Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages

ARVIND K. SUJEETH, KEVIN J. BROWN, and HYOUKJOONG LEE, Stanford University
TIARK ROMPF, Oracle Labs and EPFL
HASSAN CHAFI, Oracle Labs and Stanford University
MARTIN ODERSKY, EPFL
KUNLE OLUKOTUN, Stanford University

Developing high-performance software is a difficult task that requires the use of low-level, architecture-specific programming models (e.g., OpenMP for CMPs, CUDA for GPUs, MPI for clusters). It is typically not possible to write a single application that can run efficiently in different environments, leading to multiple versions and increased complexity. Domain-Specific Languages (DSLs) are a promising avenue to enable programmers to use high-level abstractions and still achieve good performance on a variety of hardware. This is possible because DSLs have higher-level semantics and restrictions than general-purpose languages, so DSL compilers can perform higher-level optimization and translation. However, the cost of developing performance-oriented DSLs is a substantial roadblock to their development and adoption. In this article, we present an overview of the Delite compiler framework and the DSLs that have been developed with it. Delite simplifies the process of DSL development by providing common components, like parallel patterns, optimizations, and code generators, that can be reused in DSL implementations. Delite DSLs are embedded in Scala, a general-purpose programming language, but use metaprogramming to construct an Intermediate Representation (IR) of user programs and compile to multiple languages (including C++, CUDA, and OpenCL). DSL programs are automatically parallelized and different parts of the application can run simultaneously on CPUs and GPUs. We present Delite DSLs for machine learning, data querying, graph analysis, and scientific computing and show that they all achieve performance competitive to or exceeding C++ code.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; D.3.2 [**Programming Languages**]: Language Classifications—*Concurrent, distributed and parallel languages, Extensible languages*; D.3.4 [**Programming Languages**]: Processors—*Code generation, Optimization, Run-time environments*

General Terms: Languages, Performance

Additional Key Words and Phrases: Domain-specific languages, multistage programming, language virtualization, code generation

This research was sponsored by DARPA Contract SEEC: Specialized Extremely Efficient Computing, Contract no. HR0011-11-C-0007; DARPA Contract Xgraphs; Language and Algorithms for Heterogeneous Graph Streams, FA8750-12-2-0335; NSF grant SHF: Large: Domain Specific Language Infrastructure for Biological Simulation Software, CCF-1111943; Stanford PPL affiliates program, Pervasive Parallelism Lab: Oracle, AMD, Intel, NVIDIA, and Huawei; and the European Research Council (ERC) under grant 587327 "DOPPLER". Authors also acknowledge additional support from Oracle. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.
Authors' addresses: A. K. Sujeeth (corresponding author), K. J. Brown, and H. Lee, Stanford University, CA; email: asujeeth@stanford.edu; T. Rompf, Oracle Labs and EPFL; H. Chafi, Oracle Labs and Stanford University, CA; M. Odersky, EPFL; K. Olukotun, Stanford University, CA.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee.
© 2014 ACM 1539-9087/2014/03-ART134 $15.00
DOI: http://dx.doi.org/10.1145/2584665

ACM Reference Format:
Arvind K. Sujeeth, Kevin J. Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2014. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. ACM Trans. Embedd. Comput. Syst. 13, 4s, Article 134 (March 2014), 25 pages.
DOI: http://dx.doi.org/10.1145/2584665

1. INTRODUCTION

The software industry is facing a difficult dilemma. Programming languages are advancing towards higher-level abstractions, enabling complex programs to be written more productively. At the same time, hardware architectures are advancing towards more heterogeneous parallel processing elements (multiple CPUs, GPUs, and domain-specific accelerators). As the abstraction gap between applications and hardware increases, compilers are becoming increasingly ineffective at automatically exploiting hardware resources to achieve the best performance. The primary culprit is that high-level general-purpose languages lack the semantic information required to efficiently translate coarse-grained execution blocks to low-level hardware. If performance is required, programmers are forced to instead use low-level, architecture-specific programming models (e.g., Pthreads, CUDA, MPI).

Unfortunately, using these disparate programming models to optimize programs for performance typically incurs a great loss in productivity. Programmers often begin by writing a prototype in a high-level language that is concise and expressive and then translate their program to low-level, high-performance code in a time-consuming process. In order to get the best performance, programmers must have hardware expertise and develop a different version of the code for each target device that may be required. The resulting code is tied to architectural details (e.g., cacheline size, number of processor cores) and is harder to read, debug, maintain, or port to future platforms. Worse, there are now multiple versions of the original application: one that is too high level to obtain good performance, and several that are too low level to easily understand. Keeping the different versions in sync as new features are added or bugs are fixed quickly becomes untenable.

Domain-Specific Languages (DSLs) have been proposed as a solution that can provide productivity, performance, and portability for high-level programs in a specific domaizn [Chafi et al. 2011]. DSLs provide high-level abstractions directly to the user, typically leaving implementation details unspecified in order to free the DSL compiler to translate the application in different ways to low-level programming models. For example, a domain-specific compiler for linear algebra can reason about the program at the level of vector and matrix arithmetic, as opposed to loops, instructions, and scalars. A DSL compiler can also enforce domain-specific restrictions; while general-purpose languages usually perform sophisticated analyses to determine when it is safe to apply optimizations and must be conservative, DSLs can often apply optimizations that are correct by construction.

Despite the benefits of performance-oriented DSLs, the cost of developing a DSL compiler is immense and requires expertise in multiple areas, from language design to computer architecture. In addition to the initial cost, DSL compiler developers must also continously invest new effort to target new or updated hardware. As a result, few DSLs are developed, even in domains that could potentially benefit from an optimizing DSL. Furthermore, the wide range in quality, tooling, and performance of DSLs is a substantial obstacle for DSL adoption.

In this article, we synthesize the existing work and present a comprehensive overview of Delite, an open-source DSL compiler framework [Brown et al. 2011; Rompf et al. 2011]. Delite substantially reduces the burden of developing high-performance DSLs by providing common reusable components that enable DSLs to quickly and efficiently

target heterogeneous hardware [Sujeeth et al. 2013b]. We present existing Delite DSLs for machine learning (OptiML), data querying (OptiQL), graph analysis (OptiGraph), and mesh computation (OptiMesh). Utilizing a few common principles, we show that this diverse set of DSLs can be implemented using a small number of components and that there is extensive sharing of components across DSLs. In particular, we identify the important computations in each domain and show that they can be expressed with Delite parallel patterns (the four DSLs use a combined 9 parallel ops). This reuse enables each new DSL to be implemented with either small incremental changes to the framework or no changes at all. We show that each DSL provides both performance and productivity by summarizing previous results that demonstrate performance comparable to C++ with a much higher-level interface. We conclude with a discussion of the key insights from this approach and the challenges ahead.

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

2. A DOMAIN-SPECIFIC APPROACH TO HETEROGENEOUS PARALLELISM

As a programming model, DSLs sacrifice generality in order to achieve both productivity and performance. Productivity subjectively captures how easy it is to use a programming language to accomplish some task, typically in terms of the provided abstractions, the readability and maintainability of the code, and the extent to which rapid prototyping and debugging is possible. DSLs have traditionally been leveraged for productivity; since the language is specific to a particular problem domain, the syntax and the abstractions of a well-designed DSL are a close match to the applications that users want to write. These high-level abstractions enable users to focus on ideas and algorithms rather than implementation details. In contrast, an optimized C++ implementation will often play numerous tricks with memory layout (e.g., byte padding), memory reuse (e.g., blocking or manual loop fusion), and control flow (e.g., loop unrolling), even before adding threading or vectorization which obfuscate the implementation even more. While it is difficult for general-purpose compilers to perform these optimizations automatically, DSL compilers can exploit restricted semantics and high-level structure to make these automatic transformations tractable. Therefore, DSLs provide an attractive vehicle not just for raising the productivity level of programmers, but also for enabling them to write high-performance applications on modern hardware, which is parallel, heterogeneous, and distributed.

