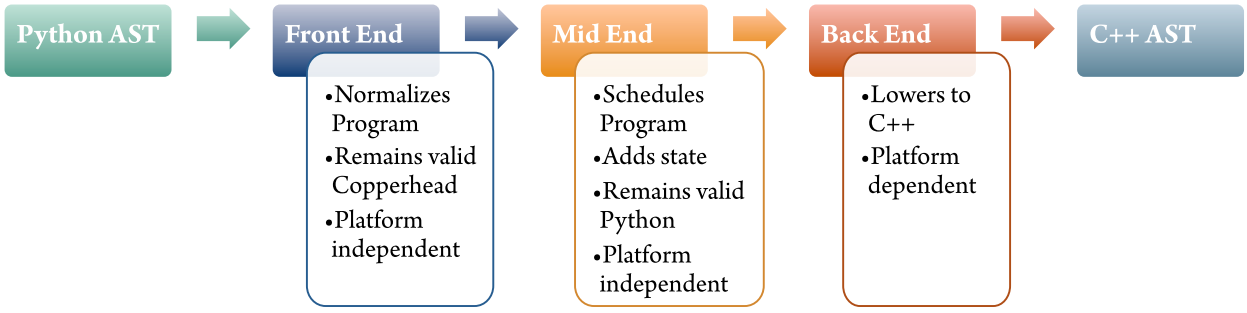



Figure 4.1: Copperhead Compiler Flow

the host procedure.

Figure 4.1 shows the high-level flow of the Copperhead Compiler. As input, the compiler accepts a standard Python Abstract Syntax Tree (AST); the compiler ultimately produces a parallelized C++ AST that implements the original computation. The compilation process is divided into three phases. The front end of the compiler is responsible for normalizing the program, producing a transformed Copperhead program that is easier to analyze and transform. The mid end of the compiler is responsible for program analysis, scheduling the program onto the target platform, and preparing the program for lowering into parallel C++. The result of the mid end is a valid Python program that implements the original computation, along with a set of annotations about type and shape information. The mid end of the compiler performs these transformations in a platform independent way. The back end of the compiler lowers the scheduled code into a parallel efficiency language in a platform specific way. More information about how this compilation flow is embedded in Python is found in Chapter 5.

## 4.2 Normalized Form

Before proceeding with the analysis and rewriting passes that transform a data parallel computation into scheduled imperative code, we consider a normalized form into which data parallel computations must be transformed in order for our analysis and rewriting passes to function correctly.

To begin, we assume that the compiler is invoked on a particular function. This assumption derives from the nature of the programs we consider compiling: since the data parallel computation is embedded in a source language, there is a clear boundary between the computation being compiled and the surrounding computation. The standard interpreter for the productivity language serving as substrate for the embedded language executes all surrounding computation normally. Therefore, the execution model is that the substrate language makes a call to the embedded language, and the compiler has visibility of the entire computation that must be performed before returning control to the host program running in the substrate language. Accordingly, it is possible to gather the entire computation into one abstract syntax tree, so that analyses and rewriting passes can operate across procedural boundaries in the original

```
@cu
def axpy(a, x, y):
    def duad(xi, yi):
        return a * xi + yi
    return map(duad, x, y)
```

Figure 4.2: Scaled vector addition before Closure Conversion

```
@cu
def axpy(a, x, y):
    def duad(xi, yi, _k0): #Note the explicit closure argument
        return _k0 * xi + yi
    return map(closure([a], duad), x, y)
```

Figure 4.3: Scaled vector addition after Closure Conversion

source code.

Therefore, the normalized form of the program includes the source code to all modules that are called from the program being compiled. If the source code is not available, as is the case with external or binary-distributed libraries, the normalized form includes simple stubs for function calls to these libraries, which indicate that no source code is available, and therefore no optimizations can be performed across function boundaries. Section 5.2.1 describes the interaction between the Copperhead compiler and external code in more detail.

Furthermore, the operations outlined in the following subsections must be performed.

### 4.2.1 Closure Conversion

Closure conversion is a classical compiler transformation that makes all variables explicit, even if they are being captured from an enclosing scope. This transformation eases compiler analysis, since after closure conversion, all data referred to by a procedure is defined in its parameter list, making the data dependences explicit. It also facilitates procedure flattening, since after closure conversion, all nested procedures are guaranteed to operate exclusively on locally defined variables.

For example, Figure 4.2 shows a simple program that implements scaled vector addition. Note that the nested procedure `duad` refers to a variable `a`, which was not defined within `duad`, but instead in the enclosing scope: the procedure `axpy`. Figure 4.3 shows the result of closure conversion on this example: the nested procedure `duad` has had its parameter list augmented with a parameter representing the closed over variable, and the call to `duad` is done through an explicit closure object that captures the variable `a` from the body of `axpy`, and passes it to each invocation of `duad` from within the closure.

```
@cu
def foo(x):
    x = x + 1
    y = x + 1
    y = y + 1
    return y
```

Figure 4.4: A procedure before Single Assignment Conversion

```
@cu
def foo(x):
    _x0 = x + 1 #Note that every identifier is assigned to exactly once
    _y0 = _x0 + 1
    _y1 = _y0 + 1
    return _y1
```

Figure 4.5: A procedure after Single Assignment Conversion

### 4.2.2 Single Assignment Conversion

Single assignment conversion is another standard compiler transformation, which eases the task of program analysis and restructuring by ensuring that every identifier is assigned to exactly once, thereby making *use-def chains* [1] explicit and unambiguous. Single assignment form is important for some of the transformations the Copperhead compiler performs, such as data parallel primitive fusion.

Traditional static single assignment form introduces Φ functions, which mark the way the control flow graph has reconverged, since conditionals may result in the same identifier being assigned in different ways. The Copperhead language is sufficiently restrictive that Φ functions are not required, specifically because of the following two properties of Copperhead code:

- All control paths within a Copperhead program must return a value
- Loops are not allowed

Due to these properties, the dominance frontier set of all nodes in a Copperhead control flow graph is empty by construction, so no Φ functions are necessary when converting Copperhead code to single assignment form.

Figure 4.4 shows a procedure before single assignment conversion. After single assignment conversion, the procedure is transformed into the code shown in Figure 4.5.

### 4.2.3 Procedure Flattening

After closure conversion, nested procedures are independent and can be flattened. This process is shown in Figure 4.6 and Figure 4.7. This transformation helps with program analysis,

```
@cu
def axpy(a, x, y):
    def duad(xi, yi, _k0):
        return _k0 * xi + yi
    return map(closure([a], duad), x, y)
```

Figure 4.6: Scaled vector addition before procedure flattening

```
@cu
def duad(xi, yi, _k0):
    return _k0 * xi + yi

@cu
def axpy(a, x, y):
   return map(closure([a], duad), x, y)
```

Figure 4.7: Scaled vector addition after procedure flattening

kernel fusion and fission, as well as satisfies the requirements of efficiency level languages, most of which do not allow nested procedures. Since we are discussing a source-to-source compilation flow, ultimately the result of our compiler must comply with the restrictions of the target efficiency language, which further motivates this transformation.

### 4.2.4 Expression Flattening

Figure 4.8 shows our `axpy` example code before expression flattening. Note the compound expression in `duad`. After single assignment conversion, compound expressions are broken into simple expressions, as shown in Figure 4.9. Expression flattening is important because it ensures that every statement of the program consists of a single operation, thus allowing the program to be restructured at statement boundaries, without the need to disentangle nested operations.

### 4.2.5 Inlining

The Copperhead compiler also performs aggressive inlining. For example, Figure 4.10 shows a simple procedure before inlining, and Figure 4.11 shows the same procedure after inlining. Inlining is an important part of normalization because it instantiates all the data parallel primitives in a unique context, which then allows them to be treated uniquely during compilation. For example, the same procedure may be mapped to different levels of the parallelism hierarchy, depending on how it appears in the program. Inlining allows each instance of a procedure to be transformed separately.

```
@cu
def duad(xi, yi, _k0):
    return _k0 * xi + yi

@cu
def axpy(a, x, y):
   return map(closure([a], duad), x, y)
```

Figure 4.8: Scaled vector addition before expression flattening

```
@cu
def duad(xi, yi, _k0):
    _tmp0 = _k0 * xi #Note that compound expressions have been flattened
    return _tmp0 + yi

@cu
def axpy(a, x, y):
   return map(closure([a], duad), x, y)
```

Figure 4.9: Scaled vector addition after expression flattening

```
@cu
def vadd(x, y):
    return map(op_add, x, y)

@cu
def foo(a, b, c):
    d = vadd(a, b)
    return vadd(d, c)
```

Figure 4.10: Example procedure before inlining

```
@cu
def foo(a, b, c):
    d = map(op_add, a, b) #Note that calls to functions have been inlined
    return map(op_add, d, c)
```

Figure 4.11: Example procedure after inlining

```
# lambda0 :: (a, a) → a
def lambda0(Aij, xj):
    return op_mul(Aij, xj)

# spvv :: ([a], [Int], [a]) → [a]
def spvv(Ai, j, _k0):
    z0 = gather(_k0, j)
    tmp0 = map(lambda0, Ai, z0)
    return sum(tmp0)

# spmv_csr :: ([[a]], [[Int]], [a]) → [a]
def spmv_csr(A_values, A_columns, x):
    return map(closure([x], spvv), A_values, A_columns)
```

Figure 4.12: SpMV procedure from Figure 3.13 after transformation by the front end compiler.

### 4.2.6 Final Result

For illustration, Figure 4.12 shows a program fragment representing the AST for the Copperhead procedure shown in Figure 3.13. Each procedure is annotated with its (potentially polymorphic) most general type, which we determine using a standard Hindley-Milner style type inference process as described in Chapter 3.

Once a program has been transformed into this normalized form, all nested parallelism has been flattened such that all data parallel operations in a given procedure can be executed at the same level of the parallel hierarchy.

## 4.3 Shape Analysis

Shape analysis determines, where possible, the sizes of all intermediate values. This analysis allows the back end of the compiler to statically allocate and reuse space for temporary values, which is critical to obtaining efficient performance. For the CUDA backend, forgoing shape analysis would require that every parallel primitive be executed individually, since allocating memory from within a CUDA kernel is not feasible. Executing every parallel primitive individually would lead to significant performance losses of up to an order of magnitude, as shown in Section 4.4.1. Because we wish to aggressively fuse operations together, we perform shape analysis to discover shapes of intermediate arrays statically, so that results can be preallocated.

Our shape analysis is conceptually similar to the size inference described in [23]. We use an internal representation that gives a unique name to every temporary value. We want to assign to each a *shape* of the form $\langle$`[`$d_1$`, ..., `$d_n$`], s`$\rangle$ where $d_i$ is the array's extent in dimension $i$ and `s` is the shape of each of its elements. The shape of a 4-element, 1-dimensional sequence `[5,4,8,1]` would, for example, be $\langle$`[4], Unit`$\rangle$ where `Unit=`$\langle$`[], `$\cdot\rangle$ is the shape reserved for indivisible types such as scalars and functions. Nested sequences are not required to have fully

```
@cu
def indeterminate_shape(b, x, y):
    if b:
        return x
    else:
        return y
```

Figure 4.13: A program with indeterminate shape

determined shapes: in the case where subsequences have differing lengths, the extent for the subsequences will be undefined, for example: ⟨[2], ⟨[*], Unit⟩⟩. Note that the dimensionality of all values are known as a result of type inference. It remains only to determine extents, where possible.

Although Copperhead currently operates only on one-dimensional sequences, our shape representation is designed to operate on multi-dimensional arrays as well. An $m \times n$ two dimensional array of scalar data has shape ⟨[$m$, $n$], Unit⟩. Supporting multi-dimensional arrays is future work for the Copperhead runtime and compiler.

We approach shape analysis of user-provided code as an abstract interpretation problem. We define a shape language consisting of Unit, the shape constructor ⟨D, s⟩ described above, identifiers, and shape functions extentof(s), elementof(s) that extract the two respective portions of a shape s. We implement a simple environment-based evaluator where every identifier is mapped to either (1) a shape term or (2) itself if its shape is unknown. Every primitive f is required to provide a function that we denote f.shape. This function returns the shape of its result given the shapes of its inputs. It may also return a set of static constraints on the shapes of its inputs to aid the compiler in its analysis. Some example shape-computing functions:

```
gather.shape(x, i) = ⟨extentof(i), elementof(x)⟩
zip.shape(x1, x2)  = ⟨extentof(x1),
                      (elementof(x1),elementof(x2))⟩
                        with extentof(x1)==extentof(x2)
```

The gather rule states that its result has the same size as the index array i while having elements whose shape is given by the elements of x. The zip augments the shape of its result with a constraint that the extents of its inputs are assumed to be the same. Terms such as extentof(x) are left unevaluated if the identifier x is not bound to any shape.

To give a sense of the shape analysis process, consider the spvv procedure shown in Figure 4.12. Shape analysis will annotate every binding with a shape like so:

```
def spvv(Ai, j, _k0):
    z0 :: ⟨extentof(j), elementof(_k0)⟩
    tmp0 :: ⟨extentof(Ai), elementof(Ai)⟩
    return sum(tmp0) :: Unit
```

In this case, the shapes for $z_0$, $tmp_0$, and the return value are derived directly from the shape rules for gather, map, and sum, respectively.

Shape analysis is not guaranteed to find all shapes in a program. Some identifiers may have data dependent shapes, making the analysis inconclusive. Figure 4.13 shows an example of such a procedure: the compiler cannot determine what shape this procedure returns, because it depends on the value of `b`. For cases where the shape is indeterminate, the compiler must insert barriers in order to first compute the shape of intermediate variables, allocate space for the intermediate variables, and then progress with the computation. This prohibits some important optimizations, such as data primitive fusion, but is required for correctness.

Future work involves allowing shape analysis to influence code scheduling, in an attempt to better match the extents in a particular nested data parallel problem to the dimensions of parallelism supported by the platform being targeted. For example, instead of implementing the outermost level of a Copperhead program in a parallel fashion, if the outer extent is small and the inner extent is large, the compiler may decide to create a code variant that sequentializes the outermost level, and parallelizes an inner level.

## 4.4 Data Parallel Primitive Scheduling

The front end of the compiler carries out platform independent transformations in order to prepare the code for scheduling. The middle section of the compiler is tasked with performing analyses and scheduling the program onto a target platform.

At this point in the compiler, a Copperhead program consists of possibly nested compositions of data parallel primitives. A Copperhead procedure may perform purely sequential computations, in which case our compiler will convert it into sequential C++ code. Copperhead makes *no attempt* to auto-parallelize sequential codes. Instead, it requires the programmer to use explicitly parallel primitives that the compiler will *auto-sequentialize* as necessary. Our compilation of sequential code is quite straightforward, since we rely on the host C++ compiler to handle all scalar code optimizations, and our restricted language avoids all the complexities of compiling a broad subset of Python that must be addressed by compilers like Cython [29].

Copperhead supports both flat and nested data parallelism. A flat Copperhead program consists of a sequence of parallel primitives that perform purely sequential operations to each element of a sequence in parallel. A nested program, in contrast, may apply parallel operations to each sequence element. Our `spmv_csr` code, shown in Figure 3.13, provides a simple concrete example. The outer `spmv_csr` procedure applies `spvv` to every row of its input via `map`. The `spvv` procedure itself calls `gather`, `map`, and `sum`, all of which are potentially parallel. The Copperhead compiler must decide how to map these potentially parallel operations onto the target hardware. This mapping depends on the composition of the program – the exact same code fragment may end up being implemented in very different ways, depending on the context in which it is instantiated. For example, a `map` operation may be launched in parallel across the machine, may be launched in parallel across a subset of the machine, or may be launched as a sequential loop. Copperhead employs a direct scheduling approach that implements nested parallelism directly on the target platform, without interposing a vectorization transform, as is commonly done in other data parallel compilers. These approaches provide maximum per-

```
@cu
def vadd(x, y):
    return map(op_add, x, y)

@cu
def vmul(x, y):
    return map(op_mul, x, y)

@cu
def of_spmv(du, dv, width, m1, m2, m3, m4, m5, m6, m7):
    e = vadd(vmul(m1, du), vmul(m2, dv))
    f = vadd(vmul(m2, du), vmul(m3, dv))
    e = vadd(e, vmul(m4, shift(du, -width, 0.0)))
    f = vadd(f, vmul(m4, shift(dv, -width, 0.0)))
    e = vadd(e, vmul(m5, shift(du, -1, 0.0)))
    f = vadd(f, vmul(m5, shift(dv, -1, 0.0)))
    e = vadd(e, vmul(m6, shift(du, 1, 0.0)))
    f = vadd(f, vmul(m6, shift(dv, 1, 0.0)))
    e = vadd(e, vmul(m7, shift(du, width, 0.0)))
    f = vadd(f, vmul(m7, shift(dv, width, 0.0)))
    return (e, f)
```

Figure 4.14: Fusion Example Code

formance on contemporary parallel processors, as we detail in the following subsections.

### 4.4.1 Data Parallel Primitive Fusion

Data parallel primitive fusion is similar to loop fusion [51] in a traditional compiler: the goal is to reduce synchronization overhead and improve data locality. Properly performing fusion can have a large performance impact, since synchronization and data movement are preeminent constraints in the exploitation of contemporary parallel processors.

For example, consider the code in Figure 4.14. This code performs a customized sparse matrix vector multiplication for a particular sparse matrix. This sparse matrix is derived from coupled five-point stencil patterns on separate two-dimensional structured grids, derived from a linear solver arising in variational optical flow methods [85]. The structure of this matrix is shown in Figure 6.5.

Bringing this into normalized form, we see the code shown in Figure 4.15.

The body of this function contains 32 flat data parallel operations. Data parallel compilers that do not perform fusion, such as NESL [8], would simply emit 32 data parallel operations for this code, even though each data parallel operation requires a synchronization and con-

```
@cu
def of_spmv(du, dv, width, m1, m2, m3, m4, m5, m6, m7):
        _e0 = map(op_mul, m1, du)
        _e1 = map(op_mul, m2, dv)
        _e_1 = map(op_add, _e0, _e1)
        _e2 = map(op_mul, m2, du)
        _e3 = map(op_mul, m3, dv)
        _f_2 = map(op_add, _e2, _e3)
        _e4 = op_neg(width)
        _e5 = shift(du, _e4, 0.0)
        _e6 = map(op_mul, m4, _e5)
        _e_3 = map(op_add, _e_1, _e6)
        _e7 = op_neg(width)
        _e8 = shift(dv, _e7, 0.0)
        _e9 = map(op_mul, m4, _e8)
        _f_4 = map(op_add, _f_2, _e9)
        _e10 = shift(du, -1, 0.0)
        _e11 = map(op_mul, m5, _e10)
        _e_5 = map(op_add, _e_3, _e11)
        _e12 = shift(dv, -1, 0.0)
        _e13 = map(op_mul, m5, _e12)
        _f_6 = map(op_add, _f_4, _e13)
        _e14 = shift(du, 1, 0.0)
        _e15 = map(op_mul, m6, _e14)
        _e_7 = map(op_add, _e_5, _e15)
        _e16 = shift(dv, 1, 0.0)
        _e17 = map(op_mul, m6, _e16)
        _f_8 = map(op_add, _f_6, _e17)
        _e18 = shift(du, width, 0.0)
        _e19 = map(op_mul, m7, _e18)
        _e_9 = map(op_add, _e_7, _e19)
        _e20 = shift(dv, width, 0.0)
        _e21 = map(op_mul, m7, _e20)
        _f_10 = map(op_add, _f_8, _e21)
        return (_e_9, _f_10)
```

Figure 4.15: Fusion Example Code after Normalization

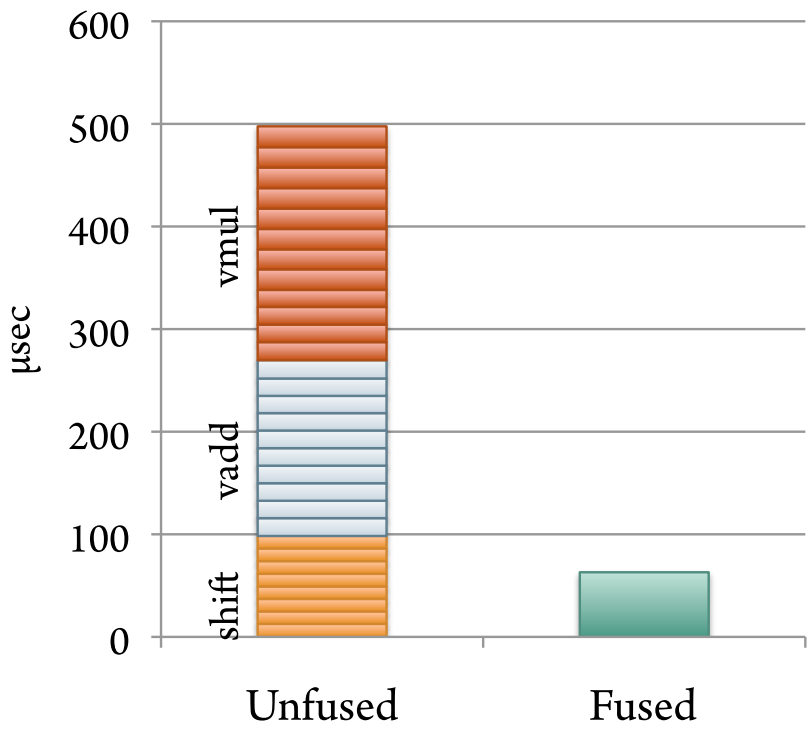



Figure 4.16: The performance impact of data parallel primitive fusion

siderable data movement to load operands and store results. Many of the operands would be loaded multiple times, even though for moderately sized problems, any on-chip cache would be overwhelmed and fail due to capacity misses. Examining the code, we can see that none of the synchronizations between data parallel primitives need to be performed, and no data movement needs to occur, other than loading the operands once and storing the results, because all the intermediate results are created and used only locally. If these operations could be fused together, we could save considerable synchronization and data movement costs.

To quantify these costs, we evaluate this computation on a problem $y = Ax$ for an $x$ of approximately 300k entries, running on an NVIDIA GeForce GTX480 GPU. The naive schedule executes 32 separate data parallel operations, while the fused schedule executes only one. Figure 4.16 illustrates these two implementations; we see a 7.9× performance differential between them. As we have explained, this performance differential arises because the fused implementation is able to avoid repeatedly loading and storing intermediate results from offchip DRAM.

