A Domain-Specific Approach To Heterogeneous Parallelism

Hassan Chafi Arvind K. Sujeeth Kevin J. Brown
HyoukJoong Lee Anand R. Atreya Kunle Olukotun

Pervasive Parallelism Laboratory
Stanford University
{hchafi, asujeeth, kjbrown, hyouklee, aatreya, kunle}@stanford.edu

Abstract

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.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming – Parallel programming; D.3.4 [*Programming Languages*]: Processors – Code generation, Optimization, Run-time environments

General Terms Languages, Performance

Keywords Parallel Programming, Domain-Specific Languages, Dynamic Optimizations

1. Introduction

Current industry trends favor chip multiprocessors consisting of simpler cores[18, 29] as well as heterogeneous systems consisting of general-purpose processors, SIMD units and accelerator devices such as GPUs[3, 31]. 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 tradeoffs. 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.

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. It is essential to develop appropriate abstractions 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. This *mass market* parallel heterogeneous 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.

There has been a resurgence in research aimed at simplifying parallel programming [8] and delivering on these goals. This paper describes key elements of an ongoing effort to create a development environment that uses a domain-specific approach to solve the issues relating to heterogeneous parallelism. The components of this environment are shown in Figure 1. The environment consists of four main components: applications composed of multiple domain-specific languages (DSLs), DSLs embedded in the Scala programming language [28], a Scala-based framework that simplifies the parallelization of DSLs and a runtime for DSL parallelization and mapping to heterogeneous architectures.

A domain-specific approach to parallel programming can address all of the goals of a *mass market* parallel heterogeneous programming model. A domain-specific language is a computer programming language of restricted expressiveness focused on a particular domain[35]. DSLs are in widespread use in a variety of domains and are becoming more popular. Examples of widely used DSLs are TeX and LaTeX for typesetting academic papers, SQL

PPoPP'11, February 12–16, 2011, San Antonio, Texas, USA.
Copyright © 2011 ACM 978-1-4503-0119-0/11/02...$10.00

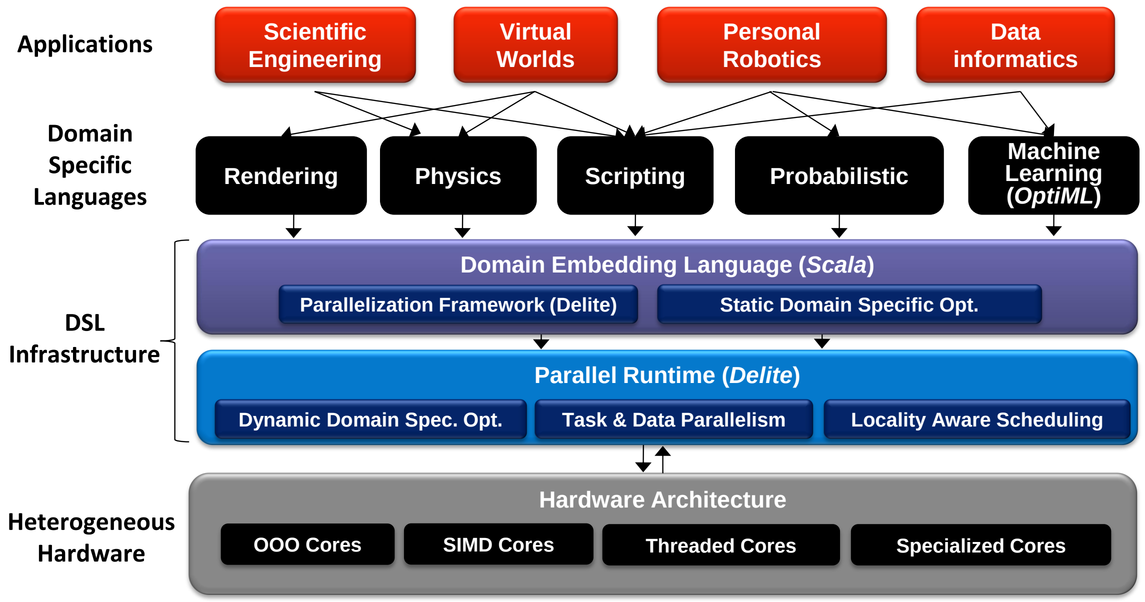


Figure 1: An environment for domain-specific programming of heterogeneous parallel architectures.

