2011 International Conference on Parallel Architectures and Compilation Techniques

A Heterogeneous Parallel Framework for Domain-Specific Languages

Kevin J. Brown* Arvind K. Sujeeth* HyoukJoong Lee* Tiark Rompf†
Hassan Chafi* Martin Odersky† Kunle Olukotun*
*Stanford University, Stanford, CA: {kjbrown, asujeeth, hyouklee, hchafi, kunle}@stanford.edu
†EPFL, Lausanne, Switzerland: {firstname.lastname}@epfl.ch

***Abstract*—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.**

***Keywords*-parallel programming; multicore processing; computer languages**

I. Introduction

Current microprocessor trends focus on larger numbers of simpler cores [1], [2] and include increasingly heterogeneous processing elements, such as SIMD units or a GPU [3]. 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. 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 best way of dividing work across the hardware is often affected by application variables such as dataset size, making it nontrivial to realize maximal performance improvement.

The 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. In order to alleviate these problems, it is essential to develop high level abstractions that allow programmers to develop high performance programs with high productivity.

Ideally a programming language should provide generality, high productivity, and produce high-performance binaries that take advantage of all the hardware resources available in a given platform. Unfortunately, no such language currently exists. A more tenable goal is to create a language that trades-off one desirable aspect to achieve the others. A compromise that has the potential to attain both productivity and performance is to use languages targeted to a specific application domain, so called domain-specific languages (DSLs) [4]. DSLs can provide a higher level of abstraction than general-purpose languages and can enable mappings between domain abstractions and efficient parallel implementations. In addition DSLs enable domain-specific static and dynamic optimizations that would not be possible with a general-purpose compiler and runtime system.

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. In previous work [5] we showed how such an internal DSL can target heterogeneous hardware. In this paper we utilize a more advanced method of embedding DSLs that allows the DSL to build and optimize an intermediate representation (IR) of the application.

Figure 1 illustrates our vision for constructing new implicitly parallel DSLs which automatically target heterogeneous hardware. 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 [6] that allows the DSL to participate in the back-end phases of compilation [7]; this approach is called *Lightweight Modular Staging* [8]. In this paper we 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 use-

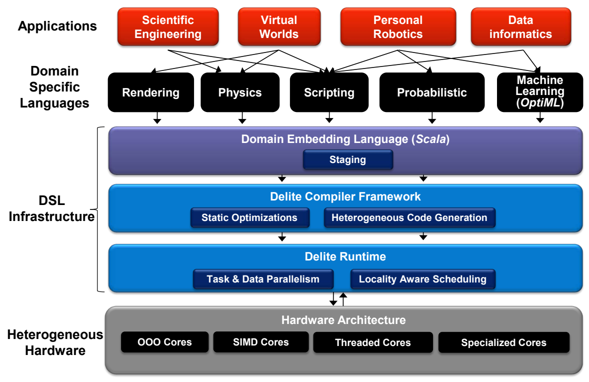

Fig. 1: An environment for domain-specific programming of heterogeneous parallel architectures.

ful 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. Our specific contributions are as follows:

- We present an end-to-end system for executing a program written in a domain-specific language on parallel heterogeneous hardware.
- We provide a framework for lifting embedded DSLs to an intermediate representation which can be optimized from multiple viewpoints.
- We utilize this framework to perform and orchestrate generic, parallel, and domain-specific optimizations in a single environment.
- We generate and execute a machine-agnostic graph of the application targeting multiple heterogeneous hardware devices.

In the rest of this paper we discuss the benefits of using DSLs for parallelism, present the Delite Compiler Framework and Runtime, and compare DSL compilation to embedded DSL libraries. We then present experimental results for both overall application performance and the effects of Delite's optimizations.

II. DSLs for Heterogeneous Parallelism

In this section we briefly illustrate the benefits of using DSLs for achieving both productivity and portable parallel performance in a heterogeneous environment. We will use OptiML [9], a DSL for machine learning, as a running example. We then address the common challenges faced when designing and building a new DSL targeted to heterogeneous parallelism.

