SCALING HIGH PERFORMANCE DOMAIN-SPECIFIC LANGUAGE IMPLEMENTATION WITH DELITE

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

Hassan Chafi
January 2013
Abstract

This thesis covers how to easily implement performance oriented embedded domain-specific languages.

Exploiting heterogeneous parallel hardware currently requires mapping application code to multiple disparate programming models. Unfortunately, general-purpose programming models available today can yield high performance but are too low-level to be accessible to the average programmer. We propose leveraging domain-specific languages (DSLs) to map high-level application code to heterogeneous devices. To demonstrate the potential of this approach we present OptiML, a DSL for machine learning. OptiML programs are implicitly parallel and can achieve high performance on heterogeneous hardware with no modification required to the source code. For such a DSL-based approach to be tractable at large scales, better tools are required for DSL authors to simplify language creation and parallelization. To address this concern, we introduce Delite, a system designed specifically for DSLs that is both a framework for creating an implicitly parallel DSL as well as a dynamic runtime providing automated targeting to heterogeneous parallel hardware. We show that OptiML running on Delite achieves single-threaded, parallel, and GPU performance superior to explicitly parallelized MATLAB code in nearly all cases.

Computing systems are becoming increasingly parallel and heterogeneous, and therefore new applications must be capable of exploiting parallelism in order to continue achieving high performance. However, targeting these emerging devices often requires using multiple disparate programming models and making decisions that can limit forward scalability. In previous work we proposed the use of domain-specific languages (DSLs) to provide high-level abstractions that enable transformations to high
performance parallel code without degrading programmer productivity. In this paper we present a new end-to-end system for building, compiling, and executing DSL applications on parallel heterogeneous hardware, the Delite Compiler Framework and Runtime. The framework lifts embedded DSL applications to an intermediate representation (IR), performs generic, parallel, and domain-specific optimizations, and generates an execution graph that targets multiple heterogeneous hardware devices. Finally we present results comparing the performance of several machine learning applications written in OptiML, a DSL for machine learning that utilizes Delite, to C++ and MATLAB implementations. We find that the implicitly parallel OptiML applications achieve single-threaded performance comparable to C++ and outperform explicitly parallel MATLAB in nearly all cases.
Acknowledgements

I would like to thank lots of people that were major contributors to this effort.
Contents

Abstract iv

Acknowledgements vi

1 Introduction 2

1.1 The Era of Heterogenous Processing 2
1.2 The Programmability Chasm 7
1.3 Trade-Offs in Programming Language Design 8
1.4 Domain-Specific Languages (DSLs) 10
1.5 Delite 12
1.6 Contributions 14
1.7 Outline 15

2 DSLs for Heterogeneous Parallelism 17

2.1 Example DSLs 18
2.1.1 OptiML 19
2.1.2 OptiQL 23
2.2 Benefits of Using High-Performance DSLs 24
2.2.1 Productivity 25
2.2.2 Performance 26
2.2.3 Portability Forward Scalability 27
2.3 The Need for a DSL infrastructure 28
2.4 DSL Implementation Strategies 29
2.5 Requirements for High-Performance DSL Infrastructure 30
List of Tables

3.1 Applications used for experimental analysis. ..................... 54
# List of Figures

1.1 Trade-Offs in Processor Design ........................................... 3
1.2 The need for specialization in processing resources ............... 5
1.3 Anton, a processor specialized to molecular dynamic simulations. 6
1.4 The Programmability Chasm .................................................. 7
1.5 Trade-Offs in Programming Languages ................................. 9
1.6 Bridging the programmability chasm with DSLs ...................... 11
1.7 Overview of the Delite DSL Infrastructure .......................... 13

3.1 Delite application execution overview .................................... 42
3.2 Class diagram of the OptiML Vector class. ............................ 46
3.3 Execution time of the various versions of our applications normalized to OptiML running in sequential mode. Speedup numbers are reported on top of each bar. ........................................... 53
3.4 Scalability of our selected OptiML applications running in a highly threaded system. Speedup is measured relative to OptiML running in sequential mode. ........................................... 56
3.5 The impact of domain-specific optimizations on k-means and SVM at 8 threads. Execution time is normalized to the 8 thread version without optimization shown in Figure 3.3. Speedup numbers are reported on top of each bar. ........................................... 58
4.1 Typical compilation phases and the language virtualization approach to embedded DSLs. This allows the DSL to adopt the front-end from a highly expressive embedding language, while customizing its intermediate representation and participating in the backend phases of its compilation.

5.1 Views of the DSL IR. DSL applications produce an IR upon execution. This IR is defined by the LMS framework with enough information to perform generic analyses and optimizations. The Delite Compiler Framework extends the IR to add parallelism information, and this view allows parallel optimizations and parallel code generation. The DSL extends the parallel IR to form a domain-specific IR, which allows for domain-specific optimizations.

5.2 Entire process of the Delite Compiler Framework and Runtime.

5.3 An overview of the Delite Runtime. The runtime uses the machine-agnostic execution graph representing the application as well as a machine description to schedule and execute the application on the available hardware. Walk-time code generation utilizes scheduling information to optimize kernels and synchronization, minimizing run-time overheads. Run-time systems execute the schedule, manage memory, and perform data transfers.

5.4 Multi-view of IR nodes in the Delite Compilation Framework. For example, the matrix addition operation \((M_1 = M_2 + M_3)\) is represented as a MatrixPlus IR node, which extends the DeliteOpZipWith IR node which again extends the Definition IR node. The generic IR view is used for traditional compiler optimizations, the parallel IR view is used for exposing parallel patterns and loop fusing optimizations, and the Domain-specific IR view is used for domain-specific optimizations.
Listings

2.1 MATLAB version of Gaussian Discriminant Analysis (GDA) . . . . . . 21
2.2 OptiML code for Gaussian Discriminant Analysis (GDA) . . . . . . 22
2.3 SQL implementation of a sample query similar to TPCH Q1. . . . . 23
2.4 LINQ implementation of a sample query similar to TPCH Q1. . . . 24
2.5 OptiQL implementation of a sample query similar to TPCH Q1. . . 24
2.6 Filtering and sorting in OptiQL . . . . . . . . . . . . . . . . . . . . . 26
3.1 An example sequential op in the OptiML DSL . . . . . . . . . . . 47
Chapter 1

Introduction

Until the early 2000s, increasing single threaded performance, largely delivered by increasing CPU clock rate frequency, enabled the majority of software developers to use straightforward and familiar sequential programming techniques to deliver more compute-intensive applications. This so called “free-lunch” era put the burden of managing the increased computational requirements squarely on the shoulder of the processor vendors [66]. Power constraints have limited the ability of microprocessor vendors to scale single-core performance with each new generation. As processor pipelines increased in complexity, their power consumption and related cooling requirements forced these vendors to consider more power efficient processor designs.

1.1 The Era of Heterogenous Processing

When implementing the hardware and, as we will discuss a bit further down, the software used to build computer systems, designers (e.g. computer architects for hw, programming language designers for sw) are forced to make trade-offs between a set of design goals. In the case of hardware, these design goals are represented by three Ps:

Performance by which we mean the peak performance that can be achieved on the system using some measure of operations over time (i.e. FLOPS).
Figure 1.1: Commercial processors in general cannot provide all three trade-offs (e.g. performance, generality and energy efficiency)
CHAPTER 1. INTRODUCTION

Power by which we mean the number of Watts consumed while executing the operations comprised in an application (e.g. FLOPS/Watt). This P is also usually referred to as the energy efficiency (i.e. Productivity) of the system.

Programability by which we mean the general applicability of the processor (e.g. Generality) to the various application areas of interest.

Computer system design is driven by the quest to achieve high performance, low power and high programmability. However, improving performance, power and programmability at the same time is often difficult because improving one characteristic of a computer system often makes another characteristic worse. For example, increasing performance often increases power and decreasing power with specialized hardware typically decreases programmability. Figure 1.1 illustrates how various commercial hw architectures usually optimize for two out of three of the above design goals. Traditional high-performance processor such as the Xeon branded x86 processors are highly programmable (i.e. general-purpose) but tend to be less energy efficient than GPUs which are high-performance, energy efficient but are not applicable (i.e. programmable or general) to all problem domains.

There is now a mounting evidence that the only way to achieve both high-performance and energy efficiency is to specialize the hardware to the problem domain. Hameed et al.[27] how how only way to achieve two to three orders of magnitude increase in energy savings and performance is through specialized application specific instructions. Figure 1.2 summarizes the findings in the paper. The Figure plots improvements both in performance and energy savings while performing an h.264 encode as we specialize the resources available on the processor. We can see that investments in increasing instruction level parallelism would only yield at most a 2x improvement in both performance and energy savings. Even an abundance of SIMD resources (i.e. GPUs) is too general of an approach and carries too much overhead to achieve the high orders of magnitude gains achieved when specialized instructions are added.

A recent example of such specialized hardware is Anton [63] (shown in Figure 1.3), built specifically for molecular dynamic simulations. Anton is currently being built
Figure 1.2: Summary of findings in a recently published paper [27] which shows that future performance gains will mainly come from heterogeneous hardware with different specialized resources.
with 512 highly specialized processors. These are clocked at just 400MHz, and the machine has modest memory, but its architecture lets it process problems in a massively-parallel way. Ultimately, this architecture will offer a performance boost of 1000x over current complex molecular simulations. These types of performance gains are game changers in terms of the new type of scientific explorations they make possible.

Most researchers agree at this point that future performance gains will mainly come from heterogeneous hardware with different specialized resources. An Example of this is the inclusion in mobile phones and tablets of specialized processors for handling the audio and video decoding duties, these tasks could have been handled by the general purpose processors available in these devices, however while more than capable of handling the task, they would do so at much higher energy cost thus depleting the device’s energy charge much more rapidly. Even for high-performance computing systems, this need for more power efficiency lead to the wide-spread adoption of chip multiprocessors which consist of simpler cores [32, 51] and these system now routinely include increasingly heterogeneous processing elements. For example,
GPUs have become essential components of modern systems due to their massively data-parallel computing capability [47, 3].

1.2 The Programmability Chasm

These heterogeneous architectures continue to provide increases in achievable performance, but unfortunately programming these devices to reach maximum performance levels is not straightforward. Each heterogeneous element has its own performance characteristics and pitfalls, and usually comes with its own programming model. This means that existing applications can no longer take advantage of the additional compute power available in these new and emerging systems without a significant parallel programming effort. Writing parallel programs, however, is not straightforward because in contrast to the familiar and standard von Neumann model for sequential programming, a variety of incompatible parallel programming models are available, each with their own set of trade-offs. Emerging heterogeneous systems further complicate this challenge as each accelerator vendor usually provides a distinct driver API.
and programming model to interface with the device.

Common examples of programming models are Pthreads or OpenMP for multi-core CPU, OpenCL or CUDA for GPU, and MPI for clusters [69, 48]. Therefore when targeting such architectures, the programmer must have a deep understanding of all the different hardware components and programming models, as well as understand how to use them together. Even with this understanding, the relative performance improvement is not easy to predict until the program is written and executed in different models with elaborate optimization phases that may also depend on the input data. Even worse, the optimized code for one system is neither portable nor guarantees high performance on another system. This difficulty of programming heterogeneous parallel architectures results in a severe loss in programmer productivity. In addition to the significantly increased effort needed to achieve correctness and performance during initial development, exposing all the low level details of each compute device is detrimental to the maintainability, future scalability, and portability of the application.

It is not realistic to expect the average programmer to deal with all this complexity. Moreover, exposing the programmer directly to the various models supported by each compute device will ultimately be detrimental to application portability, forward scalability and maintenance. As new system configurations emerge, applications will constantly need to be rewritten to take advantage of any new capabilities. This leads to a programmability chasm as illustrated in Figure 1.4. It is essential to develop appropriate abstractions that bridge this chasm so that programmers can write high-level code and not worry about low-level details that negatively impact productivity. Thus, there is a need for parallel heterogeneous programming models that target average programmers who are not interested in becoming parallel/heterogeneous programming experts.

1.3 Trade-Offs in Programming Language Design

Ideally a programming language for heterogeneous parallel computing systems should satisfy the following design goals:
Figure 1.5: Programming language designers also make similar trade-offs to those that hardware designers are forced to make.
Performance by which we mean the ability to achieve high scalability and fully take advantage of all applicable and available computation resources in the system.

Productivity by which we mean the ease of development of end-user applications.

Generality by which we mean the applicability of the programming language to the majority of problems that come up in application development.

Unfortunately, no language that satisfies all these design goals currently exists. Note how these set of trade-offs are quite similar to those discussed in section 1.1. And if one were to look at available successful programming languages, these tend to trades-off one desirable aspect to achieve the others (Figure 1.5). For example, C and C++ are general purpose, high-performance languages but arguably are less productive than python or Ruby. The latter languages are arguably less high-performance than C or C++. An interesting class of languages would be programming languages that eschew generality for productivity and performance. Because they forego generality these languages are targeted to a specific application domain and are called domain-specific languages (DSLs) [71].

1.4 Domain-Specific Languages (DSLs)

A domain-specific language (DSL) is a concise programming language with a syntax that is designed to naturally express the semantics of a narrow problem domain [71]. DSLs, sometimes called “little languages” [5] or “telescoping languages” [38], have been in use for quite some time. In fact, it is likely that most application developers have already used at least one DSL. Examples of commonly used DSLs are TeX and LaTeX for typesetting academic papers, Matlab for testing numerical linear algebra algorithms, and SQL for querying relational databases.

One can also view OpenGL as a DSL. By exposing an interface for specifying polygons and the rules to shade them, OpenGL created a high-level programming model for real-time graphics decoupled from the hardware or software used to render it, allowing for aggressive performance gains as graphics hardware evolves. Even
the Unix shell can be considered a DSL that provides a command-line interface to underlying operating system functions such as file and process management. The use of DSLs can provide significant gains in the productivity and creativity of application developers, the portability of applications, and application performance.

DSLs eschew the compromising middle ground position of general purpose high-level languages. For example, a program requiring graph analysis would express graph traversals using a breadth-first search (or depth-first search) language construct without spelling out the implementation of such traversals [?]. A corresponding domain-specific compiler can reason about the program at the level of domain operations, enabling coarse grain optimizations (e.g. eliminating whole linear algebra operations as opposed to individual arithmetic instructions). Furthermore, these abstractions leave implementation details unspecified, providing the compiler with the freedom to translate the application to a number of different low-level programming models.

Chapter 2 provides a more complete explanation of how DSLs can be used to productively develop applications targeting heterogeneous parallel systems, however, figure 1.6 gives an overview of DSLs bridge the programmability chasm identified in
section 1.2. The figure shows how a set of applications can be developed using a set of DSLs. For each DSL there is a corresponding DSL compiler responsible for mapping the high level DSL constructs into the different underlying hardware targets. Usually the result of this mapping is to generate lower level source code that uses the required API and drivers which are required to utilize the different heterogenous processing elements available in the system.

One challenge for the DSL approach is the cost of developing and maintaining multiple languages and their associated compilers and tools. Without reducing this cost, it may not be economical to adopt such an approach. The main contribution of this thesis is to show that with the aid of a reusable infrastructure, it becomes possible to cheaply build such DSL compilers. We have implemented a few prototypes of such an infrastructure. A critical component of this infrastructure is Delite\(^1\), the main research artifact discussed in this thesis.

### 1.5 Delite

Figure 1.7 provides an overview of the role of Delite, a critical part of our infrastructure for building high performance DSLs. Between the DSLs themselves and the hardware lies the DSL infrastructure. This infrastructure consists of multiple layers. The first layer of the infrastructure is a way of embedding a DSL within the general-purpose hosting language Scala [49] that allows the DSL to participate in the back-end phases of compilation [12]; this approach is called *Lightweight Modular Staging* [58]. While providing the required background on the previous two layers, this thesis will focus on the next two layers, which are collectively called Delite. The Delite Compiler Framework is capable of expressing parallelism both within and among DSL operations, as well as performing useful parallel analyses. It also provides a framework for adding domain-specific optimizations. It then generates a machine-agnostic intermediate representation of the program which is consumed by the Delite Runtime. The runtime system provides a common set of features required by most DSLs, such as scheduling work across hardware resources and managing communication.

\(^1\)Domain Extracted, Locality Informed, Task Execution
The Delite compiler framework provides a set of predefined IR nodes for the common parts of domain-specific languages. Essentially, this is equivalent to embedding the Scala language itself. This allows the DSL developer to start with the base Scala IR and then add domain-specific nodes and semantics to the language. Since the embeddings are modularized into a variety of traits, the DSL developer is free to choose the subset of functionality to include in his virtual language.

Delite also provides a set of special abstract IR nodes that can be inherited from when defining the IR of the DSL. In addition to a IR node that represents a sequential task, Delite provides IR nodes for a variety of parallel execution patterns (e.g. Map, Reduce, ZipWith, Scan). Each of these IR nodes constrains what information the DSL developer needs to provide so that Delite can automatically generate parallel code. For example, in a linear algebra DSL, the scalar*matrix multiplication IR node inherits from Map. The DSL developer would then only have to specify the mapping function; Delite handles generating the parallel code for that operation on a variety of targets such as the CPU and GPU.

A Delite program goes through multiple compilation stages before execution. The
first compilation stage, written by the DSL developer, uses Lightweight Modular Staging (LMS), explained in chapter 4, to lift the user program into an internal representation and performs applicable domain-specific optimizations. However, instead of generating explicitly parallel code, the DSL generates an IR where DSL operations are represented as Delite IR nodes (e.g. a Map node).

The IR is then compiled in subsequent stages by Delite. Delite expands the nodes generated by the DSL developer to handle Delite-specific implementation details (e.g. data chunking) and perform generic optimizations (e.g. operation fusing). Delite also generates a static schedule for straight-line subgraphs in the user program, which reduces the time required to make dynamic scheduling decisions during actual execution. In the last compilation stage, Delite maps each domain-specific operation to different hardware targets. The final result of compilation is an optimized execution graph along with the generated kernels. The Delite runtime executes the graph in parallel using the available hardware resources.