for database querying, Rails for web application development and VHDL for hardware design. OpenGL can also be viewed 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. The use of DSLs can provide significant gains in the productivity and creativity of application developers, the portability of applications, and application performance. We exploit this trend towards DSLs and propose an approach to parallel heterogeneous programming that hides the complexity of the underlying machine behind a collection of DSLs. A programmer using one or more of these DSLs writes her programs using domain-specific notation and constructs. The programs appear sequential and all parallelism and use of the heterogeneous machine resources is implicit. DSLs raise the level of abstraction and can provide a sequential model which satisfies the *productivity* goal.

An additional benefit of using a domain-specific approach is the ability to use domain knowledge to apply static and dynamic optimizations to a program written using a DSL. 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. DSLs on the other hand, with their expressive power and knowledge of the domain's data structures and algorithms make such optimizations feasible. This makes DSLs a good choice to deliver on our *performance* goal. In order to validate this approach, we are developing multiple DSLs that span a variety of domains. In this paper, we present OptiML, a DSL aimed at machine learning. We provide an overview of OptiML in Section 2.

Since interesting applications might leverage a variety of DSLs, it is critical to not only simplify the development of DSLs by creating a shared infrastructure, but also to allow these DSLs to interoperate. Our current approach is to embed these DSLs in a common embedding language. Scala, our choice for the embedding language, provides features that simplify this task [9, 16]. This approach should be applicable to any sufficiently expressive embedding language.

The ability to easily embed DSLs simplifies the task of a DSL developer. However, assistance in parallelizing and targeting heterogeneous resources is also needed. Delite, our framework and runtime for building and executing parallel DSLs provides facilities that allow DSL developers to easily parallelize their DSLs. Using Delite, a DSL developer implicitly exposes task level parallelism by enabling a run-ahead model, similar to recent proposals [13, 19], across each invocation of the DSL's operations. Delite also allows the developer to express data-level parallelism available within DSL operations. Using such a runtime allows us to deliver on our *portability and forward scalability* goal. We provide details of the Delite framework and runtime in Section 3. Our specific contributions are:

- We present OptiML, a DSL for machine learning, which provides implicitly parallel domain-specific abstractions. We show that such a DSL can be used to simplify programming heterogeneous parallel systems.
- We show how domain knowledge can aid in extracting implicit parallelism, and demonstrate the benefits of using domain-specific optimizations to further enhance the performance of some machine learning applications.
- We introduce Delite, a framework and runtime to simplify building and executing implicitly parallel DSLs. We show how we use Delite to implement and parallelize OptiML.
- We provide experimental results demonstrating the benefits of using DSLs as a means of simplifying heterogeneous parallel programming. We demonstrate that in most cases, OptiML running on Delite outperforms explicitly parallelized MATLAB

code on a set of common machine learning applications executing on a combination of CMP and GPU resources.

2. OptiML: A DSL for Machine Learning

In this section, we introduce a DSL written using Delite that will serve as a running example for the rest of this paper. OptiML 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.

Figure 2 shows a side-by-side comparison of a simple ML application written in OptiML and MATLAB. Since most machine learning users are already familiar with MATLAB, OptiML uses MATLAB-like syntax and compares favorably in terms of conciseness and productivity. OptiML inherits Scala's static type safety and type inference, as well as its tooling and IDE support. OptiML is optimized to perform well inside a JVM environment in a way that is transparent to the user; extreme care is taken to avoid boxing of primitives and patterns that reduce the effectiveness of JIT optimizations. Furthermore, the DSL includes native versions of heavyweight operations (e.g. matrix multiplication) that use JNI to call an underlying BLAS wrapper. In addition to fast sequential performance, Section 4 shows that OptiML can provide parallel performance comparable to or better than equivalent MATLAB implementations. While OptiML programs can achieve good performance with a simple imperative style, MATLAB programs usually require explicit vectorization or parallelization to achieve the best performance. This problem is compounded by the fact that usually either vectorization or parallelization must be chosen, and the trade-off between them is not obvious.

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.