A. *DSL productivity*

At the forefront of DSL design is the ability to exploit domain knowledge to provide constructs that express domain operations at a higher level of abstraction. As a consequence

```
val distances =
  Stream[Double](data.numRows, data.numRows) {
    (i,j) => dist(data(i), data(j))
  }
for (row <- distances.rows) {
  if(densities(row.index) == 0) {
    val neighbors = row find {_ < apprxWidth}
    densities(neighbors) = row count {_ < kernelWidth}
  }
}
```

Listing 1: Downsampling in OptiML

```
#pragma omp parallel for shared(densities)
for (size_t i=0; i<obs; i++) {
  if (densities[i] > 0)
    continue;
  // Keep track on observations we can approximate
  std::vector<size_t> apprxs;
  Data_t *point = &data[i*dim];
  Count_t c = 0;
  for (size_t j=0; j<obs; j++) {
    Dist_t d = distance(point, &data[j*dim], dim);
    if (d < apprx_width) {
      apprxs.push_back(j);
      c++;
    } else if (d < kernel_width) c++;
  }
  for (size_t j=0; j<apprxs.size(); j++)
    densities[apprxs[j]] = c;
  densities[i] = c;
}
```

Listing 2: Downsampling in C++

of working at this abstraction level much of the lower-level implementation details are provided by the DSL itself rather than the application programmer. This often results in a significant reduction in total number of lines of code as well as improved code readability compared to a general-purpose language.

As an example, consider the snippet of OptiML code shown in Listing 1, which shows the core of a downsampling application. In contrast to the C++ implementation shown in Listing 2, the OptiML version concisely expresses what should be accomplished rather than how it should be accomplished.

B. *Portable parallel performance*

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 the OptiML *sum* construct (shown in Listing 4). 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-reduce. 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.

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.

C. *Building DSLs*

DSLs have the potential to be a solution for heterogeneous parallelism, but this solution rests on the challenging task of building new DSLs targeting parallelism. 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.

To address the challenge of building DSLs for parallelism, we present the Delite Compiler Framework and Runtime as a means of dividing the required expertise across multiple systems developers. Delite uses DSL embedding and an extensible compilation framework to greatly reduce the effort in creating a DSL compiler, provides parallel patterns that the DSL developer can extend, performs heterogeneous code generation, and handles all the run-time details of executing the program correctly and efficiently on various hardware platforms. In short, Delite provides the expertise in parallelism and hardware. The DSL developer can then focus on being a domain expert, designing the language constructs and identifying the mapping between those domain constructs and the parallel patterns Delite provides. He or she must implement the data and control structures that inherit from Delite prototypes as well as add domain-specific optimization rules.

III. Compiler Framework

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.

A. *Compilation framework*

The Delite Compiler Framework uses and extends a general-purpose compiler framework designed for embedding DSLs in Scala called *Lightweight Modular Staging* (LMS) [8]. 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 the IR construction from the DSL user [10].

Through this mechanism the DSL compiler effectively reuses the front-end of the Scala compiler, and then takes over with the creation of the IR. Possible nodes in the IR are all constructs of the DSL or constructs the DSL developer chooses to inherit from Scala (e.g., If-Then-Else statements). The LMS framework provides all of the tools required for building the IR, performing analyses and optimizations, and generating code, which the DSL developer can then use and extend. Delite expands on this functionality by providing three primary views of the IR, namely the generic view, the parallel view, and the domain-specific view, as illustrated in Figure 2.

B. *Generic IR*

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 [11]. 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 3 shows an example of implementing a simple pattern matching optimization in OptiML.

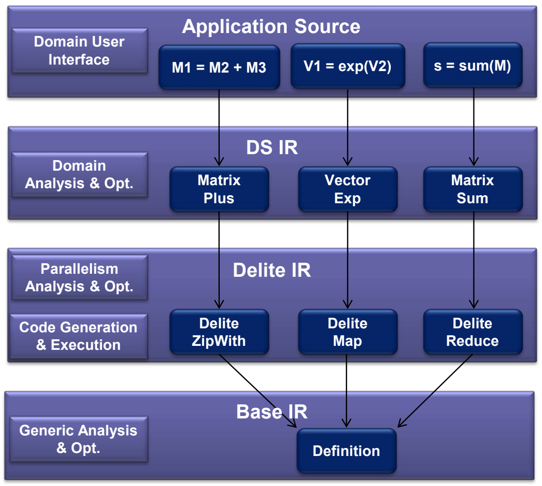

Fig. 2: Views of the DSL IR. DSL applications produce an IR upon execution. This IR is defined by the LMS framework with enough information to perform generic analyses and optimizations. The Delite Compiler Framework extends the IR to add parallelism information, and this view allows parallel optimizations and parallel code generation. The DSL extends the parallel IR to form a domain-specific IR, which allows for domain-specific optimizations.

```
override def matrix_plus[A:Manifest:Arith]
  (x: Exp[Matrix[A]], y: Exp[Matrix[A]]) =
    (x, y) match {
      // (AB + AD) == A(B + D)
      case (Def(MatrixTimes(a, b)),
            Def(MatrixTimes(c, d))) if (a == c) =>
              matrix_times(a, matrix_plus(b,d))
      // ...
      case _ => super.matrix_plus(x, y)
    }
```

Listing 3: Implementing pattern matching optimizations

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 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 mutable 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.