In order to perform fusion, the compiler must prove that data is accessed in a way that is local to the implementation of the parallel primitive. Since our parallel primitives are mapped onto a virtual hierarchical processor, the compiler must prove that data is local to the level of the virtual processor to which the enclosing procedure is being mapped. We call a set of fused data parallel primitives a *phase*. We define a *phase analysis* procedure, which locates all syn-

```
@cu
def subsum(A):
    return map(sum, A)

» print subsum([[1, 2, 3], [4, 5], [6, 7, 8]])
» [6, 9, 21]
```

Figure 4.17: A nested sum operation

chronization points due to non-local data access. We then perform *phase scheduling* to reduce the number of synchronization points by fusing primitives together. Fusion takes on different forms in the resulting code, depending on whether the enclosing procedure is being targeted at the sequential level, the SIMD vector level, the SIMD thread level, or distributed across the cores of the machine. However, phase analysis and phase scheduling operate identically, regardless of which level of the machine the procedure is being targeted at. Our phase analysis and phase scheduling procedures are much simpler than traditional analyses for loop fusion, due to the constrained language we support.

Our methodology for scheduling nested data parallel primitives also influences the design of phase analysis and scheduling. We discuss this in the next section.

## 4.5 The Flattening Transform

One approach to scheduling nested parallel programs, adopted by NESL [10] and Data Parallel Haskell [21] among others, is to apply a flattening (or vectorization) transform to the program. This transform converts a nested structure of vector operations into a sequence of flat operations. In most cases, the process of flattening replaces the nested operations with segmented equivalents.

Figure 4.17 shows code for a nested sum operation, where `sum` is mapped across sequences in a nested sequence. When using the flattening transform, this program would be flattened into a program using a segmented reduction.

Figure 4.18 illustrates how a parallel segmented reduction operation works. A parallel segmented reduction can be performed by a segmented scan operation, followed by a compaction. The segmented scan proceeds similarly to a parallel non-segmented scan, with the difference that the binary operator provided to the reduction respects subsequence boundaries, and will pass the original input through unchanged if it is requested to operate on data from a different subsequence. Figure 4.18 illustrates a parallel segmented reduction. The original nested sequence is flattened into a single data sequence, where all subsequences are concatenated. Subsequence boundaries are recorded in an auxiliary data structure. The figure shows how the segmented scan proceeds. Dashed blue arrows indicate data that an ordinary scan tree would access that are not accessed in a segmented scan due to subsequence boundaries. After the segmented scan is performed, the first element of each subsequence is compacted into

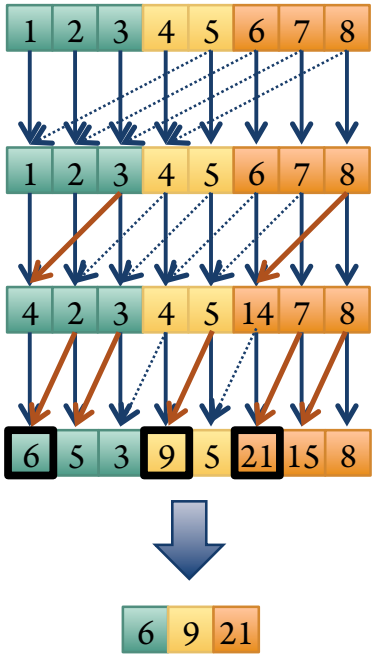



Figure 4.18: Executing a Segmented Reduction

the result of the segmented reduction. Although high performance segmented reductions are not implemented directly with this algorithm, the concept remains the same: a segmented reduction is a data parallel primitive operating on a flattened sequence, where the operation must take into consideration subsequence boundaries. The extra conditionals and data access needed to perform segmented operations impose significant overhead.

In contrast, consider performing the same operation using a direct mapping of the computation onto the parallel hierarchy. The outermost `map` would be distributed across processors of the machine. The inner `sum` would be performed by each processor independently, with each processor being assigned a single subsequence. In such a computation, the subsequence partitioning is accessed only once per core, and then the summation can proceed without further access to the subsequence partitioning. Additionally, no compaction step is necessary, because each processor generates a unique result for each subsequence. Accordingly, the direct approach may be more efficient in many circumstances.

The flattening transform is a powerful technique with two advantages:

1. The performance of flattened operations is not sensitive to the distribution of the overall work amongst the subproblems. In other words, the flattening transform creates perfect load balancing.

2. The flattening transform maps well to SIMD processors. Indeed, it was developed for completely flat SIMD arrays such as the CM-2. Performance is consistent, even when subproblem sizes don't match well with hardware vector sizes.

However, in many common cases, the performance afforded by the flattening transform is

significantly slower than the same computation implemented with a direct approach. Flattening transformations are best suited to machines that are truly flat. Most modern machines, in contrast, are organized hierarchically, as we have discussed previously. The central goal of the Copperhead compiler is thus to map the nested structure of the program onto the hierarchical structure of the machine, shown in Chapter 2.

Experience with hand-written CUDA programs suggests that direct mapping of nested constructions onto this physical machine hierarchy often yields better performance. For instance, Bell and Garland [5] explored several strategies for implementing SpMV in CUDA. Their Coordinate (COO) kernel is implemented by manually applying the flattening transformation to the `spmv_csr` algorithm. Their other kernels represent static mappings of nested algorithms onto the hardware. The flattened COO kernel only delivers the highest performance in the exceptional case where the distribution of row lengths is extraordinarily variable, in which case the load balancing provided by the flattening transform is advantageous. However, for 13 out of the 14 unstructured matrices they examine, applying the flattening transform results in performance two to four times slower than the equivalent nested implementation.

Experiences such as this lead us to the conclusion that although the flattening transform can provide high performance in exceptional cases where the workload is extremely imbalanced, the decision to apply the transform should be under programmer control, given the substantial overhead the flattening transform imposes for most workloads. We explore this further in the next section.

## 4.6 Quantifying the Flattening Transform

Since the flattening transform is traditionally used in data parallel compilers, it is important for us to justify our choice to forgo default application of the flattening transform.

In this subsection, we explore the performance consequences of employing the flattening transform, in order to quantify its impact. We expect the flattening transform will be essential for problems with a large load imbalance, but we wish to discover at what load imbalance level the flattening transform is required, compared with a direct nested parallelism mapping strategy.

To investigate this question, we compute $f(\mathbf{x_i}, \mathbf{y_i}) = ||\mathbf{x_i} - \mathbf{y_i}||^2$ for many pairs of vectors taken from $X = \{\mathbf{x_0}, \mathbf{x_1}, \ldots, \mathbf{x_n}\}$ and $Y = \{\mathbf{y_0}, \mathbf{y_1}, \ldots, \mathbf{y_n}\}$ simultaneously. The Copperhead code for this operation is in Figure 4.19:

Since the computational load for computing the vector norm varies directly with the length of the vector, this simple problem allows us to investigate the effects of load imbalance on different scheduling models.

This computation can be scheduled in several ways. Traditional data parallel compilers apply the vectorization transform. Given a notional `seg_sum(X)` segmented sum primitive that computes the segmented sum for each sub-sequence in `X`, as well as a `seg_map(f, X)` primitive that applies `f` to all elements of all sub-sequences of `X`, returning a nested sequence with the same structure as `X`, the result of the flattening transform for this computation is given in Figure 4.20:

```
@cu
def el(xij, yij):
    diff = xij - yij
    return diff * diff

@cu
def norm_diff2(xi, yi):
    el_wise = map(el, xi, yi)
    return sum(el_wise)

@cu
def multi_norm(X, Y):
    return map(norm_diff2, X, Y)
```

Figure 4.19: Copperhead code for `multi_norm`

```
@cu
def el(xij, yij):
    diff = xij - yij
    return diff * diff

@cu
def multi_norm_flattened(X, Y):
    el_wise = seg_map(el, X, Y)
    return seg_sum(el_wise)
```

Figure 4.20: Copperhead-like code for the flattened version of `multi_norm`

As you can see in Figure 4.20, the nested operations have been *lifted* into segmented operations, which is how the transform works.

In contrast, the Copperhead compiler maps the code given by Figure 4.19 directly onto the hardware. Nested operations are mapped independently onto processing elements. In the most general case, the vectors $\mathbf{x_i}$ and $\mathbf{y_i}$ are all of different lengths. For this case, we use flattened data structures that can represent arbitrarily nested sequences, and are described in Section 5.3.1. Although we use flattened data structures, the computations are mapped directly onto the machine. There are several ways in which the computations can be mapped directly onto the machine, which we will discuss in the following sections.

One important way computations can be mapped directly onto the machine is to schedule subcomputations as sequential loops running on SIMD lanes. When each of the vectors $\mathbf{x_i}$ and $\mathbf{y_i}$ have identical lengths, this approach can achieve maximum performance by representing X and Y as *uniformly nested sequences*, which are discussed in detail in Chapter 5. Importantly, uniformly nested sequences are well defined under transposition, which allows us to store X and Y in the order that best suits execution on the hardware. For singly nested sequences, this corresponds to representing a matrix X in row-major or column major ordering. We can then map each subcomputation directly into a sequential loop running on a SIMD lane, while still getting fully vectorized memory accesses, which leads to maximum performance. The ability to map computation directly onto the hardware, using data structures that match the mapping of the computation, has important performance implications, as we will see in the following subsections. We will also delineate the regime in which the flattening transform is useful.

### 4.6.1 Load Balancing

In this experiment, the length of the vectors was allowed to vary: all vectors were at least some minimum length $m$, which was then extended randomly by sampling from a Zipf power law distribution with parameter $s$. For reference, Equation 4.1 gives the probability mass function for this distribution, where $\zeta(s)$ is the Riemann Zeta function.

$$P(k;s) = \frac{k^{-s}}{\zeta(s)}, k > 0, s > 1 \tag{4.1}$$

Distributing vector lengths according to a Zipf distribution creates very difficult load balance problems that are often found in real datasets. For example, many important graphs, such as social networks and the internet, have edge distributions that follow power laws. In our case, using the power law distribution means that most of the vectors will be the same length, but there will be a few very large outlier vectors. We have illustrated a few selected datasets used in our experiment in Table 4.1.

The parameter $s$ describes the Zipf distribution used to generate the partitionings. When $s$, is high, all partitionings are of uniform size. As the parameter $s$ used to generate the partitionings decreases, the size of the partitionings become more irregular. The mean length of the partitionings increases, even though the median length of the partitionings remains almost constant. The minimum length $m$ of each partition was 512 in these experiments, in order to isolate the effects of load imbalance from those of SIMD vector mismatch.

| $s$ | min | max | mean | median | $\frac{\max}{\min}$ ($\psi$) | Flattened SPGFLOP/s | Fused Flattened SPGFLOP/s | Direct SPGFLOP/s | Uniform (size) SPGFLOP/s |
|---|---|---|---|---|---|---|---|---|---|
| 4 | 512 | 547 | 512.1 | 512 | 1.07 | 10.3 | 21.2 | 46.5 | 52.3 (512) |
| 1.86 | 512 | $6.12 * 10^5$ | 541.8 | 512 | 1195 | 10.3 | 21.3 | 29.9 | 52.5 (541) |
| 1.62 | 512 | $2.99 * 10^7$ | 1451 | 513 | 58449 | 10.4 | 22.3 | 3.79 | 53.5 (1451) |

Table 4.1: Selected Partitionings and Performance

We define the load imbalance factor as follows:

$$\psi = \frac{W_{max}}{W_{min}} \tag{4.2}$$

$W_{max}$ and $W_{min}$ are the maximum and minimum amount of work performed in a single subproblem. We expect that as $\psi$ increases, the direct approaches will become progressively less efficient, since they are exposed to load imbalance. In contrast, the performance of the flattened approach should be invariant to the load imbalance factor.

Figure 4.21 details the observed performance as a function of this load imbalance factor, running on an NVIDIA GTX 480 GPU. The "Flattened" data points represent performance for the flattened implementation, in Single Precision GFLOP/s. As we expected, performance is completely invariant to load imbalance, since the partitioning is not expressed in control flow, but instead in data structure.

The "Direct" data points represent performance for the direct mapped version, where each summation is mapped onto a CUDA thread block. The summations were mapped to CUDA thread blocks instead of CUDA threads because of the memory access patterns involved. As we discuss later, this is a problem for autotuning.

Looking at performance of the "Direct" data points, we see that for problems with small to moderate load imbalance, the direct approach is approximately 2.2× faster than the fused flattened approach, or 4.5× faster than the simple flattened approach. This holds until a load imbalance factor of approximately 1000, or in other words, until the ratio between the longest vector and the shortest vector is approximately 1000. With load imbalance factors between 1000 and 10000, the direct approach is still faster than the flattened version. Only after the load imbalance factor exceeds 10000 is the flattened version profitable.

The "Uniform" data points represent performance for a direct mapped version where all the sub-sequences are the same length, or equivalently, $\psi = 1$. In this case, the summations are mapped to sequential threadsPerformance of the uniform approach is approximately 2.5× faster than the fused flattened approach, and approximately 5× faster than the simple approach. Copperhead enables the use of aligned, transposed data structures, which in conjunction with the direct mapping strategy enables high performance in comparison to the use of the flattening transform by default. Without this mapping strategy, the performance we would attain would not be competitive with hand-written efficiency code.

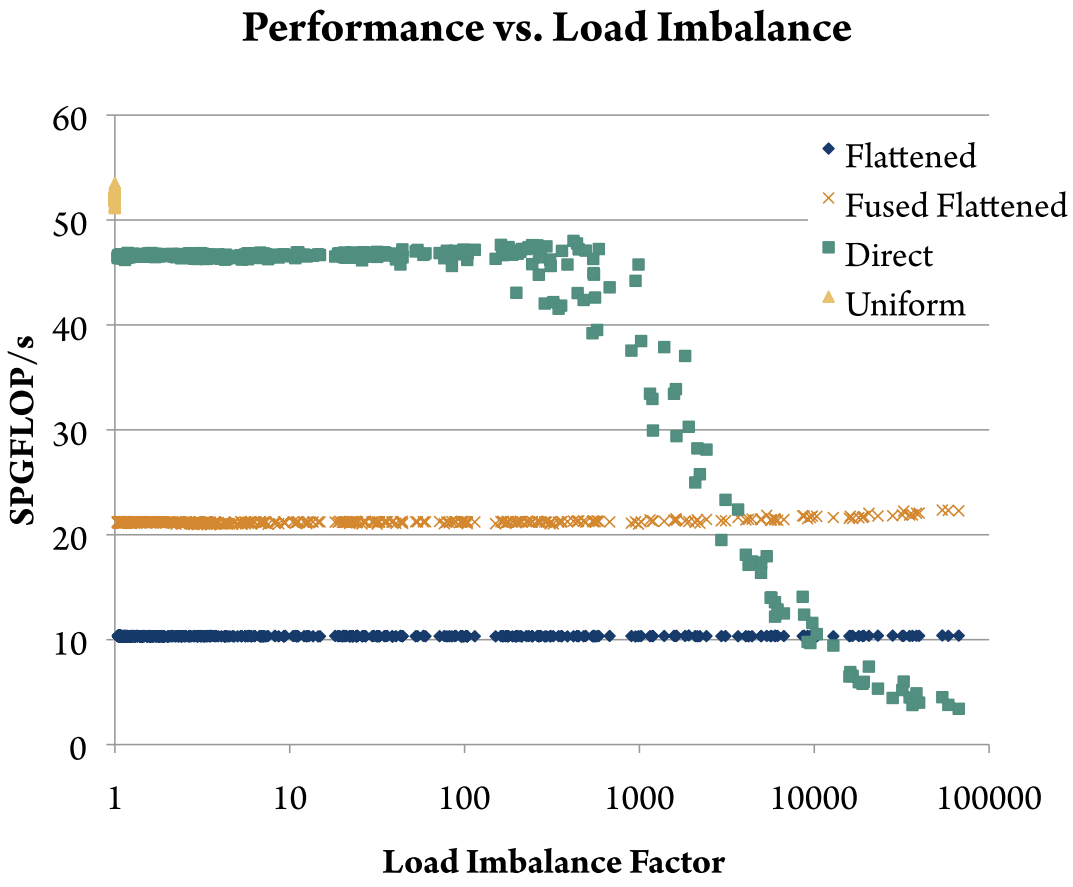



Figure 4.21: Performance comparison of flattened versus direct mapped scheduling

In summary, we find direct approaches to be faster than the flattening transform for problems with small to moderate load imbalance. For problems with extreme load imbalance, the flattening transform is compelling. However, it imposes high overhead for problems with moderate load imbalance.

### 4.6.2 SIMD Effects

As mentioned in Section 4.5, the flattening transform has two major advantages. The previous discussion centered on the the effects of the flattening transform in the presence of varying degrees of load imbalance, with minimum vector lengths long enough to avoid inefficiencies mapping to SIMD vectors. This section examines the effects of the flattening transform in the context of SIMD processing. The experiments were again run on an NVIDIA GTX 480 GPU, which has a native logical SIMD vector length of 32 elements. We ran similar experiments to those in the previous section, only focusing on the impact varying the minimum subvector length $m$ has on performance. Instead of drawing vector lengths from a Zipf distribution, we let all subvectors be some length $m$, except for a single subvector, randomly positioned in the set of subvectors, with length $\psi m$. This configuration generates the worst case load imbalance for a given value of $\psi$, since all subvectors are the same length except for one. We examined six different mapping strategies.

1. Flattened. The simple flattened approach applies the flattening transform without parallel primitive fusion.

2. Fused Flattened. The fused flattened approach applies the flattening transform with parallel primitive fusion

3. Direct Block. This direct approach maps nested parallel primitives to a small group of SIMD vectors. This approach corresponds to the direct approach used in the previous section.

4. Direct Warp. This direct approach maps nested parallel primitives to SIMD vectors. The current Copperhead compiler does not consider this level of the parallelism hierarchy when mapping, but we include results for this approach for completeness.

5. Direct Thread. This direct approach maps nested parallel primitives to sequential loops, running in parallel.

6. Uniform. This direct approach, valid only when $\psi = 1$, maps nested parallel primitives to sequential loops, running on a transposed uniform data structure. In other words, computations are mapped to the same parallelism hierarchy as the Direct Thread approach, but they operate on a transposed data structure, which greatly improves performance.

When considering how these various strategies will perform, we note the following. The flattened mapping approaches should perform equivalently regardless of SIMD width or subvector length, and we expect the fused flattened approach to perform better than the flattened

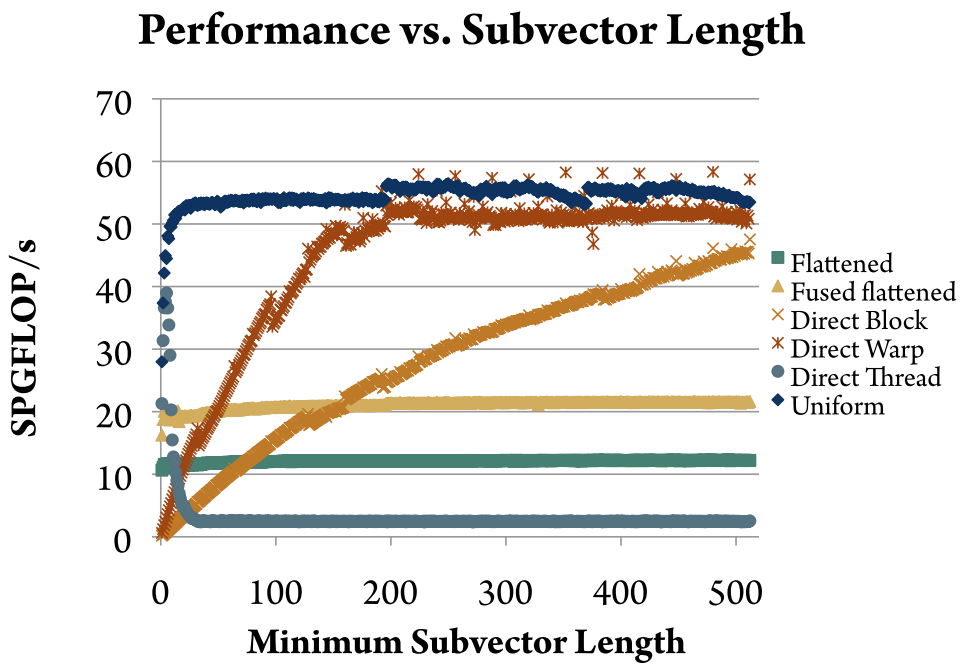



Figure 4.22: Performance comparison of nested parallelism mapping strategies, $\psi = 1$

approach, since it saves synchronization and bandwidth compared to the simple flattened approach. The Uniform direct approach should perform the best, and is in fact what most programmers would implement for a uniform computation, when it is known *a priori* that there is no load imbalance. The Direct Thread approach should perform the best for very short subvectors: for longer subvectors its performance will suffer due to inefficient use of memory bandwidth, since each sequential loop will be accessing progressively less contiguous regions of memory as subvector lengths increase, when implemented on a SIMD machine. This degrades performance, since the memory controller will be pulling in complete cache lines, but the program will only be using a fraction of the data in each cache line, effectively wasting memory bandwidth. Since many programs are memory bandwidth limited, this will be a major performance limitation. For short subvectors, this effect will not be important, and performance will approach the uniform direct approach, but for longer subvectors, performance will be poor. The direct warp and direct block approaches will perform well for longer subvectors, but performance will suffer as subvector length shrinks. The direct warp approach should perform better than the direct block approach for shorter subvectors, since the warp is a single SIMD vector, as opposed to a collection of multiple SIMD vectors, and therefore its natural vector length is smaller.