The library provides implementations of implicitly parallel ML-specific data and control structures. OptiML programs operate on the high-level mutable types `Vector[T]` and `Matrix[T]`, regardless of their underlying implementation (array-based, views, sparse, etc.). OptiML also provides data types that encode semantic information, such as `TrainingSet`, `TestSet`, `Image`, `IndexVector`, etc. Most of these structures are wrappers around matrices or vectors, but their use allows the DSL to identify additional parallelization or optimization opportunities. For example, most applications operate on training sets in a streaming fashion, and training sets are usually very large. OptiML can exploit this information when deciding whether an operation is worth parallelizing or shipping to a heterogeneous device such as a GPU. These domain-informed decisions are communicated to the Delite runtime using the interfaces described in Section 3.3.2. Although not currently supported, future versions of OptiML will have direct support for irregular structures that occur in ML, such as graphs.

In addition to these data structures, OptiML provides concise and flexible control structures. OptiML's control structures hide implementation details and expose coarse-grained parallelism inside applications. Although most operations in OptiML can be executed in parallel, this often results in fine-grained parallelism that does not scale well (e.g. vector addition). These domain specific control abstractions allow intuitive application code that is already decomposed into scalable operations. OptiML can dynamically check the size of the operation (e.g. the number and type of elements being summed) to determine whether it should submit a sequential or parallel operation to Delite. A summary of supported control structures is provided in the pseudocode snippets of Listing 1.

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

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

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

// gradient descent: implemented sequentially using
// stochastic, or in parallel using batch g.d.
gradient(trainingSet, alpha, threshold) {
  <hypothesis function, takes a training sample and
   returns a scalar>  }
```

Listing 1: Pseudocode snippets demonstrating OptiML control structures.

The control structures listed here support efficient parallelism in our current set of example applications. Two of these structures, `sum{...}` and `(0::end){...}` employ restricted semantics that

```
// x : TrainingSet[Double]

val m = x.numRows
val n = x.numCols

val (y_zeros, y_ones, mu0_num, mu1_num) = sum(0,m) { i =>
  if (x.labels(i) == false) {
    (1, 0, x(i), 0)
  else {
    (0, 1, 0, x(i))
  }
}

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

val sigma = sum(0,m) { i =>
  if (x.labels(i) == false) {
    ((x(i)-mu0).trans).outer(x(i)-mu0)
  }
  else {
    ((x(i)-mu1).trans).outer(x(i)-mu1)
  }
}
```

(a) OptiML code

