Forge: Generating a High Performance DSL Implementation from a Declarative Specification

Arvind K. Sujeeth* Austin Gibbons* Kevin J. Brown* HyoukJoong Lee* Tiark Rompf ‡† Martin Odersky† Kunle Olukotun*

*Stanford University: {asujeeth, gibbons4, kjbrown, hyouklee, kunle}@stanford.edu
‡Oracle Labs: {first.last}@oracle.com
†EPFL: {first.last}@epfl.ch

Abstract

Domain-specific languages provide a promising path to automatically compile high-level code to parallel, heterogeneous, and distributed hardware. However, in practice high performance DSLs still require considerable software expertise to develop and force users into tool-chains that hinder prototyping and debugging. To address these problems, we present Forge, a new meta DSL for declaratively specifying high performance embedded DSLs. Forge provides DSL authors with high-level abstractions (e.g., data structures, parallel patterns, effects) for specifying their DSL in a way that permits high performance. From this high-level specification, Forge automatically generates both a naïve Scala library implementation of the DSL and a high performance version using the Delite DSL framework. Users of a Forge-generated DSL can prototype their application using the library version, and then switch to the Delite version to run on multicore CPUs, GPUs, and clusters without changing the application code. Forge-generated Delite DSLs perform within 2x of hand-optimized C++ and up to 40x better than Spark, an alternative high-level distributed programming environment. Compared to a manually implemented Delite DSL, Forge provides a factor of 3-6x reduction in lines of code and does not sacrifice any performance. Furthermore, Forge specifications can be generated from existing Scala libraries, are easy to maintain, shield DSL developers from changes in the Delite framework, and enable DSLs to be retargeted to other frameworks transparently.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Concurrent, distributed and parallel languages, Extensible languages; D.3.4 [*Programming Languages*]: Processors—Code generation, Optimization, Runtime environments

Keywords Code Generation; Multi-Stage Programming; Domain-Specific Languages; Parallel Programming

GPCE '13, October 27–28, 2013, Indianapolis, Indiana, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2373-4/13/10…$15.00.
http://dx.doi.org/10.1145/2517208.2517220

1. Introduction

In order to achieve high performance on modern hardware, software must be parallel, heterogeneous, and distributed [36]. Unfortunately, developing software that meets these goals is complex, and high performance applications are often cobbled together in a piecemeal fashion. For example, an expert application developer may rewrite a performance-critical portion of her application in CUDA [29] in order to leverage a GPU, and rewrite a different portion in Hadoop [1] to scale out to a large number of cores. Each of these tasks requires expertise both in the idiosyncracies of the programming model and in the characteristics of the underlying hardware. Worse, rewriting parts of the application for performance obscures the high-level algorithm and intent, making code harder to read and maintain and increasing the complexity of the overall system.

General-purpose languages have so far been unable to maintain a high level of abstraction and still provide efficient automatic compilation to parallel and heterogeneous hardware. State-of-the-art solutions like OpenCL [38] are relatively close in abstraction layer to C and still require careful programming to achieve good performance on a particular device. On the other hand, dynamic languages like Python and Ruby provide succint, high-level abstractions that lead to great productivity benefits, but compile (in a standard installation) only to sequential processors and are much slower than their low-level counterparts.

Domain-specific languages (DSLs) are a promising means of maintaining high-level abstractions while automatically compiling to parallel, heterogeneous, and distributed hardware [10]. The key advantage that DSLs have over general-purpose languages is the ability to reason about data structures and operations at the level of domain abstractions (e.g. `Matrix` or `Graph` vs. `Array`). By exploiting this high-level structure, it has been shown that DSL applications can be compiled from a single source to multicore CPUs, GPUs, and even clusters [5, 14, 21]. However, DSLs have two substantial drawbacks: they are usually more difficult to construct than a high-level library, and because they use specialized tool-chains, DSL applications are harder to prototype and debug.

In our previous work, we proposed the Delite framework [4] to mitigate these problems. Delite is a library for developing compiled, embedded DSLs inside the general-purpose programming language Scala. We showed that using a common host language and compilation framework enables reuse of the Scala tool-chain and important infrastructure like optimizations and code generators, thereby substantially reducing the effort to create a new DSL. Furthermore, we demonstrated that Delite DSLs can generate code for different devices and are competitive with or exceed the performance of alternate systems in different domains [35]. While we believe Delite to be the state-of-the-art in high performance embedded DSL development, there are still two main issues:

1. *DSL authors* must have considerable expertise with Scala and Delite in order to implement expressive, safe, and efficient DSLs. Delite is a highly flexible architecture for heterogeneous code generation, but this expressiveness adds boilerplate and complexity to the common case. As a result, developing compiled embedded DSLs with Delite, while easier than external DSLs, requires more programming language and software engineering expertise than the average domain expert has.
2. *DSL users* must have at least some knowledge of the Delite stack. Even though DSLs are embedded in Scala, only parts of the Scala tool-chain can be used when executing user programs: generating code at runtime that is executed on heterogeneous devices makes prototyping and debugging difficult. For example, it is no longer possible to step through the program or set a breakpoint in an interactive IDE debugger.

In this paper, we present Forge, a new meta DSL for high performance embedded DSL development that addresses these issues by capturing recurring patterns in high performance DSL development. Forge provides a high-level, declarative API for specifying DSL data structures and operations in a manner similar to an annotated high-level library. Unlike a high-level library, Forge builds an IR of the DSL specification itself, which enables it to generate different concrete implementations of the DSL. We generate both a high productivity implementation (a pure Scala library version), and a high performance implementation (a Delite version). In the future, if a different backend is desired, it is straightforward to generate a new implementation without modifying the DSL specification. From a DSL user's point of view, the Forge-generated library can be used very similarly to any other Scala library. DSL applications can be prototyped interactively in a REPL or developed in any IDE that supports Scala (e.g. Eclipse) using standard debugging techniques like breakpoints. When an application has been tested on a small dataset, the DSL user can then "flip the switch" and run the exact same source code in Delite on multicore CPUs, GPUs, or clusters. Therefore, by raising the level of abstraction and adding a level of code generation, Forge both simplifies the development of high performance embedded DSLs and makes them more accessible to end users.