C. *Parallel 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 1. 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.

D. Domain-specific 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 `Transpose(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 4. The OptiML

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

Listing 4: The summation representing the bulk of computation in Gaussian Discriminant Analysis

compiler's pattern matcher recognizes that a summation of outer products can be implemented much more efficiently as a single matrix multiplication [9]. Specifically, it recognizes

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

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.

E. Heterogeneous code 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 [12]) 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.

IV. Heterogeneous Runtime

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.

A. *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 [13], 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 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.

B. *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

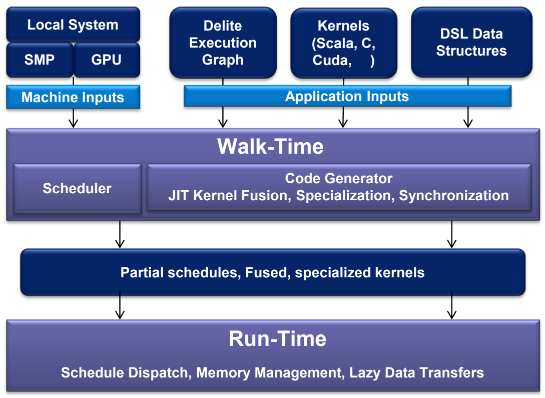
Local System
SMP
GPU
Delite Execution Graph
Kernels (Scala, C, Cuda,)
DSL Data Structures
Machine Inputs
Application Inputs
Walk-Time
Scheduler
Code Generator
JIT Kernel Fusion, Specialization, Synchronization
Partial schedules, Fused, specialized kernels
Run-Time
Schedule Dispatch, Memory Management, Lazy Data Transfers

Fig. 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.

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 runtime 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.

C. *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.

V. DSL Compilers vs. DSL 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 [5] 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).

A. 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 III. 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 indirection 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 optimizations on the generated kernels that are only reasonable on GPU architectures. These code generators are also easily extensible to new target languages and architectures, making the execution target(s) of Delite DSLs truly independent of the DSL hosting language.

B. 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 IV-C. 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 VI-C), static optimizations (Section VI-D), and runtime optimizations (Section VI-E) not possible in our previous work in our experiments.

VI. Experimental Analysis

In this section we present results for overall application performance for various machine learning applications written in OptiML (using Delite) and compare them to sequential C++ counterparts as well as multiple explicitly parallelized MATLAB versions. We also show benchmarks that provide insights into the achievable performance benefits from some of Delite's new optimizations, namely the benefits of GPU code generation compared to a library-based approach (launching pre-written kernels), static optimizations, and runtime schedule compilation.

A. Methodology

We compare the OptiML version of our applications to C++ implementations using the Armadillo linear algebra library [14] and multiple MATLAB implementations. We used MATLAB 7.11 [15] with its Parallel Computing Toolbox to write explicitly parallelized versions of each application for multi-core CPU results, as well as a different implementation that utilizes MATLAB's GPU computing support. Each of the four versions of the applications are algorithmically identical, but for the MATLAB versions, we made a reasonable effort to vectorize and parallelize the code. In situations where we were forced to choose between the two options, we chose the version that produced the fastest execution time at the highest thread count. For the GPU versions, we offloaded all computationally expensive operations to the GPU and attempted to make the best data locality choices when managing the memory transfers. A summary of our applications appears in Table I.

NAME	DESCRIPTION	INPUT SIZE
Gaussian Discriminant Analysis (GDA)	Generative learning algorithm for modeling the probability distribution of a set of data as a multivariate Gaussian	1,200 x 2,048 matrix
Naïve Bayes (NB)	Fast, low-work supervised learning algorithm for classification	50,000 x 1,448 matrix
k-means Clustering (k-means)	Unsupervised learning algorithm for finding similar clusters in a dataset	1,048,576 x 3 matrix
Support Vector Machine (SVM)	Optimal margin classifier, implemented using the Sequential Minimal Optimization (SMO) algorithm	400 x 1,448 matrix
Linear Regression (LR)	Unweighted and locally weighted linear regression	2,048 vectors
Restricted Boltzmann Machine (RBM)	Stochastic recurrent neural network, without connections between hidden units	2,000 hidden units 2,000 dimensions

TABLE I: Applications used for performance evaluations.

