Composition and Reuse with Compiled Domain-Specific Languages

Arvind K. Sujeeth[1], Tiark Rompf[2,3], Kevin J. Brown[1], HyoukJoong Lee[1], Hassan Chafi[1,3], Victoria Popic[1], Michael Wu[1], Aleksandar Prokopec[2], Vojin Jovanovic[2], Martin Odersky[2], and Kunle Olukotun[1]

[1] Stanford University
{asujeeth, kjbrown, hyouklee, hchafi, viq, mikemwu, kunle}@stanford.edu
[2] École Polytechnique Fédérale de Lausanne (EPFL) {firstname.lastname}@epfl.ch
[3] Oracle Labs {firstname.lastname}@oracle.com

Abstract. Programmers who need high performance currently rely on low-level, architecture-specific programming models (e.g. OpenMP for CMPs, CUDA for GPUs, MPI for clusters). Performance optimization with these frameworks usually requires expertise in the specific programming model and a deep understanding of the target architecture. Domain-specific languages (DSLs) are a promising alternative, allowing compilers to map problem-specific abstractions directly to low-level architecture-specific programming models. However, developing DSLs is difficult, and using multiple DSLs together in a single application is even harder because existing compiled solutions do not compose together. In this paper, we present four new performance-oriented DSLs developed with Delite, an extensible DSL compilation framework. We demonstrate new techniques to compose compiled DSLs embedded in a common backend together in a single program and show that generic optimizations can be applied across the different DSL sections. Our new DSLs are implemented with a small number of reusable components (less than 9 parallel operators total) and still achieve performance up to 125x better than library implementations and at worst within 30% of optimized stand-alone DSLs. The DSLs retain good performance when composed together, and applying cross-DSL optimizations results in up to an additional 1.82x improvement.

1 Introduction

High-level general purpose languages focus on primitives for abstraction and composition that allow programmers to build large systems from relatively simple but versatile parts. However, these primitives do not usually expose the structure required for high performance on today's hardware, which is parallel and heterogeneous. Instead, programmers are forced to optimize performance-critical sections of their code using low-level, architecture-specific programming models (e.g. OpenMP, CUDA, MPI) in a time-consuming process. The optimized low-level code is harder to read and maintain, more likely to contain hard-to-diagnose bugs, and difficult to port to other platforms or hardware.

Domain-specific languages (DSLs) have been proposed as a solution that can provide productivity, performance, and portability for high-level programs in

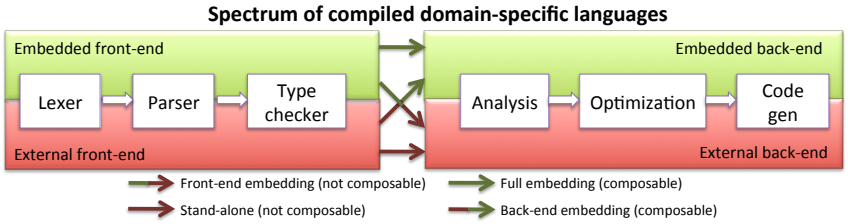


Fig. 1: Major phases in a typical compiler pipeline and possible organizations of compiled DSLs. Front-end embedding in a host language (and compiler) is common, but for composability, back-end embedding in a host compiler (i.e. building on top of an extensible compiler framework) is more important.

a specific domain [1]. DSL compilers reason about a program at the level of domain operations and so have much more semantic knowledge than a general purpose compiler. This semantic knowledge enables coarse-grain optimizations and the translation of domain operations to efficient back-end implementations. However, the limited scope of DSLs is simultaneously a stumbling block for widespread adoption. Many applications contain a mix of problems in different domains and developers need to be able to compose solutions together without sacrificing performance. In addition, DSLs need to interoperate with the outside world in order to enable developers to fall-back to general purpose languages for non performance-critical tasks.

Existing implementation choices for DSLs range from the internal (i.e. purely embedded) to external (i.e. stand-alone). Purely embedded DSLs are implemented as libraries in a flexible host language and emulate domain-specific syntax. The main benefit of internal DSLs is that they are easy to build and compose, since they can interoperate freely within the host language. However, they do not achieve high performance since the library implementation is essentially an interpreted DSL with high overhead and since general purpose host languages do not target heterogeneous hardware. On the other end of the spectrum, stand-alone DSLs are implemented with an entirely new compiler that performs both front-end tasks such as parsing and type checking as well as back-end tasks like optimization and code generation. Recent work has demonstrated that stand-alone DSLs can target multiple accelerators from a single source code and still achieve performance comparable to hand-optimized versions [2, 3]. The trade-off is that each DSL requires a huge amount of effort that is not easy to reuse in other domains, DSL authors must continuously invest new effort to target new hardware, and DSL programs do not easily interoperate with non-DSL code.

Another class of DSLs, compiled embedded, occupies a middle-ground between the internal and external approaches [4–7]. These DSLs embed their front-end in a host language like internal DSLs, but use compile- or run-time code generation to optimize the embedded code. Recently, these techniques have also been used to generate not only optimized host language code, but heterogeneous code, e.g. targeted at GPGPUs. The advantage of this approach is that the languages are easier to build than external versions and can provide better performance than internal versions. However, these DSLs still give up the composability of purely embedded DSLs and the syntactic freedom of stand-alone DSLs.

Like previous compiled embedded DSLs, our goal is to construct DSLs that resemble self-optimizing libraries with domain-specific front-ends. However, we propose that to build high-performance composable DSLs, *back-end embedding* is more important than the *front-end embedding* of traditional compiled embedded DSLs. Figure 1 illustrates this distinction. We define back-end embedding as a compiled DSL that inherits, or extends, the latter portion of the compiler pipeline from an existing compiler. Such a DSL can either programmatically extend the back-end compiler or pass programs in its intermediate representation (IR) to the back-end compiler. By embedding multiple DSLs in a common back-end, they can be compiled together and co-optimized. The fact that many compiled DSLs target C or LLVM instead of machine code and thus reuse instruction scheduling and register allocation can be seen as a crude example of back-end embedding. However, the abstraction level of C is already too low to compose DSLs in a way that allows high-level cross-DSL optimizations, parallelization, or heterogeneous code generation. Just as some general purpose languages are better suited for front-end embedding than others, not all compiler frameworks or target languages are equally suited for back-end embedding.

In this paper, we show that we can compose compiled DSLs embedded in a common, high-level backend and use them together in a single application. Our approach allows DSLs to build on top of one another and reuse important generic optimizations, rather than reinventing the wheel each time. Optimizations can even be applied across different DSL blocks within an application. The addition of composablity and re-use across compiled DSLs pushes them closer to libraries in terms of development effort and usage while still retaining the performance characteristics of stand-alone DSLs. In other words, we regain many of the benefits of purely embedded DSLs that were lost when adding compilation. We build on our previous work, Lightweight Modular Staging (LMS) and Delite [8–10], frameworks designed to make constructing individual compiled embedded DSLs easier. Previous work demonstrated good performance for OptiML, a DSL for machine learning [11]. However, it did not address how to compose different DSLs together, or show that similar performance and productivity gains could be obtained for different domains. We present new compiled embedded DSLs for data querying (OptiQL), collections (OptiCollections), graph analysis (OptiGraph), and mesh computation (OptiMesh). We show that the DSLs were easier to build than stand-alone counterparts, can achieve competitive performance, and can also be composed together in multiple ways.

Specifically, we make the following contributions:

- We implement four new DSLs for different domains and show that they can be implemented with a small number of reusable components and still achieve performance exceeding optimized libraries (up to 125x) and comparable to stand-alone DSLs (within 30%).
- We are the first to show both fine-grained and coarse-grained composition of high performance compiled DSLs.
- We demonstrate that different DSLs used in the same application can be co-optimized to additionally improve performance by up to 1.82x.