Figure 4.22 shows the performance as a function of subvector length for these various mapping strategies, for $\psi = 1$. As expected, the direct uniform approach is fastest, regardless of SIMD width. The various direct mapping approaches make sense for $m > 50$ or so, which makes sense given that the natural SIMD vector length of this machine is 32. The direct thread

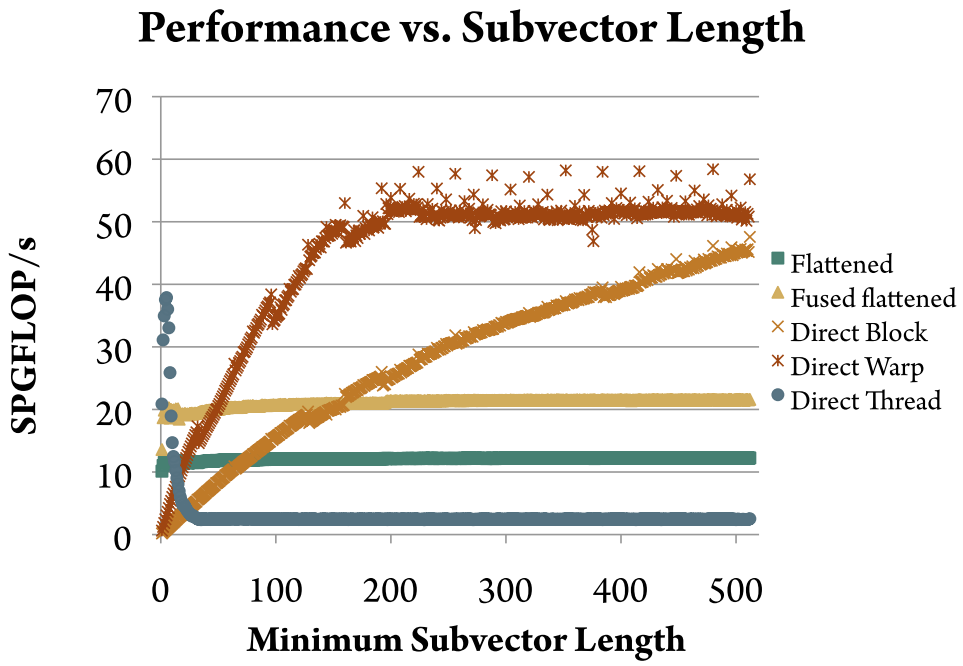



Figure 4.23: Performance comparison of nested parallelism mapping strategies, $\psi = 10$

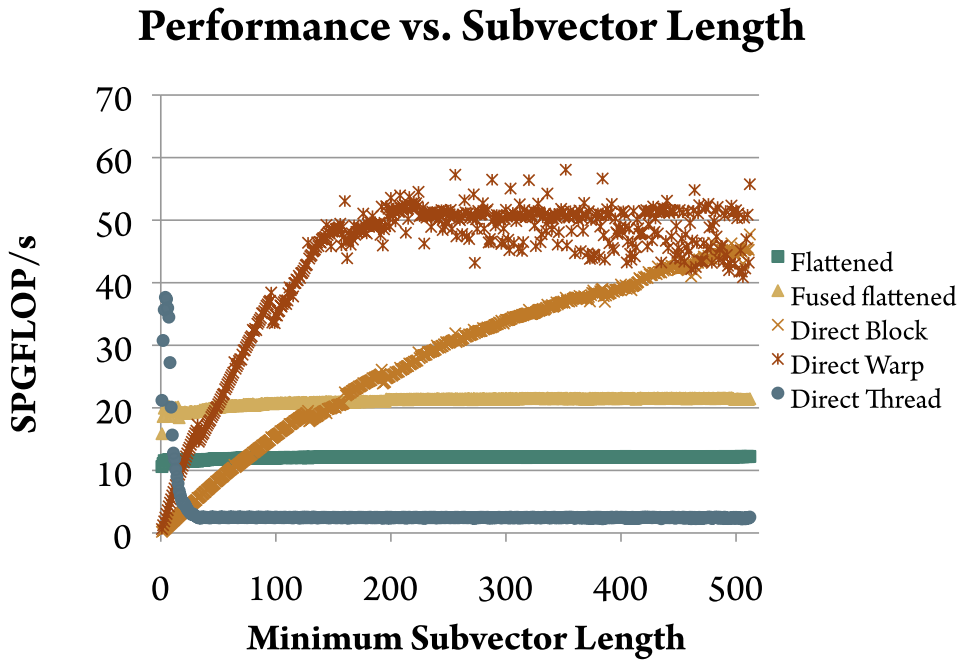



Figure 4.24: Performance comparison of nested parallelism mapping strategies, $\psi = 100$

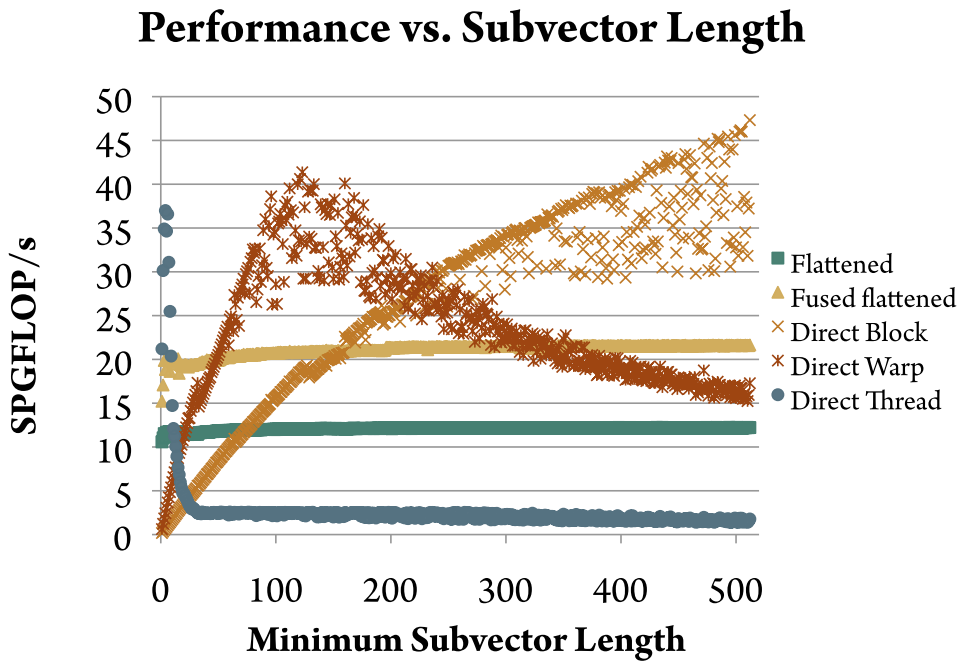



Figure 4.25: Performance comparison of nested parallelism mapping strategies, $\psi = 1000$

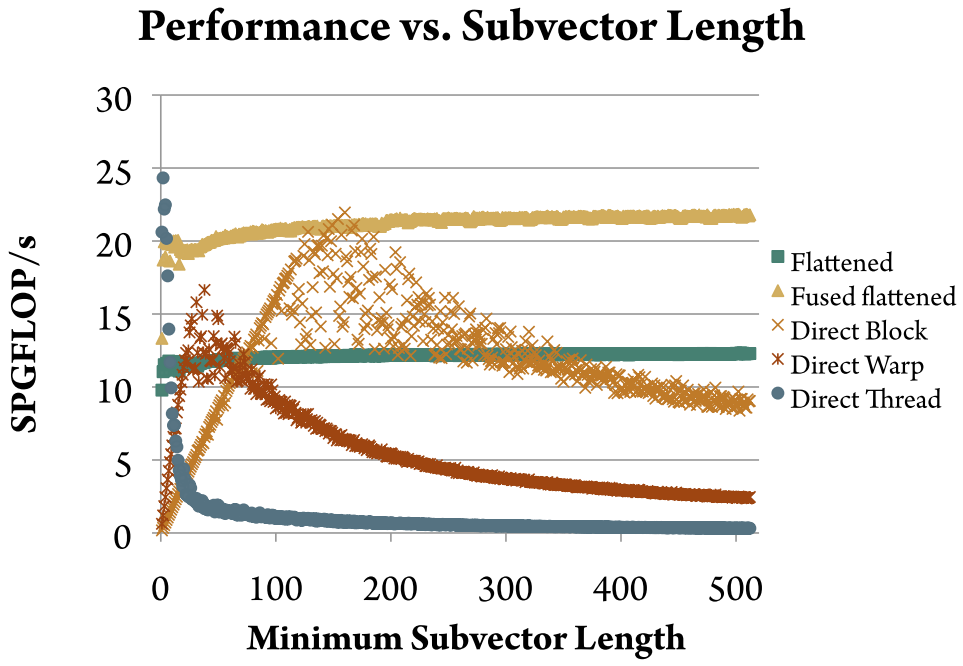



Figure 4.26: Performance comparison of nested parallelism mapping strategies, $\psi = 10000$

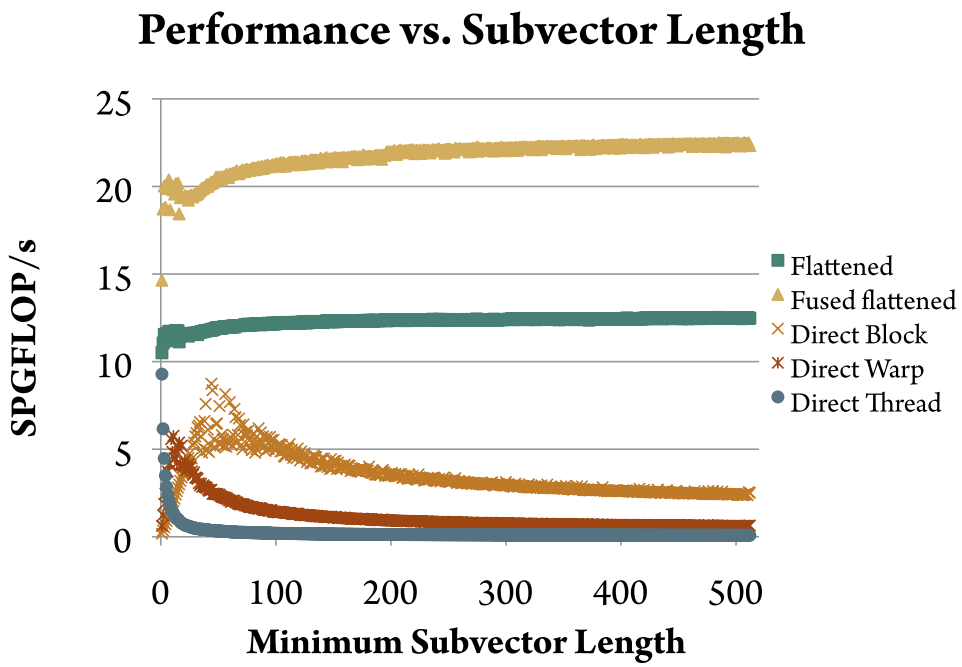



Figure 4.27: Performance comparison of nested parallelism mapping strategies, $\psi = 100000$

approach performs well only when $m < 10$. In this figure, $\psi = 1$, which enables the uniform direct approach. This important special case is only available because we use a direct mapping strategy as opposed to the flattening transform. In order to achieve performance comparable to handwritten efficiency code, the direct mapping strategy is essential.

Figure 4.23 shows the same curves for $\psi = 10$. The uniform direct approach is not longer applicable, but the direct mapping approaches are best for all subvector lengths except a narrow range, where $10 \leq m < 50$. The same observation holds true for Figure 4.24, where the load imbalance factor $\psi = 100$.

Figure 4.25 examines these strategies for $\psi = 1000$. At this more significant load imbalance factor, the observations made earlier do not hold. In particular, the direct warp strategy becomes less attractive compared to the direct block strategy. The direct warp strategy is less efficient because the hardware scheduler on the GTX 480 schedules groups of SIMD vectors, not SIMD vectors themselves. In the presence of significant load imbalance, mapping strategies that target the SIMD vector directly incur extra load imbalance overhead that cannot be compensated for by the hardware work distributor. Accordingly, the outlier subvector tends to dominate, leading to poor performance. At load imbalance factors of $\psi = 10000$ and $\psi = 100000$, this phenomenon intensifies, as shown in Figure 4.26 and Figure 4.27. Once the load imbalance factor becomes extreme, with $\psi \geq 1000$, none of the direct mapping approaches perform well, regardless of mapping strategy.

Summarizing, we find that even when considering SIMD effects, for problems with moderate load imbalance, direct approaches are usually better than flattened approaches. Flattening

transforms are most useful for problems where subproblem size does not map well onto the SIMD vectors of the target platform, or for where load imbalance is extremely high.

### 4.6.3 Summary

These results quantify our intuition expressed earlier. The combination of a hardware work distributor for dynamic load balancing, as well as a programming model that encourages oversubscription of the machine with very small tasks, which are then scheduled dynamically to work around load balancing problems, does much to mitigate load imbalance problems.

These results may be unexpected for programmers used to targeting CPUs without hardware work distributors. If the programming and execution model can't efficiently support the creation and execution of thousands of small tasks, load imbalances are more prominent [58]. On the other hand, we show that architectures that exploit fine grained parallelism and efficient work distribution can handle significant load imbalances without severe performance degradation.

Summarizing, we choose not to employ the flattening transform for two reasons: first, it is not necessary for many computations, given hardware load balancing, and second, it costs significant performance compared with the direct mapping strategy. The flattening transform is still useful: there are real-world problems with extreme load imbalance, or subproblems that don't map well to SIMD execution units. These problems require the flattening transform in order to perform efficiently. However, its usefulness on hardware platforms like GPUs is more limited than in past data parallel compiler efforts, and we believe it should be employed under programmer direction, rather than being used indiscriminately to implement all forms of nested parallelism. Direct mapping approaches are required in order to achieve performance competitive with hand-written efficiency code.

## 4.7 Scheduling Methodology

Our compiler thus performs a static mapping of nested programs onto a parallelism hierarchy supported by the target parallel platform.

Returning to our `spmv_csr` example being compiled to the CUDA platform, the `map` within its body will become a CUDA kernel call. The compiler can then choose to map the operations within `spvv` to either individual threads or individual blocks. Which mapping is preferred is in general program and data dependent; matrices with very short rows are generally best mapped to threads while those with longer rows are better mapped to blocks. Because this information is not in general known until run time, we currently present this decision as an option that can be controlled when invoking the compiler. The programmer can arrange the code to specify explicitly which mapping strategy is preferred, an autotuning framework can explore which is better on a particular machine, or a reasonable default can be chosen based on static knowledge (if any) about the problem being solved. In the absence of any stated preference, the default is to map nested operations to sequential implementations.

## 4.8 Phase Analysis and Scheduling

In this section, we define the phase analysis and scheduling procedures used to discover opportunities for fusion and mapping of a computation to the parallel hierarchy provided by a target platform.

### 4.8.1 Phase Analysis

Phase analysis discovers where synchronization points are required in the program due to data access pattern. Portions of the computation that can be performed without synchronization are then candidates for fusion. To perform phase analysis, we view procedures in the program as data dependence graphs, and derive the set of edges in the graph that require synchronization.

Let $\mathcal{O}$ be the set of data parallel operations supported by the language, such as `map`, `reduce`, `scatter`, and so forth, such as those defined in Chapter 3.

Let $\mathcal{P}$ be the set of levels in the parallel hierarchy described by the machine. In this work, we consider the virtualized parallel hierarchy presented by the efficiency programming environments OpenCL [84] and CUDA [67], which is shown in Table 4.2. This hierarchy is closely related to the hardware parallelism hierarchy shown in Figure 2.4, with two important differences. Firstly, it is virtualized, meaning that the number of computations expressed at each level of the virtualized parallel hierarchy need not correspond to the physical parallelism available on the parallel platform. When more parallelism is expressed than the hardware can accomodate, the hardware and efficiency runtime are responsible for looping over the parallelism expressed in the program, using the resources of the processor. Secondly, although the hardware has four levels (Chip, Core, SIMD Vector, SIMD Lane), the virtualized hierarchy has only three levels (Distributed, Block, and Sequential). The *Distributed* level corresponds to distributing computation across the cores of the chip. The *Block* level corresponds to execution via teams of threads that can synchronize and communicate arbitrarily, which corresponds to a single core of the chip. The *Sequential* level corresponds to execution by a single SIMD lane on the chip. The SIMD Vector level, which is part of the physical parallelism hierarchy, is not directly exposed by these efficiency programming environments. Although programmers can target it, as we have in Section 4.6.2, targeting SIMD vectors directly using these programming models leads to brittle and poorly supported code, and is conceptually an optimization due to hardware limitations that prevent efficient execution with minimal sized computations at the *Block* level. Accordingly, our compiler does not target a notional level of the parallelism hierarchy that would correspond to SIMD Vectors. Still, we find that targeting the parallelism hierarchy provided by CUDA and OpenCL allows us to achieve high efficiency.

Other parallel hierarchies are possible, for example, a clustered architecture with multiple sockets might introduce higher levels in the hierarchy, which would describe how computations are mapped onto the cluster. We could also target SIMD vectors directly; although they are not part of the parallel hierarchy explicitly exposed in OpenCL or CUDA, they are exposed by the hardware, and can be exploited for better performance in some situations, as we illustrated in Section 4.6.2.

| Element of $\mathcal{P}$ | Description |
|---|---|
| Distributed | Computation is distributed across a chip |
| Block | Computation is performed in parallel by small teams of work-items or threads |
| Sequential | Computation is performed in a sequential loop inside a work-item or thread |

Table 4.2: The parallel hierarchy $\mathcal{P}$ provided by OpenCL and CUDA

The Copperhead compiler allows the programmer to specify a parallel hierarchy to be targeted, by selecting levels of the hierarchy that the computation should be mapped to. Currently, the compiler can target two parallel hierarchies. First, the default hierarchy is the {*Distributed, Sequential*} hierarchy, where the outermost level of data parallel operations is distributed across cores of the chip, and the inner levels are sequentialized and executed on the SIMD lanes of the processor. Second, the programmer can choose to compile to the {*Distributed, Block*} hierarchy, where the outermost level is again distributed across cores of the chip, and inner levels are executed cooperatively by teams of SIMD vectors.

Once the procedures have been transformed into normalized form, each procedure can then be represented as a directed acyclic graph that will be executed on a unique level of the parallelism hierarchy. We define the graph as a set of nodes and edges:

$$G = \langle \mathcal{N}, \mathcal{E} \rangle$$

- $\mathcal{N}$ is the set of data parallel operations in the data dependence graph. Each operation $n \in \mathcal{N}$ is described by a tuple $\langle n_o, n_h \rangle \in \mathcal{O} \times \mathcal{P}$. $n_o$ represents the kind of operation performed in $n$, and $n_h$ represents the level of the parallel hierarchy to which the procedure has been mapped. $n_h$ will be identical for all nodes in a given procedure, since the program is in canonical form.

- $\mathcal{E}$ represents the data dependences in the program. An edge from operation $n_i$ to operation $n_j$ represents a data dependence between operation $n_i$ and $n_j$. Because our procedures are in normalized form, each edge corresponds to a unique identifier in the procedure. Each dependence $e \in \mathcal{E}$ is labeled with a tuple $\langle e_b, e_c, e_d \rangle \in \{False, True\} \times \mathcal{C} \times \mathcal{D}$. $e_b$ represents the presence or absence of synchronization between the producer and consumer operations. Computing $e_b$ is the goal of phase analysis, or in other words, to discover the synchronization points in the data dependence graph. $e_c$ represents the *completion* of the identifier, as produced by operation $n_i$. $e_d$ represents the *directionality* of the identifier, as produced by operation $n_i$. We will define the completion and directionality spaces next.

The *completion space* is a totally ordered set $\mathcal{C}$, where each $c \in \mathcal{C}$ represents a completion state. We use a very simple completion space: $\mathcal{C} = \{None, Local, Global\}$, ordered as written, with the following meanings:

- *None*: The variable is not completed. For inputs to data parallel operations, this means that no completion is required in order for the operation to proceed. This is true, for example, for functional arguments, which are statically defined at code generation and compilation time, and therefore do not need to be computed. For outputs from data parallel operations, this means the result of the operation is not completed without a synchronization. For example, the result of a `reduce` requires a synchronization in order to be completed. Similarly, the result of a `scatter` requires a synchonization in order to be completed.

- *Local*: The variable is completed locally with respect to the iteration structure it is composed in. For inputs to data parallel operations, declaring the completion as *local* is a strong assertion: it means that the data parallel operation will not access any element of that input except the current element defined by the iteration structure of the level of the parallel hierarchy is composed in. For example, all explicit inputs to a `map` operation require *local* completion, which follows from the definition of `map`. For outputs from data parallel primitives, *local* completion means that the result is produced element by element with respect to the iteration structure it is composed in. For example, the result of a `map` operation is produced with *local* completion.

- *Global*: The variable is globally completed. For inputs to data parallel operations, *global* completion means that the operation needs random access to the input's data - or in other words, the data must be globally completed before the operation can be performed. For outputs from data parallel operations, *global* completion means the result can be utilized without any additional synchronization, regardless of the computation that may be utilizing the result.

In addition to defining a completion space, we also define a directionality space, which is the set $\mathcal{D} = \{None, Backward, Forward\}$, with the following meanings:

- *None*: No directionality. Variables that are produced and consumed in parallel contexts are labeled with this directionality, as are variables produced and consumed in sequential contexts that do not depend on the directionality of the underlying sequential context. For example, the inputs and outputs of `map` have *None* directionality.

- *Backward*: Backward directionality. This is used for variables that are produced or consumed from element $n-1$ to element 0 in a sequential context. For example, the input and output of a sequential `rscan` operation, which executes a reverse prefix-sum operation, both have *Backward* directionality.

- *Forward*: Forward directionality. This is used for variables that are produced or consumed from element 0 to element $n-1$ in a sequential context. For example, the input and output of a sequential `scan` operation both have *Forward* directionality.

To perform phase analysis, all inputs to the procedure are initialized as having *Global* completion and *None* directionality. We then traverse the graph in a breadth first manner. At each node $n$, we examine the completion and directionality requirements of each input to $n$, given

| Primitive | Input Completion | Output Completion | Input Directionality | Output Directionality |
|---|---|---|---|---|
| `map` | $(N, L, \ldots, L)$ | $L$ | $(N, \ldots, N)$ | $N$ |
| `reduce` | $(N, L, G)$ | $N$ | $(N, N, N)$ | $N$ |
| `scan` | $(N, L)$ | $N$ | $(N, N, N)$ | $N$ |
| `rscan` | $(N, L)$ | $N$ | $(N, N, N)$ | $N$ |
| `gather` | $(G, L)$, | $L$ | $(N, N)$ | $N$ |
| `scatter` | $(L, L, G)$ | $N$ | $(N, N, N)$ | $N$ |

Table 4.3: Completion and Directionality types for selected data parallel primitives at the Distributed Level

| Primitive | Input Completion | Output Completion | Input Directionality | Output Directionality |
|---|---|---|---|---|
| `map` | $(N, L, \ldots, L)$ | $L$ | $(N, \ldots, N)$ | $N$ |
| `reduce` | $(N, L, G)$ | $N$ | $(N, N, N)$ | $N$ |
| `scan` | $(N, L)$ | $N$ | $(N, F, N)$ | $F$ |
| `rscan` | $(N, L)$ | $N$ | $(N, B, N)$ | $B$ |
| `gather` | $(G, L)$, | $L$ | $(N, N)$ | $N$ |
| `scatter` | $(L, L, G)$ | $N$ | $(N, N, N)$ | $N$ |

Table 4.4: Completion and Directionality types for selected data parallel primitives at the Sequential Level

the level of the parallelism hierarchy that $n$ has been assigned to. These requirements are given by a lookup table described as a function of $n_o$ and $n_h$, which describes completion and directionality requirements for every input to node $n$, as well as the completion and directionality of the output of node $n$. For each input edge $e$ to $n$, we lookup $n_{e_c}$ and $n_{e_d}$, the completion and directionality requirements for the corresponding input to node $n$. If $e_c < n_{e_c}$, or $(n_{e_d} \neq None) \wedge (e_d \neq n_{e_d})$, then we mark $e_b = True$, recording that a synchronization is required. These two considerations guarantee that we are accessing all inputs to an operation in a way that is consistent with the iteration structure provided by the corresponding level of the parallelism hierarchy, which then potentially removes the need for synchronization.