```
% 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
```

(b) MATLAB code

Figure 2: Side-by-side comparison of OptiML and MATLAB code for Gaussian Discriminant Analysis (GDA).

benefit parallelism; all accesses inside their closures must be disjoint (i.e. only access the *i*th element of any data structure). This allows them to be trivially parallelized using chunking and for locality to be preserved within the chunks. Because this corresponds to the great majority of ML use-cases, we have not found this restriction to be unduly constraining. A general-purpose language or compiler might provide constructs with similar restrictions (e.g. *parfor*), but cannot provide practical guidance on when to use them or require that they be used.

Finally, OptiML currently supports two dynamic, domain-specific optimizations[10]. We evaluate the benefit of these optimizations in Section 4. We briefly describe them below:

- **Relaxed Dependencies:** in some iterative applications, there is limited data parallelism. However, when the algorithm is robust to minor perturbations, dependencies can be relaxed to provide more task parallelism at the cost of a marginal loss in accuracy. OptiML supports a run-time configurable relaxation percentage. *untilconverged* is an example of a relaxable operation; if it is relaxed, some number of iterations are allowed to run in parallel and intentionally race. OptiML relaxes operations by not registering the dependency with Delite.
- **Best Effort Computations:** in the same vein as relaxed dependencies, it is not strictly necessary to execute every computation to achieve acceptable results in ML. To exploit this, OptiML provides data structures with "best effort" semantics. The semantics are defined in the DSL by various policies that the DSL user can choose from. This optimization can reduce total execution time and sequential overhead in some applications, resulting in better single-threaded performance as well as scaling.

We are developing a prototype version of OptiML using polymorphic embedding with Scala[9]. Polymorphic embedding allows a library to directly analyze and transform programs written using the library. This will remove the need for a compiler plugin and allow future versions of OptiML to perform static optimizations as well.

3. Delite: A Framework for Parallelizing DSLs