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

2 Background

In this paper, we investigate the problem of composing different DSLs embedded in a common back-end. The back-end we will use for our examples is Delite, but any back-end with similar structure could use the same techniques. Similarly, while Delite is typically targeted from embedded Scala front-ends, it could also be targeted from an external parser. Delite is essentially a Scala library that DSL authors can use to build an intermediate representation (IR), perform optimizations, and generate parallel code for multiple hardware targets. To illustrate at a high-level how Delite works, consider a simple example of a program using a Delite Vector DSL to add two vectors:

```
val (v1,v2) = (Vector.rand(1000), Vector.rand(1000))
val a = v1+v2
println(a)
```

The DSL implementation maps each language statement (`Vector.rand`, `+`, `println`) in this program to parallel operators (ops), each of which represents a specific parallel pattern (e.g. map, reduce, fork/join, sequential). These patterns are provided by Delite and extended by the DSL. The mapping is accomplished using a technique called Lightweight Modular Staging (LMS), a form of *staged metaprogramming* [12]. The essence is that the DSL implements operations on types wrapped in an abstract type constructor, `Rep[T]`. Type inference is used to hide this wrapped type from application code, as in the above snippet. Instead of immediate evaluation, DSL operations on `Rep[T]` construct an IR node representing the operation. For example, when the statement `v1+v2` is executed, it actually calls a DSL method that constructs a Delite IR node and returns a well-typed placeholder (`Rep[Vector[Double]]`) for the result. To ensure that all host language operations can be intercepted and lifted, we use a modified version of the Scala compiler for front-end compilation, *Scala-Virtualized* [13], that enables overloading even built-in Scala constructs such as `if (c) a else b`. DSLs can perform domain-specific optimizations by traversing and transforming the IR; Delite uses the same mechanisms to perform generic optimizations (such as dead code elimination) for all DSLs. Finally, after the full IR has been constructed and optimized, Delite generates code for each operation, based on its parallel pattern, to multiple targets (Scala, C++, CUDA). The resulting generated code is executed in a separate step to compute the final answer.

Figure 2 illustrates the key reusable components of the Delite compiler architecture: common IR nodes, data structures, parallel operators, built-in optimizations, traversals, transformers, and code generators. 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.

3 High-level Common Intermediate Representation

To achieve high performance for an application composed out of multiple DSLs, each DSL must provide competitive performance, not incur high overhead when crossing between DSL sections, and be co-optimizable. Moreover, for this sort of composition to be a practical approach, DSLs targeting narrow problem domains

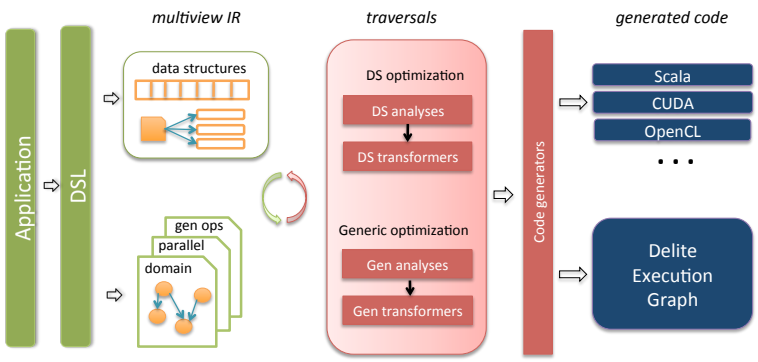


Fig. 2: Components of the Delite Framework. An application is written in a DSL, which is composed of data structures and structured computations represented as a multi-view IR. The IR is transformed by iterating over a set of traversals for both generic (Gen) and domain-specific (DS) optimizations. Once the IR is optimized, heterogeneous code generators emit specialized data structures and ops for each target along with the Delite Execution Graph (DEG) that encodes dependencies between computations.

and modern hardware should be as easy to construct as optimized libraries (or as close as possible). One way of achieving these goals is through a common intermediate representation containing reusable, high level computation and data primitives with a minimal set of restrictions that enable efficiency and optimization. In this section we describe how the Delite IR meets this criteria and propose a simple data exchange format for embedded DSLs.

3.1 Structured Computation

In order to optimize composed DSL blocks, they must share, or be transformable to, a common intermediate representation (IR) at some level. This level should retain enough high-level semantics to allow coarse-grained optimization and code generation; once an operation has been lowered to low-level primitives such as instructions over scalars, it is as difficult to target heterogeneous hardware as in a general purpose language. If DSLs do not share a common IR at some level, the best we can do is attempt to compose their generated code, which is a low-level and error-prone strategy that does not allow co-optimization.

We have proposed parallel patterns as a base IR that is well-suited to optimization and code generation for different DSLs and heterogeneous devices [8]. The new DSLs presented in this paper offer more evidence that parallel patterns are a strong choice of base IR. Previously, Delite supported `Sequential`, `Loop`, `Map`, `Reduce`, `ZipWith`, `Foreach`, and `Filter` patterns. To support the new DSLs, we added support for `GroupBy`, `Join` (generic inner join), `Sort`, `ForeachReduce` (foreach with global reductions), and `FlatMap` patterns. Using the patterns as a base IR allows us to perform *op fusion* automatically for each DSL, which is a key optimization when targeting data parallel hardware such as GPUs.

3.2 Structured Data

In the same way that a common, structured IR allows DSL operations to be optimized and code generated in a uniform way, a common data representation

is also required at some level. In addition to enabling the internal representation of DSL data structures to be reusable, a common representation facilitates communication between DSL blocks and enables pipeline optimizations; a DSL can directly consume the output of another DSL's block, so we can (for example) fuse the first DSL's operation that constructs the output with the second DSL's operation that consumes it.

We use structs of a fixed set of primitives as the common data representation. Delite DSLs still present domain-specific types to end users (e.g. `Vector`) and methods on instances of those types (e.g. `+`) get lifted into the IR. However, the back-end data structures must be implemented as structs (for example, `Vector` can be implemented with an `Int` length field and an `Array` data field). The fields currently allowed in a Delite `Struct` are: numerics (e.g. `Int`), `Boolean`, `String`, `Array`, `HashMap`, or other `Structs`. 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 (DFE) automatically [14]. 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 data structures by lifting the **new** keyword defining an anonymous class [13]. 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 to the record are type-checked by the Scala-Virtualized compiler. In this way, all DSL and user types can uniformly be lowered to one of the primitives, or a `Struct` of primitives.

3.3 Data Exchange Format

The final piece required for composability is the ability for application developers to convert from domain-specific data types to common data types in order to communicate between DSL blocks. This is necessary because we do not want to expose the internal representation of the domain-specific types to users.

A simple solution that we use is for DSL authors to optionally implement methods `to/from{Primitive}` between each DSL data type and a corresponding primitive type. For example, a DSL author would provide `toArray` and `fromArray` methods on DSL types where this makes sense (e.g. `Vector`), or `toInt` and `fromInt` for a type `NodeId`. These methods become part of the DSL specification and enable application developers to export and import DSL objects depending on their needs. For example, a graph could be exported using a snippet like:

```
// G: Graph
// returns an array of node ids and an array of edge ids
new Record { val nodes = G.nodes.map(node => node.Id.toInt).toArray
             val edges = G.edges.map(edge => edge.Id.toInt).toArray }
```

or just one of the properties of the graph could be exported using:

```
// G: Graph, numFriends: NodeProperty[Int]
// returns an array of ints corresponding to the number of friends of each node
```

```
G.nodes.map(node => numFriends(node)).toArray
```

We consider in the next section how to actually compose snippets together. Assuming this facility, though, importing is similar. For example, a vector could be constructed from the output of the previous snippet using:

```
// numFriends: Array[Int]
val v = Vector.fromArray(numFriends)
```

The key aspect of the data exchange format is that it should not prevent optimization across DSL blocks or impose substantial overhead to box and unbox the results. We handle this by implementing the `to/from` functions as either scalar conversions or loops in the common IR. Then, for example, a loop constructing an array will be automatically fused together with the loop consuming the array, resulting in no overhead while still ensuring safe encapsulation. The loop fusion algorithm is described in previous work [14].

4 Methods for Composing Compiled DSLs

In this section, we describe two ways of composing compiled DSLs that are embedded in a common back-end. The first way is to combine DSLs that are designed to work with other DSLs, i.e. DSLs that make an "open-world" assumption. The second way is to compose "closed-world" DSLs by compiling them in isolation, lowering them to a common, high-level representation, recombining them and then optimizing across them. Both methods rely on DSLs that share a common high-level intermediate representation as described in the previous section.