This is the second Delite prototype we have built. An initial Delite prototype which adopted a library-based approach to building high-performance DSLs is also described in this thesis. There were significant limitations to the library-based approach which we discuss in chapter 3

1.6 Contributions

This thesis documents the following contributions:

- Domain-specific languages are an effective way for productively programming heterogeneous parallel. Specifically the inclusion of high-level constructs and restrictions allow for the implicit extraction of parallelism. In addition, the domain-specific knowledge encoded that the DSL compiler can reason about allows for further optimizations.

- The Delite framework which simplifies the development of high-performance DSLs. We show how the Delite framework helps the DSL authors can leverage Delite Ops to implement their domain-specific parallel operations.
• Optimizations can be applied on three different views of the program’s IR which leads to re-usable optimizations across the different DSLs that can implemented using Delite.

• The Delite framework is re-usable, we walk the reader through a couple DSL case studies and refer the reader to more recent publications for further case studies.

• The resulting DSL compilers built using Delite achieve high-performance as we show in the case studies.

1.7 Outline

The rest of the thesis is organized as follows:

• Chapter 7 [OUT OF ORDER WHILE DRAFTING THE REST OF THE THESIS] gives some background and discusses related work that is useful to understanding this thesis.

• Chapter 2 introduces a couple of DSLs that we will be using as case studies throughout the thesis. The chapter also explains why using DSLs eases the burden of heterogeneous parallel programming. The chapter also discusses various DSL implementation techniques.

• Chapter 3 describes the first Delite prototype (DeliteLib) which adopted a library-based approach for parallelizing embedded domain-specific languages. The chapter concludes with a discussion on the shortcomings of such an approach.

• Chapter 4 details the necessary building blocks to achieve high performance using embedded domain-specific languages. The chapter introduces the notion of language virtualization and explains how embedded DSLs can participate in compilation.
• Chapter 5 presents the second prototype of Delite (DeliteComp) which builds on the foundations laid out by the previous chapter and provides an infrastructure for rapidly implementing high-performance DSLs.

• Chapter ?? provides performance results of our case study DSLs and show that they perform efficiently.

• Chapter 6 concludes and provides a look ahead into what we think are important areas for future research in the field.

The next chapter discusses some of the different programming languages and models for heterogeneous parallel system that have been proposed. It also provides the reader with references to related work in domain-specific languages and optimizations.
A successful parallel programming model should be driven by the following goals:

- **Productivity**: the application developer can, ideally, write programs without having to use any explicit parallel or heterogeneous constructs.

- **Performance**: the application should achieve good performance without sacrificing productivity. The system metric should be performance per man-hour.

- **Portability and Forward Scalability**: the application should leverage the varying amount of compute resources across different systems, both existing and emerging. The forward scalability goal manifests itself across two dimensions: the number of a particular compute resource and the diversity of compute resource types.

To provide productivity, a parallel programming model must raise the level of abstraction above that of current low-level programming models such as Pthreads and CUDA. Ideally, such a programming model would also be general, allowing arbitrary semantics to be expressed. However, despite decades of research, no such programming model currently exists. Instead, it seems that generality, productivity, and performance are usually at odds with one another, and must be carefully traded
off in successful programming languages (recall Figure 1.5). For example, C++ is
general and high performance, but is usually not considered as productive as higher
level languages like Python and Ruby. On the other hand, Python and Ruby have a
difficult time competing with C++ in terms of performance.

Figure 1.5 shows that it should also be possible to focus on performance and pro-
ductivity while trading off generality. This can be done by targetting the program-
ing language used to a particular domain. Such specialized languages are referred
to as Domain-Specific Languages (DSLs) [71, 30]. DSLs are not in themselves novel,
a typical definition of a DSL is a language or library with restrictive expressiveness
-targeted at a particular kind of problem. There are many DSLs in popular use today.
Make, for example, uses a DSL to simplify managing application build process. This
thesis was written in LaTex, a DSL for typesetting documents. RAILS is a ruby-based
DSL for creating web-applications. One could imagine using C++ for accomplishing
the previously mentioned tasks but it would significantly more difficult and time con-
ssuming to do so. These example DSLs have focused primarily on productivity. We
are interested in using DSLs for high-performance.

Domain-specific languages can deliver performance and productivity by providing
carefully designed APIs that are easy for developers in the domain to use. DSL
code often more closely resembles pseudo-code than C code, deferring most if not
all of the implementation details to the language. This deferral of responsibility is
extremely important. It allows the DSL developer to use the most efficient parallel
implementation and target different devices transparently to the application. This is
feasible only because the DSL does not try to do everything; instead, it tries to do a
few specific things very well.

2.1 Example DSLs

Before expanding on the benefits of using DSLs for high-performance, we think it
would be useful to introduce a few DSLs that members of the Pervasive Parallelism
Laboratory (PPL) have implemented. We will use some of these DSLs throughout
the rest of this thesis as examples for the concepts presented in order to make the
rest of the discussion more concrete to the reader.

2.1.1 OptiML

OptiML \cite{optimal} is a DSL for machine learning, a popular and growing domain. Machine learning (ML) is generally concerned with learning patterns from data and applying the learned models to tasks such as regression, classification, clustering, and estimation. ML is a particularly good domain for studying parallelism: it is at the core of several emerging applications (e.g. collaborative filtering, object recognition), it contains applications and datasets that are time-bound in practice, and it has both regular and irregular parallelism at varying granularity.

Most machine learning kernels use linear algebra operations and prominently feature vectors and matrices. MATLAB is one of the most commonly used languages for ML applications and incorporates some of the same features that make a DSL appealing. MATLAB increases productivity by providing an easy syntax for manipulating and using vectors and matrices, and improves performance by providing fast, optimized implementations of linear algebra operations and common algorithms. While it is effective as a language for linear algebra, it is intended to be general and applicable across many application domains. By using a DSL specifically targeted at machine learning, it is possible to capture higher level information that can be used to increase productivity, improve performance, and expose more parallelism.

To accomplish these goals, OptiML exploits the following key aspects of machine learning:

- Many algorithms are iterative and/or solve constrained optimization problems. These algorithms have kernels with a fixed structure that are executed repeatedly many times.

- Most applications operate on large datasets, and many of these datasets contain significant redundancy (e.g. network traffic traces). This implies that some data points in a training set may not always be essential.

- Many algorithms are probabilistic and can acceptably trade-off accuracy for
performance.

- Kernels contain large amounts of data parallelism at varying granularity.

- Most algorithms are limited by low arithmetic intensity and therefore memory bandwidth. Most accesses are either streaming (disjoint) or reductions.

To give you a flavor of OptiML, listing 2.1 shows a the implementation of a simple ML application, Gaussian Discriminant Analysis (GDA), written in OptiML and MATLAB while listing 2.2 shows the same application implemented in OptiML. Since most machine learning users are already familiar with MATLAB, OptiML uses MATLAB-like syntax and compares favorably in terms of conciseness and productivity.
% x: Matrix
% y: Vector

m = length(y);
n = size(x, 2);
y_ones = 0;
y_zeros = 0;
mu0_num = zeros(1,n);
mu1_num = zeros(1,n);

for i=1:m
    if (y(i) == 0)
        y_zeros = y_zeros + 1;
        mu0_num = mu0_num + x(i,:);
    else
        y_ones = y_ones + 1;
        mu1_num = mu1_num + x(i,:);
    end
end

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

sigma = zeros(n, n);
for i=1:m
    if (y(i) == 0)
        sigma = sigma + (x(i,:)-mu0)’*(x(i,:)-mu0);
    else
        sigma = sigma + (x(i,:)-mu1)’*(x(i,:)-mu1);
    end
end

Listing 2.1: MATLAB version of Gaussian Discriminant Analysis (GDA).
Listing 2.2: OptiML code for Gaussian Discriminant Analysis (GDA)

OptiML was the first DSL developed using Delite and was the driving use case for the initial design and implementation of both the library-based approach (DeliteLib) and the compiler-based approach (DeliteComp) of the Delite infrastructure. In fact, OptiML is the only DSL that was ever implemented using the first version of Delite (library-based). All other DSLs that we will present were implemented using DeliteComp.
2.1.2 OptiQL

OptiQL is a DSL for querying data. OptiQL is inspired and is quite similar to LINQ. In its current form, OptiQL can query in-memory collections (similarly to LINQ to Objects), but can be extended in the future to query databases as well. LINQ, and thus OptiQL, itself resembles another language that many people are familiar with, namely SQL. The key idea of all these languages is to let users specify the queries they would like to make on their data in a highly declarative language and then let an underlying system (e.g. a relational database management system) compile the query into an execution plan. This execution plan is optimized using domain-specific knowledge, in this case relational algebra.

To give you a flavor of OptiQL, we will show three listings which implement a query similar to the first query in the TPCH benchmark. Listing 2.3 shows the SQL implementation of this query, while listings 2.4 and 2.5 respectively show the LINQ and OptiQL implementations.

```sql
select
    l_linestatus,
    sum(l_quantity) as sum_qty
    sum(l_extendedprice*(1.0-l_discount)) as sum_discprice
    avg(l_extendedprice) as avg_price
    count(*) as count_order
from
    lineitems
where
    l_shipdate <= date'1998-12-01'
group by
    l_linestatus
order by
    l_linestatus;
```

Listing 2.3: SQL implementation of a sample query similar to TPCH Q1.
var q = from l in lineItems where(l.l_shipdate <= DateTime.Parse('1998-12-01')) select l group l by new {l.l_linestatus} into g select new {
g.Key.l_linestatus,
  sum_qty = g.Sum(l => l.l_quantity),
  sum_discprice =
    g.Sum(l => l.l_extendedprice*(1.0-l.l_discount)),
  avg_price = g.Avg(l => l.l_extendedprice),
  count_order = g.Count()
} OrderBy(l => l.l_linestatus)

Listing 2.4: LINQ implementation of a sample query similar to TPCH Q1.

val q = lineitems Where (_.l_shipdate <= Date('1998-12-01')).
groupBy (_.l_linestatus) Select { case (key,g) => new {
  l_linestatus = key
  sum_qty = g.Sum(_.l_quantity)
  sum_discprice =
    g.Sum(l => l.l_extendedprice*(1.0-l.l_discount))
  avg_price = g.Avg(_.l_extendedprice)
  count_order = g.size
}} OrderBy(_.l_linestatus)

Listing 2.5: OptiQL implementation of a sample query similar to TPCH Q1.

The interested reader can also refer to Green-Marl [?], a DSL for graph analysis, and Liszt [28], a DSL for solving mesh-based partial differential equations, for further examples of high-performance DSLs developed at PPL. Now that we have shown the reader a few example DSLs, we are ready to discuss in more depth the benefits of using DSLs to achieve high-performance in parallel heterogeneous systems in a productive manner.

### 2.2 Benefits of Using High-Performance DSLs

We will discuss the benefits of using high-performance DSLs by illustrating how they can be used to achieve the goals we outlined at the beginning of this chapter.
2.2.1 Productivity

DSLs can help achieve productivity by allowing us to hide the complexity of low-level parallel programming from the average user. The DSL user implements their algorithms using constructs that express domain operations at a higher level of abstraction. As a consequence of working at this abstraction level much of the lower-level implementation details are provided by the DSL itself rather than the application programmer.

As an example, consider the snippet of OptiQL code shown in Listing 2.6. The snippet shows a collection of contacts being filtered to only those whose state is California and then sorted by last name then first name. The snippet reflects the description of what needs to be done without any further implementation details specified. Anyone who has ever used a query language like SQL should be quite familiar with the constructs provided by OptiQL. Another example of such high level constructs is the \texttt{sum}, or summation, construct of OptiML. The GDA example shown in listing 2.2 makes use of this construct in line 19. The construct has an equivalent meaning to its corresponding mathematical definition which is to sum the result of function for different values of the index of summation. As in the previous example, no further implementation details are specified by the user. Many algorithms in machine learning use summation when written in pseudo-code, DSLs allow us to match the conciseness of this pseudo-code.

This lack of implementation details results in a significant reduction in total number of lines of code as well as improved code readability compared to a general-purpose language. This not only has positive impact on productivity during the initial implementation of a particular algorithm but also increases the maintainability of the programs written using these DSLs. The lack of implementation details also applies to the parallelization of these algorithms. All the examples shown so far are devoid of any explicit parallelization or parallelism construct. All these details are left for the DSL to handle.
In addition to providing a means of writing concise, maintainable code, DSLs can also expose significantly more semantic information about the application than a general-purpose language. In particular domain constructs can expose structured, coarse-grained parallelism within an application. The DSL developer must identify the mapping between domain constructs and known parallel patterns, and with the proper restrictions this allows the DSL to generate safe and efficient low-level parallel code from application source using a sequential programming model.

As an example, consider again the OptiML `sum` construct (shown in Listing 5.1). Summations occur quite frequently in machine learning applications that focus on condensing large input datasets into concise, useful output. The construct allows the user to supply an anonymous function producing the elements to be summed that is subject to the restricted semantics enforced by the OptiML compiler. The anonymous function is not allowed to access arbitrary indices of data structures or mutate global state. This restriction is not overly constraining for the majority of use cases and allows the function to be implemented efficiently as a map operation to compute the result of applying the function to the various values of the index of summation followed by a reduce to compute the sum of these values. In addition, the anonymous function is often non-trivial to evaluate, and therefore exposes coarse-grained parallelism which can be exploited to achieve strong scaling.

In addition to this implicit extraction of parallelism, the domain-specific semantic information exposed by the DSL can be used to apply domain-specific optimizations. In the case of OptiML, we can use linear algebra simplification rules to optimize the program. These domain-specific optimizations can eliminate whole matrix operations while yielding a functionally equivalent implementation to the original non-optimized one. In the case of OptiQL, we can use relational algebra rules to re-order operations...
and again yield functionally equivalent but more optimized implementations of user programs.

Most of these domain specific optimizations would not be possible if the program was written in a general purpose language. General-purpose languages are limited when it comes to optimization for at least two reasons. First, they must produce correct code across a very wide range of applications. This makes it difficult to apply aggressive optimizations—compiler developers must err on the side of correctness. Second, because of the general-purpose nature needed to support a wide range of applications (e.g. financial, gaming, image processing, etc.), compilers can usually infer little about the structure of the data or the nature of the algorithms the code is using. In contrast, DSL compilers can use aggressive optimization techniques using knowledge of the data structures and algorithms derived from the DSL. This makes it possible to deliver good performance on heterogeneous architectures using DSLs.

2.2.3 Portability Forward Scalability

Along with the ability to identify the parallelism inherent in an application, domain abstractions can also abstract away implementation details sufficiently to generate parallel code optimized for various hardware devices. The lack of implementation artifacts in the application source ultimately allows DSL programs to be portable across multiple current and future architectures. For example, real-time 3d computer graphics are implemented using high-level abstractions provided by a library like OpenGL [?]. Programs written using OpenGL are then compiled by a 3d accelerator driver to run on a particular hardware architecture. This allows 3d card vendors to compete and innovate in terms of hardware architecture without having to completely rewrite existing software to take advantage of new hardware capabilities.

The same can be said about SQL, which is mostly standard. In this case, software vendors (i.e. IBM, Oracle, SAP, etc...) can compete and innovate in terms of the capabilities of the relational database management system that will ultimately translate the SQL query into machine code. It is also mostly trivial to parallelize SQL queries, again with little to no modifications of existing code. There are now
on-going effort to use special purpose hardware to accelerate SQL programs with no required changes to the source code. All this is possible as a direct result of the high level, declarative nature of the language.

2.3 The Need for a DSL infrastructure

DSLs provide a structured foundation to identify and exploit parallel execution patterns specific to particular domains. They are not a silver bullet, however. They cannot parallelize existing sequential code, and they do not on their own eliminate the parallel programming burden for the DSL developer.

The first obvious challenge is designing and constructing a new language, namely implementing a full compiler (i.e., a lexer, parser, type checker, analyzer, optimizer, and code generator). In addition, the DSL must have the facilities to recognize parallelism in applications, and then to generate parallel code that is optimized for different hardware devices (e.g., both the CPU and GPU). This requires the DSL developer to be not only a domain expert, but also an expert in parallelism (to understand and implement parallel patterns) as well as architecture (to optimize for low-level hardware-specific details). Finally, the DSL developer must write a significant amount of plumbing whose implementation can have a significant impact on application performance and scalability. This includes choosing where and how to execute the parallel operations on a given hardware platform, managing data transfers across address spaces, and synchronizing reads and writes to shared data.

If we wish to create a few DSLs to handle each of the numerous possible algorithmic domains efficiently, then it would be too costly to handle all the above issues for each DSL in isolation. However, it is important to note that a large number of applications would require a relatively fewer number of DSLs. We believe these DSLs share some recurring parallel patterns which a DSL infrastructure can provide as building blocks. The DSL infrastructure can then take the responsibility of transforming these building blocks to the low level programming models required to use the different heterogeneous processing elements. Before discussing the requirements of this DSL infrastructure, we first discuss the current DSL implementation strategies.
2.4 DSL Implementation Strategies

Traditional DSLs fall into two categories. External DSLs, which are completely independent and allow total design freedom, but consequently require the developer to write a complete compiler, and internal DSLs, which are embedded in a host language. Internal DSLs are significantly easier to develop, but traditionally sacrifice the ability to perform static analyses and optimizations.

External DSLs

Traditionally, DSLs have been developed from the ground-up using custom compiler infrastructure. This is called the “external DSL approach”. There are two problems with this approach to DSLs. First, developing these new languages to a sufficient degree of maturity is an enormous effort. This investment would have to include not just language specifications and construction of their optimizing compilers and libraries, but also all the other aspects of modern tooling including IDEs, debuggers, profilers, build tools as well as documentation and training. It is hard to see how such an investment can be made repeatedly for each specialized domain. Second, DSLs do not exist in a vacuum but have to interface to other parts of a system. For instance, a climate simulation program could have a visualization component that is based on a graphics DSL. It is not clear how multiple separate DSLs would inter-operate without creating a new DSL that integrates the desired combination of the others.