Languages and frameworks for declaratively specifying DSLs are not new [16, 17, 24, 26]. Forge follows in the footsteps of these efforts by focusing on abstractions and code generation for high performance, heterogeneous computing. To our knowledge, Forge is also the first embedded meta DSL. It is implemented using staging, and therefore shares the same infrastructure as existing Delite DSLs. Staging also provides additional benefits: Forge specifications are Scala programs, and any Scala modularity feature (objects, classes, traits) can be used to compose specifications. We can also make use of staging to perform computation inside the specification itself. For example, the DSL specification can be parameterized over configuration flags, implemented as regular Scala values. This enables generating multiple variants of a DSL–essentially implementing a product line approach to DSL development. Furthermore, since Forge is staged, its IR can be constructed by invoking methods in the Forge API. This enables us to develop external parsers that call Forge methods to build the Forge IR. A key use case is to use reflection to parse existing Scala classes. Forge can then be used as an identity transformer, generating a Forge specification from the IR that can be further modified by a DSL developer. This scaffolding ability allows DSL developers to start with an existing library implementation and automatically generate a skeleton Forge specification as a starting point for a Forge DSL.

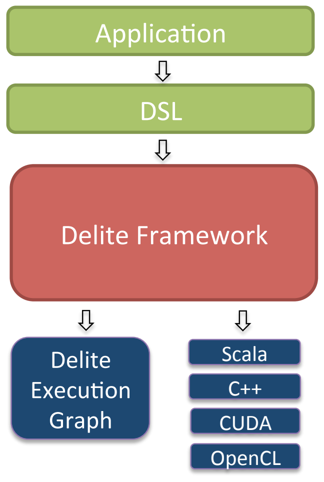

Figure 1. An overview of the Delite compilation pipeline. Applications are written in one or more DSLs. Each DSL can perform analyses and optimizations at the domain level. Delite then performs its own set of generic analyses and transformations before finally generating code for various low-level hardware programming models.

The rest of this paper is structured as follows. In Section 2, we provide essential background on Delite required to understand how the Forge-generated compiled version works. In Section 3 we provide an overview of the Forge language and show how DSLs are written in Forge. Section 4 describes how Forge is implemented internally and describes the artifacts that Forge generates in more detail. Section 5 presents case studies on three DSLs we have implemented in Forge (including two that were first implemented as stand-alone Delite DSLs and one that was implemented from scratch). Finally, Section 6 summarizes the related work and Section 7 concludes.

Forge is open-source. The source code, including examples presented in this paper, is available at:
`http://github.com/stanford-ppl/Forge/`

2. Background

Delite is a compilation framework for high performance DSLs embedded in Scala, built on top of Lightweight Modular Staging (LMS) [31]. Figure 1 depicts the Delite compilation pipeline.

Applications are written in a DSL embedded in Scala. For example, consider a toy application using a `SimpleVector` DSL:

```
trait MyApp extends SimpleVectorDSLInterface {
  def main() {
    val v: Rep[Vector[Int]] = Vector[Int](10)
    val y = v+2
    print(y(1))
  }
}
```

The DSL syntax (e.g. **val**,+) is legal Scala, although implicit conversions and curried functions can be used to emulate external DSL syntax [9]. However, unlike normal Scala library code, we use staged metaprogramming (*staging*) to defer key computations to a later time. The main idea behind LMS is that future-stage computation is encoded by wrapping types in an abstract type constructor `Rep[T]`. DSLs internally define `Rep[T]` to be `Exp[T]`, an expression in an intermediate representation (IR). The IR is built during Scala run-time (also known as *staging time*), by implementing methods on the abstract `Rep[T]` types to construct IR nodes rather than imme-

```
// abstract interface exposed to DSL users
trait Base {
  type Rep[+T]
  implicit def unit[T](x: T): Rep[T]
}

// using staging to build an IR
trait BaseExp extends Base {
  abstract class Exp[+T:Manifest]
  abstract class Def[+T]
  type Rep[T] = Exp[T]

  case class Const[+T:Manifest](x: T) extends Exp[T]
  case class Sym[+T:Manifest](id: Int) extends Exp[T]

  implicit def unit[T](x: T) = Const(x)
  implicit def toAtom(d: Def[T]): Exp[T] =
    findOrCreateDefinition(d) // elided

  // a Delite IR node representing a parallel operation
  abstract class DeliteOpMap[Col](in: Exp[Col]) extends Def[Col]
}

// defines how to generate target (low-level) Scala code
trait ScalaCodegen {
  val IR: BaseExp; import IR._

  // constructs a program schedule of the IR by traversing
  // dependencies backwards, and calls emitNode in order
  def emitBlock(b: Exp[T]): Unit

  // implements code generation for individual IR nodes
  def emitNode(sym: Sym[T], rhs: Def[T]): Unit = rhs match {
    case _: DeliteOpMap => // emit parallel code
  }
}
```

Listing 1. Simplified core of LMS/Delite

diately evaluate a result. Note that while LMS uses `Exp[T]` to represent staged computation, the representation of the abstract types is polymorphic and DSLs can in principle be implemented in multiple ways using different implementations of `Rep[T]` (a fact that Forge relies heavily on, as discussed later). This technique is known as polymorphic embedding [7, 20]. After the IR is constructed, Delite traverses it to perform optimizations and generate code. The user then invokes the Delite runtime to run the generated code. Compared to running a pure Scala library, the generated code is efficient (abstractions have been programmatically removed by staging), heterogeneous, and parallel, which can lead to orders of magnitude improvement in performance [35]. Key optimizations performed by LMS and Delite include code motion, fusion, struct unwrapping, and array of struct to struct of array (AoS to SoA) conversion. Unlike general-purpose compilers, these optimizations are performed at the granularity of domain operations and data structures.