Our experiments were performed on a Dell Precision T7500n with two quad-core Xeon 2.67 GHz processors, 24GB of RAM, and an NVidia Tesla C2050. For the CPU we generated Scala code that was executed on Oracle's Java SE Runtime Environment 1.7.0 and the HotSpot 64-bit server VM with default options. For the GPU, we generated and executed Cuda v3.2. For each experiment we timed the computational part of the application, ignoring initialization steps. We ran each application (with initialization) ten times without interruption in order to warm up the JIT, and averaged the last five runs to smooth out fluctuations due to garbage collection and other variables.

B. Performance results

In Figure 4 we present the performance of OptiML, C++, and MATLAB from one to eight CPU threads, as well as with one GPU. Execution times are normalized to the single-threaded OptiML version of each application. The implicitly parallel OptiML code achieves single-threaded performance comparable to single-threaded C++ and outperforms the explicitly parallel MATLAB version in nearly all cases. For some of the C++ versions of the apps (GDA and Linear Regression), we observed that the total cost of allocating new matrix objects was very high, and therefore we optimized the memory usage in ways that prevent parallelizing the applications. Specifically we pre-allocated and re-used storage for temporary matrices generated within loops rather than having the functions return a newly allocated result for each call. This significantly reduced the execution time of the C++ versions of the applications, and we show these hand-optimized results in addition to the original results in Figure 4. The OptiML versions of the applications always perform a new allocation in order to allow parallelization.

For most of the CPU results, we were able to achieve superior performance results to MATLAB and comparable results to C++ using our analysis and code generation facilities, which produces code that is specialized, efficient, and contains minimal abstractions and levels of indirection. We wrote the C++ implementations single-threaded using the Armadillo library to perform all linear algebra operations.

The GPU performance numbers are dependent on the suitability of the application to the GPU, which is directly related to Delite's ability to generate Cuda kernels. It is clear from the large variance in the performance results that achieving maximum performance requires supporting heterogeneous systems and being able to choose the best resource for execution. Some applications, like GDA and RBM, achieved significant speedup by utilizing the GPU, while others achieved better performance with multi-core scaling.

For GDA, Naive Bayes, and k-means, we were able to achieve significantly better GPU performance results compared to MATLAB due to code generation of custom Cuda kernels. The MATLAB versions only launched kernels for individual language operations, which in these applications are much too fine-grained with little data-parallelism. Code generating user functions created fewer kernels and exposed much more data-parallelism per kernel for the GPU to exploit. In the cases of Naive Bayes and k-means this generation was a relatively straightforward translation of each function using the Delite ops created by the application. For Naive Bayes transferring the input data to the GPU was very expensive and consumed most of the execution time, allowing the CPU to perform much better overall. SVM uses an iterative convergence algorithm with a few expensive linear algebra operations that can be parallelized within each loop iteration. While this parallelism was sufficient for CPU speedups, the overhead of transferring data to and from the GPU every loop iteration resulted in a GPU execution of nearly identical time to the single-threaded CPU execution.

For GDA, the significant GPU speedup came from exploiting multiple GPU-specific hardware characteristics during code generation of the summation kernel shown in Listing 4. Since the summation function dynamically allocates multiple objects, the Cuda code generator implemented the summation loop serially and parallelized the inner data-parallel operations (which do not require dynamic allocations) in order to fit the Cuda programming model. It then used work duplication for the vector operations in order to localize all the work needed to compute a given element of the output matrix. Once this transformation was performed the code generator was able to keep all intermediate results for the entire summation in registers and eliminate all main memory accesses besides loading the inputs and storing the output by assigning each GPU thread to compute a single element of the output matrix.

For Naive Bayes and k-means we also achieved significantly better performance and scalability on the CPU. In the MATLAB version of k-means we were forced to choose between vectorization and parallelization. Vectorization produced the fastest execution time at 8 threads, which is why the MATLAB results do not scale. Both of these applications contain numerous small operations, which the OptiML compiler is able to inline and eliminate a significant amount of overhead, resulting in significantly faster kernels that also scale.