Besides DSLs, there are two major high-level programming models that have had some success in parallel and heterogeneous computing. The first is to use a restricted general-purpose programming model, such as MapReduce [Dean and Ghemawat 2004; Apache 2014; Zaharia et al. 2011], Pregel [Malewicz et al. 2010], OpenCL [The Khronos Group 2014], or Intel Cilk Plus [Intel 2014]. These systems have the advantage of being suitable for many different applications within a single language, but often force programmers to express their algorithms in a model that is unnatural to the problem, thereby sacrificing some productivity. They are also usually unable to achieve the best performance across different hardware devices, because the languages are still too low level for a single version of the source code to be optimal for different hardware. For example, it is not uncommon to have to rewrite an OpenCL kernel in slightly different ways in order to get the best performance on either a CPU or a GPU. A second approach is to simply compose applications out of libraries of high-performance primitives, such as Intel MKL [Intel 2013] or hand-written, hand-parallelized C++ code. While this approach has the benefit of being the easiest to integrate of any of the alternatives, composing an application out of library calls that are individually high performance does not necessarily result in a high-performance application. The main issue is that each library call is a black box to an optimizing compiler; critical optimizations that

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

There are many implementation choices for DSLs and DSL compilers. *External* DSLs are stand-alone languages with custom compilers. These DSLs have the most flexibility, since their syntax, semantics, and execution environments can be completely domain specific. However, they require considerable effort and compiler expertise to construct. Compiler frameworks such as LLVM [Lattner and Adve 2004] can reduce some of the burden by providing built-in optimizations and code generators, but DSL authors must still define how to map from the high-level DSL constructs to the low-level Intermediate Representation (IR). On the other hand, *internal* DSLs are embedded in a host language. The most common implementation is *pure* or *shallow* embedding, where DSLs are implemented as "just a library" in a flexible host language. Purely embedded DSLs often use advanced host language features (e.g., function currying) to emulate domain-specific syntax, but are legal programs in the host language and share the host language's semantics. As a result, they are simple to construct and DSL users can reuse the host language's tool-chain (e.g., IDE). However, they are constrained to the performance and back-end targets of the host language. In Delite, we attempt to strike a balance between these two approaches; Delite DSLs are *embedded* DSLs that can still be *compiled* to heterogeneous hardware.

3. EMBEDDING COMPILERS WITH STAGING

In order to reduce the programming effort required to construct a new DSL, Delite DSLs are implemented as embedded DSLs inside Scala, a high-level hybrid object-oriented functional language that runs on the Java Virtual Machine (JVM) and interoperates with Java. We use Scala because its combination of features makes it well suited for embedding DSLs [Chafi et al. 2010]. Most importantly, its strong static type system and mix-in composition[1] make DSLs easier to implement, while higher-order functions, function currying, and operator overloading enable the DSLs to have a flexible syntax despite being embedded. The choice of Scala as our embedding language is a convenient one, but is not fundamental to the approach.

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

We use a technique called lightweight modular staging, a form of *staged metaprogramming*, to generate an IR from Scala application code [Rompf and Odersky 2010]. Consider the following small snippet of code in a simple Vector DSL.

```
val v1 = Vector.rand(1000)
val v2 = Vector.rand(1000)
```

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

```
val a = v1+v2
println(a)
```

The key idea is that all DSL types (e.g., `Vector`) are wrapped in an abstract type constructor, `Rep[T]`. `Rep` stands for representation; a `Rep[Vector[T]]` is a representation of a `Vector[T]` that is not tied to a particular concrete implementation. Type inference is used to hide this wrapped type from application code, as given before. Instead of immediate evaluation, DSL operations on `Rep[T]` construct an IR node representing the operation. For example, the `Vector.rand` statement is implemented inside the DSL to construct an IR node, `VectorRand`, and return a symbol of type `Rep[Vector[Double]]`. Inside the compiler, `Rep[T]` is defined concretely to be `Exp[T]`, representing an expression. This avoids exposing internal compiler types to user programs. The process of constructing IR nodes from application operations is called *lifting*. To ensure that all host language operations can be intercepted and lifted, we use a modified version of the Scala compiler, *Scala-virtualized* [Moors et al. 2012], that enables overloading even built-in Scala constructs such as `IfThenElse`. Delite also provides lifted versions of many Scala library operations (e.g., string methods); these can be inherited by DSLs for free.

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

(1) The DSL application (Scala code) is compiled using *scalac* to generate Java bytecode.
(2) The Java bytecode is executed (staged) to run the DSL compiler in order to build the IR, perform optimizations, and generate code.
(3) The resulting generated code is compiled for each target (e.g., C++, CUDA).

In return for more compilation, we have two primary benefits: Delite DSL compilers are simpler to build than stand-alone DSL compilers, and DSL programs can retain high-level abstractions without suffering a runtime penalty, since the generated code is low level and first order.

4. THE DELITE FRAMEWORK

All Delite DSLs share the same architecture for building an intermediate representation, traversing it, and generating code. The Delite framework was first discussed in detail by Brown et al. [2011]; it has since been enhanced to support data structures and transformations using the techniques outlined by Rompf et al. [2013]. Here we briefly describe the key principles of the Delite framework. We then present the semantics for the parallel patterns that Delite currently supports. Finally, we describe the heterogeneous runtime service that consumes the generated code from the DSL compilers and executes it on multiple hardware resources.

4.1. Principles

Several requirements led to design choices that make the Delite compiler architecture different from traditional compilers. These include the need to support a large number of IR node types for domain operations, to specify optimizations in a modular way, and to generate code for a variety of different targets. Delite's architecture meets these requirements by organizing DSLs around common design principles. These principles

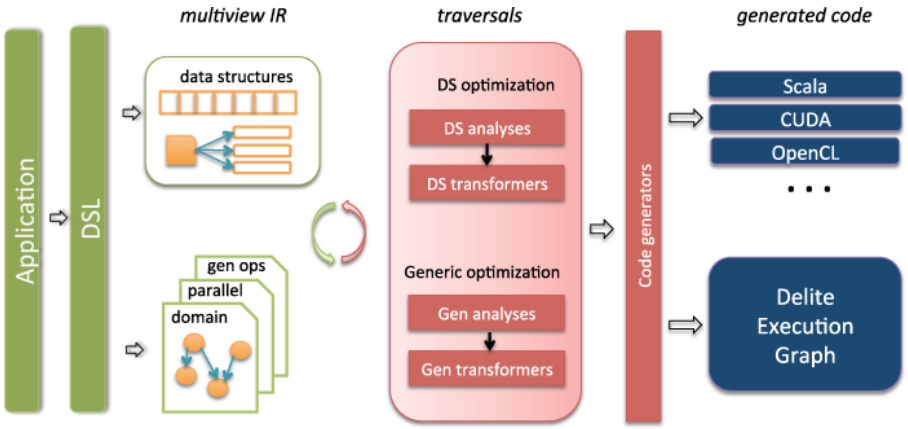

Fig. 1. Components of the Delite framework. An application is written in a DSL, which is composed of data structures and structured computations represented as a multiview IR. The IR is transformed by iterating over a set of traversals for both Generic (Gen) and Domain-Specific (DS) optimizations. Once the IR is optimized, heterogeneous code generators emit specialized data structures and ops for each target along with the Delite Execution Graph (DEG) that encodes dependencies between computations.

are implemented with a set of reusable components: common IR nodes, data structures, parallel operators, built-in optimizations, traversals, transformers, and code generators (Figure 1). DSLs are developed by extending these reusable components with domain-specific semantics. Furthermore, Delite is modular; any service it provides can be overridden by a particular DSL with a more customized implementation. The remainder of this section discusses Delite's principles and the components that implement them.