4.1 Open-world: Fine-grained Cooperative Composition

"Open-world" composition means essentially designing embedded DSLs that are meant to be layered or included by other embedded DSLs as modules. For this kind of composition to be feasible, all of the DSLs must be embedded in the same language and framework in both the front-end and the back-end.

Once embedded in a common host environment, DSL composition reduces to object composition in the host language. This is the classic "modularity" aspect of DSLs built using LMS [9]. For example, consider the following DSL definition for a toy DSL `Vectors` embedded in Scala:

```
trait Vectors extends Base with MathOps with ScalarOps with VectorOps
```

The DSL is composed of several traits that contain different DSL data types and operations. Each trait extends `Base` which contains common IR node definitions. A different DSL author can extend `Vectors` to create a new DSL, `Matrices`, simply by extending the relevant packages:

```
trait Matrices extends Vectors with MatrixOps with ExtVectorOps
```

Each of the mixed-in traits represents a collection of IR node definitions. Traits that contain optimizations and code generators can be extended in the same way. These traits define `Matrices` as a superset of `Vectors`, but `Matrices` users only interact with `Matrices` and do not need to be aware of the existence of `Vectors`. Furthermore, since the composition is via inheritance, the `Matrices` DSL can extend or overload operations on the `Vector` data type, e.g. inside `ExtVectorOps`.

Since `Vectors` is encapsulated as a separate object, it can be reused by multiple DSLs.

Open-world DSLs can also be composed by application developers. For example, suppose we also have a visualization DSL:

```
trait Viz extends Base with GraphicsOps with ChartOps with ImageOps
```

A Scala program can effectively construct a new DSL on the fly by mixing multiple embedded DSLs together:

```
trait MyApp extends Matrices with Viz {
  def main() {
    // assuming relevant DSL functions
    val m = Matrix.rand(100); display(m.toArray)
  }
}
```

In this example we make use of the data exchange format described in Section 3.3 in order to communicate data between the DSLs. When DSL users invoke a `Viz` operation, that operation will construct a `Viz` IR node. Note that after mix-in, the result is effectively a single DSL that extends common IR nodes; optimizations that operate on the generic IR can occur even between operations from different DSLs. This is analogous to libraries building upon other libraries, except that now optimizing compilation can also be inherited. DSLs can add analyses and transformations that are designed to be included by other DSLs. The trade-off is that DSL authors and application developers must be aware of the semantics they are composing and are responsible for ensuring that transformations and optimizations between DSLs retain their original semantics. Namespacing can also be a pitfall; DSL traits cannot have conflicting object or method names since they are mixed in to the same object. We avoid this problem by using conventions like DSL-specific prefixes (e.g. `viz_display`).

4.2 Closed-world: Coarse-grained Isolated Composition

As the previous section pointed out, there are issues with simply mixing embedded DSLs together. In particular, some DSLs require restricted semantics in order to perform domain-specific analyses and transformations that are required for correctness or performance. These "closed-world" DSLs are not designed to arbitrarily compose with other DSLs. Furthermore, any DSL with an external front-end is necessarily closed-world, since the parser only handles that DSL's grammar. However, it is still possible to compose closed-world DSLs that share a common back-end. Either they are implemented in the same language as the back-end and programmatically interface with it (as with Scala and Delite), or they target a serialized representation (such as source code) that is the input to a common back-end. This kind of coarse grained composition has three steps:

1. Independently parse each DSL block and apply domain-specific optimizations
2. Lower each block to the common high-level IR, composing all blocks into a single intermediate program
3. Optimize the combined IR and generate code

We discuss how we implemented these steps in Delite next.

Scopes: independent compilation DSLs that have an external front-end can be independently compiled by invoking the DSL parser on a string. However, in order to independently compile DSL blocks that are embedded in a host language, we need a coarse-grained execution block. We have modified the Scala-Virtualized compiler to add `Scopes` for this purpose. A `Scope` is a compiler-provided type that acts as a tag to encapsulate DSL blocks. For example, using the `Vectors` DSL from the previous section, we can instantiate a `Scope` as follows:

```
def Vectors[R](b: => R) = new Scope[VectorsApp, VectorsCompiler, R](b)
```

`VectorsApp` and `VectorsCompiler` are Scala traits that define the DSL interface and its implementation, respectively. The Scala-Virtualized compiler transforms function calls with return type `Scope` into an object that composes the two traits with the given block, making all members of the DSL interface available to the block's content. The implementation of the DSL interface remains hidden, however, to ensure safe encapsulation. The object's constructor then executes the block. The result is that each `Scope` is staged and optimized independently to construct the domain-specific IR.

Given the previous definition, a programmer can write a block of `Vectors` DSL code inside a Scala application, which then gets desugared, making all member definitions of `VectorsApp`, but not of `VectorsCompiler`, available to the `Scope`'s body:

```
Vectors {
  val v = Vector.rand(100)
  // ...
}
```

(a) Scala code with DSL scope

```
abstract class DSLprog extends VectorsApp {
  def apply = {
    val v = Vector.rand(100)
    // ...
  }
}
(new DSLprog with VectorsCompiler).result
```

(b) Scala code after desugaring

Lowering and composing The ability to compile blocks of DSL code into independent IRs is the first step, but in order to compose multiple blocks in a single application we still need a way to communicate across the blocks and a way to combine the IRs. Consider the following application snippet:

```
val r = Vectors {
  val (v1,v2) = (Vector.rand(100),Vector.rand(100))
  DRef(linreg(v1,v2).toArray) // return linear regression of v1, v2
}
Viz { display(r.get) }
```

We again use the `toArray` functionality to export the data from `Vectors` into a common format that `Viz` can handle. However, before we lower to a common representation, the type of the output of `Vectors` is a symbol with no relation to `Viz`. Therefore, we introduce the path independent type `DRef[T]` to abstract over the path dependent scope result. During staging, `r.get` returns a placeholder of type `Rep[DeliteArray[Double]]`. When the IRs are lowered and stitched together, the placeholder is translated to the concrete symbolic result of `linreg(v1,v2).toArray`. This mechanism is type-safe, preserves scope isolation, and does not occlude optimizations on the lowered IR.

After performing domain-specific optimizations and transformations, the IR for each scope is lowered to the base IR in a language-specific fashion. We use staging to perform this translation by extending our previous work on staged interpreters for program transformation [14] to support transforming IR nodes to an arbitrary target type. A `Transformer` is a generic `Traversal` that maps symbolic IR values (type `Exp[A]`) to values of type `Target[A]`, where `Target` is an abstract type constructor. During the traversal, a callback `transformStm` is invoked for each statement encountered.

The extended `Transformer` interface for cross-DSL transformation is:

```
trait Transformer extends Traversal {
  import IR._
  type Target[A]
  var subst = immutable.Map.empty[Exp[Any], Target[Any]]
  def apply[A](x: Exp[A]): Target[A] = ...    // lookup from subst
  override def traverseStm(stm: Stm): Unit =  // called during traversal
    subst += (stm.sym -> transformStm(stm))   //   update substitution with result
  def transformStm(stm: Stm): Target[Any]     // to be implemented in subclass
}
```

To transform from one IR to another, lower-level IR language, we instantiate a transformer with type `Target` set to the `Rep` type of the destination IR:

```
trait IRTransformer extends Transformer {
  val dst: IR.DST
  type Target[A] = IR.Rep[A]
  def transformStm(stm: Stm): Target[Any] =   // override to implement custom transform
    IR.mirror(stm.rhs, this)                  //   call default case
}
```

The type of the destination IR `dst: IR.DST` is constrained by the source IR to handle all defined IR nodes. This enables implementing a default case for the transformation (**def** `mirror`), which maps each source IR node to the corresponding smart constructor in the destination IR.