In order to demonstrate what this looks like under the covers, we show a bare-bones core of LMS and Delite in Listing 1, and the `SimpleVector` DSL implemented with this core in Listing 2. The core LMS traits define IR building blocks, such as expressions, definitions, and particular types of leaf expressions (e.g. constants and symbols). Delite adds on top of these to provide specialized classes of IR nodes that represent parallel patterns (e.g. `DeliteOpMap`). Although not shown here, LMS and Delite also provide facilities to construct *structs*. Structs are simple aggregate data types consisting of high performance primitives (e.g. scalars, arrays, and other structs) that Delite automatically optimizes and generates code for.

```
trait SimpleVectorDSLInterface extends Base {
  trait Vector[T] // an abstract DSL data type
  object Vector {
    // SourceContext has debugging info (e.g. line # of call-site)
    def apply[T:Manifest](n: Rep[Int])(implicit ctx: SourceContext)
      = vector_new[T](n)
  }

  // indirection required for abstract static method
  def vector_new[T:Manifest](n: Rep[Int])
    (implicit ctx: SourceContext): Rep[Vector[T]]
  // sugar for infix operators in Scala-Virtualized
  def infix_+[T:Manifest:Numeric](x: Rep[Vector[T]],
                                  y: Rep[T]): Rep[Vector[T]]
  // an overloaded version. due to type erasure, we need to use
  // an implicit to statically disambiguate the method signature
  def infix_+[T:Manifest:Numeric](x: Rep[Vector[T]],
    y: Rep[Vector[T]])(implicit o: Overloaded1): Rep[Vector[T]]
  def print(x: Rep[Any]): Rep[Unit]
  // infix_apply (element access) elided for space
}

trait SimpleVectorDSLImpl extends SimpleVectorDSLInterface
  with BaseExp {
  // parallel domain-specific IR nodes
  case class VPlusS[T:Manifest:Numeric](x: Exp[Vector[T], y: Exp[T])
    extends DeliteOpMap[Vector[T]] { def func = a => a+y }
  // sequential domain-specific IR nodes
  case class Print(x: Exp[Any]) extends Def[Unit]

  // construct IR node when method is called
  // vector_new, overloaded infix_+ elided
  def infix_+[T:Manifest:Numeric](x: Exp[Vector[T]], y: Exp[T])
    = VPlusS(x,y)
  def print(x: Rep[Any]) = reflectEffect(Print(x))

  // constructs transformed IR nodes
  override def mirror[A:Manifest](e: Def[A], f: Transformer) =
    e match {
      case VPlusS(x,y) => infix_+(f(x),f(y))
      // ...
      case _ => super.mirror(e,f)
    }
}

trait SimpleVectorDSLCodegen extends ScalaCodegen {
  val IR: SimpleVectorDSLImpl; import IR._

  override def emitNode(s: Sym[Any], r: Def[Any]) = r match {
    case Print(x) => emitValDef(s, "println("+quote(x)+")")
    case _ => super.emitNode(s,r)
  }
}
```

Listing 2. SimpleVector DSL implementation using core

The DSL implementation traits extend the core traits to actually construct IR nodes when a method is called, to define helper methods required by LMS and Delite (such as `mirror`, which defines how to construct a transformed node), and to provide code generators for any node that is not a predefined pattern. A real implementation would use much more sophisticated versions of these IR building blocks provided by LMS and Delite, but the basic principle is the same.

Listing 2, although simplified, still demonstrates key productivity pitfalls. First, there is a significant amount of boilerplate (e.g. `mirror`), which makes the DSL implementation verbose and hinders readability. The boilerplate is required to perform functions that Delite cannot easily infer; for example, in the `mirror` case, Delite does not know the name or arguments of the smart constructor that it needs to invoke to clone a node while still triggering any domain-specific rewrites that may be defined (which can be defined as overrides of the smart constructor). Second, DSL authors must be experts in Scala library development. They are exposed to implicit conversions, case classes, `Manifest` and `SourceContext` (which are Scala compiler-constructed types that carry around metadata), and even must know how to handle overloaded static method resolution in the presence of type erasure.

These issues arise because of the details of implementing a compiled embedded DSL in Scala. In other languages, there are different issues; for example, developing an external DSL requires DSL authors to actually deal with the entire process of lexing, parsing and type-checking. The key observation that we exploit with Forge is that by raising the level of abstraction, we can shield DSL developers from these implementation details.

3. Language Specification

Forge, as a meta DSL, provides methods to directly declare DSL constructs like types and operations that we saw in the previous section. Forge abstracts over the key concerns of high performance DSL development: front-end syntax, data structures, operation semantics, and parallel implementation. Forge aims to simplify DSL development by narrowing the gap between a DSL specification, which may be written as a text document, and the DSL implementation, which depends on the language and frameworks that the DSL compiler is implemented in. In the next section, we describe the implementation internals of Forge. In this section, we introduce its surface syntax and key abstractions.

To introduce Forge, we show how we can write the `SimpleVector` DSL example from Section 2 (including the struct definitions that were previously elided):

```
trait SimpleVectorDSL extends ForgeApplication {
  def dslName = "SimpleVector"

  def specification() {
    val T = tpePar("T")
    val Vector = tpe("Vector", T)
    data(Vector, ("_length", MInt), ("_data", MArray(T)))
    static (Vector) ("apply", T, MInt :: Vector(T), effect = mutable)
      implements allocates(Vector, ${$0}, ${ Array[T]($0) })
    direct (Vector) ("print", Nil, MString :: MUnit, effect = simple)
      implements codegen(scala, ${ println($0) })

    withTpe(Vector) {
      compiler ("vector_raw_data") (Nil :: MArray(T))
        implements getter(0, "_data")

      infix ("apply") (("n",MInt) :: T)
        implements composite ${ vector_raw_data($self)($n) }

      infix ("+") (("y",T) :: Vector(T), TNumeric(T))
        implements map((T,T), 0, ${ a => a+$y })

      infix ("+") (Vector(T) :: Vector(T), TNumeric(T))
        implements zip((T,T,T), (0,1), ${ (a,b) => a+b })
    }
  }
}
```

Every Forge specification must implement two methods: `dslName` is simply the DSL name, and `specification` is a method that contains all the DSL declarations (which could be spread across multiple files and dynamically invoked). In this example, we first declare a named type parameter, `T`, and a generic type `Vector[T]`. The **`data`** statement says that `Vector` is a struct containing two fields, `_length` and `_data`. The static method `apply` constructs a new `Vector`; **`static`** specifies that the user syntax for this method will be of the form `Vector(args)`, where `args` are the arguments to the op that are specified by the signature `MInt :: Vector(T)`. This signature says that the op takes a single argument of type `Int` (the length of the `Vector`), and returns a value of type `Vector[T]`. The **`effect`** `= mutable` annotation specifies that this op has the semantics of allocating a mutable data structure. Finally, **`allocates`** is an implementation pattern that constructs a new instance of `Vector` by initializing each field in the struct to an appropriate value. The `${..}` syntax is a marker for the Forge preprocessor, which quotes the argument as a formatted string, replacing argument names (specified with a preceding "$") with their synthetic names. Although these formatted strings are not type-checked when compiling or running Forge, the code in them *is* type-checked when compiling the generated DSL. The preprocessor handles tricky string escape issues while enabling users to benefit from syntax highlighting in an IDE. The last interesting construct in this example is **`withTpe`**, which introduces a Forge *syntactic scope*. Inside this scope, shorter versions of the op declaration methods are injected into the current lexical environment. The shorter versions implicitly take the enclosing **`tpe`** as the first argument (as well as its corresponding `tpePars`), which mimics the declaration style of instance methods in OO programming languages.