Phase analysis is related to the construction of a fusion graph in classical loop fusion approaches[51]. A fusion graph is a graph where nodes represent loop nests, and edges represent dependences between loops. When constructing a fusion graph, loops are examined for compatible headers, which must have exactly the same number of iterations. Loops with incompatible headers are marked with fusion-preventing edges, which are similar to synchronization points in our formulation. Loops with no loop-carried dependences are marked as parallel, and loops with data dependences are marked as sequential. The goal of phase analysis similar, since it constructs graphs where nodes represent data parallel operations, and edges represent dependences, a subset of which are marked as requiring synchronization, which is similar to a fusion-preventing edge.

Phase analysis differs in several ways from constructing fusion graphs for loop nests in several ways. First, phase analysis operates on data parallel operators, instead of loop nests. We discover synchronization points, which are similar to fusion-preventing edges in a loop fusion graph, by checking whether the completion and directionality types of data parallel operators are compatible. Loop fusion performs a similar operation by enforcing that loop headers are compatible. Checking the compatibility of completion and directionality types is a looser constraint that takes advantage of the more constrained kinds of loops used to implement data parallel computations. This allows more operations to be fused than would be allowed in a classical loop fusion operation. For example, we are able to fuse `map` operations of different lengths together, and we are able to fuse a sequential `map` operation with a sequential `rscan` operation, both of which would have fusion preventing edges in a classical loop fusion graph. This improves performance. Additionally, classical loop fusion distinguishes only between sequential and parallel loops, and is not designed for fusing nested loops together. We allow data parallel operations to be mapped to many iteration structures, corresponding to the various levels provided by the parallelism hierarchy of a particular machine, and we fuse them at all levels of the hierarchy. This is important for performance.

### 4.8.2 Phase Scheduling

After data dependences are recorded, we traverse the graph by breadth first search, labeling each node $n$ with the number of synchronizations required to reach $n$. Nodes with identical labels are collected and placed in a *phase*, ordered according to the ordering present in the data dependence graph, which represents a computation that can be performed without any synchronization at the given level of the parallelism hierarchy. This performs data parallel primi-

```
@cu
def foo(x, y):
    a = map(bar, x)
    b = map(baz, y)
    c = permute(a, b)
    d = permute(x, y)
    e = map(c, d)
    return e
```

Figure 4.28: Simple flat data parallel example

tive fusion.

It is important to note that phase scheduling can be considered as an optimization problem, since fusion doesn't affect program semantics, once we have performed phase analysis. From the set of all nodes with identical labels, we can choose which to fuse, and which not to fuse. If there are $k$ nodes with the same level, an exhaustive exploration of potential legal fusions would require considering all $2^k$ choices. Examining the ramifications of performing fusion can be complex, since fusion affects the working set of a computation in non-trivial ways. Depending on how much commonality can be extracted from the fused set of computations by the efficiency compiler, it is difficult to know *a priori* what the performance implications of a fusing a particular choice of nodes would be. Since the cost of not fusing operations is so high, as we established in Section 4.4.1, we employ a maximalist heuristic that fuses as many nodes together as possible. For many programs, this appears to be the correct choice, as shown in Chapter 6 although future work should investigate this further.

### 4.8.3 Phase Analysis and Scheduling Example

As an example, consider the following simple example problem, shown in Figure 4.28, with its accompanying data dependence graph, shown in Figure 4.29.

After phase analysis, two synchronization points are found, and are marked in Figure 4.30, with bars across the edges where synchronization is required. Phase scheduling then discovers the four operations that can be performed without any synchronization, which are fused together in *phase 0*. For example, we discover that the computation of `a`, `b`, `c`, and `d` can all be performed in one fused primitive, without synchronization. One synchronization is required before we use `c` and `d` in the computation of `e`. The final `map` operation is placed in *phase 1*.

This procedure has drastically reduced the number of synchronization points: instead of having a synchronization after every data parallel operation, which would create five synchronization points, phase analysis and scheduling has reduced the number of synchronization points to two.

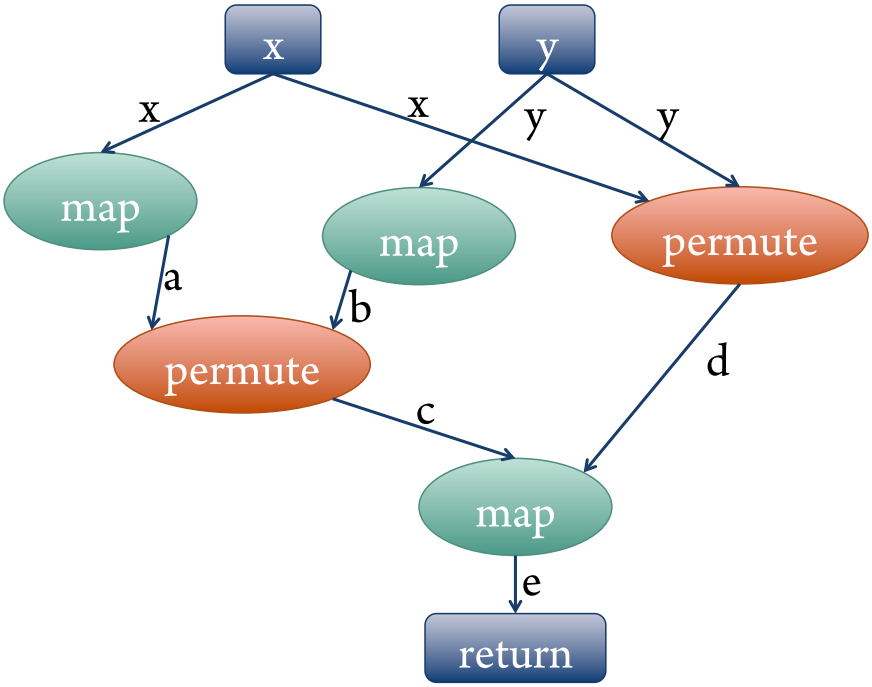



Figure 4.29: Data Dependence graph for code in figure 4.28

.

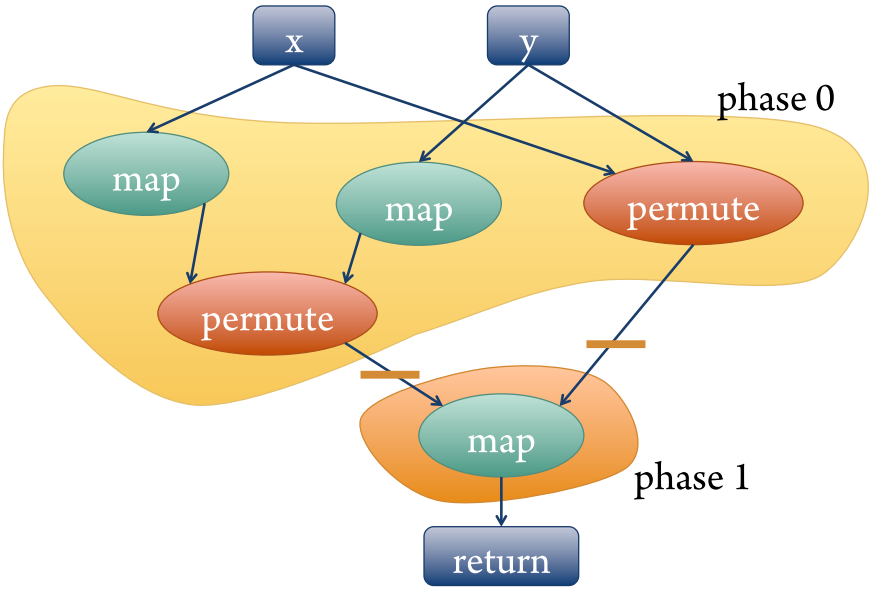



Figure 4.30: Analyzed and Scheduled data dependence graph for code in figure 4.28

.

```
@cu
def vadd(x, y):
    return map(op_add, x, y)

@cu
def foo(A, B):
    X = map(vadd, A)
    Y = map(vadd, B)
```

Figure 4.31: Unfused primitives

### 4.8.4 Limitations

This scheme finds opportunities for fusion within a single procedure. Because we have performed inlining during normalization, opportunities for fusion across procedures are often exposed in the data dependence graph and can be exploited by this procedure. However, our inlining is incomplete, and will not necessarily find all opportunities for fusion.

For example, Figure 4.31 shows a simple procedure where our phase analysis and scheduling would not find an opportunity for fusion. The two calls to `map` in `foo` would be fused together. Assume they have been mapped onto the distributed level of the parallelism hierarchy. Then their calls to `vadd` will be mapped to a lower level of the parallelism hierarchy, assume the sequential level. It turns out that both calls to `vadd` could be fused at the sequential level, thus creating only a single data parallel primitive at both the sequential level as well as the distributed level. However, our inlining procedure during normalization is not sufficient to expose both calls to `vadd` as operations that could be fused, and so the compiler will generate two sequential loops, even though the computation could be implemented with only one.

In general, to do this would require performing phase analysis and scheduling level by level, proceeding from the highest level of the parallel hierarchy to the lowest level. At each level, after performing phase analysis and scheduling, we would discover the primitives that will be fused together at that level. These primitives may themselves contain nested data parallel operations. All nested data parallel operations from primitives that have been fused together would then be aggregated into a new procedure, which would then be analyzed as we have outlined. This would expose opportunities for fusion across procedural boundaries throughout the parallelism hierarchy, unlike our current solution, which only finds opportunities for fusion across procedural boundaries at the outermost level of nested parallelism, due to the way our inlining works during normalization of the program. Currently, we do not perform this optimization, which is left to future work.

## 4.9 Using On-chip Memories

Highly parallel microprocessors stress the memory subsystem and are often bottlenecked by data transfer times. To mitigate data transfer times, they typically have hierarchical on-chip

```
@cu
def dot(x, y):
    return sum(map(op_mul, x, y))

@cu
def onchip(nested, reused):
    def foo(item):
        return dot(item, reused)
    return map(foo, nested)
```

Figure 4.32: Closed over data may be intensively reused

memory subsystems, composed of progressively smaller amounts of memory that can be accessed quickly at high bandwidth. Using on-chip memory effectively is often key to performance. In this section, we discuss how the Copperhead compiler identifies opportunities for using on-chip memory and then takes advantage of it.

On-chip memory is useful to communcate and synchronize independent threads. For example, consider a reduction tree, where independent threads are reducing an array of data down to a single element, by applying an associative reduction operator. By using on-chip memory to communicate partial reduction results between threads as they cooperate to execute the reduction tree, the computation avoids moving data on and off chip. This usage of on-chip memory is captured in Copperhead through the use of data parallel primitives that encapsulate these patterns. Concretely, primitives like `reduce`, `sum`, `scan`, and so on will use on-chip memory to coordinate their execution.

Another use case for on-chip memory is as a bandwidth multiplier for data that is intensively reused by many threads. The Copperhead compiler supports this case through closure analysis.

Consider the code in Figure 4.32. `onchip` computes many dot products, one for each subsequence in `nested`, and all against the variable `reused`. If `reused` is small enough to fit in an on-chip memory structure, it may be beneficial to do so, since all instantiations of `foo` are going to read the data in `reused`. In other words, `reused` is being broadcast to all instantiations of `nested`.

In Copperhead code, this broadcast/reuse pattern is easy to identify through a simple closure analysis. During normalization of the call to `onchip`, the compiler identifies that `reused` has been closed over and used in `foo`. The fact that `foo` is used in a closure that is used in a `map` operation means that the data closed over is being broadcast to all instantiations of `foo`. In fact, closing over data in this manner is the only mechanism by which Copperhead programs can broadcast data, since the semantics of `map` mean that every data argument to `map` is dereferenced and operated on independently. If the operation being performed in a `map` requires some data that is the same for all instantiations of `map`, the only way to effect this is to close over that data. Closing over data makes the broadcast obvious to the compiler, and therefore the compiler can examine data being closed over for placement in on-chip memory.

```
» x = [0, 1, 2, 3]
» y = [2.3, 4.7, 1.2, 3.4]
» soa = zip(x, y)

» print soa
» [(0, 2.3), (1, 4.7), (2, 1.2), (3, 3.4)]
» print soa.type
» Seq(Tuple(Int, Double))
```

Figure 4.33: Creating an Array of Structures

We use the result of shape analysis, outlined in Section 4.3, to discover the shape of all data that has been closed over. We unify general types derived from type inference with the concrete datatypes presented to the current instantiation of the procedure, in order to discover the size in bytes of elements in data that has been closed over. We then proceed to greedily pack closed over data into on-chip memory based on its size. Choosing which data should be placed in on-chip memory is done in a platform independent way in the mid end of the compiler, which has a simple model of how large the L1 cache or scratchpad of each core is on the target platform, and can then mark which variables should be transferred to L1 cache or scratchpad memory. The back end, where platform specific code generation takes place, takes care of actually transferring marked data into the on-chip memory structures.

Using on-chip memory can significantly improve performance. For example, a Copperhead program performing Support Vector Machine Training, detailed in Chapter 6, saw average performance improve from 37 GFLOP/s to 75 GFLOP/s, thereby doubling performance.

There are potentially other ways the Copperhead compiler could use on-chip memory, which remain future work. Still, the combination of on-chip memory usage during data parallel primitive evaluation, coupled with closure analysis to identify intensively reused data, provides the Copperhead compiler with some ability to use on-chip memory, which has important performance benefits.

## 4.10 Structures of Arrays

The Copperhead language allows the programmer to create arrays of structures using `zip`, which combines several arrays into a single array of structures. Figure 4.33 shows how to create an array of structures using `zip`. It is well known that for SIMD architectures, the structure of arrays data layout is usually more efficient, because adjacent SIMD lanes can load adjacent data elements in a structure of arrays layout, while in an array of structures layout, adjacent SIMD lanes load strided data elements.

Accordingly, the Copperhead compiler has the freedom to use the array of structures layout internally, even if the programmer has used `zip` to create a structure of arrays in their code. Choosing data layouts can be a complicated problem. In this case, we employ a simple heuris-

tic: internal, temporary variables created by the Copperhead compiler are always created in an structure of arrays format. If they are used as a structure of arrays, the Copperhead compiler translates loads from the data structure into multiple loads from the several component arrays in a structure of arrays format. For results that are returned to the user, the compiler must insert transposition code necessary to transform a structure of arrays into an array of structures. However, in many cases this is not necessary, since the array of structures is only used temporarily, in which case it need never to be constructed, and the call to `zip` performs no work. This is also important to efficiency.

## 4.11 Conclusion

In this chapter, we have discussed techniques for embedded data parallel compilation. We outlined how to normalize Copperhead programs into a form where analysis and compiler transformations are easily done. We discussed the need for data parallel primitive fusion, as well as the motivations for direct mapping strategies for data parallel primitives, in lieu of the flattening transform. We explained our shape analysis system, which derives shapes of various data elements in a program, which is necessary in order to preallocate space for results, which is essential for primitive fusion. We detailed our phase analysis and scheduling procedure, which restructures a computation into fused phases that proceed without synchronization, thus greatly increasing efficiency. The remaining task, then, to create a working programming environment, is to create a runtime that efficiently embeds this language and compiler into a productivity language. We discuss this task in the following chapter.

# 5 The Copperhead Runtime

In this chapter, we discuss how C++ code generated by the compiler is compiled and executed, as well as the data structures employed by the Copperhead runtime. These details are important for two reasons. First, the techniques we presented in the previous chapter are not sufficient to guarantee efficient code execution; details of data structures and code generation are also critical to efficiency. Second, a naive embedding of a dynamically generated efficiency layer code in a productivity language has the potential to overwhelm extra performance from parallel execution with extraneous compilation, data marshalling, and binding overheads. We show that with careful attention to implementation, binding dynamically generated efficiency layer code with a productivity language can be done efficiently. We do this through runtime static compilation, which allows us to amortize compilation overhead across many calls to the same procedure, despite the fact that we are compiling at runtime. Additionally, we explain how the Copperhead runtime is envisioned to operate on systems without access to compilers, which has special importance in mobile computing.

## 5.1 CUDA C++ Back End

Figure 4.1 describes the flow of the Copperhead compiler. Chapter 4 detailed the front end and mid end of the compiler, and in this section we discuss the back end of the Copperhead compiler, which lowers the mapped and scheduled AST generated by the mid end into platform specific code. The mapped and scheduled AST for our `spmv_csr` example is shown in Figure 5.1. The compiler has scheduled `spmv_csr` to the *Distributed* level of the parallelism hierarchy, and `spvv` to the *Sequential* level of the parallelism hierarchy. As previously mentioned, we currently have a single back end that generates code for the CUDA platform. Figure 5.2 shows an example of such code for our example `spmv_csr` procedure. It consists of a sequence of function objects for the nested `lambdas`, closures, and procedures used within that procedure. The templated function `spmv_csr_phase0` corresponds to the one and only *Distributed* phase of the computation, which is invoked by the C++ implementation of the Copperhead entry point procedure `spmv_csr`. The C++ interface will be called from Python, and ultimately will be responsible for converting Python data structures into C++ data structures and vice versa.

Not shown here is the host code that invokes the parallel kernel. It is the responsibility of the host code to marshal data where necessary, allocate any required temporary storage on the

```
def _lambda0(Aij, xj):
    return op_mul(Aij, xj)
def spvv(Ai, j, _K0):
    z0 = gather(_K0, j)
    tmp0 = map(_lambda0, Ai, z0)
    return sum(_e0)
def spmv_csr_phase0(x, A_columns, A_values):
    return map(closure([x], spvv), A_values, A_columns)
def spmv_csr(A_values, A_columns, x):
    return spmv_csr_phase0(x, A_values, A_columns)
```

Figure 5.1: Normalized, scheduled output of Mid-end compiler

GPU, and make the necessary CUDA API calls to launch the kernel. We discuss the host code in more detail in Section 5.2.

We generate templated CUDA C++ code that makes use of a set of templatized sequence types, such as `stored_sequence` and `nested_sequence` types, which hold sequences in memory. Fusion of sequential loops and block-wise primitives is performed through the construction of compound types. For example, in the `spvv` functor shown in Figure 5.2, the calls to `gather` and `transform` perform no work. Instead they construct `gathered_sequence` and `transformed_sequence` structures that lazily perform the appropriate computations upon dereferencing. Work is only performed by the last primitive in a set of fused sequential or block-wise primitives. In this example, the call to `seq::sum` introduces a sequential loop that then dereferences a compound sequence, at each element performing the appropriate computation. When compiling this code, the C++ compiler is able to statically eliminate all the indirection present in this code, yielding machine code that is as efficient as if we had generated the fused loops directly.

We generate fairly high level C++ code, rather than assembly level code, for two reasons. First, existing C++ compilers provide excellent support for translating well structured C++ to efficient machine code. Emitting C++ from our compiler enables our compiler to utilize the vast array of transformations that existing compilers already perform . More importantly, it means that the code generated by our Copperhead compiler can be reused in external C++ programs. Systems like Copperhead enable developers to prototype algorithms in a high-level language and then compile them into template libraries that can be used by a larger C++ application, which is an important usage scenario.

## 5.2 Runtime

Copperhead code is embedded in standard Python programs. Python function decorators indicate which procedures should be executed by the Copperhead runtime. When a Python program calls a Copperhead procedure, the Copperhead runtime intercepts the call, compiles

```
struct lambda0 {
    template<typename _a > __device__
    _a operator()(_a Aij, _a xj) { return Aij * xj; }
};

struct spvv {
    template<typename _a > __device__
    _a operator()(stored_sequence<_a> Ai,
                  stored_sequence<int> j,
                  stored_sequence<_a> _k0) {
        gathered<...> z0 = gather(_k0, j);
        transformed<...> tmp0 =
               transform<_a>(lambda0(), Ai, z0);
        return seq::sum(tmp0);
    }
};

template<typename T2>
struct spvv_closure1 {
    T2 k0;
    __device__ _spvv_closure1(T2 _k0) : k0(_k0) { }

    template<typename T0, typename T1> __device__
    _a operator()(T0 arg0, T1 arg1) {
        return spvv()(arg0, arg1, k0);
    }
};

template<typename _a > __device__
void spmv_csr_phase0(stored_sequence<_a> x,
                     nested_sequence<int,1> A_columns,
                     nested_sequence<_a, 1> A_values,
                     stored_sequence<_a> _return) {
    int i = threadIdx.x + blockIdx.x*blockDim.x;

    if( i < A_values.size() )
        _return[i] = spvv_closure1<_a>(x)(A_values[i],
                                          A_columns[i]);
}

extern "C" __global__ void spmv_csr_kernel0_int(...) {
    // (1) Wrap raw pointers from external code
    //     into sequence structures.
    ...

    // (2) Invoke the templated entry point
    spmv_csr_phase0(x, A_columns, A_values, _return);
}
```

Figure 5.2: Sample CUDA C++ code generated for `spmv_csr`. Ellipses (...) indicate incidental type and argument information elided for brevity.

the procedure, and then executes it on a specified execution place. The Copperhead runtime uses Python's introspection capabilities to gather all the source code pertaining to the procedure being compiled. This model is inspired by the ideas from Selective, Embedded, Just-In-Time Specialization [13].

The Copperhead compiler is fundamentally a static compiler that may be optionally invoked at runtime. Allowing the compiler to be invoked at runtime matches the no-compile mindset of typical productivity-level programmers. However, the compiler does not perform dynamic compilation optimizations specific to the runtime instantiation of the program, such as treating inputs as constants, tracing execution through conditionals, and so on. Forgoing these optimizations enables the results of the Copperhead compiler to be encapsulated as a standard, statically compiled binary, and cached for future reuse or incorporated as libraries into standalone programs that are not invoked through the Python interpreter.

Consequently, the runtime compilation overhead we incur is analogous to the build time of traditional static compilers, and does not present a performance limitation. The Copperhead compiler itself typically takes on the order of 300 milliseconds to compile our example programs, and the host C++ and CUDA compilers typically take on the order of 20 seconds to compile a program. Both of these overheads are not encountered in performance critical situations, since Copperhead's caches obviate the need for recompilation. The actual runtime overhead, compared with calling an equivalent C++ function from within C++ code, is on the order of 100 microseconds per Copperhead procedure invocation. We detail these overheads in Section 5.5.

Copperhead's CUDA runtime generates code to implement a computation in three separate parts:

1. A set of CUDA C++ kernels that represent the different parallel phases of the procedure, as well as a C++ function that invokes those kernels and allocates data for temporary variables.

2. A C++ wrapper that provides an interface to the computation suitable for calling from Python.

3. A Python driver procedure that is responsible for allocating data for results that will be visible to the host Python program, as well as calling the C++ implementation.