Internal or Embedded DSLs

Embedded DSLs [31] (eDSLs) overcome the problems with external DSLs and make DSL development more tractable. An embedded DSL lives inside of a host language. It is quite like a framework or a library, consisting of a set of classes and operations on objects of those types. There are several advantages to using embedded DSLs for application writers. Programmers do not have to learn a completely new syntax. Multiple DSLs can also more easily be combined in the same application. The DSLs can also leverage the existing host language ecosystem. There are some drawbacks to traditional DSL embedding however. Since the eDSL is just a library in the host
language, it is limited in terms of the type of optimizations it can apply to the user program. It is especially difficult for such eDSLs to reason about the whole program. Another important limitation is that the eDSL can only support hardware targets that are supported by the host language.

2.5 Requirements for High-Performance DSL Infrastructure

In order to evaluate different approaches to providing a DSL infrastructure, we need to first agree on a set of requirements that need to be satisfied by a given DSL infrastructure. The following set of requirements are what we are mostly concerned about for high-performance DSLs.

**Syntax**: This requirement encompasses the need flexibility and a high degree of expressiveness when implementing the DSL syntax so that we can provide domain-specific syntactic sugar. This sugar enhances productivity as it allows domain users to program algorithms using concepts that are familiar to them. The syntactic sugar also allows us to easily extracts the domain-specific semantics from the user application and thus enable more analysis and optimization. Finally syntactic sugar can also restrict users from implementing algorithms in a way that preclude their efficient parallelization.

**Analysis and Optimization**: This requirement encompasses the need to reason about large portions (ideally all) of the program so that proper analysis and optimization of the application is possible. There should be also be a way to express domain-specific optimizations.

**Parallelization**: This requirement encompasses the need to efficiently parallelize the DSL language constructs. This may involve applying different parallelization strategies to the different constructs. An understanding of the access patterns encoded by the DSL construct is essential for proper parallelization.

**Code Specialization**: This requirement encompasses the need to retarget parts of the application to different HW processing elements. This is essential to be able
to take advantage of accelerators such as GPUs. We specify parts of the application as it may not be possible to retarget all parts of a given application to the different HW targets.

The additional cross-cutting requirement that we should mention but not insist on here, as it is subjective and hard to measure, is the degree to which the infrastructure makes it easy or difficult to achieve the above requirements.

The approach that we took was to start with the embedded DSL approach and then improve upon its limitations. DeliteLib, our first prototype DSL infrastructure didn’t require any changes to the host language and its compiler. This library based approach provides some of the potential of DSLs with little effort both in the implementation of the DSLs and the DSL infrastructure. However, the approach has some drawbacks that ultimately limit its usefulness. We solve most of the limitations of the approach while retaining the ease of DSL implementation in our second prototype, DeliteComp, which involved modifying the compiler of the host language. In the next chapter, we take a closer look at DeliteLib.
Chapter 3

DeliteLib: A Library-based Approach

DeliteLib embodies our initial effort at building a DSL infrastructure. DeliteLib did not require any changes to the host language compiler. Hence we consider this first version of the Delite infrastructure, DeliteLib, a library-based approach to solving the problem of simplifying the development of high-performance DSLs. While this approach has some substantial limitations, which will be discussed at the end of the chapter, it nonetheless relies on the same key concepts that are used in DeliteComp, the compiler based approach to creating a high-performance eDSL infrastructure. A more compact description of this approach has also been published [13].

First, let’s present an overview of the DeliteLib approach in terms of the requirements we have developed in section 2.5:

**Syntax:** In order to provide the syntax flexibility required, we adopt a flexible host language, Scala [49]. Scala is a general purpose programming language that smoothly integrates features of object-oriented and functional languages. Its features combine to make it a good host language for the development of embedded domain-specific languages. We will discuss some of these features using examples a bit further in this chapter.

**Analysis and Optimization:** We enable analysis and optimization of embedded DSL programs by building at runtime a dynamic representation. This representation
is the result of deferring the execution of DSL operations. This chapter details how this can be accomplished. Once we have an internal representation (IR) of the program (albeit a partial one) it becomes possible to analyse and optimize the program by transforming this IR.

**Parallelization:** To simplify the parallelization of DSL operations we provide a set of re-usable and extensible execution patterns that the DSL authors leverage during the implementation of their DSL. This is one of the most important contributions of our work and while the implementation details differ, DeliteComp provides a similar mechanism for the parallelization of DSL operations. In both cases, the DSL author maps their domain-specific operations to one or more Delite OP which as we will see doesn’t require a lot of effort.

**Code Specialization:** This prototype of Delite doesn’t do much in terms of providing support for code specialization. The DSL author is responsible for providing implementations (e.g. kernels) of each DSL operation they which to target to a hardware target (i.e. GPU) other than multicore. DeliteLib, however, handles data movement and kernel dispatch on behalf of the DSL author.

The rest of the chapter will delve into the details. We first will look at some of the features of Scala that makes it a good host for embedding DSLs. This will give the reader a good intuition of what we mean by DSL. We then will discuss how we defer DSL operations in order to construct an execution graph consisting of Delite OPs. Parallelization is handled at the Delite OP level. We then discuss the limited support DeliteLib provides for code specialization. We finally evaluate this prototype using OptiML, a machine learning DSL and our first DSL implemented using this infrastructure.

### 3.1 DSL embedding in Scala

In this section we show how Scala’s language features can be used to implement embedded DSLs. These resulting embedded DSLs feel like extensions to the Scala language, however they are just libraries that make clever use of the host’s language feature. While this discussion is specific to Scala, there are other language that
provide similar features (i.e. C#) or provide other features (i.e. haskell) that can be used to achieve the same result, which is to make libraries feel like language extensions. To aid our discussion, we will be using an example shown in listing 3.1. Code snippets provided in this thesis should work with version 2.9.x of Scala.

```scala
// v: Vector[Int]
// m: size of v
// sums the square of the values in the vector
val sumOfSquares = sum(0,m) { i =>
    v(i)*v(i)
}
```

The example shows a summation construct being used to sum the square of the values in a given vector (defined outside of the `sum` construct). This is a frequent pattern in machine learning (ML) applications and thus is provided as part of the OptiML DSL. Note the way this construct is expressed in this example implies that `sum` is a native construct provided by the language. Take a minute and think about how one might provide such a functionality in C or the current version of the Java language (7 as of this writing). It would not be possible without changing the C or Java parser. You could provide a library that did essentially the same thing (e.g. summing the square of a vector that is passed in as an argument), but what if I wanted to sum only the even values? what if I wanted to compute something out of the values before summing? This is the main advantage of a DSL over a library after all, it provides the user with higher order constructs. Ideally a good host language would allow the DSL author to extend the language with domain-specific constructs (i.e. summation). However, the ability to provide these domain-specific constructs shouldn’t come at a high cost to the DSL author.

Ideally little to no knowledge of the host language’s compiler internals should be required. Scala is a good example of a language that goes a long way in solving this problem by providing a set of features that allow DSL authors to build libraries that look and feel like language constructs. It is worth mentioning that Lispers have been enjoying such features for many decades. Language like Scala thanks to their seamless interoperability with Java are just introducing a more mainstream contingent of programmers to features like lambda expressions. We will examine a few versions
of a sequential implementation of \textit{sum} to discuss some of these features. The first version of \textit{sum} that we present can only sum the values of a vector of integer values.

```scala
// Sums the values of a vector of integer values
def sumVectorInt(start: Int, end: Int, vec: Vector[Int]): Int = {
  var res = 0
  for (i <- start to end) {
    res = res + vec(i)
  }
  res
}

// usage
// vec: Vector[Int]
val res1: Int = sumVectorInt(1,4,vec)
```

This is a pretty straightforward implementation of a summation construct of limited value. It only can sum vectors, and specifically only vectors of integer values. This is what you would expect to find in a traditional library. If I wanted to sum a vector of floats then I would have to write an additional function. This is solved by using a mechanism like c++ templates or java Generics. But what if I wanted to double the values of the vector before summing them? I would have to write another function to do this doubling. So let’s build a more general purpose version of the summation function. We are going to make use of Scala’s support for higher-order functions. Higher-order functions can take as arguments other functions. This is very handy in extracting and re-using the functionality common in all summations, namely iterating and summing the result of the function that returns the summands.
Our summation construct is now a lot more powerful. We can sum over any arbitrary function that returns an integer value (e.g., Int). The block that is being passed into `sumInt` can itself call other function, so any valid scala code is allowed. The ability to pass blocks of code to functions is the most important requirement for a host language to be effective at embedding DSLs. Most of the other features we will discuss enhance the look and feel of the DSL. For example, using `sumInt` still feels like making a library call. Two Scala features combine to remedy this. First, Scala allows curly braces and parenthesis to be used interchangeable when calling a function. Second, Scala allows multiple list of arguments to be defined for a function. This last feature was mainly intended to allow for function currying (e.g., partially applying functions). We redefine our function as follows:
def sumIntCurly(start: Int, end: Int) 
  (block: (Int) => Int): Int = {
    var res = 0
    for (i <- start to end) {
      res = res + block(i)
    }
    res
  }

// usage
// vec: Vector[Int]
val res4: Int = sumIntCurly(1,4) {i => vec(i) * vec(i)}

This version of the summation construct can now be used in a more natural, language-like fashion. The next refinement is to make the `sum` construct we have implemented so far be more generic. Meaning, we would like to sum over functions that return numeric values of any type. Scala like many object oriented languages support parametric polymorphism not only for data types (i.e. classes) but also functions. We parametrize our `sum` function on type `T` as shown in the following code snippet. The new problem that is introduced (and also solved) is that we can no longer add (e.g. apply the `+` operator) to the values or objects of type `(T)` that are returned by applying `block` (there is no way to solve this issue in Java as version 6 of the language). We use a few Scala features to solve this elegantly. We first declare a third function parameter `block`. These parameters are preceded by the keyword `implicit:`
def sum[T](start: Int, end: Int)(block: (Int) => T)(implicit numeric: Numeric[T]): T = {
  var res: T = numeric.zero
  for (i <- start to end) {
    res = numeric.plus(res, block(i))
  }
  res
}

// usage
// vec: Vector[Int]
val res5 = sum(1, 4) {i =>
  vec(i) * vec(i)
}

The implicit keyword serves a few functions in Scala; it can be used to designate a list of parameters as implicit. When calling a function with implicit parameters one can omit passing arguments to these implicit parameters. The compiler instead looks for an identifier (val or var) to an object that is in scope, has been marked as implicit at its definition site and is of the same type as the parameter. This object is automatically passed in as argument. The compiler will return an error if there is more than one reference in scope. In the example above, we mark the numeric parameter as implicit. Scala imports by default a few type definitions and immutable instances into every module; this includes a few implicit objects that are of type Numeric[T], one for each primitive type T. Which of these objects is passed depends on the type T used when invoking sum. Furthermore, the actual type T is inferred (via type inference) from the return type of the block supplied to the function sum. The supplied numeric object implements numeric methods such as addition as well as returns important constants such as zero that are specific to the type T. This means that we can use this code for any type T as long as the compiler can find in scope an instance that implements Numeric[X] for that type T.

All these language features (e.g. implicit parameters, type parameters, type inference, curly braces, multiple parameter blocks) combine to yield a very generic and re-usable sum function that has the look and feel of a first-class language construct or language extension. The next set of features we discuss aren’t required to improve for
the `sum` construct per-se but are useful to DSL implementers so we will cover them. One such feature also employs the keyword implicit. If the implicit keyword is used in front of a function definition (the name of the function is of no consequence) that accepts an object of type \( X \) and returns an object of type \( Y \) an implicit conversion from type \( X \) to type \( Y \) is defined. This led to the Pimp my Library pattern \[?] which allows DSL implementers to take an existing type (such as a simple sequential collection) and enrich it with new functionality (other languages such as C# have adopted similar functionality to allow the implementation of DSLs such as LINQ).

Let’s look at an example:

```scala
def sum[T](start: Int, end: Int)(block: (Int) => T)(implicit numeric: Numeric[T]): T = {

  var res: T = numeric.zero
  import numeric._
  for (i <- start to end) {
    res = res + block(i)
  }
  res
}
```

```scala
// usage
// vec: Vector[Int]
val res6 = sum(1,4) {i =>
  vec(i) * vec(i)
}
```

You will have to squint to see any difference, but there is a big difference between this example and the previous one. Note that in the previous example, in order to add \( res \) with \( block(i) \) (line 6) we had to call the plus method of the passed in numeric object. In this example, we simply use invoke the \(+\) method directly on \( res \). But all we know about \( res \) is that it is of type \( T \) and we know nothing about \( T \) so the compiler should have complained that \(+\) is not defined for type \( T \) (line 8). However, on line 6, we imported an implicit conversion from type \( T \) to type \( Numeric[T] \). The implicit conversion wraps the \( res \) object inside an instance of \( Numeric[T] \) which defines and implements a \(+\) method; this is the Pimp my Library pattern in action.
Implicit parameters are also used to ask the compiler to supply compile time information at runtime. For example, we would like to know at runtime what the actual type $T$ is. This is critical on the JVM as all generic type parameters get erased at runtime. Without such a feature, we would lose our ability to obtain the actual type $T$ and we wouldn’t for example be able to instantiate primitive arrays of type $T$ inside the generic code (which knows nothing of the actual runtime type information for $T$).

This final example requests an implicit Manifest object to be passed in. When the compiler sees a request for an implicit $\text{Manifest}[T]$, it constructs an instance specific to the type $T$ inferred at the call site of the $\text{sum}$ construct. $\text{Manifest}[T]$ is an opaque descriptor that can be used to recover the erased type information and for instance can be used to instantiate native arrays of the type $T$.

```scala
def sum[T](start: Int, end: Int)
  (block: (Int) => T)
  (implicit numeric: Numeric[T],
   manifest: Manifest[T]): T = {
  var res: T = numeric.zero
  import numeric._
  for (i <- start to end) {
    res = res + block(i)
  }
  println(‘‘Return type: ‘‘ + manifest.toString())
  res
}
```

In the above example, we use the manifest to print the type of $T$ without manifest information, all what would print is AnyRef (which stands for java Object).

Hopefully, you should be feeling a lot more comfortable with some of the Scala features that we one can employ to build libraries and give them the look and feel of language extensions, we will call such libraries eDSLs for the remainder of this thesis. The next section discusses how one can parallelize such eDSLs using the Delite infrastructure. Specifically, we will discuss the first Delite prototype which did not require any changes to Scala compiler (library-based), we call this prototype DeliteLib.
3.2 DSL Parallelization Using DeliteLib

The previous section has shown how one can use a combination of expressive language features to make a library look like a language extension or DSL. This section will discuss how one can use the DeliteLib to introduce parallelism into an embedded DSL.

To take advantage of our infrastructure, the DSL author needs to follow certain conventions when designing their DSLs. A DSL author will typically start with some domain analysis that includes looking at various applications within the domain (some already parallelized). The DSL will be looking for recurring patterns of execution or operations that occur across a variety of programs. These operations alongside the types on which they operate become the backbone of a particular DSL design.

DeliteLib provides the DSL author with general data-parallel (and a single task) execution patterns that can be extended by the author when implementing their DSLs, we call these Delite OPs. Thus, the act of implementing a parallel version of a DSL with Delite(Lib) consists of mapping domain operations to appropriate Delite OPs. As we will see, this process of mapping from a domain-specific operation to one of the general patterns is very simple. After the mapping process is complete, Delite(Lib) handles most (all in many cases) parallelization aspects on behalf of the DSL.

Delite(Lib)’s execution model enables implicit parallelization of applications by providing facilities for deferral of method execution. Each method invocation can be packaged as a Delite op and submitted to the Delite runtime. Ops encode their immediate dependencies which allows the runtime to build a dynamic execution graph.

To illustrate the steps in execution, consider the simple application code snippet written using OptiML shown in Figure 5.2. The application thread executes sequentially (Figure 5.2a). However, each call to the plus and times methods returns immediately after submitting its op to the runtime. These ops encode any dependencies on other data objects. A Matrix proxy (this object implements the Matrix interface but has no valid data elements) is returned as the result of the method invocation (Figure 5.2b). The application is oblivious to the fact that computation is deferred
Application

```scala
def example(a: Matrix[Int], b: Matrix[Int], c: Matrix[Int], d: Matrix[Int]) = {
  val ab = a * b
  val cd = c * d
  return ab + cd
}
```

Calls OptiML DSL methods

```scala
def *(m: Matrix[Int]) = delite.defer(OP_mult(this, m))
def +(m: Matrix[Int]) = delite.defer(OP_plus(this, m))
```

OptiML DSL defer OP execution to Delite

Delite maps task graph to resources

Hardware Schedule

Figure 3.1: Delite application execution overview.

and “runs-ahead” allowing more ops to be submitted. The submitted ops form a dynamic task graph (Figure 5.2c). Given a program’s task graph, the runtime system is able to target a variety of parallel architectures automatically (Figure 5.2d). Independent parts of the task graph are scheduled to run in parallel and data movement is minimized with a scheduling algorithm that takes communication costs into account. Therefore, Delite automatically provides implicit task parallelism for each DSL at the operation granularity that the DSL defines. Furthermore, data-parallel tasks can be further decomposed into multiple independent tasks yielding more parallelism.

The next set sections provide more details on how Delite(Lib) simplifies the process of parallelizing eDSLs.

### 3.3 Deferred Execution via Proxies

Delite(Lib) consist of a framework and a runtime. The framework is used to implement and parallelize eDSLs while the runtime is responsible for running programs
that are written using these eDSLs. A framework typically consist of a mix of con-
ventions and software artifacts in support of these conventions. The main convention
we mentioned so far is that the DSLs consist of a set of types and a set of operations
defined for each type. A DSL implemented using the DeliteLib framework is implic-
itly parallel and takes advantage of both task-level and data-level parallelism. We
will discuss how DeliteLib facilitates task-level parallelism in this section and hold
the discussion about data-level parallelism to the next section.