Taking the Vectors DSL as an example, we define:

```
trait Vectors extends Base {
  def vector_zeros(n: Rep[Int]): Rep[Vector[Double]]
}
trait VectorExp extends Vectors with BaseExp {
  type DST <: Vectors
  case class VectorZeros(n: Rep[Int]) extends Def[Vector[Double]]
  def vector_zeros(n: Rep[Int]): Rep[Vector[Double]] = VectorZeros(n)
  def mirror[A:Manifest](e: Def[A], f: IRTransformer): f.dst.Rep[A] = {
    case VectorZeros(n) => f.dst.vector_zeros(f(n))
    ...
  }
}
```

The use of Scala's dependent method types `f.dst.Rep[A]` and the upper-bounded abstract type `DST <: Vectors` ensure type safety when specifying transformations. Note that the internal representation of the destination IR is not exposed, only its abstract `Rep` interface. This enables, for example, interfacing with a textual code generator that defines `Rep[T] = String`. Perhaps more impor-

tantly, this enables programmatic lowering transforms by implementing a smart constructor (e.g. `vector_zeros`) to expand into a lower-level representation using arrays instead of constructing an IR node that directly represents the high-level operation.

An alternative to using staged interpreters is to simply generate code to a high-level intermediate language that maps to the common IR. We implemented this by generating code to the "Delite IL", a low-level API around Delite ops, that is itself staged to build the Delite IR. For example, a `Reduce` op in the original application would be code generated to call the method

```
reduce[A](size: Rep[Int], func: Rep[Int] => Rep[A], cond: List[Rep[Int]] => Rep[Boolean],
          zero: => Rep[A], rFunc: (Rep[A],Rep[A]) => Rep[A])
```

in the Delite IL.

The staged interpreter transformation and the code generation to the Delite IL perform the same operation and both use staging to build the common IR. The staged interpreter translation is type-safe and goes through the heap, while the Delite IL goes through the file system and is only type-checked when the resulting program is (re-)staged. On the other hand, for expert programmers, a simplified version of the Delite IL may be desirable to target directly.

Cross DSL optimization After the IRs have been composed, we apply all of our generic optimizations on the base IR (i.e. parallel patterns). Like in the open-world case, we can now apply optimizations such as fusion, common subexpression elimination (CSE), dead code elimination (DCE), dead field elimination (DFE), and AoS to SoA conversion across DSL snippets. Since the base IR still represents high level computation, these generic optimizations still have much higher potential impact than their analogs in a general purpose compiler. Fusion across DSL snippets is especially useful, since it can eliminate the overhead of boxing and unboxing the inputs and outputs to DSL blocks using the `to/from{Primitive}` data exchange format. The usefulness of applying these optimizations on composed blocks instead of only on individual blocks is evaluated in Section 6. Note that the cross-DSL optimizations fall out for free after composing the DSLs; we do not have to specifically design new cross-DSL optimizations.

4.3 Interoperating with non-DSL code

The previous section showed how we can compose coarse-grain compiled DSL blocks within a single application. However, it is also interesting to consider how we can compose DSL blocks with arbitrary host language code. We can again use `Scope`, but with a different implementation trait, to accomplish this. Consider the following, slightly modified definition of the `Vectors` scope:

```
def Vectors[R](b: => R) = new Scope[VectorsApp, VectorsExecutor, R](b)
```

Whereas previously the `Vectors` scope simply compiled the input block, the trait `VectorsExecutor` both compiles and executes it, returning a Scala object as a result of the execution. `VectorsExecutor` can be implemented by programmatically invoking the common back-end on the lowered IR immediately. This enables us to use compiled embedded DSLs within ordinary Scala programs:

```
def main(args: Array[String]) {
  foo() // Scala code
```

```
    Vectors { val v = Vector.ones(5); v.pprint }
    // more Scala code ..
  }
```

This facility is the same that is required to enable interactive usage of DSLs using the REPL of the host language, which is especially useful for debugging. For example, we can use the new `Vectors` scope to execute DSL statements inside the Scala-Virtualized REPL:

```
scala> Vectors { val v = Vector.ones(5); v.pprint }
[ 1 1 1 1 1 ]
```

5 New Compiled Embedded DSL Implementations

We implemented four new high performance DSLs embedded inside Scala and Delite. In this section, we briefly describe each DSL and show how their implementation was simplified by reusing common components. The four new DSLs are OptiQL, a DSL for data querying, OptiCollections, an optimized subset of the Scala collections library, OptiGraph, a DSL for graph analysis based on Green-Marl [3] and OptiMesh, a DSL for mesh computations based on Liszt [2]. Despite being embedded in both a host language and common back-end, the DSLs cover a diverse set of domains with different requirements and support non-trivial optimizations.

5.1 OptiQL

OptiQL is a DSL for data querying of in-memory collections, and is heavily inspired by LINQ [15], 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 1.1 shows an example snippet of OptiQL code that expresses a query similar to Q1 in the TPC-H benchmark. The query first excludes any line item with a ship date that occurs after the specified date. It then groups each line item by its status. Finally, it summarizes each group by aggregating the group's line items and constructs a final result per group.

Since OptiQL is SQL-like, it is concise and has a small learning curve for many developers. However, unoptimized performance is poor. Operations always semantically produce a new result, and since the in-memory collections are typically very large, cache locality is poor and allocations are expensive. OptiQL uses compilation to aggressively optimize queries. Operations are fused into a single loop over the dataset wherever possible, eliminating temporary allocations, and datasets are internally allocated in a column-oriented manner, allowing OptiQL to avoid allocating columns that are not used in a given query. 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.

5.2 OptiCollections

Where OptiQL provides a SQL-like interface, OptiCollections is an example of applying the underlying optimization and compilation techniques to the Scala

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

Listing 1.1: OptiQL: TPC-H Query 1 benchmark

```
1  val sourcedests = pagelinks flatMap { l =>
2    val sd = l.split(":")
3    val source = Long.parseLong(sd(0))
4    val dests = sd(1).trim.split(" ")
5    dests.map(d => (Integer.parseInt(d), source))
6  }
7  val inverted = sourcedests groupBy (x => x._1)
```

Listing 1.2: OptiCollections: reverse web-links benchmark

collections library. The Scala collections library provides several key generic data types (e.g. `List`) with rich APIs that include expressive functional operators such as `flatMap` and `partition`. The library enables writing succinct and powerful programs, but can also suffer from overheads associated with high-level, functional programs (especially the creation of many intermediate objects). OptiCollections uses the exact same collections API, but uses Delite to generate optimized, low-level code. Most of the infrastructure is shared with OptiQL. The prototype version of OptiCollections supports staged versions of Scala's `Array` and `HashMap`. Listing 1.2 shows an OptiCollections application that consumes a list of web pages and their outgoing links and outputs a list of web pages and the set of incoming links for each of the pages (i.e. finds the reverse web-links). In the first step, the `flatMap` operation maps each page to pairs of an outgoing link and the page. The `groupBy` operation then groups the pairs by their outgoing link, yielding a `HashMap` of pages, each paired with the collection of web pages that link to it.

The example has the same syntax as the corresponding Scala collections version. A key benefit of developing OptiCollections is that it can be mixed in to enrich other DSLs with a range of collection types and operations on those types. It can also be used as a transparent, drop-in replacement for existing Scala programs using collections and provide improved performance.

5.3 OptiGraph

OptiGraph is a DSL for static graph analysis based on the Green-Marl DSL [3]. 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

```
1 val PR = NodeProperty[Double](G)
2 for (t <- G.Nodes) {
3   val rank = (1-d) / N + d*
4     Sum(t.InNbrs){w => PR(w)/w.OutDegree}
5   diff += abs(rank - PR(t))
6   PR <= (t,rank)
7 }
```

(a) OptiGraph: PageRank

```
1 N_P<Double> PR;
2 Foreach (t: G.Nodes) {
3   Double rank = (1-d) / N + d*
4     Sum(w:t.InNbrs){w.PR/w.OutDegree()};
5   diff += | rank - t.PR |;
6   t.PR <= rank @ t;
7 }
```

(b) Green-Marl: PageRank

Fig. 3: The major portion of the PageRank algorithm implemented in both OptiGraph and Green-Marl. OptiGraph is derived from Green-Marl, but required small syntactic changes in order to be embedded in Scala.

