PRODUCTIVITY AND PERFORMANCE WITH EMBEDDED DOMAIN-SPECIFIC LANGUAGES

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Arvind K. Sujeeth
May 2014

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Oyekunle Olukotun, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Christos Kozyrakis

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Chris Re

Approved for the Stanford University Committee on Graduate Studies.

Patricia J. Gumport, Vice Provost for Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format. An original signed hard copy of the signature page is on file in University Archives.

Abstract

Modern computing is transitioning from a predominantly sequential processing model to heterogeneous platforms that combine sequential, parallel, specialized, and distributed processors. However, mainstream tools and programming languages have not kept up with these changes. As a result, programmers often use sequential programming models that do not take full advantage of the computational resources available.

The primary reason programming languages have not been able to keep up is simple: programming heterogeneous and parallel systems is difficult. It requires expertise in everything from the computer architecture of low-level devices to compiler optimization to synchronization primitives and communication protocols. In particular, the generality of high-level programming language abstractions makes it difficult to perform sophisticated parallel optimization or generate code efficiently for different devices. Instead, programmers who require high performance must often abandon high-level abstractions and program using low-level models (e.g. CUDA [85], OpenCL [118]) or explicitly parallel models and languages (e.g. MapReduce [33], X10 [25]). While these systems at least enable programmers to achieve high performance, they also set up a trade-off between productivity and performance: programmers must take far more responsibility for low-level implementation details than they do in high-level sequential models. Implementing a high performance solution is time-consuming, error-prone, and usually not portable to different devices or datasets. This additional programming complexity means that high performance solutions are out of reach for most non-experts.

We believe that the specialization of programming languages into domain-specific

languages (DSLs) enables new opportunities for productive high performance programming on modern systems. DSLs have been around for decades, but have traditionally been focused on improving programmer productivity. We demonstrate that DSL compilers can also exploit high-level data and control structure and domain-specific optimizations to efficiently compile implicitly parallel DSLs to multicore CPUs, GPUs and clusters. By sacrificing generality, we achieve both productivity (DSL users use high-level abstractions to write their program once) and performance (the programs are compiled to run on heterogeneous systems). We use the Lightweight Modular Staging (LMS) [101] and Delite [14] frameworks as a starting point. LMS and Delite are Scala frameworks that provide common, reusable components that simplify the implementation of embedded DSL compilers. We use LMS to stage Scala programs (a form of metaprogramming that defers the execution of some statements) in order to build an intermediate representation (IR). We use Delite to organize our DSLs in terms of parallel patterns and to optimize and compile the patterns to different devices.

In this dissertation, we present the key aspects of LMS and Delite that are required to construct high performance embedded DSL compilers. We develop OptiML, an implicitly parallel DSL for machine learning, and describe its design and implementation. OptiML is mostly functional, but aggressively optimizes away intermediate data structures in order to achieve performance comparable to an imperative, mutable application. We show that OptiML is both productive and performant (it outperforms MATLAB [78] and is competitive with hand-optimized C++ in nearly all cases). We present new techniques for composing compiled embedded DSLs, and validate these techniques by implementing and composing new compiled embedded DSLs for data querying (OptiQL), graph analysis (OptiGraph), scientific computing (OptiMesh) and collections (OptiCollections). The ability to compose compiled DSLs makes it possible to use them like libraries while still achieving high performance. Furthermore, we present a case study on the degree of reuse achieved across the DSL implementations and show that common compiler infrastructure has a real-world impact in reducing the effort required to build a DSL. Finally, we introduce Forge, a new meta DSL for DSL development that provides a high-level specification language for

parallel and heterogeneous DSLs. Forge can generate multiple implementations of a DSL, including a pure Scala library (interpreted) version and a Delite (compiled) version. The interpreted version can be used interactively, effectively producing a DSL read-evaluate-print-loop (REPL). Forge makes Delite DSL development comparable to library development while also enabling users to prototype and debug their applications using standard Scala tools before switching to the high performance version (which can be done without changing the source code). This body of work demonstrates for the first time that embedded DSLs can approximate both the productivity of libraries and the performance of hand-optimized code.

Acknowledgements

There are many people who have their fingerprints on my journey as a Ph.D. student. Without them, this document could not have been produced. I owe them all a great debt of gratitude.

First, I would like to thank my advisor, Kunle Olukotun, who shepherded me through this process without making it a burden. His advice and patience allowed me to explore without running too far astray, and he has seeded many of the ideas in this dissertation. Christos Kozyrakis and Pat Hanrahan have provided me guidance throughout my graduate career. Chris Ré has been the source of countless enlightening and humorous discussions that have changed my perspectives. Stephen Boyd and Andrew Ng have taught me a great deal, both in their challenging classes and in private conversation.

Across the pond, Martin Odersky and the Scala team at EPFL have been instrumental in this work. Without their collaboration, the result would have looked much different, and far worse. I am also grateful for their hospitality in hosting me in Lausanne for six weeks as we laid early foundations.

There are too many colleagues and friends to thank. Students in Kunle's, Pat's, and Alex's groups, among others, have taught me so much over the years. In particular, Nathan Bronson and Zach DeVito have provided invaluable insights into different projects at different times. My friends from Lyman, around campus, and in San Francisco have kept my spirits up when things looked dark. Most of all, the Delite team has been a second family to me over the course of my Ph.D. Hassan Chafi brought me into the fold and pushed me to think harder and better. Kevin Brown, HyoukJoong Lee, and Tiark Rompf have been my partners in crime as we built these systems from

the ground up. More recently, Chris De Sa, Chris Aberger and Jithin Thomas have helped pick up the torch. I am excited to see where they take it.

Finally, my family has been the unwavering source of my confidence that I could complete this journey. They have helped me find my way at every step, and their love and support has been my emotional anchor. I owe the best of who I am to them.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 The Shift to Parallel, Heterogeneous and Distributed Computing

Recent work has documented the shifting landscape of computer architecture over the past two decades [114, 22, 98]. Previously, vendors could rely solely on advances in out-of-order (OOO) superscalar processors and increasing clock frequency to deliver higher IPC (instructions per clock cycle). With each new generation of processors, software developers were able to leverage this performance increase to create more compelling applications without changing their programming model. Furthermore, existing applications also benefited from this performance increase with no extra effort. Although the number of transistors on chips continues to increase according to Moore's Law [81], this automatic performance improvement (also called the "free lunch" [114]) ended in the early 2000s due to the inability to efficiently deliver power to and dissipate heat from single core processors. This phenomenon is known as the "power wall" and has resulted in a leveling off of single core clock frequencies (Figure 1.1).

Since single core frequency scaling is no longer an option, processor vendors have begun utilizing the additional transistors to develop multiple, sometimes specialized, processing cores on the same chip [2]. The combination of different processors with

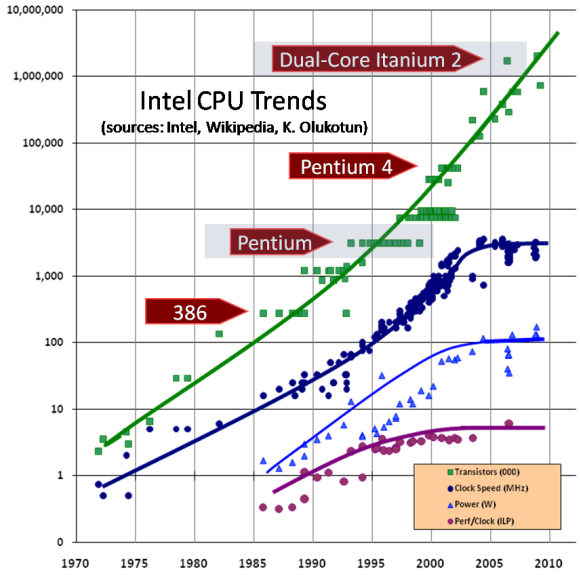


Figure 1.1: The leveling off of sequential processor speed due to the power wall. *Source:* Herb Sutter, The Free Lunch is Over: A Fundamental Turn Toward Concurrency in Software, Dr. Dobb's Journal, March 2005 [114].

different capabilities on a single platform is called a heterogeneous system and developing programs that can utilize some or all of the processors is known as heterogeneous computing. Although parallel and distributed computing have rich histories dating back several decades, this fundamental shift in commodity processors brought parallel computing from niche fields (e.g. scientific and supercomputing) to the mainstream. At the same time as commodity CPUs began shipping with multiple cores, graphical processing units (GPUs) gradually became less specialized and general-purpose GPU (GPGPU) computing emerged as an alternative to CPUs for performance-intensive tasks. The primary enabler of GPGPUs was a programming model (either nVIDIA's CUDA [85] or OpenCL [118]) that allowed software developers to program the massively multi-threaded devices using simple, device-independent abstractions.

While processors have been converging towards massively parallel devices with a large theoretical aggregate floating point operations per second (FLOPs), disk and memory storage has also rapidly become cheaper. For example, the cost of a megabyte of DRAM has fallen by 40% per year over the long term [50]. As a result, it is now possible to both store and process quantities of data at a scale previously impossible

(Google is estimated to have indexed 280 exabytes in 2009 while the size of the largest data warehouse is increasing at an annual rate of more than 170%) [98]. The "data revolution" has ushered in new paradigms and tools ([33], [4], [132]), new algorithms ([84], [134]) as well as entirely new fields (data science, large-scale machine learning). In order to effectively utilize all of the available computational power and process large amounts of data, software developers must both "scale up" on a single processor and "scale out" to a large number of processors.

However, developing scalable applications requires fundamentally different programming models than traditional sequential programming, since data and operations must be carefully partitioned and synchronized across available resources. Furthermore, programming for specialized processors usually requires using a device-specific programming model, since high-level languages are unable to target them directly. The availability of affordable, high memory machines has led to a recent resurgence in sequential programming, since many datasets can be filtered and stored in memory on a single shared-memory machine, enabling fast sequential execution. This is complementary to, and not a replacement of, the shift towards parallel, heterogeneous, and distributed computing. Therefore, there is a critical and growing need to make current and future parallel and heterogeneous systems more accessible to programmers.

1.2 Productivity vs. Performance

The shift to parallel, heterogeneous and distributed systems in mainstream computing has set up a difficult productivity-performance trade-off for software developers. Productivity subjectively captures how easy it is to use a programming language to accomplish some task, typically in terms of the provided abstractions, the readability and maintainability of the code, and the extent to which rapid prototyping and debugging is possible. High-level general purpose languages have continuously improved productivity by focusing on primitives for abstraction and composition that allow programmers to build large systems from relatively simple but versatile parts.

However, these primitives do not usually expose the structure required for high performance on parallel and heterogeneous hardware. As the abstraction gap between applications and hardware increases, compilers are becoming increasingly ineffective at automatically exploiting hardware resources to achieve the best performance.

For example, state-of-the-art solutions like OpenCL [118] are relatively close in abstraction layer to C and still require careful programming to achieve good performance on a particular device. On the other hand, dynamic languages like Python and Ruby provide succint, high-level abstractions that lead to great productivity benefits, but compile (in a standard installation) only to sequential processors and are much slower than their low-level counterparts. The primary culprit is that high-level general purpose languages lack the semantic information required to efficiently translate coarse-grained execution blocks to low-level hardware. General-purpose compiler analyses and automatic parallelization are especially difficult in the presence of unconstrained aliasing and side-effects (in particular, mutation).

Since high-level languages cannot compile efficiently to these systems, programmers are forced to optimize performance-critical sections of their code using low-level, architecture-specific programming models (e.g. OpenMP, CUDA, MPI) in a time-consuming process. Optimizing performance-critical sections in this way results in applications that are cobbled together in a piecemeal fashion. For example, an expert application developer may rewrite a performance-critical portion of her application in CUDA [85] in order to leverage a GPU, and rewrite a different portion in Hadoop [4] to scale out to a large number of cores.

Therefore, using these disparate programming models to optimize programs for performance typically incurs a great loss in productivity. A common development cycle begins with programmers writing a prototype in a high-level language that is concise and expressive. They then translate their program to low-level, high performance code in a time-consuming process. In order to get the best performance, programmers must have hardware expertise and develop a different version of the code for each target device that may be required. The resulting code is tied to architectural details (e.g. cache-line size, number of processor cores) and is harder to read, debug, maintain, or port to future platforms. Worse, there are now multiple versions

of the original application: one that is too high-level to obtain good performance, and several that are too low-level to easily understand. Keeping the different versions in sync as new features are added or bugs are fixed quickly becomes untenable.

This productivity-performance trade-off is a limiting factor in the complexity, efficiency, and scalability of software systems, particularly for those designed and implemented by non-experts. It is a problem that will only be exacerbated by future architectural changes and the proliferation of new and specialized processors. In order to alleviate the burden on application developers, we need our compilers to get smarter.

1.3 A Domain-Specific Approach

One way to get smarter compilers is to give up generality. Domain-specific languages (DSLs) are a promising means of maintaining high-level abstractions while automatically compiling to parallel, heterogeneous, and distributed hardware [21]. DSLs have traditionally been leveraged for productivity. Since a DSL is specific to a particular problem domain, the syntax and the abstractions of a well-designed DSL are a close match to the applications that users want to write. These high-level abstractions enable users to focus on ideas and algorithms rather than implementation details.

Recent results demonstrate that DSLs are also suitable for optimizing compilation. The key advantage that DSLs have over general-purpose languages is the ability to reason about data structures and operations at the level of domain abstractions (e.g. `Matrix` or `Graph` vs. `Array`). For example, a domain-specific compiler for linear algebra can reason about the program at the level of vector and matrix arithmetic, as opposed to loops, instructions and scalars. A DSL compiler can also enforce domain-specific restrictions; while general-purpose languages usually perform sophisticated analyses to apply optimizations and must be conservative, DSLs can often apply optimizations that are correct by construction. By exploiting this high-level structure, it has been shown that DSL applications can be compiled from a single source to multicore CPUs, GPUs, and even clusters [15, 35, 53]. In contrast, an optimized C++ implementation will often play numerous tricks with memory layout (e.g. byte padding), memory

reuse (e.g. blocking or manual loop fusion), and control flow (e.g. loop unrolling), even before adding threading or vectorization which obfuscate the implementation even more.

However, DSLs have two substantial drawbacks: they are usually more difficult to construct than a high-level library, and because they use specialized tool-chains, DSL applications are harder to prototype and debug. Despite the benefits of performance-oriented DSLs, the cost of developing a DSL to a sufficient degree of maturity is immense and requires expertise in multiple areas, from language design to computer architecture. In addition to the initial cost, DSL compiler developers must also continously invest new effort to target new or updated hardware. As a result, few DSLs are developed, even in domains that could potentially benefit from an optimizing DSL. Furthermore, the wide range in quality, tooling, and performance of DSLs is a substantial obstacle for DSL adoption.

Besides DSLs, there are two major high-level programming models that have had some success in parallel and heterogeneous computing. The first is to use a restricted general-purpose programming model, such as MapReduce [33, 4, 133], Pregel [77], OpenCL [118], or Intel Cilk [56]. These systems have the advantage of being suitable for many different applications within a single language, but often force programmers to express their algorithms in a model that is unnatural to the problem, thereby sacrificing some productivity. They are also usually unable to achieve the best performance across different hardware devices, because the languages are still too low-level for a single version of the source code to be optimal for different hardware. For example, it is not uncommon to have to rewrite an OpenCL kernel in slightly different ways in order to get the best performance on either a CPU or a GPU. A second approach is to simply compose applications out of libraries of high performance primitives, such as Intel MKL [58] or hand-written, hand-parallelized C++ code. While this approach has the benefit of being the easiest to integrate of any of the alternatives, composing an application out of library calls that are individually high performance does not necessarily result in a high performance application. The main issue is that each library call is a black box to an optimizing compiler; critical optimizations that improve memory locality and reuse, such as *loop fusion*, cannot be applied across

different library calls. Therefore, a pipeline of library calls will often create far more intermediate data, or perform more traversals, than a hand-optimized version that did not use a library would. DSLs are an alternative point in this trade-off space; they are initially more unfamiliar to users, but are still high-level and admit important optimizations.

There are many implementation choices for DSLs and DSL compilers. *External* DSLs are stand-alone languages with custom compilers. These DSLs have the most flexibility, since their syntax, semantics, and execution environments can be completely domain-specific. However, they require considerable effort and compiler expertise to construct. Compiler frameworks such as LLVM [69] can reduce some of the burden by providing built-in optimizations and code generators, but DSL authors must still define how to map from the high-level DSL constructs to the low-level intermediate representation (IR). On the other hand, *internal* DSLs are embedded in a host language. The most common implementation is *pure* or *shallow* embedding, where DSLs are implemented as "just a library" in a flexible host language. Purely embedded DSLs often use advanced host language features (e.g. function currying) to emulate domain-specific syntax, but are legal programs in the host language and share the host language's semantics. As a result, they are simple to construct and DSL users can reuse the host language's tool-chain (e.g. IDE). However, they are constrained to the performance and back-end targets of the host language.

1.4 Library Productivity with Compiled Performance

The goal of this work is to develop techniques and infrastructure to enable embedded DSLs to realize the productivity of libraries with the performance of compiled and hand-optimized code. We refer to *compiled* in the sense of a *DSL compiler*, rather than a host language compiler. In contrast, a purely embedded DSL implemented as a library in a general-purpose language can be viewed as a *DSL interpreter*. Another way of looking at an embedded DSL compiler is as a self-optimizing library. To more precisely define our goals, we distinguish between two types of users:

- **DSL developers** should be able to develop embedded DSLs like libraries: they

should be concise, use high-level abstraction, and be able to build upon other DSLs to modularly add features, including compiler optimizations.

- **DSL users** should be able to use and compose DSLs like libraries; as much as possible, they should be able to use high-level debuggers, IDEs, and REPLs. Furthermore, they should be able to compose DSLs like libraries to build larger applications out of multiple DSLs.

Therefore, "productivity" with embedded DSLs has two dimensions: productivity of the DSL developer and productivity of the DSL user. Performance is also cross-cutting. The DSL developer must be able to implement DSL constructs in a high performance way and be able to add domain-specific compiler optimizations. DSL users should get high performance with a single version of their source code across different hardware devices and datasets. Furthermore, they should not lose performance when composing DSLs, so being able to optimize across DSL boundaries is critical.

In this dissertation, we demonstrate that we can develop frameworks and embedded DSLs that meet most of these goals. In order to deliver highly productive high performance DSLs to end users, we describe how to embed DSL compilers in Scala using Lightweight Modular Staging (LMS) and Delite [101, 14]. LMS is a Scala metaprogramming framework that enables building an IR from Scala programs at runtime. Delite is a heterogeneous, parallel programming framework that builds on top of LMS and simplifies the process of developing high performance DSLs by providing common reusable components, such as parallel patterns and code generators. In this way, Delite attempts to strike a balance between the internal and external DSL approaches: Delite DSLs are embedded DSLs that can still be optimized and compiled to different hardware targets. We present the key pieces that enable high performance and the translation of high-level DSL programs to parallel and heterogeneous hardware. We show how the important abstractions in the Delite framework are implemented and how to construct simple DSLs and optimizations.

To validate this approach on a real-world domain we present OptiML [112], a DSL for machine learning implemented with LMS and Delite. OptiML is a mostly functional DSL that makes it simple to express both mathematics and data transformation with high-level operators on vectors and matrices. In order to recover performance, OptiML relies heavily on Delite's optimization and code generation capability to target multicore CPUs, GPUs and clusters from a single program. OptiML was the first Delite DSL and we show by way of example how it is implemented using Delite.

Next, we develop methods to compose compiled embedded DSLs [113]. We describe the essential requirements for a DSL framework and the component DSLs and show that with the right representation, DSL users can interoperate with multiple DSLs in a single program and DSL developers can construct new DSLs by extending existing DSLs. This composition does not preclude optimization; DSLs composed together share generic optimizations and we can optimize across DSL snippets in the user program. As case studies, we present four new compiled embedded DSL implementations for different domains (data querying, scientific computing, graph analysis, and collections). We show that each DSL substantially reuses common components, simplifying their development, and achieves performance competitive with or exceeding existing solutions. We also show that we can compose the DSLs together without losing performance.

Finally, we present Forge, a meta DSL for DSL development [111]. While the previous work largely achieved our performance goal, DSL users and developers were still exposed to the embedding implementation. DSL developers required expertise in advanced Scala features and DSL users had limited debugging and interactivity capabaility. Forge mitigates these issues by presenting a declarative syntax for DSL specification and then generating multiple DSL implementations from that syntax. Forge generates both a high performance Delite version of the DSL as well as a highly productive pure Scala library version of the DSL. Both versions share the same syntax using a technique called polymorphic embedding [52]. DSL users can develop and prototype their application using the library version (which includes the ability to use standard Scala IDEs and the Scala REPL) and then switch to the Delite version using a command-line flag to scale up and scale out. Since the Delite version

involves DSL compilation and the library version does not, we call the Delite version the "compiled" implementation.

We re-implement OptiML in Forge, and show that the OptiML specification resembles a Scala library implementation and is far more concise than the original implementation while achieving the same performance. This final result shows that using the techniques, frameworks, and DSLs described in this dissertation, it is possible to develop state-of-the-art compiled embedded DSLs that approximate the productivity of libraries and the performance of external compilers and hand-optimized code.

1.5 Contributions

The artifacts presented in this dissertation were developed in collaboration with the Scala LMS team at École Polytechnique Fédérale de Lausanne (EPFL) and the Delite team at Stanford University. We build off of the dissertations of Tiark Rompf [99] and Hassan Chafi [22], where LMS and Delite were first presented, respectively. Rompf describes the original design and implementation of LMS and its implications for compiler design, program generators, and embedded DSLs. Chafi argues for using performance-oriented DSLs as a "mass market" parallel programming model and presents important abstractions that a DSL framework should contain. The most important of these are *Delite ops*, which are structured parallel execution patterns that can be efficiently translated to different hardware devices.

This dissertation summarizes the most important aspects of LMS and Delite as they relate to the productivity and performance of embedded DSLs and extends the previous work where necessary (e.g. we present new parallel patterns, transformations, and implementation techniques). The core contribution of this work is to develop both the tools required to enable LMS- and Delite- based DSLs to be productive and performant and the DSL implementations themselves that validate the feasibility of the approach and the usefulness of the tools. Specifically:

- We present the design and implementation of OptiML, a new DSL designed to enable machine learning developers to easily take advantage of heterogeneous

parallelism. Without any explicit parallelization, OptiML outperforms parallel MATLAB by an average of 3.52x on 8 cores and 3.98x with a GPU. OptiML is also competitive with hand-optimized parallel C++ implementations.

- We show that DSLs for different domains (data querying, scientific computing, graph analysis, collections) can be implemented in Delite with a small number of reusable components and still achieve performance exceeding optimized libraries (up to 125x) and comparable to stand-alone DSLs (within 30%).
- We are the first to show both fine-grained and coarse-grained composition of high performance compiled DSLs. We demonstrate that different DSLs used in the same application can be co-optimized to improve performance by up to an additional 1.82x.
- We present Forge, a new meta DSL for high performance embedded DSL development. Forge provides a high-level, declarative API for specifying DSL data structures and operations in a manner to similar to an annotated high-level library. Forge makes performance-oriented DSLs simpler to develop, easier to debug, and possible to retarget to different back-ends. The OptiML specification implemented in Forge is approximately 6x shorter in lines of code than the original version. To our knowledge, Forge is the first embedded and staged meta DSL.

1.6 Outline

Chapter 2 introduces necessary background and demonstrates how one can embed a DSL compiler inside a host language (in this case Scala) using staged metaprogramming. We present in detail a basic Delite DSL implementation and describe the key principles that enable Delite DSLs to achieve high performance. Chapter 3 presents the design and implementation of OptiML, the first implicitly parallel embedded DSL implemented with Delite. Chapter 4 introduces requirements and techniques for compiled DSL composition. In this chapter, we also present case studies on four new

Delite DSL implementations (OptiQL, for data querying; OptiMesh, for scientic computing; OptiGraph, for graph analysis; and OptiCollections, for collection processing). We characterize the reuse achieved in each DSL implementation and evaluate their performance compared to alternative systems in each domain. Chapter 5 presents the design and implementation of Forge, a new meta DSL that simplifies the process of developing Delite DSLs. Chapter 6 provides a literature survey of related work in the fields of staged metaprogramming, high performance DSLs, parallel programming and language workbenches. Finally, Chapter 7 summarizes the current state of the art and discusses unsolved problems related to performance-oriented DSLs.

Chapter 2

Embedding DSL Compilers with LMS and Delite

In this chapter, we present a comprehensive overview of Delite, an open-source DSL compiler framework developed in collaboration with teams at EPFL and Stanford University [110, 14, 105]. Delite substantially reduces the burden of developing high performance DSLs by providing common reusable components that enable DSLs to quickly and efficiently target heterogeneous hardware [113]. We first describe why and how we embed DSL compilers inside the Scala host language. Then we discuss the principles of Delite's design and how Delite is used to implement embedded compilers. We conclude with a detailed discussion of Delite's facilities for performing optimizations and achieving high performance: analyses, transformations, and heterogeneous code generators.

Delite is open-source and freely available at: `http://github.com/stanford-ppl/Delite/`.

2.1 Embedding Compilers with Staging

In order to reduce the programming effort required to construct a new DSL, Delite DSLs are implemented as embedded DSLs inside Scala, a high-level hybrid object-oriented functional language that runs on the Java virtual machine (JVM) and interoperates with Java. We use Scala because its combination of features makes it well-suited for embedding DSLs [20]. Most importantly, its strong static type system and mix-in composition[1] make DSLs easier to implement, while higher-order functions, function currying and operator overloading enable the DSLs to have a flexible syntax despite being embedded. The choice of Scala as our embedding language is a convenient one, but is not fundamental to the approach.

The key difference between Delite DSLs and traditional purely embedded DSLs is that Delite DSLs construct an intermediate representation (IR) in order to perform optimizing compilation and target heterogeneous devices. Instead of running Scala application code that executes DSL operations, we will run Scala application code that *generates* code that will execute DSL operations. Thus, Scala is both the language we use to implement our DSL compilers as well as the language that serves as the front-end for the DSLs. At run-time, on the other hand, the code being executed can be Scala, C++, CUDA, or anything else. One way of viewing this approach is as a method of bringing optimizing compilation to domain-specific libraries.

We use a technique called Lightweight Modular Staging, a form of *staged metaprogramming*, to generate an IR from Scala application code [102]. Throughout this dissertation, we will use a running example of a small `SimpleVector` DSL to illustrate concepts. `SimpleVector` contains a single data type (`Vector`), and basic mathematical operators. Consider the following small example program using this DSL:

```
object SimpleVectorRunner extends SimpleVectorApplication with AddVectors
trait AddVectors extends SimpleVector {
  def main() {
    // explicit types provided for demonstration only
```

[1] Mix-in composition is a restricted form of multiple inheritance. There is a well-defined linearization order to determine how virtual method calls get dispatched within mixed-in classes. This allows functionality to be layered and overridden in a modular way.

```
    // normally, these would be omitted and type inference would be used
    val v1: Rep[Vector[Double]] = Vector.rand(1000)
    val v2: Rep[Vector[Double]] = Vector.rand(1000)
    val a = v1+v2
    println(a)
  }
}
```

For this initial example, we have elided the implementation of the DSL traits `SimpleVectorApplication` and `SimpleVector`; these will be presented in Section 2.7. For now, we focus on the application code inside the `main` method. The LMS library enables us to perform type-directed staging to defer execution of certain operations to a later time (i.e. stage). [2] LMS contrasts with traditional multi-stage programming approaches that use explicit operators (e.g. `quote`,`unquote`) to demarcate staged program fragments [116]. Staging itself is a form of directed partial evaluation; the literature on partial evaluation and staging is rich, so we refer the reader there for more detail [63, 115, 45].

The key idea behind LMS is that data types that are wrapped in an abstract type constructor, `Rep[T]`, are deferred to a later stage. Computations on types that are not wrapped in `Rep` are performed immediately when the Scala program is run. `Rep` stands for representation; a `Rep[Vector[T]]` is a representation of a `Vector[T]` that is not tied to a particular concrete implementation. Type inference is used to hide this wrapped type from application code, as indicated above. Delite DSLs typically wrap *all* DSL types in `Rep` in order to construct a complete IR for the program.

For example, we use the following Scala definition to define the `Vector.rand` statement abstractly:

```
// DSL interface, no implementation
object Vector {
  def rand(len: Rep[Int]): Rep[Vector[Double]] = vector_rand(len)
}
```

[2] *Credits*: Design by Tiark Rompf, implementation mostly by Tiark Rompf with help from the author and others in the LMS/Delite teams.

```
def vector_rand(len: Rep[Int]): Rep[Vector[Double]]
```

The DSL program, defined in the scope `SimpleVector`, sees only these abstract definitions. Therefore the DSL application code is well-typed, but the references being passed in the Scala code are simply placeholders for results to be computed at a later stage. It is the `SimpleVectorApplication` trait that provides concrete implementations of these abstract DSL methods and therefore forms a complete program that can be run. It is important to note that *run* in this context means to execute the Scala program which will call the DSL methods, which can in principle do anything. Abstracting over the type representation in this way is called *polymorphic embedding* [52]. However, if these methods construct an IR (as in an embedded compiler), then running the "program" actually runs the DSL compiler, which generates code to be executed later. Therefore, LMS performs run-time code generation. One important aspect of this design is that it not only provides flexibility by cleanly separating interface from implementation, it also provides important safety guarantees: the user program can never introspect on compiler types.

To construct an IR, we must first define our basic building blocks: expressions and definitions. We present in Listing 2.1 a very simplified definition of such a `Base` trait; see Rompf [99] for a more detailed treatment. The `Base` trait defines the abstract `Rep` type while the `BaseExp` trait sets `Rep[T] = Exp[T]`, where `Exp[T]` represents an expression in our IR, which can be either constants (`Const`) or symbols (`Sym[T]`). Symbols are unique identifiers for definitions (`Def[T]`) which represent nodes in our IR. For example, we might define the IR node for `Vector.rand` as follows:

```
case class VectorRand(len: Exp[Int]) extends Def[Vector[Double]]
```

With these definitions at hand, implementing the method `vector_rand` to construct the IR node is simple:

```
// toAtom called explicitly for illustration; normally invoked implicitly
def vector_rand(x: Exp[Int]) = toAtom(VectorRand(len))
```

Note that since we are inside the implementation trait (`SimpleVectorApplication`), the `len` parameter is more specifically an `Exp[Int]`. We simply construct an instance of the case class, which is the IR node, and then call `toAtom` to map this definition to a

```
1  // abstract interface exposed to DSL users
2  trait Base {
3    type Rep[+T]
4    implicit def unit[T](x: T): Rep[T]
5  }
6
7  // using staging to build an IR
8  trait BaseExp extends Base {
9    abstract class Exp[+T:Manifest]
10   abstract class Def[+T]
11   type Rep[T] = Exp[T]
12
13   case class Const[+T:Manifest](x: T) extends Exp[T]
14   case class Sym[+T:Manifest](id: Int) extends Exp[T]
15
16   implicit def unit[T](x: T) = Const(x)
17   implicit def toAtom(d: Def[T]): Exp[T] =
18     findOrCreateDefinition(d) // elided
19 }
20
21 // defines how to generate target (low-level) Scala code
22 trait ScalaCodegen {
23   val IR: BaseExp; import IR._
24
25   // constructs a program schedule of the IR by traversing
26   // dependencies backwards, and calls emitNode in order
27   def emitBlock(b: Exp[T]): Unit
28
29   // implements code generation for individual IR nodes
30   def emitNode(sym: Sym[T], rhs: Def[T]): Unit = rhs match {
31     // match on node types and emit code
32   }
33 }
```

Listing 2.1: Simplified core of LMS

symbol which has type `Rep[Vector[Double]]`. `toAtom` is where LMS performs common subexpression elimination. IR nodes that are functionally pure and have identical inputs can be mapped to the same symbol using a lookup table.

The process of constructing IR nodes from application operations is called *lifting*. To ensure that all host language operations can be intercepted and lifted, we use a modified version of the Scala compiler, *Scala-Virtualized* [82], that enables overloading even built-in Scala constructs such as `IfThenElse`. Delite also provides lifted versions of many Scala library operations (e.g. string methods) with code generators for multiple targets; these can be inherited by DSLs for free.

IR nodes implicitly encode dependencies as inputs. For example, the `VectorPlus` node constructed from the `v1+v2` statement would have two inputs, `v1` and `v2`, that each refer to a different `VectorRand` node. Therefore, we can obtain the graph of the IR for any result by following dependencies backwards. We can then perform optimizations on the graph and generate target code for each IR node. This is what the `emitBlock` method does when called on a particular symbol. [3] Finally, the generated code is compiled and executed in a separate step to compute the program result. In summary, Delite DSLs typically undergo three separate stages of compilation:

1. The DSL application (Scala code) is compiled using *scalac* to generate Java byte code

2. The Java byte code is executed (staged) to run the DSL compiler in order to build the IR, perform optimizations and generate code

3. The resulting generated code is compiled for each target (e.g. C++, CUDA)

In return for more compilation, we have two primary benefits: embedded DSL compilers are simpler to build than stand-alone DSL compilers, and DSL programs can retain high-level abstractions without suffering a run-time penalty, since the generated code is low-level and first-order. In particular, although we typically lift all DSL operations, Scala abstractions like classes, traits, method calls, and closures are

[3] Some important details, such as how to handle effects, have been omitted from this presentation for the sake of clarity. See Rompf [99] for details.

programmatically removed by staging (methods and closures are inlined by expanding them into their IR). As we will discuss and demonstrate later, this can have significant performance benefits compared to ordinary Scala library code.

2.2 The Delite Framework: Overview

All Delite DSLs share the same architecture for building an intermediate representation, traversing it, and generating code. The Delite framework was first discussed in detail by [14]; it has since been enhanced to support data structures and transformations using the techniques outlined by [103]. In this section we present an overview of the Delite framework and describe its key principles.

Several requirements led to design choices that make the Delite compiler architecture different from traditional compilers. These include the need to support a large number of IR node types for domain operations, to specify optimizations in a modular way, and to generate code for a variety of different targets. Delite's architecture meets these requirements by organizing DSLs around common design principles. These principles are implemented with a set of reusable components: common IR nodes and data structures, built-in optimizations, traversals, transformers, and code generators (Figure 2.1). DSLs are developed by extending these reusable components with domain-specific semantics. Furthermore, Delite is modular; any service it provides can be overridden by a particular DSL with a more customized implementation. The remainder of this section discusses Delite's principles and the components that implement them.