RBM is composed of many fine-grained linear algebra operations and matrix multiplications which MATLAB ships to BLAS, which is how the MATLAB version performs at

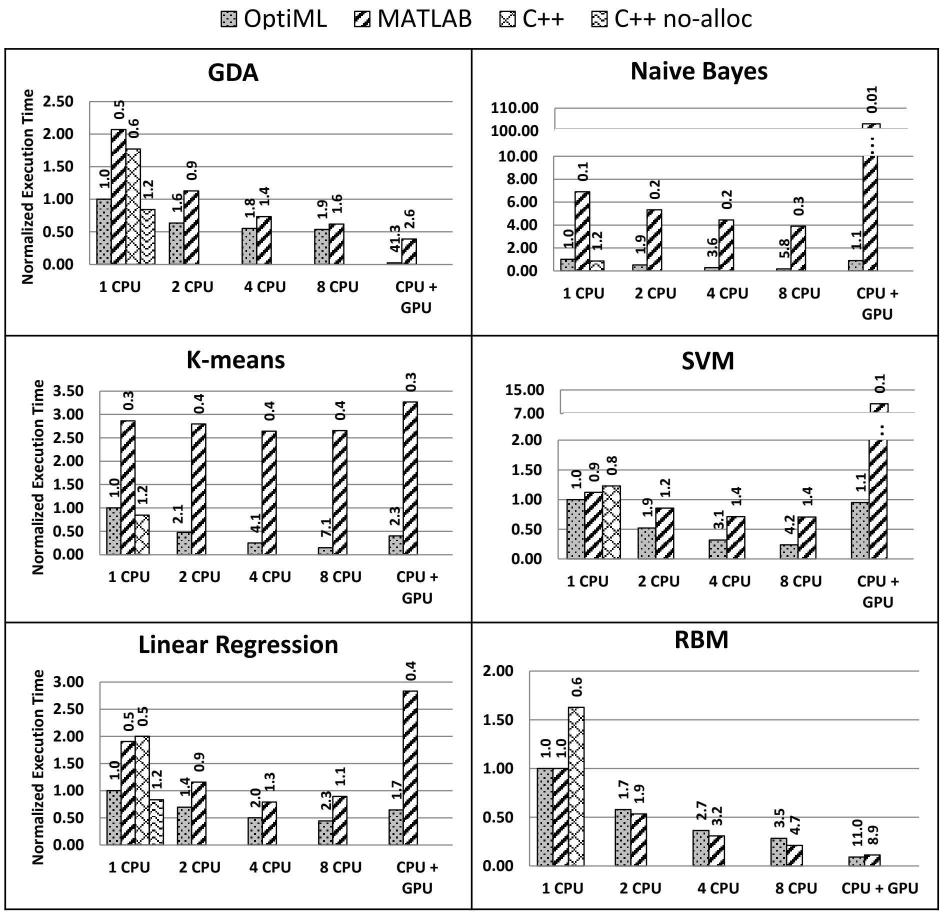

Fig. 4: Execution time of the various versions of our applications normalized to single-threaded OptiML. Speedup numbers are reported on top of each bar.

least as well as the other versions and scales (through parallel BLAS). We therefore believe OptiML should achieve comparable results to MATLAB by simply making more OptiML operations use BLAS implementations.

C. GPU compilation vs. library approach

In this experiment we investigate the impact of GPU compilation on Delite and OptiML, and the result is shown in Figure 5. Applications that mostly consist of matrix/vector operations supported by common GPU libraries, such as RBM, show similar performance in both approaches since Delite can simply generate code that calls the efficiently implemented GPU libraries (e.g., CUBLAS). Naive Bayes, however, spends most of its time executing a user-defined Map operation, which Delite generates as a single GPU kernel. In the library-based version the Map must be broken down into multiple library calls, introducing more kernel launch overhead and additional main memory accesses across kernel boundaries, which results in 2.3x slower execution. For Naive Bayes the ability to generate a custom Cuda kernel from a user-defined function was critical, but being able to analyze and optimize the generated code was not. Therefore, while Delite code generation outperformed the Delite library, a simpler source-to-source translation approach similar to our previous work [5] is also viable here. GDA, however, is an example of where generating an efficient custom Cuda kernel was only possible by building an IR composed of both the application function and the DSL functions called within the application function. With that IR Delite is able to localize all the work required for each output element within a thread so that all intermediate results are kept in registers. This static optimization significantly reduces the number of GPU main memory accesses, resulting in 5.5x speedup compared to the library-based equivalent.