### 5.2.1 Runtime Static Compilation

The resulting C++ and CUDA code is compiled using standard C++ and CUDA compilers, linked together, and loaded into the Python interpreter. A naive implementation of such a system would create significant runtime overheads due to repeated compilation steps, as well as the bindings between C++ and Python. This would significantly diminish the utility of a programming environment such as Copperhead. However, these overheads can be mitigated through careful engineering, as we show.

Figure 5.3 shows a flowchart of the runtime compilation system we employ. A Copperhead function object is a Python `callable` object, created by decorating a Python function

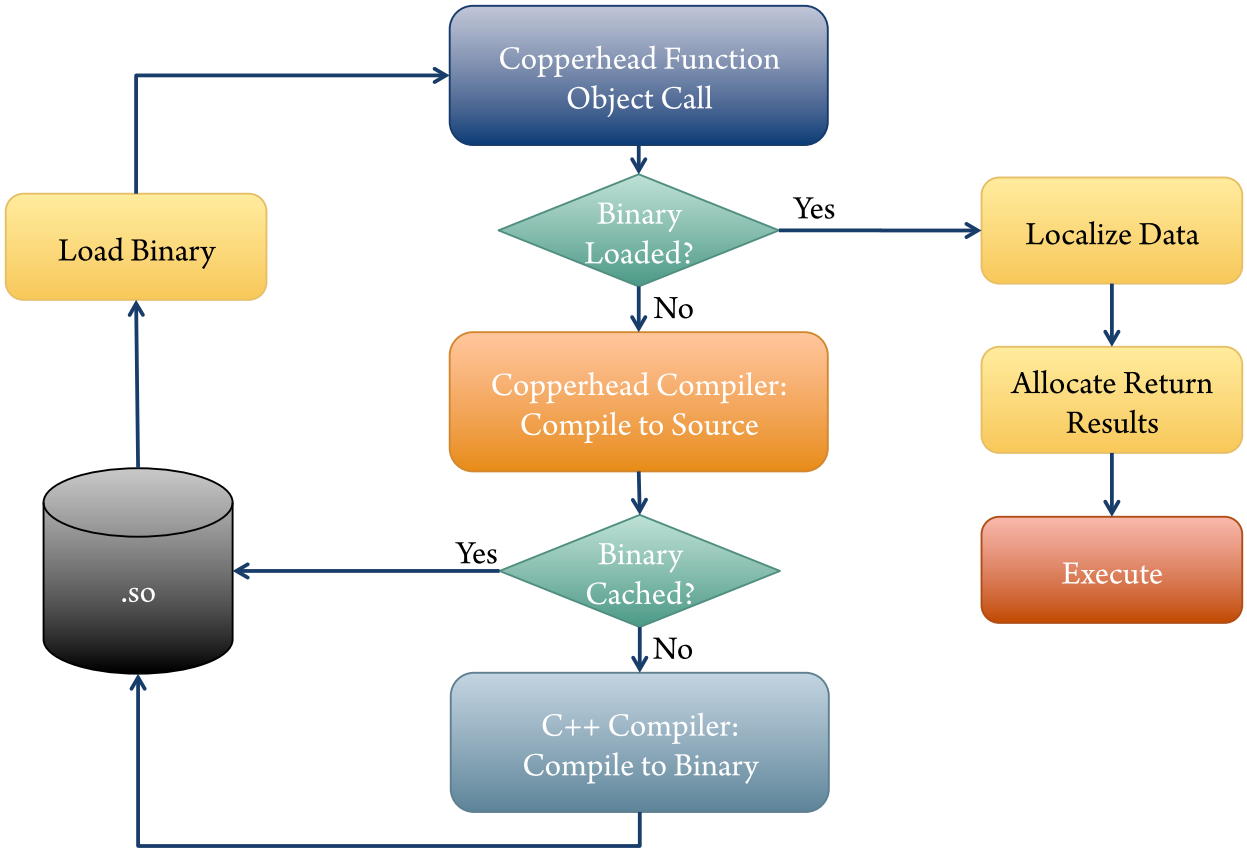



Figure 5.3: Runtime Compilation

with the `@cu` function decorator. When a Copperhead function object is called, the function object first checks a runtime cache to see if a binary appropriate for that function call is available and loaded into the Python interpreter. If so, the binary will be called directly. The first step is to localize all input data to the function, making sure that it exists on the place where the binary will execute. We discuss execution places in Section 5.2.2. Localizing input data may involve data transfers and copies. Additionally, if data is provided to the Copperhead runtime using Python data structures, such as lists or `numpy` arrays, the runtime converts them to Copperhead data structures at this point, assigning all input data a Copperhead type and allowing data to be managed between places by the Copperhead runtime. Once the data is localized and converted to Copperhead data structures, the Copperhead function object calls the Python procedure generated by the Copperhead compiler. The Python driver procedure allocates return results using a memory pool to lower allocation overheads for repeated allocations of similar sized objects. Allocating data in the Python driver procedure ensures that the Python interpreter handles Copperhead data through the standard garbage collection facilities that Python programmers expect. The Python driver function then calls the compiled C++ implementation of the Copperhead function, which allocates temporary variables and steps through a sequence of CUDA kernel invocations. Results are returned directly.

```
# Execution on GPU 0
with places.gpu0:
   z = add_vectors(x, y)

# Native Python execution
with places.here:
   z = add_vectors(x, y)
```

Figure 5.4: Using Execution Places

If the runtime cache does not contain a compiled version of the Copperhead function, it obtains the source code from the underlying Python description of the Copperhead function using standard Python API calls for introspection, in this case `inspect.getsource()`. The Copperhead compiler transforms this input code into the resulting Python, C++, and CUDA code. Invoking the C++ and CUDA compilers is very expensive, so to avoid runtime compilation costs, the compiled binaries are cached persistently on disk. After the Copperhead compiler has generated C++ and CUDA code, it checks this persistent cache to see if compiled binaries are available. If so, it loads them into the Copperhead function object, and the function call proceeds as if the procedure had been present in the runtime cache.

If neither the runtime cache nor the persistent cache contains a compiled binary for the function, the Copperhead runtime invokes C++ and CUDA compilers to build a library object that can be loaded into Python. Copperhead uses PyCUDA [54] and CodePy [55] to provide mechanisms for compiling, persistent caching, linking and executing CUDA and C++ code. The Copperhead runtime uses Thrust [43] to implement fully parallel versions of certain data parallel primitives, such as `reduce` and variations of `scan`.

### 5.2.2 Places

In order to manage the location of data and kernel execution across multiple devices, the Copperhead runtime defines a set of *places* that represent these heterogeneous devices. Data objects are created at a specific place. Calling a Copperhead procedure will execute a computation on the current target place. Figure 5.4 shows how the current target place is is controlled via the Python `with` statement. Currently we support two kinds of places: CUDA capable GPUs and the native Python interpreter. Copperhead is designed to allow other types of places, with corresponding compiler back ends to be added. For instance, multi-core x86 back end would be associated with a new place type.

To faciliate interoperability between Python and Copperhead, all data is duplicated, with a *local* copy in the Python interpreter, and a *remote* copy that resides at the place of execution. Data is lazily transferred between the local and remote place as needed by the program. This eliminates extraneous copy overheads.

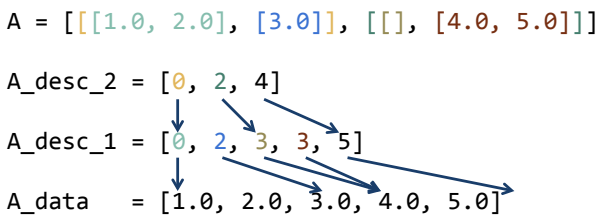



Figure 5.5: Implementing an Arbitrarily Nested Sequence

## 5.3 Data Structures

As discussed in Chapter 3, Copperhead supports one-dimensional sequences in both flat and nested forms, as well as tuples. The Copperhead runtime and compiler are free to implement these data structures in ways that maximize performance. In this section, we discuss the various data structures used to implement Copperhead sequences.

### 5.3.1 Arbitrarily Nested Sequences

Copperhead supports arbitrarily nested sequences with one restriction: all elements of a sequence must have the same type. For example, $[1, 2, 3]$ can be represented as a Copperhead sequence with type `Seq(Int)`. On the other hand, the list $[1, 2, 3.0]$, although a perfectly valid Python list, cannot be represented as a Copperhead sequence, because not all subelements of this list have the same type.

As a corollary to this restriction, the nesting depth of all subsequences must be the same, since the nesting depth of a sequence is exposed in the type system. In other words, $[[1, 2], [3]]$ is representable in Copperhead, with type `Seq(Seq(Int))`. On the other hand, $[[1, 2], 3]$ is not, since the first element has type `Seq(Int)`, while the second element has type `Int`.

Arbitrarily nested sequences make no restriction on the length of their subsequences. For example, it is perfectly legal to have a sequence with type `Seq(Seq(Int))` and shape $\langle$`[2], ⟨[*], Unit⟩`$\rangle$. The outer-most extent describes a one-dimensional sequence with two elements. The inner-most extent is undefined, meaning that each of the two elements have different extents. For example, the first element could be a flat sequence of one million elements and the second element could be an empty sequence.

Copperhead represents arbitrarily nested sequences using a generalization of the common Compressed Sparse Row data structure for sparse matrices. The nested sequence is stored as a single, contiguous flat sequence of data elements, along with a descriptor sequence for each level of nesting. The descriptor sequences provide the necessary information to build a view of each subsequence, including empty subsequences.

For example, consider a nested sequence with type`Seq(Seq(Seq(Int)))` and shape $\langle$`[2], ⟨[2], ⟨[*],Unit⟩⟩`$\rangle$. Figure 5.5 shows such a sequence, along with the descriptor sequences and data sequence that Copperhead uses to represent this sequence.

Each element of a descriptor sequence points to the beginning of a subsequence, in either the next descriptor sequence or the data sequence. The length of each subsequence can be de-

termined by subtracting adjacent elements in the descriptor sequence. An extra index is added to each descriptor sequence to make length calculations for the last subsequence the same as all other subsequences. This data structure allows for arbitrary slicing and dereferencing in constant time, which is important for performance.

This data structure is opaque to the programmer, who does not have access to the descriptor sequence or the underlying flat data sequence.

In other words, arbitrarily nested sequences are represented using a flattened representation. Data parallel compilers that use the flattening transform also use similar flattened representations. However, the fact that the data structure is flattened does not mean the computation operating on the data structure must also be flattened, as we discussed in Chapter 4.

The data structure we employ is flexible and can be implemented efficiently in C++, allowing for arbitrary dereferencing and slicing in constant time. However, it is more flexible than necessary for some applications, and this flexibility comes at an efficiency cost. Arbitrarily nested sequences can not be physically transposed, because transposition would require padding subsequences so that they are all the same length. This padding is not possible in general, due to memory size restrictions. Consider an $n$-element nested sequence like those we used in Section 4.6.2, where all subsequences are $m$ elements long, except for one subsequence with length $\psi m$. The storage required for the data sequence would then be $(\psi + n - 1)m$ elements. The transposed nested sequence would require $\psi mn$ elements of storage. The difference between the amounts of storage needed for these representations can be arbitrarily large, depending on $\psi$, which is generally a data dependent factor. Consequently, transposing arbitrarily nested sequences is not practical.

Attaining efficient performance sometimes requires using the transposed data structure, depending on how the computation has been mapped to the parallelism hierarchy. Transposing the data structure enables adjacent SIMD lanes to load unit-strided data for certain mappings onto the parallelism hierarchy. This is essential for maximum performance on contemporary microprocessors. When adjacent SIMD lanes load unaligned, non-unit-strided data, the memory subsystem operates inefficiently, wasting memory bandwidth, which is often a key performance bottleneck. Accordingly, the use of arbitrarily nested sequences should be reserved for problems where the shape of the sequence is not uniformly nested, and the flexibility is required. For the important special case where the programmer knows that the sequence is uniformly nested, Copperhead allows the use of a different data structure, which we explain in the next section.

### 5.3.2 Uniformly Nested Sequences

Uniformly nested sequences are sequences where the shape is completely defined, or in other words, all subsequences have the same shape. For uniformly nested sequences, a set of strides and lengths are sufficient to describe the nesting structure; descriptor sequences are not required. Uniformly nested sequences also allow the data to be arbitrarily ordered, because they are well defined under arbitrary transposition. When the Copperhead programmer creates a uniformly nested sequence, they either specify the data ordering or provide a tuple of strides directly. This interface allows the programmer to express data layouts analogous to row-

and column-major ordering for doubly nested sequences, extending naturally to more deeply nested sequences. The programmer may provide the strides directly, which allows the subsequences to be aligned arbitrarily, or the programmer may simply describe the ordering and the shape of the nested sequence..

For example, consider the following uniformly nested sequence, with shape $\langle 3, \langle 3, Unit \rangle \rangle$:

$$[[1,2,3],[4,5,6],[7,8,9]]$$

With the lengths defined as $(3,3)$, and the strides defined as $(3,1)$, the data will be laid out in memory as:

$$[1,2,3,4,5,6,7,8,9]$$

This layout is analogous to a row-major layout with no padding. With the strides defined as $(1,3)$, the data is laid out as:

$$[1,4,7,2,5,8,3,6,9]$$

With strides of $(1,4)$, the data is laid out in memory as:

$$[1,4,7,*,2,5,8,*,3,6,9,*]$$

The $*$ elements denote padding elements used for alignment. This layout corresponds to a column-major ordering with each column padded to be a multiple of four elements in length.

Allowing the programmer to construct uniformly nested sequences takes advantage of knowledge the programmer may have about the data being used in a Copperhead program, and can provide important performance benefits when data access patterns with standard nested sequences are not matched well to the processor's memory hierarchy.

The performance differential between using a uniform nested sequence versus a general nested sequence can be large, we have seen performance improvements of two to three times by using the correct data structure. At present, the Copperhead programmer is responsible for choosing the format of uniformly nested sequences, although future work may investigate autotuning over alignments and data layouts.

In order to use uniform nested sequences, the programmer simply constructs a uniform nested sequence in Python, using a constructor provided by the Copperhead runtime. The Copperhead compiler will generate code using the appropriate data container, based on the inputs it receives at compilation time.

## 5.4 Foreign Function Interface

All programming environments exist in a broader ecosystem of pre- and co-existing code. Accordingly, Copperhead does not assume that all useful programs will be implemented natively. Instead, we have made it simple to create libraries for Copperhead that make it possible to use preexisting code within Copperhead programs.

To wrap existing code into a library function callable from within Copperhead code, a programmer creates a wrapper that satisfies the following requirements:

1. A C++ wrapper must be written that operates on Copperhead C++ data structures and is callable from C++.

2. The wrapper must obey the side-effect free semantics of a Copperhead function. If a function being wrapped necessarily operates through side-effects, the wrapper must include the necessary copies to contain the side-effects to within the wrapper.

Once a wrapper has been created, the programmer creates a Python stub function that describes the inputs to the function, decorated with a valid Copperhead type and shape for the function. The wrapper also describes additional compilation flags that should be passed to the C++ compiler, in order to link against the appropriate precompiled libraries.

The Copperhead runtime treats functions using the foreign function interface as black boxes that cannot be optimized. Consequently, no opportunities for data parallel primitive fusion between Copperhead operations and foreign functions will be exploited. Currently, foreign functions can only target the distributed level of the parallelism hierarchy. Future work may relax this restriction to allow the Copperhead compiler to compose foreign functions that target different levels of the parallelism hierarchy, and to permit limited fusion across foreign functions.

The Copperhead runtime uses this foreign function interface for selected parallel primitives, such as `reduce` and `scan`, which are provided by the Thrust library.

## 5.5 Runtime Overheads

In this section, we quantify the various overheads involved in embedding the Copperhead runtime and compiler in Python. Some effort has been made to reduce runtime overheads, but there are still some obvious opportunities to reduce overheads further. In general, the current runtime overheads are low enough that they do not limit performance.

As follows from our discussion in section 5.2.1, there are three types of runtime overhead produced by the Copperhead runtime:

1. Full compilation. This occurs when a particular Copperhead function has not been previously compiled. The runtime cache does not contain an implementation of the function, and neither does the persistent cache. In this case, the Copperhead compiler generates efficiency code, the host compiler compiles efficiency code into a binary that is then serialized in the persistent binary cache, the binary is loaded, inputs to the function are localized to the execution place, and the function is executed.

2. Copperhead compilation. This occurs when a Copperhead function has been compiled previously, but not during this run of the Python interpreter. In this case, the Copperhead compiler generates efficiency code, the runtime locates a previously compiled implementation and loads it into the Python interpreter, inputs to the function are localized, and the function is executed.

3. No compilation. This occurs when a Copperhead function has been compiled previously, during the current run of the Python interpreter. All that must be done is localize

| Benchmark | Full Compilation | Copperhead Compilation | No Compilation |
|---|---|---|---|
| SGEMV | 6.39$s$ | 0.52$s$ | 195$\mu s$ |
| SGBMV | 6.04$s$ | 0.49$s$ | 202$\mu s$ |
| SGEMM | 5.82$s$ | 0.49$s$ | 171$\mu s$ |
| Lanczos iteration | 7.46$s$ | 0.63$s$ | 360$\mu s$ |

Table 5.1: Runtime Overheads

inputs to the function and execute it. This is the most common case in code that executes computations repeatedly, such as in an optimization loop.

Table 5.1 shows these three runtime overheads for various functions. The BLAS functions (SGEMV, SGBMV, SGEMM) are simple Copperhead programs that call external BLAS routines. The Lanczos iteration composes multiple calls to external routines as well as Copperhead generated code. Runtime overheads for the most common case, where the function has previously been called, are small: on the order of hundreds of microseconds. The two-level binary caching system we employ effectively mitigates runtime overheads, since the full impact of compilation is on the order of 10 seconds, while the impact of loading from the persistent cache is on the order of 500 milliseconds. Without the binary caching system we use, runtime overheads would indeed be prohibitive. However, with overheads on the order of hundreds of microseconds for the most common case, most programs will not be bottlenecked by Copperhead runtime issues.

Although our decision to employ runtime compilation could potentially introduce disastrous runtime overheads, these low overheads demonstrate our decision to employ runtime compilation has not compromised our runtime performance.

## 5.6 Systems without compilers

Since the Copperhead runtime depends on efficiency-level compilers to implement compiled code, some might wonder how the Copperhead runtime would operate on systems without access to efficiency-level compilers. There are several reasons why a system would not have access to an efficiency-level compiler. For example, Microsoft Windows systems ship without any compiler, and the most commonly used compilers for Windows come from Microsoft as part of a commercial product. Additionally, some platforms, like Apple's mobile iOS platform, disallow compilation on the platform due to security reasons, since doing so would open up the capability for malicious code to be imported and executed on the device.

The Copperhead runtime is designed to operate on such systems, since they form an important part of today's computing environment. Developers simply to run their Copperhead code on an environment that does have compilers in order to build up a binary cache of all the binaries that a particular program needs. The developer then may ship their code as an archive containing Python code as well as the persistent Copperhead cache, which may then execute as any other Python program does that uses natively compiled libraries.

In other words, the Copperhead runtime strategy is not fundamentally different than the standard compilation flow used by efficiency-layer code. We do not overspecialize at runtime, which would necessitate the constant construction of new binaries, which would then require access to a compiler or the ability to construct and execute arbitary machine code. We compile at runtime as a convenience, in order to fit the mindset of typical productivity programmers, who do not use the standard static compilation flow. We may make use of information gathered at runtime to build up a cache of binary implementations for a particular function, similarly to how autotuners operate. However, our approach is carefully constructed to avoid *requiring* runtime compilation. This also enables the output of our code generator to be used in traditional compilation flows, to integrate with code written in other ways, outside the productivity interpreter, if such a usage mode is desired.

## 5.7 Conclusion

In this section, we have outlined the Copperhead runtime, which is responsible for handling data structures, performing runtime compilation and caching, as well as executing Copperhead programs on heterogeneous targets. We discussed the runtime static compilation model we employ, as well as how it can be applied to platforms without access to compilers. Importantly, we have demonstrated that embedded languages with runtime compilation can be efficient, performing with minimal overhead on the order of a few hundred microseconds per call, which is negligable for many applications. In the following chapter, we examine application performance for Copperhead programs.

# 6 Results

In this chapter, we investigate the performance of Copperhead code in several example programs. We select our example programs from computationally intensive workloads used in Computer Vision and Machine Learning applications, inspired by the Recognition, Mining and Synthesis taxonomy of future computationally intensive workloads [31], [2].

As we compare performance, we compare to published, well-optimized, hand-crafted CUDA implementations of the same computation. Since Copperhead programs look like Python programs and can be executed in the Python interpreter, we could potentially consider comparing to the performance of the same programs running in the Python interpreter. However, the performance of the Python interpreter is very low compared to efficient low-level implementations. Typically we see a factor of 100 to 1000× slowdown when executing sequential code in the Python interpreter compared to C++ performance. Consequently, although such comparisons would lead to large performance improvements from using Copperhead as opposed to using Python, the performance improvements we would cite would be of limited use. Most Python programmers know that if they have major computation to do in Python, they should use a library that implements most of the computation in an efficiency language like C. In fact, most of the Python standard library, as well as numerical computing packages such as `numpy` and `scipy`, are implemented in C, with Python bindings to allow them to be called from within Python. Comparing Copperhead code running in the Python interpreter to compiled Copperhead code running on a parallel platform would therefore be a somewhat pointless comparison. Instead, we compare to hand-written CUDA code to show that the Copperhead compiler produces code that executes with performance comparable to hand-written efficiency code. This is a higher standard, but a more useful one.

Our compiler can't apply all the transformations that a human can, so we don't expect to always achieve the same performance as well-optimized code. Still, we aim to show that high-level data parallel computations can perform within striking distance of human optimized efficiency code.

The tradeoff between performance and programmer productivity is fairly fundamental to any programming environment. We do not claim that our approach eliminates this tradeoff, always providing maximum performance and maximum productivity. Instead, we argue that giving up relatively small factors of performance can yield large productivity benefits. For example, If a programming environment could get within a factor of two of hand-optimized efficiency code, while dramatically simplifying the programming burden required to attain this

performance, we believe this tradeoff would be attractive to many programmers.

Additionally, we illustrate how Copperhead programs can interoperate with widely-used Python modules for numeric computation and data visualization. Since Copperhead is an embedded language, it is straightforward to use the broader capabilities of Python to create fully featured applications, not just code their computationally intensive kernels.

Finally, we discuss programmer productivity in Copperhead. Copperhead programs are significantly terser than handwritten CUDA programs, yet in many cases perform comparably. Although it is difficult to measure programmer productivity without conducting user studies, Copperhead programs are written at a significantly higher level of abstraction than programs written in efficiency languages, which we argue supports our claim that Copperhead is a productive environment for parallel programming.