Constructing a new DSL is a difficult process in general. Delite significantly simplifies this process by providing a shared framework and runtime to implement and execute implicitly parallel DSLs. Each DSL that utilizes the Delite framework inherits the Delite execution model which provides the capability of executing on heterogeneous hardware.

3.1 A deferred execution model

Delite'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 3. The application thread executes sequentially (Figure 3a). 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 3b). The application is oblivious to the fact that computation is deferred and "runs-ahead" allowing more ops to be submitted. The submitted ops form a dynamic task graph (Figure 3c). Given a program's task graph, the runtime system is able to target a variety of parallel architectures automatically (Figure 3d). Independent parts of the task graph are scheduled to run in parallel and data movement is minimized with a

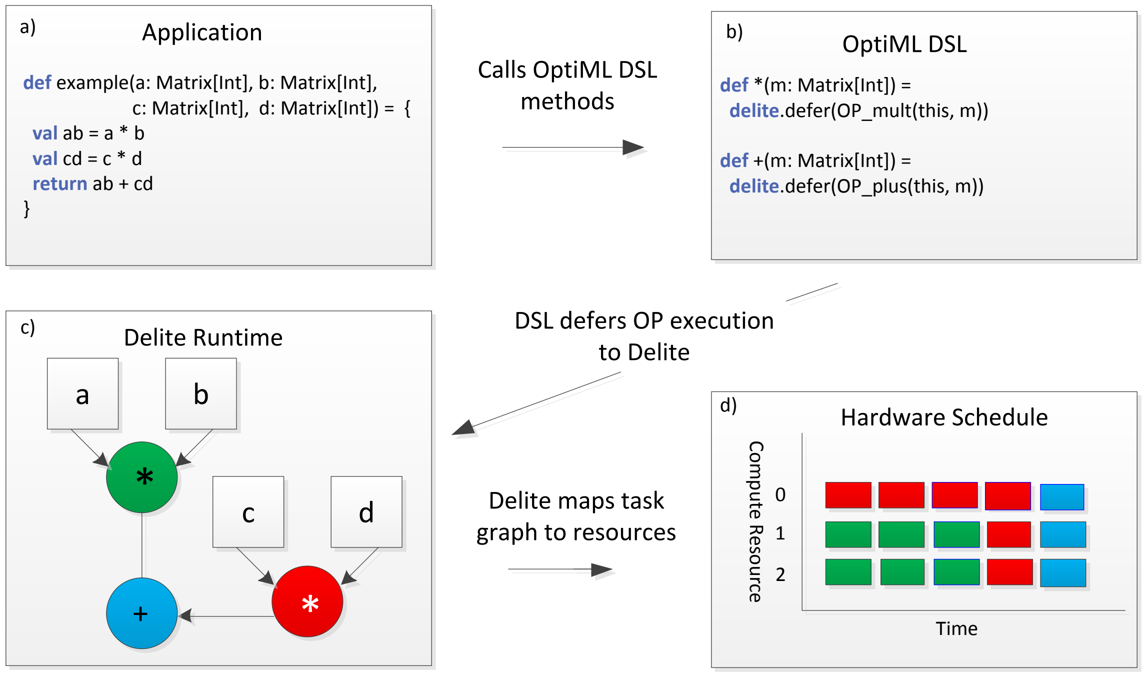

a)
Application
def example(a: Matrix[Int], b: Matrix[Int],
c: Matrix[Int], d: Matrix[Int]) = {
val ab = a * b
val cd = c * d
return ab + cd
}
Calls OptiML DSL methods
b)
OptiML DSL
def *(m: Matrix[Int]) =
delite.defer(OP_mult(this, m))
def +(m: Matrix[Int]) =
delite.defer(OP_plus(this, m))
DSL defers OP execution to Delite
c)
Delite Runtime
a
b
*
c
d
+
*
Delite maps task graph to resources
d)
Hardware Schedule
Compute Resource
0
1
2
Time

Figure 3: Delite application execution overview.

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.

Each Delite op returns a proxy, which unlike most implementations of futures, is *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. Figure 4 shows the Delite class diagram for the Vector object in the OptiML DSL. A Scala `trait` is similar to a Java interface. 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.

3.2 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.

3.2.1 Delite ops

When defining a Delite op, the DSL author specifies its dependencies and the return type of the proxy. Listing 2 shows how a simple op can be written to subtract two vectors. Its input arguments, `v1` and `v2` (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 3.2.2.

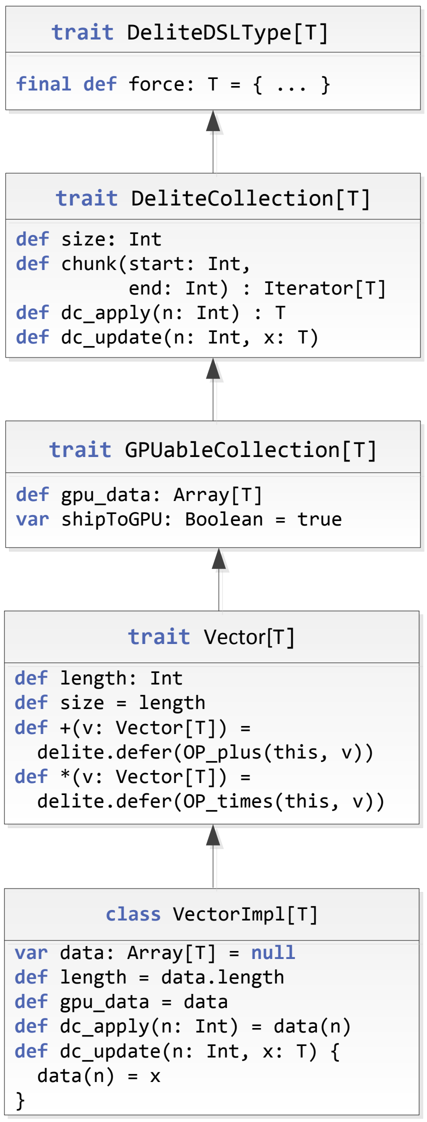
```
protected[optiml] case class OP_-[A]
  (v1: Vector[A], v2: Vector[A])
  extends DeliteOP_SingleTask[Vector[A]](v1,v2) {

