HAVE ABSTRACTION AND EAT PERFORMANCE TOO: OPTIMIZED HETEROGENEOUS COMPUTING WITH PARALLEL PATTERNS

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

Kevin James Brown
March 2018

© 2018 by Kevin James Brown. All Rights Reserved.
Re-distributed by Stanford University under license with the author.

This work is licensed under a Creative Commons Attribution-Noncommercial 3.0 United States License.
http://creativecommons.org/licenses/by-nc/3.0/us/

This dissertation is online at: http://purl.stanford.edu/ny040cz9411

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 hardware is trending towards increasingly parallel and heterogeneous architectures. However, mainstream programming languages still focus on simplifying the programming of a single general purpose CPU, providing only the barest of tools for exploiting parallel and distributed systems. This presents programmers with a broad set of new considerations and challenges when attempting to target these architectures. Multicore architectures are now ubiquitous, but utilizing the additional processors to improve performance requires utilizing low-level communication primitives which introduce an entirely new class of potential synchronization bugs. In some machines, the processors are additionally spread across multiple sockets, where each socket can access some system memory faster than the rest, creating non-uniform memory access (NUMA). This introduces the additional complexity of reasoning about how both code and data are distributed across each processor to maximize locality. It is also increasingly common for machines to contain one or more specialized accelerators, such as GPUs. Unfortunately each accelerator typically comes with a unique programming model and language which the programmer must then integrate into the CPU program in an ad-hoc way. Finally, utilizing multiple machines simultaneously requires the addition of a low-level networking layer and even stricter data locality requirements than NUMA.

Combining together multiple classic parallel programming models such as using MPI to communicate between machines, Pthreads to parallelize across cores within a machine, and CUDA to offload work to each GPU, can produce a highly efficient implementation but is extremely low-level, tedious, and error-prone. What we want

instead is a single system and programming model that is optimized both for modern hardware (high performance) and for modern programmers (high productivity). This system should be capable of running a variety of parallel applications, provide sequential baseline performance comparable to hand-optimized implementations, and continue to function well both as more resources are added to a single machine (scale up) and more machines are interconnected (scale out).

A promising solution for providing higher-level programming abstractions capable of hiding all this additional complexity is domain-specific languages (DSLs). By providing abstractions at the level of abstract domain constructs (e.g., `Matrix` and `Vector` for linear algebra) rather than low-level hardware and operating system constructs (e.g., arrays, threads, and locks) DSL applications are naturally written in a manner that is much more hardware-independent and therefore can be transformed and optimized for different hardware targets. Delite is a compiler framework for building performance-oriented DSLs that aims to reduce the burden on DSL developers by providing many reusable components. By implementing domain constructs in terms of high-level parallel patterns such as `Map` and `Reduce`, each DSL defers the low-level details of implementing each pattern on each hardware target to Delite. Using this technique, Delite DSL applications have been shown to run efficiently on both multicore CPUs and GPUs from a single application source. Delite has previously been limited, however, to exploiting a single level of parallelism over shared-memory architectures and could not efficiently target NUMA or distributed architectures. The primary limitation of Delite's original design is that parallel patterns alone are not always sufficient to obtain portable performance, in particular when the patterns are nested. Then the optimal traversal order becomes both data layout and architecture dependent.

In this dissertation, we introduce the Delite Multiloop Language (DMLL), a new intermediate language based on common parallel patterns that captures the necessary semantic knowledge to efficiently target distributed heterogeneous architectures. Combined with a straightforward array-based data structure model, the language semantics naturally capture a set of powerful transformations over nested parallel

patterns that restructure computation to enable distribution and optimize for heterogeneous devices. A new analysis phase determines what data to distribute based on its access patterns and the locations of its input(s). This analysis also triggers the DMLL rewrite rules to improve locality when it detects a suboptimal access pattern. We augmented Delite by replacing its internal parallel pattern and data structure representations with DMLL. In order to more easily reason about data access patterns and improve performance, we use multiple data structure optimizations, include eliminating structure allocations, eliminating unused fields, and performing array-of-struct (AoS) to struct-of-array (SoA) transformations. Finally we exploit DMLL's restricted semantics and the limited data structure model to perform specialized garbage collection during parallel operations that is significantly more space and time efficient than existing general purpose garbage collection techniques. All together these additional transformations enabled improved single-threaded performance, greater parallel scalability, smaller memory footprints, transparently targeting distributed memory architectures, and automated data movement and distribution for existing Delite DSL applications spanning machine learning, data querying, and graph analytics.

Acknowledgments

There are many individuals I have had the privilege to work with during my graduate career. Without their contributions and influences, large and small, this dissertation would not have been possible.

I would first like to thank my advisor, Kunle Olukotun, for his continuous stream of fresh ideas for the next big research challenge, unwavering support, and ample patience. Christos Kozyrakis provided additional guidance from my very first quarter to my last. Chris Ré became a new and welcome presence towards the end of my tenure, and greatly aided my research with novel parallel algorithms as well as countless entertaining conversations. Nathan Bronson was always willing to aid a younger, more confused version of myself. Everything I know about the bowels of the Java Virtual Machine I owe to him.

Martin Odersky and the rest of the Scala team at EPFL were instrumental in guiding the initial formation of this work and helping to see it through. From their ideas on language design to implementing custom extensions in the Scala compiler, we built an international collaboration that lasted many years and publications.

Most of all, I would like to thank my Delite family: Arvind Sujeeth, HyoukJoong Lee, Tiark Rompf, and Hassan Chafi. They were instrumental in both designing and implementing everything in this dissertation and much more. We all lost many hours of sleep together and this work would be far less than one-fifth as complete without them. Their experiences and insights were all invaluable in reaching this point. More importantly, they are still some of my best friends.

Other students I had the privilege to work with include Chris Aberger, Chris De Sa, David Koeplinger, and Raghu Prabhakar. When my original team had all

graduated, they were there to fill the void with fresh ideas, projects, and enthusiasm.

Last but certainly not least I want to thank my family, who have supported me unconditionally in all my endeavors. I could never have reached this point without them. I owe everything I have to them.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Distributed Heterogeneous Computing

As single processor performance has leveled off in recent years, advancements in hardware design have shifted. Performance improvements now come from providing multiple parallel processors as well as specialized hardware that can execute a specific class of operations more quickly and efficiently than general purpose processors. For example, today's high-end servers contain multiple independent processing cores within a single socket (multicore). They also contain multiple such sockets, where each socket is directly connected to an even fraction of the total system memory. While the hardware still provides a shared-memory abstraction to software, this design creates non-uniform memory access (NUMA) latencies as each processor can access the data stored in its local memory significantly faster than the remote memories. In addition the server may have one or more specialized processing units, or accelerators, such as GPUs and FPGAs.

While the peak theoretical performance of these servers is increasing regularly, developing programs that can actually achieve anywhere near that peak performance is incredibly challenging. Parallel programming using a low-level API such as Pthreads introduces an entirely new class of potential correctness bugs caused by improper synchronization and communication between threads. Since these bugs may only manifest with a specific interleaving of thread execution, they are also much more

difficult to detect and correct than traditional sequential bugs. NUMA effects then introduce an additional set of performance bugs. Many parallel programs will run correctly on a NUMA machine but without any additional speedup due to a large number of remote memory accesses bottlenecking execution.

Exploiting any available accelerators requires further complicating the application by rewriting portions of it in a new programming language for that accelerator and then managing the data movement between the accelerator and system memory. These accelerator languages typically have a completely different programming paradigm from traditional sequential languages. CUDA [62], for example, is a language for NVidia GPUs that presents the user with a multidimensional grid of threads that each operate on a small piece of input data simultaneously with limited synchronization capabilities. To actually benefit from the available accelerator(s), the programmer must first master the new programming paradigms, decide which components of a given application could benefit from acceleration, re-implement those components, and add additional vendor-specific synchronization between the host and accelerator.

If a single machine is still insufficient to achieve the required performance, the programmer must then resort to creating a distributed application over a cluster of machines. Similar to NUMA, the main pitfall in designing a scalable distributed program is ensuring that the majority of data accesses are to local memory. Remote memory accesses are not only much more expensive but require yet another programming model such as MPI to manage serializing and transporting the data over the network.

The inclusion of all these disparate programming models not only makes programs challenging to develop initially, but also makes updating and maintaining them more difficult. Porting the exact same application from a single machine with a GPU to a cluster of machines without GPUs is not transparent but rather requires a substantial rewrite. In short, such programs are not only significantly harder to write but also non-portable to new hardware, and therefore significantly harder to maintain. In practice, the complexity of such programs make them non-tractable for all but the experts.

What we need instead is a single system and programming model that can efficiently utilize modern heterogeneous distributed hardware and also provide high-level, hardware-independent abstractions. We identify the following requirements for such a system:

Optimized Node Processing: To fully utilize modern machines the system must be capable of efficiently and hierarchically parallelizing across multiple cores, sockets, and accelerators within each node of the cluster.

Efficient Communication: To support modern distributed memory architectures the system must provide efficient partitioning and communication primitives that can move data between remote memories without going through disk and within a single memory without serialization and messaging overheads.

Flexible Programming Model: To support a wide variety of parallel applications the system must provide support for both nested and irregular parallelism from a high-level, user-friendly programming model that is portable to heterogeneous hardware targets.

Previous work has attempted to address some of these components, though we are not familiar with any existing system that satisfactorily addresses them all. Such systems often fall into one of two categories: domain-specific languages and parallel patterns.

1.2 A Domain-Specific Solution

Domain-specific languages (DSLs) provide much higher-level abstractions than general-purpose languages. For example, a DSL for linear algebra may include a built-in `Matrix` construct with operations like matrix multiplication and addition defined as part of the language specification rather provided by a library. While DSLs are most commonly used to improve programmer productivity, such abstractions can also serve to improve performance by abstracting over low-level hardware details [25, 13, 76, 50]. Since the core language constructs in DSLs tend to be much more coarse-grained than in general-purpose languages, the compiler has more freedom to implement operations efficiently for different hardware targets. For example, a linear algebra DSL

can simply include multiple hardware-specific matrix multiplication implementations internally and select between them, whereas a general purpose language must resort to the largely intractable problem of analyzing a given library implementation and attempting to automatically transform it.

By sacrificing generality, users instead obtain both productivity (high-level abstractions allow programs to be written once in a portable manner) and performance (the programs are compiled to run in parallel on heterogeneous systems). SQL is one of the best mainstream examples of this approach. At the expense of a restricted programming model, users can write high level, unoptimized queries and get high-performance implementations automatically. In recent years several new performance-oriented DSLs have been introduced in the data analytics space, covering domains such as machine learning [76] and graph analytics [29, 77] that target hardware such as multicore, GPUs, and clusters.

However the mechanics of performing the required transformations to achieve performance comparable to hand-optimized implementations are in no way obvious or simple for the developer of a given DSL to implement. Furthermore, the inherent need for multiple DSLs to cover the space of performance-limited applications means that multiple DSLs will likely require a large set of similar optimizations. For these reasons, researchers have also sought more general high-performance programming solutions. While increasing generality only increases the complexity of the system developer's task, it also increases the space of applications that can benefit from the system. An increasingly popular approach in this direction is to augment existing languages with *parallel patterns*.

1.3 A Pattern-based Solution

Separate from DSLs, many new parallel and distributed systems have been introduced in recent years that provide higher-level programming models, but they have typically been designed to tackle a particular piece of this heterogeneous computing landscape rather than the entire space. General-purpose languages including Java, Scala, and Haskell, now contain collections libraries in which data-parallel operations are automatically parallelized across multiple cores [64, 43]. Cluster-centric libraries

have also been introduced, such as MapReduce [24], Dryad [40], and Spark [87].

Unfortunately, however, the performance of these systems is often lackluster due to the additional library abstractions imposing significant overheads compared to low-level implementations in the host language. In fact, they may actually require hundreds of processors to surpass the performance of an optimized sequential C++ implementation [54]! However, we do not believe these overheads to be fundamental to the approach but rather an artifact of the library-based implementation. Delite, for example, provides high-level abstractions similar to Spark but then compiles those abstractions to low-level implementations for a given hardware architecture with performance comparable to hand-optimized sequential implementations [13].

The key shared abstraction in all of these designs is data-parallel patterns over collections. While there is not yet a universally agreed upon set of patterns, classic examples include `map`, which applies a supplied unary function to every element of a collection to produce a new collection of the same arity; and `reduce`, which combines every element of a collection into a single result using a supplied binary function. While the high-level vision of all these systems are similar, the implementation details vary widely. In particular, the generality and breadth of the provided parallel patterns as well as multiple additional restrictions on using those patterns make certain applications much easier to express in certain systems. We compare the programming model features and supported hardware targets of a variety of recent parallel systems in Table 1.1. This table includes multiple features useful for implementing real-world applications that are common in traditional programming models but various parallel systems have chosen to sacrifice. These features include *rich data parallelism*, the ability to use and compose a rich set of data-parallel operations beyond the classic `map` and `reduce`; and *nested programming*, the ability to logically nest parallel constructs. Without nested patterns, users are forced to use a separate (sequential) programming model within a parallel computation. Even if parallel constructs can be logically nested, the system may or may not be capable of exploiting nested parallelism at runtime; we refer to this as *nested parallelism*. Some systems can only parallelize work over a single collection at a time, thereby requiring multiple collections to be somehow joined together a priori. In contrast, systems that support *multiple collections* allow

	Programming Model Features					Supported Hardware			
System	Rich data parallelism	Nested programming	Nested parallelism	Multiple collections	Random reads	Multi-core	NUMA	Clusters	GPUs
MapReduce [24]								•	
DryadLINQ [41]	•	•						•	
Thrust [34]					•				•
Scala Collections [64]	•	•	•	•	•	•			
Delite [13]	•	•		•	•	•			•
Spark [87]	•					•		•	
Lime [2]		•	•	•	•	•			•
PowerGraph [29]					•	•		•	
Dandelion [73]	•	•				•		•	•
Delite + DMLL	•	•	•	•	•	•	•	•	•

Table 1.1: Comparison of programming model features and heterogeneous hardware targets supported by various parallel frameworks in chronological order.

parallel computations to consume multiple parallel collections. This is very useful in certain domains, e.g., linear algebra. Finally *random reads* allow arbitrary read access patterns of parallel collections rather than restricting reads to only the local element. While this feature greatly increases the complexity of data management, it is crucial in domains such as graph analytics. As shown in the table, previous systems have sacrificed many features to successfully expand to new hardware targets, with more exotic and complex hardware often strongly correlated with fewer programming features. For example, Scala parallel collections and Spark are both Scala libraries with very similar data-parallel APIs. However, parallel collections provide the complete set of features described, but only for multicore. While Spark eliminated nearly all of these features entirely for the sake of automatic data distribution across a cluster. We also notice from the table that the systems that are most feature- and hardware-complete (Delite [13], Lime [2], Dandelion [73]) are compilers rather than libraries. In the last row of the table we present the promise of a new system that augments Delite with the additional features required to fully achieve our requirements for a high-performance distributed heterogeneous programming model.

1.4 Optimizing Parallel Patterns for Distributed Heterogeneous Computing

As illustrated in Table 1.1, Delite has previously been limited to exploiting a single level of parallelism on shared-memory architectures. It did not support NUMA or distributed architectures. The primary limitation of Delite's original design is that parallel patterns alone are not always sufficient to obtain portable performance, in particular when the patterns are nested. Unlike with flat pipelines, with nested patterns the optimal traversal order becomes both data layout and architecture dependent. In general, portable performance on heterogeneous architectures requires that the programming model either

1. completely abstract over the low-level implementation differences, thereby providing a unified way of expressing the computation in the application for all targets, and/or

2. capture sufficient semantic information to automatically transform between the various implementations required for each device.

The popularity of parallel patterns is largely due to the fact that they satisfy the first criterion when composed in flat pipelines. The semantics of each pattern are sufficiently high level to be efficiently implemented on a variety of hardware. Nested patterns, however, implicitly express a traversal order, thereby requiring our second criterion (transformation) to be efficiently ported to different architectures. It is this second criterion that has been largely unexplored and unexploited by Delite and other parallel pattern systems, but which we will now explore.