The other ops in the `SimpleVector` specification are defined in a similar fashion to `apply`, using other Forge method styles and implementation patterns. Listing 3 presents an overview of the Forge language constructs. The two most important groups abstract over computation and data structures, respectively, and correspond to concise versions of the Delite abstractions of Delite ops and Delite structs. Note that `implements` is an infix method that simply invokes the Forge construct **`impl`** on the result of the op invocation. In this way, Forge separates DSL interface from DSL implementation. Implementations may be defined in a completely different file, and DSL specifications can extend other DSL specifications and override behavior by defining new implementations.

DSL metaprogramming at this level also enables opportunities for programmatically-controlled reuse. Delite DSLs are typically statically dispatched, as this is most efficient and not all target platforms support dynamic dispatch (although tagged unions can be used as an alternative). Code generation provides an alternate mechanism for code reuse in this context. We can define a common `Vector` interface as follows:

```
def addVectorCommonOps(v: Rep[DSLType], T: Rep[DSLType]) {
  val VectorCommonOps = withTpe(v)
  VectorCommonOps {
    infix ("first") (Nil :: T) implements single ${ $self(0) }
    for (rhs <- List(DenseVector(T),DenseVectorView(T))) {
      infix ("+") (rhs :: DenseVector(T))
        implements zip((T,T,T), (0,1), ${ (a,b) => a+b })
      // ..
    }
    // other common ops
  }
}
```

The **`for`** statement in this example is statically evaluated during staging, so we can call `addVectorCommonOps` for different types of `Vectors` and replicate the common interface on each type. This enables each method to be invoked efficiently by end users on different types of `Vectors` without requiring implicit conversions, dynamic dispatch, or type classes. The trade-off, of course, is the potential for code explosion, which can negatively impact DSL compile time. This method is also insufficient for DSL users to write generic methods over `Vectors`. However, in DSLs with limited class hierarchies, this can be a sufficient, and concise, replacement for full-blown polymorphism.

```
Types:
ftpe(args, ret, freq)
  define a new function type (args) => ret
tpePar(name)
  define new type parameter
tpe(name, tpePars, stage)
  define new type
tpeClass(name, tpePars)
  define new type class
tpeClassInst(name, tpePars, tpeClass)
  define new type class instance

Data structures:
data(tpe, (fieldName,fieldTpe)*)
  associate tpe with the given struct
impl(op, allocates(tpe))
  implementation pattern to allocate a struct
impl(op, getter(tpe,field))
  implementation pattern to read a field
impl(op, setter(tpe,field))
  implementation pattern to write a field

Annotations:
effect ::= simple | mutable | write | error
  declare an op has the corresponding effect semantic
freq ::= hot | cold | normal
  code motion hints
stage ::= now | future
  declare whether a generated type should be T or Rep[T]
aliasHint ::= nohint | contains(arg) | copies(arg)
  declares relationships between operations and inputs

Methods:
arg(name,tpe,default)
  define a new op argument
static | infix | direct | compiler | fimplicit
 (grp, name, tpePars, signature, effect, aliasHint)
  defines a new op with the specified syntax style and parameters
(args :: retTpe)
  defines a new method signature
impl(op, codegen | single | composite | map | filter |
        groupby | reduce | zip | foreach)
  defines the implementation for an op based on a predefined pattern

Miscellaneous:
grp(name)
  declares a group of ops that do not belong to a type
extern(name)
  declares an op group implemented in external code
lift(grp)(tpe)
  declares a conversion from the given tpe to a Rep[.]
lookupOp | lookupGrp | lookupTpe (name)
  returns a previously declared DSL construct
withTpe(tpe)
  construct a new syntactic scope

Parallel Collections:
parallelize(tpe) as ParallelCollection | ParallelCollectionBuffer (ops)
  declares that the provided type implements a ParallelCollection
  interface with the given ops
```

Listing 3. Forge Language Overview

While Forge tries to make declaring DSL semantics simple and concise, it is important to remember that Forge is meant for high performance embedded DSL *compilers*. It is not a goal of ours to reproduce exactly a sequential library interface. Instead, the Forge abstractions are intended to capture the critical semantics required to implement parallel DSLs on multiple hardware devices. Unsurprisingly, since Forge is based on our experience with developing DSLs in Delite, the Forge abstractions are a high-level version of concepts in Delite (like parallel patterns, effects, and alias / code motion hints). By designing Forge as a new language, we gain the flexibility to easily add and refine these abstractions over time.

4. Compilation Pipeline

Forge is implemented as an embedded LMS DSL. Implementing it as an external DSL (or within an alternative DSL definition environment like Spoofax [26]) could make its syntax cleaner (e.g., we would not need to wrap names in strings), but would require more development effort compared to simply reusing the existing LMS infrastructure. One interesting aspect of Forge's implementation that mitigates the need for an external grammar is its unique use of Scala-Virtualized *scopes*. Scala-Virtualized is an experimental branch of the Scala compiler with additional support for DSL embedding [32]. A scope is a Scala-Virtualized construct that desugars a lexical block of code to an instantiation of a pre-defined Scala class, wrapping the block's contents inside a method of the class and then invoking the class constructor. We previously used this technique to implement coarse-grained DSL interoperability; the scope isolated the DSL interface inside the block, allowing DSL code to be invoked from within ordinary Scala programs [35]. In Forge, we use scopes to implement the **withTpe** construct from the previous section, injecting new method signatures into a lexical scope while maintaining type safety. This is an example of using Scala-Virtualized to enable expressive embedded syntax in a way that would not normally be possible in a statically typed language. As described in Section 1, using staging also provides Forge with other benefits (such as the use of Scala composition, uniformity with the generated DSLs, and the ability to use staging time computations to statically manipulate DSL fragments). Like other LMS DSLs, Forge specifications are legal Scala, except for the blocks denoted with `${..}` which are preprocessed before compilation as described in Section 3. The preprocessor is implemented as a precompile hook in SBT (Simple Build Tool) [40], the predominant build tool for Scala. It is small (around 300 lines of code) and performs a simple forward pass to quote next-stage code in DSL methods using Scala string interpolation.

Figure 2 illustrates the Forge compilation pipeline. When run, Forge constructs an IR of the DSL consisting of types, operations, data structures, etc. Whereas Delite DSLs traverse the IR and use different code generators to generate code for different platforms, Forge traverses the IR and uses different code generators to generate different DSL implementations. Since the information required to generate a sequential Scala library is a subset of that needed to generate the Delite DSL, it is simple to generate the library version. It is also relatively straightforward to add a new code generator to Forge to retarget a DSL to a new back-end (for example, an alternative runtime) without changing the DSL specification or Forge internals.