Common IR The Delite IR uses the LMS sea of nodes representation [89], rather than a traditional control-flow graph (CFG); nodes explicitly encode their data and control dependencies as inputs, but are otherwise free to "float". This representation is well suited for parallelism, since it does not maintain any extra constraints on program order (e.g. syntactic) than are necessary for correctness. Additionally, Delite lifts data structures into the IR by requiring the DSL developers declare their data structures as Delite *structs*, which are limited to a set of primitive field types. Delite

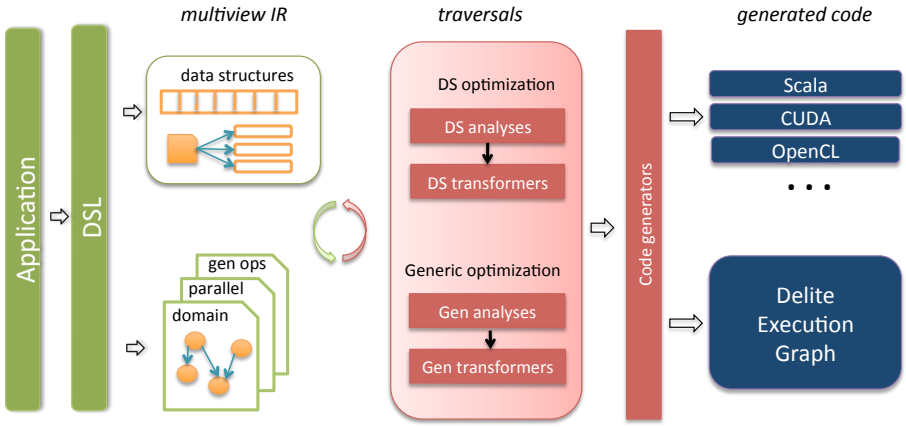


Figure 2.1: Components of the Delite Framework. An application is written in a DSL, which is composed of data structures and structured computations represented as a multi-view IR. The IR is transformed by iterating over a set of traversals for both generic (Gen) and domain-specific (DS) optimizations. Once the IR is optimized, heterogeneous code generators emit specialized data structures and ops for each target along with the Delite Execution Graph (DEG) that encodes dependencies between computations.

provides parallel patterns as built-in IR node types that DSLs extend with domain-specific information. For example, a vector addition node can simultaneously be viewed in three ways in Delite: a generic layer, `Def[T]` from LMS, a parallel layer, `DeliteOpZipWith`, and a domain-specific layer, which is the actual operation, `VectorPlus`. The IR can be optimized by viewing nodes at any layer, enabling both generic and domain-specific optimizations to be expressed on the same IR.

Explicit DSL semantics DSL authors communicate DSL semantics to Delite via a set of simple APIs. Effectful operations are marked using the `reflectEffect` or `reflectWrite` methods when constructing an IR node. Delite uses this information to honor read-after-write, write-after-read and write-after-write dependencies when constructing the program schedule. DSL authors can also optionally specify additional semantic information about particular IR nodes that enables Delite to be more aggressive with optimizations. For example, a DSL author can specify whether an input symbol is likely to be accessed more or less often during an operation using the `freqSyms` method, which allows the code motion algorithm to better determine whether to hoist a computation out, or push a computation into, a candidate block.

Similarly, the `aliasSyms` method allows DSL authors to declare if an operation constructs an alias to one of its inputs (e.g. by storing a reference to it), which can enable Delite to be less conservative when performing optimizations. The key idea behind all of these mechanisms is that by communicating a minimal amount of domain-specific information, we can often efficiently parallelize and optimize DSL programs without complicated generic analyses.

Generic optimizations Delite performs several optimizations for all DSLs (these optimizations can be programmatically disabled by a DSL if desired). At the generic IR layer, Delite performs common subexpression elimination (CSE), dead code elimination (DCE), and code motion. Although these are traditional optimizations, it is important to note that in the Delite framework they occur on symbols that represent coarse-grained domain-specific operations (such as `MatrixMultiply`), rather than generic language statements as in a typical general-purpose compiler. Therefore, the potential impact of these optimizations in a program is substantially greater. At the Delite op layer, Delite implements *op fusion* [103]; data parallel ops that have a producer-consumer or sibling relationship are fused together, eliminating temporary allocations and extra memory accesses. Finally, as mentioned previously, Delite performs AoS→SoA conversion and dead field elimination for data structures, which is essential to increasing the number and performance of ops that can be run on wide, column-oriented devices such as GPUs.

Domain-specific optimizations Domain-specific optimizations can be implemented as rewrite rules using pattern matching on IR nodes; we can layer these rewritings in a modular way by putting them in separate traits[4].

More sophisticated analyses or optimizations are possible by implementing `Traversals` and `Transformers`. Traversals schedule the IR and visit each node in order, allowing the DSL author to define an arbitrary operation (such as collecting statistics for an analysis) at each node. Transformers combine traversals with rewrite rules, and can be run as phases in a predetermined order. Transformers are ideal for performing

[4]A trait is a type of Scala class that supports mix-in composition.

lowering optimizations, i.e. translating a domain-specific operation to a different, lower-level representation that can be optimized further. One example is performing device-specific optimizations: we can define a transformer that will transform nested parallel loops into a flattened parallel loop for highly multi-threaded devices. Transformers are also useful for performing domain-specific rewrites on the IR that require the entire IR to be constructed first. Section 2.4 describes Delite's analysis and transformation facilities in detail.

Code generation The final stage of a Delite DSL is code generation. A Delite code generator is simply a traversal that performs code generation, i.e. emits code for a particular platform. Delite provides code generators for a range of targets, which currently includes Scala, C++, CUDA and OpenCL. The result of code generation for a Delite DSL is two major artifacts: the Delite Execution Graph (DEG), a data-flow representation of the operations in the program; and kernels, code that implements particular ops for different platforms. The DEG describes an op's inputs, outputs, and what platforms it can be executed on. This representation enables Delite to also exploit task parallelism in an application by running independent ops in parallel. Finally, the Delite runtime schedules and executes the kernels on the available hardware resources.

2.3 The Delite IR

The Delite IR provides a way for the framework to reason about both computation and data structures at a high level. Computation is represented by extending a generic parallel pattern layer, called *Delite ops*, that Delite DSLs use in order to inherit parallel optimization and code generation. Data structures are restricted to high performance primitives and must be declared using a simple Delite API, which enables Delite to reason about and perform optimizations on them.

2.3.1 Parallel Patterns

Delite ops represent reusable parallel patterns such as `Reduce`. Delite ops have op-specific semantic restrictions (such as requiring disjoint access) that must be upheld by the DSL author. The ops operate on `DeliteCollections`, an interface implemented by DSL data structures to provide access to the underlying elements. DSL authors are responsible for mapping domain operations to Delite ops (e.g. a domain operation `VectorPlus` to the Delite op `ZipWith`); Delite provides code generators from Delite ops to multiple platforms. This division allows domain experts to focus on identifying parallel patterns within the domain (domain decomposition) and concurrency experts to focus on implementing the parallel patterns as efficiently as possible on each platform.

Delite currently supports `Sequential`, `Map`, `Reduce`, `ZipWith`, `Foreach`, `Filter`, `GroupBy`, `Sort`, `ForeachReduce`, and `FlatMap` ops. The semantics of the ops are as follows[5]:

- **Sequential**: The input to `Sequential` is an arbitrary function, `func: => R`. `Sequential` executes the function as a single task on a general-purpose CPU core. It is the primary facility by which DSL authors can control the granularity of task parallelism of sequential DSL operations, since all `Sequential` ops are guaranteed not to be split across multiple resources. The result of `Sequential` is simply the result `R` returned by the function.

- **Map**: The inputs to `Map` are a collection `DeliteCollection[A]` and a mapping function `map: A => R`. `Map` processes the input collection in parallel, applying the `map` function to every element to produce the output collection, a `DeliteCollection[R]`.

- **Reduce**: The inputs to `Reduce` are a collection `DeliteCollection[A]`, a reduction function `reduce: (A,A) => A`, and a zero element `A`. The `reduce` function must be associative. `Reduce` performs a parallel tree-reduction on the input collection,

[5]Ops that return collections also require an `alloc` function to instantiate a new `DeliteCollection` of the appropriate type.

optimized for each device. The result of `Reduce` is a single element `A`.

- **ZipWith**: The inputs to `ZipWith` are two collections, `DeliteCollection[A]` and `DeliteCollection[B]`, and a mapping function `zip: (A,B) => R`. `ZipWith` processes both collections in parallel, applying the `zip` function to pairwise elements to produce the output collection, a `DeliteCollection[R]`.

- **Foreach**: The inputs to `Foreach` are a collection `DeliteCollection[A]` and a side-effectful function `func: A => Unit`. `Foreach` is implemented by processing the collection in parallel and invoking `func` on each element. The DSL author must exercise caution when using `Foreach`; the supplied `func` should be safe to execute in any order and in parallel (e.g. disjoint writes to an output collection). `Foreach` has no output.

- **Filter**: The inputs to `Filter` are a collection `DeliteCollection[A]`, a predicate function `pred: A => Boolean`, and a mapping function `map: A => R`. `Filter` is implemented as a parallel scan. The input collection is processed in parallel, and `map` is applied to each element for which `pred` returns `true`. The result of the `map` is appended to a thread-local buffer. Finally, the output size and starting offset of each thread are computed and each thread then writes its portion of the result to the output collection in parallel. The result of a `Filter` is a `DeliteCollection[R]` of size $<=$ the original `DeliteCollection[A]`.

- **GroupBy**: The inputs to `GroupBy` are a collection `DeliteCollection[A]` and two functions `keyFunc: A => K` and `valFunc: A => R`. An optional third input is a function `reduce: (R,R) => R`. `GroupBy` processes the input collection in parallel, applying `keyFunc` to each element of the input to determine a bucket index and `valFunc` to determine the value appended to the bucket. `GroupBy` can either *collect* or *reduce* elements; the result of a collect is a `Map[K,DeliteCollection[R]]`.

The result of a reduce is a `Map[K,R]` obtained from applying `reduce` to all the elements in a bucket.

- **Sort**: The inputs to `Sort` are a collection, `DeliteCollection[A]` and a comparator function `comp: (A,A) => Int`. The implementation of `Sort` is platform-specific; Delite will normally generate a library call to an optimized sort for a particular backend, specialized for primitive types. The output of `Sort` is a `DeliteCollection[A]`.

- **ForeachReduce**: (*foreach with global reductions*) The inputs to `ForeachReduce` are the same as `Foreach` (a collection and function block), but the `ForeachReduce` function may contain instances of `DeliteReduction`, an interface for binary reduction ops (e.g. `+=`). `DeliteReduction` requires an input variable, `in: Var[A]`, a right-hand side expression `rhs: A`, and a reduction function, `reduce: (A,A) => A`. `ForeachReduce` is implemented by examining the function body. If the body contains reductions, we construct a composite op containing the original `Foreach` op without reductions and one or more new `Reduce` ops. `ForeachReduce` executes the composite op as a single parallel loop; reductions are stored in a buffer and written to their corresponding input variable only after the loop completes. If there are no reductions, we construct a normal `Foreach`. Like `Foreach`, the output of `ForeachReduce` is `Unit` (i.e. `void`).

- **FlatMap**: (*map followed by flatten*) The inputs to `FlatMap` are a collection `DeliteCollection[A]` and a function `mapF: A => DeliteCollection[R]`. `FlatMap` processes the elements of the input collection in parallel and applies `mapF` to each element to create a new collection per element. The resulting collections are then concatenated by writing their elements into the output `DeliteCollection[R]` in parallel. Since the size of the output collection is not known beforehand, `FlatMap` computes the output size and starting offset of each thread using a parallel scan.

Since new ops may be required for new DSLs, Delite is designed to make adding new ops relatively easy. Most of the existing Delite ops extend a common loop-based abstraction, `MultiLoop` [6]. `MultiLoop` is a highly generic data parallel loop; the loop body consists of one or more *elems*. An *elem* can be a `collect`, `reduce`, `bucket`, or a `foreach`, corresponding to appending elements to an output collection, reducing elements into an accumulator, grouping elements into multiple collections, or applying a side-effectful function, respectively. Thus, `MultiLoop` generalizes most of the Delite ops described above.

To formalize this notion [7], we use the following notation [103]:

`Generator kinds:` $\mathcal{G}$ ::= `Collect` | `Reduce(`$\oplus$`)` | `Bucket(`$\mathcal{G}$`)`

`Yield statement: xs` $\leftarrow$ `x`

`Contexts:` $E[.]$ ::= `loops and conditionals`

The core loop abstraction is

`loop(s)` $\overline{\texttt{x} = \mathcal{G}}$ `{ i =>` $\overline{E[\texttt{x} \leftarrow \texttt{f(i)}]}$ `}`

where `s` is the size of the loop and `i` the loop variable ranging over $[0, \texttt{s})$. We say that a loop can compute multiple results $\overline{\texttt{x}}$, each of which is associated with a generator $\mathcal{G}$. The generators correspond to the functional loop elems described previously. `Collect` creates a flat array-like data structure, `Reduce(`$\oplus$`)` reduces values with the associative operation $\oplus$ and `Bucket(`$\mathcal{G}$`)` creates a nested data structure, grouping generated values by key and applying $\mathcal{G}$ to those with matching key. Loop bodies consist of *yield* statements `x`$\leftarrow$`f(i)` that pass values to generators (of this loop or an outer loop), embedded in some outer context $E[.]$ that might consist of other loops or conditionals. Note that a yield statement `x`$\leftarrow$`..` does not introduce a binding for `x`, but passes a value to the generator identified by `x`. For `Bucket` generators, yield takes (key,value) pairs.

Therefore, the signature for each generator is defined as follows:

```
G ::= Collect_s(c)(f)          : Coll[V]
    | Reduce_s(c)(f)(r)        : V
```

[6] *Credits:* Design by Tiark Rompf. Implementation by Tiark Rompf with help from Kevin Brown, HyoukJoong Lee, and the author.

[7] *Credits:* Presentation by Tiark Rompf

```
| BucketCollect_s(c)(k)(f)   : Coll[Coll[V]]
| BucketReduce_s(c)(k)(f)(r) : Coll[V]

c:  Index => Boolean     condition
k:  Index => K           key function
f:  Index => V           value function
r:  (V,V) => V           reduction
s:  Int                  loop size
```

This model is expressive enough to represent many common collection operations:

```
x=v.map(f)     loop(v.size) x=Collect   { i => x ← f(v(i)) }
x=v.sum        loop(v.size) x=Reduce(+) { i => x ← v(i) }
x=v.filter(p)  loop(v.size) x=Collect   { i => if (p(v(i))) x ← v(i) }
x=v.flatMap(f) loop(v.size) x=Collect   { i => val w = f(v(i))
                              loop(w.size) { j => x ← w(j) }}
x=v.distinct   loop(v.size) x=Bucket(Reduce(rhs)) { i =>  x ← (v(i), v(i)) }
```

We see that `MultiLoop` is a unifying abstraction that allows us to reason about many different kinds of loop bodies under a common framework. For example, it is the abstraction that enables Delite's fusion algorithm to combine multiple ops into a single op. It also allows us to implement the code generation for most ops once (via the code generation for `MultiLoop`), instead of for each op individually. To add a new op to Delite, the key work that must be done is to either express it in terms of existing ops, or to define the code generation of the new op for each target platform. Once an op is added to Delite, all DSLs can take advantage of it. As more DSLs are implemented with Delite, the coverage of ops increases and the likelihood of a new op being necessary decreases.

2.3.2 Data Structures

Delite DSLs present domain-specific types to end users (e.g. `Vector`) and methods on instances of those types get lifted into the IR. However, DSL back-end data structures are restricted to Delite `Structs`, C-like structs with a limited set of field types (primitives). The fields currently allowed in a Delite `Struct` are: numerics (e.g. `Int`),

`Boolean`, `String`, `Array`, and `Map`. `Structs` are instantiated by passing Delite a list of fields, represented as a hashmap from field name to field type. We use the following simple interface to represent structs: [8]

```
// generic struct interface
trait StructExp extends BaseExp {
  abstract class StructTag
  case class Struct[T](tag: StructTag, elems: Map[String,Rep[Any]]) extends Def[T]
  case class Field[T](struct: Rep[Any], key: String) extends Def[T]
  def struct[T](tag: StructTag, elems: Map[String,Rep[Any]]) = Struct(tag, elems)
  def field[T](struct: Rep[Any], key: String): Rep[T] = struct match {
    case Def(Struct(tag, elems)) => elems(key).asInstanceOf[Rep[T]]
    case _ => Field[T](struct, key)
  }
}
```

A new `Struct` is instantiated by calling the `struct` method, which constructs an IR node representing the struct and stores the map between its field names and (symbolic) values. The type of the field is stored with its symbol. Thus, data structures themselves are part of the IR. By restricting the content of data structures and lifting them into the IR, Delite is able to perform optimizations such as array-of-struct (AoS) to struct-of-array (SoA) conversion and dead field elimination automatically [103].

For example, in order to perform struct unwrapping, we only need to rewrite field accesses to the struct to forward to the underlying field. If all such field accesses can be forwarded (i.e. the struct is not mutable and there are no effects), then no symbol is a dependency of the IR node representing the struct, and the struct wrapper itself will be DCE'd. We can implement this forwarding rewrite simply:

```
override def field[T](struct: Exp[Any], index: String): Exp[T] =
  val field = struct match {
    case Def(Struct(tag, elems)) => elems.find(_._1 == index)
    case _ => None
```

[8] *Credits:* Design and implementation by Tiark Rompf with help from Kevin Brown and the author.

```
    }
    // return short-circuited field, or fall-back
    // to constructing a field access IR node
    field.getOrElse(super.field[T](struct, index))
  }
```

Furthermore, since the set of primitives is fixed, Delite can implement these primitives on each target platform (e.g. C++, CUDA) and automatically generate code for DSL structs. This also means that Delite is can uniformly handle serializing and/or copying structs across address spaces in the format the target platform expects, which is a difficult problem with unrestricted data structures.

Delite also supports user-defined data structures by lifting the **new** keyword defining an anonymous class [82]. An application developer can write code like the following:

```
val foo = new Record {
  val x = "bar"
  val y = 42
}
```

`Record` is a built-in type provided by Delite that serves as a tag for the Scala-Virtualized compiler [100]. The Scala-virtualized compiler forwards any invocation of **new** with a `Record` type to Delite, which will then call the `struct` method to construct a corresponding `Struct`. Field accesses on the record are type-checked by the Scala-virtualized compiler and then forwarded to Delite as well.

2.4 Analysis and Transformation

As discussed in the overview in Section 2.2, Delite provides both built-in optimizations and facilities for DSL authors to define their own domain-specific analyses and optimizations. In both cases, the analyses and optimizations operate on an IR structured in the LMS way (a "sea-of-nodes"). LMS' architecture makes certain kinds of optimizations extremely simple. For example, since all DSL operations are contained in modular traits, and IR node construction is a simple method call at stage-time, we can implement pattern rewriting simply by overriding the appropriate method

call. To implement the simple optimization to eliminate addition with zeros in our SimpleVector DSL example, we need only define the following trait:

```
trait VectorMathOpsExpOpt extends VectorMathOps {
  override def vec_plus[T](lhs: Exp[Vector[T], rhs: Exp[Vector[T]) =
    (lhs,rhs) match {
      case Def(VecZeros(n), b) => b
      case Def(a, VecZeros(n)) => a
      case _ => super.vec_plus(lhs,rhs)
    }
}
```

Here, we assume a VecZeros analog for the VecRand node we defined in the previous example. This rewrite rule checks the symbolic values of the arguments to vec_plus before constructing an IR node; if either the lhs or rhs are a zero vector, we simply skip the addition. Otherwise, we call **super** to construct the VecPlus node like normal. These so-called "smart constructors" are a modular and composable way of layering pattern rewriting optimizations.

However, we sometimes want to perform whole-program static analyses that go beyond local rewriting rules. Ideally, we also want to maintain expressiveness and simplicity; that is, we want to express the result of a transformation as ordinary code, rather than explicitly constructing new IR trees to replace the old one. One of the major insights of our embedded compiler approach is that we can use staging not only to simplify program generation, but also to simplify program transformation by implementing internal compiler passes as staged interpreters [103]. The main idea is that we can define a custom transformer as a traversal of the IR we constructed during staging, and use the same kind of pattern-matching as smart constructors to match on blocks that we want to transform. When we find a match, we can express the result of the transformation as a staged code fragment (just like the original DSL code) and then construct subsequent nodes to refer to our new fragment instead of the old one. Because of our heavy reliance on our existing technology (LMS), we need relatively little new machinery to define transformers, and transformations themselves can also be defined succinctly.

```
34 // traversal
35 trait ForwardTraversal {
36   val IR: Expressions; import IR._
37   def traverseBlock[T](b: Block[T]): Unit =
38     focusExactScope(block) { stms =>
39       stms foreach traverseStm }
40   def traverseStm[T](s: Stm[T]): Unit =
41     blocks(stm.rhs) foreach traverseBlock
42 }
43
44 // transform
45 trait ForwardTransformer extends ForwardTraversal {
46   val IR: Expressions; import IR._
47   var subst: Map[Exp[_],Exp[_]]
48   def transformExp[T](s: Exp[T]): Exp[T] = // lookup s in subst
49   def transformBlock[T](b: Block[T]): Exp[T] = scopeSubst {
50     traverseBlock(b); transformExp(b.res) }
51   def transformDef[T](d: Def[T]): Exp[T] = mirror(d, this)
52   override def traverseStm(s: Stm[T]) = {
53     val e = transformDef(s.rhs); subst += (s.sym -> e); e }
54 }
```

Listing 2.2: Traversal and Transformation Interface

Listing 2.2 shows the core of our traversal and transformation implementation [9]. To implement a custom transformer, we simply extend `ForwardTransformer` and override `transformStm` appropriately. The base `Transformer` needs a default case, which we call *mirroring*. The DSL author defines the `mirror` method on every DSL IR node to construct a transformed copy by invoking the IR node's smart constructor on the arguments that result from the substitution map. This is the mechanism that enables subsequent (mirrored) nodes to refer to a transformed IR snippet. By using smart constructors within the transformation, we also allow transformations and rewrites

[9] *Credits:* Design and implementation by Tiark Rompf with help from the author and other members of the LMS/Delite teams.

Algorithm 1 Pseudocode for Stencil Analysis.

```
1: AccessPatterns: A ::= Const | One | Interval | Unknown
2: Contexts: E[.] ::= loops and conditionals
3: allStencils = new Map(loopId, Map(arrayId,accessPattern))
4:
5: procedure TRAVERSE(node)
6:     # id,i,len,block parameterize loop
7:     if node == loop(id,i,len,block) then
8:         process(id,i,block)
9:
10: procedure PROCESS(loopId,i,block)
11:     stencil = new Map(arrayId,accessPattern)
12:     # a(i) is an access of array a at index i  i
13:     if block == E[ a(const) ] then
14:         stencil ∪= a → Const(const)
15:     if block == E[ a(i) ] then
16:         stencil ∪= a → One
17:     if block == E1[ loop(_,j,len,E2[ a(i*s1+j*s2) ]) ] then
18:         stencil ∪= a → Interval(i*s1,s2,i*s1+len*s2)
19:     if block == E[ a(x) ] and no other match for a(x) then
20:         stencil ∪= a → Unknown
21:     allStencils(loopId) = stencil
```

to interleave, combining optimizations into a single pass and avoiding many phase ordering issues.

In order to make this more concrete, we now show how one can implement a *stencil analysis* using this interface. The goal of the analysis is to determine the access patterns of Delite ops in order to inform future scheduling and partitioning decisions. The result of the analysis is a map from ops to *stencils*, where a *stencil* is a map from `DeliteArrays` to *access patterns*. Algorithm 1 shows the pseudocode, while Listings 2.3 and 2.4 show the implementation. We consider a simple set of access patterns that are straightforward to detect once we have lowered our program representation to loops over arrays (via struct unwrapping). `Const(c)` is an access of the input array at the constant index `c`. `One` denotes that an op with loop index `i` accesses only the `ith` element of the input array. `Interval` represents a strided (stride can be 1) access of some portion of an array. If the size of the `Interval` equals the

```
55 trait StencilExp extends LoopsExp {
56   abstract class AccessPattern
57   object unknown extends AccessPattern { override def toString = "unknown" }
58   object one extends AccessPattern { override def toString = "one" }
59   case class Interval(offsetMultiplier: Exp[Int], stride: Exp[Int], length: Exp[Int])
60     extends AccessPattern
61   case class Constant(index: Exp[Int]) extends AccessPattern
62
63   // from a data symbol (e.g. DeliteArray) to its access pattern
64   type Stencil = HashMap[Exp[Any],AccessPattern]
65 }
```

Listing 2.3: Stencil Analysis IR Definitions

length of the array, then the entire array is accessed at every iteration of the op and must be replicated across nodes. Finally, `Unknown` represents an access to the array at some unknown element `x`, which may be a (possibly complicated) function of the loop body. In the pseudocode, the operator $\cup=$ adds new accesses to the stencil; if an access has already been recorded for a particular array, we store the conservative join of the two accesses. For example, if we see a `One` access followed by an `Interval` access, we record `Interval`.

```
66 trait StencilAnalysis extends ForwardTraversal {
67   val IR: DeliteOpsExp
68   import IR._
69
70   // from a loop to its stencil
71   var loopStencils = new HashMap[Exp[Any],Stencil]()
72   def addStencil(s: Exp[Any], stencil: Stencil) {
73     if (loopStencils.contains(s)) {
74       loopStencils(s) = loopStencils(s) ++ stencil
75     }
76     else {
77       loopStencils += s -> stencil
78     }
79   }
80   def run[A](b: Block[A]) = {
81     // run traversal
82     traverseBlock(b)
83     // return found stencils
84     loopStencils
85   }
86
87   /**
88    * Determine if the index x corresponds to a chunk of the input, starting from ref
89    * Uses pattern matching to detect array accesses of the form a*b + c*d
90    * Returns start, stride, and size, if found
91    */
92    def chunked(ref: Sym[Int], x: Exp[Int], context: List[Stm])
93      : Option[(Exp[Int],Exp[Int],Exp[Int])]
94
95   /**
96    * For a given IR node, determine any constraints it imposes with respect to
97    * the loop variable v.
98    */
```

```
99   def examine(x: Stm, v: Sym[Int], stencil: Stencil, context: List[Stm]) {
100    def processArrayAccess(a: Exp[DeliteArray[Any]], i: Exp[Int]) {
101      lazy val interval = chunked(v,i,context)
102      if (v == i) {
103        // access at loop index v
104        stencil += a -> one
105      }
106      // else... (other stencil cases)
107    }
108
109    x.rhs match {
110      case DeliteArrayApply(a,i) => processArrayAccess(a,i)
111      case _ =>
112    }
113  }
114
115  /**
116   * For the given loop element, determine all index constraints for all
117   * DeliteCollections used in its body.
118   */
119  def process[A](s: Sym[A], v: Sym[Int], body: Def[_]) {
120    body match {
121      case DeliteForeachElem(func) =>
122        val stencil = new Stencil()
123        val schedule = buildScheduleForResult(func)
124        schedule.foreach(examine(_,v,stencil,schedule))
125        addStencil(s, stencil)
126
127      // similar cases for Collect, Reduce, HashCollect, HashReduce
128      case _ => // anything else, skip
129    }
130  }
131
```

```
132   // callback for traversal: our atoms are top-level loops
133   override def traverseStm(stm: Stm): Unit = stm match {
134     case TP(s,l:DeliteOpLoop[_]) =>
135       process(s,l.v,l.body)
136     case _ =>
137       super.traverseStm(stm)
138   }
139 }
```

Listing 2.4: Stencil Analysis Implementation

First, we define the basic stencil data structures we need (Listing 2.3). The analysis is started by calling `run` on a `Block` (line 80), which is just a wrapper around a symbol that was returned by expanding an expression tree. `run` kicks off the traversal with `traverseBlock`, which calls `traverseStm` for every statement in the block. We override `traverseStm` (line 133) to match on loops, and for every loop, we call `process` to add the stencil for the loop to our global data structure. `process` uses the helper methods `examine` and `chunked` to construct the appropriate stencil by pattern matching on array accesses. Note that since this is only an analysis, we do not produce any staged code like we would in a transformation. In Chapter 4, we will show an example of a domain-specific transformer using this interface. One of the interesting aspects of Delite traversals and transformations is how they interact to enable each other. In this example, the stencil analysis is simplified because symbolic patterns like `0*i` or `0+i` have already been rewritten to `0` or `i` respectively, so there are less cases to consider. Similarly, struct unwrapping allows us to view the program directly in terms of accesses on `DeliteArray`, as opposed to on some domain-specific data structure. We view this ability to compose optimizations a critical feature of an extensible compiler, since it allows us to define optimizations modularly and elegantly layer domain-specific optimizations on top of general-purpose ones.

Discussion We conclude this section with a short discussion of the insights from Delite's approach to analysis and transformation. Although we have not discussed it in detail, Delite in general allows mutable operations. DSL authors must explicitly

mark mutable and effectful operations as such when constructing IR nodes (using a simple API, e.g. `reflectMutable(..)`). Effectful nodes carry around an *effect context*, which ensures that they will be scheduled in the proper order w.r.t. other effects [10]. However, as has been widely recognized for a long time, effectful operations are much more difficult to reason about. In particular, the effectful nodes are obscured in the IR, occluding the view of the actual operation. An analysis or transformation must take extra care to respect the effect context and therefore are much harder to prove correct or even debug. Therefore many optimizations in Delite simply do not work in the presence of mutation. In order to get the best performance, DSL authors must carefully balance when (or if) to allow mutable operations, and if they do allow them, they should limit and/or encapsulate them as much as possible.

Finally, we have not gone into much detail about the optimizations that Delite performs generically for all DSLs. The most important of these are struct unwrapping, array of struct to struct of array conversion (AoS $\rightarrow$ SoA), and op fusion. Rompf et. al. [103] provides detailed implementation details for the generic optimizations, including a formal description of the fusion algorithm. In our experience these optimizations, in particular fusion, are critical to achieving high performance on a wide range of DSLs. This point actually goes hand-in-hand with mutability; the less mutable a DSL interface is, the more important fusion is. Immutable semantics usually require naïvely constructing a new data structure to represent a transformed collection, but fusion eliminates these intermediate data structures when pipelining operations together. In many cases, we can start with a high-level, immutable interface that is easy to reason about and parallelize, but generate low-level, mutable, imperative code that performs as well as a hand-written C++ implementation.

2.5 Generating Efficient Code

Using staging to embed DSL compilers makes generating efficient code easier. We inherit high-level abstractions like types, methods, closures, pattern matching and traits, that can be used in DSL programs but "staged away" during DSL compilation.

[10] For more detail about our effects implementation, please see Tiark Rompf's thesis [99]

To stage away an abstraction, we simply do not lift it into the DSL IR. This means, for example, that a Scala method can still be used to organize code in the DSL application, but the method call is not lifted, so the body of the method is inlined. Similarly, Scala closures get expanded into expression trees when called. By choosing which Scala operators to overload and/or which types to lift into `Rep`, we have a fine-grained, programmatic way of removing abstractions before they reach the IR and can therefore the generated code.

This principle, which we and others have called "abstraction without regret", is a primary tenet of Delite. We want to provide expressive, high-level abstractions at compile time, but generate low-level, first-order code to be executed at run-time. In Delite, we generate code in an SSA-like form. The following example comes from the k-means clustering application common in machine learning:

```
def apply(x388:Int,x423:Int,x389:Int,
  x419:Array[Double],x431:Int,
  x433:Array[Double]) {
val x418 = x413 * x389
val x912_zero = 0
val x912_zero_2 = 1.7976931348623157E308
var x912 = x912_zero
var x912_2 = x912_zero_2
var x425 = 0
while (x425 < x423) {
  val x430 = x425 * 1
  val x432 = x430 * x431
  val x916_zero = 0.0
. . .
```

We can see that each symbol in the IR is emitted along with its rhs definition, and statements that include nested expressions (such as **`while`**) are expanded in the generated code. Constant folding has propagated constants to their use site. DSL structs, in this case containing scalars and arrays, have been unwrapped and the struct wrapper (normally a **`class`** definition) has been eliminated, so the fields are

passed directly to the kernel as dependencies. By removing the heap allocation for the class wrapper, this allows primitives to be stack allocated and avoids the otherwise requisite indirection for field accesses. For this to be most effective, all compound (non-primitive) operations should be lifted into the IR, as it is easy to accidentally re-introduce abstraction when performing coarse-grained code generation.

Although the DSL operations used in this snippet were polymorphic, these expressions have been specialized to their actual input type in the generated code. Boxing from polymorphic expressions and classes are one of the primary sources of inefficiences in general-purpose languages. In particular, on the JVM, arrays of primitives are significantly lower-overhead than generic collections and objects. To accomplish this specialization, we make use of `Manifest`s, a Scala compiler-constructed object that provides type information from the call-site of a polymorphic function. We store a manifest with every symbol to maintain its type. One (intentional) consequence of our embedded compiler approach of overloading operations on `Rep` types is that DSL operations are statically dispatched. Therefore, we can always statically specialize types in the generated code.

Static dispatch has some other important advantages. When generating code for heterogeneous devices, we are limited by our target programming model, which does not always support dynamic dispatch. Furthermore, for high performance DSLs, static dispatch provides us the guarantee that we won't have the overhead of dynamic dispatch in the generated code. If more flexibility is required, we can use tagged unions to manually implement dynamic dispatch [103]. However, in many cases we can use type classes to provide expressive statically-dispatched generic programming. For example, we can implement stage-time only type classes `IsVector` and `CanAdd` to statically dispatch methods (which construct IR nodes) during staging:

```
def foo[A,B](x: Rep[A], y: Rep[B])(implicit ev1: IsVector[A],
  ev2: IsVector[B], canAdd: [A,B]) = {
  val a = x + y // provided by CanAdd
  a.length // provided by IsVector
}
```

It is also possible to package dictionaries together with stage-time objects to present

a more OO-traditional inheritance style. We use the type `Interface[T]` to represent a uniform interface for generic types that can have different internal representations (e.g. a sparse or dense `Vector`). By providing static conversions from a concrete DSL type to an `Interface` that stores the method dictionary internally, DSL programs and generic functions in the DSL implementation do not need to care about the underlying representation.