In this dissertation, our goal is to extend a data-parallel programming model to heterogeneous distributed hardware without sacrificing the programming model features or hardware targets in Table 1.1, and without sacrificing single-machine performance. We illustrate the high-level system flow of this extension in Figure 1.1. We introduce the Delite Multiloop Language (DMLL), a new intermediate language of parallel patterns designed specifically to enable complex transformations on nested

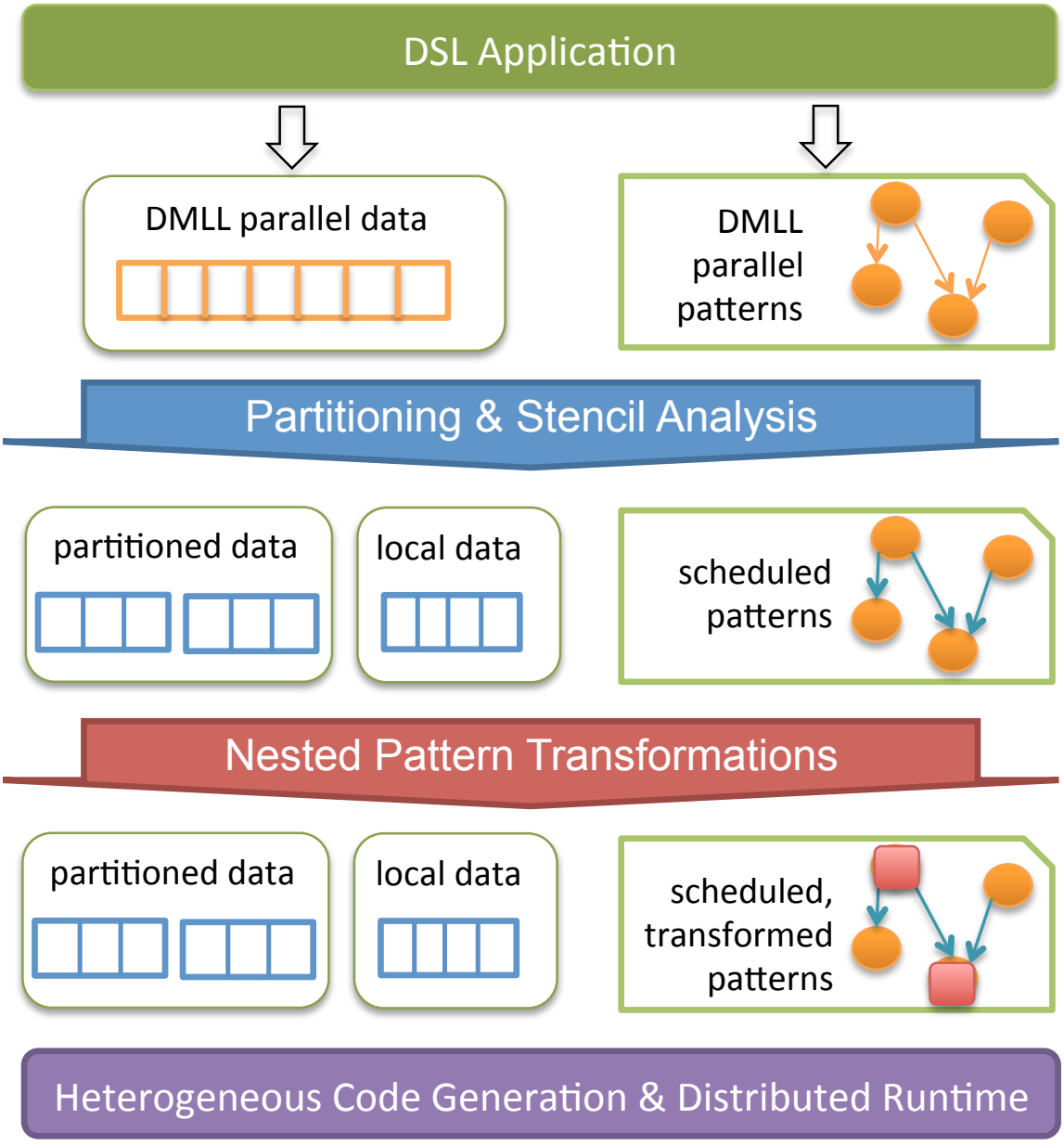

Figure 1.1: Targeting nested parallel patterns to heterogeneous distributed architectures. Applications are lowered to DMLL, compiler analyses decide which data to partition, and multiple transformations optimize the code to improve locality and run efficiently on the target hardware. The runtime then manages data distribution and communication.

patterns using relatively simple rewrite rules. The language semantics are more general than traditional user-facing patterns, enabling more reuse with fewer rules, while still specific enough to capture the necessary semantic knowledge required to efficiently target distributed heterogeneous architectures.

We also introduce a data structure model for Delite based on arrays and compound structures that is familiar to programmers. This model is restricted to enable powerful transformations, but still general enough to cover all existing Delite DSLs and applications. We implement multiple data layout transformations, such as array-of-struct (AoS) to struct-of-array transformations (SoA) using a set of simple rewrite rules that stack to enable complex transformations. These transformations not only improve performance in isolation, but are also key to simplifying the task of analyzing, partitioning, and transferring data structures. A new analysis phase determines what data to distribute based on its access patterns and the locations of its input(s). Affine access pattern analysis is relatively straightforward with our data structure model. From this starting point, we then use DMLL rewrite rules to attempt to improve locality whenever we detect a suboptimal access pattern for the given hardware target.

Once all transformations have occurred, the compiler generates optimized kernels and data structure implementations for each hardware target and passes control to the Delite runtime. We augmented the runtime to handle distributed execution across a cluster by hierarchically extending the existing multicore runtime. We also added specialized runtime garbage collection within multiloops. By exploiting DMLL's restricted semantics, we can often automatically reuse the per-thread local memory allocations within parallel regions, resulting in memory footprints akin to manually allocating temporary reusable buffers. All together these additional transformations enable improved single-threaded performance, greater parallel scalability, smaller memory footprints, transparently targeting distributed memory architectures, and automated data movement and distribution.

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 multiple previous dissertations on ideas and technology that have improved Delite over the years. Tiark Rompf [67] presented Lightweight Modular Staging (LMS), which serves as the core of Delite's compiler and code generation capabilities. Hassan Chafi [18] presented Delite as the basis for "mass market" parallel programming using performance-oriented DSLs. This includes the introduction of Delite ops, which we now refer to as parallel patterns. Arvind Sujeeth [75] showed that several different data analytics domains can be naturally expressed in Delite using parallel patterns and that doing so allows further cross-optimization than is possible with each domain separately. HyoukJoong Lee [46] demonstrated how nested parallel patterns can be efficiently mapped to accelerators such as GPUs and FPGAs.

We will summarize the most important aspects of LMS and Delite presented previously. The core contribution of this work is to demonstrate how to close the remaining gap in Delite's design to providing user-friendly abstractions that transparently and efficiently port to a variety of distributed hardware architectures. Specifically:

- We present a novel intermediate language and a corresponding set of transformations for nested parallel patterns that restructure computation for heterogeneous devices. This enables automatically generating performance-portable code for heterogeneous hardware as opposed to directly translating the algorithms as written.

- We present a novel approach for automatically performing classic data layout optimization using simple rewrite rules that cleanly stack to yield advanced transformations. Splitting and merging compound structures allows reusing existing optimizations as well as simpler generic analyses of data access patterns on each piece.

- We demonstrate that implicitly parallel patterns can be used to automatically

distribute code and data using straightforward analyses of how data is consumed and produced. This is in contrast to existing systems that require explicitly distributed data structures.

- We are the first to show a complete compiler and runtime system that can execute implicitly parallel, single-source applications across a heterogeneous cluster that includes non-uniform memory and accelerators. We present a hierarchical distributed runtime capable of orchestrating application execution split across multiple cores, NUMA nodes, cluster nodes, and GPUs. The runtime provides scheduling, synchronization, data movement, and garbage collection, while the compiler provides optimized kernels for each hardware target.

- We present experimental results for multiple applications demonstrating high performance for automatically-optimized implementations compared to manually optimized implementations using existing state-of-the-art distributed programming models.

1.6 Outline

Chapter 2 introduces necessary background on LMS and Delite and describes the key principles of Delite's design for building DSLs, extending parallel patterns, adding data structures, performing analyses and transformations, and generating code for heterogeneous hardware targets. Chapter 3 presents the Delite Multiloop Language as a base language for describing common data-parallel patterns and how it enables simple expressions of powerful transformations using rewrite rules. In particular we focus on nested pattern transformations, which present unique challenges for performance portability. Chapter 4 illustrates how Delite data structures are internally optimized by eliminating unnecessary components and reorganizing data layouts for different environments. Chapter 5 utilizes Delite's parallel pattern and data structure models from the previous two chapters to demonstrate how implicitly parallel applications can be automatically distributed using static analysis. Chapter 6 discusses our distributed runtime implementation including scheduling, data distribution, hierarchical nested parallelism, and garbage collection. In Chapter 7 we evaluate the

performance improvements of these techniques for sequential, multicore, NUMA, cluster, and GPU execution. We study applications in multiple domains and compare to both high-level systems and hand-optimized implementations. Chapter 8 provides a literature survey of the related work in the fields of high performance languages, parallel programming, and pattern-based transformations. Finally, Chapter 9 summarizes this work and discusses unsolved problems in developing programming models for heterogeneous distributed hardware.

Chapter 2

Domain-Specific Languages with Delite

2.1 Introduction

In this chapter we present an overview of Delite [17, 13], an open-source compiler framework developed by the Pervasive Parallelism Lab at Stanford University in collaboration with the LAMP lab at EPFL. Delite substantially reduces the burden of developing high performance DSLs by providing reusable components common to many languages. We first discuss the generic principles of how Delite is embedded in Scala using Lightweight Modular Staging (LMS) [70, 72]. We then discuss Delite's design and how Delite can be used to build new embedded compilers. Delite is open source and freely available at `http://github.com/stanford-ppl/Delite`.

2.2 Staged Embedded Languages

Delite DSLs are implemented as embedded DSLs within Scala, a high-level functional language that runs on the Java Virtual Machine (JVM). Scala has several features that make it well-suited for embedding DSLs with minimal syntactic overhead for the user. These features include type inference, which hides complicated DSL implementation types from the user, and function currying, which allows standard methods to be called in a syntactically similar way to built-in control structures. Finally, Scala has

very few built-in keywords, allowing most constructs to be overloaded by libraries.

The key distinction between Delite DSLs and traditional libraries is that Delite DSLs construct an intermediate representation (IR) and perform optimizing compilation. Instead of running Scala application code that executes DSL operations, we instead run Scala application code that generates code that executes DSL operations. Thus, Scala is both the language we use to implement our DSL compilers and the language that serves as the front-end for the DSLs. The generated code, however, can be any language (currently Scala, C++, and/or CUDA).

We use a technique called Lightweight Modular Staging, a form type-directed staged metaprogramming, to generate an IR from Scala application code. LMS lifts programs by supplying a new type constructor for all DSL types, called `Rep`. Consider the simple vector example below:

```
val v1: Rep[Vector[Double]] = Vector.rand(1000)
val v2: Rep[Vector[Double]] = Vector.rand(1000)
val a: Rep[Vector[Double]] = v1+v2
println(a)
```

In practice the type annotations shown are elided and inferred automatically by the Scala compiler. This results in user programs that look quite "normal". With the types explicit however, we can see that LMS works by overloading all DSL operations to consume and produce types `Rep[T]` in place of the traditional type `T`. Since all the types are different, the methods on those types actually dispatch to a completely independent method implementation (which just so happens to have the same name as the traditional implementation). These new implementations internally create a linked object graph, one node at a time. The graph resulting from the final operation then serves as the LMS IR. We can see this more explicitly by looking at the implementation of `Vector.rand`:

```
object Vector {
  def rand(length: Rep[Int]): Rep[Vector[Double]] = VectorRand(length)
  case class VectorRand(length: Rep[Int]) extends Def[Vector[Double]]
}
```

The `rand` method returns an object which *represents* the DSL operation to be performed. The object holds a reference to the input `length`, which can be either constant or symbolic, thereby implicitly forming a computation dependency graph. All of these placeholder objects have a common superclass `Def[T]`, which allows LMS to find and traverse the object's dependencies generically. What makes this simple formula so powerful is a combination of Scala's type inference and the fact that almost every operator in Scala is really a method, and therefore eligible for overloading in this manner. There are however a few built-in control constructs that are not overloadable in Scala, including `IfThenElse` and `WhileDo`. We fix this by using a custom compiler extension which desugars these control structures into specially named methods which DSL developers can override as normal [58, 68].

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 as normal.

2. The Java byte code is executed to run the DSL compiler by building the IR, performing optimizations and generating code.

3. The resulting generated code is compiled by the target language's standard compiler.

This extra compilation provides 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 as much runtime penalty since the generated code is free of Scala's higher-level abstractions. In particular, although we typically lift all DSL operations, Scala abstractions like classes, traits, method calls, and closures are systematically removed by the staging process. This can have significant performance benefits compared to ordinary Scala library code.

2.3 Building DSLs with Delite

All Delite DSLs share the same architecture for building an intermediate representation, traversing it, and generating code. Several requirements led to design choices

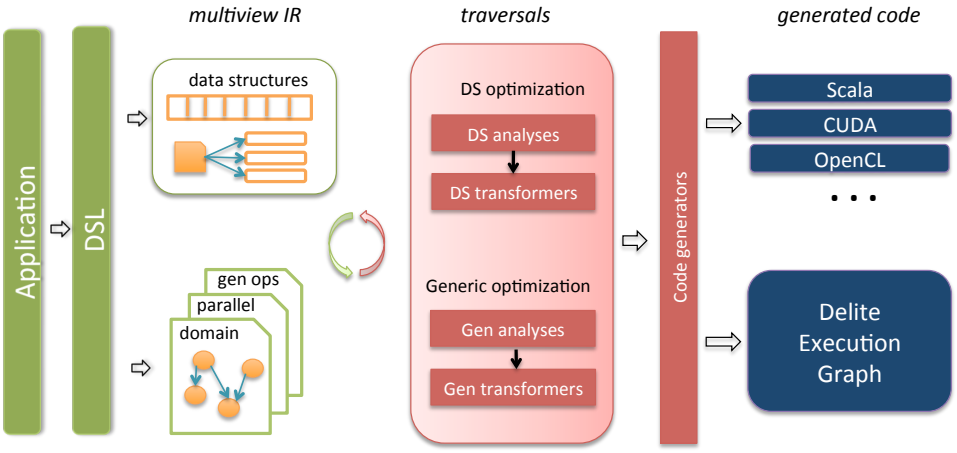

Figure 2.1: Components of the Delite Framework. An application is written in a DSL, which is composed of data structures and structured computations represented as a multi-view IR. The IR is transformed by iterating over a set of traversals for both generic (Gen) and domain-specific (DS) optimizations. 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.

that make Delite's design 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.

Delite provides a suite of parallel operations and data structures that DSL developers can extend in place of the much lower-level LMS abstractions. For example, instead of implementing `VectorRand` as a generic `Def`, in which the DSL developer is entirely responsible for providing the implementation, we can instead implement `VectorRand` as a `DeliteMap`, which provides parallelization and code generation for multiple targets for free. The only piece remaining for the DSL developer is to provide the function for every element (in this case `math.random`).

```
object Vector {
  def rand(length: Rep[Int]): Rep[Vector[Double]] = VectorRand(length)
  case class VectorRand(length: Rep[Int]) extends DeliteMapIndices[Double, Vector[Double]] {
    val func = (i: Rep[Int]) => math.random() // generate a random Double at every index
  }
}
```

We can then implement the Vector itself using Delite's `Record` abstraction which provides structural type composition of Delite's supported primitives.

```
case class Vector[T](length: Rep[Int], data: Rep[DeliteArray[T]]) extends Record
```

`Record` is a marker that denotes the type definition should be generated as a structural type at runtime. The same definition but without **`extends`** `Record` would in constrast be eliminated by staging, leaving only the invididual fields in the IR. In this example, the structure has two fields: `length`, an `Int`, and `data`, a `DeliteArray`. Since both of these types are provided by Delite, this custom Vector definition can be automatically generated for all of Delite's hardware targets. In practice this saves a great deal of effort for the DSL developer even for very simple types like in the example. Defining the type in each target language can be relatively quick, but then defining serialization routines and all the other required metadata to orchestrate data movement in a heterogeneous environment quickly becomes extremely tedious to implement manually.