Common IR. The Delite IR is a sea of nodes representation [Paleczny et al. 2001], rather than a traditional Control-Flow Graph (CFG); nodes explicitly encode their data and control dependencies as inputs, but are otherwise free to "float". This representation is well suited for parallelism, since it does not maintain any extra constraints on program order (e.g., syntactic) than are necessary for correctness. The IR is traversed by taking the symbolic result of a block and following dependencies backwards; the result is a program schedule that can either be optimized or passed to the code generation phase. One important characteristic of Delite is that IR nodes are simultaneously viewed in multiple ways: a generic layer, called `Def[T]` (representing a definition), a parallel layer, called Delite ops, and a domain-specific layer, which is the actual operation, for example, `VectorPlus`. The IR can be optimized by viewing nodes at any layer, enabling both generic and domain-specific optimizations to be expressed on the same IR.

Structured data. Delite DSLs present domain-specific types to end-users (e.g., `Vector`) and methods on instances of those types get lifted into the IR. However, DSL back-end data structures are restricted to Delite `Structs`, C-like structs with a limited set of field types (primitives). The fields currently allowed in a Delite `Struct` are: numerics (e.g., `Int`), `Boolean`, `String`, `Array`, and `Map`. By restricting the content of data structures, Delite is able to perform optimizations such as Array-of-Struct (AoS) to Struct-of-Array (SoA) conversion and dead field elimination automatically [Rompf et al. 2013]. Furthermore, since the set of primitives is fixed, Delite can implement these primitives on each target platform (e.g., C++, CUDA) and automatically generate code for DSL structs. Delite also supports user-defined structs by lifting the **`new`**

keyword defining an anonymous class [Moors et al. 2012]. An application developer can write code like the following.

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

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

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

Explicit DSL semantics. DSL authors communicate DSL semantics to Delite via a set of simple APIs. Effectful operations are marked using the `reflectEffect` or `reflectWrite` methods when constructing an IR node. Delite uses this information to honor read-after-write, write-after-read, and write-after-write dependencies when constructing the program schedule. DSL authors can also optionally specify additional semantic information about particular IR nodes that enables Delite to be more aggressive with optimizations. For example, a DSL author can specify whether an input symbol is likely to be accessed more or less often during an operation using the `freqSyms` method, which allows the code motion algorithm to better determine whether to hoist a computation out, or push a computation into, a candidate block. Similarly, the `aliasSyms` method allows DSL authors to declare if an operation constructs an alias to one of its inputs (e.g., by storing a reference to it), which can enable Delite to be less conservative when performing optimizations. The key idea behind all of these mechanisms is that by communicating a minimal amount of domain-specific information, we can often efficiently parallelize and optimize DSL programs without complicated generic analyses.

Generic optimizations. Delite performs several optimizations for all DSLs (these optimizations can be programmatically disabled by a DSL if desired). At the generic IR layer, Delite performs Common Subexpression Elimination (CSE), Dead Code Elimination (DCE), and code motion. Although these are traditional optimizations, it is important to note that in the Delite framework they occur on symbols that represent coarse-grained domain-specific operations (such as `MatrixMultiply`), rather than generic language statements as in a typical general-purpose compiler. Therefore, the potential impact of these optimizations in a program is substantially greater. At the Delite op layer, Delite implements *op fusion* [Rompf et al. 2013]; data-parallel ops that have a producer-consumer or sibling relationship are fused together, eliminating temporary allocations and extra memory accesses. Finally, as mentioned previously, Delite performs AoS→SoA conversion and dead field elimination for data structures, which is essential to increasing the number and performance of ops that can be run on wide, column-oriented devices such as GPUs.

Domain-specific optimizations. Domain-specific optimizations can be implemented as rewrite rules using pattern matching on IR nodes; we can layer these rewritings in a modular way by putting them in separate traits[2]. Consider the following simple pattern.

```
override def vector_plus(x: Exp[Vector[Int]], y: Exp[Vector[Int]]) =
  (x,y) match {
    case (a, Def(VectorZeros(len))) => a
    case _ => super.vector_plus(x,y)
  }
```

This rewrite rule fires during IR construction and checks for the case of a vector being added to zeros and eliminates the addition; if the match fails, it falls back to the original implementation of `vector_plus` which constructs an IR node.

More sophisticated analyses or optimizations are possible by implementing `Traversals` and `Transformers`. Traversals schedule the IR and visit each node in order, allowing the DSL author to define an arbitrary operation (such as collecting statistics for an analysis) at each node. Transformers combine traversals with rewrite rules, and can be run as phases in a predetermined order. Transformers are ideal for performing *lowering* optimizations, that is, translating a domain-specific operation to a different, lower-level representation that can be optimized further. One example is performing device-specific optimizations: we can define a transformer that will transform nested parallel loops into a flattened parallel loop for highly multithreaded devices. Transformers are also useful for performing domain-specific rewrites on the IR that require the entire IR to be constructed first.

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

4.2. Parallel Patterns

Delite currently supports `Sequential`, `Map`, `Reduce`, `ZipWith`, `Foreach`, `Filter`, `GroupBy`, `Sort`, `ForeachReduce`, and `FlatMap` ops. The semantics of the ops are as follows[3].

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

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

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

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

—*Reduce*. The inputs to `Reduce` are a collection `DeliteCollection[A]`, a reduction function `reduce: (A,A) => A`, and a zero element `A`. The `reduce` function must be associative. `Reduce` performs a parallel tree reduction on the input collection, optimized for each device. The result of `Reduce` is a single element `A`.

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

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

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

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

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

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

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

Since new ops may be required for new DSLs, Delite is designed to make adding new ops relatively easy. Most of the existing Delite ops extend a common loop-based abstraction, `MultiLoop`. `MultiLoop` is a highly generic data-parallel loop; the loop body consists of one or more *elems*. An *elem* can be a `collect`, `reduce`, `hash`, or a `foreach`, corresponding to appending elements to an output collection, reducing elements into an accumulator, grouping elements into multiple collections, or applying a side-effectful function, respectively. Thus, `MultiLoop` generalizes most of the Delite ops described before. It is the key abstraction that enables Delite's fusion algorithm to combine multiple ops into a single op. It also allows us to implement the code generation for most ops once (via the code generation for `MultiLoop`), instead of for each op individually. To add a new op to Delite, the key work that must be done is to either express it in terms of existing ops, or to define the code generation of the new op for each target platform. Once an op is added to Delite, all DSLs can take advantage of it. As more DSLs are implemented with Delite, the coverage of ops increases and the likelihood of a new op being necessary decreases.

4.3. Heterogeneous Runtime

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

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

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

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

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

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

5. BUILDING A NEW DELITE DSL

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

In this section, we show how one can define a very small DSL to implement the example shown in Section 3. The first step is to define a `Vector`.