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. An important feature of OptiGraph is also its built-in support for bulk synchronous consistency via *deferred assignments*.

Figure 3 shows the parallel loop of the PageRank algorithm [16] written in both OptiGraph and Green-Marl. 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 page-rank of the nodes associated with its incoming edges (`InNbrs`). OptiGraph's syntax is slightly different since it is embedded in Scala and must be legal Scala syntax. However, the differences are small and the OptiGraph code is not more verbose than the Green-Marl version. In the snippet, `PR` is a node property associating a page-rank value with every node 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` Delite op implicitly injects the necessary synchronization into the **`for`** loop.

5.4 OptiMesh

OptiMesh is an implementation of Liszt on Delite. Liszt is a DSL for mesh-based partial differential equation (PDE) solvers [2]. 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 1.3 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.

```
1  for (edge <- edges(mesh)) {
2    val flux = flux_calc(edge)
3    val v0 = head(edge)
4    val v1 = tail(edge)
5    Flux(v0) += flux // possible write conflicts!
6    Flux(v1) -= flux
7  }
```

Listing 1.3: OptiMesh: simple flux computation

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

Fig. 4: Sharing of DSL operations and optimizations.

OptiMesh solves this challenge by implementing the same domain-specific transformation as Liszt. First, the OptiMesh program is symbolically evaluated with a real mesh input to obtain a stencil of mesh accesses in the program. After the stencil is collected, an interference graph is built and disjoint loops are constructed using coloring. OptiMesh uses a Delite `Transformer` to simplify this implementation – the transformation is less than 100 lines of code. OptiMesh was the only new DSL we implemented that required a domain-specific transformation; the others were able to produce high performance results just by reusing generic optimizations and parallel code generation.

5.5 Reuse

By embedding the front-ends of our DSLs in Scala, we did not have to implement any lexing, parsing, or type checking. As we showed in the OptiGraph vs. Green-Marl example, the syntactic difference compared to a stand-alone DSL can still be relatively small. By embedding our DSL back-ends in Delite, each DSL was able to reuse parallel patterns, generic optimizations, common library functionality (e.g. math operators), and code generators for free. One important characteristic of the embedded approach is that when a feature (e.g. a parallel pattern) is added to support a new DSL, it can be reused by all subsequent DSLs. For example, we added the `ForeachReduce` pattern for OptiGraph, but it is also used in OptiMesh.

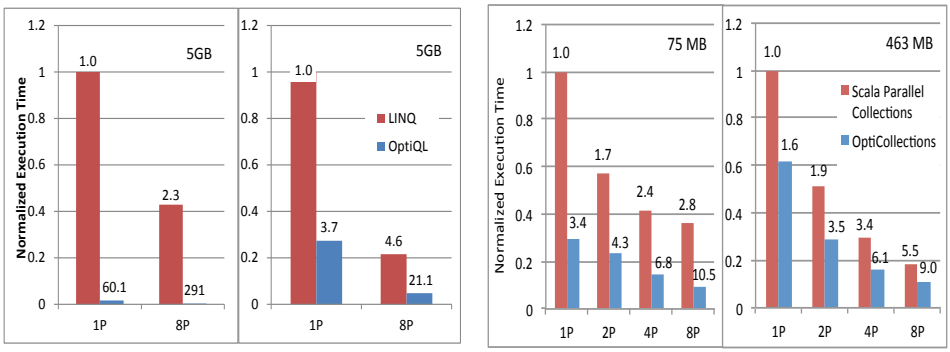
Figure 4 summarizes the characteristics and reuse of the new DSLs introduced in this section. The DSLs inherit most of their functionality from Delite, in the form of a small set of reused parallel patterns and generic optimizations. The DSLs use just 9 Delite ops total; seven ops (77.7%) were used in 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.

6 Case Studies

We present four case studies to evaluate our new DSLs. The first two case studies compare individual DSL performance against existing alternative programming environments and the second two evaluate applications composing the DSLs in the two ways described in this paper (open and closed world). OptiQL is compared to LINQ [15] and OptiCollections to Scala Collections [17]. LINQ and Scala Collections are optimized libraries running on managed platforms (C#/CLR and Scala/JVM). We compare OptiMesh and OptiGraph to Liszt [2] and Green-Marl [3] respectively, stand-alone DSLs designed for high performance on heterogeneous hardware. Liszt and Green-Marl both generate and execute C++ code and have been shown to outperform hand-optimized C++ for the applications shown in this section. Each Delite DSL generated Scala code for the CPU and CUDA code for the GPU. For composability, we compare against an analogous Scala library implementation of each application when using a combination of DSLs to solve larger application problems.

All of our experiments were performed on a Dell Precision T7500n with two quad-core Xeon 2.67GHz 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. We ran each application ten times (to warm up the JIT) and report the average of the last 5 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.

6.1 Compiled Embedded vs. Optimized Library



(a) OptiQL: TPC-H Q1 (left) and Q2 (right) (b) OptiCollections: web-link benchmark

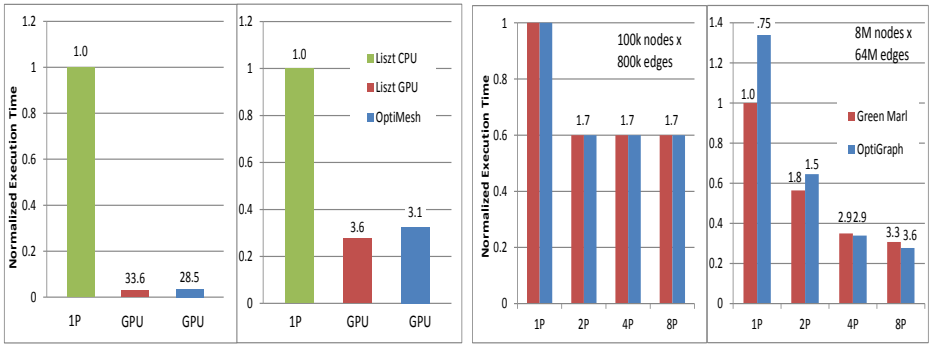
Fig. 5: Normalized execution time of applications written in OptiQL and OptiCollections. Speedup numbers are reported on top of each bar.

OptiQL: TPC-H queries 1 & 2 Figure 5(a) compares the performance of queries 1 & 2 of the popular TPC-H benchmark suite on OptiQL vs. LINQ. Without any optimizations, OptiQL performance for the queries (not shown) is comparable

to LINQ performance. However, such library implementations of the operations suffer from substantial performance penalties compared to an optimized and compiled implementation. Naïvely, the query allocates a new table (collection) for each operation and every element of those tables is a heap-allocated object to represent the row. The two most powerful optimizations we perform are converting to an SoA representation and fusing across `GroupBy` operations to create a single (nested) loop to perform the query. Transforming to an SoA representation allows OptiQL to eliminate all object allocations for Q1 since all of the fields accessed by Q1 are JVM primitive types, resulting in a data layout consisting entirely of JVM primitive arrays. All together these optimizations provide 125x speedup over parallel LINQ for Q1 and 4.5x for Q2.

OptiCollections: Reverse web-link graph Figure 5(b) shows the result for the reverse web-link application discussed in Section 5.2 running on OptiCollections compared to Scala Parallel Collections. Scala Parallel Collections is implemented using Doug Lea's highly optimized fork/join pool with work stealing [18]. The OptiCollections version is significantly faster at all thread counts and scales better with larger datasets. The improvement is due to staged compilation, which helps in two ways. First, OptiCollections generates statically parallelized code. Unlike Scala collections, functions are inlined directly to avoid indirection. On the larger dataset, this does not matter as much, but on the smaller dataset the Scala collections implementation has higher overhead which results in worse scaling. Second, the OptiCollections implementation benefits from fusion and from transparently mapping `(Int,Int)` tuples to `Long`s in the back-end. These optimizations greatly reduce the number of heap allocations in the parallel operations, which improves both scalar performance and scalability.