In the limit that all DSL operations and types can be implemented with Delite's abstractions, the DSL developer is left with zero code generation remaining to implement. For any special cases, the DSL developer can still extend LMS abstractions directly and define their own code generation for those operations.

2.4 Delite IR: Parallel Patterns

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 compound structures of primitives, which must be

declared using a simple API, allowing Delite to reason about and perform optimizations on them.

Delite ops represent reusable parallel patterns such as `Map` and `Reduce`. Each op has specific semantic restrictions (such as requiring disjoint access) that must be upheld by the DSL author. Parallel ops consume and produce a `DeliteCollection[T]`, which is a generic interface for Delite's built-in collections like `DeliteArray[T]`, as well as structures containing a Delite parallel collection, such as `Vector[T]` above. DSL authors are responsible for mapping domain operations to Delite ops, while Delite provides code generators from Delite ops to multiple platforms. This division allows domain experts to focus on identifying parallelism 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 the following parallel patterns:

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 an identity element `A`. The reduce function must be associative. Reduce performs a parallel tree-reduction on the input collection, optimized for each device. The result of `Reduce` is a single element `A`. Reducing an empty collection returns the identity element.

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`s are a collection `DeliteCollection[A]` and a predicate function `pred: A => Boolean`. Filter is implemented as a parallel scan on most devices. The result of a `Filter` is a `DeliteCollection[A]` containing the subset of elements for which `pred` returns `true`, order preserving.

GroupBy: The inputs to `GroupBy` are a collection `DeliteCollection[A]` and two functions `keyFunc: A => K` and `valFunc: A => R`. `GroupBy` processes the input collection in parallel, applying `keyFunc` to each element of the input to determine a bucket and `valFunc` to determine the value appended to the bucket. The result is a `DeliteMap[K,DeliteCollection[R]]`.

GroupByReduce: The inputs to `GroupByReduce` are a collection `DeliteCollection[A]` and three functions `keyFunc: A => K`, `valFunc: A => R`, and `reduce: (R,R) => R`. `GroupByReduce` processes the input collection in parallel, applying `keyFunc` to each element of the input to determine a bucket. It then applies `valFunc` to determine the value to be reduced into the bucket, which it passes into `reduce` along with the current accumulated value for the bucket. The result is a `DeliteMap[K,R]`.

Sort: The inputs to `Sort` are a collection, `DeliteCollection[A]` and a comparator function `comp: (A,A) => Int`. The `Int` represents less than, equal to, or greater than. 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]`.

FlatMap: The inputs to `FlatMap` are a collection `DeliteCollection[A]` and a function that maps each element to an entire collection of elements `mapF: A => DeliteCollection[R]`. `FlatMap` processes the elements of the input collection in parallel and applies `mapF` to each element to create a nested collection. The resulting inner collections are then concatenated by copying their elements into the flat output `DeliteCollection[R]` in parallel.

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`. This allows us to implement optimizations and code generation for a smaller set of primitives than the above without sacrificing any performance optimizations. We discuss the interface and power of the `Multiloop` abstraction in detail in Chapter 3.

2.5 Code Generation

Staged compilation makes generating efficient code much easier. We can use high-level abstractions like types, methods, closures, pattern matching and traits, in both the DSL library implementation and DSL applications. These abstractions are all evaluated, and therefore eliminated, during staging leaving only lifted DSL and Delite operations. 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
   ...
```

Each symbol in the IR is emitted along with its 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 structures, in this case containing scalars and arrays, have been unwrapped and the struct wrapper (normally a class definition) has been eliminated, so each field is passed around independently. By removing the heap allocation for the class wrapper, the remaining primitives are

stack allocated, avoiding the indirection of field accesses. We discuss Delite's data structure optimizations in detail in Chapter 4.

Although all of the DSL operations used to produce this code were polymorphic, the resulting expressions have all been specialized to their actual parameter type in the generated code. Boxing from polymorphic expressions and classes are one of the primary sources of inefficiencies in general-purpose languages such as Java.

Most of the above optimizations actually occur by eliminating abstraction from the host language before it even reaches the IR. Therefore, all Delite code generators benefit from these efficiency gains, and they are strictly guaranteed by the type system. 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++). While the Scala code generator is often a relatively straightforward translation from IR constructs to target language constructs, more restrictive programming models may still require additional target-specific analyses to emit efficient code [47]. Rather than generating a straight-line program, the code generators package each operation into a function, or *kernel*, in the target language. These kernels are then dispatched and executed in parallel across heterogeneous devices by the Delite runtime.

2.6 Runtime

The Delite runtime 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 generated kernel of the application, the execution environments for which the compiler was able to generate an implementation of the kernel (e.g., Scala, C++, CUDA, OpenCL, etc.), and the kernels' input/output dependencies. 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. Sequential kernels are completely specified by the compiler, but parallel ops link in static methods from the runtime that provide information about the local hardware (such as thread ids), dispatch work to available threads, and implement blocking synchronization mechanisms.

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 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. The runtime's hierarchical execution model that exploits nested parallelism across clusters, NUMA machines, and GPUs is discussed in more detail in Chapter 6.

2.7 Existing DSLs

Finally we briefly introduce the primary three DSLs that have been implemented with Delite to-date. These DSLs span the domains of data querying, machine learning, and graph analytics, and are all high performance compared to comparable alternatives.

OptiQL OptiQL is a DSL for data querying of in-memory collections, and is heavily inspired by LINQ [55], specifically LINQ to Objects. OptiQL is a pure language that consists of a set of implicitly parallel query operators, such as `Select`, `Average`, and

```
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 2.1: OptiQL: TPC-H Query 1 benchmark

`GroupBy`, that operate on OptiQL's core data structure, the `Table`, which contains a user-defined schema.

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.

Listing 2.1 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.

OptiML OptiML [76] is a DSL for machine learning designed to handle iterative statistical inference problems. 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. 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

```
1  val kMeans = untilconverged(tolerance){ oldMeans =>
2    // assign each sample to the closest centroid
3    val clusters = matrix.groupRowsBy { row =>
4      // calculate distances to current centroids
5      val allDistances = oldMeans mapRows { centroid =>
6        dist(row, centroid)
7      }
8      allDistances.minIndex
9    }
10
11   // move each cluster centroid to the
12   // mean of the points assigned to it
13   val newMeans = clusters.map(e => e.sum / e.length)
14   newMeans
15 }
```

Listing 2.2: OptiML: k-means clustering

is more important than in other domains where map-reduce has been traditionally used.

The key data types in OptiML programs are `Vector` and `Matrix`. 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`. Listing 2.2 shows an example of the k-means clustering algorithm in OptiML.

OptiGraph OptiGraph is a DSL for static graph analysis based on the Green-Marl DSL [36] but with more functional abstractions. 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. Furthermore, OptiGraph defines constructs

```
1  val pageRank = untilconverged(tolerance){ oldRank =>
2    graph.mapNodes { n =>
3      damp + (1.0-damp) * sumOverNeighbors(graph.inNeighbors(n)){ w =>
4        oldRank(w) / graph.outDegree(w)
5      }
6    }
7  }
```

Listing 2.3: OptiGraph: PageRank

for BFS and DFS order graph traversal, in addition to standard parallel iteration. Users can traverse and reduce over every node in the graph, as well as over a given node's incoming or outgoing neighbors. Listing 2.3 shows an example of the PageRank algorithm in OptiGraph.

Chapter 3

The Delite MultiLoop Language

3.1 Introduction

Parallel patterns have become an increasingly popular choice for abstracting parallel programming and appear in multiple recent systems [34, 53, 43, 24, 87, 13]. Previous work has shown how a relatively small number of parallel patterns can naturally capture a wide range of operations across multiple domains [77]. The key benefit of these patterns is that they abstract over the low-level implementation differences and encode the parallelism and synchronization requirements at a sufficiently high level to be portably mapped to hardware targets as varied as large data centers [24] and an FPGA [2]. The goal of the Delite MultiLoop Language (DMLL) [12] is to capture these parallel patterns in a formalism that is as general as possible without sacrificing critical semantic knowledge that enables transformations. This allows DMLL transformations to capture a large space of programs with relatively few rules while still supporting complex patterns that traditional loop optimization frameworks cannot capture.

3.2 Using Nested Parallel Patterns

Modern distributed systems often provide a parallel pattern API but also enforce certain restrictions that require the user to reason about data movement explicitly.

```
1  //matrix: large dataset being clustered
2  val matrix = Matrix.fromFile(path)(parseMatrix)
3  val clusters = Matrix.fromFunction(numClusters, numFeatures)(
4                                       (i,j) => math.random())
5
6  //shared-memory version
7  //data implicitly shuffled via indexing operation 'matrix(as)'
8  val assigned = matrix.mapRows { row =>
9    clusters.mapRows(centroid => dist(row, centroid)).minIndex
10 }
11 val newClusters = clusters.mapIndices { i =>
12   val as = assigned.filter(a => a == i)
13   matrix(as).sumRows.map(s => s / as.count)
14 }
15
16 //distributed-memory version
17 //data explicitly shuffled via 'groupBy' operation
18 val clusteredData = matrix.groupRowsBy { row =>
19   clusters.mapRows(centroid => dist(row, centroid)).minIndex
20 }
21 val newClusters = clusteredData.map(e => e.sum / e.count)
```

Figure 3.1: Two ways of writing the core computation (one iteration) of k-means clustering using data-parallel patterns.

To better illustrate the challenges of writing applications within such restrictive distributed programming models, consider how we might parallelize and distribute the classic k-means application, which can be implemented very succinctly in two different ways using data-parallel patterns as shown in Figure 3.1. In this example we show a single iteration of the core computation loop for simplicity. We assume that 2D collections (matrices) are available and can implement classic parallel patterns (e.g., `map`, `filter`, `reduce`, `groupBy`) that operate on a single dimension (e.g., `mapRows`). The application assigns each data point to the nearest cluster, and then computes new cluster locations by averaging the data points assigned to each current cluster. With some basic pipeline fusion optimizations our initial implementation performs quite well on multi-core but suffers at larger scales due to the parallelization over the relatively small dataset `clusters`. Attempting to partition the data is also quite challenging since as written an unknown subset of `matrix` is required to compute an element of `newClusters`. Therefore this implementation cannot be directly ported to typical distributed programming models.

We must instead figure out how to rewrite k-means in a more distributed-friendly style, shown in the second half of Figure 3.1. Now all of the outer-level parallelism is clearly over the matrix rows, which are explicitly shuffled for the following average, making the optimal data distribution strategy much more apparent. Once we have found this strategy we now need to rewrite the application to expose this information explicitly to the distributed framework. In Spark, for example, this can be achieved by lowering the matrix to an `RDD[Vector]` where each `Vector` represents a matrix row (an RDD is a linear distributed collection). Unfortunately, performing this lowering makes targeting GPUs very difficult due to the inefficiencies of reducing vector types. For GPUs we instead need a different lowering transformation that transposes the operation and reduces scalars instead of vectors. We present solutions to these issues in the remainder of this section.

Generating efficient implementations of applications written like the above example requires multiple high-level optimizations. Making parallel patterns compose efficiently is often the single most important optimization required. Previous work has considered optimizing across flat pipelines [19, 43], but does not consider nested

parallelism. Nested patterns are extremely useful in terms of expressiveness, but present unique challenges when targeting distributed memory environments compared to shared memory environments, as the nesting structure affects the data access patterns, and therefore how the data must be distributed. In addition the optimal traversal order for many algorithms is actually architecture-dependent. Therefore there is not always one "correct" way for the user to express nested computation, rather the compiler must be capable of transforming it into an optimal structure for each architecture to maintain single-source, architecture-agnostic applications.

With a flat parallelism model (or when the parallelism in the application happens to be flat), sibling loop fusion is often sufficient to achieve high performance across heterogeneous architectures. For the remainder of this section we focus instead on transforming nested parallel patterns, which provide a unique set of challenges. The ability to compose and nest parallel patterns is very important both from a user productivity point-of-view (users don't have to use different operators based on the program context) and from a performance point-of-view (multiple levels of parallelism are exposed and therefore can be exploited), but is often disallowed in cluster programming models due to the need to tackle the various issues highlighted in this section. We first show how we can transform such nested ops into more efficient forms for a given hardware target, and then how we can automatically analyze op access patterns to inform data partitioning.

Many algorithms, for example from machine learning, analyze a large main dataset with respect to one or more smaller dimensions. On modern hardware platforms, the order in which multidimensional data is traversed makes a big difference; for all but the simplest data processing tasks, some traversal orders perform better than others. On clusters, the difference between access patterns is even more pronounced than on shared memory architectures as the access patterns must align with the physical partitioning of data to avoid shuffling data across machines. Unfortunately, many algorithms are more natural to express in a suboptimal traversal order.

3.3 Parallel Patterns as MultiLoops

To define these transformations we present the Delite Multiloop Language (DMLL) as an intermediate language for modeling a wide range of parallel patterns. In this language we represent high-level data-parallel patterns as *multiloops*, a highly flexible loop abstraction that can contain multiple fused loop-based patterns. A multiloop is a single-dimensional traversal of a fixed-size integer range that may produce zero or more values at each iteration. Each multiloop contains a set of *generators*, which capture the high-level structure of the loop body (the parallel pattern) and accumulate the loop outputs. After the loop terminates, each generator returns a result to the surrounding program. Previous work has shown how multiloops lend themselves well to advanced loop transformations, including pipeline fusion and horizontal fusion (returning multiple disjoint outputs from a single traversal) [71]. Each multiloop is typically constructed with a single generator for its body (it returns a single output), but after horizontal fusion may contain multiple generators. Here we extend and modify the previous language to capture nested pattern transformations and make it portable to heterogeneous accelerators. The current set of DMLL generators is defined in Figure 3.2(a). A *collect* generator accumulates all generated values and returns a collection as the result of the loop. This is general enough to implement the classic operations of `map`, `zipWith`, `filter`, and `flatMap`. A *reduce* generator performs on-the-fly reduction with a given associative operation. A *bucket-collect* or *bucket-reduce* generator collects or reduces values in buckets indexed by keys. A *bucket-collect* with only the key function defined is often called `groupBy`.

Each generator contains multiple functions that capture the key user-defined components of the computation. Keeping the components separated rather than composed into a single block in the IR makes it possible to later compose the functions in different ways to generate efficient code for multiple hardware targets. Figure 3.2(b) illustrates DMLL semantics by providing one possible sequential implementation. For example, to perform a *collect* operation on the CPU we can emit code that uses the *condition* block to guard appending the result of the *value function* block to an output buffer. There are many more possible implementations however. For the GPU

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

Figure 3.2: DMLL: Syntax and Types.

we can instead evaluate the *condition* block for all indices up front, allocate an output buffer of the correct size, and then perform a second loop to write the result of the *value function* directly to the correct output location. The *bucket* generators are more complicated to implement but still sufficiently abstract. In particular each implementation can transparently decide to maintain the buckets by hashing (CPU) or sorting (GPU).