We can package the method dictionary for `DenseVector[T]` into an `Interface[Vector[T]]` as follows:

```
trait VecOpsCls[A] {
  type Self <: Vector[A]
  def length: Rep[Int]
  def +(x: Rep[Self]): Rep[Self]
  def +(x: Interface[Vector[A]]): Rep[Self]
}

// wrapper
class VInterface[A](val ops: VecOpsCls[A])

// method dictionary for DenseVector
class DenseVecOpsCls[A](x: Rep[DenseVector[A]) extends VecOpsCls[A] {
  type Self = DenseVector[A]
  def length: Rep[Int] = densevector_length(x)
  // dispatch for other methods in VecOpsCls
}
implicit def denseToIntf[A](lhs: Rep[DenseVector[A]]) =
  new VInterface[A](new DenseVecOpsCls[A](lhs))

// generic methods
// forward the call to the wrapped method dictionary
class InterfaceVecOpsCls[A:Manifest](val i: VInterface[A]) {
  def length = i.ops.length
  def +(y: Interface[Vector[A]]) = i.ops.+(y)
```

```
}
```

Users then interact with `Interface[Vector[A]]` as follows:

```
def foo[A,B](x: Interface[Vector[A]], y: Interface[Vector[A]]) = {
  val a = x + y // returns Interface[Vector[A]]
  a.length // returns Rep[Int]
}
```

All dispatch is still resolved statically but this time there are some restrictions to the use of wrapper objects. Many constructs (like **`if then else`**) uniformly expect `Rep` types but when the internal `Rep` value is extracted from a wrapper the method dictionary is lost.

Finally, everything we have discussed in this section so far relates to removing abstraction from the host language before reaching the IR. Therefore, all Delite code generators benefit from these efficiency gains. Delite code generators are each defined as a `Traversal` over the IR. Each generator walks the IR and emits code for the desired target language (e.g. Scala, CUDA, C++). The last consideration for efficient code generation is the ability to execute different implementation strategies for different devices. For example, on GPUs we want to stride through memory in a column-oriented way, while for CPUs we want to stride by rows to maximize spatial locality within cache lines. Similarly, GPUs require a different implementation for reductions than CPUs to achieve the best performance. The generated kernels should either already be specialized to a particular device, or (as in Delite's case) should leave the runtime the flexibility to compose the kernels in different ways. For example, we generate loops in each target language as an API that the runtime can call to process n elements at a time. This gives the runtime flexibility in how it schedules a particular loop across the available devices and threads, rather than fixing these values in the generated code. The restrictions in the parallel pattern semantics are also important to ensure that we can generate fast code, since we know that we can safely chunk loops and do not need to insert checks that would add overhead.

2.6 Heterogeneous Runtime

The Delite runtime [11] provides multiple services that implement the Delite execution model across heterogeneous devices. These services include scheduling, communication, synchronization, and memory management. The runtime accepts three key artifacts from the Delite compiler. The first is the Delite Execution Graph (DEG), which enumerates each op of the application, the execution environments for which the compiler was able to generate an implementation of the op (e.g., Scala, C++, CUDA, OpenCL, etc.), and the dependencies among ops. The DEG encodes different types of dependencies separately (e.g. read-after-write and write-after-read), which allows the runtime to implement the synchronization and communication required by each type of dependency as efficiently as possible.

The second artifact produced by the compiler is the generated code for each op, which is wrapped in functions / kernels that the runtime can invoke. Most parallel Delite ops are only partially generated by the compiler and rely on the runtime to patch the compiler generated code into the appropriate execution "skeleton" for the particular hardware. For example, the skeleton for a parallel `Filter` could either run the compiler-generated predicate function on every element and then use a parallel scan to determine the location in the output to write each element that passes the predicate, or it could append each passing element to a thread-local buffer and then concatenate the buffers at the end. This freedom allows for parallelism experts to implement highly specialized and unique skeletons per hardware device that are injected into the final application code just before running, avoiding the need to completely recompile the application per device.

The third artifact is the set of data structures required by the application. This includes the implementation of each structural type as well as various functions which define how to copy instances of that type between devices (e.g. between the JVM and the GPU).

The runtime combines the DEG information with a description of the current machine (e.g. number of CPUs, number of GPUs) and then schedules the application

[11] *Credits*: Kevin Brown is the primary author of the Delite runtime.

onto the machine at walk-time (just before running). It then creates an execution plan (executable) for each resource that launches each op assigned to that resource and adds synchronization, data transfers, and memory management functions between ops as required by the schedule and DEG dependencies. For example satisfying a data dependency between ops on two different CPU threads is achieved by acquiring a lock and passing a pointer via the shared heap, while satisfying the same dependency when one op is scheduled on the CPU and the other on the GPU will result in invoking a series of CUDA runtime routines.

Finally the runtime provides dynamic memory management when required. For ops that run on the JVM we rely on the JVM's garbage collector to handle memory management, but for the GPU the Delite runtime provides a simple garbage collector. First, the runtime uses the dependency information from the DEG, combined with the application schedule, to determine the live ranges of each input and output to a GPU kernel. Then during execution any memory a kernel requires for its inputs and outputs is allocated and registered along with the associated symbol. Whenever the GPU heap becomes too full to successfully allocate the next requested block, the garbage collector frees memory determined to be dead at the current point in execution until the new allocation is successful.

2.7 Building a New Delite DSL

The task of constructing a Delite DSL consists mainly of defining domain-specific data structures (e.g. `Vector`), domain-specific operations (e.g. `VectorPlus`), and domain-specific optimizations (e.g. rewrites).

In this section, we show how to extend the `SimpleVector` example from Section 2.1 into a very small, but complete, DSL in Delite. The first step is to define a `Vector`:

```
class Vector[T] extends Struct with DeliteCollection[T]
  { val length: Int; val data: Array[T] }
```

Next, we need to define lifted operations for `Vector.rand` and `+`; Delite already provides a lifted `println` method, so we don't need to define that. We define our ops in a Scala trait as follows:

```
import delite.BaseOps
trait VectorMathOps extends BaseOps {
  /* syntax exposed to DSL programs */
  object Vector { def rand(n: Rep[Int]) = vec_rand(n) }
  def infix_+[T](lhs: Rep[Vector[T]], rhs: Rep[Vector[T]])
    = vec_plus(lhs, rhs)

  /* abstract methods hide implementation from DSL progs */
  def vec_rand(n: Rep[Int]): Rep[Vector[Double]]
  def vec_plus[T](lhs: Rep[Vector[T]], rhs: Rep[Vector[T]]): Rep[Vector[T]]
}
```

These definitions make the operations visible to the application. However, we must still provide an implementation for these definitions that will construct the Delite IR:

```
import delite.BaseOpsExp
trait VectorMathOpsExp extends VectorMathOps with BaseOpsExp {
  /* construct IR nodes when DSL methods are called */
  // toAtom constructs a new symbol for the IR node
  def vec_rand(n: Exp[Int]) = toAtom(VecRand(n))
  def vec_plus[T](lhs: Exp[Vector[T]], rhs: Exp[Vector[T]]) =
    toAtom(VecPlus(lhs,rhs))

  /* IR node definitions */
  case class VecRand(n: Exp[Int]) extends DeliteOpMap {
    def alloc = NewVector(n) // NewVector def elided
    // thread-safe lifted Math.random func provided by Delite
    def func = e => Math.random()
  }
  case class VecPlus[T](inA: Exp[Vector[T]], inB: Exp[Vector[T]])
    extends DeliteOpZipWith {
    def alloc = NewVector(inA.length) // == inB len check elided
    def func = (a,b) => a+b
  }
```

```
}
```

This snippet shows how we construct two Delite parallel ops, `DeliteOpMap` and `DeliteOpZipWith`, by supplying the functions that match the ops' interface (`func` and `alloc`). In both cases, `func` is required to be pure (have no side effects). Since effects must be explicitly annotated when constructing an IR node, this condition is easy to check at staging-time.

We have now defined the data structures and operations for our DSL. Since the `Vector.rand` and `+` functions are both data parallel ops that operate on the same range, Delite will automatically fuse all of the operations together, resulting in a single loop without any temporary allocations to hold the `Vector.rand` results. Furthermore, since they are both Delite patterns, Delite handles all of the codegen for these ops for multiple targets automatically. The last thing we need to do is package the traits together in a way that can be easily exposed to end users:

```
// ApplicationRunner contains the implementation traits (*Exp)
// and is used to construct the final Scala object to execute
trait SimpleVectorApplicationRunner extends SimpleVectorApplication with DeliteApplication
  with SimpleVectorExp

// SimpleVectorExp packages all of the *Exp traits
trait SimpleVectorExp extends VectorMathOpsExp { // with ...
  this: SimpleVectorApplicationRunner =>
}

// Application contains only in the interface traits (*Ops)
// this is the scope that the 'main' application method is exposed to
trait SimpleVectorApplication extends SimpleVector

// SimpleVector packages all of the *Ops traits
trait SimpleVector extends VectorMathOps { // with ..
  this: SimpleVectorApplication =>
}
```

These traits complete the definition of our small DSL, and will enable us to run the example in Section 2.1. The syntax `this: Foo =>` is Scala syntax for a compile-time assertion that the given trait will eventually be mixed together with `Foo`. This enables us to establish contracts between the different modules of the DSLs. If we had custom code generators, we would also need to package those up in similar traits, which are then be exposed to Delite using the `getCodeGenPkg` method (not shown). Running the program will construct these IR nodes, which will then be scheduled and code generated automatically by Delite. We could take this example a step further by implementing a `Vector.zeros` method in a similar fashion to `Vector.rand`, and then implement the rewrite rule from 2.4 to optimize additions with zero vectors. If we then change our original example from Section 2.1 to make `v2` a call to `Vector.zeros`, the rewrite rule will statically optimize the addition away.

2.8 Summary

In this chapter, we showed how to embed DSL compilers inside an expressive host language using Lightweight Modular Staging (LMS) in Scala. LMS is a library-based solution that enables DSL compilers to easily construct an intermediate representation (IR) of a Scala program and generate code during (Scala) run-time. Staging is well suited to our goal of productive, high performance DSLs because it allows DSL developers to present high-level abstractions to end users, but use staging to programmatically remove those abstractions before generating code. We presented Delite, a framework that provides parallel patterns, generic optimizations and heterogeneous code generators on top of LMS. We demonstrated in detail how Delite simplifies the development of the back-end of an optimizing compiler by providing abstractions for custom analyses, transformations, and efficient code generation. As a simple example, we showed how to build a toy `SimpleVector` DSL in both LMS and Delite. In the next chapter, we'll look at the design and implementation of a real DSL – OptiML, a DSL for machine learning.

Chapter 3

OptiML: an Implicitly Parallel DSL for Machine Learning

In order to demonstrate how we can develop a compiled embedded DSL that provides high productivity and performance in a real domain, we present OptiML, a DSL for machine learning. The goal of OptiML is to allow machine learning practitioners to write code in a highly declarative manner and still achieve high performance on a variety of underlying parallel, heterogeneous devices. The same OptiML program should run well and scale on a CMP (chip multi-processor), a GPU, a combination of CMPs and GPUs, clusters of CMPs and GPUs, and eventually even FPGAs and other specialized accelerators. In order to achieve this, we implement OptiML as a compiled embedded DSL using Delite [1].

3.1 Introduction

Machine learning (ML) is a branch of artificial intelligence (AI) that uses statistical inference to learn patterns from data. ML is widely used to solve data analysis problems, performing tasks such as classification (determining the *label* of a data sample), clustering, or recommendation. In most cases, the accuracy of a statistical model

[1] OptiML was the first "real" DSL implemented in Delite and LMS, and provided many of the driving use-cases for their design.

constructed by an ML algorithm depends on the amount of data the model is *trained* on. Therefore, ML is a domain that requires significant amounts of computational power.

However, as we have argued, computational resources in modern systems are parallel and heterogeneous. In order to take advatnage of these, machine learning developers must have expert knowledge in different programming models each aimed at a specific component of the stack: a message passing library for clusters (e.g. MPI), a threaded library to take advantage of parallelism available in a single compute node (e.g. OpenMP) and a data parallel programming model (e.g. CUDA and OpenCL) to take advantage of the GPU. Multiple programming models are needed because no single model is the right choice for all situations. Furthermore, a significant analysis effort is required to match the various parts of the application to the different programming models, and a mix of programming models is required to achieve peak performance [72].

Ideally, ML developers could leverage these heterogeneous parallel machiness with a programming language that is general, productive and results in high-performance execution. However, no such language currently exists. In practice, most developers use dynamic languages like MATLAB [78], R, or Python, and write sequential code that can only be run on a single CPU (due to the difficulty of writing parallel or distributed code). OptiML bridges the gap between ML algorithms and heterogeneous hardware by providing a high-level, MATLAB-like programming environment while compiling to low-level Scala, C++, and CUDA code.

3.2 Design

The OptiML language focuses on describing *what* an operation should do, rather than *how* it should do it, deferring the *how* to the language implementation and runtime. OptiML describes ML operations using restricted semantics and data structures that generate efficient parallel and heterogeneous code. In this section, we describe the design and the key features of OptiML.

Table 3.1: Example domain-specific data structures

	Sub-type	Semantics
Matrix	Image	Iteration can access pixels within a window
	Training Set	Only streaming (next, prev) access. Can be file-backed and larger than memory
Vector	Indices	Can be used to index vectors and matrices
	Vertices	Iteration can access neighboring vertices
	Edges	Iteration can access connected vertices
	View	A view of a contiguous section of a matrix Updates propagate to the underlying matrix

3.2.1 Domain Model

OptiML is designed to handle iterative statistical inference problems, in particular those that can be expressed by the Statistical Query Model [66] which has been shown to cover a large subset of ML algorithms [27]. These algorithms usually exhibit a combination of regular and irregular data parallelism at varying granularities. OptiML allows these problems to be expressed as dense or sparse linear algebra operations, or as first-class operations on Graph-based data structures. The majority of operations in this model are summation-based (e.g. dot product) and can be parallelized using composable map-reduce operators. However, because ML algorithms typically have many fine-grained operations with low arithmetic intensity, efficiency is more important than in other domains where map-reduce has been traditionally used.

3.2.2 Language Overview

The key data types in OptiML programs are `Vector`, `Matrix`, and `Graph`. These data types are polymorphic and flexible, and come in multiple variants (e.g. `SparseVector`, `DenseVector`). If they are used with scalar values they will be efficient and leverage BLAS and GPU support for applicable operations. However, they can also be used with other types (e.g. a vector of vectors). `Vector`, in particular, can be thought of as an array-like container that takes on its mathematical meaning when used with data types that support arithmetic operations. `Vector` and `Matrix` support all of the standard linear algebra operations used in most ML algorithms. They also provide a wide range of convenient collection operators, such as *map*, *count*, and *filter*. These bulk transformation operators are particularly important in data preprocessing, a key step

```
/* this structure has type-specific restrictions: */

// unordered iteration over elements:
// implemented as a parallel foreach
for (e <- object) { .. }

/* these structures have no indexing restrictions: */

// untilconverged: implemented sequentially, but can
// be parallelized dynamically using optimizations
untilconverged(x, threshold) { x =>
  <new value of x>  }

// sequential
while(condition) { .. }
```

Listing 3.1: Pseudocode snippets demonstrating OptiML control structures.

```
/* the following structures are restricted to
   accessing elements with provided index i only: */

// sum: implemented as a parallel tree-reduce
val ans = sum(begin, end){ i =>
  <ith value to sum>  }

// aggregate: returns a concatenated list of results
// implemented as a parallel tree-reduce
val ans = aggregate(v) { i =>
  <ith value to append to buffer> }

// vector construction: implemented as a parallel map
val my_vector = (0::end) { i =>
  <ith value of my_vector>  }

// matrix construction: implemented as a parallel map
val my_matrix = (0::endRow, 0::endCol) { (i,j) =>
  <(i,j)th value of my_matrix> }
```

Listing 3.2: Pseucode snippets demonstrating OptiML functional operators.

in most ML workflows. Tthe Graph type allows machine learning algorithms based around networks and graphical models (such as belief propagation) to be naturally expressed through iteration over vertices and edges.

In addition to these three core types, OptiML also supports specialized types that support even richer domain-specific operations (e.g. *histogram* on Image). Examples of these types are shown in Table 3.1. A comprehensive listing can be found in the language specification [109].

For convenience, most OptiML data types can be used in a normal imperative way (e.g. using a 'while' loop and assigning each index to a value), but this will result in suboptimal parallel performance. Instead, OptiML encourages the use of the domain-specific control structures and functional operators listed in Listing 3.1 and Listing 3.2. These structures provide the OptiML compiler with additional semantic information while also restricting the operations they support (illegal use of operations will cause a compiler error). By supporting domain-specific access patterns, OptiML can efficiently encode common operations without the performance sacrifices associated with rare and expensive cases. For example, OptiML only allows neighboring vertices in a Graph to be accessed in a bulk operation (e.g. foreach) and automatically synchronizes accesses to these elements while running the operation in parallel. In contrast, a general purpose compiler has to be conservative, because it is unaware of the data structures being used and allows arbitrary memory access patterns to be expressed.

3.2.3 Using OptiML

We demonstrate OptiML syntax and control structures by showing how the k-means clustering algorithm is written in OptiML:

```
untilconverged(mu, tol){ mu =>
  // calculate distances to current centroids
  val c = (0::m){i =>
    val allDistances = mu mapRows { centroid =>
      // distance from sample x(i) to centroid
      ((x(i)-centroid)*(x(i)-centroid)).sum
```

```
    }
    allDistances.minIndex
  }

  // move each cluster centroid to the
  // mean of the points assigned to it
  val newMu = (0::k,*) { i =>
    val (weightedpoints, points) = sum(0,m) { j =>
      if (c(i) == j){
        (x(i),1)
      }
    }
    if (points == 0) Vector.zeros(n)
    else weightedpoints / points
  }

  newMu
}
```

This example highlights the usage of four OptiML structures: *untilconverged{..}*, *vector constructor (0::m){..}*, *matrix constructor (0::k,*){..}*, and *sum{..}*. Unlike MATLAB functions, each of these structures accepts as an argument any user-defined function that meets the corresponding requirement in Listing 3.1 or Listing 3.2. The syntax `x => y` represents a function that takes a value x and returns a value y. *untilconverged* iterates until the difference between `mu` and `newMu` falls below a provided tolerance `tol`. *vector constructor* computes each value of the new `c` vector, which represents the closest cluster for each training sample. *matrix constructor* computes a new $k \times n$ Matrix by computing a new vector for each new cluster location.

These abstractions represent commonly occuring operations in machine learning; in future sections, we will show how we use them to generate high-performance parallel code. This example also shows that vectors (e.g. *x(i)*) and matrices can be used with normal arithmetic syntax. The *mapRows* function is used to perform an operation on every row of a matrix in a concise way. Together, these features allow

for programs that resemble pseudocode or scripts. This can be extremely useful during algorithm prototyping, when one wishes to focus on algorithm description rather than implementation details. In the following section, we explore in more detail how OptiML can be used to write more productive code.

3.3 Productivity

To demonstrate how OptiML's machine learning abstractions can increase programmer productivity and application readability, we compare an OptiML application to a corresponding C++ version. The application we will explore is used for visual object detection on the Willow Garage PR2 robot. The algorithm searches across an image for matches against a database of binary gradient templates and produces a list of object detections and their locations in the image [11]. The snippet below is used to filter image gradients via non-max suppression:

C++:

```
void gradMorphology(Mat &gradient, Mat &clnGradient) {
  int rows = gradient.rows;
  int cols = gradient.cols;
  // zero out borders
  uchar *bptrTop = gradient.ptr<uchar>(0);
  uchar *bptrBot = gradient.ptr<uchar>(rows - 1);
  for(int x = 0; x < cols; ++x, ++bptrTop, ++bptrBot)
    *bptrTop = 0; *bptrBot = 0;
  for(int y = 1; y < rows - 1; ++y) {
    uchar *bptr = gradient.ptr<uchar>(y);
    *bptr = 0; *(bptr + cols - 1) = 0;
  }
  // ... 23 lines omitted
  for(int y = 0; y < rows - 2; ++y) {
    // ... 5 lines omitted
    uchar *c = clnGradient.ptr<uchar>(y+1);
```

```
    // ... 6 lines omitted
    for(int x = 0; x < cols - 2; ++x, ++c) {
      // ... 3 lines omitted
      int maxindx = 1; int maxcnt = counts[1];
      for(int j = 2; j < 9; ++j) {
        if(counts[j] > maxcnt)
          maxindx = j; maxcnt = counts[j];
      }
      if(maxcnt > 1) { *c = maxindx; }
      // ... 3 lines omitted
    }
  }
}
```

OptiML:

```
def gradMorphology(gradient: GrayscaleImage) = {
  // zero out borders
  gradient.top(1) = 0; gradient.bottom(1) = 0
  gradient.left(1) = 0; gradient.right(1) = 0

  // returns a new Matrix of the same dimensions
  gradient filter(3, 3) { window =>
    val (max,maxIdx) = window.histogram.maxWithIndex
    if (max > 1) maxIdx else 0
  }
}
```

This part of the algorithm does a sliding window computation (filter) over the image, computing a histogram for each window. The output of the filter is the index of the largest histogram value within the window, if the value is greater than 1 (otherwise the output is 0). The C++ code consists of several nested loops that compute the window efficiently, but the resulting code is difficult to read and even harder to parallelize because it contains writes to shared data structures. In contrast, the OptiML code is succinct and expresses only algorithmic intent. The *filter* operation

accepts any user-defined function that computes a result without writing to shared data structures. Thus, it is more expressive than MATLAB's fixed function convolution operators, but still restrictive enough to allow OptiML to generate efficient parallel or CUDA code. This example shows how OptiML's restricted semantics lead programmers towards patterns that can be naturally expressed and easily parallelized.

While both the C++ and OptiML versions of the previous example are written sequentially, the OptiML code executes in parallel. It is important to note that the OptiML code does not include implementation details for the target parallel architectures. The same source code currently targets systems with CPUs and/or GPUs, and will also compile to new parallel hardware as support is added to the OptiML compiler. For example, we are currently working on adding support for clusters, where each compute node contains a combination of CMPs and GPUs; OptiML programs will automatically inherit this support. In contrast, with MATLAB a developer must choose up front whether to write a loop to be run in parallel (using *parfor*), on a GPU (using *gfor* with Jacket), or in its most efficient sequential form (using *vectorization*). This decision is difficult or impossible to make statically, because the right choice depends on factors such as data size, parallelization overhead, and the availability and characteristics of the target hardware.

3.4 Performance

OptiML reasons about programs at the level of domain-specific operations, which enables it to provide better parallel performance than a library or general-purpose language. At the heart of OptiML's ability to deliver high performance is its ability to build, analyze, and optimize an intermediate representation (IR) of the user program. In this section we show how we build and reason about this IR by illustrating how OptiML is implemented in Delite.

3.4.1 Building an IR

OptiML is implemented in the same way as the `SimpleVector` DSL in Chapter 2.7, but now we have more sophisticated types and operations. In this section, we show how to tackle the requirements of a real-world DSL through selected examples from OptiML. The full OptiML implementation is open source, and available at: `http://stanford-ppl.github.io/Delite/optiml/`[2].

Stage-time only types: IndexVector

The type `IndexVector` is an OptiML type that represents an immutable, contiguous range of integer indices. This type can be be used to index into collections (vectors or matrices), as well as used to construct new vectors or matrices. The most interesting aspect of `IndexVector` is that it is a stage-time only abstraction. It is logically a struct containing two integers (start and end), but since it is immutable, we can always statically rewrite accesses to an `IndexVector` to operations on its underlying fields. As a result, the "struct wrapper" (or class wrapper) for `IndexVector` never needs to be generated, and instead we pass the integer arguments directly by value in the generated code.

First, we define the `IndexVector` type and an IR node that represents constructing a new `IndexVector`:

```
trait IndexVectorOpsBase extends Base {
  abstract class IndexVector
  def indexvector_new(start: Rep[Int], end: Rep[Int]): Rep[IndexVector]
}

trait IndexVectorOpsBaseExp extends IndexVectorOps with BaseExp {
  case class IndexVectorNew(start: Exp[Int], end: Exp[Int]) extends Def[IndexVector]
  def indexvector_new(s: Exp[Int], e: Exp[Int]) = reflectPure(IndexVectorNew(s, e))
}
```

[2]Last retrieved 1/3/2014

Now all we need to do is provide `apply` and `length` methods (for simplicity, these are the only methods we will define on `IndexVector` in this example; in practice OptiML implements the entire immutable subset of the `DenseVector` interface).

```
trait IndexVectorOps extends IndexVectorOpsBase {
  implicit class IndexVectorOpsCls(x: Rep[IndexVector]) {
    def apply(n: Rep[Int]) = indexvector_apply(x,n)
    def length = indexvector_length(x)
  }

  def indexvector_apply(x: Rep[IndexVector], n: Rep[Int]): Rep[Int]
  def indexvector_length(x: Rep[IndexVector]): Rep[Int]
}

trait IndexVectorOpsExp extends IndexVectorOps with IndexVectorOpsBaseExp {
  // short-circuit directly to underlying fields!
  def indexvector_apply(x: Rep[IndexVector], n: Rep[Int]) = x match {
    case Def(IndexVectorNew(start, end)) =>
      if (n > end) error("IndexVector access out of bounds")
      start + n
  }
  def indexvector_length(x: Rep[IndexVector]) = x match {
    case Def(IndexVectorNew(start, end)) => end - start
  }
}
```

With these definitions, an end-user can construct a new `IndexVector` and access its elements, but no `IndexVector` class will ever be generated. This is exactly the same mechanism we described previously in the more general context of Delite structs (Chapter 2.3.2), and in practice we actually use a `Struct` to do the short-circuiting.

Syntactic sugar: vector constructor

Next, we demonstrate how a flexible host language like Scala enables embedded syntax that looks like an external DSL. The OptiML syntax for *vector constructor* is:

```
(0::m) { (i,j) => // compute ith value of output vector }
```

We implement the left-hand side of this statement in Scala using an implicit conversion from a `Rep[Int]` to an anonymous class that defines a `::` method:

```
implicit class IndexVectorSugar(start: Rep[Int]) {
  def ::(end: Rep[Int]) = indexvector_new(start, end)
}
```

Now, the statement `0::m` will construct an `IndexVector` from `0` to `m`. Next, we add an overloaded `apply` method on `IndexVector` that accepts a function:

```
implicit class VectorOpsCls(x: Rep[IndexVector]) {
  def apply[A](func: Rep[Int] => Rep[A]) = vector_construct(x,func)
}
def vector_construct[A](x: Rep[IndexVector], func: Rep[Int] => Rep[A]): Rep[DenseVector[A]]
```

This is the only interface exposed to users (for program safety, they cannot introspect to examine the expression tree). We use an abstract method `vector_construct` to decouple the method syntax from the method implementation inside the compiler. Finally, inside the corresponding `Exp` trait, we implement `vector_construct` to construct a new `DenseVector` using a `DeliteOpMap`:

```
case class VectorConstruct[A](in: Exp[IndexVector], func: Exp[Int] => Exp[A])
  extends DeliteOpMap[Int,A,DenseVector[A]] {
    val size = in.length
    def alloc = DenseVector[A](size)
  }

def vector_construct[A](x: Exp[IndexVector], func: Exp[Int] => Exp[A]) =
  reflectPure(VectorConstruct(x, func))
```

This completes the implementation of *vector constructor*.

Functional control: untilconverged

An important aspect of OptiML's design, compared to alternatives like MATLAB [78], R [96], NumPy [62], or Julia [9], is its emphasis on functionally pure operations to

enable parallelization and optimization. `untilconverged` is an example of a functional control structure. It accepts as argument an initial value and a function that, given the value of a previous iteration, computes the value of the next iteration. In this way it produces an iterative result by chaining together pure operations, rather than via mutation as in a traditional **`while`** loop. The advantage is that we can maintain a functionally pure programming model for this common operation, rather than forcing users to resort to side effects.

We implement `untilconverged` efficiently in OptiML using double buffering and fusion. First, we allocate two buffers (of some polymorphic type `A`). The first buffer is passed into the user-defined function, which produces an output value `out`. We write `out` into the second buffer in place, using a parallel loop. If `out` is produced by a parallel loop inside the user-defined function, the production of `out` is fused with the write into the buffer, and no intermediate object is allocated. Note that while this is an improvement over functional solutions without fusion, it is not a guarantee without other restrictions; if the user can construct `out` sequentially, then there will be a new allocation each iteration since fusion cannot eliminate it.

Composite op: mean

So far, we have demonstrated how we can construct IR nodes that correspond directly to a Delite op. How do we construct IR nodes that represent a composition of Delite ops? One way is to inline the composite operation, which simply creates multiple IR nodes. For example, consider the `mean` operation, which computes the mean of the elements in a numeric vector. We can define this operation inside OptiML as follows:

```
def densevector_mean[A](x: Exp[DenseVector[A]])(implicit conv: Exp[A] => Exp[Double]) = {
  x.map(conv).sum / x.length
}
```

Instead of creating a single IR node representing `mean`, this will create a subgraph of IR nodes, just like if the user had written this function (one for `map`, one for `sum`, and so on). What if, though, we needed to pattern match on the `mean` operation in order to implement some rewrite optimization? We could match on the entire subgraph, but

that is not ideal. Instead, we use Delite's `Transformer` facility to implement a *lowering* transformation from the composite operation `mean` to its inlined implementation:

```
trait DenseVectorOpsExp extends DenseVectorOps with DeliteOpsExp with LoweringTransform {
  case class DenseVectorMean[A](x: Exp[DenseVector[A]], conv: Exp[A] => Exp[Double])
    extends Def[Double]

  def densevector_mean[A](x: Exp[DenseVector[A]])(implicit conv: Exp[A] => Exp[Double]) = {
    reflectPure(DenseVectorMean(x,conv))
  }

  // lower to composite implementation at a later phase
  override def onCreate[A:Manifest](s: Sym[A], d: Def[A]) = (d match {
    case DenseVectorMean(x) => s.atPhase(deviceIndependentLowering) {
      x.map(conv).sum / x.length
    }
    case _ => super.onCreate(s,d)
  }).asInstanceOf[Exp[A]]
}
```

Now we can perform domain-specific rewrites on the `DenseVectorMean` IR node before the `deviceIndependentLowering` transformer is run. After the transformation, any rewrites defined on the lower-level ops (`VectorMap`, `VectorSum`, etc.) will also be triggered to perform further optimization. Therefore we can easily (and modularly) extend our compiler's semantic model to implement hierarachical domain-specific transformations.

External libraries: matrix multiplication

There exist a number of high performance linear algebra routines in BLAS [70] and LAPACK [3] that we want to take advantage of in OptiML. A classic example is matrix multiplication, which is already optimized for different architectures in good BLAS implementations. In order to make an external method call, we need to stage the call to defer it to run-time. For Scala and Java libraries, we can create an abstract

IR node (a `Def`) and implement a code generator to emit the library call:

```
trait DenseMatrixOpsExp extends BaseExp {
  case class DenseMatrixMultiply(x: Exp[DeliteArray[Double]], y: Exp[DeliteArray[Double]])
    extends Def[DeliteArrayDouble]

  def densematrix_multiply(x: Exp[DenseMatrix[Double]], y: Exp[DenseMatrix[Double]]) = {
    val z = reflectPure(DenseMatrixMultiply(x._data, y._data)
    densematrix_fromarray(x.numRows, y.numCols, z)
  }
}

trait DenseMatrixScalaGen extends ScalaGenBase {
  val IR: DenseMatrixOpsExp
  import IR._

  override def emitNode(sym: Sym[Any], rhs: Def[Any]) = rhs match {
    case Def(DenseMatrixMultiply(x,y)) =>
      emitValDef(sym, "myFastLib.matmult("+quote(x)+","+quote(y)+")")
    case _ => super.emitNode(sym, rhs)
  }
}
```

This will work provided we include the jar file for "myFastLib" in the project's classpath (for example, as an SBT [122] managed dependency). For native (C or C++) libraries, we need to do more work. We must tell Delite where to find the library (so that it can link it in at run-time when compiling the generated kernels) as well as how to invoke the external call using JNI. To do this, we use `DeliteOpExternal`:

```
case class DenseMatrixMultiplyBLAS(xR: Exp[Int], x: Exp[DeliteArray[Double]],
                                   yC: Exp[Int], y: Exp[DeliteArray[Double]])
  extends DeliteOpExternal[DeliteArray[Double]]] {

  def inputs = scala.List(x,y)
  def alloc = DeliteArray[A](xR * yC)
```

```
  val funcName = "matMult"
}
```

In a separate trait, we define the external library interface by implementing a method that emits the required JNI code for for each library function (e.g. `DenseMatrixMultiplyBLAS`), and at run-time we must provide an XML configuration file specifying the location of the library (and optional parameters like compilation flags).

More complex data structures: sparse matrices

OptiML implements a `SparseVector` as a struct containing two arrays, one for values and one for indices. OptiML provides both a `SparseMatrixBuildable` type and a `SparseMatrix`. `SparseMatrixBuildable` is implemented as a mutable struct of three arrays in the COO (coordinate list) format, and is efficient for sparse matrix construction. A `SparseMatrixBuildable` contains a method `finalize`, which converts it to an immutable `SparseMatrix`. `SparseMatrix` is also represented as a struct of three arrays, but in the CSR (compressed sparse row) format, which is efficient for computation. To illustrate how we implement a simple sparse type in Delite, `SparseVector` is implemented as:

```
case class SparseVectorNew[T](length: Rep[Int], isRow: Rep[Boolean])
  extends DeliteStruct[SparseVector[T]] {

  val elems = collection.Seq(("_length", length),
                             ("_isRow", isRow),
                             ("_data", DeliteArray[T](32)),
                             ("_indices", DeliteArray[Int](32)),
                             ("_nnz", 0))
}

def sparsevector_new[T](length: Rep[Int], isRow: Rep[Boolean]) =
  reflectPure(SparseVectorNew(length, isRow))
```

In this example, we have selected 32 as a default buffer size for the non-zero elements. Parallelizing operations on `SparseVector` is trivial if we only need to operate on the non-zero elements (for example, the function `mapnz`). In this case, we perform a

`map` on the `_data` array and wrap the result in a new `SparseVector`. Parallelizing sparse operations in general has not yet been attempted in OptiML. In particular, Delite (at the time of this writing) does not contain any irregular parallel ops. Instead, operations like sparse vector addition are implemented as sequential loops that efficiently scan the argument vectors.

One interesting way we can take advantage of our DSL IR is to statically detect zero-preserving operations, even while providing a generic interface (`map` instead of `mapnz`) [103]. We define the following, stage-time evaluated function:

```
def isZeroFunc(f: Rep[A] => Rep[B], len: Rep[Int]) =
  // use symbolic evaluation to test the user-defined
  // function f against a zero-valued argument
  f(defaultValue[A]) match {
    case Const(x) if (x == defaultValue[B]) => true
    case _ => false
  }
```

When the user-provided function symbolically returns a "zero" value on a "zero" input, `isZeroFunc` returns true and we can transform the `map` to a `mapnz`. This is particularly useful if we want to provide a generic `Interface[Vector[T]]` (Chapter 2.5), where the actual `Vector[T]` instance may be dense or sparse. In this case, we can use staging to statically transform from a representation-agnostic implementation of `map` to a representation-sensitive one.