Along with an input Forge specification, Forge also allows external Scala code that should be added to the DSL implementations. This code is placed in a configurable directory, and copied by Forge automatically to the generated directory. External code provides an escape hatch for any situation that Forge does not support. Domain-specific optimizations, such as pattern rewrites and transformations, are defined as external code using Delite APIs directly, and made visible to the generated DSL using Forge's **extern**

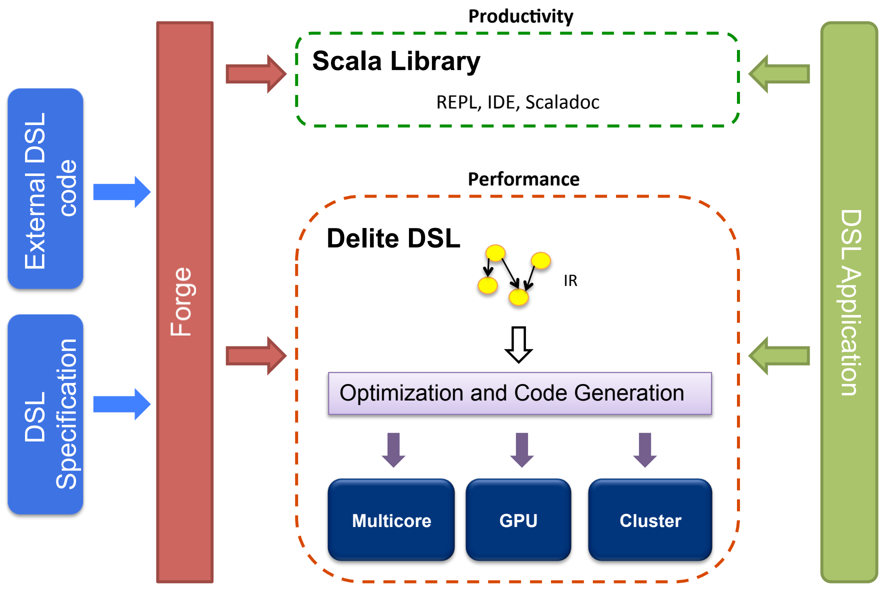

Figure 2. An overview of the Forge compilation pipeline. Forge takes as an input a DSL specification, as described in Section 3, and optional external DSL code. Forge generates two DSL implementations from these components: a high-productivity pure Scala version and a high-performance Delite version.

command. We chose this method because Delite's APIs for pattern rewriting and transformers [33] are already high-level and declarative; it is not obvious how we could abstract over these APIs in a useful way. Furthermore, these sorts of optimizations are impossible to implement in the Scala library version, so are only relevant to the Delite version.

Once the DSL implementations are generated, there is no further dependency on Forge, and DSL users can use either one. An important aspect of Forge's code generation is that it leverages polymorphic embedding not only for the Delite implementation, but also for the library implementation. In the Delite version, `Rep[T]` = `Exp[T]` as in normal Delite DSLs, but in the library version, `Rep[T]` = `T`, and Forge generates concrete classes and methods on those classes for DSL types. The key point is that DSL users now use exactly the same interface when writing their application, and need only run a different Scala object to switch between versions:

```
object MyAppInterpreter extends SimpleVectorApplicationInterpreter
  with MyApp
object MyAppCompiler extends SimpleVectorApplicationCompiler
  with MyApp

trait MyApp extends SimpleVectorApplication {
  def main() = {
    val v = Vector.rand(10)
    println("v.sum: " + sum(v))
  }
}
```

Running scala on `MyAppInterpreter` after compiling will run the Scala application, while running `MyAppCompiler` will run the Delite version to stage it and generate code for different devices. When Forge generates a DSL, it also generates the SBT project file for the DSL. A DSL user simply has to run `sbt; console`, and they will be dropped into a Scala REPL with all interpreter DSL dependencies pre-loaded. This provides DSL users a way to prototype their application in the Scala REPL (they can even copy and paste app code as normal) and debug their applications inside Scala IDEs in the ordinary way. In the development cycle, this also means that DSL users can also avoid expensive compilation and staging time until they actually require high performance. When a user has finished debugging, he can switch to a larger dataset and invoke the Delite DSL. Therefore, although we have added an additional compilation step in the multi-stage compilation pipeline for DSL authors, the development cycle for DSL users can be considerably shortened. Since there are far more users than authors, this is normally a good trade-off. In order to maintain incremental compilation across the multiple stages while developing the DSL, Forge uses `rsync` to copy files to the generated directory.

In addition to supporting DSL construction from scratch, Forge includes a *scaffolding* code generator that allows DSL developers to bootstrap off of regular Scala classes. This generator serializes the Forge IR to re-emit a Forge specification (i.e., from an existing Forge specification, it acts like an identity generator). In order to generate a skeleton Forge specification using reflection, we use staging to build the Forge IR by simply calling the appropriate Forge methods while traversing the class. After the specification is generated, a DSL author can then fill in the gaps by adding semantic annotations (e.g. effects) and alternate code generators. As an example, if we reflect the standard `String` class using:

```
importAuto[java.lang.String]
```

Forge will generate a skeleton specification like the following:

```
val String = tpe ("java.lang.String")
val StringOps = grp ("String")

infix (StringOps)("trim", Nil,
  ((String) :: String), effect = simple) implements
  (codegen(scala, ${ $0.trim }))

infix (StringOps)("toLowerCase", Nil,
  ((String) :: String), effect = simple) implements
  (codegen(scala, ${ $0.toLowerCase }))

infix (StringOps)("replaceAll", Nil,
  ((String,String) :: String), effect = simple) implements
  (codegen(scala, ${ $0.replaceAll($1) }))

...
```

The implementation of importAuto is straightforward. It uses Scala reflection to traverse the methods of the given class and stages the corresponding Forge commands on the fly:

```
def importAuto[T:TypeTag] = {
  val scalaType = typeTag[T].tpe
  val forgeType = toForgeType(scalaType)
  val forgeClass = grp(scalaType.toString)
  lift (forgeClass) (forgeType)

  for (m <- scalaType.members if m.isMethod) {
    val methType = method.asTerm.typeSignature
    val args = toForgeArgs(methType.params)
    val ret = toForgeType(methType.resultType)

    infix (forgeClass) (m.name.toString, Nil,
      ((forgeType :: args)) :: ret, Nil, simple) implements
        (codegen(scala, quotedArg(0) + "." + m.name + argList(args)))
  }
}
```

This approach could easily be extended to read other Forge constructs from Scala or Java method annotations. With a sizable coverage of the Forge language, such annotations could provide a lightweight alternative front-end, at least for classes for which the developer is in control of the source code.