In this section we will write examples in pseudocode using standard data-parallel operators available in DMLL. A simple map-reduce example looks like:

```
x.map(e => math.exp(e)).reduce((a,b) => a + b)
```

We translate this language to our DMLL formalism by implementing `map` using a `Collect` generator with an always true condition and similarly `reduce` using a `Reduce` generator:

```
C = Collect_x(_)(i => math.exp(x(i)))
Reduce_C(_)(i => C(i))((a,b) => a + b)
```

We denote the always true condition as `_`. $\texttt{Collect}_x$ is shorthand for a `Collect` over the size of `x` (and similarly for $\texttt{Reduce}_C$). We can then fuse these operations with a simple rewrite rule:

```
C = Collect_s(c1)(f1)          →  Reduce_s(c1&c2)(f2(f1))(r)
Reduce_C(c2)(i => f2(C(i)))(r)
```

Note that `Collect` is more general than `Map` and therefore the rule applies to many more cases than the simple example. In fact we can further generalize this rule to apply to any generator `G` that consumes a `Collect`.

```
[[ Collect_s(c)(f): Coll[V] ]] =
val out = new Coll[V]
for (i <- 0 until s) {
  if (c(i)) out += f(i)
}

[[ Reduce_s(c)(f)(r): V ]] =
var out = ident[V] // init to identity value
for (i <- 0 until s) {
  if (c(i)) out = r(out, f(i))
}

[[ BucketCollect_s(c)(k)(f): Coll[Coll[V]] ]] =
val out = new Coll[Coll[V]]
val map = new Map[K,Index] // hash keys to output indices
for (i <- 0 until s) {
  if (c(i)) out(map(k(i))) += f(i)
}

[[ BucketReduce_s(c)(k)(f)(r): Coll[V] ]] =
val out = new Coll[V]
val map = new Map[K,Index]
for (i <- 0 until s) {
  if (c(i)) out(map(k(i))) = r(out(map(k(i))), f(i))
}
```

Figure 3.3: DMLL Semantics: Sequential Implementations

(GroupBy-Reduce)	A = BucketCollect$_s$(c)(k)(f_1) Collect$_A$(_)(i => Reduce$_{A(i)}$(_)(f_2)(r))	$\rightarrow$	H = BucketReduce$_s$(c)(k)(f_2(f_1))(r) Collect$_H$(_)(i => H(i))
(Conditional Reduce)	Collect$_{s_1}$(_)(i => Reduce$_{s_2}$(j => g(j) == h(i))(f)(r))	$\rightarrow$	H = BucketReduce$_{s_2}$(_)(g)(f)(r) Collect$_H$(_)(i => H(h(i)))
(Column-to-Row Reduce)	Collect$_{s_1}$(_)(i => Reduce$_{s_2}$(c)(f)(r))	$\rightarrow$	R = Reduce$_{s_2}$(c)(fv)(rv) Collect$_{s_1}$(_)(i => R(i))
(Row-to-Column Reduce)	Reduce$_{s_1}$(c)(fv)(rv:(a_1,b_1) => Collect$_{s_2}$(_)(i => r(a_1(i),b_1(i)))) *iff* size(a_1) == size(b_1) == s_2	$\rightarrow$	Collect$_{s_2}$(_)(i => Reduce$_{s_1}$(c)(f)(r))

Figure 3.4: DMLL: Nested Parallel Pattern Transformations.

C = Collect$_s$(c_1)(f_1) $\rightarrow$ G$_s$(c_1&c_2)(k(f_1))(f_2(f_1))(r)
G$_C$(c_2)(i => k(C(i)))(i => f_2(C(i)))(r)

This rule alone captures all traditional pipeline fusion optimizations in DMLL (e.g., `map-reduce`, `filter-groupBy`, etc.).

3.4 Transforming Parallel Patterns

We use the DMLL formalism to express a set of useful transformations on nested patterns, motivated with simple code examples drawn from classic applications in the domains of machine learning and data querying. First consider the following simple aggregation query common in SQL-like languages:

```
lineItems.groupBy(item => item.status).map(group =>
  group.map(item => item.quantity).reduce((a,b) => a + b))
```

As written, it creates a bucket for each `lineItem` status. Each bucket is then summed to return an array of scalar values. However, it is not necessary to construct the buckets first and then reduce them. We can instead transform this computation into a single multiloop with a `BucketReduce` generator:

```
BucketReduce(lineItems.size)(_)(i => lineItems(i).status)
  (i => lineItems(i).quantity)((a,b) => a + b))
```

This version, while less easy to read, performs only a single traversal and reduces the quantities as they are assigned to buckets. We therefore generalize this transformation to the *GroupBy-Reduce* rule in Figure 3.4. This rule matches a `BucketCollect` that is consumed by a `Collect` which contains a nested `Reduce` of each bucket. The expanded

lambda expression `i => Reduce`$_s$`(c)(f)(r)` denotes a pattern match on the `Collect` function, where an enclosing context is implicitly assumed (i.e., the function may contain statements besides the `Reduce`). We can then combine the functionality contained within the `BucketCollect` and the `Reduce` into a single `BucketReduce`. The `Collect` that consumes the transformed `BucketReduce` simply expresses an identity loop over H, but also implicitly contains the remaining (untransformed) enclosing context of the pre-transformed `Collect`. For example, if the application instead averages each group, the division after the sum will remain in the untransformed `Collect` context. In the simple case where the context is empty, this extra identity loop is simply optimized away.

This optimization is often important for high performance because many high level programming languages expose some form of `GroupBy` operation but typically do not expose any kind of "`GroupByAndReduce`" even though it is a quite powerful operation that can express a large class of problems.

We see this pattern appear not only in many data-querying applications but in machine learning applications as well. Consider the k-means clustering algorithm in the second half of Figure 3.1. While the computation is much more complex than the first example, operates on matrices rather than flat arrays, and performs summations over vector types rather than scalar types, the overall outer structure of the computation in terms of parallel patterns is the same and the same rewrite rule applies. The same pattern also appears in the k-nearest neighbors application, which uses grouping to count the fraction of k data samples per data label and select the label with the largest count. In fact the recent widespread success of the MapReduce programming model is a testament to just how often this pattern occurs in real applications.

But what if k-means is written as in the first half of Figure 3.1 rather than the second? As written it is not immediately clear how to parallelize this application across a cluster efficiently, as we have a repeated traversal over a large dataset `matrix` nested inside a traversal over the clusters of `matrix`, and the number of clusters is orders of magnitude smaller than the number of rows in the dataset. The simple strategy of just parallelizing the outer loop can actually perform quite well in a multi-core environment, but for distributed memory it will require the entirety of `matrix` to be broadcast

to every machine computing the updated cluster locations, defeating the purpose of distributing the data across multiple machines. The required transformation is not immediately obvious, and requires introducing intermediate data structures. The pattern to notice is that the application is conditionally reducing values (e.g., reducing some subset of a dataset) within an outer loop where the reduction predicate is a function of the outer loop index. This leads us to the *Conditional Reduce* transformation, which when applied yields the transformed code for k-means shown in Figure 5.2. The transformation breaks the dependency on the outer loop index and lifts the inner reduction out of the loop by pre-computing all of the partial reductions with a single pass over the dataset. Note that we are using the condition function of the original `Reduce` as the key function of the `BucketReduce` so each partial reduction in the original code is accumulated into a separate bucket. Now we traverse the large dataset `matrix` only once to pre-compute sums for each cluster in parallel and store them indexed by cluster. Then a second loop performs just index lookups. It is important to note that even though `ss` and `cs` are expressed as separate traversals loop fusion will merge these two `BucketReduce` loops, along with the loop that computes `assigned`, into a single traversal. In fact, after transformation and fusion take place we end up with the *exact same* optimized code as the result of applying the *GroupBy-Reduce* rule to the `groupBy` formulation of k-means. Furthermore we can now see a straightforward partitioning and parallelization strategy of the large dataset across the cluster. One can also view this transformation as a kind of *push-pull* transformation, which is common to many domains, including graph algorithms, which must decide whether to push or pull node data across edges. The original version of k-means parallelizes over the cluster centroids and pulls potentially remote samples to the local centroid, while the transformed version parallelizes over the samples and pushes local samples to the potentially remote centroids.

3.5 Optimizing for Heterogeneous Architectures

Rather than flattening nested patterns it is often necessary to instead interchange their order. Consider the example of logistic regression. The textbook description translates into parallel patterns in a straightforward way:

```
1 val newTheta = Range(0, x.numCols) map { j =>
2   val gradient = Range(0, x.numRows) map { i =>
3     x(i,j)*(y(i) - hyp(oldTheta, x(i)))
4   }.sum
5   oldTheta(j) + alpha * gradient
6 }
```

For each feature (column) `j`, the algorithm computes a gradient of the sample dataset in a nested summation loop. Unfortunately, this implementation can be rather inefficient. In practice, the number of samples (rows) in the dataset is orders of magnitude larger than the number of features, and as such it is the samples that should be distributed. As written however this algorithm requires every sample to be available for summation for every feature (`x(i)` in the code returns an entire sample (matrix row)), therefore every sample needs to be broadcast to all of the processors computing the summations. This detail is easy to ignore in shared memory environments, but critical when attempting to generate a reasonable distributed implementation. Therefore it is much more efficient to parallelize over the samples, traversing the big data set only once and accumulating results for each feature in parallel. The implementation requires several individual transformations: fissioning the imperfectly nested loop, interchanging the nested loops, and vectorizing the reduction. The final result is shown below:

```
1 val gradientVec = Range(0, x.numRows) map { i =>
2   Range(0, x.numCols) map { j =>
3     x(i,j)*(y(i) - hyp(oldTheta, x(i)))
4   }
5 }.sum
6 val newTheta = Range(0, x.numCols) map { j =>
7   val gradient = gradientVec(j)
8   oldTheta(j) + alpha * gradient
9 }
```

This version makes crucial use of the fact that our multiloop reduction facility is not limited to scalar values but is also able to reduce collections, using `Collect` (i.e., `zipWith`) to implement vectorized addition. Instead of constructing a vector of sums, we are computing a sum of vectors. ; or in matrix terms, rather than summing each column

individually, we are summing an entire row at a time.

Despite the seemingly complex changes to the application code, we can generalize this variety of loop interchange with the simple *Column-to-Row Reduce* rule in Figure 3.4. In this rule `fv` and `rv` denote vectorized versions of `f` and `r` which are implemented by wrapping each scalar function with a `Collect`. Once again the transformed `Collect` contains the remaining untransformed enclosing context of the original `Collect`. Applying this rule to the logistic regression example provides a better traversal pattern when targeting multiple CPUs as well as a better way of partitioning the dataset `x`. However when considering GPU execution the original traversal order shown was actually superior since it reduces scalars rather than vectors and reducing non-scalar types on a GPU is typically very inefficient due to limited shared memory capacity. Fortunately we can also express the inverse transformation to a *Column-to-Row Reduce*, the *Row-to-Column Reduce* in Figure 3.4, to invert the loops to create multiple scalar reduces.

To generate code for clusters of GPUs for this example it is therefore necessary to perform a *Column-to-Row Reduce* transform for parallelization across the cluster and then apply the *Row-to-Column Reduce* when generating the GPU kernel implementation, effectively distributing over samples (rows) and then summing over features (columns) within each node. This is always possible as the two transformation rules are completely reversible, which is straightforward to show by simply substituting one into the other and reducing identities. The *Row-to-Column Reduce* allows us to generate far more efficient GPU code for the k-means example as well, which also reduces vectors as written. We have observed this rule to be useful in many applications in which the user wishes to somehow reduce the columns of a matrix. Examples in machine learning include ridge regression and Naïve Bayes. We refer the reader to previous work [47] for details on how to generate GPU kernels from nested patterns after these transformations have been applied.

Any language which employs these techniques must either support a single canonical way of structuring the computation and then automatically transform the code from the canonical form to target-specific implementations, or even better the user should be able to write the application either way and have the compiler restructure

it for each target by applying, in this example, either a *Column-to-Row Reduce* or a *Row-to-Column Reduce*, as appropriate.

, which constructs a vector via a nested summation to compute the variance over samples for each feature, and Naïve Bayes, which constructs the spam classification vector via a nested summation of the subset of the data samples which have been labeled as spam.

```
1 val phi_spam = (0::x.numFeatures) { j =>
2   val spamWordCount =
3     sumIf(0,m){ i => x.labels(i) == SPAM }{ i => x(i,j) }
4   val spamTotalWords =
5     sumIf(0,m){ i => x.labels(i) == SPAM }{ i => words(i) }
6   (spamWordCount + 1) / (spamTotalWords + x.numFeatures)
7 }
```

This example appears slightly complicated by the fact that only a subset of the samples in the dataset contribute to the final result, but fusing the filter into the condition block in the `Reduce` generator, which decides whether or not the value for a given loop index should be passed to the generator for reduction.

3.6 Discussion: Designing Transformations

In many situations there is no single "correct" way to structure the application efficiently for heterogeneous devices, however the high-level semantics of DMLL still provide sufficient information to automatically transform to an optimized representation per device. The rules in Figure 3.4 represent highly recurring patterns in the applications we have studied within the domains of machine learning and data querying, and using these few generator patterns, it is easy to add new rules for other powerful optimizations as they are needed. Ultimately what these transformation rules enable is a more relaxed end-user programming model. Users can write applications in a straightforward way and still obtain portable performance to multiple architectures. While in general the system can always fall back to warnings or errors that require the user to manually restructure the application, we believe that it is important that this transformation facility be extensible by DSL authors, power users, etc. to help ensure that such errors occur as infrequently as possible.

Since these transformations are meant to improve locality, the only missing piece for deciding when to apply them is calculating the original and resulting memory access patterns and knowing which is a better access pattern for the target hardware. We perform this step for distributed memory in the following section. For the GPU we always perform a Row-to-Column Reduce when possible since it enables utilizing shared memory.

For example, the rules presented here do not completely capture *push-pull* transformations on graph traversals as both the original and transformed code will still have an `Unknown` stencil. However a graph analytics DSL built on top of such a parallel pattern language should be able to add logic to enable the `Conditional Reduce` transformation for graph applications and still leverage all of the other benefits of using the host language rather than having to rebuild the entire infrastructure from scratch.

Chapter 4

Data Layout Optimizations

4.1 Introduction

In addition to optimizing computation, Delite also automatically optimizes data structures for different hardware targets. Just as Delite raises computation abstractions by making parallel patterns language primitives, it also raises data structure abstractions by making parallel collections primitives. Generating efficient low-level code can then be decomposed into two concerns: implementing each built-in collection type efficiently on the current hardware target, and transforming user-defined composite types into efficient sets of primitive types.

4.2 Restricted Data Structure Model

Delite supports a core set of primitive data types and allows users to compose primitives types using a traditional record (structural type) model. Since Delite aims to automatically target a large set of low-level programming models such as C and OpenCL, all code is statically dispatched and there are no object abstractions. We support several standard primitive types, such as `Int`, `Double`, `Boolean`, and `String`. In addition, we also support collections such as `DeliteArray` and `DeliteMap`.

Collections are all implicitly parallel and are purposefully abstract to allow more

implementation freedom. For example, a `DeliteMap` could be maintained internally using either hashing or sorting. A `DeliteArray` could map directly to the target programming model's array abstraction, to an offset pointer within a (larger) pre-allocated memory block, or to a set of native arrays each within a separate memory region.

Each field of a `Struct` is restricted to be one of these primitive types or another `Struct`. `Struct`s can be deeply nested, but to enable multiple analyses and optimizations, they cannot contain circular references.

With this model, we can automatically generate the implementation of every data structure in a Delite program. Scalar primitive types are mapped directly to the low-level programming model's equivalent type, and primitive collection implementations are included for each target as a static library that's linked into the compiler-generated program. Finally, custom struct definitions are automatically generated and included with the generated kernels. In addition to the definitions, Delite also generates custom transfer functions that can serialize and copy types between address spaces (e.g., between the JVM, C++, and CUDA) on the same or different machines. We currently use Google Protocol Buffers [30] to implement efficient serialization across machines. In addition to simply generating a correct implementation, we can also exploit this model to optimize the user's structures.

4.3 Structure Optimizations

Using ideas from previous work [71], we implemented multiple data structure optimizations including struct unwrapping, dead field elimination (DFE), and array-of-struct to struct-of-array (AoS to SoA) transformations, which work together to reduce complex data structures to simple arrays of primitives. In addition to yielding more efficient generated code by removing indirections and enabling vectorization, these optimizations greatly simplify data structure partitioning and distribution. By unwrapping structs, any arrays contained within are directly exposed and the analysis can reason about each primitive array individually.

We will first look at the simpler case of struct unwrapping and dead field elimination. In isolation, unwrapping structs and instead using each primitive field individually has one important consequence in JVM languages: it transforms heap allocations

of objects into stack allocations of primitives. Given this goal to eliminate struct allocations, we also have a way to perform dead field elimination for free. Once the struct is unwrapped, each field of the struct becomes an independent symbol in the IR, and if any of those symbols are unused they will be eliminated by the compiler's standard dead code elimination.

This optimization can be performed very elegantly and simply using LMS pattern matching rewrite rules. We show here the case where the struct is immutable and therefore can be eliminated freely. Delite also supports mutable structs, however they cannot be safely optimized away in this fashion.

```
// Struct creation
// Create an IR node that holds a map from field names to the symbols for the fields
def newStruct[T](elems: Map[String, Sym[Any]]): Sym[T] =
  Struct[T](elems) // Emit a struct allocation in the generated code

// Read a field value from the struct
def fieldRead(struct: Sym[Any], field: String) = struct match {
  case Def(Struct(elems)) => elems(field) // Return the original symbol for the field
  case _ => FieldRead(struct, field) // Emit a field access in the generated code
}
```

With this simple definition and rewrite rule, every field access of the struct attempts to bypass the struct object and use the original field symbol directly. If the bypass succeeds for every read, the `Struct` allocation node becomes unused and is eliminated by dead code elimination. Once the struct allocation is dead, any field symbol passed into the struct allocation that is never read out of the struct also becomes dead code and is eliminated, effectively performing dead field elimination.