D. Static optimizations

Op fusing can significantly improve application performance by computing multiple loop-based operations within a single loop, which can result in improved cache behavior, fewer necessary memory allocations, and a reduction in the total number of memory accesses. In this experiment we compare the results of applying fusing and Delite's generic optimizations to the Downsampling application shown in Listing 1 to the performance of the hand-optimized C++ version shown in Listing 2. A straightforward generation of the

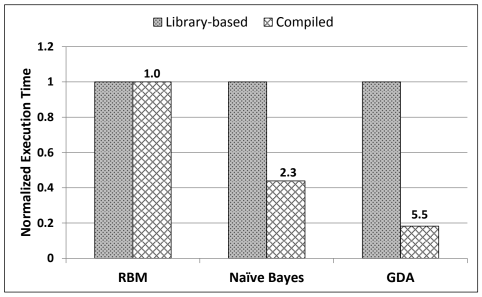

Fig. 5: GPU normalized execution time of three of the apps shown in Figure 4 using Delite code generation compared to a pure library-based approach. Speedups are reported at the top of each bar.

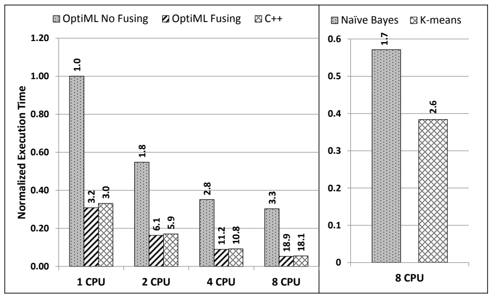

Fig. 6: left) Normalized execution time of Downsampling in C++ and OptiML with and without fusing optimizations. right) Execution time of Naive Bayes and k-means in OptiML with fusing enabled normalized to OptiML with fusing disabled. Speedups are reported at the top of each bar.

OptiML version results in over 3x slower execution compared to the C++ version, shown in Figure 6. This performance is equivalent to the best achievable in a purely library-based approach for Delite on this application. However, as can be seen in Listing 1, each row in the loop is only actually needed if the condition holds, and enabling code motion pushes the DSL row initialization function inside the condition as desired. In addition, enabling fusing generates all of the bulk operations using only a single pass over the input dataset (instead of 3 passes) and eliminates the two temporary Vector allocations. These transformations result in generated code that is 7% faster than the C++ version that performs all of these optimizations manually. It is important to note that these optimizations require performing transformations across both DSL library code and application code, which would not be possible in a library-based approach.

Enabling fusing also had a significant performance impact on Naive Bayes and k-means, resulting in 1.7x and 2.6x faster execution, respectively, as shown in Figure 6. The results shown in Figure 4 for these apps include fusing optimizations, with the C++ versions manually optimized.

Section III-D describes a much more efficient implementation of the GDA algorithm using matrix multiplication that the OptiML compiler can generate automatically from the description in Listing 4 using IR pattern matching. This transformation significantly reduces execution time by eliminating memory allocations and improving cache behavior, which results in both improved single-threaded performance and improved multi-threaded performance, as the memory system becomes less of a scaling bottleneck. Figure 4 shows our performance results for GDA without this transformation. Enabling this optimization yielded 20.4x faster execution with one thread and 48.3x faster execution with eight threads compared to our original performance results.

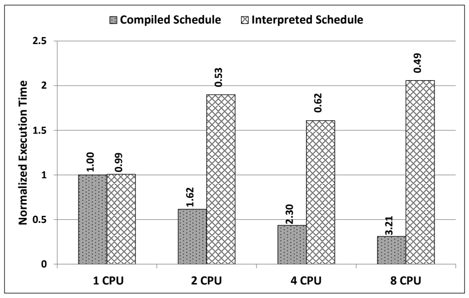

Fig. 7: Normalized execution time of GDA using a 64 element input for the compiled and interpreted implementations of the Delite Runtime. Speedups are reported at the top of each bar.