In the future, we plan to add additional code generators to Forge to generate additional artifacts. For example, if we allow users to specify Scaladoc annotations in the spec, we can generate the Scaladoc annotations in the Scala library implementation and leverage Scaladoc to generate the HTML API docs. In contrast, with ordinary Delite, there are no concrete classes in the DSL implementation and no place to put the Scaladoc annotations. We also plan to explore generating versions of the DSL that are less human readable, but are faster to compile (for example, by passing all implicits and specifying all types explicitly).

5. Evaluation

We have implemented three DSLs with Forge: OptiML (machine learning) [34], OptiQL (data querying) [35], and OptiWrangler (data transformation). Two of these (OptiML and OptiQL) were based on existing Delite DSLs, and we show that Forge significantly simplified their implementation without sacrificing performance. Furthermore, the new Forge implementations automatically produce library versions as we have discussed, so for the first time, OptiML and OptiQL can be used in a lightweight interactive way. OptiWrangler is a new implementation of Wrangler [25], and was implemented directly in Forge instead of ported from an existing Delite DSL.

For each DSL, we show that Forge lives up to the promise of DSL authors being able to write their DSL once, DSL users being able to write their application once, and being able to run efficiently on heterogeneous, parallel devices. We compare the performance of the Forge-generated DSL implementations to hand-optimized C++, to Spark [41], a Scala library for multicore and cluster computing, and to the previous Delite implementation (when available). In general, the hand-optimized C++ code is low-level and complex while the Spark version is high-level Scala that is much easier to read and to use. These two data points provide a strong measure of where the embedded DSL implementations stand in terms of productivity and performance for end users. The DSL application code is single-source and high-level, but Delite uses staging and also performs key optimizations like fusion and struct unwrapping to generate kernels that are low-level and first-order (either Scala or CUDA). In most cases, the generated Delite code closely resembles the hand-optimized C++.

DSL	Forge specification	Delite (manual)	Forge generated
OptiML	1322	7416	11743
OptiQL	301	862	1287
OptiWrangler	343	n/a	1814

Table 1. LOC for Forge implementations of each DSL vs. existing Delite implementations.

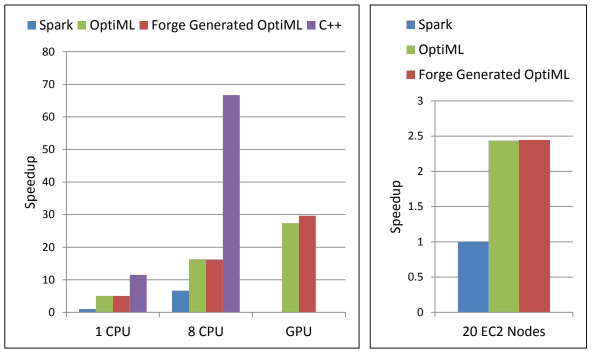

Figure 3. Speedup of Delite versions and manually-written C++ over Spark with LR on a 500k x 100 element dataset (multicore) and 10M x 100 (cluster).

Experimental Methodology We ran Forge on each DSL to generate a Scala library version and the Delite version. For each DSL application, Delite generated parallel Scala code for the CPU and CUDA code for the GPU (when possible). We ran the generated Scala code over a cluster using the Delite runtime's support for distributed computing [5], which uses Apache Mesos [19] and Google Protocol Buffers [18] as the underlying communication layer. The Spark experiments were run with Spark 0.7.0.

Multicore CPU and GPU 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 Scala code was executed on Oracle's Java SE Runtime Environment 1.7.0 and the HotSpot 64-bit server VM with a maximum heap size of 40GB. The generated CUDA code was executed with CUDA v3.2. The C++ implementations were compiled using g++ 4.4.7 with -O3. The cluster CPU experiments were performed on Amazon EC2 using 20 m1.xlarge instances. Each machine contained 4 virtual cores, 15GB of memory, and 1Gb Ethernet connections between the 20 machines. We used the default JVM available on Amazon Linux, Java 1.6.0b24 with default options, for all three systems. We ran each application ten times (to warm up the JIT) and report the average of the last 5 runs.

When counting lines of code (LOC) to compare Forge specifications of OptiML and OptiQL to the previous Delite versions, we used CLOC [13], and counted only the subset of features of the original DSLs that we re-implemented in the Forge spec.

5.1 OptiML

OptiML is a DSL for machine learning built around a linear algebra core, which we call OptiLA. OptiML provides implicitly parallel vectors, matrices, and graphs that support bulk collection operators (e.g. **map**, **filter**) as well as standard math operators when used with numeric types. OptiML also supports control structures for iterative algorithms (such as untilconverged and gradient descent) that are common in machine learning. The Forge implementation of OptiML includes the dense data types (DenseVector, DenseMatrix), most OptiML mathematical functions, and the control structures. Row 1 of Table 1 shows the LOC count of the OptiML specification vs. the corresponding subset of the original implementation, as well as the LOC generated by Forge. Forge provides nearly a 6x reduction in LOC over the original implementation, while generating roughly 50% more LOC because it also generates the library version, which was not a part of the original OptiML. The savings comes mainly from reducing boilerplate and automatically generating the embedded DSL structure and appropriate calls into the Delite API (as discussed in Section 2). We also gain some savings by using staging and code generation to implement common `Vector` operations on different kinds of `Vectors`, whereas the previous implementation used a verbose packaging of type classes to achieve (almost) the same functionality. Qualitatively, the OptiML specification is also simpler to read and modify than the previous version, primarily because there is much less clutter than in the original embedding implementation.

To evaluate performance, we ran Logistic Regression (LR), a simple classification algorithm for predicting the discrete value of a data sample (for example, whether a particular email is spam or not). Figure 3 shows the results running on multicore CPUs, a GPU, and across the 20 node EC2 cluster. The Delite version of Forge-generated OptiML achieves the same performance as the original OptiML implementation because we are able to generate the same code from the Forge specification. The Delite versions are about 2x slower than the low-level C++, but 2.5x faster than the high-level Spark. This C++ implementation is optimized to manually fuse all loops; it allocates an output buffer per thread in order to parallelize, but otherwise contains no intermediate allocations. Furthermore, the C++ code is byte-padded to prevent false sharing, which initially caused a 3x slow-down when naïvely parallelized using OpenMP. The Delite versions fail to reach this level of performance because the fusion algorithm misses one opportunity between different parts of a reduction kernel; we believe in the future we can extend the algorithm to cover this case, which will result in the generated Delite CUDA code outperforming the manual C++ even when starting from the high-level DSL code.