We say that the DSL is implicitly parallel as there is no explicit parallelization
construct exposed to the DSL user. A program that uses a DSL implemented in
Delite(Lib) looks imperative and is a collection of calls to DSL operations which yield
results and subsequent calls to operations on those results. DeliteLib parallelization
works by having the application thread “run-ahead”, instead of actually computing
results when calling the DSL operations, it simply gets back a proxy, which unlike
most implementations of futures, is \textit{transparent} to the caller. The proxy has the same
return type as a concrete result, and can be used interchangeably with other proxies
and concrete instances. This allows a DSL user to write code that is oblivious to the
underlying execution model.

EXPR DSL example

In order to illustrate these concepts, we will show how one can implement a very
simple DSL that represents arithmetic expressions using the DeliteLib framework.
The first step in implementing our DSL is to start with a simple embedding in Scala
using some of the techniques we discussed in section 3.1 (in this case not much is
required).
class Expr(val value: Int) {

    def +(rhs: Expr): Expr = {
        new Expr(this.value + rhs.value)
    }

    def -(rhs: Expr): Expr = {
        new Expr(this.value - rhs.value)
    }

    def *(rhs: Expr): Expr = {
        new Expr(this.value * rhs.value)
    }

    override def toString() = value.toString()
}

and a sample usage of the DSL in its current form is as follows:

\Usage:\nval a = new Expr(4)
val b = new Expr(5)
val c = new Expr(8)

val d = (a*b)+(a*c)
println(d)
\outputs:\n52

As an additional more realworld example, figure 3.2 shows the Delite class diagram for the Vector object in the OptiML DSL. Think of a Scala trait as similar to a Java interface for the time being. Here, Vector[T] is the only type the DSL user interacts with. In this example the data reference in VectorImpl will initially be null upon creation and the VectorImpl object acts as a proxy. Once the op responsible for the creation of the proxy completes execution, the VectorImpl object acts as a concrete instance with its data reference set to the concrete array containing the result. The force method inherited by Vector and all other DSL types enforces synchronization.
when required, preventing data from being accessed before it exists. If a proxy result is required (e.g. due to a control dependency) but is not yet ready, the proxy is implicitly forced to execute and return a concrete result.

3.4 Delite Ops

3.4.1 A framework for creating implicitly parallel DSLs

Delite allows DSL authors to integrate a DSL into the Delite runtime by providing a framework of extendable types and interfaces, the most important of which are the Delite op archetypes.

When defining a Delite op, the DSL author specifies its dependencies and the return type of the proxy. Listing 3.1 shows how a simple op can be written to subtract two vectors. Its input arguments, $v_1$ and $v_2$ (which may themselves be proxies), are added as dependency edges on the task graph when this op is submitted to the runtime. Although this op can be written in a data-parallel manner, for this example we use a simple sequential implementation by extending DeliteOP.SingleTask. This type can be used for any sequential task and requires only a task method, which will be invoked by Delite when the op is executed. Data-parallel ops have a richer interface that simplifies data decomposition and parallel execution; these are described in Section 3.4.2.

The DSL author is free to package work into Delite ops however he or she deems best. So far we’ve focused on how a method call can be naturally translated into an op, but there are other possibilities. The sum control structure in OptiML creates two Delite ops, one to generate all the temporary results and another to perform the final summation.

3.4.2 Data-parallel operations

Delite exposes data-parallelism by providing support for various data decomposition patterns. The framework provides data-parallel op archetypes as classes that can be extended. Some of the currently supported archetypes include op_map, op_reduce
trait DeliteDSLType[T]
final def force: T = { ... }

trait DeliteCollection[T]
def size: Int
def chunk(start: Int, ... = data.length
def gpu_data = data
def dc_apply(n: Int) = data(n)
def dc_update(n: Int, x: T) {
  data(n) = x
}

class VectorImpl[T]
var data: Array[T] = null
def length = data.length
def gpu_data = data
def dc_apply(n: Int) = data(n)
def dc_update(n: Int, x: T) {
  data(n) = x
}

Figure 3.2: Class diagram of the OptiML Vector class.
CHAPTER 3. DELITELIB: A LIBRARY-BASED APPROACH

protected[optiml] case class OP_−[A]
(v1: Vector[A], v2: Vector[A])
exends DeliteOP_SingleTask[Vector[A]](v1,v2) {
  def task = {
    if (v1.length != v2.length)
      throw new IndexOutOfBoundsException
    val result = Vector[A](v1.length)
    for (k <- 0 until v1.length){
      result(k) = v1(k) - v2(k)
    }
    result
  }
}

Listing 3.1: An example sequential op in the OptiML DSL.

protected[optiml] case class OP_−[A]
(val collA: Vector[A], val collB: Vector[A],
val out: Vector[A])
  def func = (a,b) => a - b
}

Listing 3.2: An example data-parallel op in the OptiML DSL.

and op.zipWith. Each op operates on a DeliteCollection, which is a trait that each collection-based DSL type implements. This trait and its implementation for OptiML's Vector is shown in Figure 3.2. The DSL author provides data-parallel ops by extending one of the archetypes in a very similar manner as a sequential task. In the data-parallel case, however, rather than providing the task to be executed, the DSL author simply provides the function to be performed. Listing 3.2 shows how the same subtraction op from Listing 3.1 can be written to obtain automated data-parallel execution.

Delite is responsible for sizing and scheduling the chunks of a DeliteCollection,
allowing the runtime to adapt the needs of the application to the available resources in the system. \texttt{DeliteCollection} provides Delite with a flat, uniform interface to the elements of arbitrary data structures. This allows the DSL author to encode domain-specific decompositions. Specifically, the DSL author must implement a \textit{size} and \textit{chunk} method for the collection. As the name implies, \textit{size} is the total number of elements of the collection. \textit{chunk} takes two indices and returns an iterator over the requested range. The DSL author is free to construct the iterator in any way, and can therefore optimize the decomposition for locality without Delite knowing anything about the underlying data structure. For array-based collections, Delite maximizes performance by iterating over the array elements directly utilizing flat read and write methods (\texttt{dc_apply} and \texttt{dc_update}) that the DSL author implements. In addition, primitive array-based collections can choose to implement the \texttt{GPUableCollection} interface which allows Delite to manage shipping the op’s corresponding CUDA kernel to the GPU. The details of Delite’s GPU support are discussed in Section 3.5.2. All data-parallel ops in Delite maintain the illusion of atomic execution, so the DSL author does not need to worry about the case of a partially complete data-parallel operation.

\section*{3.4.3 Handling side effects}

In the preceding discussion of the Delite execution model we have implicitly assumed all ops are functionally pure (side-effect free). However, Delite also allows ops that mutate the state of DSL objects. Delite deals with the possibility of side effects and potential race conditions by restricting, tracking and isolating mutating operations. First, ops are restricted to only mutate the state of DSL objects. This restriction combined with the fact that all op inputs are either primitives (immutable) or DSL objects prevents data consumed by ops to be mutated arbitrarily. Next, any op that mutates data or has side effects must explicitly declare the object(s) it is going to mutate. The DSL author provides this information using the op interface in essentially the same way as declaring inputs. Finally, using these declarations Delite adds the anti-dependencies created by these mutating ops as it builds up the task graph and
enforces sequential correctness at execution time by honoring these additional edges in the task graph. This mechanism allows Delite to support fast, mutating ops and avoid costly copy-by-value operations.

### 3.4.4 Assisting GPU code generation

Similar to merge in the sense that we need the DSL author to provide GPU kernels. The Delite Runtime is still responsible for managing communication on behalf of the DSL author and user.

In order for the Delite runtime to target a GPU device automatically, the DSL must provide a corresponding CUDA kernel for each op that the DSL wants to be shipped to the GPU. In the case of regular operations this task is simplified by automatic code generation from Scala code to CUDA code. This is feasible due to restricted semantics: the op must use disjoint memory accesses that allow it to be transformed into a data-parallel operation in a straightforward way. The DSL author can mark each op for which he or she wishes Delite to generate a CUDA kernel using the `@GPU` annotation. A compiler plugin is then used to map the regular Scala code to CUDA. For ops that are irregular or benefit greatly from hand optimization, the DSL author must provide an appropriate CUDA kernel.

### 3.5 A heterogeneous parallel runtime

We now turn our attention to how Delite executes ops on heterogeneous parallel hardware. The current version of Delite supports execution on multiple CPUs and a GPU in a single machine. We plan to expand to supporting clusters in the near future as well as other accelerators as they become available.

#### 3.5.1 Scheduling

Delite schedules ops to run from the window of currently deferred ops, honoring the dependencies and anti-dependencies present in the task graph. Ops are scheduled using a low-cost clustering heuristic in order to minimize communication costs among
ops as well as scheduling overhead. Data-parallel ops are submitted to the runtime as a single op and later split into the desired number of op chunks. The number of chunks is chosen at scheduling time based on the size of the collection and the availability of hardware resources in the system. When scheduling op chunks, the locality concern of the scheduler is modified to consider the dependencies among individual chunks of the op input and output data structures rather than the objects as a whole. Scheduled ops are enqueued to run on the selected resource and perform a final safety check immediately before executing to ensure all dependencies and anti-dependencies have been satisfied at that time.

3.5.2 Supporting GPU execution

Unlike other environments such as MATLAB which require nontrivial user effort to explicitly mark operations and data structures that should be shipped to the GPU, Delite schedules onto CPU and GPU resources in a way that is completely transparent to the user, allowing a single version of the application source code to target either execution environment. The DSL author determines which ops are appropriate for the GPU and implements an additional interface that allows Delite to manage shipping the op to the GPU. The DSL author provides a CUDA kernel for each op as described in Section 3.4.4. The interface also allows the DSL author to disable GPU execution of the op when desired (e.g., when the dataset is very small). The GPU manager can override this hint when locality concerns dictate that poor computation performance is preferable to high communication cost.

In order to support automated GPU execution, Delite implements a memory manager for each GPU device in addition to all the facilities it provides for CPU execution. The entire main memory of the device is pre-allocated at startup and then managed as a cache by Delite at run-time. When ops are scheduled for execution onto the GPU, Delite ships the corresponding CUDA kernel to the GPU device and automatically injects the required memory transfers if all of the op’s inputs are not currently in the device cache. All data transferred to the device remains there for reuse until the CPU requires a result. Ops that merely produce temporary results may never return
data back to the CPU. This optimization fits naturally into our model of DSL objects that only contain valid data upon requirement. When data created by or mutated by the GPU is required by the CPU, the GPU memory manager automatically transfers the appropriate data back to the CPU main memory. Finally, if the CPU performs a mutating operation the GPU cache copy (if one exists) is invalidated.

3.6 Runtime Features

3.6.1 Scheduling

3.6.2 GPU Device Management

3.7 Evaluation of DeliteLib

In this section, we present performance results for a set of machine learning applications written in OptiML and running on Delite. Each application is written without any explicit parallelization. We compare our results to reference implementations written in MATLAB, including GPU implementations that use either MATLAB 7.11’s beta support or AccelerEyes’s Jacket [2]. We also show how each application scales on a highly threaded system and analyze what limits the application’s scalability. We conclude by showing the impact of the domain-specific optimizations described in Section ?? on performance.

3.7.1 Methodology

We implemented four versions of each application: an OptiML version, a MATLAB version, a MATLAB version using MATLAB 7.11’s GPU constructs, and a MATLAB version using Jacket’s GPU constructs. We use the same OptiML application code to run on multiple CPUs or on a combination of CPU+GPU; we use a run-time configuration option to enable or disable shipping OptiML ops that have corresponding GPU kernels to the GPU device. The MATLAB versions are algorithmically identical to the OptiML versions, but we make a reasonable effort to vectorize and parallelize
the MATLAB code. For MATLAB parallelization, we used the parallel computing toolbox (specifically the `parfor` construct). When both vectorization and parallelization are possible for a particular loop, we choose whichever method yields the fastest running time at the highest processor count. For the GPU version of the MATLAB applications, we offloaded all GPU-supported operations that were computationally intensive. Jacket supports more operations than MATLAB, such as matrix and vector indexing, and we took advantage of those.

We present results on two machines with significantly different characteristics. The performance and optimization experiments are run on a Dell Precision T7500n with two quad-core Intel Xeon X5550 2.67 GHz processors. Each core has 2-way hyperthreading for a total of 16 hardware thread contexts. It has 24GB of RAM and an NVIDIA GTX 275 GPU. This system is representative of currently available high performance client machines. The scalability experiments are run on a Sun UltraSPARC T2+ with four 8-core 1.16 GHz processors. Each core has 8-way multithreading for a total of 256 hardware threads. It has 128 GB of RAM. Although the T2+’s single-threaded performance is much slower than the X5550’s, its large number of hardware threads and greater memory bandwidth make it much more suited for studying scalability at high thread counts.

We ran our experiments using Sun’s Java SE Runtime Environment 1.6.0_16-b01 and the HotSpot 64-bit server VM with default options. For each experiment we time the computation part of the application, ignoring initialization steps. We run each application, including initialization steps, 10 times to warm up the JIT and smooth out fluctuations due to GC. We report the average time of the last five executions. Table 3.1 presents the applications used in our study. These applications all come from the machine learning domain.

### 3.7.2 Performance comparison

We begin by comparing the performance of the various versions of our machine learning applications running on the Intel-based system. The results of this experiment are shown in Figure 3.3. The figure shows execution time normalized to the OptiML
Figure 3.3: Execution time of the various versions of our applications normalized to OptiML running in sequential mode. Speedup numbers are reported on top of each bar.
version running in sequential mode without any Delite overheads (OptiML ops are not deferred in this case, but instead are executed as they are encountered). We discuss issues relating to the scalability of each application in the next subsection. Single-threaded performance of the OptiML version of our applications is better than MATLAB in most cases. The MATLAB parallel constructs use MPI, which adds significant overhead. To measure the impact of these parallel overheads on single-threaded performance, we have written and run sequential MATLAB versions of our applications and found that OptiML, including Delite overheads, matches or exceeds sequential MATLAB performance. Although MATLAB scripts are interpreted, linear algebra operations utilize native BLAS libraries under the hood. Therefore in applications that are largely composed of matrix and vector operations, such as RBM, the single-threaded MATLAB version performed comparably to a C implementation also calling BLAS functions as well as to the OptiML implementation. Applications requiring more loops, conditionals, and pointer chasing, such as LBP, exhibited worse single-threaded performance in MATLAB compared to OptiML largely due to interpretive overheads.

In all applications except for RBM, OptiML outperforms the explicitly parallelized MATLAB versions. RBM has several fine-grained vector operations for which

<table>
<thead>
<tr>
<th>NAME</th>
<th>DESCRIPTION</th>
<th>PERFORMANCE COMPARISON INPUT</th>
<th>SCALABILITY TEST INPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gaussian Discriminant Analysis (GDA)</td>
<td>Generative learning algorithm for modeling the probability distribution of a set of data as a multivariate Gaussian</td>
<td>1,200x1,024 matrix</td>
<td>1,200x512 matrix</td>
</tr>
<tr>
<td>Loopy Belief Propagation (LBP)</td>
<td>Graph-based inference algorithm using message-passing</td>
<td>23,768 edges 3,630 nodes</td>
<td>23,768 edges 3,630 nodes</td>
</tr>
<tr>
<td>Naive Bayes (NB)</td>
<td>Fast, low-work supervised learning algorithm for classification</td>
<td>25,000x1,448 matrix</td>
<td>12,000x1,448 matrix</td>
</tr>
<tr>
<td>K-means Clustering (K-means)</td>
<td>Unsupervised learning algorithm for finding similar clusters in a dataset</td>
<td>262,144x3 matrix</td>
<td>25,000x100 matrix</td>
</tr>
<tr>
<td>Support Vector Machine (SVM)</td>
<td>Optimal margin classifier, implemented using the Sequential Minimal Optimization (SMO) algorithm</td>
<td>800x1,448 matrix</td>
<td>800x1,448 matrix</td>
</tr>
<tr>
<td>Restricted Boltzmann Machine (RBM)</td>
<td>Stochastic recurrent neural network, without connections between hidden units</td>
<td>2,000 Hidden Units 2,000 Dimensions</td>
<td>2,000 Hidden Units 2,000 Dimensions</td>
</tr>
</tbody>
</table>

Table 3.1: Applications used for experimental analysis.
MATLAB uses highly optimized BLAS implementations. We are adding support for more fine-grained BLAS implementations within OptiML to close the gap. The parallel MATLAB version of k-means exhibited poor performance at all thread counts, so we opted to use a vectorized version instead. This is an example of the difficult and unclear trade-offs required to get scalable performance with MATLAB. LBP is an example of an application that is not well-suited to MATLAB. One possible implementation that exploits parallelism required a great degree of pointer indirection which is slow in MATLAB. Our chosen implementation removes pointer indirection by storing messages in a single shared array. This yields better performance but eliminates the ability to parallelize the application safely using \texttt{parfor}.

We now compare the performance of our applications running on a combination of CPU and GPU resources. GDA and RBM achieve good speedups compared to the CPU-only version. This is due to two factors: first, these applications do not require frequent synchronization between the CPU and GPU. Second, they use large matrices with regular memory access patterns. SVM is similar except that the CPU and GPU must exchange data on every iteration of the convergence loop, resulting in significantly worse performance. The Delite GPU manager (with input from the DSL) dynamically determines which ops should be shipped to the GPU to maximize the overall performance. This determination is at the granularity of individual operations. This results in improved performance by avoiding the overhead of executing small kernels on the GPU. In contrast, the MATLAB and Jacket GPU implementations require the application to explicitly specify which data structures reside on the GPU; all subsequent operations on GPU data structures must occur on the GPU. NB and k-means contain loops with regular memory access patterns that can easily be translated to CUDA kernels. This results in substantially improved performance over the MATLAB implementations, which only ship individual MATLAB operations. Finally, LBP is a low dimensional, highly irregular graph algorithm and is not easily mapped to GPUs.
Figure 3.4: Scalability of our selected OptiML applications running in a highly threaded system. Speedup is measured relative to OptiML running in sequential mode.