6.2 Compiled Embedded vs. External DSL



(a) OptiMesh: Shallow Water (left) Scalar Convection (right)

(b) OptiGraph: PageRank

Fig. 6: Normalized execution time of applications written in OptiMesh and OptiGraph. Speedup numbers are reported on top of each bar.

Productivity First, we consider the programming effort required to build OptiMesh and OptiGraph compared to the stand-alone versions they were based

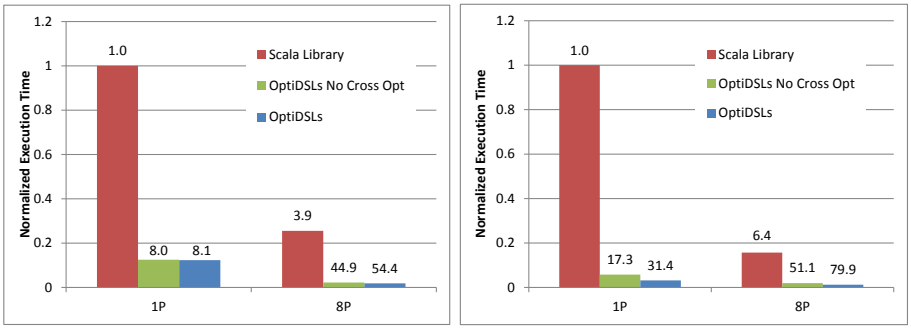
on. Each Delite DSL compiler took approximately 3 months by a single graduate student to build using Delite. OptiMesh (≈5k loc) and OptiGraph (≈2.5k loc) were developed by 1st year Ph.D. students with no prior experience with Scala or Delite. In comparison, Liszt (≈25k loc Scala/C++) took a group of 2-3 compiler experts approximately one year to develop and Green-Marl (≈30k loc mostly C++) took a single expert approximately 6 months. As discussed in Section 5.5, most of the code reduction is due to reuse from being both front and back-end embedded (in Scala and Delite respectively). The Delite DSLs did not need to implement custom parsers, type checkers, base IRs, schedulers, or code generators. Similarly, while OptiMesh and OptiGraph do not implement all of the optimizations performed by Liszt and Green-Marl, they inherit other Delite optimizations (e.g. fusion) for free. For comparison, the Delite framework is ≈11k and LMS is ≈7k lines of Scala code. It is interesting to note that ≈4k lines of the Liszt code is for Pthreads and CUDA parallelization; a major benefit of using Delite is that parallelization for multiple targets is handled by the framework.

OptiMesh: Shallow water simulation & Scalar convection Figure 6(a) shows the performance of OptiMesh and Liszt on two scientific applications. Each application consists of a series of `ForeachReduce` operations surrounded by iterative control-flow to step the time variable of the PDE. It is well-suited to GPU execution as mesh sizes are typically large and the cost of copying the input data to the GPU is small compared to the amount of computation required. However, the original applications around which the Liszt language was designed were only implemented using MPI. Liszt added GPU code generation and demonstrated significant speedups compared to the CPU version. For both OptiMesh applications, Delite is able to generate and execute a CUDA kernel for each colored foreach loop and achieve performance comparable to the Liszt GPU implementation. Liszt's (and OptiMesh's) ability to generate both CPU and GPU implementations from a single application source illustrates the benefit of using DSLs as opposed to libraries that only target single platforms.

OptiGraph: PageRank Figure 6(b) compares the performance of the PageRank algorithm [16] implemented in OptiGraph to the Green-Marl implementation on two different uniform random graphs of sizes 100k nodes by 800k edges and 8M nodes by 64M edges, respectively. This benchmark is dominated by the random memory accesses during node neighborhood exploration. Since OptiGraph's memory access patterns and the memory layout of its back-end data structures are similar to those of Green-Marl, OptiGraph's sequential performance and scalability across multiple processors is close to that of Green-Marl. Although the smaller graph fits entirely in cache, the parallel performance is limited by cache conflicts when accessing neighbors and the associated coherency traffic. The sequential difference between OptiGraph and Green-Marl in the larger graph can be attributed to the difference between executing Scala generated code vs. C++. However, as we increase the number of the cores, the benchmark becomes increasingly memory-bound and the JVM overhead becomes negligible.

```
1  def valueIteration(actionResults: Rep[Map[Action, (Matrix[Double],Vector[Double])]],
2          initValue: Rep[Vector[Double]], discountFactor: Rep[Double], tolerance: Rep[Double]) = {
3    val bestActions = Seq[Action](initValue.length)
4    var (value, delta) = (initValue, Double.MaxValue)
5    while (abs(delta) > tolerance) {
6      val newValue = Vector.fill(0,value.length) { i =>
7        val allValues = actionResults map { case (action,(prob,cost)) =>
8          (action, (prob(i) * value(i) * discountFactor + cost(i)).sum) }
9        val minActionValue = allValues reduce { case ((act1,v1),(act2,v2)) =>
10         if (v1 <= v2) (act1,v1) else (act2,v2) }
11       bestActions(i) = minActionValue.key; minActionValue.value }
12     delta = diff(newValue, value); value = newValue }
13   (value, bestActions) }
```

Listing 1.4: Value Iteration of a Markov Decision Process



(a) small action set (b) large action set

Fig. 7: Normalized execution time of value iteration of a Markov decision process. Performance is shown both with and without cross-DSL optimization. Speedup numbers are reported on top of each bar.

6.3 Open-world composability

We illustrate open-world composability by implementing the value iteration algorithm for a Markov decision process (MDP), shown in Listing 1.4, using two different DSLs, OptiLA and OptiCollections. OptiLA is a DSL for linear algebra providing `Matrix` and `Vector` operations, and is specifically designed to be included by other DSLs by making no closed-world assumptions. OptiLA is a refactored portion of OptiML [11], a machine learning DSL designed for statistical inference problems expressed with vectors, matrices, and graphs. Although OptiML originally contained its own linear algebra components, we have found that several DSLs need some linear algebra capability, so we extracted OptiLA and modified OptiML to extend it using the techniques in Section 4.1. OptiCollections was also designed to be included by other DSLs and applications, as described in Section 5.2.

The algorithm uses an OptiCollections `Map` to associate each user-defined `Action` with a probability density `Matrix` and cost `Vector`. OptiLA operations (line 8) compute the value for the next time step based on an action, and OptiCollections operations apply the value propagation to every action and then find

the minimal value over all actions. This process is repeated until the minimizing value converges.

Figure 7 shows the performance for this application implemented using Scala Parallel Collections compared to using OptiLA and OptiCollections. For both datasets, the OptiDSL version shows significant speedup over the library implementation as well as improved parallel performance, due mainly to two key optimizations: loop fusion and AoS to SoA transformation. The latter transformation eliminates all of the tuples and class wrappers in lines 7-10, leaving only nested arrays. The OptiDSLs "No Cross Opt" bar simulates the behavior of compiling the code snippets for each DSL independently and then combining the resulting optimized code snippets together using some form of foreign function interface. Therefore this version does not include SoA transformations or fusion across DSLs, but does still fuse operations fully contained within a single DSL, most notably the OptiLA code snippet on line 8 of Listing 1.4. Figure 7(a) shows only very modest speedups after adding DSL cross-optimization. This is because the majority of the execution time is spent within the OptiLA code snippet, and so only fusion within OptiLA was necessary to maximize performance. Figure 7(b), however, shows the behavior for different data dimensions. In this case the total execution time spent within the OptiLA section is small, making fusion across the nested OptiCollections/OptiLA operations critical to maximizing performance.

Overall this case study shows that while sometimes applications can achieve good performance by simply offloading key pieces of computation to a highly optimized implementation (applications that call BLAS libraries are a classic example of this), other applications and even the same application with a different dataset require the compiler to reason about all of the pieces of computation together at a high level and be capable of performing optimizations across them in order to maximize performance.