E. *Runtime schedule compilation*

As discussed in Section IV-B, the runtime utilizes post-scheduling code generation to minimize overheads and bottlenecks that a non-specialized execution can incur. For this experiment, we implemented an “interpreted” version of the runtime in which the execution graph is traversed and scheduled dynamically by a master thread. The master thread schedules each op and then enqueues it for execution on the selected worker thread. As each op is dequeued by a worker thread, it synchronizes with every op it depends on, executes, and then synchronously stores its output(s). This purely dynamic approach closely follows the design of the library runtime in our previous work [5], and was necessary since the execution graph was created dynamically by a master thread, as discussed in Section V-B. Such a design, however, necessarily spends more time executing code at run-time that is not from application kernels, and therefore creates a much greater minimum op size required to achieve performance scaling. In this experiment we ran GDA on a small input (64 elements), and compare the performance results between the interpreted version and the compiled version, shown in Figure 7. At these execution times, the interpreted version has too much run-time overhead and only slows down with multiple processors. The compiled version however, achieves comparable scaling to that observed for the larger dataset shown in Figure 4.

VII. Related Work

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 [16]. 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 [17]. Menon et al. apply high level transformations to MATLAB code, producing performance gains in both interpreted and compiled code [18]. Guyver et al. present a way to annotate library methods with domain-specific knowledge and show significant performance improvements [19]. CodeBoost [20] 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 [21] and Template Haskell [22]. Expression Templates [23] allow customizable generation, and TaskGraph [24] performs runtime code generation from C++. Telescoping languages [25] are efficient DSLs created from annotated component libraries. Designated multi-stage programming languages include MetaML [26] and MetaOCaml [27]. The Delite framework is built on top of the *Lightweight Modular Staging* approach [8], inspired from the related work on embedding typed languages by Carette et al. [28] and Hofer et al. [29]. Libraries using domain-specific code generation and optimization include ATLAS [30] (linear algebra), FFTW [31] (discrete Fourier transform), and SPIRAL [32] (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 [33] and OpenCL [34] 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 [35] 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 [36] 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 [37] provides automatic Cuda code generation from a data-parallel subset of Python. FlumeJava [38] is a Java library targeting Google's MapReduce [39] that optimizes the data-flow graph to create an efficient pipeline of MapReduce operations. Intel's Array Building Blocks [40] 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) [41] 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 [42] 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 [43] automatically translates LINQ [44] 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 [45], Fortress [46], and X10 [47]. 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 [48], Fortress, High Performance Fortran [49], NESL [50], and X10. DSLs which utilize the Delite framework are able exploit implicit data parallelism as well as implicit task parallelism.

VIII. Conclusion

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.

Acknowledgment

The authors thank the reviewers for their feedback, Michael Garland for shepherding this paper, and Peter B. Kessler for reviewing drafts. This research was sponsored by the DOE through Sandia order 942017, the Army under contract AHPCRC W911NF-07-2-0027-1, DARPA through Oracle order US103282, the ERC under grant 587327 "DOPPLER", and the Stanford Pervasive Parallelism Lab affiliates program, which is supported by Oracle/Sun, NVIDIA, AMD, NEC, and Intel.

References

[1] K. Olukotun, B. A. Nayfeh, L. Hammond, K. G. Wilson, and K. Chang, “The case for a single-chip multiprocessor,” in *ASPLOS ’96*.

[2] 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.

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

[4] A. van Deursen, P. Klint, and J. Visser, "Domain-specific languages: an annotated bibliography," *SIGPLAN Not.*, vol. 35, no. 6, pp. 26–36, 2000.

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

[6] M. Odersky, "Scala," http://www.scala-lang.org, 2011.

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

[8] T. Rompf and M. Odersky, "Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls," ser. GPCE, 2010.

[9] A. K. Sujeeth, H. Lee, K. J. Brown, T. Rompf, M. Wu, A. R. Atreya, M. Odersky, and K. Olukotun, "OptiML: an implicitly parallel domain-specific language for machine learning," in *Proceedings of the 28th International Conference on Machine Learning*, ser. ICML, 2011.

[10] T. Rompf, A. K. Sujeeth, H. Lee, K. J. Brown, H. Chafi, M. Oderksy, and K. Olukotun, "Building-blocks for performance oriented dsls," *Electronic Proceedings in Theoretical Computer Science*, 2011.

[11] M. Paleczny, C. Vick, and C. Click, "The java hotspot(tm) server compiler," in *In USENIX Java Virtual Machine Research and Technology Symposium*, 2001, pp. 1–12.

[12] NVIDIA, "CUDA," http://developer.nvidia.com/object/cuda.html.

[13] J. Fisher, "Walk-time techniques: catalyst for architectural change," *Computer*, vol. 30, no. 9, pp. 40–42, Sep. 1997.

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

[15] MathWorks, "Matlab," http://www.mathworks.com/products/matlab/.

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

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

[18] 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*. New York, NY, USA: ACM, 1999, pp. 53–65.