3.4.2 Analyses and Optimizations

OptiML performs several static and dynamic optimizations. Static optimizations are applied as transformations on the OptiML IR before code generation, while dynamic optimizations are implemented as part of OptiML data types or control structures.

Static domain-specific optimizations

OptiML inherits general and well-known static optimizations such as common subexpression elimination, dead code elimination, and loop hoisting through Delite and

LMS. All of these optimizations occur at the granularity of DSL operations (e.g. vector plus). It also provides domain-specific simplication rewrite rules for linear algebra.

For example, a simplification that can be exploited in Gaussian Discriminant Analysis (GDA) is:

$$\sum_{i=0}^{n} \vec{x}_i * \vec{y}_i \rightarrow \sum_{i=0}^{n} X(:,i) * Y(i,:) = X * Y$$

This optimization converts a summation of outer products into a single matrix multiplication. We also use pattern rewriting to identify sequences of operations that should be generated differently depending on target device, consolidating them into a single IR node that can be generated accordingly.

Op fusion is a static optimization performed by Delite, enabled by OptiML's classification of operators into Delite ops (e.g. vector + vector as `DeliteOpZipWith`), that is particular important in machine learning. As a practical example, consider the following line from the SMO algorithm [91] for SVM (`*:*` is a dot product):

```
val eta = (X(i)*:*X(j)*2) - (X(i)*:*X(i)) - (X(j)*:*X(j))
```

Here, OptiML automatically fuses all of the dot product calculations into a single loop instead of 4 (3 for each dot product plus 1 for the scalar multiplication). For the entire SMO algorithm, op fusing reduces 35 loops to 11. More importantly, fusing an operation can eliminate allocations of intermediate data structures. The `*:*` operator can be implemented as `(X(i) * X(j)).sum` and op fusing will ensure that no intermediate vector will be allocated to hold the result of `X(i) * X(j)`.

Dynamic domain-specific optimizations

In addition to the static optimizations above, there are two ways OptiML allows users to exploit the statistical nature of machine learning and its subsequent tolerance for relaxed consistency.

Best-effort computing: because many ML algorithms are iterative and probabilistic, they are often robust to minor variations in computation [80]. OptiML allows users to trade-off accuracy, if they choose, for better performance, by using best effort data structures. These data structures drop computations according to a policy,

which can improve single-threaded execution time and reduce sequential bottlenecks, improving parallel scalability.

Relaxed dependencies: for the same reasons as above, it is sometimes useful to allow ML algorithms to intentionally race, which again can improve parallel performance at the expense of strict consistency for some operations. OptiML provides a version of the *untilconverged* construct that allows some number of iterations to be run in parallel. Recent work [134, 84] has shown the potential for this optimization.

3.4.3 Code Generation

Nearly all of OptiML's code generation is handled automatically by Delite via Delite ops (Chapter 2.3.1). By generating code from high-level parallel ops, Delite is able to generate code that is optimized for each target device; for example, GPUs prefer to stride column-wise through a row-major Matrix as this maximizes the global memory bandwidth utilization by coalescing the memory requests from multiple threads, whereas CPUs prefer to stride row-wise to maximize single-threaded locality. We implement the appropriate stride in Delite by decomposing the parallel op (determining the # of threads and what elements to assign to each thread) in a device-specific way that maximizes performance per device.

OptiML's heterogeneous code generation is essential to providing a portable, productive programming model with the best possible performance. Some operations either do not fit the CUDA model or actually perform worse on a GPU, and it is necessary to be able to run these on a CPU. Since application programmers do not specify any device-specific details, OptiML can generate multiple versions and select the best one at runtime. In contrast, with both MATLAB and AccelerEyes' GPU support, the programmer must specify which data structures should be resident on the GPU device memory. Manually determining which data structures should go to the GPU and deciding when to bring the data back to obtain the best performance is difficult and varies with problem size.

3.4.4 Discussion

The DSL design process in Delite, which we have illustated with OptiML, is the identification of domain abstractions and their mapping to Delite ops. This mapping defines the granularity of DSL computation as well as their semantic restrictions (e.g. mutability). While Delite provides a structured way to perform the mapping, the design process remains a hard problem. The more expressive a DSL is, the less likely it will be translateable in an efficient way to many different targets. For example, the coarse granularity and restricted semantics of OptiML's control and functional structures enable the OptiML compiler to safely and efficiently target the same application code to both CMPs and GPUs, without requiring sophisticated analyses. However, OptiML also allows users to write imperative code (e.g. a **`while`** loop) when necessary. This trades off parallelism for expressivity and requires end-user expertise to be used effectively. More specifically, it is an example of the trade-off between relying solely on high-level functional abstractions (and optimizing them away as described in Chapter 2.5) vs. providing sharper, low-level tools as well. The eventual goal of OptiML is to remove all imperative constructs while maintaining the ability to express the important problems in ML without convoluted restructuring.

3.5 Evaluation

This section presents results for a set of machine learning algorithms implemented in OptiML. We compare productivity to C++ and performance to reference implementations written using existing alternative systems, including MATLAB, GraphLab [75], and C++. In addition, we analyze the performance improvements achievable due to OptiML's static and dynamic optimizations described in Section 3.4.2. Finally, we provide an in-depth case study using OptiML to derive benefits in a real application (analysis of protein folding simulation data).

3.5.1 Methodology

We first compare the productivity of OptiML to C++ by showing side-by-side comparisons. We use C++ at a comparison since it highlights what is typically required to achieve the best performance on multicore CPUs (for GPUs, hand-coded CUDA would also be required). We qualitatively evaluate the additional expressiveness OptiML provides for these applications. Due to the length of some applications, we omit them from the comparison; for others, we extract only the key algorithm. Since k-means was already explored in depth in Section 3.2.3, we omit it. Additionally, instead of comparing linear regression (which is too simple to be interesting), we compare logistic regression.

For the first set of performance experiments, we compare our applications to multiple MATLAB implementations. We used MATLAB 7.11 with its CPU parallelization and GPU support, as well as GPU support from AccelerEyes's Jacket [1]. Each application is algorithmically identical, but for the MATLAB versions we made a reasonable effort to vectorize and parallelize the CPU code, and make the best data locality choices for the GPU. In cases where both vectorization or parallelization was possible, we report the results for the version that had the best performance at 8 CPUs.

We also present two ML applications that are not well suited to MATLAB, and therefore chose an alternative language to provide a performance baseline. We implemented a version of loopy belief propagation (LBP) in OptiML and compare it to a baseline implementation in GraphLab, which is a C++ library for ML graph applications. We also compare the binarized gradient template matching (TM) algorithm described in section 3.3 to a hand-optimized C++ baseline.

For each of the experiments we timed the computation component of the application, without initialization. We ran each application (with initialization) 10 times in order to warm up the JIT and smooth out fluctuations due to garbage collection and other variables. We present here the averaged time of the last five executions. We used a Dell Precision T7500n with two quad-core Intel Xeon X5550 2.67 GHz processors, 24GB of RAM, and an NVidia Tesla C2050.

3.5.2 Productivity Comparison

We compare three common ML applications written in OptiML to their C++ counterparts: Gaussian Discriminant Analysis (GDA), Naïve Bayes (NB), and Logistic Regression (LogReg).

GDA:

GDA (Figure 3.1) builds a Gaussian model from the data by estimating the Gaussian parameters `mu` and `sigma` (mean and variance) from the data. In this example, `x` is the training data and `y` is the labels (which can be either true or false). The OptiML version of the application uses higher-level constructs like `count` and `sum`, while the C++ version uses manual **`for`** loops. Furthermore, the C++ version (which uses the uBLAS library [129]) does not have infix notation for operators like `outer` and `trans`, making the code a little more difficult to read. Finally, note the use of the `noalias` function in the C++ version. This is a uBLAS function that is used to improve performance if the programmer can assert that that the right-hand expression does not alias the left-hand return value. These little details are difficult to get right; in OptiML, they are handled by the DSL rather than the end-user.

NB:

Naïve Bayes (Figure 3.2) is a classic machine learning classification algorithm. It assumes a conditionally independent model, which enables the computation of the model parameters (`phi_y0`, `phi_y1`, `phi`) to be very simple. In this example, the training set is a set of documents, where each document is represented as a count of the number of times each word in the dictionary appeared in that document. The label for the document is either nonSpam (0) or spam (1). We compute the model parameters as summations over the training set. Since the application is so simple, the C++ version is also quite straightforward. Still, the OptiML version employs higher-level constructs like vector constructors and summations. Furthermore, since OptiML automatically performs fusion and code motion, the programmer does not need to worry about manually pre-calculating or fusing computations. This allows the OptiML code to be structured in a way that directly reflects the model parameters

```
val y_zeros = y count { _ == false }
val y_ones = y count { _ == true }
val mu0_num =
  sumRowsIf(0,m) { !y(_) } { x(_) }
val mu1_num =
  sumRowsIf(0,m) { y(_) } { x(_) }

val phi = 1./m * y_ones
val mu0 = mu0_num / y_zeros
val mu1 = mu1_num / y_ones

val sigma = sum(0, m) { i =>
  if (y(i) == false) {
    (((x(i)-mu0).t) ** (x(i)-mu0))
  }
  else {
    (((x(i)-mu1).t) ** (x(i)-mu1))
  }
}
```

(a) OptiML GDA

```
double y_ones = 0.0, y_zeros=0.0;
vector<double> mu0_num(n);
vector<double> mu1_num(n);

for(uint i=0; i<m ; i++) {
  if(y(i)==false) {
    y_zeros++;
    mu0_num += row(x,i);
  } else {
    y_ones++;
    mu1_num += row(x,i);
  }
}

double phi = 1.0/m * y_ones;
vector<double> mu0 = mu0_num / y_zeros;
vector<double> mu1 = mu1_num / y_ones;

matrix<double> sigma(n,n);
for(uint i=0; i<m; i++) {
  if(y(i) == false) {
    matrix<double> out =
      outer_prod(trans(row(x,i)-mu0),
                 row(x,i)-mu0);
    noalias(sigma) += out;
  } else {
    matrix<double> out =
      outer_prod(trans(row(x,i)-mu1),
                 row(x,i)-mu1);
    noalias(sigma) += out;
  }
}
```

(b) C++ GDA

Figure 3.1: GDA snippets in OptiML and C++

```
val phi_y1 = (0::ts.numFeatures) { j =>
  val spamWordCount =
    ts.data(spamIndices).getCol(j).sum
  val spamTotalWords =
    wordsPerDoc(spamIndices).sum
  (spamWordCount + 1) /
  (spamTotalWords + ts.numFeatures)
}

val phi_y0 = (0::ts.numFeatures) { j =>
  val nonSpamWordCount =
    ts.data(nonSpamIndices).getCol(j).sum
  val nonSpamTotalWords =
    wordsPerDoc(nonSpamIndices).sum
  (nonSpamWordCount + 1) /
  (nonSpamTotalWords + ts.numFeatures)
}

val phi_y = ts.labels.sum / ts.numSamples
```

(a) OptiML NB

```
mat featuresTrans = trans(features);
for(uint j = 0; j < numTokens; j++) {
  double spamWordCount = 0.0;
  double spamTotalWords = 0.0;
  double nonSpamWordCount = 0.0;
  double nonSpamTotalWords = 0.0;

  for (uint i = 0; i < numTrainDocs; i++) {
    if (classifications(i) == 1) {
      spamWordCount += featuresTrans(j,i);
      spamTotalWords += wordsPerDoc(i);
    }
    else {
      nonSpamWordCount += featuresTrans(j,i);
      nonSpamTotalWords += wordsPerDoc(i);
    }
  }
  phi_y1(j) = (spamWordCount + 1) /
              (spamTotalWords + numTokens);
  phi_y0(j) = (nonSpamWordCount + 1) /
              (nonSpamTotalWords + numTokens);
}

double spamCount = 0.0;
for(uint i = 0; i < labels.length; i++) {
  spamCount += labels(i); // 0 or 1
}
double phi_y = spamCount / numTrainDocs;
```

(b) C++ NB

Figure 3.2: Naïve Bayes snippets in OptiML and C++

(each statement computes a single parameter).

LogReg:

Logistic Regression (Figure 3.3) is one of the most popular machine learning algorithms due to its simplicity and effectiveness. In this example, we use LogReg to learn a binary classifier. Logistic regression is similar to linear regression where we wish to learn the weights of a predictor function, but in this case we take advantage of the fact that the output labels are discrete rather than continuous by mapping the input values through the sigmoid function $g(z) = 1/(1+e^{-z})$. The OptiML version is straightforward: we learn the weights using gradient descent, which is easily expressible with `untilconverged`, `sum`, and `sigmoid`. In the C++ version, we do the same thing, but it is far more complex in order to achieve good performance. In the previous two examples, we showed only sequential C++ code. Here, we also demonstrate what is required to efficiently parallelize the implementation. We use OpenMP [24], but must still be careful to safely create thread-local buffers to avoid bottlenecking on the shared parameter vector. This forces us to structure the code in a particular way that is not relevant to the algorithm. Furthermore, we must be aware of OpenMP parameters like `schedule(static)`. As a result, the C++ version is not accessible to many programmers, and is difficult to maintain moving forward.

3.5.3 Performance Comparison

Figure 3.4 compares the performance of OptiML to C++ and MATLAB for six classic machine learning algorithms. For C++ we used the Armadillo linear algebra library [106] and wrote hand-optimized imperative implementations. We also made a reasonable effort to optimize the MATLAB implementations, including vectorizing the code where possible. All OptiML implementations show better or comparable performance to the MATLAB and C++ library implementations. This is because, as we showed in the previous section, OptiML applications are written in a way that exposes high-level semantics in the IR. By optimizing and generating code from parallel patterns,

```
val alpha = 1.0
val w = untilconverged(theta) {
  cur =>
  val gradient = sum((0::x.numRows) { i =>
    x(i)*(y(i) - sigmoid(cur *:* x(i)))
  })
  cur + gradient*alpha
}
```

(a) OptiML LogReg

```
// 10 initialization lines omitted
do {
  // 5 setup lines omitted
  #pragma omp parallel private(i,j)
  {
  int thread = omp_get_thread_num();
  for (j = 0; j < numFeatures; j++) {
    all_gradients[thread][j] = 0.0;
  }
  #pragma omp for private(i,j) schedule(static)
  for (i = 0; i < numSamples; i++) {
    double hyp_temp = 0;
    for (j = 0; j < numFeatures; j++) {
      hyp_temp += -1.0 * cur[j] * x[i][j];
    }
    double hyp =
      v[i] - (1.0 / (1.0 + exp(hyp_temp)));
    for (j = 0; j < numFeatures; j++) {
      all_gradients[thread][j] += x[i][j] * hyp;
    }
  }}
  for (i = 0; i < numThreads; i++) {
    for (j = 0; j < numFeatures; j++) {
      gradient[j] += all_gradients[i][j];
    }
  }
  for (j = 0; j < numFeatures; j++) {
    gradient[j] = cur[j] + alpha * gradient[j];
  }
  diff = 0; for(i = 0; i < numFeatures; i++)
    diff += fabs(gradient[i] - cur[i]);
} while (diff > tol && iter < maxIter);
// 6 clean-up lines omitted
```

(b) C++ LogReg

Figure 3.3: Logistic Regression snippets in OptiML and C++

we are able to generate code that is similar to the manually optimized implementation. The OptiML compiler eliminates overhead by inlining a large number of small operations and uses op fusion to generate more efficient kernels, avoiding unnecessary intermediate allocations. One application where MATLAB performs slightly better than OptiML is the Restricted Boltzmann Machine (RBM). This is because RBM is dominated by BLAS [70] operations that MATLAB offloads to an efficient BLAS library, and there is not much room for compiler optimizations in the application.

The reason some C++ implementations show better performance (e.g. k-means) is because the C++ version aggressively re-uses allocated memory across loop iterations while the Delite version performs a new allocation each iteration. While this improves sequential performance, it also prevents the code from being parallelized. The OptiML version, in contrast, scales well across multiple cores, and for k-means achieves an even greater 13x speedup by executing parallel ops on the GPU. Generating efficient GPU code for k-means requires Delite to perform multiple compiler transformations. The application as written in OptiML contains nested implicitly data-parallel operators that allocate results. While CPU code generators can easily perform the nested allocations locally within a thread, dynamic thread-local allocations are either not supported or not efficient on current GPUs. Therefore the Delite compiler transforms the outer parallel op into a sequential loop that can run on the CPU and generates the inner parallel ops as parallel CUDA kernels.

GDA shows the most benefit from efficient GPU code generation. Delite generates CUDA kernels that keep the intermediate results in registers instead of the device memory and applies work duplication to remove dependencies between threads, which maximizes the GPU performance. RBM also shows good performance on the GPU since the application heavily uses floating point matrix multiplication operations, which runs efficiently on the GPU using a large number of GPU cores. On the other hand, Naïve Bayes does not perform well on the GPU because the arithmetic intensity of the application is too low to fully utilize the compute capability of the GPU and the performance is limited by the transfer cost of moving the input data from the main memory to GPU device memory. The Support Vector Machine (SVM) application

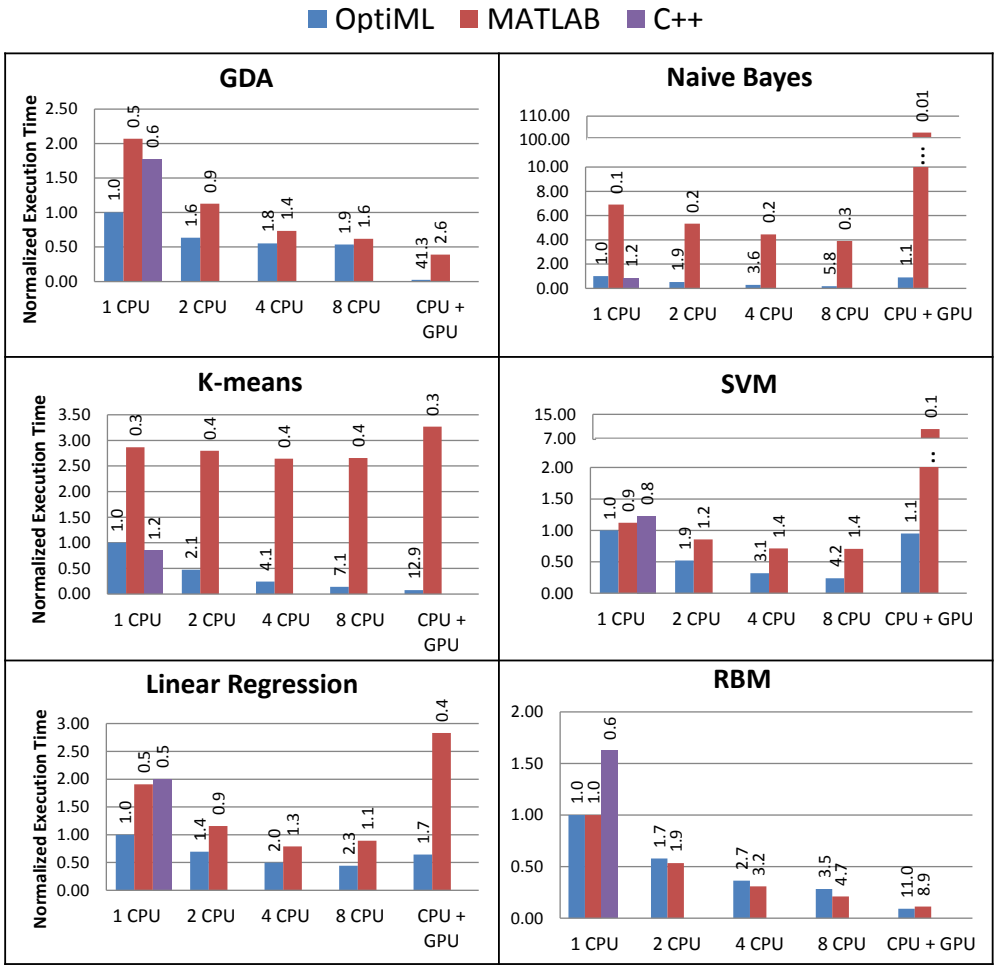


Figure 3.4: Execution time of our applications compared to C++ and MATLAB and normalized to single-threaded OptiML. Speedup numbers are reported on top of each bar.

also does not perform well because it is an iterative algorithm that has intermediate-sized data-parallel operations for each iteration, which is not enough to fully utilize the GPU and requires frequent data movement between the CPU and GPU.

Figure 3.5 compares the performance of two larger applications written in OptiML to a baseline written in an alternative environment. LBP is compared against an equivalent implementation in GraphLab. The results show that OptiML achieves performance and scaling close to GraphLab, which is written in C++ and designed specifically for Graph-based algorithms. TM is compared to a baseline C++ implementation which is single-threaded and designed for high performance robotics. The

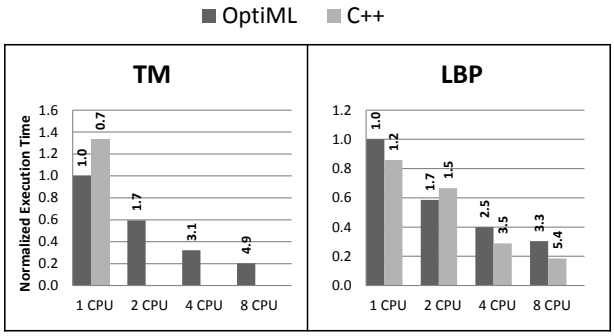


Figure 3.5: Execution time of larger applications compared to C++, and normalized to single-threaded OptiML. Speedup numbers are reported on top of each bar.

OptiML version outperforms the C++ version while being significantly shorter and easier to read. It also scales to 8 cores, while the C++ baseline would require non-trivial manual parallelization to achieve any parallel performance.

Impact of Optimizations Section 3.4.2 described multiple static and dynamic domain-specific optimizations. The performance results presented in Figures 3.4 and 3.5 included the static optimizations of common subexpression elimination, dead code elimination, code motion, and linear algebra rewrites. We studied the benefits of op fusing on the downsampling part of a bioinformatics application. This part of the application streams over a large dataset, performing multiple operations on each sample. Without optimization, the OptiML version is 3x slower than a hand-optimized, manually-parallelized C++ version, as shown in Figure 3.6. After fusing, the OptiML version is approximately as fast as the C++ version.

We next look at the additional improvement from applying relaxed dependencies to SVM and best-effort computation to k-means. The SMO implementation of SVM contains inter-loop dependencies that prevent parallelization across iterations. In previous work we have shown that relaxing dependencies between the outer loop iterations, allowing iterations to sometimes run in parallel, can increase performance by 1.8x with less than 1% loss in classification accuracy [21]. For k-means we demonstrated that a best-effort convergence policy that drops distance calculations that

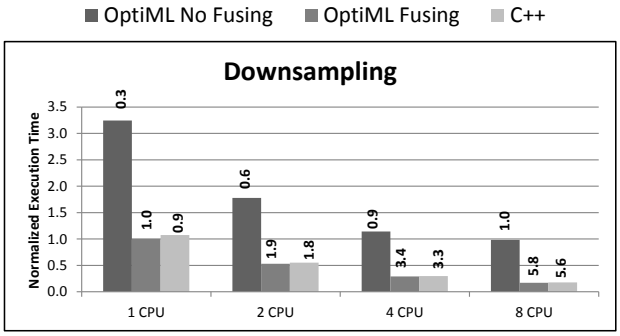


Figure 3.6: Normalized execution time of Downsampling in C++ and OptiML with and without op-fusing optimizations. Speedup numbers are reported on top of each bar.

have remained unchanged in the previous n iterations creates a unique tradeoff between accuracy and performance for different values of n. Specifically, we observed speedups of 1.8x with a 1.2% loss in accuracy, 4.9x with a 4.2% loss, and 12.7x with a 7.4% loss in accuracy [21].

3.5.4 MSMBuilder: Accelerating the Analysis of Protein Folding Simulations

We conclude our evaluation with a case study of OptiML applied to a real-world analysis problem. MSMBuilder [10] is an open source software package for accelerating molecular dynamics simulations and gaining new insights from the results by building Markov State Models (MSMs). MSMs are a powerful means of modeling the structure and dynamics of molecular systems, like proteins. For this study, we ported the performance-critical clustering portion of the MSMBuilder application to OptiML. The most performance intensive component computes the root mean squared distance (RMSD) between two vectors using a an efficient quaternion-based characteristic polynomial algorithm [119]. Due to the large and increasing size of simulation data, the MSMBuilder application has long running times and is limited by the performance of the RMSD calculation.

The original version was implemented in a combination of Python (most of the application) and optimized C++ (performance-critical part). We replaced both of

these code sections with a single OptiML application. The C++ code was expert-optimized to get the best performance possible. As a consequence, other students in the research lab (besides the original author) were unable to read, maintain, or modify this code. The risk of breaking functionality and the long running times that resulted form switching to less optimized code hindered innovation, since it became difficult to prototype new algorithms. To illustrate the issues, consider the vectorized (SSE) C++ code for the inner matrix multiplication used in RMSD:

```
// npaddedatoms must be a multiple of 4
int niters = npaddedatoms >> 2;
__m128 xx,xy,xz,yx,yy,yz,zx,zy,zz;
__m128 ax,ay,az,b;
__m128 t0,t1,t2;
// Prologue
xx = _mm_xor_ps(xx,xx); xy = _mm_xor_ps(xy,xy); xz = _mm_xor_ps(xz,xz);
yx = _mm_xor_ps(yx,yx); yy = _mm_xor_ps(yy,yy); yz = _mm_xor_ps(yz,yz);
zx = _mm_xor_ps(zx,zx); zy = _mm_xor_ps(zy,zy); zz = _mm_xor_ps(zz,zz);
for (int k = 0; k < niters; k++) {
    ax = _mm_load_ps(aTx); ay = _mm_load_ps(aTy); az = _mm_load_ps(aTz);
    b = _mm_load_ps(bTx);
    t0 = ax; t1 = ay; t2 = az;
    t0 = _mm_mul_ps(t0,b);
    t1 = _mm_mul_ps(t1,b);
    t2 = _mm_mul_ps(t2,b);
    // ... more vector instructions
}
// Epilogue - reduce 4 wide vectors to one wide
#ifdef __SSE3__
    xx = _mm_hadd_ps(xx,xy);
    xz = _mm_hadd_ps(xz,yx);
    yy = _mm_hadd_ps(yy,yz);
    zx = _mm_hadd_ps(zx,zy);
    zz = _mm_hadd_ps(zz,zy);
```

```
    xx = _mm_hadd_ps(xx,xz);
    yy = _mm_hadd_ps(yy,zx);
    zz = _mm_hadd_ps(zz,xz);
#else
    // Emulate horizontal adds using UNPCKLPS/UNPCKHPS
    t0 = xx;
    t1 = xx;
    t0 = _mm_unpacklo_ps(t0,xz);
    // ...
```

Furthermore, the integration of C++ code into the Python program required convoluted wrappers:

```
if (!PyArg_ParseTuple(args,"iiiOOOf",&nrealatoms,&npaddedatoms,&rowstride,
                      &ary_coorda,&ary_coordb,&ary_Ga,&G_y)) {
  return NULL;
}
// Get pointers to array data
AData  = (float*) PyArray_DATA(ary_coorda);
BData  = (float*) PyArray_DATA(ary_coordb);
GAData  = (float*) PyArray_DATA(ary_Ga);
// Get dimensions of arrays (# molecules, maxlingos) (Note: strides are in BYTES, not INTs)
arrayADims = PyArray_DIMS(ary_coorda); arrayAStrides = PyArray_STRIDES(ary_coorda);
arrayBDims = PyArray_DIMS(ary_coordb); arrayBStrides = PyArray_STRIDES(ary_coordb);
// Make sure arrays are of float32 data type
CHECKARRAYTYPE(ary_coorda,"Array A");
CHECKARRAYTYPE(ary_coordb,"Array B");
// ...
```

This wrapper code provides the plumbing to convert Python types to their C++ counterparts, and performs some basic error checking on the hand-off. In contrast, we can write the entire RMSD calculation as a few lines of high-level OptiML code:

```
// mX and mY are input matrices
// automatically vectorized matrix multiply
val M = mX * mY
```

```
// form the 4x4 symmetric Key matrix K
val k00 = M(0,0) + M(1,1) + M(2,2); val k01 = M(1,2)-M(2,1); val k02 = M(2,0)-M(0,2)
val k03 = M(0,1)-M(1,0); val k11 = M(0,0)-M(1,1)-M(2,2);     val k12 = M(0,1)+M(1,0)
val k13 = M(2,0)+M(0,2); val k22 = -1f*M(0,0)+M(1,1)-M(2,2); val k23 = M(1,2)+M(2,1)
val k33 = -1f*M(0,0) - M(1,1) + M(2,2)
val K = DenseMatrix((k00, k01, k02, k03),
                    (k01, k11, k12, k13),
                    (k02, k12, k22, k23),
                    (k03, k13, k23, k33))

// coefficients of the characteristic polynomial
val c2 = -2f*square(M).sum
val c1 = -8f*det(M)

// 4x4 determinant of the K matrix
val c0 = det(K)

// iterate newton descent
val linit = (gx + gy) / 2.0
val lambda =
  untilconverged(linit) { lambda =>
    val l2 = lambda*lambda
    val b = (l2 + c2)*lambda
    val a = b + c1
    lambda - (a * lambda + c0) / (2.0*lambda*l2 + b + a)
  }

val rmsd2 = (gx + gy - 2.0 * lambda) / realLen
if (rmsd2 > 0) sqrt(rmsd2).toFloat else 0f
```

In all, we were able to replace 542 lines of combined Python/C++ with 329 lines of OptiML code (a 40% savings). Furthermore, the OptiML code retains the high-level

expressiveness of the original Python, but without any of the complex C++ core.

Figure 3.7 compares the performance of various systems computing the RMSD on a set of randomly generated input vectors. "MSMBuilder" is the original expert-optimized C++ version [10]. At the other end of the spectrum, we show the result of an unoptimized Python implementation using NumPy [62], which is far more readable but orders of magnitude slower. In contrast, our compiler starts from high-level code but generates highly efficient code that resembles the hand-optimized C++ and achieves nearly comparable performance.

The OptiML program represents the simulation data as a `DenseVector` of 3-d coordinates (x,y,z), but the array-of-struct to struct-of-array transformation performed by Delite allows this to be stored efficiently as three primitive arrays. Fusion (both horizontal and vertical) packs computation into tight loops, eliminating unnecessary allocations. Finally, we implemented a vectorized code generator for `c x k * k x c` matrix multiplication, where `c` is small and constant. This is an example where we can generate more efficient code than BLAS by extending the DSL compiler with a specialization for a particular input class. This kind of specialization is a distinctive feature of Delite DSLs. The extension itself (in this case, code generator) is simple to add (compared to traditional compilers) because it is written as high-level Scala operating on a high-level IR. Furthermore, once we have encoded this new domain knowledge, other OptiML applications can benefit from it transparently.[3]

The small difference between our performance and the MSMBuilder version is mainly attributable to OptiML's simpler determinant implementation (the C++ computes a 3x3 matrix determinant as a Laplacian expansion) and overhead in Delite. For example, our generated code also contains a large number of stack variables due to the single static assignment (SSA) form used by our compiler, which can add a small overhead if not optimized by the target compilers (e.g. icc). Due to the extreme level of optimization in the MSMBuilder version, we also show a more standard C++ version using BLAS instead of manual SSE; our auto-generated code substantially

[3]As this dissertation has argued, one goal is to make extending the compiler as easy as extending libraries. Still, we don't expect application developers to do this. We propose that in most domains, additional domain knowledge incorporated over time will cover the majority of use cases, rather than requiring a new rule for every application.

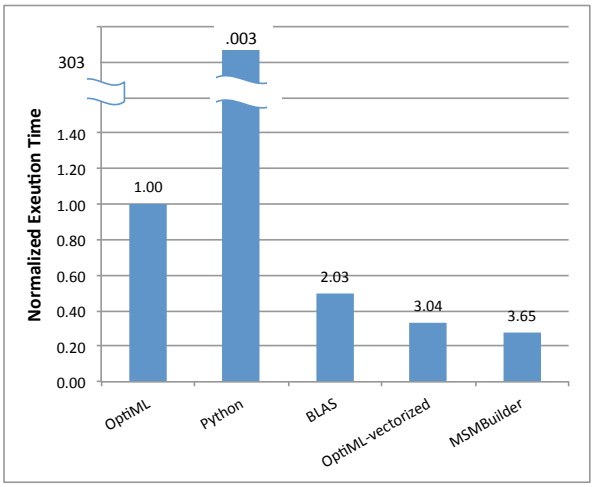


Figure 3.7: Execution time for RMSD microbenchmark across different systems, normalized to parallelized OptiML. Speedup numbers are reported on top of each bar.

outperforms this version.

As we have shown, we are able to achieve competitive performance on multicore CPUs using our DSL compiler by leveraging common optimizations and specialized code generators. However, we can now go one step further: we can automatically generate CUDA code using Delite's built-in code generator without any additional application developer effort. Figure 3.8 shows the result. Since the critical RMSD section is arithmetically intense, it benefits from GPU acceleration. Other portions of the MSMBuilder application that are not as performance-critical can still be run on the CPU via the Delite runtime.

Overall, this case study shows that by leveraging the structure in our abstract model, it is possible to achieve extremely fast performance on the computational core of a real-world application without sacrificing readability or programmer productivity. Starting from code at the same or higher-level than numeric Python, we can generate GPU code that is more than 2x faster than a manually-optimized C++ implementation that took months to develop.

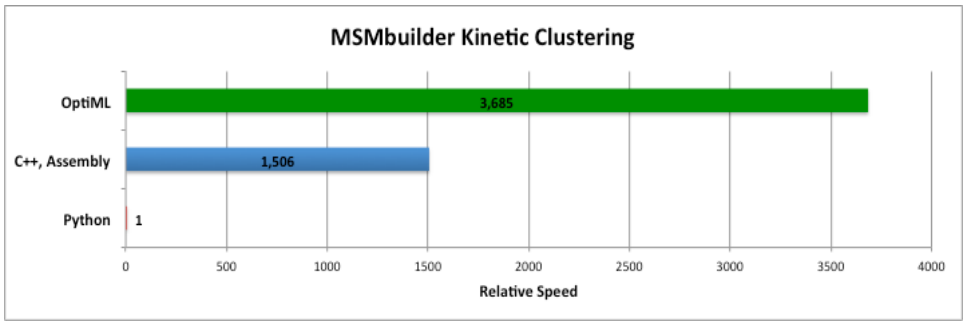


Figure 3.8: Speedup of OptiML GPU on MSMBuilder compared to Python and C++.