## 6.1 Sparse Matrix Vector Multiplication

Sparse Matrix Vector Multiplication is an essential kernel used in many different applications, from image and video analysis to finite element methods for structural analysis, among others. We start by examining Copperhead performance on Sparse Matrix Vector Multiplication kernels because they are simple to understand, and the performance results we achieve are illustrative.

We examine the performance of Copperhead generated code for three different SpMV kernels: compressed sparse row, vector compressed sparse row, and ELL. The CSR kernel is generated by compiling the Copperhead procedure for CSR SpMV onto the standard {*Distributed, Sequential*} parallelism hierarchy, which distributes computations along the outermost parallelism dimension to independent threads. Data parallel operations are sequentialized into loops inside each thread. For the CSR kernel, each row of the matrix is then processed by a different thread, and consequently, adjacent threads process widely separated data. This yields suboptimal performance on any microprocessor, since the memory subsystem will load full cache lines of data that are then only partially used, reducing effective memory bandwidth. Figure 3.13 shows Copperhead code for the CSR kernel.

The vector CSR kernel generally improves on the performance of the CSR kernel by mapping the same code to a different parallelism hierarchy: the {*Distributed, Block*} hierarchy, where the outermost data parallel operation is executed via independent thread blocks, and inner data parallel operations are executed via block-wise operations. The Copperhead code for the vector CSR kernel is identical to the code for the scalar CSR kernel, it is just compiled differently. In this case, mapping to the block-wise hierarchy improves memory performance for many sparse matrices, since memory accesses are vectorizeable and therefore more efficient. As we saw in Chapter 4, the choice of parallelism hierarchy is data dependent, and so for some matrices, the scalar CSR kernel is more appropriate. When writing these two variants in efficiency-level code, the resulting implimentations look quite different and cannot be systematically transformed into one another. However, since we compile with direct mapping onto parallelism hierarchies, these two kernels are exposed as compilation variants that can be autotuned over. This is more productive than manually performing two completely different

```
@cu
def spmv_ell(data, idx, x):
    def kernel(i):
        return sum([Aj[i] * x[J[i]] for Aj,J in zip(data, idx)])

    return map(kernel, indices(x))
```

Figure 6.1: SpMV procedure for sparse matrices in ELL format.

$$A = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix}$$

```
vals = [[1,2,5,6],[7,8,3,4],[*,*,9,*]]
cols = [[0,1,0,1],[1,2,2,3],[*,*,3,*]]
```

Figure 6.2: Sparse Matrix and its Representation in ELL Format

mappings, as a programmer would have to do without the techniques we describe in this work.

The ELL representation stores the nonzeros of the matrix in a dense $M$-by-$K$ array, where $K$ bounds the number of nonzeros per row, and rows with fewer than $K$ nonzeros are padded [5]. The array is stored in column-major order, so that after the computation has been mapped to the platform, adjacent threads will be accessing adjacent elements of the array. Figure 6.2 shows a small sparse matrix, with $M = 4$ and $K = 3$, along with its ELL representation. The `vals` array consists of $K$ subsequences, each of $M$ elements. The first subsequence in `vals` consists of the first non-zero element in each row, the second consists of the second non-zero element in each row, etc. The `cols` array matches the `vals` array, notating from which column each non-zero originally came from.

The ELL representation is generally more efficient than the CSR reprentation for matrices where the variance in the number of non-zero entries per row is low. Its efficiency comes from the dense structures used to store the matrix, which can ensure unit-strided memory accesses and dense SIMD operations, even on unstructured matrices. This format is not useful for matrices where there is a large load imbalance in the number of non-zero elements per row, since the ELL format essentially transposes the matrix, and as we discussed in Section 5.3.1, performing this transposition can be infeasible for matrices with large load imbalance factors. However, on matrices that are generally well load balanced, ELL is generally more efficient. Figure 6.1 shows Copperhead code for the ELL kernel.

We compare against CUSP [5], a C++ library for Sparse Matrix Vector multiplication, running on an NVIDIA GTX 480 GPU. We use a suite of eight unstructured matrices that were used by Bell and Garland [5], and that are amenable to ELL storage: in other words, the conversion to ELL format does not introduce prohibitively large numbers of padding elements. Copperhead generated code achieves identical performance for the scalar CSR kernel, and on average provides 45% and 79% of Cusp performance for the vector CSR and ELL kernels, respectively. Table 6.1 gives detailed performance results for the different implementations, kernels, and matrices.

| | Matrix | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Implementation | Dense | Protein | FEM Spheres | FEM Cantilever | FEM Harbor | QCD | FEM Ship | Epidemiology | Average |
| Manual Scalar CSR | 1.3 | 1.1 | 1.2 | 1.2 | 1.2 | 1.2 | 1.2 | 5.9 | 1.8 |
| Copperhead Scalar CSR | 1.3 | 1.1 | 1.2 | 1.1 | 1.2 | 1.2 | 1.2 | 6.1 | 1.8 |
| Manual Vector CSR | 22.4 | 15.4 | 11.2 | 10.7 | 10.8 | 7.5 | 10.0 | 8.6 | 12.1 |
| Copperhead Vector CSR | 14.3 | 7.9 | 4.9 | 4.8 | 4.0 | 3.0 | 3.9 | 0.3 | 5.4 |
| Manual ELL | 4.3 | 13.0 | 17.3 | 15.5 | 9.5 | 19.3 | 13.8 | 14.9 | 13.3 |
| Copperhead ELL | 4.4 | 7.9 | 13.1 | 11.5 | 5.5 | 18.2 | 8.8 | 15.1 | 10.6 |

Table 6.1: Double Precision Sparse Matrix Vector Multiplication Performance in GFLOP/s

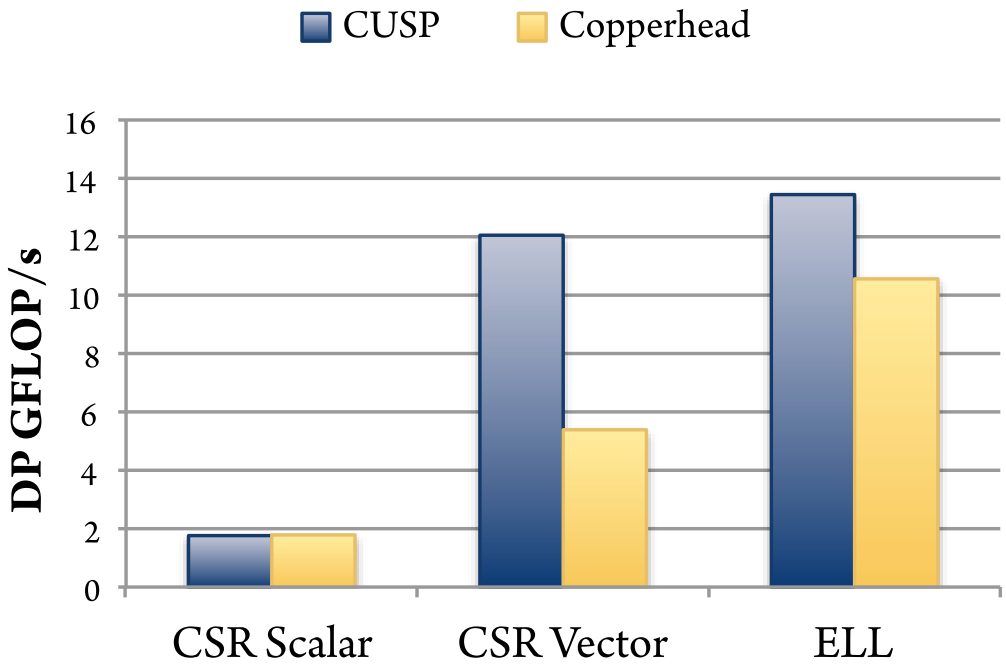



Figure 6.3: Average Double Precision Sparse Matrix Vector Multiply Performance

Our relatively low performance on the vector CSR kernel occurs because of a specialized optimization that the CUSP vector CSR implementation takes advantage of, but the Copperhead compiler does not: namely the ability to compile directly to the SIMD vector level of the parallelism hierarchy. As mentioned previously, we chose not to compile to this level of the parallelism hierarchy because it is not directly exposed in CUDA, and implementations that target it tend to be fairly brittle. This optimization is an important workaround for the limitations of today's CUDA processors, but we considered it too special purpose to implement in the Copperhead compiler. Still, our performance is generally within a factor of two of native CUDA code, which we find encouraging.

## 6.2 Preconditioned Conjugate Gradient Linear Solver

The Conjugate Gradient method is widely used to solve sparse systems of the form $Ax = b$, where $A$ is a positive semi-definite matrix. We examine performance on a preconditioned conjugate gradient solver written in Copperhead, which forms a part of an fixed-point non-linear solver used in Variational Optical Flow methods [85] used for video analysis. For reference, we give the preconditioned conjugate gradient algorithm we use in Algorithm 1.

Figure 6.4 shows the computed optical flow field during the solution of a particular optical flow problem. This figure was generated directly from a Python program using Copperhead to perform computation, using the `matplotlib` Python library. This highlights the ability of

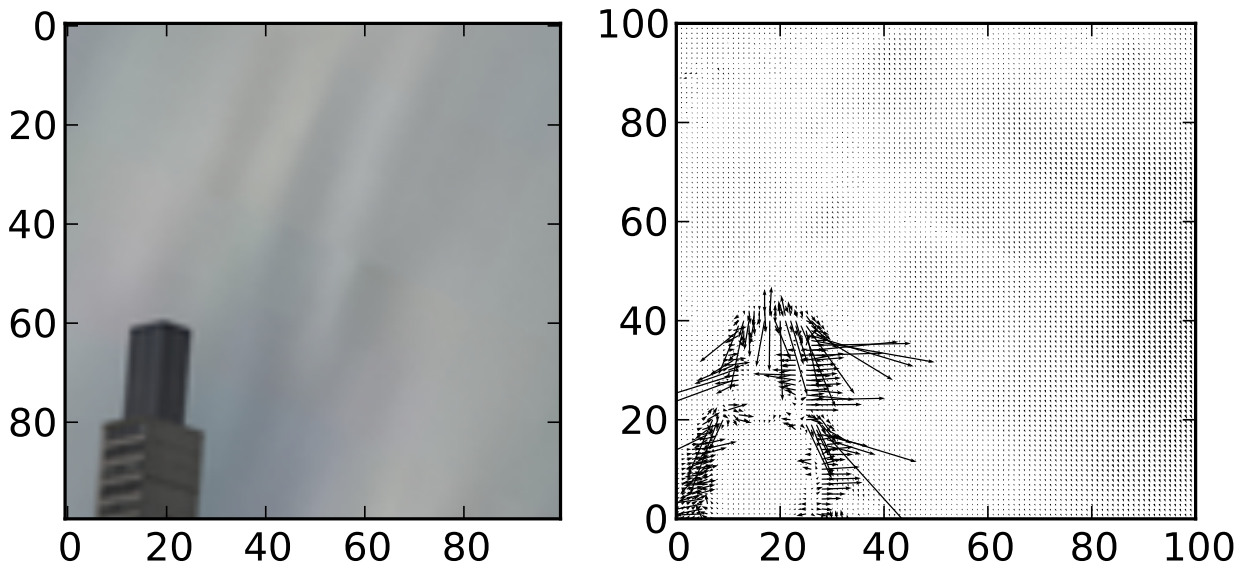



Figure 6.4: Left: closeup of video frame. Right: gradient vector field for optical flow

Copperhead programs to interoperate with other Python modules, in order to write complete programs, and not just their computationally intensive kernels.

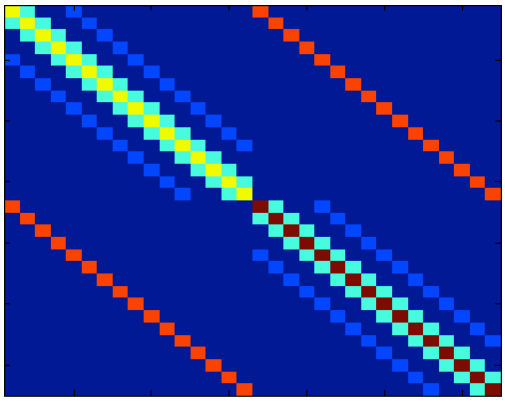


Figure 6.5: Structure of the matrix from the optical flow problem

Conjugate gradient performance depends strongly on matrix-vector multiplication performance. Although we could have used a preexisting sparse-matrix library and representation, in this case we know some things about the structure of the matrix, which arises from a coupled five-point stencil pattern on a vector field, illustrated in Figure 6.5. Taking advantage of this structure, we can achieve significantly better performance than any library routine by creating a custom matrix format and sparse matrix-vector multiplication routine. Copperhead is built for scenarios such as these, where off-the-shelf libraries perform poorly, and the productivity gains we provide make it feasible for programmers to write custom computations that perform more efficiently.

**Algorithm 1** Preconditioned Conjugate Gradient Method

**Input:** PSD matrix $A$, Preconditioner matrix $M^{-1}$, initial vector $\mathbf{x}_0$, right hand side vector $\mathbf{b}$, number of iterations $n$
Initialize: $\mathbf{r}_0 = \mathbf{b} - A\mathbf{x}_0$
Initialize: $\mathbf{z}_0 = M^{-1}\mathbf{r}_0$
Initialize: $\mathbf{p}_0 = \mathbf{z}_0, k = 0$
**repeat**
  $\alpha_k = \frac{\mathbf{r}_k^T \mathbf{z}_k}{\mathbf{p}_k^T A \mathbf{p}_k}$
  $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k$
  $\mathbf{r}_{k+1} = \mathbf{r}_k - \alpha_k A \mathbf{p}_k$
  $\mathbf{z}_{k+1} = M^{-1}\mathbf{r}_{k+1}$
  $\beta_k = \frac{\mathbf{z}_{k+1}^T r_{k+1}}{\mathbf{z}_k^T \mathbf{r}_k}$
  $\mathbf{p}_{k+1} = \mathbf{z}_{k+1} + \beta_k \mathbf{p}_k$
  $k = k + 1$
**until** $k > n$

In addition to writing a custom sparse-matrix vector multiplication routine, practically solving this problem requires the use of a preconditioner, since without preconditioning, convergence is orders of magnitude slower. We utilize a block Jacobi preconditioner for $M^{-1}$ in Algorithm 1. Figure 6.6 shows the Copperhead code for computing the preconditioner, which involves inverting a set of symmetric 2x2 matrices, with one matrix for each point in the vector field, as well as applying the preconditioner, which involes a large number of symmetric 2x2 matrix multiplications. The matrices are represented as three sequences $a, b, c$:

$$m_i = \begin{bmatrix} a_i & b_i \\ b_i & c_i \end{bmatrix}$$

Since we are using a block Jacobi preconditioner, forming the preconditioner involves directly computing the set of inverted matrices

$$p_i = m_i^{-1} = \begin{bmatrix} a_i & b_i \\ b_i & c_i \end{bmatrix}^{-1} = \frac{1}{a_i c_i - b_i b_i} \begin{bmatrix} c_i & -b_i \\ -b_i & a_i \end{bmatrix}$$

Applying the preconditioner is performed by computing many 2x2 matrix multiplications against the vector at each point in the vector field.

Figures 6.7 and 6.8 show the remainder of the Copperhead code for an iteration of this preconditioned conjugate gradient solver.

We implemented the entire solver in Copperhead. The custom SpMV routine for this matrix runs within 10% of the hand-coded CUDA version published in [85], achieving 49 SP GFLOP/s on a GTX 480, whereas a hand-tuned CUDA version achieves 55 SP GFLOP/s on the same hardware. Notably, using an off-the-shelf SpMV format such as ELL for this prob-

```
@cu
def vadd(x, y):
    return map(lambda a, b: return a + b, x, y)
@cu
def vmul(x, y):
    return map(lambda a, b: return a * b, x, y)
@cu
def form_preconditioner(a, b, c):
    def det_inverse(ai, bi, ci):
        return 1.0/(ai * ci - bi * bi)
    indets = map(det_inverse, a, b, c)
    p_a = vmul(indets, c)
    p_b = map(lambda a, b: -a * b, indets, b)
    p_c = vmul(indets, a)
    return p_a, p_b, p_c
@cu
def precondition(u, v, p_a, p_b, p_c):
    e = vadd(vmul(p_a, u), vmul(p_b, v))
    f = vadd(vmul(p_b, u), vmul(p_c, v))
    return e, f
```

Figure 6.6: Forming and applying the block Jacobi preconditioner

lem would sacrifice large amounts of performance; in this case a hand-coded, optimized ELL SpMV would perform about a factor of 2.5× slower than the custom SpMV routine we wrote in Copperhead. Overall, for the complete preconditioned conjugate gradient solver, the Copperhead generated code yields 71% of the performance of the custom CUDA implementation, which is well within a factor of two of hand-optimized efficiency code and therefore represents good performance.

## 6.3 Lanczos Eigensolver

To demonstrate Copperhead's interoperabililty with foreign functions, such as BLAS, we implement a simple eigensolver using the Lanczos Method [28]. This eigensolver is useful for finding the extreme eigenvalues and eigenvectors of a symmetric matrix. Such problems are important in many contexts, for example, the normalized cuts method for image contour detection and segmentation [17]. Algorithm 2 shows the algorithm we employ. Note that in this description, we create a new Lanczos vector $\mathbf{v_i}$ at every iteration of the computation. The Lanczos vectors are accumulated as columns of a matrix $V$, where vector $\mathbf{v_i}$ is column $V_i$ of the matrix. For a problem where the initial Lanczos vector $\mathbf{v_1}$ has $n$ elements, after $i$ iterations, the matrix $V$ of Lanczos vectors then has dimensions $n \times i$. Similarly, $\alpha_i$ and $\beta_i$ are accumulated as entries of vectors **ff** and **fi**.

Like the preconditioned conjugate gradient solver in section 6.2, this algorithm is iterative

```
@cu
def of_spmv(du, dv, width, m1, m2, m3, m4, m5, m6, m7):
    e = vadd(vmul(m1, du), vmul(m2, dv))
    f = vadd(vmul(m2, du), vmul(m3, dv))
    e = vadd(e, vmul(m4, shift(du, -width, 0.0)))
    f = vadd(f, vmul(m4, shift(dv, -width, 0.0)))
    e = vadd(e, vmul(m5, shift(du, -1, 0.0)))
    f = vadd(f, vmul(m5, shift(dv, -1, 0.0)))
    e = vadd(e, vmul(m6, shift(du, 1, 0.0)))
    f = vadd(f, vmul(m6, shift(dv, 1, 0.0)))
    e = vadd(e, vmul(m7, shift(du, width, 0.0)))
    f = vadd(f, vmul(m7, shift(dv, width, 0.0)))
    return (e, f)

@cu
def dot(x, y):
    return sum([xi * yi for xi, yi in zip(x, y)])

@cu
def vsub(x, y):
    return map(op_sub, x, y)

@cu
def init_cg(ux, vx, du, dv, width, m1, m2, m3, m4, m5, m6, m7):
    u, v = of_spmv(ux, vx, width, m1, m2, m3, m4, m5, m6, m7)
    ur = vsub(du, u)
    vr = vsub(dv, v)
    return ur, vr
```

Figure 6.7: Initializing the Conjugate Gradient Iterations

```
@cu
def pre_cg_iteration(ux, vx, ur, vr, ud, vd, uz, vz, width, \
        m1, m2, m3, m4, m5, m6, m7, p1, p2, p3):
    uAdi, vAdi = of_spmv(ud, vd, width, m1, m2, m3, m4, m5, m6, m7)
    urnorm = dot(ur, uz)
    vrnorm = dot(vr, vz)
    rnorm = urnorm + vrnorm
    udtAdi = dot(ud, uAdi)
    vdtAdi = dot(vd, vAdi)
    dtAdi = udtAdi + vdtAdi
    alpha = rnorm / dtAdi
    ux = axpy(alpha, ud, ux)
    vx = axpy(alpha, vd, vx)
    urp1 = axpy(-alpha, uAdi, ur)
    vrp1 = axpy(-alpha, vAdi, vr)
    uzp1, vzp1 = precondition(urp1, vrp1, p1, p2, p3)
    urp1norm = dot(urp1, uzp1)
    vrp1norm = dot(vrp1, vzp1)
    beta = (urp1norm + vrp1norm)/rnorm
    udp1 = axpy(beta, uzp1, urp1)
    vdp1 = axpy(beta, vzp1, vrp1)
    return ux, vx, urp1, vrp1, uzp1, vzp1, udp1, vdp1, rnorm
```

Figure 6.8: Preconditioned Conjugate Gradient Iteration

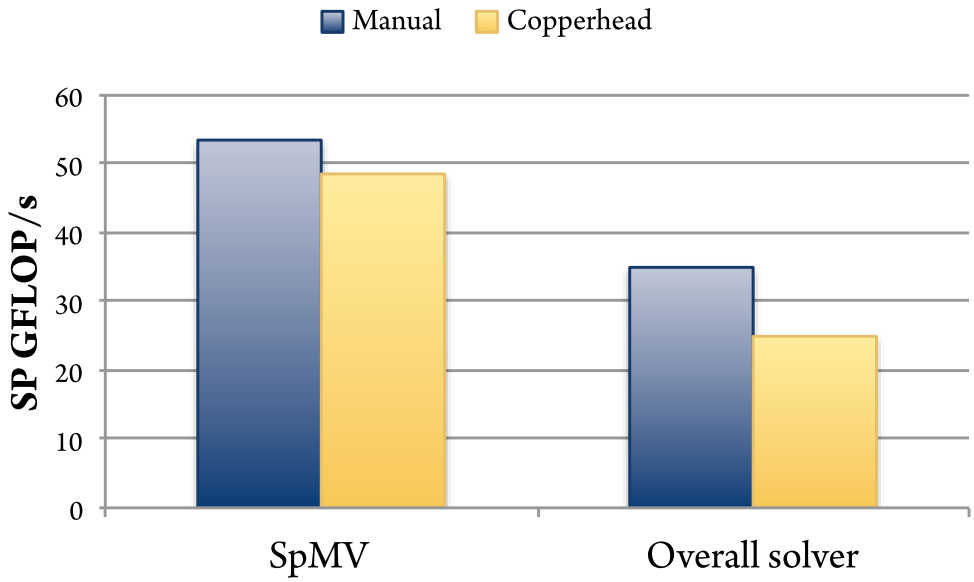



Figure 6.9: Performance on Preconditioned Conjugate Gradient Solver

**Algorithm 2** Lanczos Eigensolver using Full Reorthogonalization

**Input:** Symmetric Matrix $A$, Initial normalized vector $\mathbf{v_1}$ as the first column of a matrix $V$.
Initialize: $i = 1$
**repeat**
  $\mathbf{z}_i = A\mathbf{v}_i$
  $\alpha_i = \mathbf{z}_i^T V_i$
  $\mathbf{z}_i = \mathbf{z}_i - VV^T\mathbf{z}_i$
  $\beta_i = ||\mathbf{z}_i||$
  $V_{i+1} = \mathbf{z}_i / \beta_i$
  $i = i + 1$
**until** Converged

```
from cublas import sgemv # Foreign function library
from spmv import spmv_ell # Native Copperhead library

@cu
def dot(x, y):
    return sum(map(op_mul, x, y))

@cu
def norm(x):
    return sqrt(dot(x, x))

@cu
def lanczos_step(A_data, A_index, V, v_i, alpha):
    # Compute one lanczos iteration.
    z = spmv_ell(A_data, A_index, v_i)
    alpha_i = dot(z, v_i)
    tmp = sgemv(1, 1., V, z, 0., alpha)
    z = sgemv(0, -1., V, tmp, 1.,  z)
    beta_i = norm(z)
    return (alpha_i, beta_i, [zi/beta_i for zi in z])
```

Figure 6.10: Lanczos eigensolver iteration

and depends on a sparse matrix vector multiplication. However, performance also depends on dense matrix operations for reorthogonalization. Convergence testing involves forming $T_j$, the symmetric $j \times j$ tridiagonal matrix with diagonal equal to $\alpha_1, \alpha_2, \ldots, \alpha_j$, and upper diagonal equal to $\beta_1, \beta_2, \ldots, \beta_{j-1}$, and then finding the eigendecomposition $S\Theta S = T_j$. Finding this eigendecomposition is much simpler than the original eigenproblem because the extreme eigenvalues and associated eigenvectors of a symmetric tridiagonal matrix can be found efficiently using bisection and inverse iteration methods [3]. Although nontrivial, convergence testing is not computationally intensive compared to the main Lanczos iteration computation, so we do not consider it in the Copperhead code we describe here.