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

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

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

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

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

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

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

This snippet shows how we construct two Delite parallel ops, `DeliteOpMap` and `DeliteOpZipWith`, by supplying the functions that match the ops' interface (`func` and `alloc`). In both cases, `func` is required to be pure (have no side-effects). Since effects must be explicitly annotated when constructing an IR node, this condition is easy to check at staging time. This trait completes the definition of our small DSL, and will enable us to run the example in Section 3. Running the program will construct these IR nodes, which will then be scheduled and code generated automatically by Delite (since every IR node is a Delite op). Furthermore, since the `Vector.rand` and `+` functions are both data-parallel ops that operate on the same range, Delite will automatically fuse all of the operations together, resulting in a single loop without any temporary allocations to hold the `Vector.rand` results.

We could take this example a step further by implementing a `Vector.zeros` method in a similar fashion to `Vector.rand`, and then implement the rewrite rule shown in Section 4.1 that optimizes additions with zero vectors. If we change the example from Section 3 to make `v2` a call to `Vector.zeros`, the rewrite rule will automatically optimize the addition away.

In this example, we demonstrated how to directly construct an embedded DSL using the Delite APIs. In recent work, however, we have also explored simplifying this process by using a higher-level meta-DSL called Forge [Sujeeth et al. 2013a]. Forge provides a high-level, declarative API for specifying DSL data structures and operations in a manner similar to an annotated high-level library. DSL developers targeting Forge instead of Delite directly can reduce the total lines of code in their implementation by 3–6x due to the reduction in boilerplate and low-level embedding implementation details, at the cost of some flexibility. Forge can also shield DSL developers from changes in Delite or allow DSLs to be retargeted to other back-ends (like a pure Scala implementation) transparently.

6. DSLS

In this section, we present existing Delite DSLs for machine learning (OptiML), data querying (OptiQL), graph analysis (OptiGraph), and mesh computation (OptiMesh). Sujeeth et al. first presented these DSLs [Sujeeth et al. 2011, 2013b]; we expand on this discussion by highlighting Delite components that were reused in the implementation of the DSLs and identifying new features that had to be added to the framework to support a particular DSL. We also show how Delite supports implementing nontrivial domain-specific analyses and transformations for OptiMesh. This section outlines the productivity benefits of using the DSLs, which contain high-level domain-specific abstractions. Section 7 shows that the DSLs are all able to achieve high performance through the Delite framework across this diverse set of domains.

```
// ts: TrainingSet[Double]
// returns a vector (phi_y1) of numTokens length,
// corresponding to each return value
val phi_y1 = (0::numTokens) { j =>
  val spamWordCount =
    sumIf(0, numTrainDocs) { ts.labels(_) == 1 } { i => ts.t(j,i) }
  val spamTotalWords =
    sumIf(0, numTrainDocs) { ts.labels(_) == 1 } { i => wordsPerEmail(i) }
  (spamWordCount + 1) / (spamTotalWords + numTokens)
}
```

Listing 1. Naive Bayes snippet in OptiML.

6.1. OptiML

OptiML is a DSL for machine learning designed for statistical inference problems expressed with vectors, matrices, and graphs [Sujeeth et al. 2011]. In particular, OptiML adds flexible collection operators and machine learning abstractions on top of core linear algebra operations, such as matrix-vector calculations. The language promotes a functional style by restricting when side-effects may be used. For example, an object must be cloned using the `.mutable` method before it can be mutated. Similarly, OptiML language constructs (e.g., `sum`) do not allow side-effects to be used within their anonymous function arguments. These restrictions prevent many common parallel programming pitfalls by construction and enable OptiML to inherit key optimizations from Delite. The most important of these optimizations is op fusion; since machine learning algorithms often contain sequences of fine-grained linear algebra operations, OptiML uses op fusing at the vector-matrix granularity to enable concise programs without unnecessary temporary object allocations. OptiML also performs domain-specific optimizations such as applying linear algebra simplifications via rewritings. Listing 1 shows an example snippet of OptiML code from the naive Bayes algorithm. The `(n::m){i => ...}` statement constructs a vector by mapping an index i to a corresponding result value. This snippet computes the frequency of spam words in a corpus of emails.

Reuse. OptiML uses Delite ops to implement bulk collection operators on vectors and matrices (e.g., `map`, `reduce`, `filter`) as well as arithmetic operators (e.g., `+`,`*`). As the first Delite DSL, many of the original ops and optimizations were designed with OptiML applications as use cases. OptiML relies heavily on Delite's op fusion and uses Delite's CUDA code generation support to target GPUs.

6.2. OptiQL

OptiQL is a DSL for data querying of in-memory collections, and is heavily inspired by LINQ [Meijer et al. 2006], specifically LINQ to Objects. OptiQL is a pure language that consists of a set of implicitly parallel query operators, such as `Select`, `Average`, and `GroupBy`, that operate on OptiQL's core data structure, the `Table`, which contains a user-defined schema. Listing 2 shows an example snippet of OptiQL code that expresses a query similar to Q1 in the TPC-H benchmark. The query first excludes any line item with a ship date that occurs after the specified date. It then groups each line item by its status. Finally, it summarizes each group by aggregating the group's line items and constructs a final result per group.

Since OptiQL is SQL like, it is concise and has a small learning curve for many developers. However, unoptimized performance is poor. Operations always semantically produce a new result, and since the in-memory collections are typically very large, cache locality is poor and allocations are expensive. OptiQL uses compilation to aggressively

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

Listing 2. OptiQL: TPC-H Query 1 benchmark.

optimize queries. Operations are fused into a single loop over the dataset wherever possible, eliminating temporary allocations, and datasets are internally allocated in a column-oriented manner, allowing OptiQL to avoid allocating columns that are not used in a given query. Although not implemented yet, OptiQL's eventual goal is to use Delite's pattern rewriting and transformation facilities to implement other traditional (domain-specific), cost-based query optimizations.

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

New Delite features required. We implemented the user-defined data structure support discussed in Section 4 to support lifting user-defined schemas in OptiQL. The `GroupBy` op and corresponding hashmap semantics were added to support `GroupBy`. `FlatMap` was added to support `Join` and `Sort` was added for `OrderBy`.

6.3. OptiGraph

OptiGraph is a DSL for static graph analysis based on the Green-Marl DSL [Hong et al. 2012]. OptiGraph enables users to express graph analysis algorithms using graph-specific abstractions and automatically obtain efficient parallel execution. OptiGraph defines types for directed and undirected graphs, nodes, and edges. It allows data to be associated with graph nodes and edges via node and edge property types and provides three types of collections for node and edge storage (namely, `Set`, `Sequence`, and `Order`). Furthermore, OptiGraph defines constructs for `BFS` and `DFS` order graph traversal, sequential and explicitly parallel iteration, and implicitly parallel in-place reductions and group assignments. Another important feature of OptiGraph is its built-in support for bulk synchronous consistency via *deferred assignments*.

Listing 3 shows the parallel loop of the PageRank algorithm [Page et al. 1999] written in OptiGraph. PageRank is a well-known algorithm that estimates the relative importance of each node in a graph (originally of Web pages and hyperlinks) based on the number and PageRank of the nodes associated with its incoming edges (`InNbrs`). In the snippet, `PR` is a node property associating a page-rank value with every node

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

Listing 3. OptiGraph: core of the PageRank algorithm.

in the graph. The <= statement is a deferred assignment of the new page-rank value, `rank`, for node `t`; deferred writes to `PR` are made visible after the **`for`** loop completes via an explicit assignment statement (not shown). Similarly, += is a scalar reduction that implicitly writes to `diff` only after the loop completes. In contrast, `Sum` is an in-place reduction over the parents of node `t`. This example shows that OptiGraph can concisely express useful graph algorithms in a naturally parallelizable way; the `ForeachReduce` op implicitly injects the necessary synchronization into the **`for`** loop.