3.6 Summary

In this chapter, we presented OptiML, a DSL for machine learning, and the first Delite DSL. OptiML enables machine learning researchers and developers to program at a high level and still exploit parallel hardware (multicore and GPUs). This is particularly important as the size of datasets continues to grow and many useful machine learning algorithms are facing computational challenges.

We demonstrated the productivity of OptiML code with several application examples. We have shown that the OptiML compiler can perform domain-specific optimizations and (via Delite) generate efficient parallel code for heterogeneous devices without exposing any parallelism or device details to OptiML users. We presented experimental results showing that OptiML code outperformed explicitly parallelized MATLAB code on a heterogeneous system consisting of multicore CPUs and a GPU. Furthermore, we showed that OptiML code can even perform comparably to optimized C++ code in real applications.

OptiML shows that Delite DSLs are a feasible approach to highly productive, high performance programming. However, key questions remain. Delite's components enabled OptiML to achieve high performance, but are they a good match for other domains? How can we compose multiple high performance DSLs together into a single application? In the next chapter, we investigate these questions.

Chapter 4

Composition and Reuse with Multiple DSLs

In this chapter, we turn our attention to multiple DSLs executing in a single environment. We first explore techniques for interoperating with multiple DSLs within one program. Next, we implement new DSLs for different domains to study the incremental cost of developing a new DSL in Delite and to validate that Delite provides high performance in domains other than machine learning.

4.1 Introduction

The limited scope of DSLs, while an advantage for high-level compilation, is also a barrier to widespread adoption. Many applications contain a mix of problems in different domains and developers need to be able to compose solutions together without sacrificing performance. In addition, DSLs need to interoperate with the outside world in order to enable developers to fall-back to general purpose languages for non performance-critical tasks.

As previous chapters have described, our goal is to develop compiled embedded

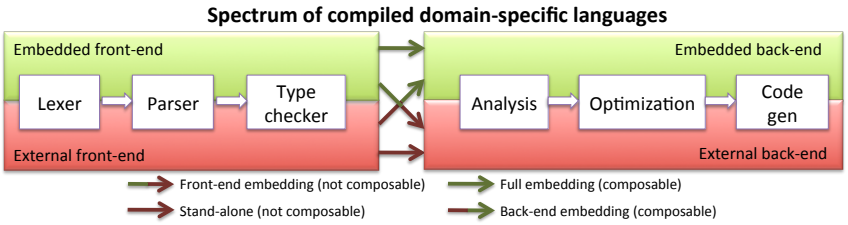


Figure 4.1: Major phases in a typical compiler pipeline and possible organizations of compiled DSLs. Front-end embedding in a host language (and compiler) is common, but for composability, back-end embedding in a host compiler (i.e. building on top of an extensible compiler framework) is more important.

DSLs that resemble self-optimizing libraries with domain-specific front-ends. However, we propose that to build high-performance composable DSLs, *back-end embedding* is more important than the *front-end embedding* of traditional compiled embedded DSLs. Figure 4.1 illustrates this distinction. We define back-end embedding as a compiled DSL that inherits, or extends, the latter portion of the compiler pipeline from an existing compiler. Such a DSL can either programmatically extend the back-end compiler or pass programs in its intermediate representation (IR) to the back-end compiler. By embedding multiple DSLs in a common back-end, they can be compiled together and co-optimized. The fact that many compiled DSLs target C or LLVM instead of machine code and thus reuse instruction scheduling and register allocation can be seen as a crude example of back-end embedding. However, the abstraction level of C is already too low to compose DSLs in a way that allows high-level cross-DSL optimizations, parallelization, or heterogeneous code generation. Just as some general purpose languages are better suited for front-end embedding than others, not all compiler frameworks or target languages are equally suited for back-end embedding.

In this chapter, we show that we can compose compiled DSLs embedded in a common, high-level backend (Delite) and use them together in a single application. Our approach allows DSLs to build on top of one another and reuse important generic optimizations, rather than reinventing the wheel each time. Optimizations can even be applied across different DSL blocks within an application. The addition of composablity and re-use across compiled DSLs pushes them closer to libraries in terms of development effort and usage while still retaining the performance characteristics of

stand-alone DSLs. In other words, we regain many of the benefits of purely embedded DSLs that were lost when adding compilation. The previous chapter demonstrated good performance for OptiML, a DSL for machine learning [112]. However, so far we have not addressed how to compose different DSLs together, or shown that similar performance and productivity gains can be obtained for other domains. We present new compiled embedded DSLs for data querying (OptiQL), collections (OptiCollections), graph analysis (OptiGraph), and mesh computation (OptiMesh). We show that the DSLs were easier to build than stand-alone counterparts, can achieve competitive performance, and can also be composed together in multiple ways.

The source code for the new DSLs we have developed is open-source and freely available at: `http://github.com/stanford-ppl/Delite/`.

4.2 High-level Common Intermediate Representation

To achieve high performance for an application composed out of multiple DSLs, each DSL must provide competitive performance, not incur high overhead when crossing between DSL sections, and be co-optimizable. Moreover, for this sort of composition to be a practical approach, DSLs targeting narrow problem domains and modern hardware should be as easy to construct as optimized libraries (or as close as possible). One way of achieving these goals is through a common intermediate representation containing reusable, high level computation and data primitives with a minimal set of restrictions that enable efficiency and optimization. In this section we describe how the Delite IR meets this criteria and propose a simple data exchange format for embedded DSLs.

4.2.1 Structured Computation

In order to optimize composed DSL blocks, they must share, or be transformable to, a common intermediate representation (IR) at some level. This level should retain enough high-level semantics to allow coarse-grained optimization and code generation; once an operation has been lowered to low-level primitives such as instructions over

scalars, it is as difficult to target heterogeneous hardware as in a general purpose language. If DSLs do not share a common IR at some level, the best we can do is attempt to compose their generated code, which is a low-level and error-prone strategy that does not allow co-optimization.

In Chapter 2.3.1 we proposed parallel patterns as a base IR that is well-suited to optimization and code generation for different DSLs and heterogeneous devices [14]. The new DSLs presented in this paper offer more evidence that parallel patterns are a strong choice of base IR. Previously, Delite supported `Sequential`, `Loop`, `Map`, `Reduce`, `ZipWith`, `Foreach`, and `Filter` patterns. To support the new DSLs, we added support for `GroupBy`, `Sort`, `ForeachReduce` (foreach with global reductions), and `FlatMap` patterns. Using the patterns as a base IR allows us to perform *op fusion* automatically for each DSL, which is a key optimization when targeting data parallel hardware such as GPUs.

4.2.2 Structured Data

In the same way that a common, structured IR allows DSL operations to be optimized and code generated in a uniform way, a common data representation is also required at some level. In addition to enabling the internal representation of DSL data structures to be reusable, a common representation facilitates communication between DSL blocks and enables pipeline optimizations; a DSL can directly consume the output of another DSL's block, so we can (for example) fuse the first DSL's operation that constructs the output with the second DSL's operation that consumes it.

We use structs of a fixed set of primitives as the common data representation (Chapter 2.3.2). Delite DSLs still present domain-specific types to end users (e.g. `Vector`) and methods on instances of those types (e.g. `+`) get lifted into the IR. However, the back-end data structures must be implemented as structs (for example, `Vector` can be implemented with an `Int` length field and an `Array` data field). The fields currently allowed in a Delite `Struct` are: numerics (e.g. `Int`), `Boolean`, `String`, `Array`, `HashMap`, or other `Structs`.

Agreeing on a simple set of high level data primitives allows DSLs to communicate

efficiently in-memory (even across multiple device and address spaces). It also allows Delite to manage communication transparently for all DSLs, utilizing this common layer. As we will see, this simple set is still flexible enough to implement real DSLs in diverse domains.

4.2.3 Data Exchange Format

The final piece required for composability is the ability for application developers to convert from domain-specific data types to common data types in order to communicate between DSL blocks. This is necessary because we do not want to expose the internal representation of the domain-specific types to users.

A simple solution that we use is for DSL authors to optionally implement methods `to/from{Primitive}` between each DSL data type and a corresponding primitive type. For example, a DSL author would provide `toArray` and `fromArray` methods on DSL types where this makes sense (e.g. `Vector`), or `toInt` and `fromInt` for a type `NodeId`. These methods become part of the DSL specification and enable application developers to export and import DSL objects depending on their needs. For example, a graph could be exported using a snippet like:

```
// G: Graph
// returns an array of node ids and an array of edge ids
new Record { val nodes = G.nodes.map(node => node.Id.toInt).toArray
             val edges = G.edges.map(edge => edge.Id.toInt).toArray }
```

or just one of the properties of the graph could be exported using:

```
// G: Graph, numFriends: NodeProperty[Int]
// returns an array of ints corresponding to the number of friends of each node
G.nodes.map(node => numFriends(node)).toArray
```

We consider in the next section how to actually compose snippets together. Assuming this facility, though, importing is similar. For example, a vector could be constructed from the output of the previous snippet using:

```
// numFriends: Array[Int]
val v = Vector.fromArray(numFriends)
```

The key aspect of the data exchange format is that it should not prevent optimization across DSL blocks or impose substantial overhead to box and unbox the results. We handle this by implementing the `to/from` functions as either scalar conversions or loops in the common IR. Then, for example, a loop constructing an array will be automatically fused together with the loop consuming the array, resulting in no overhead while still ensuring safe encapsulation. The loop fusion algorithm is described in previous work [103].

4.3 Methods for Composing Compiled DSLs

In this section, we describe two ways of composing compiled DSLs that are embedded in a common back-end. The first way is to combine DSLs that are designed to work with other DSLs, i.e. DSLs that make an "open-world" assumption. The second way is to compose "closed-world" DSLs by compiling them in isolation, lowering them to a common, high-level representation, recombining them and then optimizing across them. Both methods rely on DSLs that share a common high-level intermediate representation as described in the previous section.

4.3.1 Open-world: Fine-grained Cooperative Composition

"Open-world" composition means essentially designing embedded DSLs that are meant to be layered or included by other embedded DSLs as modules. For this kind of composition to be feasible, all of the DSLs must be embedded in the same language and framework in both the front-end and the back-end.

Once embedded in a common host environment, DSL composition reduces to object composition in the host language. This is the classic "modularity" aspect of DSLs built using LMS [101]. For example, consider the following DSL definition for our toy `SimpleVector` DSL:

```
trait SimpleVector extends Base with MathOps with ScalarOps with VectorOps
```

The DSL is composed of several traits that contain different DSL data types and operations. Each trait extends `Base` which contains common IR node definitions. A

different DSL author can extend `SimpleVector` to create a new DSL, `SimpleMatrix`, simply by extending the relevant packages:

```
trait SimpleMatrix extends SimpleVector with MatrixOps with ExtVectorOps
```

Each of the mixed-in traits represents a collection of IR node definitions. Traits that contain optimizations and code generators can be extended in the same way. These traits define `SimpleMatrix` as a superset of `SimpleVector`, but `SimpleMatrix` users only interact with `SimpleMatrix` and do not need to be aware of the existence of `SimpleVector`. Furthermore, since the composition is via inheritance, the `SimpleMatrix` DSL can extend or overload operations on the `Vector` data type, e.g. inside `ExtVectorOps`. Since `SimpleVector` is encapsulated as a separate object, it can be reused by multiple DSLs.

Open-world DSLs can also be composed by application developers. For example, suppose we also have a visualization DSL:

```
trait Viz extends Base with GraphicsOps with ChartOps with ImageOps
```

A Scala program can effectively construct a new DSL on the fly by mixing multiple embedded DSLs together:

```
trait MyApp extends SimpleMatrix with Viz {
  def main() {
    // assuming relevant DSL functions
    val m = Matrix.rand(100); display(m.toArray)
  }
}
```

In this example we make use of the data exchange format described in Section 4.2.3 in order to communicate data between the DSLs. When DSL users invoke a `Viz` operation, that operation will construct a `Viz` IR node. Note that after mix-in, the result is effectively a single DSL that extends common IR nodes; optimizations that operate on the generic IR can occur even between operations from different DSLs. This is analogous to libraries building upon other libraries, except that now optimizing compilation can also be inherited. DSLs can add analyses and transformations that are designed to be included by other DSLs. The trade-off is that DSL authors and application developers must be aware of the semantics they are composing and are

responsible for ensuring that transformations and optimizations between DSLs retain their original semantics. Namespacing can also be a pitfall; DSL traits cannot have conflicting object or method names since they are mixed in to the same object. We avoid this problem by using conventions like DSL-specific prefixes (e.g. `viz_display`).

4.3.2 Closed-world: Coarse-grained Isolated Composition

As the previous section pointed out, there are issues with simply mixing embedded DSLs together. In particular, some DSLs require restricted semantics in order to perform domain-specific analyses and transformations that are required for correctness or performance. These "closed-world" DSLs are not designed to arbitrarily compose with other DSLs. Furthermore, any DSL with an external front-end is necessarily closed-world, since the parser only handles that DSL's grammar. However, it is still possible to compose closed-world DSLs that share a common back-end. Either they are implemented in the same language as the back-end and programmatically interface with it (as with Scala and Delite), or they target a serialized representation (such as source code) that is the input to a common back-end. This kind of coarse grained composition has three steps:

1. Independently parse each DSL block and apply domain-specific optimizations
2. Lower each block to the common high-level IR, composing all blocks into a single intermediate program
3. Optimize the combined IR and generate code

We discuss how we implemented these steps in Delite next.

Scopes: independent compilation DSLs that have an external front-end can be independently compiled by invoking the DSL parser on a string. However, in order to independently compile DSL blocks that are embedded in a host language, we need a coarse-grained execution block. We have modified the Scala-Virtualized compiler to add `Scopes` for this purpose. A `Scope` is a compiler-provided type that acts as a tag to

```
SimpleVector {
  val v = Vector.rand(100)
  // ...
}
```

(a) Scala code with DSL scope

```
abstract class DSLprog extends SimpleVectorApp {
  def apply = {
    val v = Vector.rand(100)
    // ...
  }
}
(new DSLprog with SimpleVectorCompiler).result
```

(b) Scala code after desugaring

encapsulate DSL blocks. For example, using the `SimpleVector` DSL, we can instantiate a `Scope` as follows:

```
def SimpleVector[R](b: => R) = new Scope[SimpleVectorApp, SimpleVectorCompiler, R](b)
```

`SimpleVectorApp` and `SimpleVectorCompiler` are Scala traits that define the DSL interface and its implementation, respectively. The Scala-Virtualized compiler transforms function calls with return type `Scope` into an object that composes the two traits with the given block, making all members of the DSL interface available to the block's content. The implementation of the DSL interface remains hidden, however, to ensure safe encapsulation. The object's constructor then executes the block. The result is that each `Scope` is staged and optimized independently to construct the domain-specific IR.

Given the previous definition, a programmer can write a block of `SimpleVector` DSL code inside a Scala application, which then gets desugared, making all member definitions of `SimpleVectorApp`, but not of `SimpleVectorCompiler`, available to the `Scope`'s body:

Lowering and composing The ability to compile blocks of DSL code into independent IRs is the first step, but in order to compose multiple blocks in a single application we still need a way to communicate across the blocks and a way to combine the IRs. Consider the following application snippet:

```
val r = SimpleVector {
  val (v1,v2) = (Vector.rand(100),Vector.rand(100))
```

```
  DRef(linreg(v1,v2).toArray) // return linear regression of v1, v2
}
Viz { display(r.get) }
```

We again use the `toArray` functionality to export the data from `SimpleVector` into a common format that `Viz` can handle. However, before we lower to a common representation, the type of the output of `SimpleVector` is a symbol with no relation to `Viz`. Therefore, we introduce the path independent type `DRef[T]` to abstract over the path dependent scope result. During staging, `r.get` returns a placeholder of type `Rep[DeliteArray[Double]]`. When the IRs are lowered and stitched together, the placeholder is translated to the concrete symbolic result of `linreg(v1,v2).toArray`. This mechanism is type-safe, preserves scope isolation, and does not occlude optimizations on the lowered IR.

After performing domain-specific optimizations and transformations, the IR for each scope is lowered to the base IR in a language-specific fashion. We use staging to perform this translation by extending our previous work on staged interpreters for program transformation [103] to support transforming IR nodes to an arbitrary target type. A `Transformer` is a generic `Traversal` that maps symbolic IR values (type `Exp[A]`) to values of type `Target[A]`, where `Target` is an abstract type constructor. During the traversal, a callback `transformStm` is invoked for each statement encountered.

The extended `Transformer` interface for cross-DSL transformation is [1]:

```
trait Transformer extends Traversal {
  import IR._
  type Target[A]
  var subst = immutable.Map.empty[Exp[Any], Target[Any]]
  def apply[A](x: Exp[A]): Target[A] = ...    // lookup from subst
  override def traverseStm(stm: Stm): Unit =  // called during traversal
    subst += (stm.sym -> transformStm(stm))   //   update substitution with result
  def transformStm(stm: Stm): Target[Any]     // to be implemented in subclass
}
```

[1] *Credits:* Design by Tiark Rompf.

To transform from one IR to another, lower-level IR language, we instantiate a transformer with type `Target` set to the `Rep` type of the destination IR:

```
trait IRTransformer extends Transformer {
  val dst: IR.DST
  type Target[A] = IR.Rep[A]
  def transformStm(stm: Stm): Target[Any] =   // override to implement custom transform
    IR.mirror(stm.rhs, this)                  //   call default case
}
```

The type of the destination IR `dst: IR.DST` is constrained by the source IR to handle all defined IR nodes. This enables implementing a default case for the transformation (**def** `mirror`), which maps each source IR node to the corresponding smart constructor in the destination IR.

Taking the `SimpleVector` DSL as an example, we define:

```
trait SimpleVector extends Base {
  def vector_zeros(n: Rep[Int]): Rep[Vector[Double]]
}
trait SimpleVectorExp extends SimpleVector with BaseExp {
  type DST <: SimpleVector
  case class VectorZeros(n: Rep[Int]) extends Def[Vector[Double]]
  def vector_zeros(n: Rep[Int]): Rep[Vector[Double]] = VectorZeros(n)
  def mirror[A:Manifest](e: Def[A], f: IRTransformer): f.dst.Rep[A] = {
    case VectorZeros(n) => f.dst.vector_zeros(f(n))
    ...
  }
}
```

The use of Scala's dependent method types `f.dst.Rep[A]` and the upper-bounded abstract type `DST <: SimpleVector` ensure type safety when specifying transformations. Note that the internal representation of the destination IR is not exposed, only its abstract `Rep` interface. This enables, for example, interfacing with a textual code generator that defines `Rep[T] = String`. Perhaps more importantly, this enables programmatic lowering transforms by implementing a smart constructor (e.g. `vector_zeros`)

to expand into a lower-level representation using arrays instead of constructing an IR node that directly represents the high-level operation.

An alternative to using staged interpreters is to simply generate code to a high-level intermediate language that maps to the common IR. We implemented this by generating code to the "Delite IL", a low-level API around Delite ops, that is itself staged to build the Delite IR. For example, a `Reduce` op in the original application would be code generated to call the method

```
reduce[A](size: Rep[Int], func: Rep[Int] => Rep[A], cond: List[Rep[Int]] => Rep[Boolean],
        zero: => Rep[A], rFunc: (Rep[A],Rep[A]) => Rep[A])
```

in the Delite IL.

The staged interpreter transformation and the code generation to the Delite IL perform the same operation and both use staging to build the common IR. The staged interpreter translation is type-safe and goes through the heap, while the Delite IL goes through the file system and is only type-checked when the resulting program is (re-)staged. On the other hand, for expert programmers, a simplified version of the Delite IL may be desirable to target directly.

Cross DSL optimization After the IRs have been composed, we apply all of our generic optimizations on the base IR (i.e. parallel patterns). Like in the open-world case, we can now apply optimizations such as fusion, common subexpression elimination (CSE), dead code elimination (DCE), dead field elimination (DFE), and AoS to SoA conversion across DSL snippets. Since the base IR still represents high level computation, these generic optimizations still have much higher potential impact than their analogs in a general purpose compiler. Fusion across DSL snippets is especially useful, since it can eliminate the overhead of boxing and unboxing the inputs and outputs to DSL blocks using the `to/from{Primitive}` data exchange format. The usefulness of applying these optimizations on composed blocks instead of only on individual blocks is evaluated in Section 5.5. Note that the cross-DSL optimizations fall out for free after composing the DSLs; we do not have to specifically design new cross-DSL optimizations.

4.3.3 Interoperating with non-DSL code

The previous section showed how we can compose coarse-grain compiled DSL blocks within a single application. However, it is also interesting to consider how we can compose DSL blocks with arbitrary host language code. We can again use `Scope`, but with a different implementation trait, to accomplish this. Consider the following, slightly modified definition of the `SimpleVector` scope:

```
def SimpleVector[R](b: => R) = new Scope[SimpleVectorApp, SimpleVectorExecutor, R](b)
```

Whereas previously the `SimpleVector` scope simply compiled the input block, the trait `SimpleVectorExecutor` both compiles and executes it, returning a Scala object as a result of the execution. `SimpleVectorExecutor` can be implemented by programmatically invoking the common back-end on the lowered IR immediately. This enables us to use compiled embedded DSLs within ordinary Scala programs:

```
def main(args: Array[String]) {
  foo() // Scala code
  SimpleVector { val v = Vector.ones(5); v.pprint }
  // more Scala code ..
}
```

This facility is the same that is required to enable interactive usage of DSLs using the REPL of the host language, which is especially useful for debugging. For example, we can use the new `SimpleVector` scope to execute DSL statements inside the Scala-Virtualized REPL:

```
scala> SimpleVector { val v = Vector.ones(5); v.pprint }
[ 1 1 1 1 1 ]
```

4.4 New Compiled Embedded DSL Implementations

We implemented four new high performance DSLs embedded inside Scala and Delite. In this section, we briefly describe each DSL and show how their implementation was simplified by reusing common components. The four new DSLs are OptiQL, a DSL for

data querying, OptiCollections, an optimized subset of the Scala collections library, OptiGraph, a DSL for graph analysis based on Green-Marl [53] and OptiMesh, a DSL for mesh computations based on Liszt [35]. Despite being embedded in both a host language and common back-end, the DSLs cover a diverse set of domains with different requirements and support non-trivial optimizations.

4.4.1 OptiQL

OptiQL [2] is a DSL for data querying of in-memory collections, and is heavily inspired by LINQ [79], specifically LINQ to Objects. OptiQL is a pure language that consists of a set of implicitly parallel query operators, such as `Select`, `Average`, and `GroupBy`, that operate on OptiQL's core data structure, the `Table`, which contains a user-defined schema. Listing 4.1 shows an example snippet of OptiQL code that expresses a query similar to Q1 in the TPC-H benchmark. The query first excludes any line item with a ship date that occurs after the specified date. It then groups each line item by its status. Finally, it summarizes each group by aggregating the group's line items and constructs a final result per group.

Since OptiQL is SQL-like, it is concise and has a small learning curve for many developers. However, unoptimized performance is poor. Operations always semantically produce a new result, and since the in-memory collections are typically very large, cache locality is poor and allocations are expensive. OptiQL uses compilation to aggressively optimize queries. Operations are fused into a single loop over the dataset wherever possible, eliminating temporary allocations, and datasets are internally allocated in a column-oriented manner, allowing OptiQL to avoid allocating columns that are not used in a given query. Although not implemented yet, OptiQL's eventual goal is to use Delite's pattern rewriting and transformation facilities to implement other traditional (domain-specific), cost-based query optimizations.

Reuse The major operations in OptiQL are all data-parallel and map to the Delite framework as follows: `Where` → `Filter`; `GroupBy` → `GroupBy`; `Select` → `Map`; `Sum` →

[2] *Credits*: Hassan Chafi and Kevin Brown are the primary authors of OptiQL.

```
1 // lineItems: Table[LineItem]
2 val q = lineItems Where(_.l_shipdate <= Date(''1998-12-01'')).
3 GroupBy(l => (l.l_linestatus)) Select(g => new Record {
4   val lineStatus = g.key
5   val sumQty = g.Sum(_.l_quantity)
6   val sumDiscountedPrice = g.Sum(l => l.l_extendedprice*(1.0-l.l_discount))
7   val avgPrice = g.Average(_.l_extendedprice)
8   val countOrder = g.Count
9 }) OrderBy(_.returnFlag) ThenBy(_.lineStatus)
```

Listing 4.1: OptiQL: TPC-H Query 1 benchmark

`Reduce`; `Average` → `Reduce`; `Count` → `Reduce`; `OrderBy` → `Sort`; `Join` → `GroupBy`; `FlatMap`. Since the OptiQL operations map to Delite ops, Delite automatically handles parallel code generation. Furthermore, since OptiQL applications consist mainly of sequences of pure data parallel operations (with potentially large intermediate results, e.g. a `Select` followed by a `Where`), OptiQL benefits substantially from op fusion and inherits this optimization from Delite. OptiQL's `Table` and user-defined schemas are implemented as Delite `Structs`; Delite's array-of-struct to struct-of-array transformation allows OptiQL to provide a row-store abstraction while actually implementing a back-end column store. Delite `Structs` also automatically provide *dead field elimination* (the elimination of struct fields that are not used in the query) for free.

New Delite features required We implemented the user-defined data structure support discussed in Section 2.3.2 to support lifting user-defined schemas in OptiQL. The `GroupBy` op and corresponding hashmap semantics were added to support `GroupBy`. `FlatMap` was added to support `Join` and `Sort` was added for `OrderBy`.

4.4.2 OptiCollections

Where OptiQL provides a SQL-like interface, OptiCollections [3] is an example of applying the underlying optimization and compilation techniques to the Scala collections library. The Scala collections library provides several key generic data types

[3] *Credits:* Aleksandar Prokopec and Vojin Jovanovic are the primary authors of OptiCollections.

(e.g. `List`) with rich APIs that include expressive functional operators such as `flatMap` and `partition`. The library enables writing succinct and powerful programs, but can also suffer from overheads associated with high-level, functional programs (especially the creation of many intermediate objects). OptiCollections uses the exact same collections API, but uses Delite to generate optimized, low-level code. Most of the infrastructure is shared with OptiQL. The prototype version of OptiCollections supports staged versions of Scala's `Array` and `HashMap`. Listing 4.2 shows an OptiCollections application that consumes a list of web pages and their outgoing links and outputs a list of web pages and the set of incoming links for each of the pages (i.e. finds the reverse web-links). In the first step, the `flatMap` operation maps each page to pairs of an outgoing link and the page. The `groupBy` operation then groups the pairs by their outgoing link, yielding a `HashMap` of pages, each paired with the collection of web pages that link to it.

The example has the same syntax as the corresponding Scala collections version. A key benefit of developing OptiCollections is that it can be mixed in to enrich other DSLs with a range of collection types and operations on those types. It can also be used as a transparent, drop-in replacement for existing Scala programs using collections and provide improved performance.

Reuse Scala collections have bulk operators that map exactly to Delite ops (e.g. `map`, `reduce`, `sort`, `filter`). In order to implement `HashMap`, OptiCollections required the same support for the `Hash` op as OptiQL, so this was reused. The other operations in OptiCollections are implemented as Delite `Sequential` ops. Since OptiCollections implements a subset of the Scala library, it relies more on the lifted set of Scala provided by Delite than the other DSLs; conditionals, strings, numerics, while loops and ranges are all reused heavily. OptiCollections also benefits from Delite generic optimizations, especially fusion.

New Delite features required We added `FlatMap` to support the corresponding collections operation; it could have been implemented as a `Map`,`Reduce`, but this is less

```
1 val sourcedests = pagelinks flatMap { l =>
2   val sd = l.split(":")
3   val source = Long.parseLong(sd(0))
4   val dests = sd(1).trim.split(" ")
5   dests.map(d => (Integer.parseInt(d), source))
6 }
7 val inverted = sourcedests groupBy (x => x._1)
```

Listing 4.2: OptiCollections: reverse web-links benchmark

efficient, since it does not make use of a parallel scan to write the output. This example shows that deciding when to make a domain op a new parallel pattern is not always straightforward; it requires parallelism expertise to identify the situation and to implement the new pattern. However, the DSL approach shields the user from this problem, shifting the burden to the DSL and framework authors.

4.4.3 OptiGraph

OptiGraph [4] is a DSL for static graph analysis based on the Green-Marl DSL [53]. OptiGraph enables users to express graph analysis algorithms using graph-specific abstractions and automatically obtain efficient parallel execution. OptiGraph defines types for directed and undirected graphs, nodes, and edges. It allows data to be associated with graph nodes and edges via node and edge property types and provides three types of collections for node and edge storage (namely, `Set`, `Sequence`, and `Order`). Furthermore, OptiGraph defines constructs for `BFS` and `DFS` order graph traversal, sequential and explicitly parallel iteration, and implicitly parallel in-place reductions and group assignments. An important feature of OptiGraph is also its built-in support for bulk synchronous consistency via *deferred assignments*.

Figure 4.2 shows the parallel loop of the PageRank algorithm [88] written in both OptiGraph and Green-Marl. PageRank is a well-known algorithm that estimates the relative importance of each node in a graph (originally of web pages and hyperlinks)

[4] *Credits:* Victoria Popic was the original author of OptiGraph. Kevin Conley, Yonathan Perez and Chris Aberger have all contributed to its ongoing development.

based on the number and page-rank of the nodes associated with its incoming edges (`InNbrs`). OptiGraph's syntax is slightly different since it is embedded in Scala and must be legal Scala syntax. However, the differences are small and the OptiGraph code is not more verbose than the Green-Marl version. In the snippet, `PR` is a node property associating a page-rank value with every node in the graph. The `<=` statement is a deferred assignment of the new page-rank value, `rank`, for node `t`; deferred writes to `PR` are made visible after the **`for`** loop completes via an explicit assignment statement (not shown). Similarly, `+=` is a scalar reduction that implicitly writes to `diff` only after the loop completes. In contrast, `Sum` is an in-place reduction over the parents of node `t`. This example shows that OptiGraph can concisely express useful graph algorithms in a naturally parallelizable way; the `ForeachReduce` Delite op implicitly injects the necessary synchronization into the **`for`** loop.

Reuse OptiGraph uses Delite's `ForeachReduce`, `Filter`, `Map` and `Reduce` ops to implement its parallel operators. `DFS` is implemented sequentially. `BFS` is implemented as a sequence of `ForeachReduce` and `Map`,`Reduce` ops in a while loop; the `ForeachReduce` op processes the current level and the `Map`,`Reduce` ops compute the nodes to be processed in the next level. OptiGraph inherits conditionals, while loops, and arithmetic operations from Delite and benefits from generic optimizations including code motion, DCE and CSE.

New Delite features required We added `ForeachReduce` to support OptiGraph's scalar reductions inside parallel foreaches. Furthermore, this feature is a generic operation supported by other parallel programming models (e.g. OpenMP) and can be reused by other DSLs (e.g. OptiMesh).

4.4.4 OptiMesh

OptiMesh [5] is an implementation of Liszt on Delite. Liszt is a DSL for mesh-based partial differential equation (PDE) solvers [35]. Liszt code allows users to perform iterative computation over mesh elements (e.g. cells, faces). Data associated with

[5] *Credits:* Michael Wu is the primary author of OptiMesh.

```
1 val PR = NodeProperty[Double](G)
2 for (t <- G.Nodes) {
3   val rank = (1-d) / N + d*
4     Sum(t.InNbrs){w => PR(w)/w.OutDegree}
5   diff += abs(rank - PR(t))
6   PR <= (t,rank)
7 }
```

(a) OptiGraph: PageRank

```
1 N_P<Double> PR;
2 Foreach (t: G.Nodes) {
3   Double rank = (1-d) / N + d*
4     Sum(w:t.InNbrs){w.PR/w.OutDegree()};
5   diff += | rank - t.PR |;
6   t.PR <= rank @ t;
7 }
```

(b) Green-Marl: PageRank

Figure 4.2: The major portion of the PageRank algorithm implemented in both OptiGraph and Green-Marl. OptiGraph is derived from Green-Marl, but required small syntactic changes in order to be embedded in Scala.

```
1 for (edge <- edges(mesh)) {
2   val flux = flux_calc(edge)
3   val v0 = head(edge)
4   val v1 = tail(edge)
5   Flux(v0) += flux // possible write conflicts!
6   Flux(v1) -= flux
7 }
```

Listing 4.3: OptiMesh: simple flux computation

mesh elements are stored in external fields that are indexed by the elements. Listing 4.3 shows a simple OptiMesh program that computes the flux through edges in the mesh. The **for** statement in OptiMesh is implicitly parallel and can only be used to iterate over mesh elements. `head` and `tail` are built-in accessors used to navigate the mesh in a structured way. In this snippet, the `Flux` field stores the flux value associated with a particular vertex. As the snippet demonstrates, a key challenge with OptiMesh is to detect write conflicts within for comprehensions given a particular mesh input.

Reuse The major parallel operation in OptiMesh is the **for** statement. OptiMesh also allows scalar reductions inside a **for** comprehension, so it uses the `ForeachReduce` op that was added for OptiGraph. `Map`, `Reduce` and `ZipWith` are also used for vector and matrix arithmetic. OptiMesh reuses the same generic optimizations as OptiGraph and inherits variables, conditionals and math functions (including trigonometric functions)

from Delite.

New Delite features required OptiMesh's only previously unsupported requirement was the `ForeachReduce` op, which was added for OptiGraph. Therefore, OptiMesh required no additional features; this is an example of how the incremental cost of new DSLs decreases as the existing base of reusable components increases.

Domain-specific transformations OptiMesh implements the same analyses and transformations as the stand-alone Liszt language. First, we perform *stencil detection* [35]. The idea is to symbolically evaluate an OptiMesh program with a real mesh input and record fields that are read and written in a single iteration of a parallel foreach. Using the stencil, we build an interference graph and run a coloring algorithm to find disjoint sets of the original foreach that are safe to run in parallel. We implement stencil detection in Delite before code generation by constructing a new traversal, `StencilCollector`. `StencilCollector` visits each node in program order and looks for a `Foreach` op; when it finds one, it records that it is inside a `Foreach` and begins to record subsequent ops to an in-memory data structure representing the stencil. After the `StencilCollector` traversal is finished, OptiMesh has the stencil.