### 3.7.3 Scalability study

Figure 3.4 shows the result of running our OptiML applications in a highly threaded environment. The UltraSPARC T2+ has a total of 64 integer ALUs and 32 floating point units. Hence, most of the applications are limited to a maximum speedup of around 64. Most applications also suffer from low arithmetic intensity, which as we noted earlier is a characteristic of the machine learning domain, leading to sub-linear speedups. We next look at what limits each application’s scalability in detail:

**Gaussian Discriminant Analysis (GDA):** GDA scaling is limited in two ways. While algorithmically the program is both very parallel and regular, low arithmetic intensity causes the memory system to become a bottleneck. Additionally, the final reduction phase in the application causes greater serial overhead at high thread counts.

**Naive Bayes (NB):** NB also suffers from low arithmetic intensity. It simply streams through a large dataset and collects some statistics. This amounts to very little computation and while we stream through chunks of the dataset in parallel, we are ultimately limited again by the memory system.

**K-means Clustering:** k-means, unlike many of our applications, has high arithmetic intensity. The algorithm is limited by the number of clusters and the dimensionality of the training data. We chose a representative dataset containing a large number of training samples and clusters which results in good scalability.
Loopy Belief Propagation (LBP): LBP is a graph-based algorithm characterized by irregular computation. Delite uses dynamically load-balanced tasks for the parallel work, which provides much better performance than a static schedule. However, LBP is limited by the fact that a single node in the graph can become a bottleneck if it has a significant portion of the total number of messages to be sent in an iteration.

Support Vector Machine (SVM): Our SVM implementation is a simplified version of a widely used algorithm called Sequential Minimal Optimization (SMO). SMO is an iterative algorithm with loop carried dependencies across iterations. Within each iteration, there is limited fine grained data-parallelism that does not scale well. Beyond 32 threads, SVM runs out of parallel work and is limited by Amdahl's law.

Restricted Boltzmann Machine (RBM): RBM is similar to SVM in that it is iterative with dependencies carried across iterations. However, each iteration is dominated by large matrix multiplications and thus RBM exhibits good scalability.

3.7.4 The benefit of domain-specific optimizations

In Section ??, we described two domain-specific optimizations that OptiML supports: best effort computation and relaxed dependencies. Figure 3.5 shows the improved performance that results from applying these optimizations to k-means and SVM. For k-means, we used a converging best effort policy that dropped distance calculations that were unchanged in the previous n iterations. We show results for three different values of n, each of which results in a different trade-off between performance and accuracy. The error we report is the average percentage difference in the final cluster locations. For this experiment, we clustered a 262144 pixel RGB image, and the best effort optimization drastically reduces computation time with only minor differences in the discovered clusters.

As we mentioned in the previous section, the SMO implementation of SVM has inter-loop dependencies that prevent parallelization across iterations. In this experiment, we used a smaller training set with less available data parallelism. However, we enabled the relax dependency optimization in OptiML, which allows two iterations
to run in parallel inside the \textit{until converged} implementation (note that no changes are required in the application code). This optimization is able to improve performance despite unprotected races to shared state because the algorithm is probabilistic and relatively robust to this kind of manipulation. The iterations frequently access disjoint parameters and SMO is ultimately able to complete faster with less than a 1% loss in classification accuracy.

### 3.8 Limitations of Library Approach

OptiML is implemented as an embedded DSL within Scala. The current version consists of a compiler plugin and a library. The compiler plugin allows OptiML to support user-supplied anonymous functions as parameters to operations. If an anonymous function depends on or mutates another data structure, this dependency must be registered with the runtime. The compiler plugin inspects closures inside OptiML applications and extracts dependencies, wrapping them in an object that is dynamically passed on to the runtime. The plugin can also statically check that OptiML programs obey the restricted semantics of its control structures, described later.
Chapter 4

Building Blocks for High-Performance DSLs

[TODO: Don’t claim this is my contribution, but we contributed to LMS development. This is mainly background to Delite 2.0. ] [TODO: Make sure I talked about the benefits of embedded DSLs elsewhere, also some of this text may end up having to move to the previous chapter ]

We have discussed how embedded DSLs can overcome the problems associated with external DSLs and simplify DSL development. However, the main problem with traditional embedded DSLs is that they are essentially nothing more than libraries implemented in a sufficiently expressive language which provides their user with the illusion of using a DSL. We have shown how one could go about handling some of the issues required to take advantage of the parallelism and heterogeneity offered by modern computing systems. However not all issues can be cleanly handled by such an approach. For example, it is difficult to reason about the whole program as the library can only see a window of upcoming domain operations. Some optimizations such as dead code elimination where one would be able to eliminate whole chunks of the program if results are not needed can be foiled by inserting by the inability of the library to speculate beyond conditionals. One could introduce a fake conditional (e.g. IF as opposed to if) and some efforts like Array Building Blocks [46] do just that, however this might be confusing to the DSL users thus reducing the benefits of
embedding which included not having to learn lots of new syntax.

Another major limitation of the library-based approach to embedding is that the DSL is limited to targeting the underlying hardware supported by the host system. In the case of DeliteLib, this is whatever the JVM supports. As of this writing the JVM doesn’t support GPU hardware directly. In DeliteLib, the DSL author had to provide pre-written GPU kernels that the runtime can then manage dispatching work to, however wouldn’t it be nice for the DSL author not to worry about writing kernels for DSL operations that follow the same execution patterns supported by Delite OPs. After all, for such operations, the DSL author doesn’t have to write any explicitly parallel code for DeliteLib to expose and take advantage of data-parallelism on the CPU. This is a fundamental limitation of DeliteLib, we have duck taped around it by adding a compiler plugin to generate some of these kernels for map operations, but at that point, we could no longer claim that this is a library-based approach. In addition, writing a scala compiler plugin is a low-level affair. One needs to be aware of the Scala AST format, which is to low-level for what we envisioned a DSL author would need to interact with.

So we tried to investigate how to still achieve the benefits of embedded DSLs but also provide some of the capabilities of stand-alone DSLs. Namely, the ability to reason about the whole program statically and the ability to generate code for a variety of target platforms without having to write compiler plugins and become versed in the inner workings of the host language’s compiler. This involved thinking about the capabilities of the underlying host language. Specifically, what does it take to be a good hosting language for high-performance embedded DSLs. We grouped these capabilities under the term language virtualization.

For parallel DSLs to be a tractable approach, they must be easy enough to create for many domains. Traditionally, there are two types of DSLs: internal and external. Internal DSLs are embedded in a host language, and are sometimes called the ”just-a-library” approach [?]. These DSLs typically use a flexible host language to provide nice syntactic sugar over library calls. While this is the easiest possible approach (no compiler necessary), it fundamentally constrains the capabilities of the DSL.
purely embedded, or library-based, DSL cannot build or analyze an intermediate representation (IR) of user programs. This means that they can only perform dynamic analyses and optimizations and these handicaps can severely impact achievable parallel performance. More importantly, without an IR, DSLs cannot do their own code generation, which prevents retargeting DSL code to heterogeneous devices.

The other class of DSLs is external. These are the DSLs that are implemented as a standalone language [?]. While these DSLs obviously do not have the limitations of internal DSLs, they are extremely difficult to build. The DSL developer must define a grammar and implement the entire compiler framework as well as tooling to make it useful (e.g., IDE support). This is clearly not a scalable approach. To solve this dilemma, we propose a hybrid approach. We use the concept of language virtualization [12] to characterize a host language that allows implementing embedded DSLs that are virtually indistinguishable from standalone DSLs. A virtualizable host language provides an expressive, flexible front-end that the DSL can borrow, while allowing the DSL to leverage metaprogramming facilities to build and optimize an IR. Figure 4.1 demonstrates this separation.

Language virtualization is an effective way to define and implement DSLs inside a sufficiently flexible host language. However, building parallel DSLs adds new challenges, such as implementing parallel patterns, launching and scheduling parallel tasks, synchronizing communication, and managing multiple address spaces. Therefore, we implemented a reusable compiler infrastructure and runtime (the Delite Compiler Framework and Runtime) to make developing parallel DSLs even easier. The
framework provides common parallel execution patterns, and DSL developers can easily implement a DSL operation by mapping it to one of the patterns. This mapping process requires minimum effort because most of the behavior is already encoded in the pattern. For example, to implement an operation that iterates over a collection and updates each element by applying a given function, only the function behavior needs to be specified by the DSL developer. The boilerplate code (e.g., iterating over the loop, updates, parallelization) is managed by the framework and the runtime. The framework also provides code generators for those patterns, and therefore the DSL can target heterogeneous hardware (multi-core CPU, GPU, etc) without developing code generators for each target. In addition, efficient scheduling and exploiting the task & data parallelism with proper synchronization are managed by the runtime, which used to be another burden on the DSL developer. Therefore, our system allows DSL developers to focus on the language design rather than on the implementation details and enables exploiting heterogeneous parallel performance without writing any low-level parallel code.

4.1 Language Virtualization

We start by proposing a definition of language virtualization, we then discuss the requirements of language virtualization captured by our proposed definition.

4.1.1 Definition

A programming language is virtualizable with respect to a class of programming languages if and only if it can provide an environment that makes the embedded implementation of these programming languages essentially identical to a corresponding stand-alone language implementation in terms of expressiveness, performance and safety – with only modestly more effort than implementing a simple embedding of the language.

4.1.2 Requirements
[TODO: We should discuss what is absolutely required and what is not, and what are the workarounds]

In this section, we will expand on the requirements that have been emphasized in our definition of language virtualization. We will also give examples of how these requirements can be addressed. We do not provide a solution for the problem of language virtualization as we do not believe this is a solved problem, we merely provide a framework by which we can evaluate potential solutions to this problem. We also note that full language virtualization is not required to start achieving benefits in the presence of partial language virtualization.

Modest Effort

It makes sense to discuss this requirement first, as the solution provided for the other requirements need to also be measured in terms of the amount of effort they require on the part of the DSL author.

We are able to provide language constructs such as the summation construct without having to modify or even become aware of the compiler’s internals. Modest effort is the only criterion that has no counterpart in hardware virtualization. However, it serves an important purpose since an embedded language implementation that takes a DSL program as a string and feeds it into an external, specialized stand-alone compiler would trivially satisfy criteria expressiveness, performance and safety. Building this implementation, however, would include the effort of implementing the external compiler, which in turn would negate any benefit of the embedding. In a strict sense, one can argue that virtualizability is not a sufficient condition for a particular language being a good embedding environment because the “simplest possible” embedding might still be prohibitively expensive to realize.

However, having to implement the lifting for each new DSL that uses a slightly different AST representation would still violate the effort criterion. Using an existing multi-stage language such as MetaOCaml [25, 67] would also likely violate this criterion, since the staged representation cannot be analyzed (for safety reasons we will consider shortly) and any domain-specific optimizations would require effort comparable to a stand-alone compiler. Likewise, compile-time metaprogramming approaches
such as C++ templates [72] or Template Haskell [64] would not achieve the goal, since they are tied to the same target architecture as the host language and their static nature precludes dynamic optimizations (i.e. recompilation).

What is needed here is a dynamic multi-stage approach with an extensible common intermediate representation (IR) architecture. In the context of Scala, we can make extensive use of traits and mixin-composition to provide building blocks of common DSL functionality (API, IR, optimizations, code generation), including making parts of Scala’s semantics available as traits. This approach, which we call lightweight modular staging [59], is described below and allows us to maintain the effort criterion. A key element is to provide facilities to compile a limited range of Scala constructs to architectures different from the JVM, Scala’s primary target.

**Expressiveness**

Conventional libraries are constructed as a collection of data structures and functions with fixed functionality. Embedded DSLs on the other hand are more flexible, they might still include libraries and domain-specific data structures but DSLs also include language constructs which allow a DSL user to design and implement new algorithms in the domain in question.

These language constructs have a syntactic component which allows the construct to be used in the first place. We have shown in section 3.1 how language features can be used to do just that.

However, a host language usually provides its own set of basic building blocks (i.e. for-loop, conditionals, etc.). A DSL may need to override the behavior of such constructs.

Language constructs also have a semantic component. A DSL must be able to check whether the constructs it provides are used correctly. This implies that the DSL must be able to analyze the program that the user wrote. It must also be able to expose the domain-knowledge to the rest of the compilation infrastructure so that this knowledge can be used in optimization and parallelization.

We can maintain expressiveness by overloading all relevant host language constructs. In Scala, for example, a for-loop such as
for (x <- elems if x % 2 == 0) p(x)

is defined in terms of its expansion

```scala
elems.withFilter(x => x % 2 == 0)
  .foreach(x => p(x))
```

Here, `withFilter` and `foreach` are higher-order methods that need to be defined on the type of `elems`. By providing suitable implementations for these methods, a domain-specific language designer can control how loops over domain collections should be represented and executed.

To achieve full virtualization, analogous techniques need to be applied to all other relevant constructs of the host language. For instance, a conditional control construct such as

```scala
if (cond) something else somethingElse
```

would be defined to expand into the method call

```scala
__ifThenElse(cond, something, somethingElse)
```

where `__ifThenElse` is a method with two call-by-name parameters:

```scala
def __ifThenElse[T]
  (cond: Boolean, thenp: => T, elsep: => T)
```

Domain languages can then control the meaning of conditionals by providing overloaded variants of this method which are specialized to domain types.

In the same vein, all other relevant constructs of the host language need to map into constructs that are extensible by domain embeddings, typically through overloading method definitions.

**Performance**

Need to be able to optimize code.

Need to be able to target different new parallel hardware such as the new accelerators.

*Performance* implies that programs in the embedded language must be amenable to extensive static and dynamic analysis, optimization, and code generation, just
as programs in a stand-alone implementation would be. For many embedded languages, in particular those that are the focus of this paper, this rules out any purely interpretation-based solutions.

As we have argued above, achieving performance requires the ability to apply extensive (and possibly domain-specific) optimizations and code generation to embedded programs. This implies that embedded programs must be available at least at some point using a lifted, AST-like intermediate representation.

Pure embeddings, even if combined with (hypothetical) powerful partial evaluation as suggested in [31], would not be sufficient if the target architecture happens to be different from the host language target. What is needed is essentially a variant of staged metaprogramming, where the embedded “object” program can be analyzed and manipulated by a “meta” program that is part of the embedding infrastructure.

However, any DSL will also contain generic parts, some of which will be host language constructs such as function definitions, conditionals or loops. These must be lifted into the AST representation as well.

This ability to selectively make constructs ‘liftable’ (including their compilation) such that they can be part of (compiled) DSL programs while maintaining expressiveness, safety and effort is an essential characteristic of virtualizable languages.

Safety

Type checking is still provided by the host language

Tied to the semantics requirement, which is error checking is possible as well

Can hide the internals of the DSL optimizer from the DSL user (or optimizations become unsound)

Safety means that the embedded implementation is not allowed to loosen guarantees about program behavior. In particular, host-language operations that are not part of the embedded language’s specification must not be available to embedded programs.

There are two obstacles to maintaining safety. The first is to embed a typed object language into a typed meta language. This could be solved using a sufficiently powerful type system that supports an equivalent of GADTs [62, 54] or dependent
types [52]. The second problem is that with a plain AST-like representation, DSL programs can get access to parts of their own structure. This is unsafe in general and also potentially renders optimizations unsound. Fortunately, there is a technique known as finally tagless[11] or polymorphic embedding[29] that is able to solve both problems at once by abstracting over the actual representation used.

The combination of lightweight modular staging and polymorphic embedding provides a path to virtualize Scala and actually maintains all four of the criteria listed in the definition of language virtualization.

4.1.3 In Practice: Scala-Virtualized

These discussions resulted in Scala-Virtualized, a branch of the Scala compiler implemented and maintained (as of this writing) by Tiark Rompf and Adriaan Moors.

Talk about what is possible to overload now:

- have a table of virtualized built-ins
- talk about how they can be overridden
- talk about virtualized new (row types)

Talk about Philipp Haller’s work on source context, reference implicit discussion in section 3.1

4.1.4 What is Still Missing?

Not all the requirements need to be fully met before one can start building interesting DSLs. However the few that are essential is supporting higher-ordered functions and the ability to overload original language constructs. This is essentially what Scala-Virtualized provides us with.

The rest can be supplemented via a software infrastructure that fills in these gaps (i.e. LMS + Delite). We cover LMS in the next section.
4.2 Lightweight Modular Staging

Scala-Virtualized added some required capabilities to support high-performance DSLs. It allowed us to overload conditionals and other built-in constructs so that the DSL can control all aspects of the language. However, these changes did not address the ability of DSLs to reason about the whole program or how to generate code that is targetted at platforms other than those directly supported by the host’s virtual machine. This piece of the puzzle is handled by a technique developed by Tiark Rompf called lightweight-modular staging (LMS) [58]. This technique is an essential building block that Delite uses to simplify the work of implementing a high-performance DSL. To this foundational library, Delite adds features that support implementing the portable parallelism in a DSL. This section will introduces the features of the LMS library and how they can be used to optimize and retarget programs that are implemented using an embedded DSL.

4.2.1 REPresenting Code

We illustrate our approach of virtualization through lightweight modular staging and polymorphic embedding by means of the following very simple linear algebra example.

```scala
trait TestMatrix { this: MatrixArith =>
  //requires mixing-in a MatrixArith implementation
  //when instantiating TestMatrix
  def example(a: Matrix, b: Matrix,
               c: Matrix, d: Matrix) = {
    val x = a*b + a*c
    val y = a*c + a*d
    println(x+y)
  }
}
```

The embedded DSL program consists of the `example` method in the trait `TestMatrix`. It makes use of a type `Matrix` which needs to be defined in trait `MatrixArith`. The clause

```
this: MatrixArith =>
```
in the first line of the example is a self-type annotation [50]; it declares the type of this to be of type MatrixArith, instead of just TestMatrix, which it would be if no annotation was given. The annotation has two consequences: First, all MatrixArith definitions are available in the type of the environment containing the example method, so this effectively constitutes an embedding of the DSL program given in example into the definitions provided by MatrixArith. Second, any concrete instantiation of TestMatrix needs to mix-in a concrete subclass of the MatrixArith trait, but it is not specified which subclass. This means that concrete DSL programs can be combined with arbitrary embeddings by choosing the right mix-in.