  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 2: An example sequential op in the OptiML DSL.

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.

3.2.2 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` and `op_zipWith`. Each op operates on a `DeliteCollection`, which is a `trait` that each collection-based DSL type implements. This



```
trait DeliteDSLType[T]
final def force: T = { ... }
```

```
trait DeliteCollection[T]
def size: Int
def chunk(start: Int,
        end: Int) : Iterator[T]
def dc_apply(n: Int) : T
def dc_update(n: Int, x: T)
```

```
trait GPUableCollection[T]
def gpu_data: Array[T]
var shipToGPU: Boolean = true
```

```
trait Vector[T]
def length: Int
def size = length
def +(v: Vector[T]) =
  delite.defer(OP_plus(this, v))
def *(v: Vector[T]) =
  delite.defer(OP_times(this, v))
```

```
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 4: Class diagram of the OptiML Vector class.

```
protected[optiml] case class OP_-[A]
  (val collA: Vector[A], val collB: Vector[A],
  val out: Vector[A])
  extends DeliteOP_ZipWith2[A,A,A,Vector] {

  def func = (a,b) => a - b
}
```

Listing 3: An example data-parallel op in the OptiML DSL.

trait and its implementation for OptiML's Vector is shown in Figure 4. 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 shows how the same subtraction op from Listing 2 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. `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 *size* and *chunk* method for the collection. As the name implies, *size* is the total number of elements of the collection. *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 (*dc_apply* and *dc_update*) that the DSL author implements. In addition, primitive array-based collections can choose to implement the 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 3.3.2. 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.

3.2.3 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.

3.2.4 Assisting GPU code generation

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.

3.3 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.

3.3.1 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.

3.3.2 Supporting GPU execution

Unlike other environments such as MATLAB which require non-trivial 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 3.2.4. 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.

4. Experimental Analysis

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 2 on performance.

4.1 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 1 presents the applications used in our study. These applications all come from the machine learning domain.

4.2 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 5. The figure shows execution time normalized to the OptiML 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

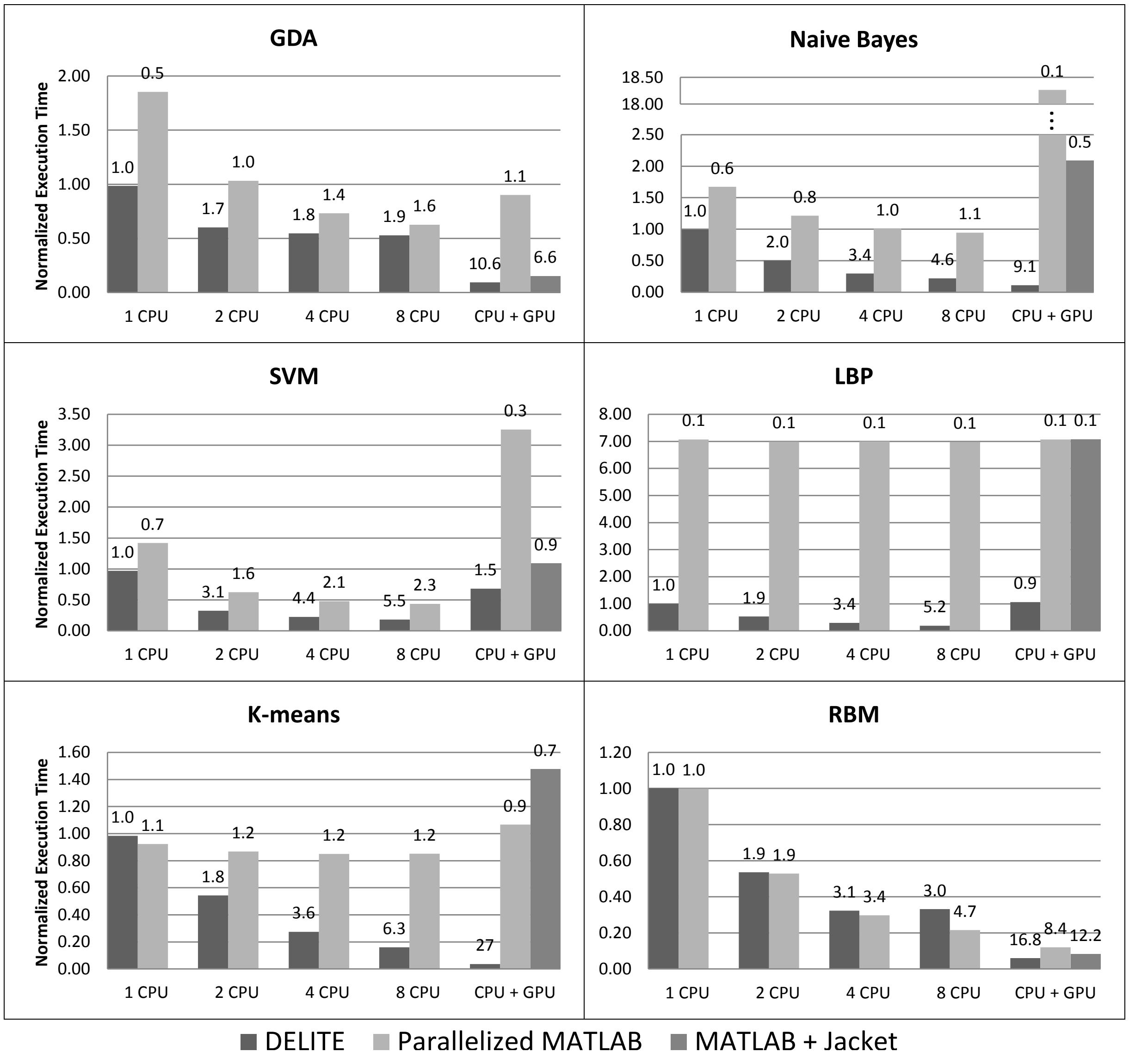


Figure 5: Execution time of the various versions of our applications normalized to OptiML running in sequential mode. Speedup numbers are reported on top of each bar.

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 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 *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 struc-

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

Table 1: Applications used for experimental analysis.

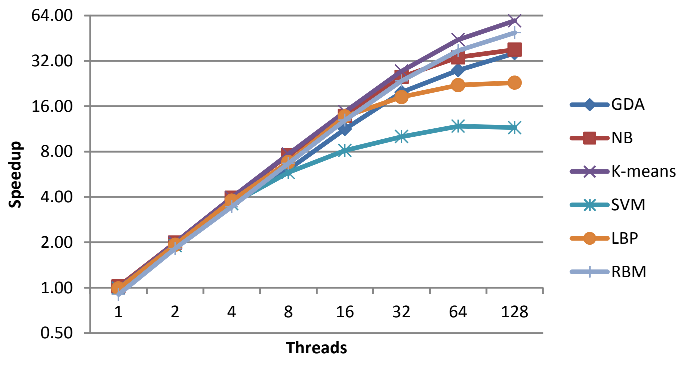


Figure 6: Scalability of our selected OptiML applications running in a highly threaded system. Speedup is measured relative to OptiML running in sequential mode.

tures 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.

4.3 Scalability study

Figure 6 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.

4.4 The benefit of domain-specific optimizations