The next step is to use the stencil to perform the actual transformations. We accomplish this by implementing a new transformer, `ColoringTransformer`, shown in Listing 4.4. `ColoringTransformer` replaces each `Foreach` in the original IR with one or more new `Foreach`es, each corresponding to a different colored set (essentially *loop fission*). After the transformation completes, control is passed back to Delite and code generation proceeds as normal. This example demonstrates that Delite transformations are concise and expressive (the transformed version is written as normal source code, and then re-staged to an IR). It also shows that unstaged code and staged code can be mixed within a transformation; this flexibility is important when using analysis results (static expressions) within transformations (lifted expressions), as OptiMesh does.

```
8  /* This trait is part of the IR and constructs new nodes */
9  trait ColoringTransformerExp extends OptiMeshExp { self =>
10   // result is a symbol representing multiple Foreach ops
11   def colorFor(f: Foreach[_], colors: Array[Array[Int]]) = {
12     // colors contains indices of each new foreach
13     var i = 0
14     // while loop is immediately unrolled (not lifted)
15     while (i < colors.length) {
16       // DeliteArray lifts the 'for' stm to a Foreach op
17       for (c <- DeliteArray(colors(i))) {
18         f.func(c)
19       }
20       i += 1
21     }
22 } }
23
24 /* This trait is a Transformer that traverses the IR */
25 trait ColoringTransformer extends ForwardTransformer {
26   val IR: ColoringTransformerExp; import IR._
27   // colorMap constructed using stencil (impl elided)
28   val colorMap: Map[Foreach[_],Array[Array[Int]]]
29   override def transformStm(stm: Stm): Exp[Any] = stm match {
30     case TP(s,l:Loop[_]) =>
31       l.body match {
32         // transform a Foreach symbol into a new symbol
33         // representing multiple colored Foreaches
34         case f:Foreach[_] => colorFor(f,colorMap(f))
35         case _ => super.transformStm(stm)
36       }
37     case _ => super.transformStm(stm)
38 } }
```

Listing 4.4: Loop coloring transformation for OptiMesh

DSL	Delite Ops	Generic Opts.	Domain-Specific Opts.
OptiQL	Map, Reduce, Filter, Sort, GroupBy, FlatMap	CSE, DCE, code motion, fusion, SoA, DFE	
OptiCollections	Map, Reduce, ZipWith, Filter, Sort, GroupBy, FlatMap	CSE, DCE, code motion, fusion, SoA, DFE	
OptiGraph	ForeachReduce, Map, Reduce, Filter	CSE, DCE, code motion, fusion	
OptiMesh	ForeachReduce	CSE, DCE, code motion	stencil collection & coloring transformation

Table 4.1: Sharing of DSL operations and optimizations.

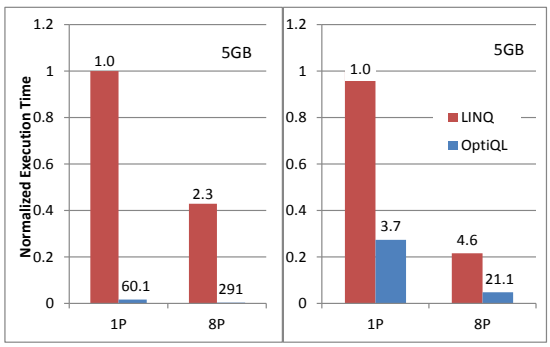
4.4.5 Reuse Summary

By embedding the front-ends of our DSLs in Scala, we did not have to implement any lexing, parsing, or type checking. As we showed in the OptiGraph vs. Green-Marl example, the syntactic difference compared to a stand-alone DSL can still be relatively small. By embedding our DSL back-ends in Delite, each DSL was able to reuse parallel patterns, generic optimizations, common library functionality (e.g. math operators), and code generators for free. One important characteristic of the embedded approach is that when a feature (e.g. a parallel pattern) is added to support a new DSL, it can be reused by all subsequent DSLs. For example, we added the `ForeachReduce` pattern for OptiGraph, but it is also used in OptiMesh.

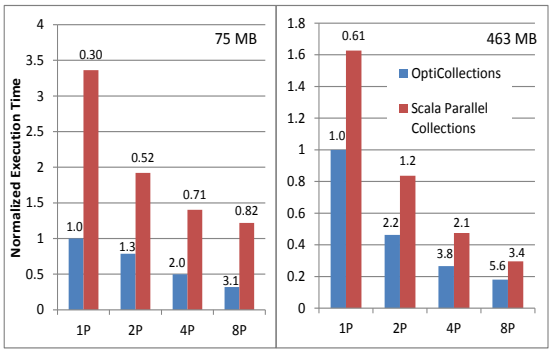
Table 4.1 summarizes the characteristics and reuse of the new DSLs introduced in this section. The DSLs inherit most of their functionality from Delite, in the form of a small set of reused parallel patterns and generic optimizations. The DSLs use just 9 Delite ops total; seven ops (77.7%) were used in at least two DSLs; three (33.3%) were used in at least three DSLs. At the same time the DSLs are not constrained to built-in functionality, as demonstrated by OptiMesh's domain-specific optimizations.

4.5 Case Studies

We present four case studies to evaluate our new DSLs. The first two case studies compare individual DSL performance against existing alternative programming environments and the second two evaluate applications composing the DSLs in the two ways described in this paper (open and closed world). OptiQL is compared to LINQ



(a) OptiQL: TPC-H Q1 (left) Q2 (right)



(b) OptiCollections: Web-link benchmark

Figure 4.3: Normalized execution time of applications written in OptiQL and OptiCollections. Speedup numbers are reported on top of each bar.

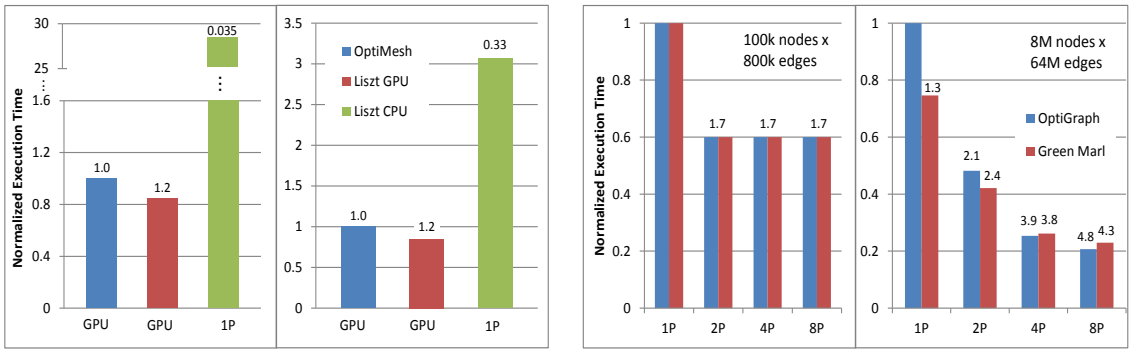
[79] and OptiCollections to Scala Collections [92]. LINQ and Scala Collections are optimized libraries running on managed platforms (C#/CLR and Scala/JVM). We compare OptiMesh and OptiGraph to Liszt [35] and Green-Marl [53] respectively, stand-alone DSLs designed for high performance on heterogeneous hardware. Liszt and Green-Marl both generate and execute C++ code and have been shown to outperform hand-optimized C++ for the applications shown in this section. Each Delite DSL generated Scala code for the CPU and CUDA code for the GPU. For composability, we compare against an analogous Scala library implementation of each application when using a combination of DSLs to solve larger application problems.

All of our experiments were performed on a Dell Precision T7500n with two quad-core Xeon 2.67GHz processors, 96GB of RAM, and an NVidia Tesla C2050. The CPU generated Scala code was executed on the Oracle Java SE Runtime Environment 1.7.0 and the Hotspot 64-bit server VM with default options. For the GPU, Delite executed CUDA v4.0. We ran each application ten times (to warm up the JIT) and report the average of the last 5 runs. For each run we timed the computational portion of the application. For each application we show normalized execution time relative to our DSL version with the speedup listed at the top of each bar.

4.5.1 Compiled Embedded vs. Optimized Library

OptiQL: TPC-H queries 1 & 2 Figure 4.3(a) compares the performance of queries 1 & 2 of the popular TPC-H benchmark suite on OptiQL vs. LINQ. Without any optimizations, OptiQL performance for the queries (not shown) is comparable to LINQ performance. However, such library implementations of the operations suffer from substantial performance penalties compared to an optimized and compiled implementation. Naïvely, the query allocates a new table (collection) for each operation and every element of those tables is a heap-allocated object to represent the row. The two most powerful optimizations we perform are converting to an SoA representation and fusing across `GroupBy` operations to create a single (nested) loop to perform the query. Transforming to an SoA representation allows OptiQL to eliminate all object allocations for Q1 since all of the fields accessed by Q1 are JVM primitive types, resulting in a data layout consisting entirely of JVM primitive arrays. All together these optimizations provide 125x speedup over parallel LINQ for Q1 and 4.5x for Q2.

OptiCollections: Reverse web-link graph Figure 4.3(b) shows the result for the reverse web-link application discussed in Section 4.4.2 running on OptiCollections compared to Scala Parallel Collections. Scala Parallel Collections is implemented using Doug Lea's highly optimized fork/join pool with work stealing [71]. The OptiCollections version is significantly faster at all thread counts and scales better with larger datasets. The improvement is due to staged compilation, which helps in two ways. First, OptiCollections generates statically parallelized code. Unlike Scala collections, functions are inlined directly to avoid indirection. On the larger dataset, this does not matter as much, but on the smaller dataset the Scala collections implementation has higher overhead which results in worse scaling. Second, the OptiCollections implementation benefits from fusion and from transparently mapping `(Int,Int)` tuples to `Long`s in the back-end. These optimizations greatly reduce the number of heap allocations in the parallel operations, which improves both scalar performance and scalability.



(a) OptiMesh: Shallow Water (left) Scalar Convection (right)

(b) OptiGraph: PageRank

Figure 4.4: Normalized execution time of applications written in OptiMesh and OptiGraph. Speedup numbers are reported on top of each bar.

4.5.2 Compiled Embedded vs. External DSL

Productivity First, we consider the programming effort required to build OptiMesh and OptiGraph compared to the stand-alone versions they were based on. Each Delite DSL compiler took approximately 3 months by a single graduate student to build using Delite. OptiMesh (≈5k loc) and OptiGraph (≈2.5k loc) were developed by 1st year Ph.D. students with no prior experience with Scala or Delite. In comparison, Liszt (≈25k loc Scala/C++) took a group of 2-3 compiler experts approximately one year to develop and Green-Marl (≈30k loc mostly C++) took a single expert approximately 6 months. As discussed in Section 4.4.5, most of the code reduction is due to reuse from being both front and back-end embedded (in Scala and Delite respectively). The Delite DSLs did not need to implement custom parsers, type checkers, base IRs, schedulers, or code generators. Similarly, while OptiMesh and OptiGraph do not implement all of the optimizations performed by Liszt and Green-Marl, they inherit other Delite optimizations (e.g. fusion) for free. For comparison, the Delite framework is ≈11k and LMS is ≈7k lines of Scala code. It is interesting to note that ≈4k lines of the Liszt code is for Pthreads and CUDA parallelization; a major benefit of using Delite is that parallelization for multiple targets is handled by the framework.

OptiMesh: Shallow water simulation & Scalar convection Figure 4.4(a) shows the performance of OptiMesh and Liszt on two scientific applications. Each application consists of a series of `ForeachReduce` operations surrounded by iterative control-flow to step the time variable of the PDE. It is well-suited to GPU execution as mesh sizes are typically large and the cost of copying the input data to the GPU is small compared to the amount of computation required. However, the original applications around which the Liszt language was designed were only implemented using MPI. Liszt added GPU code generation and demonstrated significant speedups compared to the CPU version. For both OptiMesh applications, Delite is able to generate and execute a CUDA kernel for each colored foreach loop and achieve performance comparable to the Liszt GPU implementation. Liszt's (and OptiMesh's) ability to generate both CPU and GPU implementations from a single application source illustrates the benefit of using DSLs as opposed to libraries that only target single platforms.

OptiGraph: PageRank Figure 4.4(b) compares the performance of the PageRank algorithm [88] implemented in OptiGraph to the Green-Marl implementation on two different uniform random graphs of sizes 100k nodes by 800k edges and 8M nodes by 64M edges, respectively. This benchmark is dominated by the random memory accesses during node neighborhood exploration. Since OptiGraph's memory access patterns and the memory layout of its back-end data structures are similar to those of Green-Marl, OptiGraph's sequential performance and scalability across multiple processors is close to that of Green-Marl. Although the smaller graph fits entirely in cache, the parallel performance is limited by cache conflicts when accessing neighbors and the associated coherency traffic. The sequential difference between OptiGraph and Green-Marl in the larger graph can be attributed to the difference between executing Scala generated code vs. C++. However, as we increase the number of the cores, the benchmark becomes increasingly memory-bound and the JVM overhead becomes negligible.

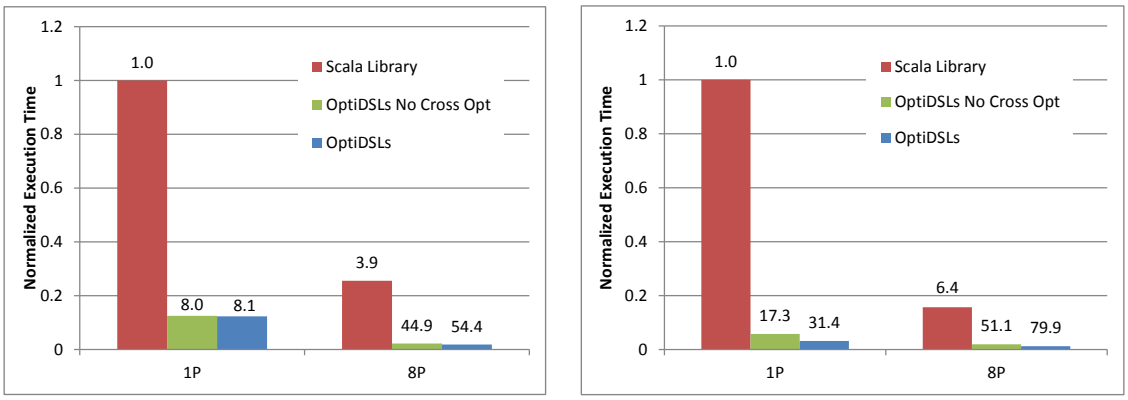
```
1  def valueIteration(actionResults: Rep[Map[Action, (Matrix[Double],Vector[Double])]],
2    initValue: Rep[Vector[Double]], discountFactor: Rep[Double], tolerance: Rep[Double]) = {
3
4    val bestActions = Seq[Action](initValue.length)
5    var (value, delta) = (initValue, Double.MaxValue)
6    while (abs(delta) > tolerance) {
7      val newValue = Vector.fill(0,value.length) { i =>
8        val allValues = actionResults map { case (action,(prob,cost)) =>
9          (action, (prob(i) * value(i) * discountFactor + cost(i)).sum)
10       }
11       val minActionValue = allValues reduce { case ((act1,v1),(act2,v2)) =>
12         if (v1 <= v2) (act1,v1) else (act2,v2)
13       }
14       bestActions(i) = minActionValue.key
15       minActionValue.value
16     }
17     delta = diff(newValue, value)
18     value = newValue
19   }
20   (value, bestActions)
21 }
```

Listing 4.5: Value Iteration of a Markov Decision Process

4.5.3 Open-world composability

We illustrate open-world composability by implementing the value iteration algorithm for a Markov decision process (MDP), shown in Listing 4.5, using two different DSLs, OptiLA and OptiCollections. OptiLA is a DSL for linear algebra providing `Matrix` and `Vector` operations, and is specifically designed to be included by other DSLs by making no closed-world assumptions. OptiLA is a refactored portion of OptiML [112], a machine learning DSL designed for statistical inference problems expressed with vectors, matrices, and graphs. Although OptiML originally contained its own linear algebra components, we have found that several DSLs need some linear algebra capability, so we extracted OptiLA and modified OptiML to extend it using the techniques in Section 4.3.1. OptiCollections was also designed to be included by other DSLs and applications, as described in Section 4.4.2.

The algorithm uses an OptiCollections `Map` to associate each user-defined `Action`



(a) small action set (b) large action set

Figure 4.5: Normalized execution time of value iteration of a Markov decision process. Performance is shown both with and without cross-DSL optimization. Speedup numbers are reported on top of each bar.

with a probability density `Matrix` and cost `Vector`. OptiLA operations (line 8) compute the value for the next time step based on an action, and OptiCollections operations apply the value propagation to every action and then find the minimal value over all actions. This process is repeated until the minimizing value converges.

Figure 4.5 shows the performance for this application implemented using Scala Parallel Collections compared to using OptiLA and OptiCollections. For both datasets, the OptiDSL version shows significant speedup over the library implementation as well as improved parallel performance, due mainly to two key optimizations: loop fusion and AoS to SoA transformation. The latter transformation eliminates all of the tuples and class wrappers in lines 7-10, leaving only nested arrays. The OptiDSLs "No Cross Opt" bar simulates the behavior of compiling the code snippets for each DSL independently and then combining the resulting optimized code snippets together using some form of foreign function interface. Therefore this version does not include SoA transformations or fusion across DSLs, but does still fuse operations fully contained within a single DSL, most notably the OptiLA code snippet on line 8 of Listing 4.5. Figure 4.5(a) shows only very modest speedups after adding DSL cross-optimization. This is because the majority of the execution time is spent within the OptiLA code snippet, and so only fusion within OptiLA was necessary to maximize performance. Figure

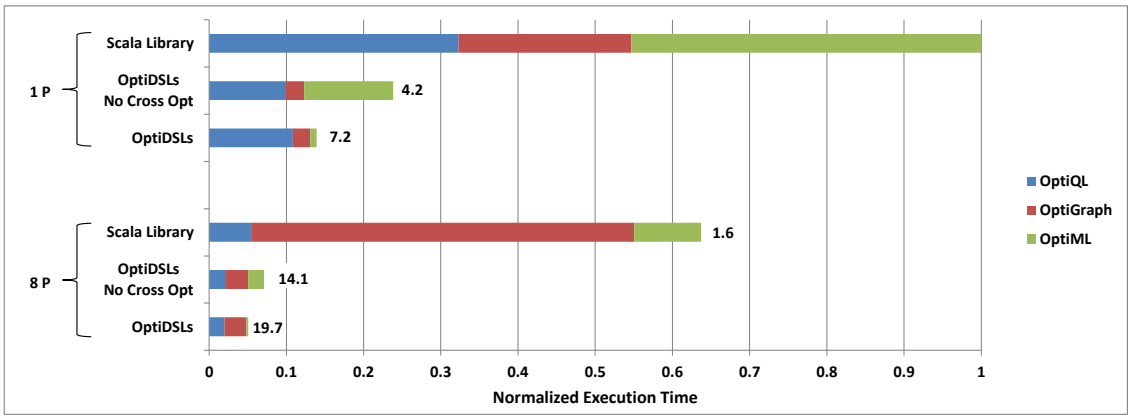


Figure 4.6: Normalized execution time of Twitter data analysis. Performance is shown for each DSL section with and without cross-DSL optimization. Speedup numbers are reported on top of each bar.

4.5(b), however, shows the behavior for different data dimensions. In this case the total execution time spent within the OptiLA section is small, making fusion across the nested OptiCollections/OptiLA operations critical to maximizing performance.

Overall this case study shows that while sometimes applications can achieve good performance by simply offloading key pieces of computation to a highly optimized implementation (applications that call BLAS libraries are a classic example of this), other applications and even the same application with a different dataset require the compiler to reason about all of the pieces of computation together at a high level and be capable of performing optimizations across them in order to maximize performance.

4.5.4 Closed-world composability

In this example, we combine OptiQL, OptiGraph, and OptiML using the closed-world composition strategy discussed in Section 4.3.2. We use the three DSLs together to implement a data analysis application on a Twitter dataset used in [131]. A truncated version of the application code is shown in Listing 4.6. The idea is to compute statistics related to the distribution of all tweets, of retweets (tweets that have been repeated by another user), and of the relationship between user connectivity and the

```
1  type Tweet = Record { val time: String; val fromId: Int; val toId: Int;
2                        val lang: String; val text: String; val RT: Boolean }
3  val q = OptiQL {
4    //tweets: Table[Tweet]
5    val reTweets = tweets.Where(t => t.lang == "en"
6                    && Date(t.time) > Date("2008-01-01")  && t.RT)
7    val allTweets = tweets.Where(t => t.lang == "en")
8    DRef((reTweets.toArray, allTweets.toArray))
9  }
10 val g = OptiGraph {
11   val in = q.get
12   val G = Graph.fromArray(in._1.map(t => (t.fromId, t.toId)))
13   val (LCC, RT) = (NodeProperty[Double](G), NodeProperty[Double](G))
14   Foreach(G.Nodes) { s =>
15     // count number of nodes connected in a triangle (omitted)
16     if (total < 1) LCC(s) = 0 else LCC(s) = triangles.toDouble / total
17     RT(s) = G.InNbrs(s).length
18   }
19   DRef((LCC.toArray, RT.toArray, in._1, in._2))
20 }
21 val r = OptiML {
22   val in = g.get
23   val scaledRT = norm(log((Vector.fromArray(in._2) + 1)))
24   val X = Matrix(Vector.ones(in._1.length), Vector.fromArray(in._1))
25   val theta = (X*X.t).inv * (X*scaledRT)  // unweighted linear regression
26   // compute statistics on tweets (omitted)
27 }
```

Listing 4.6: Twitter Graph Analysis using multiple DSLs

number of times they have been retweeted.

The application follows a typical data analytic pipeline. It first loads data from a log file containing tweets with several attributes (sender, date, text, etc.). It then queries the dataset to extract relevant information using an OptiQL `Where` statement. The filtered data is passed on to OptiGraph and OptiML which both analyze it and compute statistics. OptiGraph builds a graph where nodes are users and edges are retweets and computes the LCC (local clustering coefficient) and retweet counts for each user. The LCC is a loose approximation of the importance of a particular user. The OptiML section fits the filtered tweet data from OptiQL to a normal distribution

and also runs a simple unweighted linear regression on the LCC and retweet counts that the OptiGraph code computed. The final results of the application are the distribution and regression coefficients.

Figure 4.6 shows the performance of this application implemented with Scala Parallel Collections compared to the Delite DSLs with and without cross-DSL optimization. The graph is broken down by the time spent in each DSL section (or for the library version, in the corresponding Scala code). Since we are using the closed world model, each DSL is independently staged, lowered to the common Delite representation, and re-compiled. The application code is still a single program and the DSLs pass data in memory without any boxing overhead. Therefore, in contrast to using stand-alone compiled DSLs for each computation, we incur no serialization overhead to pipe data from one DSL snippet to the other. The Scala library version, which is also a single application, is approximately 5x slower than the non-cross-optimized DSL version on 1 thread and 10x slower on 8 threads. The speedup is due to optimizations performed by Delite across all DSLs. The OptiQL code benefits from filter fusion as well as lifting the construction of a `Date` object outside of the filter loop, which demonstrates the high-level code motion that is possible with more semantic information (the `Date` comparison is a virtual method call for Scala, so it does not understand that the object is constant in the loop). The OptiGraph version is faster than the corresponding library snippet mainly due to compiling away abstractions (we use only primitive operations on arrays in the generated code compared to Scala's `ArrayBuffer`, which has run-time overhead). The OptiGraph code is also the least scalable, since this particular graph is highly skewed to a few dominant nodes and the graph traversal becomes very irregular. Due to the additional overhead of the library version, this effect is more pronounced there. Finally, the OptiML version is faster mainly because of loop fusion and CSE across multiple linear algebra operations.

Co-optimizing the DSLs, which is enabled by composing them, produces further opportunities. The Delite compiler recognizes that OptiGraph and OptiML together require only 4 fields per tweet of the original 6 (OptiGraph uses "toId" and "fromId" and OptiML uses "text" and "hour"). The remaining fields are DFE'd by performing SoA transformation on the filter output and eliminating arrays that are not consumed

later. The other major cross DSL optimization we perform is to fuse the filter from OptiQL with their consumers in OptiGraph and OptiML. Note that the fusion algorithm is strictly data dependent; the OptiML snippet and OptiQL snippets are syntactically far apart, but can still be fused. This example also shows that since the DSL blocks are composed into a single IR, we can fuse across multiple scope boundaries when co-optimizing. All together, cross optimizations result in an extra 1.72x sequential speedup over the composed Delite DSL version without cross optimizations.

4.6 Summary

In this chapter, we presented methods to compose high performance compiled embedded DSLs at a fine ("open world") or coarse ("closed world") granularity. This ability is critical to increasing the productivity and adoption of DSL environments. In particular, we showed that we can use Delite s analysis and transformation facilities (Chapter 2.4) to perform domain-specific optimizations and also lower each DSL to a common, high-level representation that permits further optimization.

We showed that Delite can successfully target new domains by presenting four new diverse DSLs (OptiQL, OptiCollections, OptiGraph, and OptiMesh) implemented in Delite. Each DSL provides a high-level user interface and reuses common optimizations within Delite to generate efficient code for parallel and heterogeneous processors. The DSLs required only 9 parallel operators total and 7 were reused in at least two DSLs. OptiQL and OptiCollections exceed the performance of optimized library implementations by up to 125x. OptiGraph and OptiMesh are both based on existing stand-alone DSLs (Green-Marl and Liszt respectively) but require less code to build and achieve no worse than 30% slow-down. In addition to each DSL providing high performance and targeting multicore and GPU architectures, applications composing multiple DSLs perform well and benefit from cross-DSL optimization.

This chapter moves us another step closer to achieving the goal of library-level productivity with compiler-level performance, and provides new data points across different domains. However, since Delite DSLs are implemented using type-based

metaprogramming (staging), they are still not as easy to build as a typical high-level library. Furthermore, since DSL code is translated to low-level, heterogeneous processors, DSL users may still encounter error messages that are difficult to reason about while developing and testing their application. The next chapter takes a recursive approach to alleviating these issues by applying DSL principles to the domain of DSL development itself.

Chapter 5

Forge: a Meta DSL for DSL Development

In the discussion so far, we have presented high performance embedded DSLs as a programming model that is capable of enabling high-level application code to run on heterogeneous parallel devices. We showed that using a common host language and compilation framework (Delite) enables reuse of the Scala tool-chain and important infrastructure like optimizations and code generators, thereby substantially reducing the effort to create a new DSL. Furthermore, we demonstrated that Delite DSLs can generate code for different devices and are competitive with or exceed the performance of alternate systems in different domains [113]. In this chapter, we develop a meta-language to make performance-oriented DSLs even easier to develop and to use by enabling DSL semantics to be declared in a high-level specification.

5.1 Introduction

While we believe Delite to be the state-of-the-art in high performance embedded DSL development, there are still two main issues:

1. *DSL authors* must have considerable expertise with Scala and Delite in order to implement expressive, safe, and efficient DSLs. Delite is a highly flexible

architecture for heterogeneous code generation, but this expressiveness adds boilerplate and complexity to the common case. As a result, developing compiled embedded DSLs with Delite, while easier than external DSLs, requires more programming language and software engineering expertise than the average domain expert has.

2. *DSL users* must have at least some knowledge of the Delite stack. Even though DSLs are embedded in Scala, only parts of the Scala tool-chain can be used when executing user programs: generating code at runtime that is executed on heterogeneous devices makes prototyping and debugging difficult. For example, it is no longer possible to step through the program or set a breakpoint in an interactive IDE debugger.

We present Forge [111], a new meta DSL for high performance embedded DSL development that addresses these issues by capturing recurring patterns in high performance DSL development. Forge provides a high-level, declarative API for specifying DSL data structures and operations in a manner similar to an annotated high-level library. Unlike a high-level library, Forge builds an IR of the DSL specification itself, which enables it to generate different concrete implementations of the DSL. We generate both a high productivity implementation (a pure Scala library version), and a high performance implementation (a Delite version). In the future, if a different backend is desired, it is straightforward to generate a new implementation without modifying the DSL specification. From a DSL user's point of view, the Forge-generated library can be used very similarly to any other Scala library. DSL applications can be prototyped interactively in a REPL or developed in any IDE that supports Scala (e.g. Eclipse) using standard debugging techniques like breakpoints. When an application has been tested on a small dataset, the DSL user can then "flip the switch" and run the exact same source code in Delite on multicore CPUs, GPUs, or clusters. Therefore, by raising the level of abstraction and adding a level of code generation, Forge both simplifies the development of high performance embedded DSLs and makes them more accessible to end users.

Languages and frameworks for declaratively specifying DSLs are not new [42, 61, 65, 40]. Forge follows in the footsteps of these efforts by focusing on abstractions and code generation for high performance, heterogeneous computing. To our knowledge, Forge is also the first embedded meta DSL. It is implemented using staging, and therefore shares the same infrastructure as existing Delite DSLs. Staging also provides additional benefits: Forge specifications are Scala programs, and any Scala modularity feature (objects, classes, traits) can be used to compose specifications. We can also make use of staging to perform computation inside the specification itself. For example, the DSL specification can be parameterized over configuration flags, implemented as regular Scala values. This enables generating multiple variants of a DSL–essentially implementing a product line approach to DSL development. Furthermore, since Forge is staged, its IR can be constructed by invoking methods in the Forge API. This enables us to develop external parsers that call Forge methods to build the Forge IR. A key use case is to use reflection to parse existing Scala classes. Forge can then be used as an identity transformer, generating a Forge specification from the IR that can be further modified by a DSL developer. This scaffolding ability allows DSL developers to start with an existing library implementation and automatically generate a skeleton Forge specification as a starting point for a Forge DSL.

The rest of this chapter is structured as follows. In Section 5.2, we revisit the `SimpleVector` DSL example introduced in Chapter 2.1 to motivate our desire for higher-level abstractions. In Section 5.3 we provide an overview of the Forge language and show how DSLs are written in Forge. Section 5.4 describes how Forge is implemented internally and describes the artifacts that Forge generates in more detail. Section 5.5 presents case studies on three DSLs we have implemented in Forge (including two that were first implemented as stand-alone Delite DSLs and one that was implemented from scratch). Finally, Section 5.6 concludes.

Forge is open-source. The source code, including examples presented in this paper, is available at:

`http://github.com/stanford-ppl/Forge/`

```
1  trait SimpleVectorDSLInterface extends Base {
2    trait Vector[T] // an abstract DSL data type
3    object Vector {
4      // SourceContext has debugging info (e.g. line # of call-site)
5      def apply[T:Manifest](n: Rep[Int])(implicit ctx: SourceContext)
6        = vector_new[T](n)
7    }
8
9    // indirection required for abstract static method
10   def vector_new[T:Manifest](n: Rep[Int])(implicit ctx: SourceContext): Rep[Vector[T]]
11
12   // sugar for infix operators in Scala-Virtualized
13   def infix_+[T:Manifest:Numeric](x: Rep[Vector[T]], y: Rep[T]): Rep[Vector[T]]
14
15   // an overloaded version. due to type erasure, we need to use
16   // an implicit to statically disambiguate the method signature
17   def infix_+[T:Manifest:Numeric](x: Rep[Vector[T]],
18                                   y: Rep[Vector[T]])(implicit o: Overloaded1): Rep[Vector[T]]
19
20   def print(x: Rep[Any]): Rep[Unit]
21   // infix_apply (element access) elided for space
22 }
```

Listing 5.1: SimpleVector DSL interface

5.2 A Motivating Example

Listings 5.1 and 5.2 present an implementation of the `SimpleVector` DSL from Chapter 2, with slightly different operators to illustrate important productivity issues. These definitions use the simplified core of LMS/Delite from Listing 2.1.

To recap, the DSL implementation traits extend the core LMS/Delite traits to construct IR nodes when a method is called, to define helper methods required by LMS and Delite (such as `mirror`, which defines how to construct a transformed node), and to provide code generators for any node that is not a predefined pattern. A real implementation (like OptiML) uses more sophisticated versions of these IR building blocks provided by LMS and Delite, but the basic principle is the same.

Listings 5.1 and 5.2, although simplified, still demonstrate key productivity pitfalls. First, there is a significant amount of boilerplate (e.g. `mirror`), which makes

```
23 trait SimpleVectorDSLImpl extends SimpleVectorDSLInterface
24   with BaseExp {
25   // parallel domain-specific IR nodes
26   case class VPlusS[T:Manifest:Numeric](x: Exp[Vector[T], y: Exp[T])
27     extends DeliteOpMap[Vector[T]] { def func = a => a+y }
28   // sequential domain-specific IR nodes
29   case class Print(x: Exp[Any]) extends Def[Unit]
30
31   // construct IR node when method is called
32   // vector_new, overloaded infix_+ elided
33   def infix_+[T:Manifest:Numeric](x: Exp[Vector[T]], y: Exp[T]) = VPlusS(x,y)
34   def print(x: Rep[Any]) = reflectEffect(Print(x))
35
36   // constructs transformed IR nodes
37   override def mirror[A:Manifest](e: Def[A], f: Transformer) =
38     e match {
39       case VPlusS(x,y) => infix_+(f(x),f(y))
40       // ...
41       case _ => super.mirror(e,f)
42     }
43 }
44
45 trait SimpleVectorDSLCodegen extends ScalaCodegen {
46   val IR: SimpleVectorDSLImpl; import IR._
47
48   override def emitNode(s: Sym[Any], r: Def[Any]) = r match {
49     case Print(x) => emitValDef(s, "println("+quote(x)+")")
50     case _ => super.emitNode(s,r)
51   }
52 }
```

Listing 5.2: SimpleVector DSL implementation

the DSL implementation verbose and hinders readability. The boilerplate is required to perform functions that Delite cannot easily infer; for example, in the `mirror` case (lines 37-43), Delite does not know the name or arguments of the smart constructor that it needs to invoke to clone a node while still triggering any domain-specific rewrites that may be defined (which can be defined as overrides of the smart constructor). Second, DSL authors must be experts in Scala library development. They are exposed to implicit conversions, case classes, `Manifest` and `SourceContext` (which are Scala compiler-constructed types that carry around metadata), and even must know how to handle overloaded static method resolution in the presence of type erasure.

These issues arise because of the details of implementing a compiled embedded DSL in Scala. In other languages, there are different issues; for example, developing an external DSL requires DSL authors to actually deal with the entire process of lexing, parsing and type-checking. The key observation that we exploit with Forge is that by raising the level of abstraction, we can shield DSL developers from these implementation details.

5.3 Language Specification

Forge, as a meta DSL, provides methods to directly declare DSL constructs like types and operations that we saw in the previous section. Forge abstracts over the key concerns of high performance DSL development: front-end syntax, data structures, operation semantics, and parallel implementation. Forge aims to simplify DSL development by narrowing the gap between a DSL specification, which may be written as a text document, and the DSL implementation, which depends on the language and frameworks that the DSL compiler is implemented in. In the next section, we describe the implementation internals of Forge. In this section, we introduce its surface syntax and key abstractions.