In the distirbuted setting, Delite is also able to run more efficiently across nodes than Spark for the same reasons it performs better in the multicore case: fusion eliminates intermediate allocations and staging generates more efficient code than Spark, which uses high-level Scala abstractions. It is important to note that Spark, in general, is extremely efficient; it has been shown to achieve order of magnitude speedups over equivalent Hadoop implementations by intelligently keeping data in memory across multiple iterations. Delite also keeps data in memory, but at this time does not provide the same fault-tolerance guarantees as Spark.

Finally, note that the Forge-generated library version did not finish on either dataset. This version uses identical code to the pre-staged Delite version (by construction), but since it is not staged or optimized, it suffers heavily from boxing and uses far more memory. In particular, the use of polymorphic embedding in its interface imposes more dispatch and boxing overhead than the Spark version, and since it is sequential, it cannot run on multiple processors. This demonstrates that the library version is suited for interactive prototyping with small datasets, but not for high performance or large-scale execution.

5.2 OptiQL

OptiQL is a LINQ-like [27] DSL for data querying. Its primary data type is a `Table`, and it provides query operators (e.g. `Select`, `Where`, `Sum`) over them. Like OptiML, we started from an existing Delite implementation of OptiQL, and ported it to Forge. The Forge version contains all of the supported features of OptiQL, but rewrite optimizations were implemented as external DSL code directly using Delite. The rewrites performed by OptiQL perform additional fusion of query operators that go beyond the generic fusion provided by Delite.

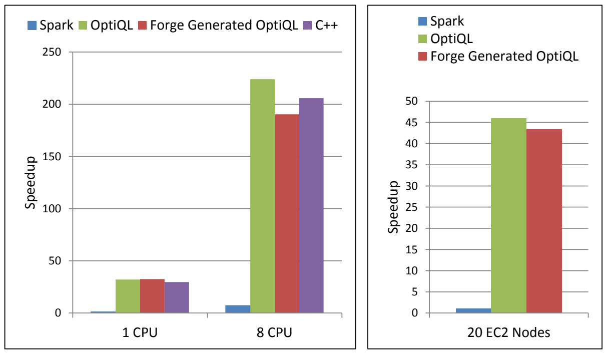

Figure 4. Speedup of Delite versions and manually-written C++ over Spark on TPC-H Q1 on a 1 GB table (multicore) and 5 GB (cluster).

Row 2 of Table 1 shows that the Forge OptiQL specification is about 3x shorter than the original Delite implementation. The difference is less dramatic than OptiML because OptiQL is a smaller DSL and a significant portion of its code is for the rewriting optimizations. Figure 4 shows the results of running the TPC-H benchmark suite query 1 (Q1) on multicore CPUs and on the cluster. TPC-H is a well-known database benchmark suite and Q1 consists of a single query containing `Select`, `Where`, `GroupBy`, and aggregate (e.g. `Sum`) statements. In the Delite version, the programmer-friendly array-of-struct representation is automatically converted to a more efficient struct-of-array representation, and then all of the operators are fused into a single, compact loop. Furthermore, fields that are part of the input dataset that are not used in the query are automatically eliminated from the generated code via dead field elimination. As a result, the Delite versions perform roughly the same as hand-optimized C++ and outperform Spark by 30x. In this case the speedup is magnified because the Delite optimizations have multiplicative effects; AoS to SoA enables fusion and struct unwrapping, which in turn enable dead-field elimination and the stack allocation of primitives. The same optimizations help Delite scale efficiently on the cluster, achieving 43x speedup over Spark. The increase is due to the fact that although the overall dataset is larger, the data per node is less in this experiment, and efficiency matters more. Similarly to OptiML, the Forge-generated Delite OptiQL version performs as well as the manual Delite version, while the Forge-generated library version is unable to finish for the same reasons as before (the lack of optimizations combined with polymorphic embedding and boxing overheads).

5.3 OptiWrangler

OptiWrangler is a DSL for structured string transformations and cleansing operations based on the interactive Wrangler system [25]. OptiWrangler exposes a single data element, `Wrangler`, and offers high level primitives (e.g. `cut`, `split`) that act over rows and columns of tabular data. In addition to managing a single table, OptiWrangler abstracts gracefully over multiple tables, allowing users to `partition` and `merge` tables without restructuring an application designed for a single table. We implemented OptiWrangler as both a Forge DSL and as a library with Spark. Unlike the other DSLs, OptiWrangler has not been previously implemented in Delite, so we use Spark as a comparison for productivity (note that in Subsections 5.1 and 5.2 the Spark implementations were for the application only, while for OptiWrangler, we implemented the entire DSL in Spark). Row 3 of Table 1 shows the code generated expansion of

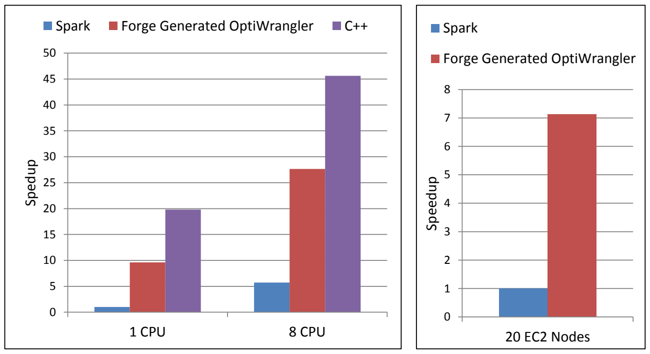

Figure 5. Speedup of Delite versions and manually-written C++ over Spark on the gene processing application with 3M sequences (multicore) and 25M (cluster).

the Forge implementation into a Delite DSL. OptiWrangler falls in between OptiML and OptiQL in terms of generated code size with a roughly 5x expansion. The Spark implementation of OptiWrangler (not shown in the table) is 253 LOC. Even though Spark is a pure Scala library, the Forge implementation is only 1.36x larger, and the difference is magnified because Forge specifications have a slightly higher fixed cost in LOC than Spark. For larger DSLs, this initial overhead would be amortized.