Figure 6.10 shows the Copperhead code for one iteration of the Lanczos eigensolver. Notice the use of standard Python `import` statements to use external libraries, one of which (`spmv`) is written purely in Copperhead, and one of which (`cublas`) uses the foreign function interface to access vendor supplied BLAS libraries. To the programmer, the distinction between these two cases is irrelevant, even if they differ greatly to the compiler.

Table 6.2 shows some performance results, obtained on a symmetrix sparse matrix with approximately 150k rows and 6.26M non-zero elements, obtained from a normalized cuts problem from image contour detection [17]. For comparison, we also present results from a simple sequential solver written using `scipy` and `numpy`, the popular Python libraries for numeric computation. Although `scipy` and `numpy` are implemented in C and do very little work in the

| Implementation | 1000 Lanczos Iterations | |
|---|---|---|
| Sequential `scipy` | 349.5 | seconds |
| CUDA | 21.4 | seconds |
| Copperhead | 23.1 | seconds |

Table 6.2: Lanczos performance results

Python interpreter, the sequential solver is 15× slower than the Copperhead code running on an NVIDIA GTX 480 GPU. The Copperhead code is only 8% slower than hand-written C++ and CUDA code using CUSP [5] and CUBLAS [67]. This includes all Copperhead and Python runtime overheads. This performance is very good, especially compared to the sequential implementation that standard Python programmers can access.

## 6.4 Quadratic Programming: Nonlinear Support Vector Machine Training

Support Vector Machines are a widely used classification technique from machine learning, in applications ranging from image recognition and bioinformatics, to text classification. Support Vector Machines classify multidimensional data by checking where the data lies with respect to a decision surface. Nonlinear decision surfaces are learned via a Quadratic Programming optimization problem, which we implement in Copperhead.

We consider the standard two-class soft-margin SVM classification problem, which classifies a given data point $x \in \mathbb{R}^n$ by assigning a label $y \in \{-1, 1\}$. In order to do so, a decision surface must be learned from a set of labeled training points. We learn this decision surface through the following Quadratic Programming optimization problem: given a labeled training set consisting of data points $x_i,\ i \in \{1, ..., l\}$ with their corresponding labels $y_i,\ i \in \{1, ..., l\}$, solve Equation 6.1.

$$\begin{aligned} \max_{\alpha}\ & F(\alpha) = \sum_{i=1}^{l} \alpha_i - \frac{1}{2}\alpha^T Q \alpha \\ \text{subject to}\ & 0 \leq \alpha_i \leq C,\ \forall i \in 1 \ldots l \\ & y^T \alpha = 0 \end{aligned} \tag{6.1}$$

In Equation 6.1, $x_i \in \mathbb{R}^n$ is training data point $i$, $y_i \in \{-1, 1\}$ is the label attached to point $x_i$, and $\alpha_i$ is a set of weights, one for each training point, which are being optimized to determine the SVM classifier. $C$ is a parameter that trades classifier generality for accuracy on the training set, and $Q_{ij} = y_i y_j \Phi(x_i, x_j)$, where $\Phi(x_i, x_j)$ is a kernel function that determines the type of decision surface we will be learning. We employ the widely-used Radial Basis Function kernel: $\Phi(x_i, x_j; \gamma) = \exp\{-\gamma ||x_i - x_j||^2\}$, where $\gamma$ is a hyperparameter chosen through crossvalidation tuning, using a separate test set.

### SMO Algorithm

The SVM Training problem can be solved by many methods, each with different parallelism implications. We have implemented the Sequential Minimal Optimization algorithm [72], with a hybrid working set selection heuristic making use of the first order heuristic proposed by [50] as well as the second order heuristic proposed by [32].

The SMO algorithm is a specialized optimization approach for the SVM quadratic program. It takes advantage of the sparse nature of the support vector problem and the simple nature of the constraints in the SVM QP to reduce each optimization step to its minimum form: updating two $\alpha_i$ weights. The bulk of the computation is then to update the Karush-Kuhn-Tucker optimality conditions for the remaining set of weights and then find the next two weights to update in the next iteration. This is repeated until convergence. We state this algorithm briefly in Algorithm 3, for reference purposes.

**Algorithm 3** Sequential Minimal Optimization

**Input:** training data $\mathbf{x}_i$, labels $y_i$, $\forall i \in \{1..l\}$, convergence tolerance $\tau$
Initialize: $\alpha_i = 0$, $f_i = -y_i$, $\forall i \in \{1..l\}$,
Initialize: $b_{high}, b_{low}, i_{high}, i_{low}$
Update $\alpha_{i_{high}}$ and $\alpha_{i_{low}}$
**repeat**
  Update $f_i$, $\forall i \in \{1..l\}$
  Compute: $b_{high}, i_{high}, b_{low}, i_{low}$
  Update $\alpha_{i_{high}}$ and $\alpha_{i_{low}}$
**until** $b_{low} \leq b_{high} + 2\tau$

For the first iteration, we initialize $b_{high} = -1$, $i_{high} = \min\{i : y_i = 1\}$, $b_{low} = 1$, and $i_{low} = \min\{i : y_i = -1\}$.

During each iteration, once we have chosen $i_{high}$ and $i_{low}$, we take the optimization step:

$$\begin{aligned} \alpha'_{i_{low}} &= \alpha_{i_{low}} + y_{i_{low}}(b_{high} - b_{low})/\eta \\ \alpha'_{i_{high}} &= \alpha_{i_{high}} + y_{i_{low}} y_{i_{high}}(\alpha_{i_{low}} - \alpha'_{i_{low}}) \end{aligned}$$

where $\eta = \Phi(x_{i_{high}}, x_{i_{high}}) + \Phi(x_{i_{low}}, x_{i_{low}}) - 2\Phi(x_{i_{high}}, x_{i_{low}})$. To ensure that this update is feasible, $\alpha'_{i_{low}}$ and $\alpha'_{i_{high}}$ must be clipped to the valid range $0 \leq \alpha_i \leq C$.

The optimality conditions can be tracked through the vector $f_i = \sum_{j=1}^{l} \alpha_j y_j \Phi(x_i, x_j) - y_i$, which is constructed iteratively as the algorithm progresses. After each $\alpha$ update, $f$ is updated for all points. This is one of the major computational steps of the algorithm, and is done as follows:

$$\begin{aligned} f'_i = f_i &+ (\alpha'_{i_{high}} - \alpha_{i_{high}}) y_{i_{high}} \Phi(x_{i_{high}}, x_i) \\ &+ (\alpha'_{i_{low}} - \alpha_{i_{low}}) y_{i_{low}} \Phi(x_{i_{low}}, x_i) \end{aligned}$$

```
@cu
def norm2_diff(x, y):
    def el(xi, yi):
        diff = xi - yi
        return diff * diff
    return sum(map(el, x, y))
@cu
def rbf(ngamma, x, y):
    return exp(ngamma * norm2_diff(x,y))
```

Figure 6.11: RBF Kernel evaluation

In order to evaluate the optimality conditions, we define index sets:

$$\begin{aligned} I_{high} &= \{i : 0 < \alpha_i < C\} \cup \{i : y_i > 0, \alpha_i = 0\} \\ &\cup \{i : y_i < 0, \alpha_i = C\} \\ I_{low} &= \{i : 0 < \alpha_i < C\} \cup \{i : y_i > 0, \alpha_i = C\} \\ &\cup \{i : y_i < 0, \alpha_i = 0\} \end{aligned}$$

Because of the approximate nature of the solution process, these index sets are computed to within a tolerance $\epsilon$, e.g. $\{i : \epsilon < \alpha_i < (C - \epsilon)\}$.

We can then measure the optimality of our current solution by checking the optimality gap, which is the difference between $b_{high} = \min\{f_i : i \in I_{high}\}$, and $b_{low} = \max\{f_i : i \in I_{low}\}$. When $b_{low} \leq b_{high} + 2\tau$, we terminate the algorithm.

### Working set selection

During each iteration, we need to choose $i_{high}$ and $i_{low}$, which index the $\alpha$ weights that will be changed in the following optimization step. The first order heuristic from [50] chooses them as follows:

$$\begin{aligned} i_{high} &= \arg\min\{f_i : i \in I_{high}\} \\ i_{low} &= \arg\max\{f_i : i \in I_{low}\} \end{aligned}$$

We compare performance against GPUSVM, a publically available CUDA library for SVM training [18]. Table 6.3 shows the throughput of the Copperhead implementation of SVM training over four datasets, which are detailed in [18]. On average, the Copperhead implementation attains 105% of GPUSVM performance, which is quite good. The reason Copperhead slightly outperforms GPUSVM is that GPUSVM was optimized for an older architecture (the NVIDIA 8800GTX, which was released in 2006), and some of the optimization choices that made sense for older architectures are no longer beneficial. The fact that we can match and in some cases slightly exceed hand-optimized code reinforces our claim that Copperhead code

```
@cu
def argextrema((a_l_idx, a_l_val, a_h_idx, a_h_val),
               (b_l_idx, b_l_val, b_h_idx, b_h_val)):
   if a_l_val < b_l_val:
       if a_h_val > b_h_val:
           return (a_l_idx, a_l_val, a_h_idx, a_h_val)
       else:
           return (a_l_idx, a_l_val, b_h_idx, b_h_val)
   else:
       if a_h_val > b_h_val:
           return (b_l_idx, b_l_val, a_h_idx, a_h_val)
       else:
           return (b_l_idx, b_l_val, b_h_idx, b_h_val)
```

Figure 6.12: Reduction operator for computing the Arg Extrema of a vector

| Dataset | Copperhead Performance SP GFLOP/s | GPUSVM Performance SP GFLOP/s |
|---|---|---|
| Web | 89.0 | 83.5 |
| USPS | 42.5 | 42.8 |
| MNIST | 106.0 | 84.3 |
| Adult | 63.2 | 75.4 |
| Average | 75.2 | 71.5 |

Table 6.3: Support Vector Machine Training Performance

```
@cu
def train(data, labels, ngamma, high, low, \
            alpha, f, d_a_high, d_a_low, \
            eps, ceps, inf, extid):
   def high_evaluation(x):
       return rbf(ngamma, x, high)
   def low_evaluation(x):
       return rbf(ngamma, x, low)
   high_kernels = map(high_evaluation, data)
   low_kernels = map(low_evaluation, data)
   f_p = [fi + d_a_high * hi + \
            d_a_low * li for fi, hi, li \
            in zip(f, high_kernels, low_kernels)]
   def high_member(ai, yi, fi):
       if (ai >= eps and ai <= ceps) or \
           (yi > 0.0 and ai < eps) or \
           (yi < 0.0 and ai > ceps):
           return fi
       else:
           return inf
   def low_member(ai, yi, fi):
       if (ai >= eps and ai <= ceps) or \
           (yi > 0.0 and ai > ceps) or \
           (yi < 0.0 and ai < eps):
           return fi
       else:
           return -inf

   high_values = map(high_member, alpha, labels, f_p)
   low_values = map(low_member, alpha, labels, f_p)

   idxes = indices(f_p)
   zipped = zip4(idxes, high_values, idxes, low_values)
   i_high_p, b_high_p, i_low_p, b_low_p = \
        reduce(argextrema, zipped, extid)

   return f_p, b_high_p, i_high_p, b_low_p, i_low_p
```

Figure 6.13: SVM Training iteration

can be efficiently compiled to contemporary parallel architectures.

## 6.5 Productivity

Productivity is difficult to measure, but as a rough approximation, Table 6.4 provides the number of lines of code needed to implement the core functionality of the examples we have put forth, in C++/CUDA as well as in Python/Copperhead. The same data is provided graphically in Figure 6.14, for reference. On average, the Copperhead programs take about four times fewer lines of code than their C++ equivalents, which suggests that Copperhead programming is indeed more productive than the manual alternatives.

| Example | CUDA & C++ | Copperhead & Python | Ratio |
|---|---|---|---|
| Scalar CSR | 16 | 6 | 2.6 |
| Vector CSR | 39 | 6 | 6.5 |
| ELL | 22 | 4 | 5.5 |
| PCG Solver | 172 | 79 | 2.2 |
| SVM Training | 429 | 111 | 3.9 |
| Average | – | – | 3.6 |

Table 6.4: Number of Lines of Code for Example Programs

Without user studies, we cannot prove that Copperhead is more productive than efficiency language environments, since the number of lines of code may be only weakly correlated with productivity. However, we point out that Copperhead programs are written at a significantly higher level of abstraction than efficiency language programs. There is no need to explicitly parallelize loops or map loops onto the parallel hierarchy of the machine, potentially in multiple ways. The compiler allows programmers to write code as if they were using arrays of structures, which is often the most natural way to express a computation, while still implementing the computation using structures of arrays, which is more efficient. Fusion also introduces issues for efficiency languages, since different code must be written for every combination of fused operations, while Copperhead code is automatically fused. The Copperhead compiler can use on-chip memory resources without being directed to do so. The amount of restructuring the Copperhead compiler does is signficant, and achieving high performance while writing efficiency layer code can only be accomplished if the programmer is educated enough and has the time to perform these restructurings manually. Accordingly, we argue that Copperhead is actually more productive, as the lines of code comparison indicates.

The Copperhead compiler and runtime are freely available on the internet. The most important confirmation of the productivity gains we provide will be Copperhead's adoption by programmers who are looking for more productive ways of parallel programming.

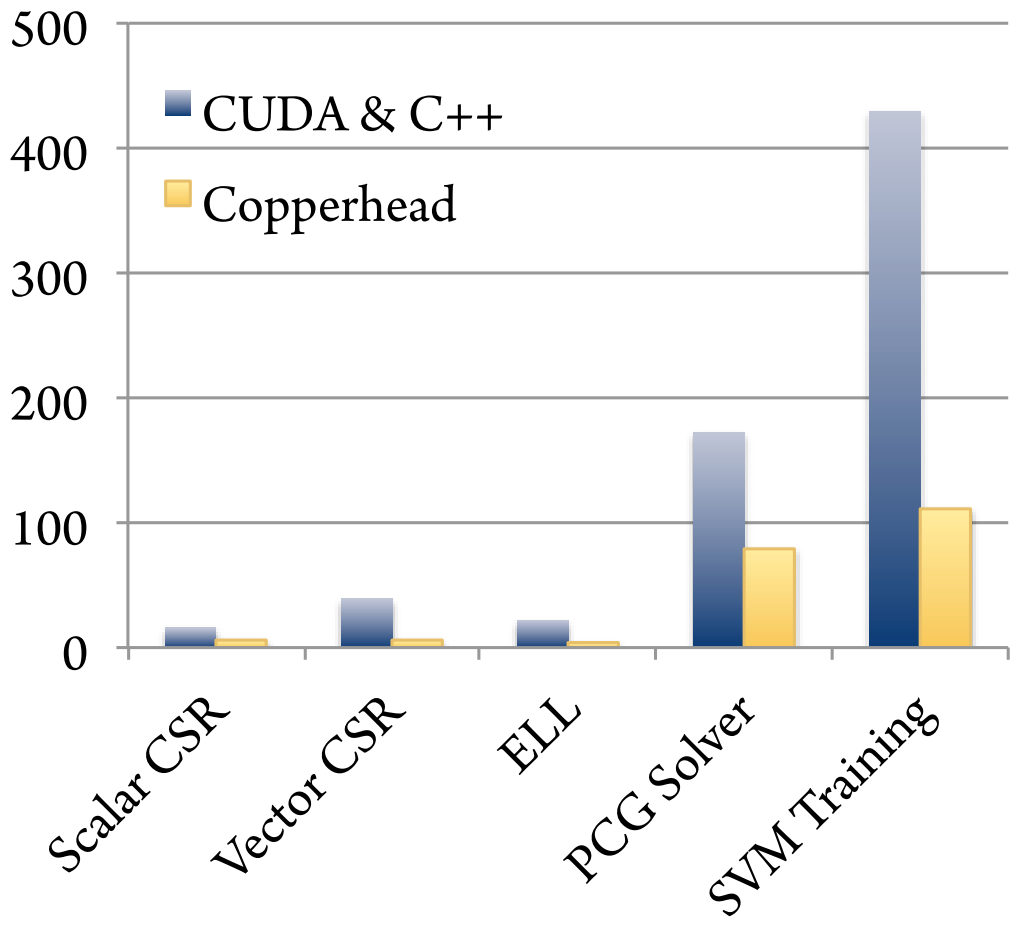



Figure 6.14: Standardized Lines of Code for Copperhead and C++ Programs

## 6.6 Conclusion

In this section, we have shown that our compilation techniques result in efficient code, yielding from 45 to 100% of the performance of hand crafted, well-optimized CUDA code, on example computations from Recognition, Mining and Synthesis workloads. We achieve this high performance with much higher programmer productivity. On average, Copperhead programs require about four times fewer lines of code than CUDA programs, comparing only the core computational portions.

Large communities of programmers choose to use productivity languages such as Python because programmer productivity is often more important than attaining absolute maximum performance. We believe Copperhead occupies a worthwhile position in the tradeoff between productivity and performance, providing performance comparable to that of hand-optimized code, but with a fraction of the programmer effort.

# 7 Conclusions

Parallelism is now mainstream. Developers of computationally intensive applications must employ parallelism in order to capitalize on contemporary microprocessors. Since contemporary microprocessors exploit parallelism in multiple ways, using on-chip multiprocessing to execute independent tasks, while harnessing SIMD for fine-grained data parallelism, efficiently capitalizing on today's parallel processors requires carefully mapping computations onto the parallel hierarchy they provide. Low-level, efficiency programming models allow programmers to do this, but programmers must explicitly perform this mapping, which limits both retargetability as well as productivity. Dependence on low-level programming models will limit the adoption of parallel processing, since most programmers are not interested in carefully fitting a given computation to a target platform. Higher-level productivity programming models have enabled programmers to improve their programming productivity, albeit at a significant performance cost. Additionally, many of them are unable to take advantage of parallelism at all due to semantic restrictions in the language, caused by the very features that make them productive.

To overcome these problems, we advocate the use of embedded data parallel languages such as Copperhead. In this work, we have shown how nested data parallel abstractions, embedded in a productivity language such as Python, can be efficiently compiled and executed on contemporary parallel processors.

## 7.1 Contributions

This work has three main contributions, which we reiterate and contextualize in the following subsections.

### 7.1.1 Direct Mapping of Nested Data Parallelism

There are many ways a computation could be mapped onto a target platform. Traditionally, data parallel compilers have used the flattening transform, which was originally conceived for large, monolithic SIMD array processors like the CM-2. The flattening transform has several good properties: it provides consistent performance despite arbitrary load imbalance, and it maps naturally onto SIMD processors, regardless of their hardware SIMD vector length.

However, our experiments show that on contemporary processors like NVIDIA GPUs, the overheads created by the flattening transform are significant. For simple problems, the inefficiencies are on the order of 3-5× performance loss, and the gap between direct mapping approaches and the flattening transform increases as the problems become more complex, since the flattening transform expands nested data structures into flat data structures, without the possibility for locally reducing them as the computation proceeds.

Accordingly, we advocate a direct mapping strategy, where nested parallelism is mapped directly to the parallelism hierarchy provided by the target platform. For problems with small to moderate load imbalance factor $1 \leq \psi \leq 1000$, direct mapping approaches are up to 5× more efficient than flattened approaches. This is due to hardware on-chip multiprocessing, allowing different cores of the chip to perform different problems, as well as efficient work distributors that provide effective runtime load balancing. The flattening transform should still be used for problems with very high load imbalance $\psi > 1000$, or for problems where direct mappings are not a good fit for the SIMD nature of the machine, for example, if subvector lengths are in the range of $5 - 50$ for a processor with 32-wide logical SIMD units.

Since the majority of problems, including the ones we investigated in Chapter 6, can be mapped directly onto the parallel hierarchy of the target platform, we advocate a direct mapping approach. Stated differently, without a direct mapped approach, we would not be able to report performance within $45\% - 100\%$ of hand-written efficiency code, instead we would be reporting performance in the $10\% - 33\%$ of hand-written efficiency code range. While lower performance may still be an acceptable tradeoff in exchange for improved productivity, we see no need to give up significant performance when a direct mapping strategy allows the programmer to use the same abstractions *and* achieve higher performance.

### 7.1.2 Phase Analysis and Scheduling

After deciding to use a direct mapping strategy for nested parallelism, we turned our attention to how this should be performed. A simple approach would be to instantiate parallelized and sequential loops for every data parallel operation. However, this approach suffers from inefficient usage of memory bandwidth, since between every data parallel operation, the results must be written back to memory, and then reloaded back onto the chip for subsequent operations. Additionally, it introduces extraneous synchronizations, which limits the amount of parallelism the processor can sustain. We investigated the performance loss from the naive approach and found it to be a factor of 9× compared to hand-written efficiency layer code, in which the programmer has fused adjacent data parallel operations together.

Of course, it is not legal to fuse all data parallel operations together, since some of them require synchronization in various ways. We need a program analysis to discover which operations can be fused together while still respecting the semantics of the program. We introduced *phase analysis*, which discovers synchronization points at various levels of the parallel hierarchy using completion and directionality types for parallel primitives provided by the language. Once synchronization points have been discovered, we use a simple heuristic of maximalist fusion to perform *phase scheduling*, which performs fusion while respecting data dependences in the program, at all levels of the parallel hierarchy.