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

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

6.4. OptiMesh

OptiMesh is an implementation of Liszt on Delite. Liszt is a DSL for mesh-based Partial Differential Equation (PDE) solvers [DeVito et al. 2011]. Liszt code allows users to perform iterative computation over mesh elements (e.g., cells, faces). Data associated with mesh elements are stored in external fields that are indexed by the elements. Listing 4 shows a simple OptiMesh program that computes the flux through edges in the mesh. The **`for`** statement in OptiMesh is implicitly parallel and can only be used to iterate over mesh elements. `head` and `tail` are built-in accessors used to navigate the mesh in a structured way. In this snippet, the `Flux` field stores the flux value associated with a particular vertex. As the snippet demonstrates, a key challenge with OptiMesh is to detect write conflicts within **`for`** comprehensions given a particular mesh input.

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

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

```
      for (edge <- edges(mesh)) {
        val flux = flux_calc(edge)
        val v0 = head(edge)
        val v1 = tail(edge)
        Flux(v0) += flux // possible write conflicts!
        Flux(v1) -= flux
      }
```

Listing 4. OptiMesh: simple flux computation.

```
/* This trait is part of the IR and constructs new nodes */
trait ColoringTransformerExp extends OptiMeshExp { self =>
  // result is a symbol representing multiple Foreach ops
  def colorFor(f: Foreach[_], colors: Array[Array[Int]]) = {
    // colors contains indices of each new foreach
    var i = 0
    // while loop is immediately unrolled (not lifted)
    while (i < colors.length) {
      // DeliteArray lifts the 'for' stm to a Foreach op
      for (c <- DeliteArray(colors(i))) {
        f.func(c)
      }
      i += 1
    }
} }

/* This trait is a Transformer that traverses the IR */
trait ColoringTransformer extends ForwardTransformer {
  val IR: ColoringTransformerExp; import IR._
  // colorMap constructed using stencil (impl elided)
  val colorMap: Map[Foreach[_],Array[Array[Int]]]
  override def transformStm(stm: Stm): Exp[Any] = stm match {
    case TP(s,l:Loop[_]) =>
      l.body match {
        // transform a Foreach symbol into a new symbol
        // representing multiple colored Foreaches
        case f:Foreach[_] => colorFor(f,colorMap(f))
        case _ => super.transformStm(stm)
      }
    case _ => super.transformStm(stm)
} }
```

Listing 5. Loop coloring transformation for OptiMesh.

Domain-specific transformations. OptiMesh implements the same analyses and transformations as the stand-alone Liszt language. First, we perform *stencil detection* [DeVito et al. 2011]. The idea is to symbolically evaluate an OptiMesh program with a real mesh input and record fields that are read and written in a single iteration of a parallel foreach. Using the stencil, we build an interference graph and run a coloring algorithm to find disjoint sets of the original foreach that are safe to run in parallel. We implement stencil detection in Delite before code generation by constructing a new traversal, `StencilCollector`. `StencilCollector` visits each node in program order and looks for a `Foreach` op; when it finds one, it records that it is inside a `Foreach` and begins to record subsequent ops to an in-memory data structure representing

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

Fig. 2. Sharing of DSL operations and optimizations.

the stencil. After the `StencilCollector` traversal is finished, OptiMesh has the stencil.

The next step is to use the stencil to perform the actual transformations. We accomplish this by implementing a new transformer, `ColoringTransformer`, shown in Listing 5. `ColoringTransformer` replaces each `Foreach` in the original IR with one or more new `Foreaches`, each corresponding to a different colored set (essentially *loop fission*). After the transformation completes, control is passed back to Delite and code generation proceeds as normal. This example demonstrates that Delite transformations are concise and expressive (the transformed version is written as normal source code, and then restaged to an IR). It also shows that unstaged code and staged code can be mixed within a transformation; this flexibility is important when using analysis results (static expressions) within transformations (lifted expressions), as OptiMesh does.

6.5. Reuse Summary

Figure 2 summarizes the characteristics and reuse of the DSLs introduced in this section. The DSLs inherit most of their functionality from Delite, in the form of a small set of reused parallel patterns and generic optimizations. The DSLs use just nine Delite ops total; seven ops (77.7%) were used in at least two DSLs; three (33.3%) were used in at least three DSLs. At the same time the DSLs are not constrained to built-in functionality, as demonstrated by OptiMesh's domain-specific optimizations.

7. EVALUATION

In this section, we analyze the performance of the Delite DSLs. We compare applications written in each DSL to sequential C++ implementations. For OptiML we also compare against MATLAB [MathWorks 2014] since machine learning researchers often use MATLAB to implement their algorithms. Each Delite DSL generated Scala code for the CPU and CUDA code for the GPU.

All of our experiments were performed on a Dell Precision T7500n with two quad-core Xeon 2.67 GHz processors, 96GB of RAM, and an NVidia Tesla C2050. The CPU-generated Scala code was executed on the Oracle Java SE Runtime Environment 1.7.0 and the Hotspot 64-bit server VM with default options. For the GPU, Delite executed CUDA v4.0. The C++ versions were all compiled with gcc `-O3`. MATLAB code was run with MATLAB 7.11 with its parallel computing toolbox and GPU execution support. We ran each application ten times (to warm up the JIT) and report the average of the last five runs. For each run we timed the computational portion of the application. For each application we show normalized execution time relative to our DSL version with the speedup listed at the top of each bar.

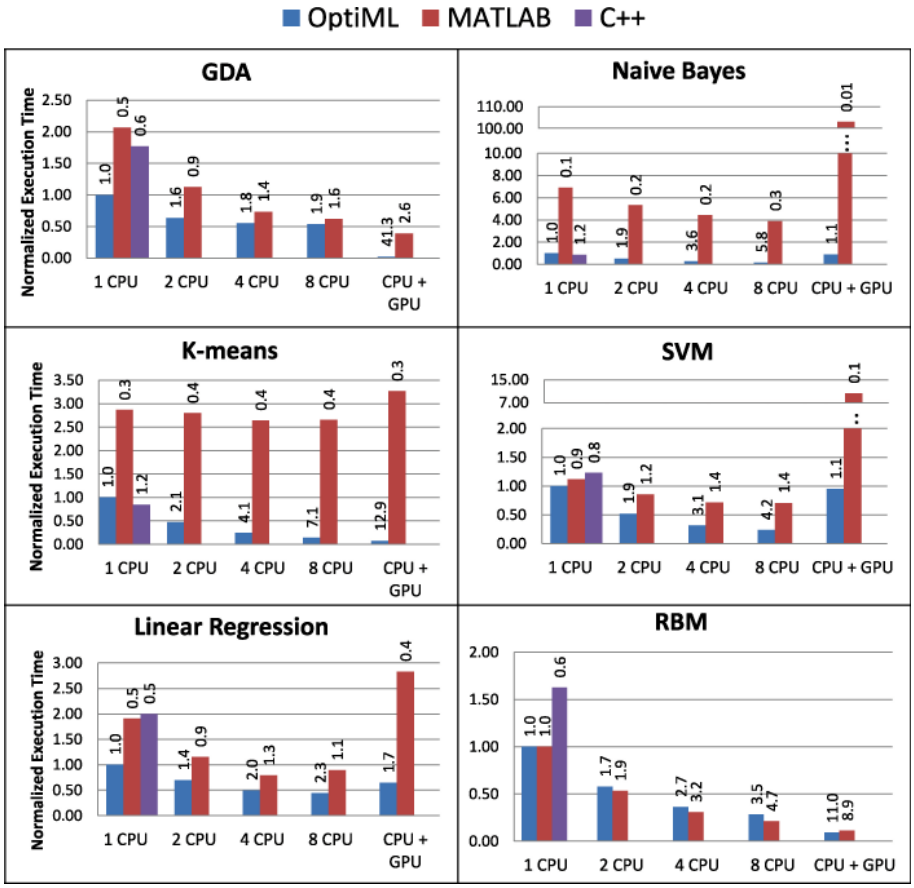

Fig. 3. Performance of OptiML applications. Speedup numbers relative to Delite 1 CPU are listed at the top of each bar.