In Section 2, we described two domain-specific optimizations that OptiML supports: best effort computation and relaxed dependencies. Figure 7 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 *untilconverged* 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.

5. Related Work

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.[35]. Mernik et al.[26] propose a pattern-based framework to help with deciding whether or not to invest in developing a domain-specific language and how to

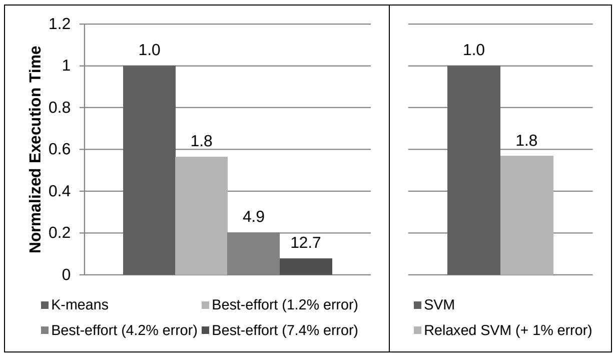


Figure 7: The impact of domain-specific optimizations on k-means and SVM at 8 threads. Execution time is normalized to the 8 thread version without optimization shown in Figure 5. Speedup numbers are reported on top of each bar.

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[14, 21]. 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[17] 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[25] and show performance gains in both interpreted and compiled code. Guyver et al[15] presents a mechanism for annotating library methods with domain-specific knowledge which yields significant improvement in performance. CodeBoost[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[36] and OpenCL[33] 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[22] 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[13], 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[23], PeakStream[30], and Accelerator[32]. 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[19], 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[20] attempts to overcome the complexity of execution graph construction and allows a programmer to write LINQ[24] 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[34], Cilk[6], X10[12] and Chapel[11]. 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[5], High Performance Fortran[1], X10 and Chapel. One could also argue that stream programming languages such as Brook[7] and to some extent CUDA[27] 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.

6. Conclusion

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).

7. Acknowledgments

This research was sponsored by the Department Of Energy (DOE) through Sandia order 942017, the Army under contract AHPCRC W911NF-07-2-0027-1 and the Defense Advanced Research Project Agency (DARPA) through Oracle order 630003198. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency, the Department of Energy, the Army or the U.S. Government. Additional funding provided by the Stanford Pervasive Parallelism Laboratory affiliates program which is supported by NVIDIA, Oracle/Sun, AMD, and NEC.

References

[1] High Performance Fortran. http://hpff.rice.edu/index.htm.

[2] AccelerEyes. Jacket. http://www.accelereyes.com/products/jacket.

[3] AMD. The Industry-Changing Impact of Accelerated Computing. Website. http://sites.amd.com/us/Documents/AMD_fusion_Whitepaper.pdf.

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

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

[6] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. In *PPOPP '95: Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 207–216, New York, NY, USA, 1995. ACM.

[7] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream computing on graphics hardware. *ACM TRANSACTIONS ON GRAPHICS*, 23:777–786, 2004.

[8] B. C. Catanzaro, A. Fox, K. Keutzer, D. Patterson, B.-Y. Su, M. Snir, K. Olukotun, P. Hanrahan, and H. Chafi. Ubiquitous parallel computing from Berkeley, Illinois, and Stanford. *IEEE Micro*, 30(2):41–55, 2010.

[9] H. Chafi, Z. DeVito, A. Moors, T. Rompf, A. Sujeeth, P. Hanrahan, M. Odersky, and K. Olukotun. Language virtualization for heterogeneous parallel computing. In *Onward!*, 2010.

[10] S. Chakradhar, A. Raghunathan, and J. Meng. Best-effort parallel execution framework for recognition and mining applications. In *Proc. of the 23rd Annual Int'l Symp. on Parallel and Distributed Processing (IPDPS'09)*, pages 1–12, 2009.