4.4 Array of Structure Optimizations

Optimizing the layout of structs containing arrays can yield significant performance gains. There are traditionally two standard layouts to choose from. The array-of-struct (AoS) or row-store format uses a single array where each entry in the array is an instance of the struct. The struct-of-array (SoA) or column-store format instead uses a separate array for each field of the struct. For nested structs, this transformation is

traditionally applied recursively, so that the element type of every array is a primitive type.

A great deal of research has been done into the trade-offs of these two formats. For Delite, we are interested in two key benefits in particular. First, JVM languages cannot allocate a flat array of objects, only an array of pointers to objects. Therefore, the SoA format is the only one which produces large blocks of contiguous memory. Second, the SoA format is preferred for vectorization, which many of Delite's parallel patterns naturally benefit from. However the AoS format is much more familiar and easy to use for users. Delite therefore transforms AoS to SoA automatically. We implement this with some relatively simple rewrite rules on all built-in `DeliteArray` operations.

```
// Example array creation: Fill an array by evaluating a function over each index in [0, length)
def arrayFromFunction[T](length: Sym[Int], value: Sym[Int] => Sym[T]) = value match {
  case Def(Struct(elems)) =>
    struct[Array[T]](elems.map { case (k,v) => (k, arrayFromFunction(length, value)) }) // SoA
  case _ => ArrayFromFunction(length, value) // Emit a primitive array in the generated code
}

// Read the element of the array at the given index
def arrayRead[T](a: Sym[Array[T]], i: Sym[Int]): Sym[T] = a match {
  case Def(Struct(elems)) =>
    struct[T](elems.map { case (k,v) => (k, arrayRead(v, i)) }) // SoA
  case _ => ArrayRead(a, i) // Emit an array index read in the generated code
}
```

Similar rewrite rules are required on any other core array operations. They all follow the same flow: whenever a logical array operation occurs, check if the logical element type of the array is a struct. If so, instead return a struct whose fields are formed by calling the same array function recursively. Once the element type is no longer a struct, emit the actual array operation.

The `arrayRead` operation illustrates how Delite maintains an AoS abstraction for the user. The logical operation is that the user reads an `Array` of structs at index `i` and gets back a `Struct`. The physical operation is that every underlying field array is read at index `i` and the results are then packaged together into a `Struct` to return

to the user. On the surface this looks inefficient, as every array access must still allocate a temporary struct. However, when this rewrite is combined with the struct unwrapping rewrite shown previously, all temporary structs are eliminated! And in this instance, any DFE that occurs after struct unwrapping is applied to entire arrays! The generated code therefore simply contains primitive arrays with no structs contained within or wrapping the arrays.

4.5 Generating Arrays from Multiloops

Delite's parallel patterns often encourage a logical array of structures layout even more so than traditional loops since they lack an explicit loop index to access multiple data structures simultaneously. As discussed in Chapter 3, however, the lower-level multiloop abstraction is capable of closing over an arbitrary number of parallel collections. This feature makes consuming an SoA'd collection in a multiloop trivial: rather than closing over a single array-of-structures, the multiloop encloses a set of independent primitive arrays instead.

However, generating an SoA'd collection as the output of a multiloop requires more work if we want to return multiple individual arrays rather than a single structure of arrays that must then be unwrapped. To start, we apply the same rewrites rule shown above above, following the pattern of the `arrayFromFunction` example, expanded to include all the input parameters of a multiloop as described in Chapter 3.

This rewrite removes all temporary structs, but leaves us with a set of independent loops that each generate a single array rather than a single loop that generate multiple arrays. To rectify this, we then apply Delite's standard loop fusion algorithm, to recreate a single loop that produces all the values simultaneously. Note that this use case is a trivial example of *horizontal* loop fusion provided by LMS [71]. For sanity, we also override the default algorithm for deciding the order in which to fuse loops. Loops split by an SoA transform are always recombined first to maintain, at minimum, the user's original traversal order. The fused results are then passed on to the generic fusion algorithm which discovers and performs any additional fusion as always.

4.6 Implications

All of the optimizations discussed in the past two chapters, including struct unwrapping, struct-of-array transformations, loop fusion, nested loop transformations, lambda inlining, and dead code elimination, work together to produce an output program that is far more optimized and simplified than any of these rewrites are capable of achieving in isolation. The end result of these transformations is often generated code which contains only tight loops over primitive arrays, which no custom structures or pointer indirections within or surrounding the arrays.

This simple final form is the `key` factor that makes several lower-level optimizations *just work*. Array access stencil analyses, for example, are relatively straightforward on raw arrays and indices, but very difficult when additional pointers become involved. After implementing this optimization, existing auto-vectorization tools became much more effective and reliable when applied to Delite's generated code. We also utilize a post-SoA stencil analysis to perform automatic data partitioning and distribution, as discussed in the next chapter.

Chapter 5

Targeting Distributed Systems

5.1 Introduction

The biggest challenge in terms of making data-parallel applications portable to heterogeneous hardware is that parallel patterns primarily abstract over the computation, but not the data structures. When targeting uniform shared memory systems this issue is easily ignored, but when targeting more complex memory systems (NUMA machines, clusters, etc.) the task of deciding how to distribute the data is more complicated. Many applications have both "large" collections that should be distributed and "small" collections which should only be computed by a single machine or be broadcast to every machine. It is for this reason that existing systems typically require the user to manually decide how data is distributed. and then use the flexibility of the parallel patterns to move the computation to the data. This essentially introduces two kinds of types within applications, distributed collections and local collections. Returning to our k-means example in Figure 3.1 this amounts to deciding that `matrix` should be distributed and `clusters` broadcast and then use different types and operations for each data structure even though they are both logically matrices, just very different sizes. In this section we show how we can use straightforward analyses and the DMLL rewrite rules to automatically transform this example to the explicitly fused and distributed implementation that a programmer would normally write manually. We determine how to partition data structures based on how the

parallel patterns consume and produce them, thus allowing the user to use a single collection type and uniform API throughout the application.

5.2 Data Partitioning Analysis

To decide which data structures to partition in a program we first need to know whether each input dataset should be partitioned. We obtain this information by having the user annotate each data source (e.g., the file reader operations) with this information. Alternative approaches include using JIT compilation when the inputs are available or exploiting domain knowledge. How exactly this information is obtained does not affect the rest of the discussion. Once we have this information we use a forward dataflow analysis (Algorithm 1) that uses the principle of "move the computation to the data" to decide whether or not every other data structure should be partitioned based on where and how they are produced. `Local` means the collection should be allocated entirely in one memory region and `Partitioned` means the collection should be spread across multiple memory regions (NUMA regions, machines, etc.). We use the notion of partitions to abstract over the different physical memory layouts. When generating code for a NUMA machine, the data structure is partitioned across each socket's memory and when generating code for a cluster it is distributed across machines. (and for clusters of NUMA machines, partitioned across both).

If a parallel operation consumes only `Local` data, then its output is also `Local`. If instead the pattern consumes one or more `Partitioned` collections, the pattern itself (the code) and any other local inputs will be broadcast to the partitioned data. Whether or not the output data is created as `Partitioned` is determined by the type of pattern (i.e., a `map` generates a `Partitioned` output but a `reduce` generates a `Local` output). Note that we have said nothing about the data access patterns yet. Even though the parallel pattern is partitionable, it may still require a significant amount of communication between partitions to execute. Sometimes the communication is fundamental (e.g., graph applications), but other times it can be avoided by restructuring the computation.

Applying this algorithm to k-means we end up with the transformed code in Figure 5.1, where data structures have been explicitly lowered to `PartitionedArray` or

Algorithm 1 Pseudocode for Partitioning Analysis.

```
1: DataLayout ::= Local | Partitioned
2: Stencil ::= Interval | Const | All | Unknown
3: transforms: List[RewriteRule]
4: Input: layouts: Map[Sym,DataLayout]
5: //all layouts initialized to Local unless specified otherwise
6: procedure TRAVERSE(op)
7:     if layouts(inputs(op)) contains Partitioned  then
8:         if isParallel(op) then
9:             CheckInputStencil(op)
10:            if outputIsPartitionable(op) then
11:                layouts(output(op)) := Partitioned
12:        else
13:            if not isWhitelisted(op) then
14:                warn()
15: procedure CHECKINPUTSTENCIL(op)
16:    stencils := ComputeInputStencils(op)
17:    if stencils contains Unknown then
18:        for transform in transforms do
19:            xformed := transform(op)
20:            newStencils := ComputeInputStencils(xformed)
21:            if not (newStencils contains Unknown) then
22:                op := xformed return
23:        warn()
24: procedure COMPUTEINPUTSTENCILS(op)
25:    //for each array read operation within op,
26:    //use standard affine analysis to classify as one of Stencil patterns
```

`LocalArray`. The two inputs `matrix` and `clusters` are given as `Partitioned` and `Local`, respectively. Pipeline fusion eliminates multiple intermediate data structures including the result of `mapRows` on line 4, as well as `as` and `matrix(as)` on lines 8-9. `sum` is determined to be `Local` since it is the output of a `reduce` and each `map` on lines 3, 6, and 10 inherits the same type as its input. The only problem with this lowering is the random access of partitioned `matrix` within a local loop. We can detect such problematic cases automatically using a standard read stencil analysis.

```
1  val matrix = PartitionedArray.fromFile(parseMatrix)
2  val clusters = LocalArray.fromFunction(...)
3  val assigned: PartitionedArray = matrix.mapRows { row =>
4    clusters.mapRows(c => dist(row, c)).minIndex //fused
5  }
6  val newClusters: LocalArray = clusters.mapIndices { i =>
7    //note: PartitionedArray matrix accessed at dynamic indices
8    val as = assigned.filter(a => a == i)
9    val sum: LocalArray = matrix(as).sumRows //fused with line 8
10   sum.map(s => s / as.count)
11 }
```

Figure 5.1: k-means after the data partitioning analysis

5.3 Read Stencil Analysis

In order to automatically determine how to partition input data across ops, we perform a stencil analysis to statically detect the (conservative) range of a `DeliteArray` an op may access. The analysis is shown in Algorithm 2. The result of the analysis is a map from ops to *stencils*, where a *stencil* is a map from `DeliteArrays` to *access patterns*. The stencil is passed to the runtime, which uses the information to determine a schedule for data movement across nodes to distribute the op across the cluster.

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 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 this case, we either have to replicate the array across nodes, or be prepared to tolerate remote accesses during the op's execution, which will result in additional communication latency. 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.

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

For example, if we see a `One` access followed by an `Interval` access, we record `Interval`.

The key insight of our stencil analysis algorithm is that it is simple, and that simplicity is enabled by being able to reason about parallel operators and DSL data structures as first class citizens. The input to an op can only be a `DeliteArray`, a primitive, or a struct. Structs are unwrapped before reaching the analysis, so we only have to reason about array inputs to ops. The parallel operators we currently distribute require their functions to be pure, which ensures that the stencil is read-only. We use symbolic pattern rewriting to simplify arithmetic operations like `i+Const(0)`, which can arise due to generic operations on indices. These rewritings simplify the pattern matching required to extract the components of the array access (for example `s2` in `j*s2` on line 15 in Algorithm 2).

```
1  val matrix = PartitionedArray.fromFile(parseMatrix)
2  val clusters = LocalArray.fromFunction(...)
3  def assigned = i => clusters.mapRows(centroid =>
4    dist(matrix(i), centroid)).minIndex //fused
5  )
6  //lines 7-8 horizontally fused
7  val ss = bucketReduce(true, assigned, i => matrix(i), _ + _)
8  val cs = bucketReduce(true, assigned, i => 1,         _ + _)
9  val newClusters: LocalArray = clusters.mapIndices { i =>
10   val (sum: LocalArray, count: Int) = (ss(i), cs(i))
11   sum.map(s => s / count)
12 }
```

Figure 5.2: k-means after nested pattern transformations. This is how the program is traditionally manually written for high performance in distributed systems.

5.4 Applying DMLL Transformations

Using the stencil analysis we can statically predict if partitioning may require non-local accesses. If any stencil is `Unknown` we attempt to apply a set of rewrite rules to improve the access patterns. If any of the rewrites succeed in replacing the `Unknown` stencil with `Interval` we replace the pattern with the transformed version. The set of rewrites we currently consider are those presented in Figure 3.4. These rules do not overlap and we only try to apply a single rule at a time rather than an exponential combination of them, which keeps the search space linear and order-independent. If all available transformations fail, we fall back to transferring data at runtime. We mark this last case with a warning to the user since the communication may be expensive.

In the case of k-means, line 9 creates an `Unknown` stencil for `matrix`, which triggers the `Conditional Reduce` rule, resulting in the code shown in Figure 5.2. The transformation allows `assigned` to be pipeline-fused into the `bucketReduce` and both `bucketReduce`s are horizontally fused into a single traversal over the partitioned `matrix`. Furthermore the data structures read inside the loop on line 10 now have simple access stencils of `Interval` rather than `Unknown`. As discussed in Section 3.4, this implementation is an equivalent but more optimized version of the original distribution-friendly k-means snippet shown in Figure 3.1.

5.5 Sequential Operations

Finally we consider the case where a partitioned collection is consumed by a sequential operation. We make the conservative assumption that in addition to the well-structured parallel patterns the programming model also allows arbitrary sequential code with arbitrary effects. Therefore only the parallel patterns can be distributed and the sequential code must run in order at a single location. Allowing these operations reduces the portability of the programming model. For example, when targeting a single shared-memory machine it may be perfectly reasonable for a sequential operation (e.g., `print`) to consume the collection, but in a distributed environment this would require streaming a data structure that is much too large to fit in a single memory. Disallowing these operations from consuming partitioned collections regardless of the hardware target makes the model portable but is overly restrictive for simpler targets. We believe the best practical solution is to simply handle these operations differently based on the target. If compiling for multi-core allow the operation, and if compiling for clusters disallow it. An alternative could be to only allow these operations in "debug" mode. In the algorithm we mark this case with an abstract `warn()` to the user. based on the hardware target (warning, error, etc). We also relax the restriction slightly by allowing whitelisting of operations that the compiler developer knows to be safe. For example, determining the size of a collection often doesn't require dereferencing the collection data but is instead stored as an additional field. Therefore reading that field is always allowed.

Chapter 6

Heterogeneous Runtime

6.1 Introduction

The final phase of Delite's execution is to schedule and link the generated kernels together to form a complete program. The compiler is responsible for providing all the code for kernels, data structures, and communication for as many hardware targets as possible. The runtime then distributes the kernels across whatever hardware resources are available in the current machine(s) and manages passing data amongst kernels. During execution, it dynamically determines how many resources to assign to each multiloop based on its size, load balances the multiloop's execution amongst those resources, and reclaims available memory at the end of multiloop execution.

6.2 Hierarchical Execution

The key insight to supporting a large number of execution environments within a single system is that a `Multiloop` is agnostic to whether it runs over the entire loop bounds, a subset of the loop bounds, or even a single index of the loop. We can therefore distribute multiloops across heterogeneous hardware hierarchically. We must first independently implement the core support for running an arbitrary chunk of a multiloop on each desired hardware target. We can then overlay distribution logic which decides how to best (further) divide the loop at each step.

For cluster execution, this translates into the following flow: The cluster master

first chooses whether to run a given multiloop locally or to distribute it across the cluster slaves (see Chapter 5 for how this is determined). If distributed, the master divides the total loop bounds into chunks and assigns each slave one or more chunks. Slaves are assigned chunks by combining the input data's access stencil with the physical location of each logical index to ensure all reads are local (we move the computation to the data). Each machine can then choose to further partition its chunk(s) across however many sockets, cores, and/or GPUs it has access to using similar logic but optimized for the local environment.