OptiML. First, we compare the performance of six machine learning applications in Figure 3 [Brown et al. 2011]. For C++ we used the Armadillo linear algebra library [Sanderson 2006] and wrote hand-optimized imperative implementations. We also made a reasonable effort to optimize the MATLAB implementations, including vectorizing the code where possible. All OptiML implementations show better or comparable performance to the MATLAB and C++ library implementations. This is because Delite is able to generate code that is similar to the manually optimized implementation from the high-level DSL description. The OptiML compiler eliminates overhead by inlining a large number of small operations and uses op fusion to generate more efficient kernels, avoiding unnecessary intermediate allocations. Restricted Boltzmann Machine (RBM) is an application where MATLAB performs slightly better than OptiML. This is because RBM is dominated by BLAS [Lawson et al. 1979] operations that MATLAB offloads to an efficient BLAS library, and there is not much room for compiler optimizations in the application.

The reason some C++ implementations show better performance (e.g., *k*-means) is because the C++ version aggressively reuses allocated memory across loop iterations while the Delite version performs a new allocation each iteration. While this improves

sequential performance, it also prevents the code from being parallelized. The OptiML version, in contrast, scales well across multiple cores, and for k-means achieves an even greater 13x speedup by executing parallel ops on the GPU. Generating efficient GPU code for k-means requires Delite to perform multiple compiler transformations. The application as written in OptiML contains nested implicitly data-parallel operators that allocate results. While CPU code generators can easily perform the nested allocations locally within a thread, dynamic thread-local allocations are either not supported or not efficient on current GPUs. Therefore the Delite compiler transforms the outer parallel op into a sequential loop that can run on the CPU and generates the inner parallel ops as parallel CUDA kernels.

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

As we can see from the results, each application runs best either on multicore CPU or GPU depending on characteristics such as the arithmetic intensity or the size of the data-parallel operations. In addition, manually optimizing the application for each specific target device requires a great deal of effort and is not portable. Delite's capability of targeting different devices automatically from a single source addresses this issue, and the productivity benefit will become even more crucial as more and more heterogeneous devices are introduced.

OptiQL: TPC-H Query 1. Next we compare the performance of the benchmark TPC-H Query 1 implemented in OptiQL versus C++. The OptiQL implementation is very concise and high level (as illustrated in Listing 2) and requires 7x fewer lines of code than the hand-optimized C++ version. However, a library implementation of these OptiQL operations can run orders of magnitude slower than a fully optimized implementation. As written, the query allocates a new table (collection) for every operation and every row of the table is a heap-allocated object. However, Delite's loop fusion algorithm transforms the `Where-GroupBy-Select` chain into a single operation that computes the entire result with a single pass over the input dataset. Furthermore, Delite's array-of-struct to struct-of-array optimization transforms the internal layout of the tables from a single array of pointers to objects (row oriented) to a flat array of primitives per field (column oriented). Other key optimizations Delite performs that maximize the performance of this application include lifting the construction of constant objects out of the query, inlining all of the anonymous functions, and bit-packing primitive types (e.g., a `Pair[Char,Char]` is encoded as an `Int`). As Figure 4(a) illustrates, these optimizations allow OptiQL to achieve performance comparable to hand-optimized sequential C++ code. In addition the OptiQL version can be automatically parallelized to achieve nearly 5x speedup on eight threads, while parallelizing the C++ implementation would require significant refactoring and additional programmer effort. Delite can

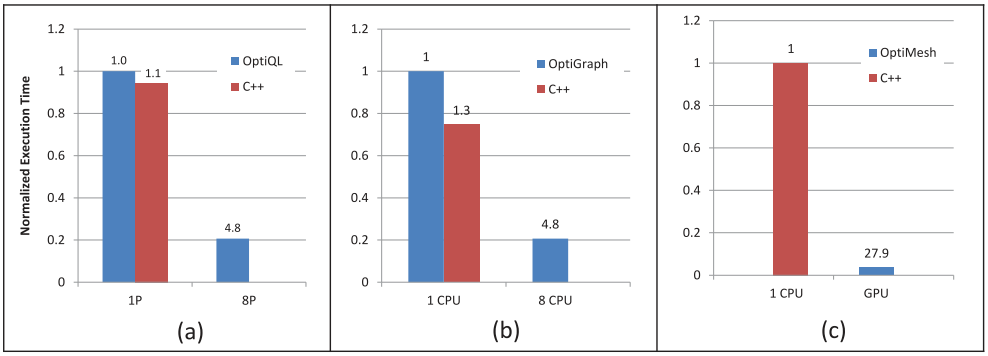

Fig. 4. Performance of OptiQL, OptiGraph, OptiMesh applications.

also generate a CUDA implementation of this query and execute it on a GPU, however, we found that the time required just to move the input data to the GPU was greater than the time required for the CPU to compute the result.

OptiGraph: PageRank. Figure 4(b) compares the performance of the PageRank algorithm [Page et al. 1999] implemented in OptiGraph to the C++ implementation on a uniform random graph of 8M nodes and 64M edges. As shown in Listing 3 each node calculates its PageRank value from the PageRank values of its neighbor nodes. Due to the nature of the graph data structure, there is not much locality in the neighbors of each node. Therefore, the application is dominated by the random memory accesses during node neighborhood exploration. This memory access pattern also generates cache conflicts and associated coherency traffic, which limits the scalability, especially when the graph size is small. The C++ implementation is about 30% faster than sequential OptiGraph because OptiGraph currently uses a back-end graph data structure that has more abstractions than the C++ implementation. For example, OptiGraph represents the neighbor node list as a nested `Array` where the index `i` of the outer `Array` contains the neighbor nodes of node `i`. On the other hand, C++ uses a flat `Array` using a CSR representation and therefore has less access overheads. However, this is not a fundamental limitation of OptiGraph and additional work to use a more optimized graph representation would eventually match the C++ performance.

OptiMesh: Shallow water simulation. Figure 4(c) compares the performance of shallow water simulation implemented in OptiMesh to a C++ implementation. Shallow water simulates the height and velocity of the free surface of the ocean over the spherical earth using an unstructured mesh. The application iterates through multiple time steps, and each time step executes a series of `ForeachReduce` operations on the mesh elements. Although the application **`for`** statements are not parallelizable because of the write conflicts between mesh elements, OptiMesh performs the loop coloring domain-specific transformation as shown in Listing 5 and generates one or more parallel `ForeachReduce` loops. The GPU can execute each of these loops very efficiently with a large number of threads, achieving nearly 28x speedup compared to the C++ implementation. This application heavily uses compute-intensive operations such as trigonometric functions, for which the GPU's specialized functional units provide significant speedup. This application also requires minimal communication between the CPU and GPU. The input mesh does not change once constructed and therefore only has to be copied to the GPU once at the beginning of execution.