6.4 Closed-world composability

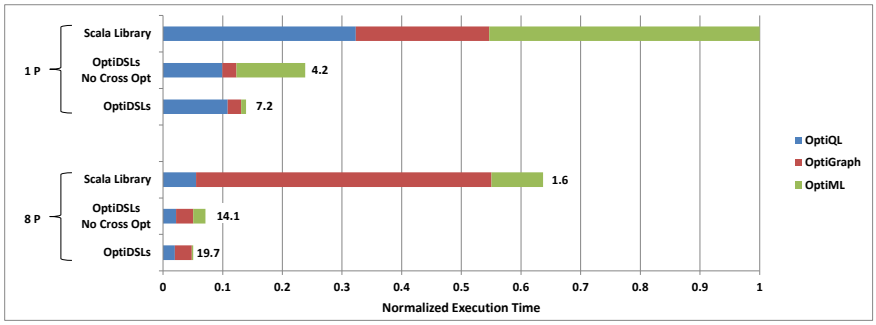


Fig. 8: Normalized execution time of Twitter data analysis. Performance is shown for each DSL section with and without cross-DSL optimization. Speedup numbers are reported on top of each bar.

In this example, we combine OptiQL, OptiGraph, and OptiML using the closed-world composition strategy discussed in Section 4.2. We use the three

```
type Tweet = Record { val time: String; val fromId: Int; val toId: Int;
                      val lang: String; val text: String; val RT: Boolean }
val q = OptiQL {
  //tweets: Table[Tweet]
  val reTweets = tweets.Where(t => t.lang == "en"
                 && Date(t.time) > Date("2008-01-01")  && t.RT)
  val allTweets = tweets.Where(t => t.lang == "en")
  DRef((reTweets.toArray, allTweets.toArray))
}
val g = OptiGraph {
  val in = q.get
  val G = Graph.fromArray(in._1.map(t => (t.fromId, t.toId)))
  val (LCC, RT) = (NodeProperty[Double](G), NodeProperty[Double](G))
  Foreach(G.Nodes) { s =>
    // count number of nodes connected in a triangle (omitted)
    if (total < 1) LCC(s) = 0 else LCC(s) = triangles.toDouble / total
    RT(s) = G.InNbrs(s).length
  }
  DRef((LCC.toArray, RT.toArray, in._1, in._2))
}
val r = OptiML {
  val in = g.get
  val scaledRT = norm(log((Vector.fromArray(in._2) + 1)))
  val X = Matrix(Vector.ones(in._1.length), Vector.fromArray(in._1))
  val theta = (X*X.t).inv * (X*scaledRT)  // unweighted linear regression
  // compute statistics on tweets (omitted)
}
```

Listing 1.5: Twitter Graph Analysis using multiple DSLs

DSLs together to implement a data analysis application on a Twitter dataset used in [19]. A truncated version of the application code is shown in Listing 1.5. The idea is to compute statistics related to the distribution of all tweets, of retweets (tweets that have been repeated by another user), and of the relationship between user connectivity and the number of times they have been retweeted.

The application follows a typical data analytic pipeline. It first loads data from a log file containing tweets with several attributes (sender, date, text, etc.). It then queries the dataset to extract relevant information using an OptiQL `Where` statement. The filtered data is passed on to OptiGraph and OptiML which both analyze it and compute statistics. OptiGraph builds a graph where nodes are users and edges are retweets and computes the LCC (local clustering coefficient) and retweet counts for each user. The LCC is a loose approximation of the importance of a particular user. The OptiML section fits the filtered tweet data from OptiQL to a normal distribution and also runs a simple unweighted linear regression on the LCC and retweet counts that the OptiGraph code computed. The final results of the application are the distribution and regression coefficients.

Figure 8 shows the performance of this application implemented with Scala Parallel Collections compared to the Delite DSLs with and without cross-DSL optimization. The graph is broken down by the time spent in each DSL section (or for the library version, in the corresponding Scala code). Since we are us-

ing the closed world model, each DSL is independently staged, lowered to the common Delite representation, and re-compiled. The application code is still a single program and the DSLs pass data in memory without any boxing overhead. Therefore, in contrast to using stand-alone compiled DSLs for each computation, we incur no serialization overhead to pipe data from one DSL snippet to the other. The Scala library version, which is also a single application, is approximately 5x slower than the non-cross-optimized DSL version on 1 thread and 10x slower on 8 threads. The speedup is due to optimizations performed by Delite across all DSLs. The OptiQL code benefits from filter fusion as well as lifting the construction of a `Date` object outside of the filter loop, which demonstrates the high-level code motion that is possible with more semantic information (the `Date` comparison is a virtual method call for Scala, so it does not understand that the object is constant in the loop). The OptiGraph version is faster than the corresponding library snippet mainly due to compiling away abstractions (we use only primitive operations on arrays in the generated code compared to Scala's `ArrayBuffer`, which has run-time overhead). The OptiGraph code is also the least scalable, since this particular graph is highly skewed to a few dominant nodes and the graph traversal becomes very irregular. Due to the additional overhead of the library version, this effect is more pronounced there. Finally, the OptiML version is faster mainly because of loop fusion and CSE across multiple linear algebra operations.

Co-optimizing the DSLs, which is enabled by composing them, produces further opportunities. The Delite compiler recognizes that OptiGraph and OptiML together require only 4 fields per tweet of the original 6 (OptiGraph uses "toId" and "fromId" and OptiML uses "text" and "hour"). The remaining fields are DFE'd by performing SoA transformation on the filter output and eliminating arrays that are not consumed later. The other major cross DSL optimization we perform is to fuse the filter from OptiQL with their consumers in OptiGraph and OptiML. Note that the fusion algorithm is strictly data dependent; the OptiML snippet and OptiQL snippets are syntactically far apart, but can still be fused. This example also shows that since the DSL blocks are composed into a single IR, we can fuse across multiple scope boundaries when co-optimizing. All together, cross optimizations result in an extra 1.72x sequential speedup over the composed Delite DSL version without cross optimizations.

7 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. Elliot et al. [4] pioneered embedded compilation and used a simple image synthesis DSL as an example. Feldspar [20] 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 [2] and Green-Marl [3] are discussed in detail in this paper. Diderot [21] is a parallel DSL for image analysis that demonstrates good performance compared to

hand-written C code using an optimized library. The Spiral system [22] progressively lowers linear transform algorithms through a series of different DSLs to apply optimizations on different levels [23]. Subsets of Spiral have also been implemented using Scala and LMS. Giarrusso et al. [24] investigate database-like query optimizations for collection classes and present SQuOpt, a query optimizer for a DSL that, like OptiCollections, mimics the Scala collections API and also uses techniques similar to LMS to obtain an IR for relevant program expressions. Our work aggregates many of the lessons and techniques from previous DSL efforts and makes them easier to apply to new domains, and to the problem of composing DSLs.

Recent work has begun to explore how to compose domain-specific languages and runtimes. Mélusine [25] uses formal modeling to define DSLs and their composition. Their approach attempts to reuse existing models and their mappings to implementations. Dinkelar et al. [26] present an architecture for composing purely embedded DSLs using aspect-oriented concepts; a meta-object is shared between all the eDSLs and implements composition semantics such as join points. MadLINQ [27] is an embedded matrix DSL that integrates with DryadLINQ [28], using LINQ as the common back-end. Compared to these previous approaches, our work is the first to demonstrate composition and co-optimization with high performance, statically compiled DSLs.

There has also been work on extensible compilation frameworks aimed towards making DSLs and high performance languages easier to build. Racket [29] is a dialect of Scheme designed to make constructing new programming languages easier. Spoofax [30] and JetBrains MPS [31] are language workbenches for defining new DSLs and can generate automatic IDE support from a DSL grammar. While these efforts also support DSL reuse and program transformation, they are generally more focused on expressive DSL front-ends, whereas Delite's emphasis is on high performance and heterogeneous compilation. Both areas are important to making DSL development easier and could be used together to complement each other. On the performance side, telescoping languages [32] 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 also optimize both the DSL and the program using it in a single step.

Outside the context of DSLs, there have been efforts to compile high-level general purpose languages to lower-level (usually device-specific) programming models. Mainland et al. [6] 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 is not an issue with our version of the Scala compiler. Copperhead [7] automatically generates and executes CUDA code on a GPU from a data-parallel subset of Python. Nystrom et al. [33] show a library-based approach to translating Scala programs to OpenCL code through Java bytecode translation. Since the starting point of these compilers is byte code or generic Python/Java statements, the opportunities for high-level optimizations are more limited relative to DSL code.