Using lightweight staging we can reason about the high-level matrix operations in this example and reduce the number of matrix multiplications from four to a single multiplication. Optimizing matrix operations is one of the classic examples of the use of C++ expression templates [73, 72] and is used by many systems such as Blitz++ [74], A++ [57, 56], and others. We do not have to change the program at all, but just the way of defining Matrix.

Here is the definition of matrix operations in MatrixArith:

```scala
trait MatrixArith {
  type Rep[T]
  type InternalMatrix
  type Matrix = Rep[InternalMatrix]

  // allows infix(+,*) notation for Matrix
  implicit def matArith(x: Matrix) = new {
    def +(y: Matrix) = plus(x,y)
    def *(y: Matrix) = times(x,y)
  }
  def plus(x: Matrix, y: Matrix): Matrix
  def times(x: Matrix, y: Matrix): Matrix
}
```

There is nothing in the definition of MatrixArith apart from the bare interface. The definition Rep[T] postulates the existence of a type constructor Rep, which we take to range over possible representations of DSL expressions. In the staged interpretation, an expression of type Rep[T] represents a way to compute a value of type T.
The definition of \texttt{InternalMatrix} postulates the existence of some internal matrix implementation, and the definition \texttt{Matrix = Rep[InternalMatrix]} denotes that \texttt{Matrix} is the staged representation of this not further characterized internal matrix type. The remaining statements define what operations are available on expressions of type \texttt{Matrix}.

Since we have not defined a concrete representation, we say that the example code, as well as the definitions of matrix arithmetic operations, are polymorphic in the chosen representation, and hence, we have polymorphically embedded [29] the language of matrix arithmetic operations into the host language Scala. We also note that the embedding is tagless [11], i.e. resolution of overloaded operations is based on static types and does not require dispatch on runtime values. If the representation is abstract, in what way does that help? The answer is that we gain considerable freedom in picking a concrete representation and, perhaps more importantly, that the chosen representation is hidden from the DSL program.

To implement the desired optimizations, we will use expression trees (more exactly, graphs with a tree-like interface), which form the basis of our common intermediate representation that we can use for most DSLs:
trait Expressions {
  // constants/symbols (atomic)
  abstract class Exp[T]
  case class Const[T](x: T) extends Exp[T]
  case class Sym[T](n: Int) extends Exp[T]

  // operations (composite, defined in subtraits)
  abstract class Op[T]

  // additional members for managing
  // encountered definitions
  def findOrCreateDefinition[T](op: Op[T]): Sym[T]
  implicit def toExp[T](d: Op[T]): Exp[T] = findOrCreateDefinition(d)

  object Def {
    // pattern-match on definitions
    def unapply[T](e: Exp[T]): Option[Op[T]] = ...
  }
}

This expression representation will do a number of useful bookkeeping tasks for us, among them automatic elimination of common sub-expressions and, more generally, preventing any expression from being generated twice (e.g. we would only need to compute $a*c$ once in our example). The implementation of this bookkeeping is in method `findOrCreateDefinition`, which can be overridden by the DSL designer to further customize the building of the AST. Now we pick $Rep[T] = Exp[T]$ and introduce suitable case classes to represent the different node types in our expression tree. We also have to provide a dummy implementation of `InternalMatrix`:
4.2.2 Generating Code

4.2.3 Optimizing Code

While we are able to eliminate redundant computation and thus optimize the example, the true power of using domain specific language is our ability to use domain knowledge to perform optimizations. In this case, we can use our knowledge of matrix operations to rewrite some of our expressions into more efficient forms. Implementing these rewritings is very simple using the framework we have developed so far. All we have to do is override the corresponding operations in one of the traits:

```scala
trait MatrixArithRepExpOpt extends MatrixArithRepExp {
  override def plus(x: Matrix, y: Matrix) = 
    (x, y) match {
      // (AB + AD) == A * (B + D)
      case (Def(Times(a, b)), Def(Times(c, d)))
        if (a == c) => Times(a, Plus(b, d))
      case _ => super.plus(x, y)
      // calls default plus() if no match
    }
}
```

Instantiating our example with
object MyMatrixApp extends TestMatrix
    with MatrixArithRepExpOpt

constructs an object that generates an optimized version of our example code. It automatically rewrites the sum of multiplications into a single multiplication of a sum of matrices:

\[
a \times (b + c + c + d)
\]

We assume the `println` operation to be overloaded such that it will compile and execute its argument if it is invoked with a staged expression.

The use of domain knowledge in this case yields a tremendous amount of reduction in required computation. This was achieved only using a library with the power of polymorphic embedding and staging, without having to change or create a custom compiler. Note that while we have only shown a simple mechanism for defining optimizations through transformation on operators, much more sophisticated analyses and optimizations are possible by iterating through the entire AST of the program as opposed to one node at time. Liszt, a DSL for mesh-based partial differential equations (PDEs), uses this full-program optimization approach to enable large-scale parallel execution.

### 4.2.4 The LMS Library

### 4.2.5 What is Still Missing?

[TODO: What have we not resolved at this level? ]

We showed how Scala-Virtualized allows DSLs to control all aspects of the host language by overloading all syntactic constructs in Scala. We also showed how LMS can provide a library with the ability to optimize and compile code to different targets. In the next chapter, we show how Delite builds on this foundation and further simplifies the task of creating high-performance DSLs that can target modern parallel and heterogeneous computing systems.
Chapter 5

DeliteComp: A Compiler-based Approach

5.1 General Architecture

The Delite Compiler Framework aims to greatly decrease the burden of developing a compiler for an implicitly parallel DSL, by providing facilities for lifting embedded DSL programs to an intermediate representation (IR), exposing and expressing parallelism, performing generic, parallel, and domain-specific analyses and optimizations, and generating heterogeneous parallel code that will be executed and managed by the Delite Runtime.

5.2 Three-level IR

The Delite Compiler Framework uses and extends a general-purpose compiler framework designed for embedding DSLs in Scala called Lightweight Modular Staging (LMS) [58]. LMS employs a form of meta-programming to construct a symbolic representation of a DSL program as it is executed. For DSLs built on top of LMS, the application code is actually a program generator and each program expression, such as if (c) a else b, constructs an IR node when the program is run (in this case IfThenElse(c,a,b)). We use abstract types and type inference to safely hide
Figure 5.1: Views of the DSL IR. DSL applications produce an IR upon execution. This IR is defined by the LMS framework with enough information to perform generic analyses and optimizations. The Delite Compiler Framework extends the IR to add parallelism information, and this view allows parallel optimizations and parallel code generation. The DSL extends the parallel IR to form a domain-specific IR, which allows for domain-specific optimizations.

The overall operation of the Delite Compiler Framework and Runtime is shown in Figure 5.2. The figure shows an example of OptiML [?], a machine learning DSL developed with our framework. OptiML provides Matrix / Vector / Graph data structures, domain-specific operations including linear algebra, and domain-specific control structures such as \textit{sum}, which is used in the code snippet in Figure 5.2. The
sum construct accumulates the result of the given block for every iteration (0 to m) and is implemented by extending the \texttt{DeliteOpMapReduce} parallel pattern of the framework.

To generate optimized executables from the high level representation of the DSL operations, the Delite compiler builds an intermediate representation (IR) of the application and applies various optimizations on the IR nodes. For example, since the two vector minus operations ($x(i) - mu0$) within the \texttt{sum} are redundant, common subexpression elimination removes the latter operation by reusing the former result. After building the IR, the code generators in the framework automatically emit computation kernels for both the CPU and the GPU. When all of the kernels of the application have been generated along with the Delite Execution Graph (DEG), which encodes the data and control dependencies of the kernels, the Delite Runtime starts analyzing the DEG to make scheduling decisions to generate execution plans. Necessary memory transfers and synchronization are added as the execution plan for each target is generated. Finally, the kernels from the compiler and the execution
plan from the runtime are compiled and linked together by target language compilers (e.g., Scala, Cuda) to generate an executable that runs on the system.

### 5.2.1 DSL IR

Author handles this, Optimizations mainly via re-write rules as we build the IR.

The DSL developer extends the Delite Compiler to create domain-specific IR nodes that extend the appropriate Delite op. It is through this simple mechanism that a DSL developer expresses how to map domain constructs onto existing parallel patterns. This highest-level view of the IR is unique for each DSL and allows for domain-specific analyses and optimizations. For example, OptiML views certain IR nodes as linear algebra operations, which allows it to use pattern matching to apply linear algebra simplification rules. These rewrites can eliminate redundant work (e.g., whenever $\text{Transpose}(\text{Transpose}(x))$ is encountered, it is rewritten to be simply $x$) as well as yield significantly more efficient implementations that are functionally equivalent to the original. As an example, consider the snippet of OptiML code for Gaussian Discriminant Analysis (GDA) shown in Listing 5.1. The OptiML compiler’s pattern matcher recognizes that a summation of outer products can be implemented much more efficiently as a single matrix multiplication $[\Sigma]$. Specifically, it recognizes

\[
\sum_{i=0}^n \vec{x}_i \ast \vec{y}_i \rightarrow \sum_{i=0}^n X(:,i) \ast Y(i,:) = X \ast Y.
\]

The transformed code allocates two matrices, populates them by performing the operations required to produce all of the inputs to the original outer product operation, and then performs the multiplication.
5.2.2 Parallel IR: Delite OPs

extends LMS IR

The Delite Compiler extends the generic IR to express parallelism within and among IR nodes. Task parallelism is discovered by tracking dependencies among nodes. This information is used by the Delite Runtime to schedule and execute the program correctly and efficiently.

IR definition nodes are extended to be a particular kind of Delite op. There are multiple op archetypes, each of which expresses a particular parallelism pattern. A Sequential op, for example, has no internal parallelism, while a Reduce op specifies the reduction of some collection via an associative operator, and can therefore be executed in parallel (as a tree-reduce). Delite ops currently expose multiple common data-parallel patterns with differing degrees of restrictiveness. Some require entirely disjoint accesses (e.g., Map and Zip), while others allow the DSL to specify the desired synchronization across shared state for each iteration (e.g., Foreach).

Most Delite data-parallel ops extend a common loop-based ancestor, the MultiLoop op. A MultiLoop iterates over a range and applies one or more functions to each index in the range. MultiLoop also has an optional final reduction stage of thread-local results to allow Reduce-based patterns to be expressed. Like Map and Zip, MultiLoop functions must have disjoint access. However, a MultiLoop may consume any number of inputs and produce any number of outputs and is the key abstraction that enables Delite to fuse data-parallel operations together. Delite will fuse together adjacent or producer-consumer MultiLoops that iterate over the same range and do not have cyclic dependencies, creating a single pipelined MultiLoop. By fusing a MultiLoop that produces a set of elements together with a MultiLoop that consumes the same set, potentially large intermediate data structures can be entirely eliminated. Since fusing ops can create new opportunities for further optimization, fusion is iterated (and previously discussed optimizations reapplied) until a fixed point is reached. In addition to allowing multiple data-parallel ops in a single loop, fusion also effectively creates optimized MapReduce and ZipReduce ops (as well as any other combination, e.g., MapReduceReduce). Since Delite ops internally extend MultiLoop, DSL authors can benefit from fusion even while using only the
simpler data parallel patterns.

Fusion can significantly improve the performance of applications by improving cache behavior and reducing the total number of memory accesses required. For example consider the OptiML code shown in Listing ???. The application performs multiple subsequent operations on the input in order to update the result. Fusing these operations into a single traversal over the input collection that generates all of the outputs at once without temporary buffer allocations can produce a significant performance improvement for large inputs.

5.2.3 Generic IR: LMS

The lowest-level view of the IR is centered around symbols and definitions. Unlike many compilers, where individual statements are fixed to basic blocks, which are connected in a control flow graph (CFG), we use a "sea of nodes" representation [?]. Nodes are only connected by their (input and control) dependencies but otherwise allowed to float freely. Nodes in the IR are represented as instances of Scala classes; dependencies are represented as fields in each class. This representation enables certain optimizations to be performed during IR construction. For example, when a side-effect free IR node is constructed, the framework first checks if a definition for the node already exists. If a definition does exist it is reused to perform global common subexpression elimination (CSE). Pattern matching optimizations are also applied during node construction. The DSL compiler can override the construction of an IR node to look for a sequence of operations and rewrite the entire sequence to a different IR node. This mechanism is easy to apply and can be used to implement optimizations such as constant folding and algebraic rewrites. Listing 5.2 shows an example of implementing a simple pattern matching optimization in OptiML.

Once the complete IR is built and all dependency information is available, transformations that require a global view of the program can take place and work towards a program schedule. Transformations that occur during scheduling include dead-code elimination, various code motion techniques (e.g., loop hoisting) and aggressive fusing of operations, in particular loops and traversals. During the course of these global
Listing 5.2: Implementing pattern matching optimizations

transformations, the sea of nodes graph is traversed and the result is an optimized program in block structure. An important point is that since the IR is composed of domain operations, all of the optimizations described here are performed at a coarser granularity (e.g., Matrix-Multiply) than in a typical compiler.

It is important to note that in a general-purpose environment, it can be difficult to guarantee the safety of many important optimizations. However, because DSLs naturally use a restricted programming model and domain knowledge is encoded in the operations, a DSL compiler can do a much better job at optimizing than a general-purpose compiler that has to err on the side of completeness. These restrictions are especially important for tackling side effects in DSL programs in order to generate correct parallel code.

In the absence of side effects, the only dependencies among nodes in the IR are input dependencies, which are readily encoded by references from each node instance to its input nodes. While Delite and OptiML favor a functional, side-effect free programming style, prohibiting any kind of side effect would be overly restrictive and not in line with the driving goal of offering pragmatic solutions. However, introducing side effects adds control-, output-, and anti-dependencies that must be detected by the compiler to determine which optimizations can be safely performed. Dependency analysis is significantly complicated if mutable data can be aliased, i.e., a write to one variable may effect the contents of another variable. The key to fine-grained
dependency information is to prove that two variables must never alias, which, in general, is hard to do. If separation cannot be ensured, a dependency must be reported. Tracking side effects in an overly conservative manner falsely eliminates both task-level parallelism and other optimization opportunities.

The approach adopted by Delite is to restrict side-effects to a more manageable level. Delite caters to a programming model where the majority of operations is side-effect free and objects start out as immutable. At any point in the program, however, a modified copy of an immutable object can be obtained. Mutable objects can be modified in-place using side-effecting operations and turned back into immutable objects, again by creating a copy. A future version of Delite might even remove the actual data copies under the hood, based on the results of liveness analysis. The important aspect is that aliasing (and deep sharing) between mutable objects is prohibited.

DSL developers explicitly designate effectful operations and specify which of the inputs are mutated and/or whether the operation has a global effect (e.g., `println`). In addition, developers can specify for each kind of IR node which of its inputs are only read and which may be aliased by the object the operation returns (the conservative default being that any input may be read or aliased). This information is used by the dependency analysis to serialize reads of anything that may alias one or more mutable objects with the writes to those objects. The target of a write, however, is always known unambiguously and no aliasing is allowed.

5.2.4 Kernel Generation

The final stage of compilation is code generation. The DSL can extend one or more code generators, which are modular objects that translate IR nodes to an implementation in a lower level language. The LMS framework provides the basic mechanisms for traversing the IR and invoking the code generation method on each node. It also provides generator implementations for host language operations. On top of that, the Delite Compiler Framework supplies generator implementations for all Delite ops. Due to the ops’ deterministic access patterns and restricted semantics, Delite
is able to generate safe parallel code for CMPs and GPUs without performing complex dependency analyses. The DSL developer can also choose to override the code generation for an individual target (e.g., Cuda [48]) to provide a hand-optimized implementation or utilize an existing library (e.g. CUBLAS, CUFFT). We currently have implemented code generators for Scala, C++, and Cuda, which allow us to leverage their existing compilers to perform further low-level optimizations.

The Delite Compiler Framework adds a new code generator which generates a representation of the application as an execution graph of Delite ops with executable kernels. The design supports control flow nodes and nested graphs, exposing parallelism within a given loop or branch. For every Delite op, the Delite generator emits an entry in the graph containing the op’s dependencies. It then invokes the other available generators (Scala, Cuda, etc.) for each op, generating multiple device-specific implementations of each op kernel. For example, if a particular operation may be well-suited to GPU execution, the framework will emit both a CPU-executable variant of the op as well as a GPU-executable variant of the op. The runtime is then able to select which variant to actually execute. Since it is not always possible to emit a given kernel for all targets, each op in the graph is only required to have at least one kernel variant. By emitting this machine-agnostic execution graph of the application along with multiple kernel variants, we are able to defer hardware specific decisions to the runtime and therefore run the application efficiently on a variety of different machines. This mechanism also allows the DSL to transparently expand its set of supported architectures as new hardware becomes available. Once Delite supports code generation and runtime facilities for the new hardware, existing DSL application code can automatically leverage this support by simply recompiling.

5.2.5 Static Optimizations

In addition to the LMS optimizations we can do loop fusing and so forth.

I should integrate this into the other sections.
5.3 Delite Runtime Overview

The Delite Runtime provides services required by DSLs to execute implicitly parallel programs, such as scheduling, data management, and synchronization, and optimizes execution for the particular machine.