8. DISCUSSION

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

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

While there is good opportunity for DSLs to become a practical alternative to low-level programming for high performance, there are also substantial challenges that remain. The proliferation of performance-oriented DSLs creates new problems. Users must choose between different, potentially incompatible DSLs to implement their application. To mitigate this, we need principled ways of inter-operating between performance-oriented DSLs. We have recently shown that Delite DSLs can be composed at different granularities by exploiting the common back-end framework [Sujeeth et al. 2013b]. However, this composition is limited (for DSLs with restricted semantics, we allow only pipeline composition), and does not address composing DSL front-ends (e.g., grammars or type systems).

Delite focuses on optimizing compilation for heterogeneous and parallel processors, but there has also been good progress in simplifying the development of the front-ends of DSL compilers (for example, via language workbenches such as Spoofax [Kats and Visser 2010] or extensible language frameworks such as SugarJ [Erdweg et al. 2011]). Combining these avenues of research is an exciting future direction. Many of these frameworks have also demonstrated the ability to automatically generate high-quality tool-chains for DSLs, such as custom IDE plugins. These tool-chains are critical for DSL adoption. However, targeting heterogeneous hardware adds new challenges for tool-chains, since bugs can manifest in many different layers or devices. One promising avenue for DSL tool-chains to go beyond their general-purpose counterparts is with

domain-specific debuggers. A graph analysis DSL, for example, could provide visualizations and step-through debugging on the state of the graph as a first-class primitive. Another open problem is heterogeneous resource scheduling and cost modeling, but there is longstanding work in multiple fields (including databases and operating systems) to draw from. Finally, there is potential for further specialization and cooperation throughout the stack; domain-specific compilers can be codesigned with domain-specific hardware to provide even more performance.

9. RELATED WORK

Our embedded DSL compilers build upon numerous previously published work in the areas of DSLs, extensible compilers, heterogeneous compilation, and parallel programming.

There is a rich history of DSL compilation in both the embedded and stand-alone contexts. Leijen and Meijer [1999] and Elliot et al. [2000] pioneered embedded compilation and used a simple image synthesis DSL as an example. Feldspar [Axelsson et al. 2011] is an embedded DSL that combines shallow and deep embedding of domain operations to generate high-performance code. For stand-alone DSL compilers, there has been considerable progress in the development of parallel and heterogeneous DSLs. Liszt [DeVito et al. 2011] and Green-Marl [Hong et al. 2012] are external DSLs for mesh-based PDE solvers and static graph analysis, respectively. Both of these DSLs target both multicore CPUs and GPUs and have been shown to outperform optimized C++ implementations. Diderot [Chiw et al. 2012] is a parallel DSL for image analysis that demonstrates good performance compared to hand-written C code using an optimized library. Püschel et al. [2005] show that domain-specific optimizations can yield substantial speedups when tuning code for particular Digital Signal Processors (DSPs). Our work aggregates many of the lessons and techniques from these previous DSL efforts and makes them easier to apply to new domains.

There has also been work on extensible compilation frameworks aimed towards making high-performance languages easier to build. Telescoping languages [Kennedy et al. 2005] automatically generate optimized domain-specific libraries. They share Delite's goal of incorporating domain-specific knowledge in compiler transformations. Delite compilers extend optimization to DSL data structures and are explicitly designed to generate parallel code for multiple heterogeneous back-ends. Delite also optimizes both the DSL and the program using it in a single step. Stratego [Bravenboer et al. 2008] is a language for program transformation using extensible rewrite rules. Stratego's organization also focuses on a reusable set of components, but targeted specifically to program transformation. Delite extends these principles to a more diverse set of components in order to target end-to-end high-performance program execution for DSLs.

Outside the context of DSLs, there have been efforts to compile high-level general-purpose languages to lower-level (usually device-specific) programming models. Mainland and Morrisett [2010] use type-directed techniques to compile an embedded array language, Nikola, from Haskell to CUDA. This approach suffers from the inability to overload some of Haskell's syntax (if-then-else expressions) which isn't an issue with our version of the Scala compiler. Nystrom et al. [2011] show a library-based approach to translating Scala programs to OpenCL code. This is largely achieved through Java bytecode translation. A similar approach is used by Lime [Auerbach et al. 2010] to compile high-level Java code to a hardware description language such as Verilog. Since the starting point of these compilers is the much lower-level bytecode or Java code (relative to DSL code), the opportunities for high-level optimizations are more limited.

Finally, there are high-level data-parallel programming models that provide implicit parallelization by providing the programmer with a data-parallel API that is transparently mapped to the underlying hardware. Recent work in this area includes Copperhead [Catanzaro et al. 2011], which automatically generates and executes CUDA code on a GPU from a data-parallel subset of Python. Array Building Blocks [Intel 2010] manages execution of data-parallel patterns across multiple processor cores and targets different hardware vector units from a single application source. DryadLINQ [Isard and Yu 2009] converts LINQ [Meijer et al. 2006] programs to execute using Dryad [Isard et al. 2007], which provides coarse-grained data-parallel execution over clusters. FlumeJava [Chambers et al. 2010] targets Google's MapReduce [Dean and Ghemawat 2004] from a Java library and fuses operations in the dataflow graph in order to generate an efficient pipeline of MapReduce operations. Our work builds on these previous publications and also allows for domain-specific optimizations, which is not possible with these other approaches. Furthermore, Delite DSLs provide a simpler interface for end-users than general-purpose frameworks.

10. CONCLUSION

In this article we showed that embedded compilers that target heterogeneous parallel architectures can be developed for multiple domains and provide both productivity and performance. We presented an overview of Delite, a compiler framework for embedded DSLs. Delite simplifies the process of developing DSL compilers by providing DSL building blocks, including domain-specific, parallel and generic optimizations. We described four DSLs (OptiML, OptiQL, OptiGraph, and OptiMesh) built using this framework and showed that the reuse of common components significantly reduced the amount of effort required to implement each new DSL. For the four DSLs we implemented, only nine parallel operators were required and seven of those were reused in at least two DSLs. Despite being developed within a common framework, our DSLs produce high-performing code (at worst within 30% of optimized C++) on multicore and GPU architectures. This work indicates that the Delite compiler framework is an effective way to turn embedded DSL programs into high-performance execution.

ACKNOWLEDGMENTS

We are grateful to the anonymous reviewers for their comments and suggestions.

REFERENCES

Apache. 2014. Hadoop. http://hadoop.apache.org/.

Joshua Auerbach, David F. Bacon, Perry Cheng, and Rodric Rabbah. 2010. Lime: A Java-compatible and synthesizable language for heterogeneous architectures. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'10)*. ACM Press, New York, 89–108.

Emil Axelsson, Koen Claessen, Mary Sheeran, Josef Svenningsson, David Engdal, and Anders Persson. 2011. The design and implementation of feldspar: An embedded language for digital signal processing. In *Proceedings of the 22nd International Conference on Implementation and Application of Functional Languages (IFL'10)*. Springer, 121–136.

Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. 2008. Stratego/xt 0.17. A language and toolset for program transformation. *Sci. Comput. Program.* 72, 1–2, 52–70.