8 Conclusion

In this paper we showed that a common back-end can be used to compose high performance, compiled domain-specific languages. The common back-end also provides a means to achieve meaningful reuse in the compiler implementations when targeting heterogeneous devices. We demonstrated this principle by implementing four new diverse DSLs (OptiQL, OptiCollections, OptiGraph, and OptiMesh) in Delite, an extensible compilation framework for compiled embedded DSLs. The DSLs required only 9 parallel operators and 7 were reused in at least two DSLs. We showed that OptiQL and OptiCollections exceed the performance of optimized library implementations by up to 125x. OptiGraph and OptiMesh are both based on existing stand-alone DSLs (Green-Marl and Liszt respectively) but require less code to build and achieve no worse than 30% slow-down. In addition to each DSL providing high performance and targeting multicore and GPU architectures, applications composing multiple DSLs perform well and benefit from cross-DSL optimization. To the best of our knowledge, this work is the first to demonstrate high performance compiled DSL composability.

Acknowledgements. We are grateful to the anonymous reviewers for their detailed suggestions, to Nada Amin for her assistance with the Scala-Virtualized compiler, and to Peter Kessler and Zach DeVito for reviewing previous versions of this paper. This research was sponsored by DARPA Contract, Xgraphs; Language and Algorithms for Heterogeneous Graph Streams, FA8750-12-2-0335; Army contract AHPCRC W911NF-07-2-0027-1; NSF grant, BIGDATA: Mid-Scale: DA: Collaborative Research: Genomes Galore, IIS-1247701; NSF grant, SHF: Large: Domain Specific Language Infrastructure for Biological Simulation Software, CCF-1111943; Stanford PPL affiliates program, Pervasive Parallelism Lab: Oracle, AMD, Intel, and NVIDIA; and 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.

References

1. Chafi, H., Sujeeth, A.K., Brown, K.J., Lee, H., Atreya, A.R., Olukotun, K.: A domain-specific approach to heterogeneous parallelism. PPoPP (2011)
2. DeVito, Z., Joubert, N., Palacios, F., Oakley, S., Medina, M., Barrientos, M., Elsen, E., Ham, F., Aiken, A., Duraisamy, K., Darve, E., Alonso, J., Hanrahan, P.: Liszt: A domain specific language for building portable mesh-based PDE solvers. SC (2011)
3. Hong, S., Chafi, H., Sedlar, E., Olukotun, K.: Green-marl: A DSL for easy and efficient graph analysis. ASPLOS (2012)
4. Elliott, C., Finne, S., de Moor, O.: Compiling embedded languages. In Taha, W., ed.: Semantics, Applications, and Implementation of Program Generation. Volume 1924 of LNCS. Springer Berlin / Heidelberg (2000) 9–26
5. Leijen, D., Meijer, E.: Domain specific embedded compilers. DSL, New York, NY, USA, ACM (1999) 109–122
6. Mainland, G., Morrisett, G.: Nikola: embedding compiled GPU functions in haskell. Haskell '10, New York, NY, USA, ACM (2010) 67–78
7. Catanzaro, B., Garland, M., Keutzer, K.: Copperhead: compiling an embedded data parallel language. PPoPP, New York, NY, USA, ACM (2011) 47–56
8. Brown, K.J., Sujeeth, A.K., Lee, H., Rompf, T., Chafi, H., Odersky, M., Olukotun, K.: A heterogeneous parallel framework for domain-specific languages. PACT (2011)

9. Rompf, T., Odersky, M.: Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. GPCE, New York, NY, USA, ACM (2010) 127–136
10. Rompf, T., Sujeeth, A.K., Lee, H., Brown, K.J., Chafi, H., Odersky, M., Olukotun, K.: Building-blocks for performance oriented DSLs. DSL (2011)
11. Sujeeth, A.K., Lee, H., Brown, K.J., Rompf, T., Wu, M., Atreya, A.R., Odersky, M., Olukotun, K.: OptiML: an implicitly parallel domain-specific language for machine learning. ICML (2011)
12. Taha, W., Sheard, T.: MetaML and multi-stage programming with explicit annotations. Theor. Comput. Sci. **248**(1-2) (2000) 211–242
13. Moors, A., Rompf, T., Haller, P., Odersky, M.: Scala-virtualized. PEPM (2012)
14. Rompf, T., Sujeeth, A.K., Amin, N., Brown, K., Jovanovic, V., Lee, H., Jonnalagedda, M., Olukotun, K., Odersky, M.: Optimizing data structures in high-level programs. POPL (2013)
15. Meijer, E., Beckman, B., Bierman, G.: LINQ: Reconciling object, relations and XML in the .NET framework. SIGMOD, New York, NY, USA, ACM (2006) 706–706
16. Page, L., Brin, S., Motwani, R., Winograd, T.: The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab (November 1999) Previous number = SIDL-WP-1999-0120.
17. Prokopec, A., Bagwell, P., abd Martin Odersky, T.R.: A generic parallel collection framework. Euro-Par (2010)
18. Lea, D.: A java fork/join framework. JAVA '00, New York, NY, USA, ACM (2000) 36–43
19. Yang, J., Leskovec, J.: Patterns of temporal variation in online media. WSDM '11, New York, NY, USA, ACM (2011) 177–186
20. Axelsson, E., Claessen, K., Sheeran, M., Svenningsson, J., Engdal, D., Persson, A.: The design and implementation of feldspar an embedded language for digital signal processing. IFL'10, Berlin, Heidelberg, Springer-Verlag (2011) 121–136
21. Chiw, C., Kindlmann, G., Reppy, J., Samuels, L., Seltzer, N.: Diderot: a parallel DSL for image analysis and visualization. PLDI, New York, NY, USA, ACM (2012) 111–120
22. Püschel, M., Moura, J., Johnson, J., Padua, D., Veloso, M., Singer, B., Xiong, J., Franchetti, F., Gacic, A., Voronenko, Y., Chen, K., Johnson, R., Rizzolo, N.: Spiral: Code generation for DSP transforms. Proceedings of the IEEE **93**(2) (feb. 2005) 232 –275
23. Franchetti, F., Voronenko, Y., Püschel, M.: Formal loop merging for signal transforms. PLDI (2005) 315–326
24. Giarrusso, P.G., Ostermann, K., Eichberg, M., Mitschke, R., Rendel, T., Kästner, C.: Reify your collection queries for modularity and speed! AOSD (2013)
25. Estublier, J., Vega, G., Ionita, A.: Composing domain-specific languages for wide-scope software engineering applications. In Briand, L., Williams, C., eds.: Model Driven Engineering Languages and Systems. Volume 3713 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2005) 69–83
26. Dinkelaker, T., Eichberg, M., Mezini, M.: An architecture for composing embedded domain-specific languages. AOSD, ACM (2010) 49–60
27. Qian, Z., Chen, X., Kang, N., Chen, M., Yu, Y., Moscibroda, T., Zhang, Z.: Madlinq: large-scale distributed matrix computation for the cloud. EuroSys '12, New York, NY, USA, ACM (2012) 197–210
28. Isard, M., Yu, Y.: Distributed data-parallel computing using a high-level programming language. SIGMOD, New York, NY, USA, ACM (2009) 987–994
29. Flatt, M.: Creating languages in racket. Commun. ACM **55**(1) (January 2012) 48–56
30. Kats, L.C., Visser, E.: The spoofax language workbench: rules for declarative specification of languages and IDEs. OOPSLA '10, New York, NY, USA, ACM (2010) 444–463
31. JetBrains: Meta Programming System (2009)
32. Kennedy, K., Broom, B., Chauhan, A., Fowler, R., Garvin, J., Koelbel, C., McCosh, C., Mellor-Crummey, J.: Telescoping languages: A system for automatic generation of domain languages. Proceedings of the IEEE **93**(3) (2005) 387–408
33. Nystrom, N., White, D., Das, K.: Firepile: run-time compilation for GPUs in scala. GPCE, New York, NY, USA, ACM (2011) 107–116