[11] B. Chamberlain, D. Callahan, and H. Zima. Parallel Programmability and the Chapel Language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, 2007.

[12] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40(10):519–538, 2005.

[13] G. F. Diamos and S. Yalamanchili. Harmony: an execution model and runtime for heterogeneous many core systems. In *HPDC '08: Proceedings of the 17th international symposium on High performance distributed computing*, pages 197–200, New York, NY, USA, 2008. ACM.

[14] R. E. Faith, L. S. Nyland, and J. F. Prins. Khepera: A system for rapid implementation of domain specific languages. In *In Proceedings USENIX Conference on Domain-Speci Languages*, pages 243–255, 1997.

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

[16] K. Havelund, M. Ingham, and D. Wagner. A case study in DSL development: An experiment with Python and Scala. In *The First Annual Scala Workshop at Scala Days 2010*, 2010.

[17] P. Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, page 196.

[18] Intel. From a Few Cores to Many: A Tera-scale Computing Research Review. Website. http://download.intel.com/research/platform/terascale/terascale_overview_paper.pdf.

[19] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 59–72, New York, NY, USA, 2007. ACM.

[20] M. Isard and Y. Yu. Distributed data-parallel computing using a high-level programming language. In *SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data*, pages 987–994, New York, NY, USA, 2009. ACM.

[21] K. Kennedy, B. Broom, A. Chauhan, R. Fowler, J. Garvin, C. Koelbel, C. McCosh, and J. Mellor-Crummey. Telescoping languages: A system for automatic generation of domain languages. *Proceedings of the IEEE*, 93(3):387–408, 2005. This provides a current overview of the entire Telescoping Languages Project.

[22] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng. Merge: a programming model for heterogeneous multi-core systems. In *ASPLOS '08*, New York, NY, USA, 2008. ACM.

[23] M. D. McCool, K. Wadleigh, B. Henderson, and H.-Y. Lin. Performance evaluation of GPUs using the RapidMind development platform. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 181, New York, NY, USA, 2006. ACM.

[24] E. Meijer, B. Beckman, and G. Bierman. LINQ: Reconciling object, relations and XML in the .NET framework. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pages 706–706, New York, NY, USA, 2006. ACM.

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

[26] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.

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

[28] M. Odersky. Scala. http://www.scala-lang.org.

[29] K. Olukotun, B. A. Nayfeh, L. Hammond, K. G. Wilson, and K. Chang. The case for a single-chip multiprocessor. In *ASPLOS '96*.

[30] PeakStream. The PeakStream platform: High productivity software development for multi-core processors. technical report, 2006.

[31] G. C. Sih and E. A. Lee. A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE Trans. Parallel Distrib. Syst.*, 4(2):175–187, 1993.

[32] D. Tarditi, S. Puri, and J. Oglesby. Accelerator: using data parallelism to program GPUs for general-purpose uses. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 325–335, New York, NY, USA, 2006. ACM.

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

[34] P. W. Trinder, H.-W. Loidl, and R. F. Pointon. Parallel and distributed Haskells. *J. Funct. Program.*, 12(5):469–510, 2002.

[35] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, 2000.

[36] P. H. Wang, J. D. Collins, G. N. Chinya, H. Jiang, X. Tian, M. Girkar, N. Y. Yang, G.-Y. Lueh, and H. Wang. Exochi: architecture and programming environment for a heterogeneous multi-core multithreaded system. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 156–166, New York, NY, USA, 2007. ACM.