[19] 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*. New York, NY, USA: ACM, 1999, pp. 39–52.

[20] 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*, Sept. 2003, pp. 65–74.

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

[22] T. Sheard and S. Jones, "Template meta-programming for Haskell," *ACM SIGPLAN Notices*, vol. 37, no. 12, pp. 60–75, 2002.

[23] T. L. Veldhuizen, "Expression templates, C++ gems," SIGS Publications, Inc., New York, NY, 1996.

[24] O. Beckmann, A. Houghton, M. Mellor, and P. H. Kelly, "Runtime code generation in c++ as a foundation for domain-specific optimisation," in *Domain-Specific Program Generation*, ser. Lecture Notes in Computer Science, C. Lengauer, D. Batory, C. Consel, and M. Odersky, Eds. Springer Berlin / Heidelberg, 2004, vol. 3016, pp. 77–210.

[25] 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*, vol. 93, no. 3, p. 387–408, 2005, this provides a current overview of the entire Telescoping Languages Project.

[26] W. Taha and T. Sheard, "Metaml and multi-stage programming with explicit annotations," *Theor. Comput. Sci.*, vol. 248, no. 1-2, pp. 211–242, 2000.

[27] C. Calcagno, W. Taha, L. Huang, and X. Leroy, "Implementing multi-stage languages using asts, gensym, and reflection," in *GPCE*, 2003, pp. 57–76.

[28] J. Carette, O. Kiselyov, and C. chieh Shan, "Finally tagless, partially evaluated," in *APLAS*, 2007, pp. 222–238.

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

[30] R. C. Whaley, A. Petitet, and J. Dongarra, "Automated empirical optimizations of software and the ATLAS project," *Parallel Computing*, vol. 27, no. 1-2, pp. 3–35, 2001.

[31] M. Frigo, "A fast fourier transform compiler," in *PLDI*, 1999, pp. 169–180.

[32] M. Püschel, J. M. F. Moura, B. Singer, J. Xiong, J. Johnson, D. A. Padua, M. M. Veloso, and R. W. Johnson, "Spiral: A generator for platform-adapted libraries of signal processing alogorithms," *IJHPCA*, vol. 18, no. 1, pp. 21–45, 2004.

[33] 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*. New York, NY, USA: ACM, 2007, pp. 156–166.

[34] The Khronos Group, "OpenCL 1.0," http://www.khronos.org/opencl/.

[35] 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: ACM, 2008.

[36] 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*. New York, NY, USA: ACM, 2008, pp. 197–200.

[37] B. Catanzaro, M. Garland, and K. Keutzer, "Copperhead: compiling an embedded data parallel language," in *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, ser. PPoPP '11. New York, NY, USA: ACM, 2011, pp. 47–56.

[38] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum, "Flumejava: easy, efficient data-parallel pipelines," in *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '10. New York, NY, USA: ACM, 2010, pp. 363–375.

[39] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *OSDI*, 2004, pp. 137–150.

[40] "Intel array building blocks," http://software.intel.com/en-us/articles/intel-array-building-blocks.

[41] Z. Budimlic, M. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. M. Peixotto, V. Sarkar, F. Schlimbach, and S. Tasirlar, "Concurrent collections," *Scientific Programming*, vol. 18, no. 3-4, pp. 203–217, 2010.

[42] 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*. New York, NY, USA: ACM, 2007, pp. 59–72.

[43] 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*. New York, NY, USA: ACM, 2009, pp. 987–994.

[44] 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*. New York, NY, USA: ACM, 2006, pp. 706–706.

[45] B. Chamberlain, D. Callahan, and H. Zima, "Parallel Programmability and the Chapel Language," *Int. J. High Perform. Comput. Appl.*, vol. 21, no. 3, pp. 291–312, 2007.

[46] G. L. S. Jr., "Parallel programming and parallel abstractions in fortress," in *IEEE PACT*, 2005, p. 157.

[47] 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.*, vol. 40, no. 10, pp. 519–538, 2005.

[48] S. L. P. Jones, R. Leshchinskiy, G. Keller, and M. M. T. Chakravarty, "Harnessing the Multicores: Nested Data Parallelism in Haskell," in *FSTTCS*, 2008, pp. 383–414.

[49] "High Performance Fortran," http://hpff.rice.edu/index.htm.

[50] G. E. Blelloch, "Programming parallel algorithms," *Commun. ACM*, vol. 39, no. 3, pp. 85–97, 1996.