To introduce Forge, we show how we can write the `SimpleVector` DSL example from Section 5.2 (including the struct definitions that were previously elided):

```
trait SimpleVectorDSL extends ForgeApplication {
  def dslName = "SimpleVector"
```

```
  def specification() {
    val T = tpePar("T")
    val Vector = tpe("Vector", T)
    data(Vector, ("_length", MInt), ("_data", MArray(T)))
    static (Vector) ("apply", T, MInt :: Vector(T), effect = mutable)
      implements allocates(Vector, ${$0}, ${ Array[T]($0) })
    direct (Vector) ("print", Nil, MString :: MUnit, effect = simple)
      implements codegen(scala, ${ println($0) })

    withTpe(Vector) {
      compiler ("vector_raw_data") (Nil :: MArray(T))
        implements getter(0, "_data")

      infix ("apply") (("n",MInt) :: T)
        implements composite ${ vector_raw_data($self)($n) }

      infix ("+") (("y",T) :: Vector(T), TNumeric(T))
        implements map((T,T), 0, ${ a => a+$y })

      infix ("+") (Vector(T) :: Vector(T), TNumeric(T))
        implements zip((T,T,T), (0,1), ${ (a,b) => a+b })
    }
  }
}
```

Every Forge specification must implement two methods: `dslName` is simply the DSL name, and `specification` is a method that contains all the DSL declarations (which could be spread across multiple files and dynamically invoked). In this example, we first declare a named type parameter, `T`, and a generic type `Vector[T]`. The **`data`** statement says that `Vector` is a struct containing two fields, `_length` and `_data`. The static method `apply` constructs a new `Vector`; `static` specifies that the user syntax for this method will be of the form `Vector(args)`, where `args` are the arguments to the

Types:

ftpe(args, ret, **freq**)
  define a new function type (args) => ret
tpePar(name)
  define new type parameter
tpe(name, tpePars, **stage**)
  define new type
tpeClass(name, tpePars)
  define new type class
tpeClassInst(name, tpePars, **tpeClass**)
  define new type class instance

Data structures:

data(**tpe**, (fieldName,fieldTpe)*)
  associate tpe with the given struct
impl(op, **allocates**(**tpe**))
  implementation pattern to allocate a struct
impl(op, **getter**(**tpe**,field))
  implementation pattern to read a field
impl(op, **setter**(**tpe**,field))
  implementation pattern to write a field

Annotations:

effect ::= simple | mutable | write | error
  declare an op has the corresponding
  effect semantic
freq ::= hot | cold | normal
  code motion hints
stage ::= now | future
  declare whether a generated type should
  be T or Rep[T]
aliasHint ::= nohint | contains(**arg**) |
  copies(**arg**) | extracts(**arg**)
  declares relationships between operations
  and inputs

Methods:

arg(name,**tpe**,default)
  define a new op argument
static | **infix** | **direct** | **compiler** | **fimplicit**
 (**grp**, name, tpePars, signature, **effect**, **aliasHint**)
  defines a new op with the specified syntax
  style and parameters
(args :: retTpe)
  defines a new method signature
impl(op, **codegen** | **single** | **composite** | **map** | **filter** |
  groupby | **reduce** | **zip** | **foreach**)
  defines the implementation for an op based
  on a predefined pattern

Miscellaneous:

grp(name)
  declares a group of ops that do not belong
  to a type
extern(name)
  declares an op group implemented in
  external code
lift(**grp**)(**tpe**)
  declares a conversion from the given
  tpe to a Rep[.]
lookupOp | **lookupGrp** | **lookupTpe** (name)
  returns a previously declared DSL construct
withTpe(**tpe**)
  construct a new syntactic scope

Parallel Collections:

parallelize(**tpe**) as ParallelCollection |
  ParallelCollectionBuffer (ops)
  declares that the provided type implements a
  ParallelCollection interface with the given ops

Listing 5.3: Forge Language Overview

op that are specified by the signature `MInt :: Vector(T)`. This signature says that the op takes a single argument of type `Int` (the length of the `Vector`), and returns a value of type `Vector[T]`. The `effect = mutable` annotation specifies that this op has the semantics of allocating a mutable data structure. Finally, **`allocates`** is an implementation pattern that constructs a new instance of `Vector` by initializing each field in the struct to an appropriate value. The `${..}` syntax is a marker for the Forge preprocessor, which quotes the argument as a formatted string, replacing argument names (specified with a preceding "$") with their synthetic names. Although these formatted strings are not type-checked when compiling or running Forge, the code in them *is* type-checked when compiling the generated DSL. The preprocessor handles tricky string escape issues while enabling users to benefit from syntax highlighting in an IDE. The last interesting construct in this example is **`withTpe`**, which introduces a Forge *syntactic scope*. Inside this scope, shorter versions of the op declaration methods are injected into the current lexical environment. The shorter versions implicitly take the enclosing `tpe` as the first argument (as well as its corresponding `tpePars`), which mimics the declaration style of instance methods in OO programming languages.

The other ops in the `SimpleVector` specification are defined in a similar fashion to `apply`, using other Forge method styles and implementation patterns. Listing 5.3 presents an overview of the Forge language constructs. The two most important groups abstract over computation and data structures, respectively, and correspond to concise versions of the Delite abstractions of Delite ops and Delite structs. Note that `implements` is an infix method that simply invokes the Forge construct **`impl`** on the result of the op invocation. In this way, Forge separates DSL interface from DSL implementation. Implementations may be defined in a completely different file, and DSL specifications can extend other DSL specifications and override behavior by defining new implementations.

DSL metaprogramming at this level also enables opportunities for programmatically-controlled reuse. Delite DSLs are typically statically dispatched, as this is most efficient and not all target platforms support dynamic dispatch (although tagged unions can be used as an alternative). Code generation provides an alternate mechanism for code reuse in this context. We can define a common `Vector` interface as follows:

```
def addVectorCommonOps(v: Rep[DSLType], T: Rep[DSLType]) {
  val VectorCommonOps = withTpe(v)
  VectorCommonOps {
    infix ("first") (Nil :: T) implements single ${ $self(0) }
    for (rhs <- List(DenseVector(T),DenseVectorView(T))) {
      infix ("+") (rhs :: DenseVector(T))
        implements zip((T,T,T), (0,1), ${ (a,b) => a+b })
      // ..
    }
    // other common ops
  }
}
```

The **for** statement in this example is statically evaluated during staging, so we can call `addVectorCommonOps` for different types of `Vector`s and replicate the common interface on each type. This enables each method to be invoked efficiently by end users on different types of `Vector`s without requiring implicit conversions, dynamic dispatch, or type classes. The trade-off is the potential for code explosion, which can negatively impact DSL compile time. This method is also insufficient for DSL users to write generic methods over `Vector`s. However, in DSLs with limited class hierarchies, this can be a sufficient, and concise, replacement for full-blown polymorphism.

While Forge tries to make declaring DSL semantics simple and concise, it is important to remember that Forge is meant for high performance embedded DSL *compilers*. It is not a goal of ours to reproduce exactly a sequential library interface. Instead, the Forge abstractions are intended to capture the critical semantics required to implement parallel DSLs on multiple hardware devices. Unsurprisingly, since Forge is based on our experience with developing DSLs in Delite, the Forge abstractions are a high-level version of concepts in Delite (like parallel patterns, effects, and alias / code motion hints). By designing Forge as a new language, we gain the flexibility to easily add and refine these abstractions over time.

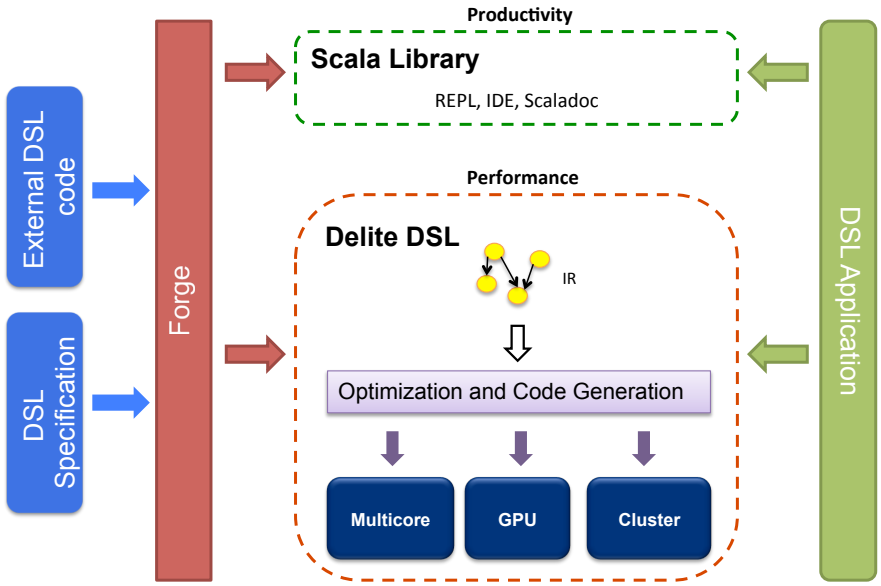


Figure 5.1: An overview of the Forge compilation pipeline. Forge takes as an input a DSL specification, as described in Section 5.3, and optional external DSL code. Forge generates two DSL implementations from these components: a high-productivity pure Scala version and a high-performance Delite version.

5.4 Compilation Pipeline

Forge is implemented as an embedded LMS DSL. Implementing it as an external DSL (or within an alternative DSL definition environment like Spoofax [65]) could make its syntax cleaner (e.g., we would not need to wrap names in strings), but would require more development effort compared to simply reusing the existing LMS infrastructure. One interesting aspect of Forge's implementation that mitigates the need for an external grammar is its unique use of Scala-Virtualized *scopes*. Scala-Virtualized is an experimental branch of the Scala compiler with additional support for DSL embedding [100]. A scope is a Scala-Virtualized construct that desugars a lexical block of code to an instantiation of a pre-defined Scala class, wrapping the block's contents inside a method of the class and then invoking the class constructor. We previously used this technique to implement coarse-grained DSL interoperability; the scope isolated the DSL interface inside the block, allowing DSL code to be invoked from within

ordinary Scala programs [113]. In Forge, we use scopes to implement the **withTpe** construct from the previous section, injecting new method signatures into a lexical scope while maintaining type safety. This is an example of using Scala-Virtualized to enable expressive embedded syntax in a way that would not normally be possible in a statically typed language. As described in Section 5.1, using staging also provides Forge with other benefits (such as the use of Scala composition, uniformity with the generated DSLs, and the ability to use staging time computations to statically manipulate DSL fragments). Like other LMS DSLs, Forge specifications are legal Scala, except for the blocks denoted with `${..}` which are preprocessed before compilation as described in Section 5.3. The preprocessor is implemented as a pre-compile hook in SBT (Simple Build Tool) [122], the predominant build tool for Scala. It is small (around 300 lines of code) and performs a simple forward pass to quote next-stage code in DSL methods using Scala string interpolation.

Figure 5.1 illustrates the Forge compilation pipeline. When run, Forge constructs an IR of the DSL consisting of types, operations, data structures, etc. Whereas Delite DSLs traverse the IR and use different code generators to generate code for different platforms, Forge traverses the IR and uses different code generators to generate different DSL implementations. Since the information required to generate a sequential Scala library is a subset of that needed to generate the Delite DSL, it is simple to generate the library version. It is also relatively straightforward to add a new code generator to Forge to retarget a DSL to a new back-end (for example, an alternative runtime) without changing the DSL specification or Forge internals.

Along with an input Forge specification, Forge also allows external Scala code that should be added to the DSL implementations. This code is placed in a configurable directory, and copied by Forge automatically to the generated directory. External code provides an escape hatch for any situation that Forge does not support. Domain-specific optimizations, such as pattern rewrites and transformations, are defined as external code using Delite APIs directly, and made visible to the generated DSL using Forge's **extern** command. We chose this method because Delite's APIs for pattern rewriting and transformers [103] are already high-level and declarative; it is not obvious how we could abstract over these APIs in a useful way. Furthermore,

these sorts of optimizations are impossible to implement in the Scala library version, so are only relevant to the Delite version.

Once the DSL implementations are generated, there is no further dependency on Forge, and DSL users can use either one. An important aspect of Forge's code generation is that it leverages polymorphic embedding not only for the Delite implementation, but also for the library implementation. In the Delite version, `Rep[T]` = `Exp[T]` as in normal Delite DSLs, but in the library version, `Rep[T]` = `T`, and Forge generates concrete classes and methods on those classes for DSL types. The key point is that DSL users now use exactly the same interface when writing their application, and need only run a different Scala object to switch between versions:

```
object MyAppInterpreter extends SimpleVectorApplicationInterpreter with MyApp
object MyAppCompiler extends SimpleVectorApplicationCompiler with MyApp

trait MyApp extends SimpleVectorApplication {
  def main() = {
    val v = Vector.rand(10)
    println("v.sum: " + sum(v))
  }
}
```

Running `scala` on `MyAppInterpreter` after compiling will run the Scala application, while running `MyAppCompiler` will run the Delite version to stage it and generate code for different devices. When Forge generates a DSL, it also generates the SBT project file for the DSL. A DSL user simply has to run `sbt; console`, and they will be dropped into a Scala REPL with all interpreter DSL dependencies pre-loaded. This provides DSL users a way to prototype their application in the Scala REPL (they can even copy and paste app code as normal) and debug their applications inside Scala IDEs in the ordinary way. In the development cycle, this also means that DSL users can also avoid expensive compilation and staging time until they actually require high performance. When a user has finished debugging, he can switch to a larger dataset and invoke the Delite DSL. Therefore, although we have added an additional compilation step in the multi-stage compilation pipeline for DSL authors, the development cycle for

DSL users can be considerably shortened. Since there are far more users than authors, this is normally a good trade-off. In order to maintain incremental compilation across the multiple stages while developing the DSL, Forge uses `rsync` to copy files to the generated directory.

In addition to supporting DSL construction from scratch, Forge includes a *scaffolding* code generator that allows DSL developers to bootstrap off of regular Scala classes. This generator serializes the Forge IR to re-emit a Forge specification (i.e., from an existing Forge specification, it acts like an identity generator). In order to generate a skeleton Forge specification using reflection, we use staging to build the Forge IR by simply calling the appropriate Forge methods while traversing the class. After the specification is generated, a DSL author can then fill in the gaps by adding semantic annotations (e.g. effects) and alternate code generators. As an example, if we reflect the standard `String` class using:

```
importAuto[java.lang.String]
```

Forge will generate a skeleton specification like the following:

```
val String = tpe ("java.lang.String")
val StringOps = grp ("String")

infix (StringOps)("trim", Nil,
  ((String) :: String), effect = simple) implements
  (codegen(scala, ${ $0.trim }))

infix (StringOps)("toLowerCase", Nil,
  ((String) :: String), effect = simple) implements
  (codegen(scala, ${ $0.toLowerCase }))

infix (StringOps)("replaceAll", Nil,
  ((String,String) :: String), effect = simple) implements
  (codegen(scala, ${ $0.replaceAll($1) }))

...
```

The implementation of `importAuto` is straightforward. It uses Scala reflection to traverse the methods of the given class and stages the corresponding Forge commands on the fly:

```
def importAuto[T:TypeTag] = {
  val scalaType = typeTag[T].tpe
  val forgeType = toForgeType(scalaType)
  val forgeClass = grp(scalaType.toString)
  lift (forgeClass) (forgeType)

  for (m <- scalaType.members if m.isMethod) {
    val methType = method.asTerm.typeSignature
    val args = toForgeArgs(methType.params)
    val ret = toForgeType(methType.resultType)

    infix (forgeClass) (m.name.toString, Nil,
      ((forgeType :: args)) :: ret, Nil, simple) implements
        (codegen(scala, quotedArg(0) + "." + m.name + argList(args)))
  }
}
```

This approach could easily be extended to read other Forge constructs from Scala or Java method annotations. With a sizable coverage of the Forge language, such annotations could provide a lightweight alternative front-end, at least for classes for which the developer is in control of the source code.

In the future, we plan to add additional code generators to Forge to generate additional artifacts. For example, if we allow users to specify Scaladoc annotations in the spec, we can generate the Scaladoc annotations in the Scala library implementation and leverage Scaladoc to generate the HTML API docs. In contrast, with ordinary Delite, there are no concrete classes in the DSL implementation and no place to put the Scaladoc annotations. We also plan to explore generating versions of the DSL that are less human readable, but are faster to compile (for example, by passing all implicits and specifying all types explicitly).

5.5 Evaluation

We have implemented three DSLs with Forge: OptiML (machine learning) [112], OptiQL (data querying) [113], and OptiWrangler (data transformation). Two of these (OptiML and OptiQL) were based on the existing Delite DSLs presented in Chapters 3 and 4 respectively, and we show that Forge significantly simplified their implementation without sacrificing performance. Furthermore, the new Forge implementations automatically produce library versions as we have discussed, so for the first time, OptiML and OptiQL can be used in a lightweight interactive way. OptiWrangler is a new implementation of Wrangler [64], and was implemented directly in Forge instead of ported from an existing Delite DSL.

For each DSL, we show that Forge lives up to the promise of DSL authors being able to write their DSL once, DSL users being able to write their application once, and being able to run efficiently on heterogeneous, parallel devices. We compare the performance of the Forge-generated DSL implementations to hand-optimized C++, to Spark [132], a Scala library for multicore and cluster computing, and to the previous Delite implementation (when available). In general, the hand-optimized C++ code is low-level and complex while the Spark version is high-level Scala that is much easier to read and to use. These two data points provide a strong measure of where the embedded DSL implementations stand in terms of productivity and performance for end users. The DSL application code is single-source and high-level, but as before, Delite uses staging and also performs key optimizations like fusion and struct unwrapping to generate kernels that are low-level and first-order (either Scala or CUDA). In most cases, the generated Delite code closely resembles the hand-optimized C++.

Experimental Methodology We ran Forge on each DSL to generate a Scala library version and the Delite version. For each DSL application, Delite generated parallel Scala code for the CPU and CUDA code for the GPU (when possible). We ran the generated Scala code over a cluster using the Delite runtime's support for distributed computing [15], which uses Apache Mesos [51] and Google Protocol Buffers [48] as the underlying communication layer. The Spark experiments were run with Spark

DSL	Forge specification	Delite (manual)	Forge generated
OptiML	1322	7416	11743
OptiQL	301	862	1287
OptiWrangler	343	n/a	1814

Table 5.1: LOC for Forge implementations of each DSL vs. existing Delite implementations.

0.7.0.

Multicore CPU and GPU experiments were performed on a Dell Precision T7500n with two quad-core Xeon 2.67 GHz processors, 96GB of RAM, and an NVidia Tesla C2050. The Scala code was executed on Oracle's Java SE Runtime Environment 1.7.0 and the HotSpot 64-bit server VM with a maximum heap size of 40GB. The generated CUDA code was executed with CUDA v3.2. The C++ implementations were compiled using g++ 4.4.7 with -O3. The cluster CPU experiments were performed on Amazon EC2 using 20 m1.xlarge instances. Each machine contained 4 virtual cores, 15GB of memory, and 1Gb Ethernet connections between the 20 machines. We used the default JVM available on Amazon Linux, Java 1.6.0b24 with default options, for all three systems. We ran each application ten times (to warm up the JIT) and report the average of the last 5 runs.

When counting lines of code (LOC) to compare Forge specifications of OptiML and OptiQL to the previous Delite versions, we used CLOC [32], and counted only the subset of features of the original DSLs that we re-implemented in the Forge spec.

5.5.1 OptiML

OptiML is a DSL for machine learning; we presented its design and original Delite implementation in Chapter 3. To briefly summarize, OptiML provides implicitly parallel vectors, matrices, and graphs that support bulk collection operators (e.g. `map`, `filter`) as well as standard math operators when used with numeric types. OptiML also supports control structures for iterative algorithms (such as `untilconverged` and `gradient descent`) that are common in machine learning. The Forge implementation of OptiML includes the dense data types (`DenseVector`, `DenseMatrix`), most OptiML

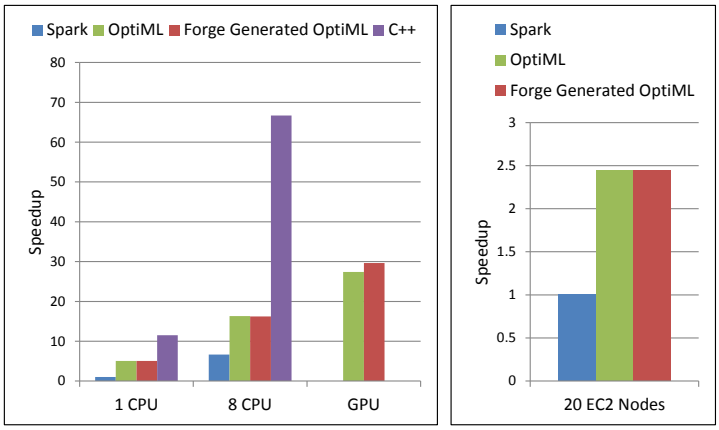


Figure 5.2: Speedup of Delite versions and manually-written C++ over Spark with LR on a 500k x 100 element dataset (multicore) and 10M x 100 (cluster).

mathematical functions, and the control structures. Row 1 of Table 5.1 shows the LOC count of the OptiML specification vs. the corresponding subset of the original implementation, as well as the LOC generated by Forge. Forge provides nearly a 6x reduction in LOC over the original implementation, while generating roughly 50% more LOC because it also generates the library version, which was not a part of the original OptiML. The savings comes mainly from reducing boilerplate and automatically generating the embedded DSL structure and appropriate calls into the Delite API (as discussed in Section 5.2). We also gain some savings by using staging and code generation to implement common `Vector` operations on different kinds of `Vector`s, whereas the previous implementation used a verbose packaging of type classes to achieve (almost) the same functionality. Qualitatively, the OptiML specification is also simpler to read and modify than the previous version, primarily because there is much less clutter than in the original embedding implementation.

To evaluate performance, we ran Logistic Regression (LR), a simple classification algorithm for predicting the discrete value of a data sample (for example, whether a particular email is spam or not). Figure 5.2 shows the results running on multicore CPUs, a GPU, and across the 20 node EC2 cluster. The Delite version of Forge-generated OptiML achieves the same performance as the original OptiML implementation because we are able to generate the same code from the Forge specification.

The Delite versions are about 2x slower than the low-level C++, but 2.5x faster than the high-level Spark. This C++ implementation is optimized to manually fuse all loops; it allocates an output buffer per thread in order to parallelize, but otherwise contains no intermediate allocations. Furthermore, the C++ code is byte-padded to prevent false sharing, which initially caused a 3x slow-down when naïvely parallelized using OpenMP. The Delite versions fail to reach this level of performance because the fusion algorithm misses one opportunity between different parts of a reduction kernel; we believe in the future we can extend the algorithm to cover this case, which will result in the generated Delite CUDA code outperforming the manual C++ even when starting from the high-level DSL code.

In the distirbuted setting, Delite is also able to run more efficiently across nodes than Spark for the same reasons it performs better in the multicore case: fusion eliminates intermediate allocations and staging generates more efficient code than Spark, which uses high-level Scala abstractions. It is important to note that Spark, in general, is extremely efficient; it has been shown to achieve order of magnitude speedups over equivalent Hadoop implementations by intelligently keeping data in memory across multiple iterations. Delite also keeps data in memory, but at this time does not provide the same fault-tolerance guarantees as Spark.

Finally, note that the Forge-generated library version did not finish on either dataset. This version uses identical code to the pre-staged Delite version (by construction), but since it is not staged or optimized, it suffers heavily from boxing and uses far more memory. In particular, the use of polymorphic embedding in its interface imposes more dispatch and boxing overhead than the Spark version, and since it is sequential, it cannot run on multiple processors. This demonstrates that the library version is suited for interactive prototyping with small datasets, but not for high performance or large-scale execution.

5.5.2 OptiQL

OptiQL (presented in Chapter 4.4.1) is a LINQ-like [79] DSL for data querying. Its primary data type is a `Table`, and it provides query operators (e.g. `Select`, `Where`,

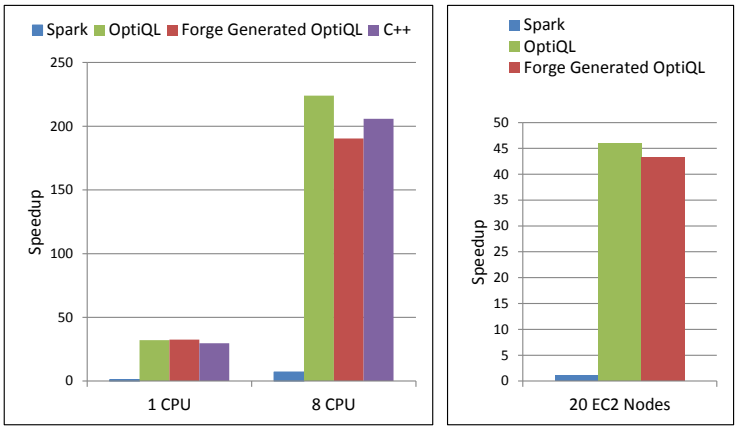


Figure 5.3: Speedup of Delite versions and manually-written C++ over Spark on TPC-H Q1 on a 1 GB table (multicore) and 5 GB (cluster).

`Sum`) over them. Like OptiML, we started from an existing Delite implementation of OptiQL, and ported it to Forge. The Forge version contains all of the supported features of OptiQL, but rewrite optimizations were implemented as external DSL code directly using Delite. The rewrites performed by OptiQL perform additional fusion of query operators that go beyond the generic fusion provided by Delite.

Row 2 of Table 5.1 shows that the Forge OptiQL specification is about 3x shorter than the original Delite implementation. The difference is less dramatic than OptiML because OptiQL is a smaller DSL and a significant portion of its code is for the rewriting optimizations. Figure 5.3 shows the results of running the TPC-H benchmark suite query 1 (Q1) on multicore CPUs and on the cluster. TPC-H is a well-known database benchmark suite and Q1 consists of a single query containing `Select`, `Where`, `GroupBy`, and aggregate (e.g. `Sum`) statements. In the Delite version, the programmer-friendly array-of-struct representation is automatically converted to a more efficient struct-of-array representation, and then all of the operators are fused into a single, compact loop. Furthermore, fields that are part of the input dataset that are not used in the query are automatically eliminated from the generated code via dead field elimination. As a result, the Delite versions perform roughly the same as hand-optimized C++ and outperform Spark by 30x. In this case the speedup is magnified because the Delite optimizations have multiplicative effects; AoS to SoA

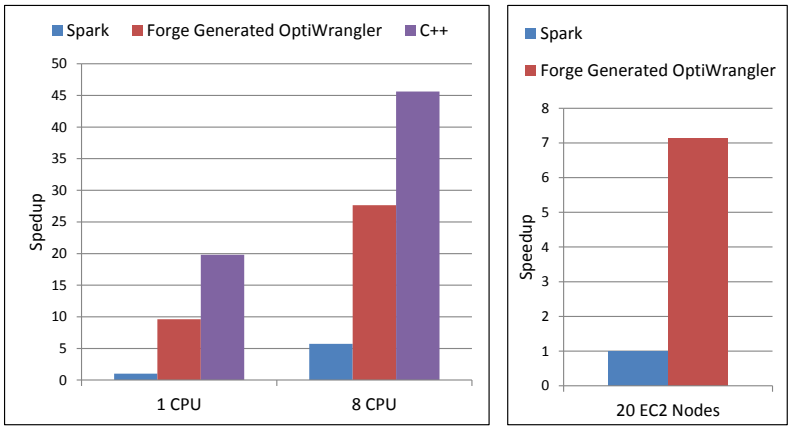


Figure 5.4: Speedup of Delite versions and manually-written C++ over Spark on the gene processing application with 3M sequences (multicore) and 25M (cluster).

enables fusion and struct unwrapping, which in turn enable dead-field elimination and the stack allocation of primitives. The same optimizations help Delite scale efficiently on the cluster, achieving 43x speedup over Spark. The increase is due to the fact that although the overall dataset is larger, the data per node is less in this experiment, and efficiency matters more. Similarly to OptiML, the Forge-generated Delite OptiQL version performs as well as the manual Delite version, while the Forge-generated library version is unable to finish for the same reasons as before (the lack of optimizations combined with polymorphic embedding and boxing overheads).

5.5.3 OptiWrangler

OptiWrangler [1] is a DSL for structured string transformations and cleansing operations based on the interactive Wrangler system [64]. OptiWrangler exposes a single data element, `Wrangler`, and offers high level primitives (e.g. `cut`, `split`) that act over rows and columns of tabular data. In addition to managing a single table, OptiWrangler abstracts gracefully over multiple tables, allowing users to `partition` and `merge` tables without restructuring an application designed for a single table. We implemented OptiWrangler as both a Forge DSL and as a library with Spark. Unlike the other DSLs, OptiWrangler has not been previously implemented in Delite, so we

[1] *Credits:* Austin Gibbons and Jithin Thomas are the primary authors of OptiWrangler.

use Spark as a comparison for productivity (note that in Subsections 5.5.1 and 5.5.2 the Spark implementations were for the application only, while for OptiWrangler, we implemented the entire DSL in Spark). Row 3 of Table 5.1 shows the code generated expansion of the Forge implementation into a Delite DSL. OptiWrangler falls in between OptiML and OptiQL in terms of generated code size with a roughly 5x expansion. The Spark implementation of OptiWrangler (not shown in the table) is 253 LOC. Even though Spark is a pure Scala library, the Forge implementation is only 1.36x larger, and the difference is magnified because Forge specifications have a slightly higher fixed cost in LOC than Spark. For larger DSLs, this initial overhead would be amortized.

We evaluate performance on an application provided by a geneticist common in his workflow. Geneticists perform many types of structured-to-structured string operations, such as removing "barcodes" from gene sequences, separating genes by type, and extracting interesting subsequences. This application tests one kind of these structured transformations. Figure 5.4 shows the application performance with Forge-generated Delite vs. Spark and C++ (no manual Delite implementation exists). The generated Delite version unwraps all structs, leaving only a tight loop over arrays of strings. It performs approximately 2x slower than the C++ version, which heavily uses mutation to avoid allocating, for example, even new strings when constructing substrings. This requires using only a careful subset of C++ stdlib functions and makes the program harder to reason about. Like the C++ implementation of LR, this version also required byte padding to scale. On the other hand, Delite outperforms the Spark implementation by nearly an order of magnitude, which in this application is driven mostly by expensive object allocations. These benefits again translate to the cluster, resulting in a 7x speedup. This application falls somewhere in between the OptiML and OptiQL examples, since it is relatively simple (so there are not many optimizations to apply), but also not computationally heavy (so efficiency is important).

5.6 Summary

This chapter presented Forge, a meta DSL for high performance embedded DSL development. Forge is unlike previous language construction approaches in that it is embedded, staged, and generates multiple DSL implementations oriented towards simplifying heterogeneous parallel processing. Forge improves on the previous state of the art of compiled embedded DSLs by generating two versions of a DSL, a Scala library implementation that can be used for prototyping and debugging, and a Delite framework implementation that outperforms alternative systems and can run on multicore CPUs, GPUs, and clusters from a single application. Forge can also generate itself, so it can be used with an external parser, or reflection, to generate a skeleton specification from existing libraries. We have demonstrated that Forge simplifies the process of developing and using Delite DSLs, achieving up to 6x reduction in LOC compared to existing Delite DSLs while still providing order of magnitude speedups compared to library-based approaches. We believe that a declarative specification enables new opportunities for compiled embedded DSLs, such as the ability to transparently modify the underlying high performance framework or retarget the DSLs to alternate backends.

Chapter 6

Related Work

This dissertation draws on a wide-range of previous work in the fields of staged metaprogramming, domain-specific languages, parallel computing, and language workbenches. In this chapter we survey the state-of-the-art in each of these fields and describe how our techniques build upon and differ from related systems.

6.1 Staged Metaprogramming

The embedding of DSLs in a host language was first described by Hudak [55]. Tobin-Hochstadt et al. [121] extend Racket, a Scheme dialect, with constructs to enhance the embeddability of other languages. We also use an enhanced version of the Scala compiler (Scala-virtualized [82]) to allow for a deeper embedding of DSLs. Feldspar [7] is an embedded DSL that combines shallow and deep embedding of domain operations to generate high performance code. Our DSLs are embedded in a similar fashion, but inherit the embedding from the Delite framework.

Taha et. al. pioneered the field of multi-stage programming with MetaML [116] and MetaOCaml [16], extensions of ML and OCaml with staging annotations to demarcate future stage code. Our DSLs, through LMS, use type-directed staging instead of explicit annotations. Several other static metaprogramming methods exist, including C++ templates [123] and Template Haskell [107]. Expression Templates [124] can produce customized generation, and are used by Blitz++ [125]. Veldhuizen

introduced active libraries [126], which are libraries that participate in compilation. Kennedy introduced telescoping languages [67], efficient DSLs created from annotated component libraries. TaskGraph [8] is a meta-programming library that sports run-time code generation in C++. Lightweight Modular Staging is built on the idea of embedding typed languages by Carette et al. [17] and Hofer et al. [52]. The use of `Rep[T]` types to define binding times in LMS is inspired by earlier work on finally tagless or polymorphic language embedding [18, 52].

Many existing program generators such as FFTW [44], ATLAS [130] and SPIRAL [94] took enormous efforts to build. LMS and Delite aim to make generative facilities more easily accessible. Terra [34] is a recent programming language that also uses staging for a similar goal, but uses a high-level dynamic language (Lua) to stage a lower-level, performance-oriented language (Terra).

Our compiler infrastructure implements a set of advanced optimizations over the staged IR in a reusable fashion. Related work includes program transformations using advanced rewrite rules [13], combining analyses and optimizations [74, 127, 28] as well as techniques for eliminating intermediate results [128, 31] and loop fusion [49]. An interesting alternative to speculative rewriting (inspired by [74]) is equality saturation [117], which encodes many ways to express an operation in the IR.

As an example of using partial evaluation in program transformation, Sperber and Thiemann [108] perform closure conversion and tail call introduction by applying offline partial evaluating to a suitable interpreter. More recently, Cook et al. [30] perform model transformation by partial evaluation of model interpreters. Partial evaluation corresponds to constant folding and specialization whereas our approach allows arbitrary compiler optimizations, arbitrary computation at staging/specialization time to remove abstraction overhead and provides strong guarantees about what becomes part of the residual code (type `Rep[T]` residual, vs `T` static).