5.3.1 Scheduling

The runtime takes as input the execution graph generated by the Delite Compiler, along with the kernels and any additional necessary code generated by the Delite Compiler, such as DSL data structures. The execution graph is a machine-agnostic description of the inherent parallelism within the application that enumerates all the ops in the program along with their static dependencies and supported target(s). The runtime schedules the application at walk-time \[ ? \], combining the static knowledge of the application behavior provided by the execution graph with a description of the current machine, i.e., the number of CPU cores, number of GPUs, etc. (see Figure 5.3). The scheduler traverses all of the nested graphs in the execution graph file and produces partial schedules for blocks of the application that are statically determinable. The partial schedules are dispatched dynamically during execution as the branch directions are resolved. The runtime scheduler currently utilizes a clustering algorithm that prefers scheduling each op on the same resource as one of its inputs. If an op has no dependencies it is scheduled on the next available resource. This algorithm attempts to minimize communication among ops and makes device decisions based on kernel and hardware availability. Data-parallel ops selected for CMP execution are split into a number of chunks (determined by resource availability) and then scheduled across multiple CPU resources.

5.3.2 Schedule compilation

In order to avoid the overheads associated with dynamically interpreting the execution graph, the runtime generates an executable for each hardware resource that invokes the kernels assigned to that resource according to the partial schedules. Since the
compiler is machine-agnostic, the runtime is responsible for generating an implementation of each data-parallel op that is specialized to the number of processors chosen by the schedule. For example, a Reduce op only has its reduction function generated by the compiler, and the runtime generates a tree-reduction implementation with the tree height specialized to the number of processors chosen to perform the reduction.

The generated code enforces the schedule by synchronizing kernel inputs and outputs across resources. The synchronization is implemented by transferring data through lock-based one-place buffers. This code generation allows for a distributed program at runtime (no master coordination thread is required) and also allows for multiple optimizations that minimize run-time overhead. For example, kernels scheduled on the same hardware resource with no communication between them are fused to execute back-to-back. All synchronization in the application is generated at this time and only when necessary (kernel outputs that do not escape a single hardware resource require no synchronization). So in the simplest case of targeting a traditional uniprocessor, the final executable code will not invoke any synchronization
primitives (e.g., locks). The runtime also injects data transfers when the communicating resources reside in separate address spaces. When shared memory is available, it simply passes the necessary pointers.

5.3.3 Execution

The current implementation of the Delite Runtime is written in Scala and generates Scala code for each CPU thread and Cuda code to host each GPU. This environment allows it to support the execution of Scala kernels, C++ kernels, and Cuda kernels that are generated by the Delite compiler (using JNI as a bridge). The runtime spawns a JVM thread for each CPU resource assigned to a kernel, and also spawns a single CPU host thread per Cuda-compliant GPU.

The GPU host thread performs the work of launching kernels on the GPU device and transferring data between main memory and the device memory. For efficiency, it allows the address spaces to become out-of-sync by default, and only performs data transfers when the schedule requires them. Delite also provides memory management for the GPU. Before each Cuda kernel is launched, any memory on the device it will require is allocated and registered. The runtime uses the execution graph and schedule to perform liveness analysis for each input and output of GPU ops to determine the earliest time during execution at which it can be freed. By default, the runtime attempts to keep the host thread running ahead as much as possible by performing asynchronous memory transfers and kernel launches. When this causes memory pressure, however, the runtime uses the results of the liveness analysis to wait for enough data to become dead, free it, and then perform the new allocations. This analysis can be very useful due to the limited memory available in current GPU devices.

5.4 Compilers vs. Libraries

As a simpler alternative to constructing a framework for building DSL compilers that target heterogeneous hardware, one could also create a framework for domain-specific libraries. In previous work [13] we presented such a framework along with an earlier
version of the OptiML DSL. This DSL could also target heterogeneous processing
elements transparently from a single application source with no explicit parallelism
and achieve performance competitive with MATLAB. These original versions of Delite
and OptiML were implemented as pure libraries in Scala (with the OptiML library
extending the Delite library).

5.4.1 Static optimizations and code generation

By introducing compilation Delite DSLs gain several key benefits that are crucial to
achieving high performance for certain applications. First of all, we add the ability
to perform static optimizations, which includes generic optimizations provided by the
Delite framework as well as domain-specific ones provided by the DSL, as discussed
in Section ?? . With a library-based approach optimizations can only be performed
dynamically.

In addition, adding code generation support can greatly improve the efficiency of
the final executables by eliminating all the DSL abstractions and layers of indirec-
tion within the generated code, leaving only type-specialized, straight-line blocks of
instructions and first-order control flow that target compilers can optimize heavily.
Code generating from an IR also makes targeting hardware other than that supported
by the DSL’s hosting language much more tractable. A common solution for libraries
is to rely on the host language’s compiler to perform code generation for the CPU and
manually provide native binaries targeting other hardware using the host language’s
foreign function interface. In our previous work we attempted to somewhat ease this
burden on the DSL author for GPUs by writing a compiler plug-in that generated
Cuda equivalents of Scala anonymous functions that had disjoint data accesses (i.e.,
maps). By building an IR, however, Delite is able to handle Cuda code generation
seamlessly for both DSL and user-supplied functions, as well as perform static opti-
mizations on the generated kernels that are only reasonable on GPU architectures.
These code generators are also easily extensible to new target languages and architec-
tures, making the execution target(s) of Delite DSLs truly independent of the DSL
hosting language.
5.4.2 Runtime optimizations

It is also important to note that many of Delite’s runtime features are contingent on full program static analyses, which are made possible by the compiler statically generating the execution graph of the application. Delite can make scheduling decisions and specialize the execution at walk-time, thereby incurring significantly less run-time overhead. Full program analysis is also essential for Delite’s ability to manage GPU memory intelligently, as discussed in Section 5.3.3. A library-based system can also obtain an execution graph of the application by dynamically deferring the execution of each operation and building up the graph at run-time. We employed such a deferral strategy in our previous work, but were unable to defer past control flow, thereby creating “windows” of the application that could be executed at a time. These windows, however, were not sufficient to allow us to intelligently free GPU memory. We instead treated the GPU main memory as a software-managed cache of the CPU main memory, which was subject to undesirable evictions and could not always handle application datasets that severely pressured the GPU memory’s capacity.

We investigate the benefits of code generation for the GPU (Section ??), static optimizations (Section ??), and runtime optimizations (Section ??) not possible in our previous work in our experiments.

For a high performance DSL to target heterogeneous parallel systems, its IR should have at least the following three major characteristics:

• It should be able to accommodate traditional compiler optimizations on DSL operations and data types

• It should expose common parallel patterns for structured parallelism

• It should encode enough domain information to allow implementation flexibility and domain-specific optimizations
Figure 5.4: Multi-view of IR nodes in the Delite Compilation Framework. For example, the matrix addition operation $M_1 = M_2 + M_3$ is represented as a MatrixPlus IR node, which extends the DeliteOpZipWith IR node which again extends the Definition IR node. The generic IR view is used for traditional compiler optimizations, the parallel IR view is used for exposing parallel patterns and loop fusing optimizations, and the Domain-specific IR view is used for domain-specific optimizations.
5.4.3 Building an Intermediate Representation (IR)

To incorporate all the aforementioned requirements, we propose a multi-view representation of the IR as depicted in Figure 5.4. A single IR node can be viewed from three different perspectives which provide different optimizations and code generation strategies. We built the Delite Compiler Framework, which is a reusable compiler infrastructure for developing DSLs, using the concept of a multi-view IR.

**Generic IR:** The most basic view of an IR node is a symbol and its definition, which is similar to a node in the flow graph of a traditional compiler framework. Therefore, we can apply all the well-known static optimizations at this level. The primary difference is that our representation has a coarser granularity because each node is a DSL operation rather than an individual instruction, and this often leads to better optimization results. For example, the common subexpression elimination (CSE) can be applied to the vector operations \( (x(i) - mu0, x(i) - mu1) \) as shown in Figure 5.2 instead of just to scalar operations. Currently applied optimizations include CSE, constant propagation, dead code elimination, and code motion.

**Parallel IR:** A generic IR node can be characterized by its parallel execution pattern. At this level of view, the Delite Compiler Framework provides a finite number of common structured parallel execution patterns in the form of DeliteOp IR nodes. Examples include DeliteOpMap which encodes disjoint element access patterns without ordering constraints, and DeliteOpForeach which allows a DSL-defined consistency model for overlapping elements. The DeliteOpSequential IR node is for the pattern that is not parallelizable. Since certain parallel patterns share a common notion of loops, multiple loop patterns can be fused into a single loop. The parallel IR optimizer iterates over all of the IR nodes of loop types (e.g., DeliteOpMap, DeliteOpZipwith, etc.), and fuses those with the same number of iterations into a single loop. This optimization removes unnecessary memory allocations and also improves cache behavior by eliminating multiple passes over data, which is especially useful for memory-bound applications.

**Domain-specific (DS) IR:** Since the parallel IR does not encode domain-specific information, there is another viewpoint for semantic information of the operation. This enables domain-specific optimizations such as linear algebra simplification. The
transformation rules are simply described by pattern matching on DS IR nodes, and the optimizer replaces the matched nodes with a simpler set of nodes. Examples on matrix operations are $(A^t)^t = A$, and $A * B + A * C = A * (B + C)$. Since the DSL developer has expertise in the execution patterns of each DS IR node, the DSL developer extends the appropriate Delite parallel IR node. In this way parallel execution patterns are abstracted away from DSL users.

This multi-view IR greatly simplifies the process of developing a new DSL since the generic IR and the parallel IR can be re-used by all DSLs, and therefore DSL developers only need to design a DS IR for each operation as an extension. In other words, DSL developers are only exposed to a high-level parallel instruction set (the parallel IR nodes) and the implementation details of each pattern on multiple targets are automatically managed by the Delite Compiler Framework.

To build the IR from a DSL application, the Delite Compiler Framework uses a technique called Lightweight Modular Staging (LMS) [58]. As the application starts executing within the framework, each operation is lifted to a symbolic representation to form an IR node rather than actually being executed. The IR nodes track all dependencies among one another and the various optimizations mentioned above are applied. After building the machine-independent IR, the Delite Compiler Framework starts the code generation phase to target heterogeneous parallel hardware.

### 5.4.4 Heterogeneous Target Code Generation

Generating a single binary executable for the application at compile time limits the portability of the application and requires runtime and hardware systems to rediscover dependency information in order to make machine-specific scheduling decisions. The Delite Compiler Framework defers such decisions by generating kernels for each IR node in multiple target programming models as well as the Delite Execution Graph describing the dependencies among kernels. Currently supported targets are Scala [49], C++, and CUDA.
**Delite Execution Graph (DEG)**

The Delite generator is the main code generator that controls multiple target generators. It first schedules IR nodes to form kernels in the execution graph, and iterates over the list of available target generators to generate corresponding target code for the kernel. It may not be possible to generate the kernel for all targets, but the kernel generation will succeed as long as at least one target succeeds. The fact that each IR node has multiple viewpoints means that they can also be generated in different ways for each view. For example, a matrix addition kernel could be generated in the domain-specific view written by the DSL developer, but it also can be generated in the parallel view since the operation is of type `DeliteOpZipWith`. Since the Delite Compiler Framework provides parallel implementations for `DeliteOp` s, DSL developers do not have to provide code generators when they extend one of the parallel IR nodes. When DSL developers already have an efficient implementation of the kernel (e.g., BLAS libraries for matrix multiplication), they can generate calls to the external library using `DeliteOpExternal`.

**GPU code generation**

GPU code generation requires additional work since the programming model has more constraints compared to the Scala and C++ targets. One major issue is memory allocation. Since dynamic memory allocation within the kernel is either not possible or not practical for performance in GPU programming models, all device memory allocations within the kernel are pre-allocated by the Delite Runtime before launching the kernel. This is enabled by the CUDA generator collecting the memory requirement information and passing it to the runtime through a metadata field in the DEG. In addition, since the GPU resides in a separate address space, input/output transfer functions are also generated so that the Delite Runtime can manage data communication. Kernel configuration information (the dimensionality and the size of each dimension) also needs to be generated by the CUDA generator.
Variants

When multiple data parallel operations are nested, various parallelization strategies exist. In a simple case, a DeliteOpMap op within a DeliteOpMap can parallelize the outer loop or the inner loop or both. Therefore, the Delite Compiler Framework generates a data parallel operation in both a sequential version and a parallel version to provide flexible parallelization options when they are nested. This feature is especially useful for the CUDA target generator to improve the coverage of GPU kernels, since parallelizing the outer loop is not always possible for GPU due to the memory allocation requirements of the kernel. In those cases, the outer loop is serialized and only the inner loop is parallelized as a GPU kernel.

Target-specific Optimizations

While machine-independent optimizations are applied when building the IR, machine-specific optimizations are applied during the code generation phase. For example, the memory access patterns that enable better bandwidth utilization may not always be the same on the CPU and the GPU. Consider a data-parallel operation on each row of a matrix stored in a row-major format. In the case of the CPU where each core has its own private cache, assigning each row to each core naturally exploits spatial cache locality and prevents false sharing. However, the GPU prefers the opposite access pattern where each thread accesses each column, because the memory controller can coalesce requests from adjacent threads into a single transfer. Therefore the CUDA generator emits code that uses a transposed matrix with inverted indices for efficient GPU execution. In addition, to exploit SIMD units for data-parallel operations on the CPU, we currently generate source code that can be vectorized by the target compiler. It would also be straightforward to generate explicit SIMD instructions (e.g., SSE, AVX).
5.5 Runtime

DSLs targeting heterogeneous parallelism require a runtime to manage application execution. The work done at this phase of execution includes generating a great deal of "plumbing" code focused on managing parallel execution on a specific parallel architecture. The implementation can be difficult to get right, both in terms of correctness and efficiency, but is common across DSLs. We therefore built a heterogeneous parallel runtime to provide these shared services for all Delite DSLs.

5.5.1 Scheduling the Delite Execution Graph (DEG)

The Delite Runtime combines the machine-agnostic DEG generated by the framework with the specification of the current machine (e.g., number of CPUs, number of GPUs, etc.) to schedule the application across the available hardware resources. The Delite Runtime schedules the application before beginning execution using the static knowledge provided in the DEG. Since branch directions are still unknown, the Delite Runtime generates a partial schedule for every straight-line path in the application and resolves how to execute those schedules during execution. The scheduling algorithm attempts to minimize communication among kernels by scheduling dependent kernels on the same hardware resource and makes device decisions based on kernel and hardware availability. Sequential kernels are scheduled on a single resource while data-parallel kernels selected for CPU execution are split by the scheduler to execute on multiple hardware resources simultaneously. Since the best strategy for parallelizing and synchronizing these data-parallel chunks is not known until after scheduling, the runtime is responsible for generating the decomposition. In the case of a Reduce kernel, for example, the framework’s code generator emits the reduction function and the runtime generates a tree-reduction implementation that is specialized to the number of processors selected to perform the reduction.
5.5.2 Generating Execution Plans for Each Hardware Resource

Dynamically dispatching kernels into a thread pool can have very high overheads. However, the knowledge provided by the DEG and static schedule of the application is sufficient to generate and compile an executable (execution plan) for each hardware resource. Each executable launches the kernels and performs the necessary synchronization for its resource according to the partial schedules. The combination of generating custom executables for the chosen schedule and delaying the injection of synchronization code until after scheduling allows for multiple optimizations in the compiled schedule that minimize runtime overhead. For example, data that does not escape a given resource does not require any synchronization. This synchronization code is customized to the underlying memory model between the communicating resources. When shared memory is available, the implementation simply passes the necessary pointers, and when the resources reside in separate address spaces it performs the necessary data transfers. Minimizing runtime overhead by eliminating unnecessary synchronization and removing the central kernel dispatch bottleneck enables applications to scale with much less work per kernel.

5.5.3 Managing Execution on Heterogeneous Parallel Hardware

Executing on heterogeneous hardware introduces new and difficult challenges compared to traditional uniprocessor or even multi-core systems. The introduction of multiple address spaces requires expensive data transfers that should be minimized. The Delite Runtime achieves this through detailed kernel dependency information provided by the DEG. The graph specifies which inputs a kernel will simply read and which it will mutate. This information combined with the schedule allows the runtime to determine at any given time during the execution if the version of an input data structure in a given address space is currently nonexistent, valid, or old.

Managing the memory in each of these address spaces is also critical. The Delite Runtime currently utilizes the JVM to perform memory management for all CPU
kernels, but GPUs have no such facilities. In addition, all memory used by a GPU kernel must be allocated prior to launching the kernel. In order to deal with these issues, the Delite Runtime pre-allocates all the data structures for a given GPU kernel by using the allocation information supplied by the framework’s GPU code generator. The runtime also performs liveness analysis using the schedule to determine the earliest point at which each kernel’s inputs and outputs are no longer needed by the GPU. By default the GPU host thread attempts to run ahead as much as possible, but when this creates memory pressure it uses the liveness information to wait until enough data becomes dead, free it, and continue executing.
Chapter 6

Conclusions and Future Work

While the techniques we presented were applied in the context of embedded DSLs, the bulk of the insights and techniques are also applicable to external DSLs.

With the increasing dominance of heterogeneous parallel systems, applications must be able to leverage parallelism to improve performance. To enable average application developers to exploit parallelism, a mass market parallel programming model should shield these developers from parallel programming complexity. To achieve this, we proposed a domain-specific approach to parallel programming that provides application developers with familiar, high-level semantics while still delivering high performance and scalability through implicit task and data parallelism. DSLs also allow domain-specific knowledge to be leveraged to optimize program execution and data decomposition. Finally, DSL methods are a convenient abstraction for targeting heterogeneous platforms since they can be translated to different target processing nodes.

As an example of this approach, we introduced OptiML, a DSL for machine learning. OptiML is built using Delite, a framework and runtime that simplifies developing implicitly parallel DSLs that target heterogeneous platforms. We demonstrated how domain knowledge can be used to extract parallelism and to optimize application code. We presented results showing that OptiML can outperform explicitly parallelized MATLAB on a set of common machine learning applications. Using a single
version of application source code, running on a combination of CMP and GPU resources, OptiML exhibits robust speedups and scalability up to 59x on 128 threads. OptiML compares favorably to MATLAB, achieving average (geometric mean) performance improvements of 3.4x on 8 cores and 5.1x on GPU (MATLAB + Jacket).