For example, the multicore partitioner puts less overall weight on the importance of data locality when dividing work amongst threads. Each thread is still assigned a contiguous chunk of loop indices to optimize cache locality and hardware prefetching. However, it also creates significantly more chunks than threads and implements dynamic load balancing across threads by assigning each thread a single chunk at a time. This provides much better scaling for irregular applications. The ratio of chunks to threads can be tuned by the user and even effectively disabled for a specific application if desired.

6.3 Nested Parallelism

Unlike many other parallel runtimes, parallel execution is not restricted to a single level of `Multiloop` in Delite. When multiloops are nested, each level is eligible for further partitioning and parallelization. This feature greatly aids Delite's goal of single-source, high-performance programs with relatively little additional implementation effort. Without nested parallelism, more compiler transformations would be required to ensure that the biggest loop is always the outermost loop. This is of course very hard to determine statically and not always achievable if all of the loops have a relatively small domain each.

Our runtime scheduler follows a simple rule: only attempt to parallelize the inner loop when there is insufficient parallelism (number of loop indices) remaining in the outer loop to saturate the available hardware. Consider for example a multiloop with domain `[0,4)` that contains another multiloop over a very large domain. Note that the bounds in this example do not have to be statically known as the runtime distributes

loops just-in-time (after the bounds of the loop have been evaluated). On a 4-core machine, only the outer loop will be parallelized, while on an 8-core machine, the outer loop will be parallelized by 4 and the inner loop by 2. In a cluster (or NUMA) environment, the outer loop may be parallelized across 4 machines (sockets), and the inner loop across all available threads within the machine (socket).

6.4 Parallel Memory Management

In addition to scheduling and data movement orchestration, the runtime performs automatic memory management for all hardware targets. For some targets this is handled directly by a lower-level runtime (e.g., the JVM), but for others we have implemented custom solutions optimized (and simplified) for Delite's programming model.

In this section, we focus specifically on memory management for large NUMA machines, which are currently handled relatively poorly by existing general-purpose implementations. Traditional heap allocators do not consider NUMA socket locality when placing objects and do not automatically split large allocations across sockets. We rectify this by creating thread-local heaps for small allocations and pinning threads to cores to ensure each thread's memory remains local. We then initialize large allocations using multiple parallel threads to ensure the allocation is physically partitioned across multiple threads' memory regions.

Automatic garbage collection of dead allocations requires additional analysis. We statically determine when to attempt to free memory and then dynamically determine what can be freed using a traditional mark-and-sweep approach. Having statically determined collection points allows us to compute the symbolic root set of pointers for the mark-and-sweep at compile time in a hardware-agnostic manner.

We also specialize this approach by making another key observation about `Multiloop` semantics: the only way for data to escape a multiloop is to be formally returned at the very end of an iteration, and all iterations are independent. Therefore a very good choice for a static garbage collection point is at the very end of each multiloop iteration. In practice we've found that after SoA optimizations, most multiloops generate arrays of primitives (and therefore return a single primitive each iteration).

Since primitives are copy-by-value, the root pointer set is empty in these cases, and we can simply reset all temporary memory at the end of each iteration with very little overhead. In fact, while functional programming models traditionally suffer from performing extensive allocations, we can effectively recover the more imperative implementation of allocating temporary memory once outside the loop and reusing it each iteration using this approach. Furthermore, since we have thread-local heaps, we automatically obtain the more complex parallel version of this optimization, which is to pre-allocate a separate temporary buffer per thread.

Chapter 7

Evaluation

In this chapter we investigate the performance impact of the optimizations discussed in this dissertation. We present performance results for a suite of applications. Each application was implemented using one of the three main Delite DSLs: OptiML (machine learning), OptiQL (data querying), or OptiGraph (graph analytics). Each implementation is therefore a high-level, concise, and functional description of the application. In contrast the alternative implementations for each application were all heavily hand-optimized to the best of our ability given the constraints of the alternative programming model.

All of the Delite's existing optimizations were enabled for each application, except where explicitly noted. This includes, but is not limited to, all of the improvements from DMLL we presented. Examples of Delite's optimizations that proved useful for these applications are common subexpression elimination, code motion, horizontal and pipeline loop fusion, nested loop transformations, struct unwrapping, struct-of-array conversions, dead code elimination, and dead field elimination.

In this chapter we will compare Delite performance on a variety of hardware targets, including single-threaded performance, large NUMA machines, clusters, and GPUs. We will also look at a range of alternative implementations, all the way from hand-optimized C implementations to high-level implementations using parallel patterns.

Benchmark	Optimizations	Data Set	Delite	C++	Δ
TPC-H Query 1	GroupBy-Reduce, pipeline fusion, AoS to SoA, CSE, DFE	TPC-H SF5 (5.3GB)	1.07s	1.82s	-41%
Gene Barcoding	pipeline fusion, DFE	3.5M genes (689MB)	0.341s	0.311s	9.6%
GDA	pipeline fusion, horizontal fusion, CSE	500k x 100 matrix (835MB)	8.50s	6.92s	23%
k-means	Conditional Reduce, Row-to-Column Reduce, pipeline fusion	500k x 100 matrix (835MB)	0.885s per iter	0.843s per iter	5.0%
Logistic Regression	Column-to-Row Reduce, Row-to-Column Reduce	500k x 100 matrix (835MB)	0.082s per iter	0.075s per iter	9.3%
PageRank	domain-specific push-pull transformation, pipeline fusion	LiveJournal [1] (1.1GB)	0.646s per iter	0.518s per iter	25%
Triangle Counting	domain-specific push-pull transformation, pipeline fusion	LiveJournal (1.1GB)	12.3s	12.4s	-0.8%

Table 7.1: Benchmarks: Delite optimizations applied and sequential performance comparison to hand-optimized.

7.1 Baseline Sequential Performance

For all of the single-machine experiments (sequential, multicore, NUMA), we used a consistent setup. The machine contained 4 sockets, with 12 Xeon E5-4657L cores per socket, and 256GB of RAM per socket (1TB RAM total). We generated C++ code from Delite, which was then compiled with gcc 4.8, optimization level 3. Each experiment was performed five times and we report the average of all runs.

First we compare Delite's sequential performance to hand-optimized C++ implementations, shown in Table 7.1. For the iterative algorithms we present execution

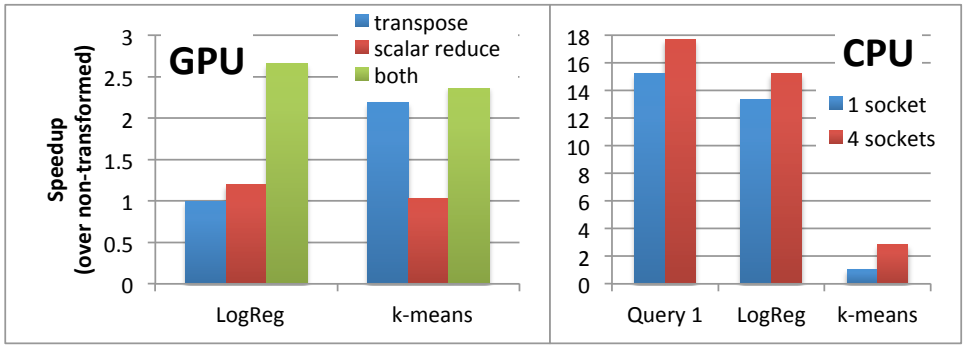

Figure 7.1: Speedups obtained by applying the DMLL nested pattern transformations for GPUs (left) and CPUs (right).

time per iteration and for the rest we present total execution time. Delite performance is within 25% of hand-optimized for every application studied. Delite attempts every optimization for each application, but we summarize the subset of optimizations that we determined were the most important in improving the performance for each application in the second column of Table 7.1.

While most applications are slightly slower than hand-optimized, Query 1 is actually faster than the C++ version. This is due to our implementation using a more efficient hash map implementation than the baseline implementation, which used the C++ standard library `unordered_map`. While this certainly means that the baseline implementation could be improved, we believe it is most representative of standard practices.

The small performance gaps in the remaining applications are mostly due to the fact that, like many functional languages, Delite applications typically allocate more memory than strictly necessary. Whereas the hand-optimized applications aggressively reuse allocated memory. Pipeline fusion reduces allocations substantially, but it cannot remove every allocation from the application.

7.2 Impact of Nested Pattern Transformations

Next we study the performance and scalability impact of the nested pattern transformations presented in Chapter 3. We use k-means clustering, logistic regression, and TPC-H Query 1 as case studies, shown in Figure 7.1. In shared memory environments

these transformations serve as performance optimizations, but for distributed memory Delite actually requires these transformations to properly partition data. Therefore we only show the impact on a shared-memory NUMA machine.

For k-means the impact is actually very small (only around 3%) when parallelizing across cores within a single socket. This is because both versions traverse the large dataset once (just in different orders) and each thread consumes sufficiently large, consecutive chunks of the dataset at a time. When parallelizing across multiple sockets however, the original shared-memory-centric implementation stops scaling due to both more limited available parallelism and an increasing memory bandwidth bottleneck as data must be shuffled amongst all the sockets randomly. In the transformed version, however, each thread reads memory from its local socket only, leading to nearly 3x speedup scaling from 1 to 4 sockets.

Logistic regression and TPC-H Query 1 both run significantly faster even in a single socket multicore configuration as the applied transformations greatly improve cache locality and optimize memory access patterns to be sequential. This class of optimization is always beneficial for CPUs whether the architecture is single-threaded or massively distributed.

Moving to the GPU, both k-means and logistic regression perform summations over vector types as implemented in the application. However Delite's CUDA code generator is only capable of using local shared memory for reduction temporaries when they have a statically known size (i.e., scalar types). The code generator instead allocates the vectors in global memories, but this leads to greatly reduced performance when reducing vector types compared to scalar types. There is also a second problem. The input matrix must be transposed from its optimal CPU layout to enable multiple threads' memory requests to be coalesced by the GPU's memory controller.

With DMLL, we can apply the *Row-to-Column Reduce* rule to transform vector reductions into scalar reductions. The GPU code generator then uses the same results of the read stencil analysis to transpose the nested loop's access pattern when generating the CUDA kernel implementation [47]. Delite performs both optimizations together for each application. By manually disabling them, we can see that for logistic regression both the *Row-to-Column* and transpose transformations must be

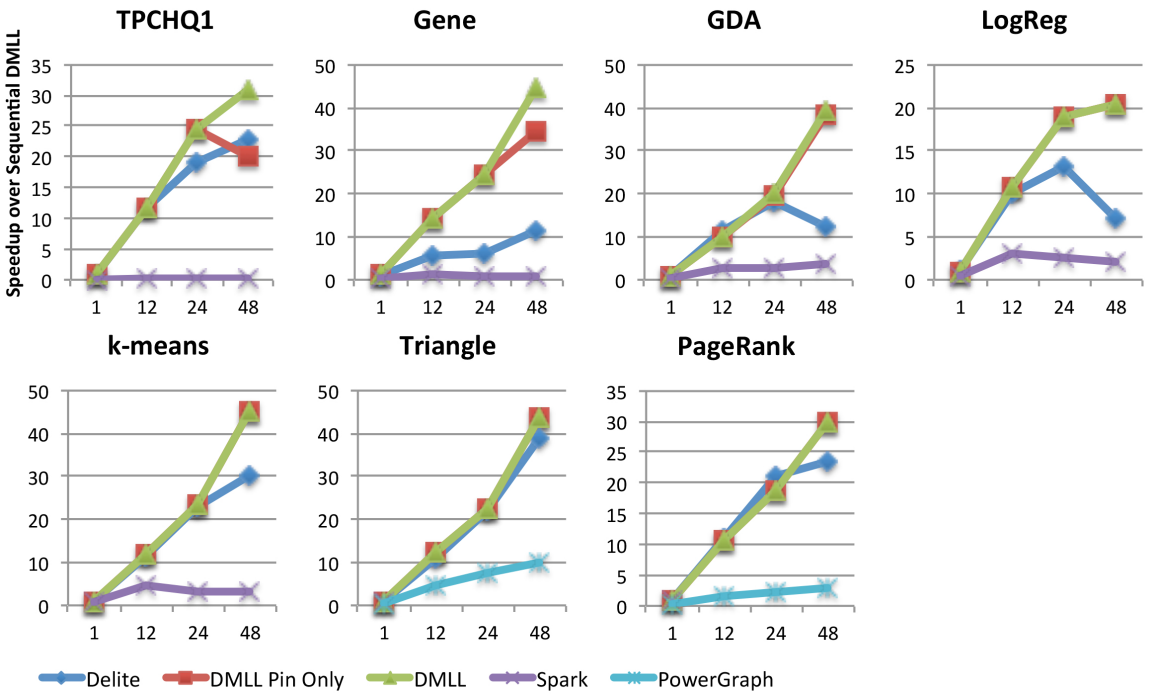

Figure 7.2: Performance and scalability comparison of DMLL, original Delite, Spark, and PowerGraph on a 4 socket machine.

combined for maximum performance. For k-means however transposing the matrix provides most of the performance improvement as it speeds up both the map and reduce phases of the operation, while the other only speeds up the reduction.

7.3 NUMA Scalability

In Figure 7.2 we show NUMA performance compared to popular alternative frameworks: Spark [87] for the machine learning and data querying apps and PowerGraph [29] for the graph analytics. We also compare to the original version of Delite [13] without DMLL improvements. We will refer to the latest, optimized implementation simply as DMLL here to make the comparison between old and new clearer.

For each application DMLL applies all of the compiler optimizations in Table 7.1

automatically. For the other systems (Spark and PowerGraph) we ported each application directly to their programming model and performed all possible optimizations manually. However not all of DMLL's optimizations were possible to express in these programming models. In particular, DMLL transparently performs AoS to SoA transformations, which enables storing the table in Query 1 as multiple arrays of primitives rather than a single array of objects. Such a transformation is not possible in Spark. Each field of the output record is produced by combining multiple fields of the input record. Therefore the input collection cannot simply be split into an RDD per field, they must be operated on as a single unit. Also, performing NUMA-aware memory allocations is not currently possible within the JVM, which severely limits the scalability of all JVM implementations on our test machine. For this reason, we use DMLL's C++ code generator rather than its Scala code generator, even though Spark uses Scala.

For DMLL we study the effects of incremental improvements to the original Delite's multicore runtime. We first simply augmented the runtime to pin each thread to a specific physical core in a locality-aware manner, and then added thread-local heaps instead of a global heap. These two items alone are sufficient to ensure that thread-local data remains local. However it still requires each logical data structure to be allocated in a single physical memory. We show the performance impact of these additions as *DMLL Pin-only* in Figure 7.2. We then added DMLL's NUMA-aware features which in addition to pinning use the analyses in Chapter 5 to partition the large arrays across multiple memory regions (DMLL), allowing multiple sockets to read independent segments of each array concurrently without contention. In general reading data from all memory banks simultaneously is necessary to maximize memory bandwidth. Therefore all data must either be thread-local (or at the very least socket-local) or partitioned across sockets.

The generated application code is essentially identical for both versions except for how each array is physically allocated. For the pin-only version each array is allocated entirely within a single socket's memory, with the socket determined by whichever thread calls `malloc` for that array. For the NUMA-aware version, partitioned arrays are instead partially allocated across every socket's memory by having multiple threads

participate in the allocation simultaneously.

As shown in Figure 7.2 the majority of benchmarks scale reasonably well up to two sockets but then stop scaling in the original Delite implementation, while the new DMLL versions continue to scale. For the first two apps the fully NUMA-aware version has the best scalability as most of the memory accesses are to the large partitioned data structures, therefore partitioning increases the available memory bandwidth, thereby increasing performance. For the next three apps most of the computation is instead within nested loops over thread-local data structures, so thread pinning with thread-local heaps is sufficient to maximize locality and memory bandwidth. The additional partitioning for the NUMA-aware version doesn't hurt though. For Triangle Counting even the baseline multicore version scales very well as the important working sets tend to fit in cache, thereby hiding all of the NUMA issues observed in the other applications. PageRank benefits from thread pinning, but requires significant inter-socket communication compared to the other apps, which limits the overall scalability and bandwidth benefit of partitioning, since data must be shuffled across sockets regardless.

Moving to the alternative implementations, we observe that all DMLL versions are significantly faster than the Spark and PowerGraph implementations. Our generated code is both faster when run sequentially, and scales better at high thread counts. The single-threaded performance improvements are largely due to the compiler optimizations discussed (shown in Table 7.1) and the much more efficient nature of Delite generated code compared to traditional library implementations. The improved scalability is obtained from DMLL's NUMA-aware optimizations and a very low overhead runtime implementation. Comparing performance at each thread count, we observe that Spark performs best relative to DMLL when each machine has relatively few compute resources. As machines get larger and more powerful however, DMLL shows much stronger scaling at higher thread counts within the machine.