6.2 High Performance DSLs

There is a rich history of DSL compilation in both the embedded and stand-alone contexts. Leijen et al. [73] and Elliot et al. [37] pioneered embedded compilation and used

a simple image synthesis DSL as an example. Feldspar [7] is an embedded DSL that combines shallow and deep embedding of domain operations to generate high performance code. For stand-alone DSL compilers, there has been considerable progress in the development of parallel and heterogeneous DSLs. Liszt [35] and Green-Marl [53] are external DSLs for mesh-based PDE solvers and static graph analysis respectively. Both of these DSLs target both multicore CPUs and GPUs and have been shown to outperform optimized C++ implementations. Diderot [26] is a parallel DSL for image analysis that demonstrates good performance compared to hand-written C code using an optimized library. Halide [97] is also a DSL for image processing. It cleverly exploits the spatial domain and streaming nature of image processing kernels to separate the description of the algorithm from its parallel implementations (tiling, vectorizing, fusing, etc.), enabling the systematic exploration of performance trade-offs for different programs. The Spiral system [93] progressively lowers linear transform algorithms through a series of different DSLs to apply optimizations on different levels [43]. Subsets of Spiral have also been implemented using Scala and LMS. Giarrusso et al. [47] investigate database-like query optimizations for collection classes and present SQuOpt, a query optimizer for a DSL that, like OptiCollections, mimics the Scala collections API and also uses techniques similar to LMS to obtain an IR for relevant program expressions. Our work aggregates many of the lessons and techniques from previous DSL efforts and makes them easier to apply to new domains.

Recent work has also begun to explore how to compose domain-specific languages and runtimes. Mélusine [39] uses formal modeling to define DSLs and their composition. Their approach attempts to reuse existing models and their mappings to implementations. Dinkelar et al. [36] present an architecture for composing purely embedded DSLs using aspect-oriented concepts; a meta-object is shared between all the eDSLs and implements composition semantics such as join points. MadLINQ [95] is an embedded matrix DSL that integrates with DryadLINQ [60], using LINQ as the common back-end. Compared to these previous approaches, our work is the first to demonstrate composition and co-optimization with high performance, statically compiled DSLs.

6.3 Parallel and Heterogeneous Computing

Outside the context of DSLs, there have been efforts to compile high-level general purpose languages to lower-level (usually device-specific) programming models. Mainland et al. [76] use type-directed techniques to compile an embedded array language, Nikola, from Haskell to CUDA. This approach suffers from the inability to overload some of Haskell's syntax (if-then-else expressions) which isn't an issue with our version of the Scala compiler. Nystrom et al. [86] show a library-based approach to translating Scala programs to OpenCL code. This is largely achieved through Java bytecode translation. A similar approach is used by Lime [5] to compile high-level Java code to a hardware description language such as Verilog. Since the starting point of these compilers is the much lower-level byte-code or Java code (relative to DSL code), the opportunities for high-level optimizations are more limited.

There are many existing parallel programming models operating at various levels of abstraction. These models are typically targeted towards application developers directly, rather than via DSLs. A popular category is high-level data-parallel programming models that provide implicit parallelization by providing the programmer with a data-parallel API that is transparently mapped to the underlying hardware. OpenCL [118] provides a standard that allows a programmer to target different hardware devices from a single environment rather than using a distinct vendor API for each device. This however doesn't eliminate the need to specialize the OpenCL code to suit the different architectures the application is executed on. Copperhead [19], which automatically generates and executes CUDA code on a GPU from a data-parallel subset of Python. Array Building Blocks [57] manages execution of data-parallel patterns across multiple processor cores and targets different hardware vector units from a single application source. DryadLINQ [60] converts LINQ [79] programs to execute using Dryad [59], which provides coarse-grained data-parallel execution over clusters. FlumeJava [23] targets Google's MapReduce [33] from a Java library and fuses operations in the data-flow graph in order to generate an efficient pipeline of MapReduce operations. Our work builds on these previous publications and also

allows for domain-specific optimizations which is not possible with these other approaches. Furthermore, Delite DSLs provide a simpler interface for end users than general purpose frameworks, which typically expose parallelism (or parallel operators) directly.

Scalable, distributed data analytics have become increasingly essential (driven by increasing amounts of data and the competitive desire to extract commercial value from this data). Many systems have been developed to make it easier to scale programs across multiple compute nodes (typically shared-nothing). Google's MapReduce [33] has inspired several of these systems, including the open-source implementation Hadoop [4] and the in-memory distributed system Spark [132] which we have discussed in detail in this dissertation. While Delite takes a composed DSL view of data analytics, these systems take the general programming model and heavily restrict it to operators (map and reduce) that can be easily distributed. In order to make Hadoop more accessible to programmers, SQL layers like Pig [46] and Hive [120] have been built on top of it. Impala [29] is a scalable SQL query engine for data stored in a Hadoop cluster. OptiQL provides a similar SQL-like interface, while using Delite as the underlying optimization and distribution engine.

In addition to generic MapReduce-like systems, distributed graph processing has emerged as a popular topic. Pregel [77] (and the open source version, Giraph [6]) employ a bulk-synchronous, vertex-centric programming model. GraphLab [75] is a C++ high-performance distributed programming model designed for graphs that supports multiple consistency models and synchronization schemes. Green-Marl [53], which we have already discussed extensively, is a high performance DSL for graph processing and the inspiration for OptiGraph.

Finally, there are several database management systems (DBMS) targeted at data analytics, including systems from Oracle [87], Vertica [54] and Greenplum [90]. These systems typically maintain a SQL interface, use column-oriented data stores, and focus heavily on query plan optimization and efficient distribution and execution. Delite relies on our array-of-struct to struct-of-array transformation to move between row-oriented and column-oriented storage. While the transformation-driven approach is not guaranteed to succeed, it usually results in an efficient layout that is easy to

distribute to multiple nodes. The Hazy project [68] combines statistical analyses with database systems to develop fast and scalable tools, algorithms, and applications for analytics.

6.4 Language Workbenches

The need for infrastructure to define DSLs has long been recognized and several languages and frameworks for declaratively specifying DSLs exist. The Kermeta workbench [83] is a metaprogramming environment based on metamodel engineering, which applies meta-languages to the problem of model transformations. Kermeta leverages DSLs for transforming models and the authors present common language constructs for model manipulation. The Eclipse Modeling Frameworks (EMF) [42] provide tools to generate code from a structured data model, specified in various languages (e.g. Java, XML). JetBrains MPS [61] and Spoofax [65] are language workbenches that enable developers to specify DSL grammars, static analyses and transformations (via rewrite rules, e.g. with Stratego [12]), and code generators. XText [40], JetBrains MPS and Spoofax all support automatically generating sophisticated toolchain support for custom languages, such as IDE plugins, without relying on a host language. SugarJ [38], on the other hand, does utilize a host language by enabling language developers to translate grammar extensions to the host grammar (Java), as well as apply rewrite rules and transformations. Forge follows the spirit of these efforts but focuses on code generation to make high performance, heterogeneous computing more accessible, based on our experiences with the Delite framework. Since our effort has been invested on the back-ends of optimizing DSL compilers, there is significant potential in combining Forge with a declarative framework for front-end grammars and compile-time (as opposed to staging-time) static analyses. This would enable DSL developers to have even more flexibility to define both highly expressive and high performance DSLs.

There has also been work on extensible compilation frameworks aimed towards making high performance languages easier to build. Racket [41] is a dialect of Scheme designed to make constructing new programming languages easier. Spoofax [65] and

JetBrains MPS [61] are language workbenches for defining new DSLs and can generate automatic IDE support from a DSL grammar. While these efforts also support DSL reuse and program transformation, they are generally more focused on expressive DSL front-ends, whereas Delite's emphasis is on high performance and heterogeneous compilation. Both areas are important to making DSL development easier and could be used together to complement each other. On the performance side, telescoping languages [67] automatically generate optimized domain-specific libraries. They share Delite's goal of incorporating domain-specific knowledge in compiler transformations. Delite compilers extend optimization to DSL data structures and are explicitly designed to generate parallel code for multiple heterogeneous backends. Delite also optimizes both the DSL and the program using it in a single step. Stratego [12] is a language for program transformation using extensible rewrite rules. Stratego's organization also focuses on a reusable set of components, but targeted specifically to program transformation. Delite extends these principles to a more diverse set of components in order to target end-to-end high performance program execution for DSLs.

Chapter 7

Conclusions and Future Work

This dissertation began by summarizing state-of-the-art techniques to embed high performance DSL compilers in Scala, using the Lightweight Modular Staging (LMS) and Delite frameworks. We presented in detail the first DSL developed using these techniques, OptiML, as well as DSLs for data querying (OptiQL), graph processing (OptiGraph), mesh analysis (OptiMesh), and collections (OptiCollections). Each DSL provides a high-level programming interface and is automatically parallelized and compiled to heterogeneous devices. Furthermore, we showed that by reusing critical infrastructure, it is possible to compose and interoperate across these DSLs in a high performance manner. Finally, to make high performance embedded DSLs easier to build and to use, we presented Forge, a meta-DSL that enables DSL developers to program to a high-level declarative specification and DSL users to prototype using a simple library version of the DSL. The net effect of this work is embedded DSLs that are simpler to develop and use than previous compiled DSLs, while achieving performance better than alternative systems in each domain (and often comparable to hand-optimized low-level code). Programming in one of these DSLs frees the developer from low-level parallel and heterogeneous computing concerns, enabling high performance with much less effort. The trade-off is generality – programs must be rewritten in one or more DSLs to realize the benefits of this approach.

Delite is a maturing DSL platform that is being used actively by research groups at Stanford and EPFL, and is beginning to spread to other early adopters in academia

and in the open-source community. We believe that the initial DSLs we have developed offer encouraging evidence that DSLs can be an effective way of obtaining high performance with low effort in numerous domains. Most of the DSLs took a single academic quarter to implement by one or two graduate students. In contrast, external DSLs typically require compiler expertise and take much longer to implement. Delite's main leverage comes from dividing the critical expertise required to implement a DSL amongst different persons depending on their area of concern. To develop a typical high performance external DSL, a compiler developer would need domain, language, and hardware expertise. Instead, Delite allows framework authors to provide the compiler and hardware expertise, and DSL authors to focus on DSL design and identifying the important domain abstractions (domain decomposition). Additionally, Delite DSLs benefit transparently from improvements in the shared infrastructure; when Delite adds support for new targets (e.g. clusters, FPGAs), all Delite DSLs can utilize these improvements with little or no change to the DSL itself.

One key reason that Delite is able to generate efficient code is its emphasis on immutable, functional parallel patterns. Functional programming has long been considered well-suited for parallelism, since it is easier to reason about, optimize, and parallelize immutable data structures. Delite leverages these ideas in its parallel patterns, while DSL users are still only exposed to more familiar domain-specific abstractions. Typically, immutable data structures also have substantial overhead due to the creation of intermediate objects. By extending the compiler's semantics with domain-specific information, i.e. by mapping domain-specific operations to IR nodes, Delite can automatically transform these functional operators to efficient imperative code. Starting from imperative code, on the other hand, greatly limits the compiler's ability to optimize and target different hardware. It is important to note that Delite is designed to allow new parallel patterns to be easily added as they are encountered; we do not believe that the current set is by any means complete, but when a new pattern is implemented, that effort can be reused by all DSLs.

While there is good opportunity for DSLs to become a practical alternative to low-level programming for high performance, there are also substantial challenges that remain. The proliferation of performance-oriented DSLs creates new problems. Users

must choose between different, potentially incompatible DSLs to implement their application, and often want to combine elements from multiple DSLs. In Chapter 4, we showed that Delite DSLs can be composed at different granularities by exploiting the common back-end framework [113]. However, this composition is limited (for DSLs with restricted semantics, we allow only pipeline composition), and does not address composing DSL front-ends (e.g. grammars or type systems).

Furthermore, isolating embedded DSL programs from their general-purpose host language remains difficult. In this dissertation, we use type-directed staging to embed DSL compilers. However, it is often not obvious to a novice user when they are programming in the host language and when they are in the DSL. Non-DSL code, while syntactically legal since we reuse the host language parser, often fails with opaque type errors (usually related to trying to convert from a `Rep[T]` to `T` or vice versa). In order to make high performance DSL snippets more practical, we should extend our interoperability techniques from Chapter 4 to support cleanly integrating pre-compiled DSL code inside a general-purpose environment. For example, Delite could support generating DSL program *stubs* that could be called using any language's foreign function interface (FFI), and execute through the Delite runtime. This level of integration would enable rewriting performance-critical portions of a large program in an embedded DSL without paying any compilation overhead when invoking the DSL snippet.

This work and its predecessors have heavily emphasized static optimization and code generation. While we have shown that this technique is effective in speeding up the common cases in different domains, we are also vulnerable to the "sufficiently smart compiler" problem. When the compiler makes the wrong optimization or scheduling choice, how does the user know, and what can she do about it? New DSLs like Halide [97] and DSL frameworks like Terra [34] make it easier for developers to systematically explore performance trade-offs and use autotuners to find the best performing code. Incorporating these techniques into Delite and providing user-facing performance measurement tools and annotations to control compiler behavior would make Delite more accessible to performance experts. There is also a growing desire and need for high-performance, large-scale programming at interactive, low-latency

time scales. In this setting, static compilation may be prohibitively expensive. Here, just-in-time (JIT) approaches like Lancet [104] combined with a dynamic runtime deployed as a service are attractive directions.

Delite focuses on optimizing compilation for heterogeneous and parallel processors, but there has also been good progress in simplifying the development of the front-ends of DSL compilers (for example, via language workbenches such as Spoofax [65] or extensible language frameworks such as SugarJ [38]). Combining these avenues of research is an exciting future direction. Many of these frameworks have also demonstrated the ability to automatically generate high-quality tool-chains for DSLs, such as custom IDE plugins. These tool-chains are critical for DSL adoption. However, targeting heterogeneous hardware adds new challenges for tool-chains, since bugs can manifest in many different layers or devices. One promising avenue for DSL tool-chains to go beyond their general-purpose counterparts is with domain-specific debuggers. A graph analysis DSL, for example, could provide visualizations and step-through debugging on the state of the graph as a first-class primitive. Another open problem is heterogeneous resource scheduling and cost modeling, but there is long-standing work in multiple fields (including databases and operating systems) to draw from. Finally, there is potential for further specialization and cooperation throughout the stack; domain-specific compilers can be co-designed with domain-specific hardware to provide even more performance.

Bibliography

[1] AccelerEyes. Jacket. `http://www.accelereyes.com`, 2010.

[2] AMD. The Industry-Changing Impact of Accelerated Computing. White Paper, 2008.

[3] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[4] Apache. Hadoop. `http://hadoop.apache.org/`.

[5] Joshua Auerbach, David F. Bacon, Perry Cheng, and Rodric Rabbah. Lime: a Java-compatible and synthesizable language for heterogeneous architectures. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA, pages 89–108, New York, NY, USA, 2010. ACM.

[6] Ching Avery. Giraph: large-scale graph processing infrastruction on Hadoop. *Proceedings of Hadoop Summit. Santa Clara, USA:[sn]*, 2011.

[7] Emil Axelsson, Koen Claessen, Mary Sheeran, Josef Svenningsson, David Engdal, and Anders Persson. The design and implementation of Feldspar: An embedded language for digital signal processing. In *Proceedings of the 22nd international conference on Implementation and application of functional languages*, IFL'10, pages 121–136, Berlin, Heidelberg, 2011. Springer-Verlag.

[8] Olav Beckmann, Alastair Houghton, Michael Mellor, and Paul H.J. Kelly. Run-time code generation in C++ as a foundation for domain-specific optimisation. In Christian Lengauer, Don Batory, Charles Consel, and Martin Odersky, editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 77–210. Springer Berlin / Heidelberg, 2004.

[9] Jeff Bezanson, Stefan Karpinski, Viral B Shah, and Alan Edelman. Julia: A fast dynamic language for technical computing. *arXiv preprint arXiv:1209.5145*, 2012.

[10] Gregory R. Bowman, Xuhui Huang, and Vijay S. Pande. Using generalized ensemble simulations and markov state models to identify conformational states. *Methods*, 49(2):197 – 201, 2009.

[11] G. Bradski and M. Muja. BiGG Detector. `http://www.ros.org/wiki/bigg_detector`, 2010.

[12] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. a language and toolset for program transformation. *Sci. Comput. Program.*, 72(1-2):52–70, June 2008.

[13] Martin Bravenboer, Arthur van Dam, Karina Olmos, and Eelco Visser. Program transformation with scoped dynamic rewrite rules. *Fundam. Inf.*, 69:123–178, July 2005.

[14] Kevin J. Brown, Arvind K. Sujeeth, HyoukJoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. A heterogeneous parallel framework for domain-specific languages. In *Proceedings of the 20th International Conference on Parallel Architecture and Compilation Techniques*, PACT, 2011.

[15] Kevin J. Brown, Arvind K. Sujeeth, HyoukJoong Lee, Tiark Rompf, Christopher De Sa, Martin Odersky, and Kunle Olukotun. Big data analytics with Delite. `http://ppl.stanford.edu/papers/delite-scaladays13.pdf`, 2013.

[16] Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. Implementing multi-stage languages using ASTs, Gensym, and Reflection. GPCE, pages 57–76, 2003.

[17] Jacques Carette, Oleg Kiselyov, and Chung chieh Shan. Finally tagless, partially evaluated. In *APLAS*, pages 222–238, 2007.

[18] Jacques Carette, Oleg Kiselyov, and Chung chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.*, 19(5):509–543, 2009.

[19] Bryan Catanzaro, Michael Garland, and Kurt Keutzer. Copperhead: compiling an embedded data parallel language. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP, pages 47–56, New York, NY, USA, 2011. ACM.

[20] H. Chafi, Z. DeVito, A. Moors, T. Rompf, A. K. Sujeeth, P. Hanrahan, M. Odersky, and K. Olukotun. Language Virtualization for Heterogeneous Parallel Computing. Onward!, 2010.

[21] H. Chafi, A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya, and K. Olukotun. A domain-specific approach to heterogeneous parallelism. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP, 2011.

[22] Hassan Chafi. *Scaling High Performance Domain-Specific Language Implementation with Delite*. PhD thesis, Stanford University, 2014.

[23] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. Flumejava: easy, efficient data-parallel pipelines. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI. ACM, 2010.

[24] Robit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. *Parallel programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[25] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40(10):519–538, 2005.

[26] Charisee Chiw, Gordon Kindlmann, John Reppy, Lamont Samuels, and Nick Seltzer. Diderot: a parallel dsl for image analysis and visualization. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, pages 111–120, New York, NY, USA, 2012. ACM.

[27] C. Chu, S. Kim, Y. Lin, Y. Yu, G. Bradski, A. Y. Ng, and K. Olukotun. Map-reduce for machine learning on multicore. In *Advances in Neural Information Processing Systems 19*. 2007.

[28] Cliff Click and Keith D. Cooper. Combining analyses, combining optimizations. *ACM Trans. Program. Lang. Syst.*, 17:181–196, March 1995.

[29] Cloudera. Impala. `http://www.cloudera.com/content/cloudera/en/products-and-services/cdh/impala.html`.

[30] William R. Cook, Benjamin Delaware, Thomas Finsterbusch, Ali Ibrahim, and Ben Wiedermann. Model transformation by partial evaluation of model interpreters. Technical Report TR-09-09, UT Austin Department of Computer Science, 2008.

[31] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream fusion: from lists to streams to nothing at all. In *ICFP*, pages 315–326, 2007.

[32] A Danial. CLOC–count lines of code. *Open source*, 2009.

[33] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. OSDI, pages 137–150, 2004.

[34] Zachary DeVito, James Hegarty, Alex Aiken, Pat Hanrahan, and Jan Vitek. Terra: a multi-stage language for high-performance computing. In *Proceedings*

of the 34th ACM SIGPLAN conference on Programming Language Design and Implementation, pages 105–116. ACM, 2013.

[35] Zachary DeVito, Niels Joubert, Francisco Palacios, Stephen Oakley, Montserrat Medina, Mike Barrientos, Erich Elsen, Frank Ham, Alex Aiken, Karthik Duraisamy, Eric Darve, Juan Alonso, and Pat Hanrahan. Liszt: A domain specific language for building portable mesh-based PDE solvers. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC, 2011.

[36] T. Dinkelaker, M. Eichberg, and M. Mezini. An architecture for composing embedded domain-specific languages. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, pages 49–60. ACM, 2010.

[37] Conal Elliott, Sigbjórn Finne, and Oege de Moor. Compiling embedded languages. In Walid Taha, editor, *Semantics, Applications, and Implementation of Program Generation*, volume 1924 of *Lecture Notes in Computer Science*, pages 9–26. Springer Berlin / Heidelberg, 2000.

[38] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. SugarJ: library-based language extensibility. In *Proceedings of the ACM international conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA, pages 391–406. ACM, 2011.

[39] Jacky Estublier, German Vega, and AncaDaniela Ionita. Composing domain-specific languages for wide-scope software engineering applications. In Lionel Briand and Clay Williams, editors, *Model Driven Engineering Languages and Systems*, volume 3713 of *Lecture Notes in Computer Science*, pages 69–83. Springer Berlin Heidelberg, 2005.

[40] Moritz Eysholdt and Heiko Behrens. Xtext: implement your language faster than the quick and dirty way. In *Proceedings of the ACM international conference companion on Object Oriented Programming Systems Languages and*

Applications companion, SPLASH '10, pages 307–309, New York, NY, USA, 2010. ACM.

[41] Matthew Flatt. Creating languages in Racket. *Commun. ACM*, 55(1):48–56, January 2012.

[42] The Eclipse Foundation. Eclipse Modeling Framework Project (EMF). `http://www.eclipse.org/modeling/emf/`, 2013.

[43] Franz Franchetti, Yevgen Voronenko, and Markus Püschel. Formal loop merging for signal transforms. In *PLDI*, pages 315–326, 2005.

[44] Matteo Frigo. A fast Fourier transform compiler. In *PLDI*, pages 169–180, 1999.

[45] Yoshihiko Futamura. Partial evaluation of computation process - an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999.

[46] Alan F Gates, Olga Natkovich, Shubham Chopra, Pradeep Kamath, Shravan M Narayanamurthy, Christopher Olston, Benjamin Reed, Santhosh Srinivasan, and Utkarsh Srivastava. Building a high-level dataflow system on top of Map-Reduce: the Pig experience. *Proceedings of the VLDB Endowment*, 2(2):1414–1425, 2009.

[47] Paolo G. Giarrusso, Klaus Ostermann, Michael Eichberg, Ralf Mitschke, Tillmann Rendel, and Christian Kästner. Reify your collection queries for modularity and speed! AOSD, 2013.

[48] Google. Protocol buffers data interchange format. `http://code.google.com/p/protobuf`, 2011.

[49] Clemens Grelck, Karsten Hinckfuß, and Sven-Bodo Scholz. With-loop fusion for data locality and parallelism. In Andrew Butterfield, Clemens Grelck, and Frank Huch, editors, *Implementation and Application of Functional Languages*, IFL, pages 178–195. Springer Berlin / Heidelberg, 2006.

[50] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2012.

[51] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI, pages 22–22. USENIX Association, 2011.

[52] C. Hofer, K. Ostermann, T. Rendel, and A. Moors. Polymorphic embedding of DSLs. GPCE, 2008.

[53] Sungpack Hong, Hassan Chafi, Eric Sedlar, and Kunle Olukotun. Green-Marl: A DSL for easy and efficient graph analysis. ASPLOS, 2012.

[54] HP. Vertica. `http://www.vertica.com/`.

[55] P. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28, 1996.

[56] Intel. Cilk Plus. `http://software.intel.com/en-us/articles/intel-cilk-plus/`.

[57] Intel. Intel array building blocks. `http://software.intel.com/en-us/articles/intel-array-building-blocks`.

[58] Intel. Intel math kernel library. `http://software.intel.com/en-us/intel-mkl`.

[59] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, EuroSys, pages 59–72, New York, NY, USA, 2007. ACM.

[60] Michael Isard and Yuan Yu. Distributed data-parallel computing using a high-level programming language. In *Proceedings of the 35th SIGMOD International Conference on Management of Data*, SIGMOD, pages 987–994, New York, NY, USA, 2009. ACM.

[61] JetBrains. Meta Programming System. `http://www.jetbrains.com/mps/`, 2009.

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

[63] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[64] Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. Wrangler: interactive visual specification of data transformation scripts. In *PART 5 —— – Proceedings of the 2011 annual conference on Human Factors in Computing Systems*, CHI '11. ACM, 2011.

[65] Lennart C.L. Kats and Eelco Visser. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In *Proceedings of the ACM international conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 444–463, New York, NY, USA, 2010. ACM.

[66] M. Kearns. Efficient noise-tolerant learning from statistical queries. *Joural of the ACM*, 45:983–1006, 1998.

[67] Ken Kennedy, Bradley Broom, Arun Chauhan, Rob Fowler, John Garvin, Charles Koelbel, Cheryl McCosh, and John Mellor-Crummey. Telescoping languages: A system for automatic generation of domain languages. *Proceedings of the IEEE*, 93(3):387–408, 2005.

[68] Arun Kumar, Feng Niu, and Christopher Ré. Hazy: making it easier to build and maintain big-data analytics. *Communications of the ACM*, 56(3):40–49, 2013.

[69] C. Lattner and V. Adve. LLVM: a compilation framework for lifelong program analysis transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86, 2004.

[70] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Softw.*, 5(3):308–323, September 1979.

[71] Doug Lea. A Java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, JAVA '00, pages 36–43, New York, NY, USA, 2000. ACM.

[72] V. W. et al. Lee. Debunking the 100x GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *Proc. ISCA*, 2010.

[73] Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *Proceedings of the 2nd conference on Domain-specific languages*, DSL, pages 109–122, New York, NY, USA, 1999. ACM.

[74] Sorin Lerner, David Grove, and Craig Chambers. Composing dataflow analyses and transformations. *SIGPLAN Not.*, 37:270–282, January 2002.

[75] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A New Parallel Framework for Machine Learning. In *UAI*, 2010.

[76] Geoffrey Mainland and Greg Morrisett. Nikola: embedding compiled GPU functions in Haskell. In *Proceedings of the third ACM Haskell symposium on Haskell*, Haskell '10, pages 67–78, New York, NY, USA, 2010. ACM.

[77] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[78] MathWorks. Matlab. `http://www.mathworks.com/products/matlab/`.

[79] Erik Meijer, Brian Beckman, and Gavin Bierman. LINQ: Reconciling object, relations and XML in the .NET framework. In *Proceedings of the 2006 ACM*

SIGMOD International Conference on Management of Data, SIGMOD. ACM, 2006.

[80] J. Meng, S. Chakradhar, and A. Raghunathan. Best-effort parallel execution framework for recognition and mining applications. In *Proc. of IPDPS*, 2009.

[81] G. E. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, 38(8):114–117, April 1965.

[82] Adriaan Moors, Tiark Rompf, Philipp Haller, and Martin Odersky. Scala-virtualized. In *Proceedings of the ACM SIGPLAN 2012 workshop on Partial Evaluation and Program Manipulation*, PEPM, 2012.

[83] Pierre-Alain Muller, Franck Fleurey, Didier Vojtisek, Zoé Drey, Damien Pollet, Frédéric Fondement, Philippe Studer, Jean-Marc Jézéquel, et al. On executable meta-languages applied to model transformations. In *Model Transformations in Practice Workshop*, MTiP, 2005.

[84] Feng Niu, Benjamin Recht, Christopher Ré, and Stephen J. Wright. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *NIPS*, 2011.

[85] NVIDIA. CUDA. `http://developer.nvidia.com/object/cuda.html`.

[86] Nathaniel Nystrom, Derek White, and Kishen Das. Firepile: run-time compilation for GPUs in scala. In *Proceedings of the 10th ACM international conference on Generative Programming and Component Engineering*, GPCE, pages 107–116, New York, NY, USA, 2011. ACM.

[87] Oracle. Big Data. `http://www.oracle.com/us/technologies/big-data/products/index.html`.

[88] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[89] Michael Paleczny, Christopher Vick, and Cliff Click. The Java Hotspot(tm) server compiler. In *In USENIX Java Virtual Machine Research and Technology Symposium*, pages 1–12, 2001.

[90] Pivotal. Greenplum. `http://www.gopivotal.com/big-data/pivotal-greenplum-database`.

[91] J. C. Platt. Sequential minimal optimization: A fast algorithm for training support vector machines, 1998.

[92] Aleksandar Prokopec, Phil Bagwell, Tiark Rompf, and Martin Odersky. A generic parallel collection framework. Euro-Par, 2010.

[93] M. Püschel, J.M.F. Moura, J.R. Johnson, D. Padua, M.M. Veloso, B.W. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R.W. Johnson, and N. Rizzolo. Spiral: Code generation for DSP transforms. *Proceedings of the IEEE*, 93(2):232 –275, feb. 2005.

[94] Markus Püschel, José M. F. Moura, Bryan Singer, Jianxin Xiong, Jeremy Johnson, David A. Padua, Manuela M. Veloso, and Robert W. Johnson. Spiral: A generator for platform-adapted libraries of signal processing alogorithms. *IJHPCA*, 18(1):21–45, 2004.

[95] Zhengping Qian, Xiuwei Chen, Nanxi Kang, Mingcheng Chen, Yuan Yu, Thomas Moscibroda, and Zheng Zhang. MadLINQ: large-scale distributed matrix computation for the cloud. In *Proceedings of the 7th ACM european conference on Computer Systems*, EuroSys '12, pages 197–210, New York, NY, USA, 2012. ACM.

[96] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2014.

[97] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In

Proceedings of the 34th ACM SIGPLAN conference on Programming Language Design and Implementation, pages 519–530. ACM, 2013.

[98] Parthasarathy Ranganathan. From microprocessors to nanostores: Rethinking data-centric systems. *Computer*, 44(1):39–48, 2011.

[99] Tiark Rompf. *Lightweight Modular Staging and Embedded Compilers: Abstraction Without Regret for High-Level High-Performance Programming*. PhD thesis, EPFL, 2012.

[100] Tiark Rompf, Nada Amin, Adriaan Moors, Philipp Haller, and Martin Odersky. Scala-virtualized: Linguistic reuse for deep embeddings. Higher-Order and Symbolic Computation (Special issue for PEPM'12).

[101] Tiark Rompf and Martin Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *Proceedings of the ninth international conference on Generative Programming and Component Engineering*, GPCE, pages 127–136, New York, NY, USA, 2010. ACM.

[102] Tiark Rompf and Martin Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. *Commun. ACM*, 55(6):121–130, 2012.

[103] Tiark Rompf, Arvind K. Sujeeth, Nada Amin, Kevin Brown, Vojin Jovanovic, HyoukJoong Lee, Manohar Jonnalagedda, Kunle Olukotun, and Martin Odersky. Optimizing data structures in high-level programs. POPL, 2013.

[104] Tiark Rompf, Arvind K. Sujeeth, Kevin J. Brown, Hassan Chafi, and Kunle Olukotun. Surgical precision JIT compilers. PLDI, 2014.

[105] Tiark Rompf, Arvind K. Sujeeth, HyoukJoong Lee, Kevin J. Brown, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Building-blocks for performance oriented DSLs. DSL, 2011.

[106] Conrad Sanderson. Armadillo: An Open Source C++ Linear Algebra Library for Fast Prototyping and Computationally Intensive Experiments. Technical Report, NICTA, 2006.

[107] T. Sheard and S.P. Jones. Template meta-programming for Haskell. *ACM SIGPLAN Notices*, 37(12):60–75, 2002.

[108] Michael Sperber and Peter Thiemann. Realistic compilation by partial evaluation. In *PLDI*, pages 206–214, 1996.

[109] Arvind K. Sujeeth. OptiML language specification 0.2. `http://stanford-ppl.github.io/Delite/optiml/downloads/optiml-spec.pdf`.

[110] Arvind K. Sujeeth, Kevin J. Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM Trans. Embed. Comput. Syst.*, 13(4s):134:1–134:25, April 2014.

[111] Arvind K. Sujeeth, Austin Gibbons, Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Martin Odersky, and Kunle Olukotun. Forge: Generating a high performance DSL implementation from a declarative specification. In *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences*, GPCE, 2013.

[112] Arvind K. Sujeeth, Hyoukjoong Lee, Kevin J. Brown, Tiark Rompf, Michael Wu, Anand R. Atreya, Martin Odersky, and Kunle Olukotun. OptiML: an implicitly parallel domain-specific language for machine learning. In *Proceedings of the 28th International Conference on Machine Learning*, ICML, 2011.

[113] Arvind K. Sujeeth, Tiark Rompf, Kevin J. Brown, HyoukJoong Lee, Hassan Chafi, Victoria Popic, Michael Wu, Aleksander Prokopec, Vojin Jovanovic, Martin Odersky, and Kunle Olukotun. Composition and reuse with compiled domain-specific languages. In *European Conference on Object Oriented Programming*, ECOOP, 2013.

[114] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3):202–210, 2005.

[115] Walid Taha. Multi-stage programming: Its theory and applications. Technical report, Oregon Graduate Institute School of Science & Engineering, 1999.

[116] Walid Taha and Tim Sheard. MetaML and multi-stage programming with explicit annotations. *Theor. Comput. Sci.*, 248(1-2):211–242, 2000.

[117] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: a new approach to optimization. In *POPL*, pages 264–276, 2009.

[118] The Khronos Group. OpenCL 1.0. `http://www.khronos.org/opencl/`.

[119] Douglas L. Theobald. Rapid calculation of RMSDs using a quaternion-based characteristic polynomial. *Acta Crystallographica Section A*, 61(4):478–480, Jul 2005.

[120] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Antony, Hao Liu, and Raghotham Murthy. Hive-a petabyte scale data warehouse using Hadoop. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 996–1005. IEEE, 2010.

[121] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages as libraries. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '11, pages 132–141, New York, NY, USA, 2011. ACM.

[122] Typesafe. Simple Build Tool. `http://www.scala-sbt.org`.

[123] D. Vandevoorde and N.M. Josuttis. *C++ templates: the Complete Guide*. Addison-Wesley Professional, 2003.

[124] Todd L. Veldhuizen. Expression templates, C++ gems. SIGS Publications, Inc., New York, NY, 1996.

[125] Todd L. Veldhuizen. Arrays in Blitz++. In *ISCOPE*, pages 223–230, 1998.

[126] Todd L. Veldhuizen. *Active Libraries and Universal Languages*. PhD thesis, Indiana University Computer Science, May 2004.

[127] Todd L. Veldhuizen and Jeremy G. Siek. Combining optimizations, combining theories. Technical report, Indiana University, 2008.

[128] Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theor. Comput. Sci.*, 73(2):231–248, 1990.

[129] Jörg Walter and Mathias Koch. The Boost uBLAS library, 2002.

[130] R. Clinton Whaley, Antoine Petitet, and Jack Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1-2):3–35, 2001.

[131] Jaewon Yang and Jure Leskovec. Patterns of temporal variation in online media. In *Proceedings of the fourth ACM international conference on Web Search and Data Mining*, WSDM '11, pages 177–186, New York, NY, USA, 2011. ACM.

[132] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, NSDI, 2011.

[133] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[134] M. A. Zinkevich, M. Weimer, A. Smola, and L. Li. Parallelized stochastic gradient descent. In *Advances in Neural Information Processing Systems*, December 2010.