As computing systems become increasingly parallel and heterogeneous, application programmers are being forced to learn multiple disparate programming models and consider more low-level hardware details in order to achieve high performance. We propose using domain-specific languages to provide a higher level of abstraction that is capable of producing high performance code without negatively impacting programmer productivity. We presented the Delite Compilation Framework and Runtime system for creating heterogeneous parallel DSLs using an example DSL for machine learning called OptiML. OptiML and other DSLs can leverage Delite to achieve high performance with significantly less effort than building a stand-alone compiler and runtime from scratch. Finally, we presented results comparing the performance of several machine learning applications written in OptiML and running on Delite to multiple MATLAB and C++ implementations.

6.1 Contributions

6.2 Availability

Fostering Community: A final challenge is to create a community around this shared infrastructure so that others can benefit and contribute their ideas and domain-expertise. We have taken the first steps by making all of our artifacts open and available\footnote{https://github.com/stanford-ppl}.

6.3 Future Work

Looking forward, we envision that DSLs will continue to play a larger role in the world of high performance, highly productive computing. To get there, several important
areas remain to be addressed:

Supporting Multiple DSLs: Although we laid the foundations here, more case studies are needed to show that our framework is indeed useful in cutting down the time and cost of creating DSLs. These case studies are underway; we have been developing many DSLs using Delite and we are finding there is significant opportunities for reuse among different DSLs.

DSL Extensibility: Related to supporting multiple DSLs, one approach to reuse is the ability to extend an existing DSL with new functionality. For example, many DSLs may want to provide some linear algebra data types and operations wrapped in domain-specific constructs. It should not be necessary for each such DSL to re-implement those linear algebra components. There should be a way to extend a base DSL (e.g. linear algebra) and add new constructs.

DSL Interoperability: Multiple DSLs implemented in a common framework present a new opportunity for tighter interoperability than has been traditionally possible with stand-alone DSLs. We are exploring ways of using multiple DSLs within a single application and using the host language to communicate between DSLs.

Abstracting Data Structures: Abstracting code alone is not sufficient. There also needs to be a way to generate different versions of DSL data structures targeted at different underlying runtimes and platforms. Data abstraction that encodes semantics about data structures in the IR allows data structures to be analyzed and reasoned about at a deeper level than what is currently possible. There is some exciting related work in this area[?]. For example, one can not only do array of structs (AoS) to struct of arrays (SoA) transaformation, but can also eliminate a set of the unused arrays (e.g. fields) if the computation only accesses a subset of the struct’s fields.

Abstracting DSL Program Analysis and Transformation: Existing work has shown how to define generic analysis and transformation mechanisms for external DSLs. In particular, the Stratego framework[?] has been used to express many general and domain-specific analysis in the context of external DSLs that aren’t particularly designed for high-performance. However, there has been much less exploration of similar frameworks for embedded DSLs. In order to generate high performance code, it is critical to develop abstractions that make it easier for DSL authors to define
analyses and transformations.

**DSL Agglomeration:** Although we are exploring several independent DSLs at this time, there are likely to be clusters of DSLs that share functionality. As more re-use and common abstractions are identified, it is important to exploit this commonality and allow functionality to be expressed using common syntactic idioms.

**DSL tooling and IDEs:** DSLs’ high level semantic knowledge of their applications can be leveraged to provide much more productive and intuitive debugging experiences. However, because of the amount of effort required in developing independent tool-chains for new languages, there hasn’t been a lot of progress this area. We believe that with a common framework and hosting language the burden of developing such a tool-chain greatly decreases while the incentives increase, since the same tool-chain can be reused across DSLs. We are currently working on a common debugging and profiling platform for Delite DSLs that attacks novel problems, such as tracking correctness and performance from high-level application source code to its corresponding generated low-level platform-specific codes.

**Host Languages for DSLs:** We have made some progress in more powerful hosting language by virtualizing Scala. Virtualized Scala is an example for retro-fitting an existing language to make it more DSL embedding friendly. It is an open question, however, what a language designed from scratch to host modern high-performance domain-specific language would look like; further research is needed in this area. Scala continues to add features that allow for more powerful DSL embeddings. As of the time of this writing a macro system is planned for Scala 2.10 in support of compile-time metaprogramming which enables different (and possibly more expressive) ways of embedding of high-performance DSLs.

### 6.4 Conclusion

More conclusions....

The ability to exploit domain knowledge to pursue high performance and productivity make DSLs an ideal platform for attacking the heterogeneous parallel programming problem.
While the techniques we presented were applied in the context of embedded DSLs, the bulk of the insights and techniques are also applicable to external DSLs.
Chapter 7

Background and Related Work

Collect all the related work from our various papers here and expand some more

7.1 Parallel Programming Languages

7.2 Heterogeneous Programming

7.3 Parallel Programming Models

7.4 Domain-specific Languages and Optimizations

General intro to domain specific languages

Conventional libraries are constructed as a collection of data structures and functions with fixed functionality. For example, a machine learning library would include a Matrix and Vector data structure

7.5 Embedded DSLs

talk about pros and cons of embedded DSLs. Talked about what people have done in this field
7.6 Compiling Embedded DSLs for performance

Talk about telescoping languages and all the work that took place in the FP community.

7.7 Related Work

There is a long history of research into parallel and concurrent programming. Parallel programming is now standard practice for scientific computations and in specialized application areas such as weather or climate forecasting.

Parallel programming codes are usually procedural/imperative with explicit segmentation between processors. Communication is done by specialized libraries such as MPI or, in shared-memory environments, parallelism is made explicit via compiler directives such as in OpenMP. Interesting alternatives such as SISAL [20], SaC [61] or NESL [7] have been tried but were either ahead of their time or too specialized to be adopted by a large community. The same holds for data-parallel Haskell [36], which is built around nested array parallelism similar to NESL. The explicit message passing paradigm of MPI is not without critique; some argue that a model based on send and receive primitives is too low-level and that communication should be represented in a more structured manner, e.g. using collective operations [24]. Google’s MapReduce framework is essentially such a model [17].

Recently, the high-performance programming language designs Chapel [15], X10 [60] and Fortress [37] have emerged from a DARPA initiative. These languages combine sophisticated type systems with explicit control of location and concurrency. They are targeted primarily at scientific applications for supercomputers. It remains to be seen to what degree they can be applied to general programming on a variety of heterogeneous hardware. We believe that some of the innovations of these languages (regions, configuration variables) can be generalized and made more powerful in the staged compilation setting that we propose.

When performance is paramount and the sheer number of possible combinations makes manual specialization intractable, program generation is an approach that is
often used. Instead of building a multitude of specialized implementations, a program
generator is built that, given the desired parameters as input, outputs the corre-
sponding specialized program. A number of high-performance programming libraries are
built in such a way, for example ATLAS[77] (linear algebra), FFTW[23] (discrete
fourier transform), and Spiral[55] (general linear transformations). However, good
program generators still take a huge effort to built and often, the resulting generator
implementation will no longer resemble the original algorithm.

The ideas underlying language virtualization are both very old and quite new. Recent related work includes the notion of universal languages [75] and relating hard-
ware virtualization criteria with partial evaluation [22]. Several important aspects of
language virtualization have been with us since the invention of Lisp[65] more than
50 years ago. The Lisp model of “code as data”, supported by macros, makes it
very easy to define and process embedded domain-specific languages. A common
viewpoint is that Lisp is not so much a programming language, but rather a way to
express language abstractions. In the words of Alan Kay: “Lisp isn’t a language, it’s
a building material”.

On the other hand, language embeddings in Lisp can be too seamless in that
they do not distinguish between the embedded DSL and the hosting framework.
Embedded DSL programs can observe the way their embeddings work and can access
fairly arbitrary parts of the host environment. These limitations can be overcome
in principle, given enough effort. A hosting environment could statically analyze
embedded domain-specific code for safety violations. However, such analyses are
non-trivial tasks, as they basically subsume implementations of static type checkers.

A common approach to DSL embedding is to assemble an abstract syntax tree
(AST) representation of embedded programs. However, even in statically typed set-
tings such as LINQ[10] or earlier work on compiling embedded languages [39, 19] the
type of the AST is publicly available. An embedded language might use this fact to
manipulate its own representation, thus undermining its virtualization.

Earlier work on optimizing domain languages through compilation of AST-like
embeddings is described by Leijen et al.[39] and Elliott et al.[19].

Elliot’s paper in particular is about compiling DSLs. Unlike Lisp macros, the
interpreter approach distinguishes between embedded language elements and host language elements by their types. Embedded code is an abstract syntax tree, which is manipulated by host code. The type of embedded code is publicly available. An embedded language might use this fact to manipulate its own representation, thus undermining its virtualization.

Tighter encapsulation is provided by staging (or multi-stage programming). Staging shares with language virtualization the idea that DSL programs are assembled and run as part of the host program execution. But it provides no control on the choice of representation of the language embeddings. The usual approach, taken e.g. by MetaOCaml[25], is to use different kinds of syntactic brackets to delineate staged expressions in a DSL, i.e., those that will be part of the generated program from the host program. Staged expressions can have holes, also marked by special syntax, into which the result of evaluating the contained generator stage expression will be placed. Finally, staged expressions can be run, which will cause them to be assembled as program source code, run through the compiler, with the resulting object code being dynamically loaded and executed. In essence, staging as implemented in MetaOCaml is similar to macros using quote/unquote/eval in Lisp, but with a static type system that ensures well-formedness and type safety for the generated code at the time the multi-stage program is compiled.

A closely related approach to code generation is template expansion, as implemented in C++[72] or Template Haskell[64]. The main difference to multi-staged programming is that all template expansion is done at compile-time. In contrast to staging, some template expansion mechanisms have a concept of user-definable rewriting rules which enable a limited form of domain-specific optimizations. However, the target of the compilation of embedded languages is always the same as the host language’s.

A different approach, described as pure embedding [31], is often used for simplicity. While structurally similar to the AST interpreter approach, no explicit data representation of the embedded program is created. Instead, the embedded DSL is just a collection of library functions and data types.
A generalized approach to language embedding which is referred to as finally tagless or polymorphic embedding was introduced by Carette et al. \cite{11} and taken up by Hofer et al. \cite{29}. The former focus on the basic mechanism of removing interpretive overhead while the latter stress modularity and the ability to abstract over and compose semantic aspects. We show in this paper how to use polymorphic embeddings and lightweight modular staging \cite{59} to optimize parallel domain specific languages.

Delite builds upon a variety of previously published work in domain-specific languages and parallel programming:

**Domain-specific languages and optimizations:** A good starting point for those interested in domain-specific languages is an annotated bibliography by Deursen et al. \cite{71}. Mernik et al. \cite{45} propose a pattern-based framework to help with deciding whether or not to invest in developing a domain-specific language and how to go about doing so. We adopt DSLs mainly for the AVOPT pattern: domain specific analysis, verification, optimization, parallelization, and transformation. Frameworks for developing domain-specific languages have also been proposed\cite{21, 38}. These frameworks mostly help authors develop external DSLs and provide tools that help in the construction and transformation of an abstract syntax tree. We adopt an approach similar to that presented by Hudak\cite{30} and embed our DSLs directly into a host language. Previous work has also shown the benefit of using domain knowledge to enhance the performance of applications: Menon et al demonstrate the benefits of applying high level transformations to MATLAB code\cite{44} and show performance gains in both interpreted and compiled code. Guyver et al\cite{26} presents a mechanism for annotating library methods with domain-specific knowledge which yields significant improvement in performance. CodeBoost\cite{4} allows for user-defined rules that are used to transform the program using domain knowledge. In contrast, Delite allows DSL developers to encode dynamic domain-specific optimizations.

**Heterogeneous programming:** There have been proposals to help programmers target heterogeneous systems. Some proposals such as EXOCHI\cite{76} and OpenCL\cite{69} provide abstractions that allow the programmer to explicitly manage and target any available accelerator. This approach reduces the ad hoc nature of directly using
vendor drivers and APIs for each device. Merge[40] builds on top of EXOCHI by providing a framework for associating a kernel variant with a particular accelerator and shifting the responsibility of selecting the appropriate kernel to the runtime. These proposals however, are still too low level for a mass market programming model. Nevertheless, we believe that such proposals are extremely useful when used by the runtime to dispatch work to the different heterogeneous devices. Harmony[18], a recent proposal, reasons about the whole program by building a data dependency graph and then scheduling independent kernels to run in parallel. Harmony can also automatically select from different variants of each deferred kernel. However, Harmony does not apply any domain-specific transformations or perform automatic data-decomposition of the program kernels.

**Data-parallel programming models and libraries:** This approach to parallel programming hides the complexity of the underlying system by only exposing the programmer to a data-parallel API. Examples of this approach include RapidMind[41], PeakStream[53], and Accelerator[68]. These APIs mostly consist of vector and array primitives which map well to SIMD-based accelerators and work well for many algorithms. However, they are inadequate for parallel algorithms that are irregular and require a more general task-based decomposition. Delite is informed by these previous proposals in the way it handles data-parallel ops. Dryad[33], a distributed execution engine for very coarse-grained data-parallel operations, targets clusters. Delite is similar in that it translates an application to an execution graph prior to mapping it to particular system configuration. Unlike Delite, Dryad applications need to explicitly construct this execution graph. DryadLINQ[34] attempts to overcome the complexity of execution graph construction and allows a programmer to write LINQ[42] programs that are automatically translated to a Dryad execution graph. In this case, LINQ could be considered as the DSL and Dryad as the runtime. Delite differentiates itself by adding facilities for authoring implicitly parallel DSLs, targets finer grained on-chip parallelism and exploits task parallelism.

**Parallel programming languages:** Parallel programming languages focus on two main categories: explicit and implicit parallelism. Explicitly parallel languages rely on the programmer to identify parallel work; notable examples include Parallel
Haskell[70], Cilk[8], X10[16] and Chapel[14]. Requiring programmers to explicitly parallelize their code may have an adverse effect on the productivity goal, and it is often difficult to achieve scalable performance using explicit constructs. Languages that support implicit parallelization often rely on data-parallel operations on parallel collections. These include NESL[6], High Performance Fortran[1], X10 and Chapel. One could also argue that stream programming languages such as Brook[9] and to some extent CUDA[48] are data-parallel languages with streams being synonymous to a parallel collection of records. OptiML provides the same facilities, but uncovers coarse-grained parallelism using domain knowledge and adds implicit task parallelism through Delite.

Delite builds upon a variety of previously published work in the areas of domain-specific languages, multi-stage compilation, and parallel programming.

**DSLs and optimizations:** 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[30]. 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[43]. Menon et al. apply high level transformations to MATLAB code, producing performance gains in both interpreted and compiled code[44]. Guyver et al. present a way to annotate library methods with domain-specific knowledge and show significant performance improvements[26]. 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.

**Multi-Stage compilation:** Many static metaprogramming techniques exist, including C++ templates[72] and Template Haskell[64]. Expression Templates[73] allow customizable generation, and TaskGraph[?] performs runtime code generation from C++. Telescoping languages[38] are efficient DSLs created from annotated component libraries. Designated multi-stage programming languages include MetaML[?] and MetaOCaml[?]. The Delite framework is built on top of the Lightweight Modular
Staging approach [58], inspired from the related work on embedding typed languages by Carette et al. [11] and Hofer et al. [?]. Libraries using domain-specific code generation and optimization include ATLAS [77] (linear algebra), FFTW [23] (discrete Fourier transform), and SPIRAL [55] (general linear transformations). Such program generators often require significant effort to create. The Delite framework and Lightweight Modular Staging aim to make such facilities easily accessible.

Heterogeneous programming: Some systems such as EXOCHI [76] and OpenCL [69] provide abstractions that allow the programmer to explicitly manage and target any available accelerator, eliminating the need to use vendor APIs for each device. Merge [40] builds on top of EXOCHI by allowing kernel variants to be associated with particular accelerators and using the runtime to select the appropriate kernel. Harmony [18] builds a data dependency graph of a program and then schedules independent kernels to run in parallel. Unlike Delite, it does not perform automatic data-decomposition or support domain-specific optimizations.

Data-parallel programming: Several programming models use a data-parallel API to hide the complexity of the underlying hardware. Copperhead [?] provides automatic Cuda code generation from a data-parallel subset of Python. FlumeJava [?] is a Java library targeting Google’s MapReduce [17] that optimizes the data-flow graph to create an efficient pipeline of MapReduce operations. Intel’s Array Building Blocks [?] provides managed execution of data parallel patterns across processor cores and is capable of targeting multiple architectures (e.g., different vector units) from a single application source. Concurrent Collections (CnC) [?] is a model that shares some similarities with the Delite task graph. Computation steps and scheduling are treated separately in CnC, whereas Delite produces optimized kernels using scheduling information. Dryad [33] executes very coarse-grained data-parallel operations over clusters and uses an explicitly constructed execution graph to map the application to a specific system configuration. DryadLINQ [34] automatically translates LINQ [42] programs to a Dryad execution graph. Here LINQ could be considered the DSL and Dryad the runtime. In contrast, Delite additionally provides facilities for developing new implicitly parallel DSLs, targets finer-grained parallelism, and exploits both task and data parallelism.
Parallel programming languages: Recent parallel programming languages include Chapel [14], Fortress [37], and X10 [16]. These languages employ explicit control over locations and concurrency and are targeted primarily at scientific applications for supercomputers. In contrast, the Delite runtime manages locations and concurrency transparently. Implicit parallelism in languages is often based on data-parallel operations on parallel collections. Languages with this feature include Chapel, Data-Parallel Haskell [36], Fortress, High Performance Fortran [1], NESL [6], and X10. DSLs which utilize the Delite framework are able exploit implicit data parallelism as well as implicit task parallelism.
Bibliography


of the 17th international symposium on High performance distributed computing, pages 197–200, New York, NY, USA, 2008. ACM.


[27] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C. Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. Understanding sources of inefficiency in general-purpose chips. In


Hassan Chafi

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.

(Kunle Olukotun)  Principal 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.

(Pat Hanrahan)

Approved for the University Committee on Graduate Studies