7.4 Heterogeneous Clusters

For the next set of experiments, we move the experimental setup to different cluster configurations. We first use Amazon EC2 to study a traditional commodity cluster,

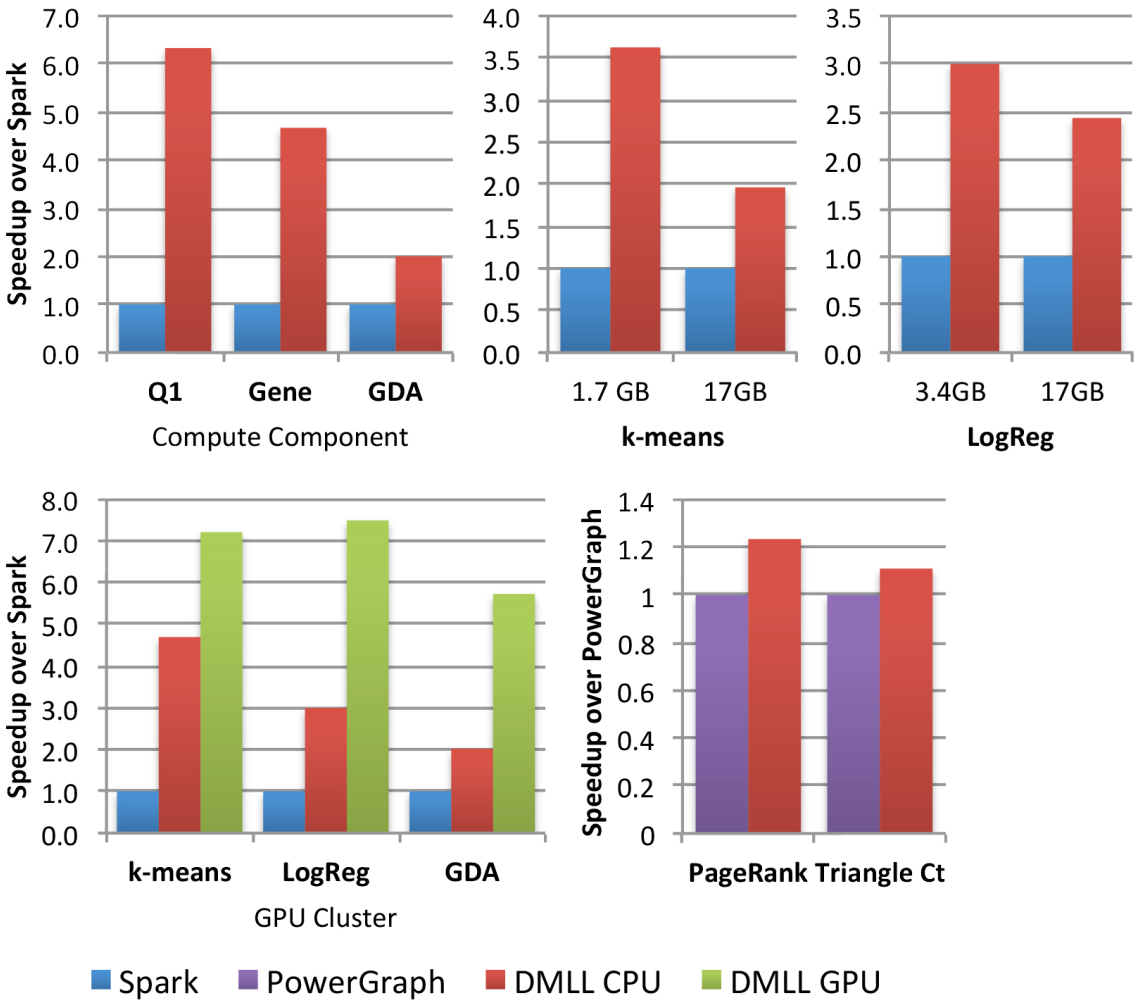

Figure 7.3: Performance of DMLL Delite compared to manually optimized implementations in alternative systems. The left three graphs were run on the 20 node Amazon cluster, and the rest on the 4 node GPU cluster connected within a single rack.

specifically 20 m1.xlarge instances. Each machine contained 4 virtual cores, 15GB of memory, and 1Gb Ethernet connections. For these experiments we generated Scala code and ran entirely on the JVM to provide the most fair comparison with Spark. We used Java 1.6.0b24 with default options. The results are shown in Figure 7.3. We do not show a comparison of Delite's original performance for this setup, as Delite could not run on multiple machines. For each application DMLL uses the same parallelization and distribution strategy as Spark, however in the Spark version this is enforced manually and in DMLL it is performed automatically by the partitioning analysis. Spark allows users to control whether collections are written to disk or kept in memory. In these experiments we kept all data entirely in memory, sacrificing fault tolerance for maximum performance. DMLL also keeps all data in memory.

The first two benchmarks (Query1 and Gene) only iterate over the primary dataset once, so in isolation they are fundamentally I/O bound, and both systems take roughly the same amount of time to read data from disk. However, we separate the input loading time from the computation time to demonstrate the potential speedup if the dataset being queried already resides in memory. GDA is similar but iterates over its dataset twice instead of once. k-means and logistic regression however both iterate over the primary dataset many times, and therefore the initial I/O cost is amortized over a large number of iterations. This iterative style of application is what Spark excels at in comparison with systems like Hadoop. Overall the performance difference between DMLL and Spark is much smaller on this configuration, comparable to the single-threaded performance difference between the two systems, as each machine has very few resources and both systems distribute the data and work across machines identically. Switching to a cluster of higher-end machines with more CPUs increases the gap, as seen in the next results on the GPU cluster.

For the GPU cluster and graph experiments we switched to a smaller high-end cluster of 4 nodes, each with 12 Intel Xeon X5680 cores, 48GB of RAM, an NVIDIA Tesla C2050 GPU, and 1Gb Ethernet within a single rack. GDA is well-suited to GPU execution and runs over 5x faster than Spark without any additional optimizations beyond what DMLL already performed for the CPU. k-means has been previously

shown to perform well on GPUs when manually optimized, but, as discussed in Section 7.2, generating efficient code automatically requires multiple transformations. For clusters of GPUs, we first apply any transformations required for distribution, creating a program capable of running on either CPUs or GPUs within each node. When generating the GPU kernel for each machine, we then apply any GPU-specific transformations for operating on the local chunk. Therefore for this application the input matrix is first distributed by rows across each machine, which works well for distributed CPUs, and then each GPU's kernel is transposed to read data column-wise instead for any machine that offloads its chunk to the local GPU.

Without these transformations the GPU code performs worse than DMLL's CPU code, but applying them provides 7.2x speedup over Spark. The same transformations are required for logistic regression. Logistic regression has much lower arithmetic intensity than k-means however, and so the improved execution time comes largely from the GPU's higher memory bandwidth rather than compute power. The other benchmarks we studied generally do not perform very well on GPUs due to the fact that the cost of moving the data to the GPU is often more expensive than just computing the result on the CPU. For iterative algorithms however this is not the case. Just as the cost of reading the data from disk is amortized over many iterations, so is the initial cost of moving the data to the GPU.

Finally we study the scalability of graph applications. These were also done on the GPU cluster since it has substantially better network performance and the graph applications fundamentally require significant inter-machine communication. We also move to compare against PowerGraph rather than Spark as it has been shown to be substantially faster than Spark for graph analytics applications [29]. Both systems implement the same high-level model of computation, namely pushing the required data to local nodes and then performing the computation locally. As such both systems transfer the data across the network in roughly the same time. The computation portion runs faster in DMLL due to the low-level nature of DMLL's generated code compared to the PowerGraph library implementation, however this is largely overshadowed by the communication, leading to comparable overall performance. Comparing the previous NUMA results, we see the usefulness of large memory NUMA

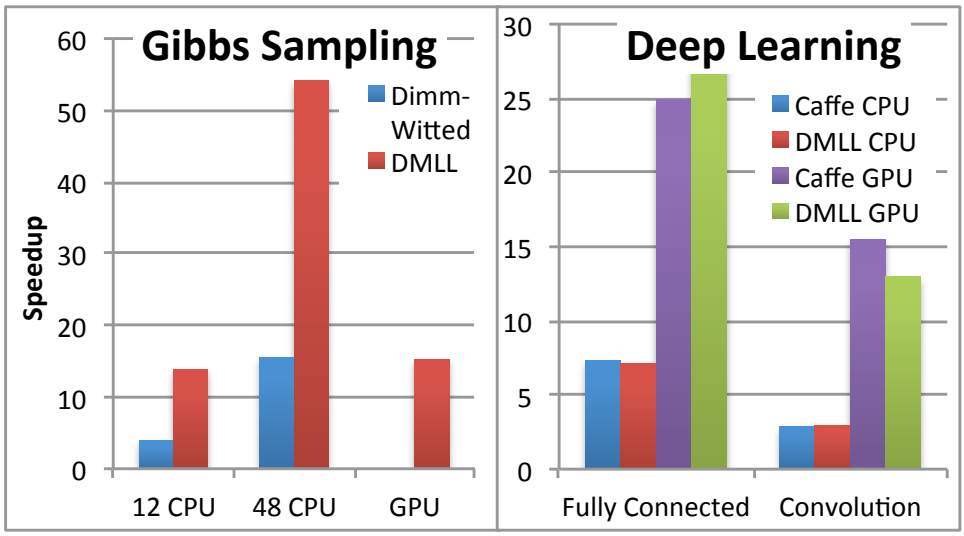

Figure 7.4: Performance of Gibbs Sampling and Deep Neural Network in DMLL compared to optimized C++ library implementations. Performance is normalized to sequential.

machines for graph analytics. For the cluster implementation of both systems, most of the execution time is spent transferring the graph over the network and building a local cache of the received remote data, which is slower than just using a single machine. In a NUMA machine however, accessing remote portions of the graph is still relatively fast, and therefore the efficiency of the generated code has a large impact on the overall performance and scalability.

7.5 Real-world Applications

In this section we study the impact of applying DMLL to real-world machine learning applications that are being actively researched and used in both academia and industry today. Our first case study is Gibbs Sampling, and the second is Deep Neural Networks (DNNs).

Gibbs Sampling Gibbs sampling on factor graphs is a popular method to solve Bayesian statistical problems, including price modeling and information extraction, that is used in several commercial data analytics engines. It is one of the core components of DeepDive [61], which uses a custom hand-optimized implementation

DimmWitted [88]. We compare DimmWitted's hand-optimized performance with Delite's automated performance across multiple sockets in our large NUMA machine from previous experiments. To put this comparison in context, note that DimmWitted is actually *100x faster* than previous Gibbs sampling implementations [88]! We examine how this is achieved next.

Gibbs sampling presents a rather unique challenge for parallel frameworks to implement as the optimal parallelization strategy is actually different for cores within a socket compared to across sockets. The DimmWitted algorithm exploits the fact that threads within a socket can communicate very inexpensively (through the last-level cache) by performing Hogwild! [60] updates. These updates are lock-free, allowing threads to read and write the shared output model completely asynchronously, and relying on hardware cache-coherence to eventually communicate data amongst threads.

This execution model quickly stops scaling beyond a single socket however due to the much higher costs of cache coherency across sockets. Therefore DimmWitted uses a hybrid approach and allocates an separate model for each socket. Within each socket all threads still share the model and perform Hogwild! updates. However now each each model is sampled independently. The program then averages the samples to produce the final output.

Expressing this algorithm using data-parallel constructs fundamentally requires the system to be able to exploit nested parallelism, as the algorithm explicitly partitions the program first over sockets, then over cores within a socket.

We can express this in OptiML by using a parallel `Reduce` with a nested parallel `Foreach`, where the models are allocated within the outer reduction. Since the algorithm explicitly allows data races, this is one of the few situations where Delite's `Foreach` pattern is actually required over `Map` or similar. For this application, DMLL automatically discovers that the outer loop is only of size `numSockets`, and proceeds to do the right thing. It first distributes the outer parallelism over each model across all sockets, and then since there are additional hardware resources remaining (all of the other cores within each socket), it parallelizes the inner loop that operates on each model across all the cores. This algorithm, while logically relatively simple, is

impossible to implement in most parallel programming systems (see Table 1.1).

As shown in Figure 7.3, using this strategy both DMLL and DimmWitted scale nearly linearly across sockets on the 4 socket machine. However the DMLL version is over 2x faster sequentially and 3x faster with multicore. The GPU implementation is limited by the random memory accesses into the factor graph, which greatly reduces the achievable memory bandwidth.

Our performance improvements in this case are due primarily the efficiency of our generated code that uses unwrapped arrays of primitives to implement the factor graph. The hand-written version, in comparison, contained more pointer indirections in the factor graph implementation for the sake of user-friendly abstractions. We have observed that these types of trade-offs happen quite often in real-world applications, even when the primary design focus is on performance rather than long-term maintainability. Ultimately, application developers want to be able to read their code after writing it, which is not a concern of Delite's code generators.

Deep Neural Network Deep neural networks (DNNs) are a class of artificial neural networks that attempt to model machine intelligence. They are widely used in industry for speech and image recognition applications. Recent advances in hardware parallelism along with the presence of a massive amount of data have stimulated DNN research. Developing and tuning DNNs has been performed empirically, by exploring different types of network layers, connections, and configurations to improve the accuracy of the result. In order to expedite this repeated process, researchers have developed DNN libraries and frameworks such Caffe [42]. Caffee provides a hand-optimized C++ and CUDA library for each layer and allows users to create simple applications on top by composing layers. We break down the application time into each layer and show comparable performance at each level in Figure 7.4.

The convolution layer is the most compute-intensive and time consuming layer in many DNN applications. The input, output, and filter of the layer are all 4 dimensional tensors, each dimension representing the batch size, the number of features, height, and width respectively. A convolution layer can be implemented in multiple ways. One common implementation is based on lowering, which reorganizes each image to a certain layout, followed by a matrix multiplication to perform the

convolution. This implementation has been widely used since there are optimized implementations of matrix multiplication for many hardware platforms. The lowering part of the computation can be implemented as a `Map`.

The fully-connected layer connects all input neurons to all output neurons. Each output neuron is a weighted sum of all input neurons. The input, weight, and output of the layer are all 2 dimensional tensors, and the computation can be done with a single matrix multiplication. Given an input matrix [batch size x input neurons] and a weight matrix [input neurons x output neurons], the layer produces an output matrix [batch size x output neurons].

We implemented these two algorithms in OptiML and obtained comparable performance to Caffee's hand-optimized CUDA implementations. The convolution layer is about 30% slower due to functional abstractions. While Caffe aggressively utilizes pre-allocated memory buffers to store the matrix results, our implementation allocates a fresh array for each time.

Chapter 8

Related Work

The work presented in this dissertation follows from a wide range of previous work in the fields of staged metaprogramming, domain-specific languages, parallel programming models, and pattern-based compiler optimizations. We survey the state-of-the-art in each of these areas below, and illustrate how the techniques in this dissertation have expanded upon the work before.

8.1 Staged Metaprogramming

The embedding of DSLs in a host language was first described by Hudak [38]. Tobin-Hochstadt et al. [80] extends racket, a scheme dialect, with constructs to enhance the embeddability of other languages. These ideas inspired our enhancements to the scala compiler to allow for a deeper embedding of Delite DSLs.

Many static metaprogramming techniques exist, including C++ templates [81] and Template Haskell [74]. Expression Templates [82] allow customizable generation, and TaskGraph [5] performs runtime code generation from C++. Designated multi-stage programming languages include MetaML [78] and MetaOCaml [14]. The Delite framework is built on top of the Lightweight Modular Staging approach [69], inspired from the related work on embedding typed languages by Carette et al. [15] and Hofer et al. [35]. Libraries using domain-specific code generation and optimization include ATLAS [86] (linear algebra), FFTW [28] (discrete Fourier transform), and SPIRAL [66] (general linear transformations). Such program generators often require

significant effort to create. The Delite framework and Lightweight Modular Staging aim to make such facilities more easily accessible.