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

Bryan Catanzaro, Michael Garland, and Kurt Keutzer. 2011. Copperhead: Compiling an embedded data parallel language. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'11)*. ACM Press, New York, 47–56.

Hassan Chafi, Zach Devito, Adriaan Moors, Tiark Rompf, Arvind K. Sujeeth, Pat Hanrahan, Martin Odersky, and Kunle Olukotun. 2010. Language virtualization for heterogeneous parallel computing (onward!). *ACM SIGPLAN Not.* 45, 10, 835–847.

Hassan Chafi, Arvind K. Sujeeth, Kevin J. Brown, Hyouk Joong Lee, Anand R. Atreya, and Kunle Olukotun. 2011. A domain-specific approach to heterogeneous parallelism. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'11).* 35–46.

Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. 2010. FlumeJava: Easy, efficient data-parallel pipelines. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'10).*

Charisee Chiw, Gordon Kindlmann, John Reppy, Lamont Samuels, and Nick Seltzer. 2012. Diderot: A parallel dsl for image analysis and visualization. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'12).* ACM Press, New York, 111–120.

Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design and Implementation (OSDI'04).* 137–150.

Zachary Devito, Niels Joubert, Francisco Palacios, Stephen Oakley, Montserrat Medina, Mike Barrientos, Erich Elsen, Frank Ham, Alex Aiken, Karthik Duraisamy, Eric Darve, Juan Alonso, and Pat Hanrahan. 2011. Liszt: A domain specific language for building portable mesh-based pde solvers. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11).*

Conal Elliott, Sigbjórn Finne, and Oege de Moor. 2000. Compiling embedded languages. In *Semantics, Applications, and Implementation of Program Generation*, Walid Taha, Ed., Lecture Notes in Computer Science, vol. 1924, Springer, 9–26.

Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. 2011. SugarJ: Library-based language extensibility. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'11).* ACM Press, 391–406.

Sungpack Hong, Hassan Chafi, Eric Sedlar, and Kunle Olukotun. 2012. Green-Marl: A dsl for easy and efficient graph analysis. In *Proceedings of the 17thInternational Conference on Architectural Support for Programming Languages and Operating Systems Systems (ASPLOS'12).* 349–362.

Intel. 2014. Cilk plus. http://software.intel.com/en-us/articles/intel-cilk-plus/.

Intel. 2010. Intel array building blocks. http://software.intel.com/en-us/articles/intel-array-building-blocks.

Intel. 2013. Intel math kernel library. http://software.intel.com/en-us/intel-mkl.

Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: Distributed data parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys'07).* ACM Press, New York, 59–72.

Michael Isard and Yuan Yu. 2009. Distributed data-parallel computing using a high-level programming language. In *Proceedings of the 35th SIGMOD International Conference on Management of Data (SIGMOD'09).* ACM Press, New York, 987–994.

Lennart C. L. Kats and Eelco Visser. 2010. The spoofax language workbench: Rules for declarative specification of languages and ides. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'10).* ACM Press, New York, 444–463.

Ken Kennedy, Bradley Broom, Arun Chauhan, Rob Fowler, John Garvin, Charles Koelbel, Cheryl Mccosh, and John Mellor-Crummey. 2005. Telescoping languages: A system for automatic generation of domain languages. *Proc. IEEE* 93, 3, 387–408.

The Khronos Group. 2014. OpenCL 1.0. http://www.khronos.org/opencl/.

Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'04).* 75–86.

Charles L. Lawson, Richard J. Hanson, David R. Kincaid, and Fred T. Krogh. 1979. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.* 5, 3, 308–323.

Daan Leijen and Erik Meijer. 1999. Domain specific embedded compilers. In *Proceedings of the 2nd Conference on Domain-Specific Languages (DSL'99).* ACM Press, New York, 109–122.

Geoffrey Mainland and Greg Morrisett. 2010. Nikola: Embedding compiled gpu functions in haskell. In *Proceedings of the 3rd ACM Haskell Symposium on Haskell (Haskell'10).* ACM Press, New York, 67–78.

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

MathWorks. 2014. Matlab. http://www.mathworks.com/products/matlab/.

Erik Meijer, Brian Beckman, and Gavin Bierman. 2006. LINQ: Reconciling object, relations and xml in the .net framework. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'06)*.

Adriaan Moors, Tiark Rompf, Philipp Haller, and Martin Odersky. 2012. Scala-virtualized. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM'12)*.

Nathaniel Nystrom, Derek White, and Kishen Das. 2011. Firepile: Run-time compilation for gpus in scala. In *Proceedings of the 10th ACM International Conference on Generative Programming and Component Engineering (GPCE'11)*. ACM Press, New York, 107–116.

Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. The pagerank citation ranking: Bringing order to the web. Tech. rep. 1999-66. Stanford Info Lab. http://ilpubs.stanford.edu:8090/422/.

Michael Paleczny, Christopher Vick, and Cliff Click. 2001. The java hotspot(tm) server compiler. In *Proceedings of the USENIX Java Virtual Machine Research and Technology Symposium*. 1–12.

Markus Püschel, Jose M. F. Moura, Jeremy R. Johnson, David Padua, Manuela M. Veloso, Bryan W. Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas Rizzolo. 2005. SPIRAL: Code generation for dsp transforms. *Proc. IEEE* 93, 2, 232–275.

Tiark Rompf and Martin Odersky. 2010. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled dsls. In *Proceedings of the 9th International Conference on Generative Programming and Component Engineering (GPCE'10)*. ACM Press, New York, 127–136.

Tiark Rompf, Arvind K. Sujeeth, Nada Amin, Kevin Brown, Vojin Jovanovic, Hyoukjoong Lee, Manohar Jonnalagedda, Kunle Olukotun, and Martin Odersky. 2013. Optimizing data structures in high-level programs: New directions for extensible compilers based on staging. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'13)*. 497–510.

Tiark Rompf, Arvind K. Sujeeth, Hyoukjoong Lee, Kevin J. Brown, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2011. Building-blocks for performance oriented dsls. In *Proceedings of the IFIP Working Conference on Domain-Specific Languages (DSL'11)*.

Conrad Sanderson. 2006. Armadillo: An open source c++ linear algebra library for fast prototyping and computationally intensive experiments. Tech. rep., NICTA. http://arma.sourceforge.net/armadillo_nicta_2010.pdf.

Arvind K. Sujeeth, Austin Gibbons, Kevin J. Brown, Hyoukjoong Lee, Tiark Rompf, Martin Odersky, and Kunle Olukotun. 2013a. Forge: Generating a high performance dsl implementation from a declarative specification. In *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences (GPCE'13)*.

Arvind K. Sujeeth, Hyoukjoong Lee, Kevin J. Brown, Tiark Rompf, Michael Wu, Anand R. Atreya, Martin Odersky, and Kunle Olukotun. 2011. OptiML: An implicitly parallel domain-specific language for machine learning. In *Proceedings of the 28th International Conference on Machine Learning (ICML'11)*.

Arvind K. Sujeeth, Tiark Rompf, Kevin J. Brown, Hyoukjoong Lee, Hassan Chafi, Victoria Popic, Michael Wu, Aleksander Prokopec, Vojin Jovanovic, Martin Odersky, and Kunle Olukotun. 2013b. Composition and reuse with compiled domain-specific languages. In *Proceedings of the European Conference on Object Oriented Programming (ECOOP'13)*.

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael Franklin, Scott Shenker, and Ion Stoica. 2011. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'11)*.

Received February 2013; revised July 2013; accepted September 2013