Our strategies for mapping data parallelism to parallel platforms are efficient, enabling us to gain performance between $45 - 100\%$ of handwritten, optimized code.

### 7.1.3 Runtime Static Compilation

Embedded domain specific languages face important choices as to how to perform the embedding in the host language, as well as deal with compilation and execution, each with their own tradeoffs. We follow the SEJITS model [13], which means Copperhead programs exist inside Python programs, and are clearly delineated via standard Python syntactic mechanisms from the surrounding Python code. We perform a limited form of Just-In-Time Specialization, which we refer to as Runtime Static Compilation.

Some languages, like Accelerator [86] and Ct [36], defer execution until a complete abstract syntax tree has been constructed, combining multiple calls into a single abstract syntax tree and performing aggressive runtime specialization once results are required. These approaches suffer from two main problems. Firstly, runtime compilation overhead is exposed at execution time, and without special care, can dominate performance advantages from specialization. Invoking a C++ compiler to compile and link a simple program for a parallel computer requires on the order of ten seconds, while the procedure itself might run only for milliseconds. Without attention to runtime compilation overheads, the practical benefit from an embedded domain specific language can be nullified. Secondly, when traces of function calls are specialized at runtime, taking advantage of large amounts of data only available at runtime, it is difficult to define a particular function as a self-sufficient entity. Runtime overspecialization therefore complicates the use of design-space exploration and makes it difficult to reuse compiled binaries.

In contrast to these approaches, we advocate runtime static compilation, where compilation is initiated at runtime for a given procedure when called from the host language, making only very limited use of information available at runtime. This ensures that the binaries produced by the compiler correspond to a procedure in the original program, and accordingly allows allows the binaries to be reused in other contexts. Runtime static compilation facilitates the use of more expensive design space exploration and autotuning approaches, since the binary can be reused many times and will not be respecialized each time the procedure is called. Finally, runtime static compilation allows the use of compiled embedded domain specific languages even when a compiler is not present on devices to which the program is deployed. As a developer runs their application during development, the runtime builds a cache of compiled binaries that can then be shipped as a compiled program, fully functional even without a compiler on the target device. We showed how a simple two-level binary caching scheme can mitigate compilation overheads, and that runtime overheads can be very low, on the order of hundreds of microseconds per procedure invocation. Since most functions we create run for 1-100 milliseconds or longer, this overhead is generally negligable.

## 7.2 Availability

So that others may use and extend our work, we have released the Copperhead runtime and compiler under the Apache 2.0 license. It is freely available at http://code.google.com/p/copperhead.

## 7.3 Future Work

Many avenues for future work exist in topics related to those examined in this thesis. We list the most important directions, focusing on broadening the reach of the Copperhead compiler by targeting other parallel platforms and incorporating autotuning, improving debugging productivity, and quantifying productivity improvements through user studies.

### 7.3.1 Alternative Backends

We have created a backend for NVIDIA CUDA platforms. However, the approaches we outline are general, and can be extended to efficiently target other parallel platforms as well, such as Intel x86 CPUs, the Intel Manycore Integrated Architecture platform (formerly known as Larrabee), and AMD Fusion APUs, which integrate OpenCL programmable GPUs with multicore x86 CPUs, among others. Providing multiple backends would speed the adoption of Copperhead, since it would not be tied to a particular vendor's product line. It would also provide an existence proof of the generality of the techniques we describe.

### 7.3.2 Autotuning

We envision developers creating custom autotuners for particular problem domains. These autotuners would engage the data parallel compiler with different parameters affecting parallel granularities for different blocking sizes, targeting different parallel hierarchies, and utilizing different data structures, to name a few. It is possible that the autotuning process could benefit from information discovered by the compiler. For example, if the compiler is mapping onto a $\{Distributed, Sequential\}$ hierarchy, using uniform nested sequences, it is a good bet that the input nested sequences should be laid out in memory using a column-major format. The compiler could also export different potential fusion choices as selections for autotuning. Since most autotuners are naturally written in productivity languages like Python, we expect that allowing programmers to exploit their domain specific knowledge about a particular problem through creating a custom autotuner would be straightforward and productive.

If Copperhead were extended to target other platforms, the need for a robust autotuning infrastructure would increase, since architectural diversity would require different tuning approaches. Additionally, heuristics, such as those used in phase scheduling and on-chip memory placement, among others, could be replaced with autotuning, which would improve the flexibility of the Copperhead compiler.

### 7.3.3 Aspect Oriented Debugging

High productivity programming requires high productivity debugging. Writing computer programs necessarily involves fixing mistakes. Debugging higher level programs can be difficult, since there is often not a clear transliteration of the program the programmer wrote into something the machine can execute. For example, temporary variables visible in the original source code may be elided completely by the Copperhead compiler, so that they are never actually fully formed in memory in a state where they can be examined through traditional breakpointing techniques.

To improve debugging productivity, the user could provide debugging requests in an aspect oriented manner, separated from the actual code being debugged. These debugging requests might ask for a particular variable to be fully instantiated so that intermediate state can be visualized, or might provide instructions on how state should be analyzed or visualized, in order to overcome the "needle-in-a-haystack" problem that often arises when debugging a parallel program where most of the computation was correct, but there was a subtle bug affecting a small portion of the data.

### 7.3.4 Usability Studies

Quantifying productivity is difficult. In our work we presented standardized lines of code on several examples, to show that data parallel programs expressed in Copperhead are more concise than their equivalents in C++. However, conciseness is perhaps only correlated with productivity. Convincingly demonstrating a true productivity advantage of our approach can only be done by inviting programmers to use it, and studying their productivity. We have started this work by opening Copperhead as a freely distributed project, we anticipate that user feedback will help us improve Copperhead as well as show compelling usability benefits. However, more usability studies would be useful to quantify the productivity benefits of a programming system like Copperhead.

## 7.4 Conclusion

The adoption of new programming environments is often slow and difficult. It is hard for programmers to learn a new way of thinking about their problem, grapple with a new set of abstractions and new runtime model. However, closing the implementation gap requires software infrastructure that can take higher level abstractions of parallel programs and transform them into efficient low level implementations.

If new software infrastructure is required, it is important that it be minimally disruptive. For this reason, we have created Copperhead as an embedded language. Programmers already familiar with Python and its widely used libraries for numeric and scientific computing, as well as data visualization, file input/output, etc., can view Copperhead as another Python library they can use, which interoperates with the ones they already know. For example, the output of Copperhead code can be visualized directly using `matplotlib`, and existing Python code

can perform file input/output, or connect to the internet to download or distribute datasets. We believe that creating Copperhead as an embedded language makes Copperhead a productive environment for programming entire applications, not just their computationally intensive kernels.

The abstractions that Copperhead employs are also designed to be minimally disruptive. Python already provides `map`, which is the foundation of data parallelism in Copperhead, as well as `reduce`, which is also a fundamental data parallel primitive. Many of the other primitives provided by Copperhead are familiar to those who have used other data parallel languages, such as NESL.

Finally, runtime static compilation allows Copperhead programs to be written similarly to other Python programs, despite the fact that running Copperhead programs invokes the Copperhead compiler as well as efficiency compilers for the parallel device being targeted. This makes Copperhead simple to use, there is no new compilation machinery that the programmer must master in order to use Copperhead. At the same time, it allows more sophisticated programmers to reuse the results of the Copperhead compiler in other programs, since calling a Copperhead procedure from Python creates an efficient low-level implementation of that procedure that has not been overspecialized.

Although learning to use a new programming environment is difficult, we believe Copperhead has several compelling benefits that will make it attractive to application developers. Firstly, the abstractions that we employ can be compiled efficiently to contemporary parallel processors, within $45 - 100\%$ of hand-optimized efficiency code. Secondly, since Copperhead is an embedded language, there is no new syntax to learn, only a library of data parallel primitives and associated data structures. Thirdly, the need for more productive parallel programming is acute, and the increased productivity Copperhead provides will motivate programmers to take advantage of its capabilities.

We believe that this work demonstrates a productive and efficient way of programming contemporary parallel processors, thus helping to close the implementation gap that jeopardizes the prospects of parallel processing.

# Bibliography


[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1986.

[2] Krste Asanović, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[3] Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, and H. van der Vorst. *Templates for the solution of Algebraic Eigenvalue Problems: A Practical Guide*. SIAM, 2000.

[4] G.H. Barnes, R.M. Brown, M. Kato, D.J. Kuck, D.L. Slotnick, and R.A. Stokes. The ILLIAC IV Computer. *Computers, IEEE Transactions on*, C-17(8):746 – 757, August 1968.

[5] Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *SC '09: Proc. Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11. ACM, 2009.

[6] T. Blank. The MasPar MP-1 architecture. In *Compcon Spring '90. Intellectual Leverage. Digest of Papers. Thirty-Fifth IEEE Computer Society International Conference.*, pages 20 –24, 1990.

[7] Guy E. Blelloch. *Vector models for data-parallel computing*. MIT Press, Cambridge, MA, USA, 1990.

[8] Guy E. Blelloch. NESL: A Nested Data-Parallel Language. (Version 3.1). Technical Report CMU-CS-95-170, Carnegie Mellon University, 1995.

[9] Guy E. Blelloch. Programming parallel algorithms. *Commun. ACM*, 39(3):85–97, 1996.

[10] Guy E. Blelloch, Jonathan C. Hardwick, Siddhartha Chatterjee, Jay Sipelstein, and Marco Zagha. Implementation of a portable nested data-parallel language. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '93, pages 102–111, New York, NY, USA, 1993. ACM.

[11] Bryan Catanzaro, Armando Fox, David Patterson, Bor-Yiing Su, Marc Snir, Kunle Olukotun, Pat Hanrahan, and Hassan Chafi. Ubiquitous Parallel Computing from Berkeley, Illinois and Stanford. *IEEE Micro*, 30(2):41–55, 2010.

[12] Bryan Catanzaro, Michael Garland, and Kurt Keutzer. Copperhead: Compiling an embedded data parallel language. In *Principles and Practices of Parallel Programming (PPoPP)*, pages 47–56, 2011.

[13] Bryan Catanzaro, Shoaib Kamil, Yunsup Lee, Krste Asanović, James Demmel, Kurt Keutzer, John Shalf, Kathy Yelick, and Armando Fox. SEJITS: Getting Productivity and Performance With Selective Embedded JIT Specialization. Technical Report UCB/EECS-2010-23, EECS Department, University of California, Berkeley, 2010.

[14] Bryan Catanzaro and Kurt Keutzer. Parallel computing with patterns and frameworks. *XRDS: Crossroads, the ACM Magazine for Students*, 17:22–27, 2010.

[15] Bryan Catanzaro, Kurt Keutzer, and Bor-Yiing Su. Parallelizing CAD: A timely research agenda for EDA. In *Design Automation Conference*, 2008.

[16] Bryan Catanzaro and Brent Nelson. Higher radix floating-point representations for fpga-based arithmetic. *Field-Programmable Custom Computing Machines, Annual IEEE Symposium on*, pages 161–170, 2005.

[17] Bryan Catanzaro, Bor-Yiing Su, Narayanan Sundaram, Yunsup Lee, Mark Murphy, and Kurt Keutzer. Efficient, High-Quality Image Contour Detection. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2381–2388, 2009.

[18] Bryan Catanzaro, Narayanan Sundaram, and Kurt Keutzer. Fast Support Vector Machine Training and Classification on Graphics Processors. In *Proceedings of the 25th International Conference on Machine Learning*, pages 104–111, 2008.

[19] Bryan Catanzaro, Narayanan Sundaram, and Kurt Keutzer. A map reduce framework for programming graphics processors. In *Workshop on Software Tools for Multi-Core systems*, 2008.

[20] Hassan Chafi, Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Anand R. Atreya, and Kunle Olukotun. A domain-specific approach to heterogeneous parallelism. In *PPOPP*, pages 35–46, 2011.

[21] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data parallel Haskell: a status report. In *Proc. 2007 Workshop on Declarative Aspects of Multicore Programming*, pages 10–18. ACM, 2007.

[22] A.E. Charlesworth. An Approach to Scientific Array Processing: The Architectural Design of the AP-120B/FPS-164 Family. *Computer*, 14(9):18 –27, sept. 1981.

[23] Siddhartha Chatterjee, Guy E. Blelloch, and Allan L. Fisher. Size and access inference for data-parallel programs. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, PLDI '91, pages 130–144, New York, NY, USA, 1991. ACM.

[24] J.C. Chaves, J. Nehrbass, B. Guilfoos, J. Gardiner, S. Ahalt, A. Krishnamurthy, J. Unpingco, A. Chalker, A. Warnock, and S. Samsi. Octave and Python: High-level scripting languages productivity and performance evaluation. In *HPCMP Users Group Conference, 2006*, pages 429–434, June 2006.

[25] Jike Chong. *Pattern-Oriented Application Frameworks for Domain Experts to Effectively Utilize Highly Parallel Manycore Microprocessors*. PhD thesis, University of California, Berkeley, 2010.

[26] Clyther: Python Language Extension for OpenCL. http://clyther.sourceforge.net/, 2010.

[27] CUDPP: CUDA Data-Parallel Primitives Library. http://www.gpgpu.org/developer/cudpp/, 2009.

[28] Jane K. Cullum and Ralph A. Willoughby. *Lanczos Algorithms for Large Symmetric Eigenvalue Computations*. Vol. I: Theory. SIAM, 2002.

[29] Cython: C-extensions for python. http://cython.org/, 2010.

[30] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '82, pages 207–212, New York, NY, USA, 1982. ACM.

[31] Pradeep Dubey. Recognition, mining and synthesis moves computers to the era of tera. *Technology Intel Magazine*, 2005.

[32] Rong-En Fan, Pai-Hsuen Chen, and Chih-Jen Lin. Working set selection using second order information for training support vector machines. *J. Mach. Learn. Res.*, 6:1889–1918, 2005.

[33] Rahul Garg and José Nelson Amaral. Compiling python to a hybrid execution environment. In *GPGPU '10: Proc. 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 19–30. ACM, 2010.

[34] J.-L. Gaudiot, W. Bohm, W. Najjar, T. DeBoni, J. Feo, and P. Miller. The Sisal model of functional programming and its implementation . In *Parallel Algorithms/Architecture Synthesis, 1997. Proceedings. Second Aizu International Symposium*, pages 112 –123, March 1997.

[35] P.P. Gelsinger. Microprocessors for the new millennium: Challenges, opportunities, and new frontiers. In *Solid-State Circuits Conference, 2001. Digest of Technical Papers. ISSCC. 2001 IEEE International*, pages 22 –25, 2001.

[36] Anwar Ghuloum, Eric Sprangle, J. Fang, G. Wu, and X. Zhou. Ct: A Flexible Parallel Programming Model for Tera-scale Architectures. Technical Report White Paper, Intel Corporation, 2007.

[37] Tianyi David Han and Tarek S. Abdelrahman. hiCUDA: a high-level directive-based language for GPU programming. In *GPGPU-2: Proc. 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 52–61. ACM, 2009.

[38] Jonathan M. D. Hill. Data parallel haskell: Mixing old and new glue. Technical Report 611, Queen Mary and Westfield College, University of London, 1993.

[39] W. Daniel Hillis. *The Connection Machine*. MIT Press, 1989.

[40] W. Daniel Hillis and Jr. Guy L. Steele. Data parallel algorithms. *Commun. ACM*, 29(12):1170–1183, 1986.

[41] W. Daniel Hillis and Lewis W. Tucker. The CM-5 Connection Machine: a scalable supercomputer. *Commun. ACM*, 36:31–40, November 1993.

[42] R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, pages 29–60, 1969.

[43] Jared Hoberock and Nathan Bell. Thrust: A parallel template library, 2009. Version 1.2.

[44] Mark Horowitz and Wikipedia, 2011. Data gathered from `http://en.wikipedia.org/wiki/Intel_Core` and personal communications with Mark Horowitz.

[45] Paul Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28, December 1996.

[46] Paul Hudak and Mark P. Jones. Haskell vs. Ada vs. C++ vs. Awk vs...an experiment in software prototyping productivity. Technical Report YALEU/DCS/RR-1049, Yale University Department of Computer Science, New Haven, CT, 1994.

[47] John D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, May-Jun 2007.

[48] Kenneth E. Iverson. A programming language. In *Proceedings of the May 1-3, 1962, spring joint computer conference*, AIEE-IRE '62 (Spring), pages 345–351, New York, NY, USA, 1962. ACM.

[49] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–.

[50] S. S. Keerthi, S. K. Shevade, C. Bhattacharyya, and K. R. K. Murthy. Improvements to Platt's SMO Algorithm for SVM Classifier Design. *Neural Comput.*, 13(3):637–649, 2001.

[51] Ken Kennedy and Kathryn McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In Utpal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing*, volume 768 of *Lecture Notes in Computer Science*, pages 301–320. Springer Berlin / Heidelberg, 1994.

[52] Kurt Keutzer and Tim Mattson. A Design Pattern Language for Engineering (Parallel) Software. *Intel Technology Journal, Addressing the Challenges of Tera-scale Computing*, 13(4), 2010.

[53] B. Khailany, W.J. Dally, U.J. Kapasi, P. Mattson, J. Namkoong, J.D. Owens, B. Towles, A. Chang, and S. Rixner. Imagine: media processing with streams. *Micro, IEEE*, 21(2):35 –46, March 2001.

[54] Andreas Klöckner. PyCUDA, 2009. http://mathema.tician.de/software/pycuda.

[55] Andreas Klöckner. Codepy. http://mathema.tician.de/software/codepy, 2010.

[56] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan C. Catanzaro, Paul Ivanov, and Ahmed Fasih. Pycuda: Gpu run-time code generation for high-performance computing. *CoRR*, abs/0911.3456, 2009.

[57] C.E. Kozyrakis, S. Perissakis, D. Patterson, T. Anderson, K. Asanović, N. Cardwell, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, R. Thomas, N. Treuhaft, and K. Yelick. Scalable processors in the billion-transistor era: IRAM. *Computer*, 30(9):75 –78, September 1997.

[58] Sanjeev Kumar, Christopher J. Hughes, and Anthony Nguyen. Carbon: architectural support for fine-grained parallelism on chip multiprocessors. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, pages 162–173, New York, NY, USA, 2007. ACM.

[59] Sean Lee, Manuel M. T. Chakravarty, Vinod Grover, and Gabriele Keller. GPU kernels as data-parallel array computations in Haskell. In *Workshop on Exploiting Parallelism using GPUs and other Hardware-Assisted Methods (EPAHM 2009)*, 2009.

[60] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. In *Proc. 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 101–110. ACM, 2009.

[61] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, March 2008.

[62] Timothy Mattson, Beverly Sanders, and Berna Massingill. *Patterns for parallel programming*. Addison-Wesley Professional, first edition, 2004.

[63] Michael McCool. Data-parallel programming on the Cell BE and the GPU using the RapidMind development platform. In *Proc. GSPx Multicore Applications Conference*, November 2006.

[64] Michael D. McCool, Zheng Qin, and Tiberiu S. Popa. Shader metaprogramming. In *Proc. Graphics Hardware 2002*, pages 57–68. Eurographics Association, 2002.

[65] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, March 2008.

[66] J.R. Nickolls. The design of the MasPar MP-1: a cost effective massively parallel computer. In *Compcon Spring '90. Intellectual Leverage. Digest of Papers. Thirty-Fifth IEEE Computer Society International Conference.*, pages 25 –28, 1990.

[67] Nvidia. Nvidia CUDA, 2007. http://nvidia.com/cuda.

[68] NVIDIA Corporation. *NVIDIA CUDA C Programming Guide*, May 2010. Version 3.1.

[69] Wilfried Oed. Cray Y-MP C90: System features and early benchmark results. *Parallel Computing*, 18(8):947 – 954, 1992.

[70] Travis E. Oliphant. Python for scientific computing. *Computing in Science and Engineering*, 9(3):10–20, 2007.

[71] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, and J.C. Phillips. GPU Computing. *Proceedings of the IEEE*, 96(5):879 –899, May 2008.

[72] John C. Platt. Fast training of support vector machines using sequential minimal optimization. In *Advances in kernel methods: support vector learning*, pages 185–208. MIT Press, Cambridge, MA, USA, 1999.

[73] William L. Plishker. *Automated Mapping of Domain Specific Languages for Application Specific Multiprocessors*. PhD thesis, University of California, Berkeley, 2006.

[74] Lutz Prechelt. An empirical comparison of seven programming languages. *IEEE Computer*, 33(10):23–29, Oct 2000.

[75] Lutz Prechelt. Are Scripting Languages Any Good? A Validation of Perl, Python, Rexx, and Tcl against C, C++, and Java. In *Advances in Computers*, volume 57, pages 205 – 270. Elsevier, 2003.

[76] The Python language reference. http://docs.python.org/2.6/reference, October 2008. Version 2.6.

[77] Richard M. Russell. The CRAY-1 computer system. *Commun. ACM*, 21:63–72, January 1978.

[78] Gary W. Sabot. *The Paralation Model: Architecture-Independent Parallel Programming*. MIT Press, Cambridge, MA, USA, 1989.

[79] Jason Sanders and Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley, July 2010.

[80] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27:18:1–18:15, August 2008.

[81] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan primitives for GPU computing. In *GH '07*, pages 97–106, 2007.

[82] Niraj R. Shah. *Programming Models for Application-Specific Instruction Processors*. PhD thesis, University of California, Berkeley, 2004.

[83] Daniel L. Slotnick, W. Carl Borck, and Robert C. McReynolds. The solomon computer. In *Proceedings of the December 4-6, 1962, fall joint computer conference*, AFIPS '62 (Fall), pages 97–107, New York, NY, USA, 1962. ACM.

[84] J.E. Stone, D. Gohara, and Guochun Shi. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science Engineering*, 12(3):66 –73, May 2010.

[85] Narayanan Sundaram, Thomas Brox, and Kurt Keutzer. Dense Point Trajectories by GPU-accelerated Large Displacement Optical Flow. In *European Conference on Computer Vision*, pages 438–445, September 2010.

[86] David Tarditi, Sidd Puri, and Jose Oglesby. Accelerator: Using Data Parallelism to Program GPUs for General-Purpose Uses. In *ASPLOS 2006*, pages 325–335, 2006.

[87] Theano: A python array expression library. http://deeplearning.net/software/theano/, 2010.

[88] L.W. Tucker and G.G. Robertson. Architecture and applications of the connection machine. *Computer*, 21(8):26 –38, August 1988.

[89] Åke Wikström. *Functional Programming using Standard ML*. Prentice-Hall, 1988.

[90] Michael Wolfe. Implementing the PGI Accelerator model. In *Proc. 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 43–50. ACM, 2010.