8.2 High Performance DSLs

DSL design can be split into two categories. External DSLs, which are completely independent languages, and internal DSLs, which borrow some degree of functionality from a hosting language. We adopt a purely embedded approach for constructing DSLs, as presented by Hudak [38]. Leijen et al. [48] and Elliot et al. [27] pioneered embedded compilation and used a simple image synthesis DSL as an example. Delite draws from that research extending it and applying it in other domains.

Previous work has shown how domain knowledge can enhance application performance. Meng et al. show the benefits of best-effort computing for recognition and mining applications [56]. Menon et al. apply high level transformations to MATLAB code, producing performance gains in both interpreted and compiled code [57]. Guyver et al. present a way to annotate library methods with domain-specific knowledge and show significant performance improvements [33]. CodeBoost [4] allows for user-defined rules that are used to transform the program using domain knowledge. Delite, on the other hand, allows DSL developers to perform domain-specific compiler transformations on the application IR.

There has also been work on extensible compilation frameworks aimed towards making high performance languages easier to build. Feldspar [3] is an instance of an embedded DSL that combines shallow and deep embedding of domain operations to generate high performance code. Delite provides a framework that allows similar embeddings but can be re-used across many DSLs. Telescoping languages [44] automatically generate optimized domain-specific libraries. They share Delite's goal of incorporating domain-specific knowledge in compiler transformations. However Delite extends optimization to DSL data structures and is 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 [10] 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.

For stand-alone DSL compilers, there has been considerable progress in the development of parallel and heterogeneous DSLs. Liszt [25] and Green-Marl [36] 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 [21] is a parallel DSL for image analysis that demonstrates good performance compared to hand-written C code using an optimized library. Püschel et al. [65] show that domain-specific optimizations can yield substantial speedups when tuning code for particular digital signal processors (DSPs). Our work aggregates many of the lessons and techniques from these previous DSL efforts and makes them easier to apply to new domains.

8.3 High-level Parallel Programming

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. [51] 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. [63] 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 [2] 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.

Existing parallel programming models operate at various different levels of abstraction. OpenCL [79] provides a standard that allows a programmer to target multiple different hardware devices from a single environment rather than using a distinct vendor API for each device. Higher level data-parallel programming models often provide implicit parallelization by providing the programmer with a data-parallel API that

is transparently mapped to the underlying hardware. Recent work in this area includes Copperhead [16], which automatically generates and executes Cuda code on a GPU from a data-parallel subset of Python. Array Building Blocks [39] manages execution of data-parallel patterns across multiple processor cores and targets different hardware vector units from a single application source.

One of the most famous distributed programming models is MapReduce [24] for large clusters. More recent proposals have focused on addressing the inefficiencies of the MapReduce model (or often just the inefficiencies of the open source implementation Hadoop [7]). DryadLINQ [41] converts LINQ [55] programs to execute using Dryad [40], which provides coarse-grained data-parallel execution over clusters. FlumeJava [19] targets Google's MapReduce [24] from a Java library and fuses operations in the data-flow graph in order to generate an efficient pipeline of MapReduce operations. Spark [87] provides a richer set of distributed operators similar to Scala collections, provides automatic pipeline fusion, and stores intermediate results in memory rather than disk for higher performance.

Other systems such as Pregel [52] and PowerGraph [29] have extended this model particularly for graph analytics. Galois [59] performs efficient graph analytics in shared memory but does not scale out to distributed systems. SnuCL [45] is a compiler and runtime that executes OpenCL programs over heterogeneous clusters, but the programmer is responsible for making parallelization and partitioning decisions.

Dandelion [73] compiles LINQ programs to clusters and GPUs. Lime compiles Java programs to GPUs [26] and FPGAs [2] but provides a much more imperative programming model with only basic data-parallel operations. X10 [20] also supports only basic data parallelism over arrays but targets explicitly distributed memory. NESL [8] and Data Parallel Haskell [43] support nested parallelism, but unlike Delite they rely on flattening transformations which can reduce efficiency.

Our goal with Delite and DMLL is to unify the most useful programming model features into a single system for data analytics and add support for one of the most performant but previously overlooked hardware targets: big-memory NUMA machines. Supporting these features is made possible primarily through multiple compiler optimizations on parallel patterns.

8.4 Pattern-based Compiler Transformations

The Delite compiler infrastructure implements a set of advanced compiler optimizations in a reusable fashion. These are largely drawn from existing literature which includes the following: program transformations achieved using rewrite rules as discussed by Bravenboer et al. [11], composable and combinable analysis and optimizations [49, 83, 22] and techniques for eliminating intermediate results [85, 23] and loop fusing [31]. More recent work applies these ideas specifically to implementing extensible compilers for high-level programs [71].

The transformations presented in this dissertation are designed to improve locality and are therefore similar in spirit to multiple previous transformation systems. Systems designed to optimize imperative loops using polyhedral analysis such as Pluto [9], PPCG [84], and Polly [32] can perform automatic tiling, parallelization, and distribution of nested loops to target CPUs and/or GPUs. However, while these systems work well for loops with completely affine memory accesses, they fail on commonly occurring data-dependent operations such as `filter` and `groupBy` [6]. The formalism we present instead exploits the higher level semantics of parallel patterns over generic **`for`** loops to overcome this limitation. This is particularly critical for data analytics compared to HPC as such data-dependent operations are extremely common.

Spartan [37] also performs automatic tiling and data distribution, but via parallel patterns on mult-dimensional arrays. In contrast to our work, Spartan focuses solely on the data distribution, and not how to optimize the computation at multiple levels. Systems such as FlumeJava [19] and Data Parallel Haskell [43] also use a rewrite system to optimize parallel patterns, but they only focus on pipeline fusion and flattening transformations across functional operators and do not consider optimizing nested parallelism.

Chapter 9

Conclusions and Future Work

In this dissertation, we introduced the Delite Multiloop Language (DMLL), a new intermediate language based on common parallel patterns that captures the necessary semantic knowledge to efficiently target distributed heterogeneous architectures. The language semantics of the `Multiloop` naturally capture a set of powerful transformations over nested parallel patterns that restructure computation to enable distribution and optimize for heterogeneous devices. We were able to express these transformations using relatively simple rewrite rules and still capture a large space of programs with each rule. This allowed us to create a data-parallel programming model for heterogeneous distributed hardware that does not sacrifice significant programming model features, hardware targets, or sequential performance. This is in sharp contrast to other high-level parallel and distributed systems that often introduce substantial overhead.

We also presented a data structure model that was sufficiently restrictive to perform advanced layout optimizations, while still expressive enough to implement languages in multiple data analytics domains. We implemented multiple data layout transformations, such as array-of-struct (AoS) to struct-of-array transformations (SoA) using a set of very simple rewrite rules that naturally stack together to create far more complex transformations. Combining a wide range of optimizations over both the parallel patterns and parallel data structures, we are able to generate code

comparable to low-level optimized imperative implementations. These transformations not only improve performance in isolation, but are also key to simplifying the task of analyzing, partitioning, and transferring data structures.

We then presented a new data distribution analysis for Delite designed to automatically determine what data to distribute based on its access patterns and the locations of its input(s). After the previous data structure transformations, traditional affine access pattern analysis becomes relatively straightforward to perform. We combined the results of this analysis with the set of DMLL rewrite rules to automatically transform applications to improve locality and enable clean partitioning of distributed data structures.

Finally we explored how multiloops can be naturally hierarchically partitioned across heterogeneous distributed hardware at runtime by repeatedly sub-dividing ranges of loop indices based on the amount of resources available. We also explored how to greatly increase the amount of parallel work available at runtime by automatically exploiting any and all nested levels of parallelism within each multiloop. We then exploited these same restricted semantics and relatively simple data structure model to perform specialized garbage collection of temporary allocations at the end of each multiloop iteration.

While there are many individual and distinct optimizations in effect, they nevertheless all work together to generate abstraction and indirection-free, low-level imperative programs. This produces improved single-threaded performance, greater parallel scalability, and smaller memory footprints for a single machine. It also enables automated data structure distribution across memory regions with relatively straightforward additional analyses and runtime management.

We applied these techniques to multiple applications within the data analytics domains of machine learning, data querying, and graph analytics, and demonstrated significantly higher performance than alternative systems. This performance is due to the combination of both high single-threaded performance (comparable to hand-optimized sequential C++ implementations), as well as strong scalability across large NUMA machines and clusters.

Overall we have demonstrated and heavily emphasized the power of static optimization and code generation, in particular how many small optimizations at multiple levels of abstraction can all stack to produce dramatic code improvements. While we believe this technique to be highly effective and a viable solution to making distributed, heterogeneous programming available to a wide range of programmers, no compiler will ever be perfect. In the absence of an infinitely smart compiler, we are left with an important question: When the compiler misses an important optimization or makes a poor scheduling choice, how is the user supposed to fix it?

One of the benefits of embedded DSLs is that the application and compiler actually co-exist in the same program, making overriding the compiler's default behavior for a given application far more tractable than it is with a stand-alone compiler. However, a natural consequence of a very smart compiler is a very complicated compiler, which makes finding individuals motivated to fix problems themselves difficult. Some systems support user-directed optimization hints to guide the compiler's search. While this does require a more capable user to provide the correct hints, understanding the purpose of certain optimizations at a high level is still far simpler and faster than actually implementing those optimizations in the application directly. Overall we believe adding annotations, increasing visibility into common optimizations, and providing runtime profiling tools to be the correct direction to enable power users to heavily optimize performance and still exert far less effort and time than manual optimization would require.

There's also a very practical answer to this question. While the code quality (and performance) for an arbitrary future application using a system like Delite may not be as optimized as the applications presented in this dissertation, it may well be *good enough*. In practice programmers very rarely know how fast their application could possibly run, they simply know if it's running fast enough for their needs. Even though they are not perfect, high level tools like Delite provide programmers the ability to get most of the way there in terms of performance for a tiny fraction of the programming effort. This is quite often the correct trade-off.

Bibliography

[1] Livejournal social network. `http://snap.stanford.edu/data/soc-LiveJournal1.html`.

[2] Joshua Auerbach, David F. Bacon, Perry Cheng, and Rodric Rabbah. Lime: a Java-compatible and synthesizable language for heterogeneous architectures. OOPSLA. ACM, 2010.

[3] 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.

[4] O.S. Bagge, K.T. Kalleberg, M. Haveraaen, and E. Visser. Design of the CodeBoost transformation system for domain-specific optimisation of C++ programs. In *Source Code Analysis and Manipulation, 2003. Proceedings. Third IEEE International Workshop on*, pages 65–74, Sept. 2003.

[5] Olav Beckmann, Alastair Houghton, Michael Mellor, and Paul H.J. Kelly. Runtime 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.

[6] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. The polyhedral model is more widely applicable than you think. Springer Verlag, 2010.

[7] Andrzej Bialecki, Michael Cafarella, Doug Cutting, and Owen O'Malley. Hadoop: a framework for running applications on large clusters built of commodity hardware. *Wiki at http://lucene. apache. org/hadoop*, 11, 2005.

[8] Guy E. Blelloch. Programming parallel algorithms. *Commun. ACM*, 1996.

[9] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral program optimization system. 2008.

[10] 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.

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

[12] Kevin J Brown, HyoukJoong Lee, Tiark Rompf, Arvind K Sujeeth, Christopher De Sa, Christopher Aberger, and Kunle Olukotun. Have abstraction and eat performance, too: Optimized heterogeneous computing with parallel patterns. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, pages 194–205. ACM, 2016.

[13] 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. PACT, 2011.

[14] Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. Implementing multi-stage languages using asts, gensym, and reflection. GPCE, pages 57–76, 2003.

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

[16] 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.

[17] 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.

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

[19] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. Flumejava: easy, efficient data-parallel pipelines. PLDI. ACM, 2010.

[20] 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.*, 2005.

[21] 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.

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

[23] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream fusion: from lists to streams to nothing at all. In *Proceedings of the 12th ACM SIGPLAN*

international conference on Functional programming, ICFP '07, pages 315–326, New York, NY, USA, 2007. ACM.

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

[25] 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.

[26] Christophe Dubach, Perry Cheng, Rodric Rabbah, David F. Bacon, and Stephen J. Fink. Compiling a high-level language for GPUs: (via language support for architectures and compilers). PLDI '12, 2012.

[27] 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.

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

[29] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, 2012.

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

[31] 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.

[32] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. Polly: performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(04):1250010, 2012.

[33] Samuel Z. Guyer and Calvin Lin. An annotation language for optimizing software libraries. In *PLAN '99: Proceedings of the 2nd conference on Domain-specific languages*, pages 39–52, New York, NY, USA, 1999. ACM.

[34] Jared Hoberock and Nathan Bell. Thrust: C++ template library for CUDA, 2009.

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

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

[37] Chien-Chin Huang, Qi Chen, Zhaoguo Wang, Russell Power, Jorge Ortiz, Jinyang Li, and Zhen Xiao. Spartan: A distributed array framework with smart tiling. USENIX Association, 2015.

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

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

[40] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. EuroSys. ACM, 2007.

[41] Michael Isard and Yuan Yu. Distributed data-parallel computing using a high-level programming language. SIGMOD. ACM, 2009.

[42] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional

architecture for fast feature embedding. In *Proceedings of the ACM International Conference on Multimedia*, MM '14, pages 675–678, New York, NY, USA, 2014. ACM.

[43] Simon L. Peyton Jones, Roman Leshchinskiy, Gabriele Keller, and Manuel M. T. Chakravarty. Harnessing the multicores: Nested data parallelism in Haskell. In *FSTTCS*, pages 383–414, 2008.

[44] 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.

[45] Jungwon Kim, Sangmin Seo, Jun Lee, Jeongho Nah, Gangwon Jo, and Jaejin Lee. Snucl: an OpenCL framework for heterogeneous CPU/GPU clusters. In *Proceedings of the 26th ACM international conference on Supercomputing*, ICS '12, pages 341–352, New York, NY, USA, 2012. ACM.

[46] HyoukJoong Lee. *High-Level Language Compilers for Heterogeneous Accelerators*. PhD thesis, Stanford University, 2014.

[47] HyoukJoong Lee, Kevin J. Brown, Arvind K. Sujeeth, Tiark Rompf, and Kunle Olukotun. Locality-aware mapping of nested parallel patterns on gpus. IEEE Micro, 2014.

[48] 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.

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

[50] 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.

[51] 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.

[52] 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. SIGMOD '10. ACM, 2010.

[53] Berna L Massingill, Timothy G Mattson, and Beverly A Sanders. A pattern language for parallel application programs. In *Euro-Par 2000 Parallel Processing*, pages 678–681. Springer, 2000.

[54] Frank McSherry, Michael Isard, and Derek G Murray. Scalability! but at what cost?

[55] Erik Meijer, Brian Beckman, and Gavin Bierman. LINQ: Reconciling object, relations and XML in the .NET framework. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD. ACM, 2006.

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

[57] Vijay Menon and Keshav Pingali. A case for source-level transformations in MATLAB. In *PLAN '99: Proceedings of the 2nd conference on Domain-specific languages*, pages 53–65, New York, NY, USA, 1999. ACM.

[58] 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.

[59] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. SOSP '13, 2013.

[60] Feng Niu, Benjamin Recht, Christopher Ré, and Stephen J Wright. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. *Advances in Neural Information Processing Systems*, 24:693–701, 2011.

[61] Feng Niu, Ce Zhang, Christopher Ré, and Jude W Shavlik. Deepdive: Web-scale knowledge-base construction using statistical learning and inference. *VLDS*, 12:25–28, 2012.

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

[63] 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.

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

[65] 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.

[66] 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.

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

[68] 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, to appear).

[69] 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.

[70] 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.

[71] 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.

[72] 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.

[73] Christopher J Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. Dandelion: a compiler and runtime for heterogeneous systems. ACM, 2013.

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

[75] Arvind Sujeeth. *Productivity and Performance with Embedded Domain-Specific Languages*. PhD thesis, Stanford University, 2014.

[76] 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.

[77] 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. ECOOP, 2013.

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

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

[80] 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.

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

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

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

[84] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. Polyhedral parallel code generation for CUDA. *ACM Trans. Archit. Code Optim.*, 9(4), January 2013.

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

[86] 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.

[87] 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. NSDI, 2011.

[88] Ce Zhang and Christopher Ré. Dimmwitted: A study of main-memory statistical analytics. *Proceedings of the VLDB Endowment*, 2014.