We evaluate performance on an application provided by a geneticist common in his workflow. Geneticists perform many types of structured-to-structured string operations, such as removing "barcodes" from gene sequences, separating genes by type, and extracting interesting subsequences. This application tests one kind of these structured transformations. Figure 5 shows the application performance with Forge-generated Delite vs. Spark and C++ (no manual Delite implementation exists). The generated Delite version unwraps all structs, leaving only a tight loop over arrays of strings. It performs approximately 2x slower than the C++ version, which heavily uses mutation to avoid allocating, for example, even new strings when constructing substrings. This requires using only a careful subset of C++ stdlib functions and makes the program harder to reason about. Like the C++ implementation of LR, this version also required byte padding to scale. On the other hand, Delite outperforms the Spark implementation by nearly an order of magnitude, which in this application is driven mostly by expensive object allocations. These benefits again translate to the cluster, resulting in a 7x speedup. This application falls somewhere in between the OptiML and OptiQL examples, since it is relatively simple (so there are not many optimizations to apply), but also not computationally heavy (so efficiency is important).

6. Related Work

The need for infrastructure to define DSLs has long been recognized and several languages and frameworks for declaratively specifying DSLs exist. The Kermeta workbench [28] is a metaprogramming environment based on metamodel engineering, which applies meta-languages to the problem of model transformations. Kermeta leverages DSLs for transforming models and the authors present common language constructs for model manipulation. The Eclipse Modeling Frameworks (EMF) [17] provide tools to generate code from a structured data model, specified in various languages (e.g. Java, XML). JetBrains MPS [24] and Spoofax [26] are language workbenches that enable developers to specify DSL grammars, static analyses and transformations (via rewrite rules, e.g. with Stratego [3]), and code generators. XText [16], JetBrains MPS and Spoofax all support automatically generating sophisticated tool-chain support for custom languages, such as IDE plugins, without relying on a host language. SugarJ [15], on the other hand, does utilize a host language by enabling language developers to translate grammar extensions to the host grammar (Java), as well as apply rewrite rules and transformations. Forge follows the spirit of these efforts but focuses on code generation to make high performance, heterogeneous computing more accessible, based on our experiences with the Delite framework. Since our effort has been invested on the back-ends of optimizing DSL compilers, there is significant potential in combining Forge with a declarative framework for front-end grammars and compile-time (as opposed to staging-time) static analyses. This would enable DSL developers to have even more flexibility to define both highly expressive and high performance DSLs.

Forge is unique as a meta DSL in that it is a staged, embedded DSL, that generates other staged, embedded DSLs with specialized implementations that share a common interface using polymorphic embedding. The embedding of DSLs in a host language was first described by Hudak [22]. Tobin-Hochstadt et. al. extend Racket, a Scheme dialect, with constructs to enhance the embeddability of other languages [39]. We also use an enhanced version of the Scala compiler (Scala-Virtualized [32]) to allow for a deeper embedding of DSLs. Feldspar [2] is an instance of an embedded DSL that combines shallow and deep embedding of domain operations to generate high performance code. Taha et. al. pioneered the field of multi-stage programming with MetaML [37] and MetaOCaml [6], extensions of ML and OCaml with staging annotations to demarcate future stage code. Forge, through LMS, uses type-directed staging instead of explicit annotations.

Forge was designed to target the Delite framework as a high performance back-end. In the field of parallel and heterogeneous computing, there have been numerous programming language and compilation approaches that have explored similar issues as, and contributed ideas to, Delite. Many of these systems are also good candidates for Forge-generated implementations, with different usability and performance trade-offs for end users of the DSL. For example, it is straightforward to generate an OpenMP [12] parallel C++ library implementation from Forge, which may integrate better with existing code bases, but would also miss out on the substantial optimizations implemented in the Delite framework (which can also generate C++ code, but at the application kernel, rather than DSL, level). OpenCL [38] is an industry-led standard for a relatively low-level, but uniform, programming model for heterogeneous devices. Forge DSLs (via Delite) can generate OpenCL from applications, allowing them to leverage the OpenCL compiler to target supported devices instead of performing low-level code generation manually. FlumeJava [11] and Dryad [23] have similar, data-flow based execution models as Delite, and also perform sophisticated optimizations on data parallel pipelines. Copperhead [8] and FirePile [30] perform run-time compilation for GPUs from Python and Scala, respectively. Forge DSLs share some of the embedding aspects of these approaches, but use staging for static whole-program analyses and generation rather than dynamic compilation, and can run in distributed environments as well as on multicore CPUs and GPUs.

7. Conclusion

We presented Forge, a meta DSL for high performance embedded DSL development. Forge is unlike previous language construction approaches in that it is embedded, staged, and generates multiple DSL implementations oriented towards simplifying heterogeneous parallel processing. Forge improves on the previous state of the art of compiled embedded DSLs by generating two versions of a DSL, a Scala library implementation that can be used for prototyping and debugging, and a Delite framework implementation that outperforms alternative systems and can run on multicore CPUs, GPUs, and clusters from a single application. Forge can also generate itself, so it can be used with an external parser, or reflection, to generate a skeleton specification from existing libraries. We have demonstrated that Forge simplifies the process of developing and using Delite DSLs, achieving up to 6x reduction in LOC compared to existing Delite DSLs while still providing order of magnitude speedups compared to library-based approaches. We believe that a declarative specification enables new opportunities for compiled embedded DSLs, such as the ability to transparently modify the underlying high performance framework or retarget the DSLs to alternate backends.

Acknowledgments

We are grateful to the anonymous reviewers for their suggestions which improved this paper. This research was sponsored by DARPA Contract, SEEC: Specialized Extremely Efficient Computing, Contract # 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.

References

[1] Apache. Hadoop. `http://hadoop.apache.org/`.

[2] E. Axelsson, K. Claessen, M. Sheeran, J. Svenningsson, D. Engdal, and A. Persson. The Design and Implementation of Feldspar: An Embedded Language for Digital Signal Processing. IFL'10, 2011.

[3] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Sci. Comput. Program.*, 72(1-2):52–70, June 2008.

[4] K. J. Brown, A. K. Sujeeth, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. A Heterogeneous Parallel Framework for Domain-Specific Languages. PACT, 2011.

[5] K. J. Brown, A. K. Sujeeth, H. Lee, T. Rompf, C. D. Sa, M. Odersky, and K. Olukotun. Big Data Analytics with Delite. `http://ppl.stanford.edu/papers/delite-scaladays13.pdf`, 2013.

[6] C. Calcagno, W. Taha, L. Huang, and X. Leroy. Implementing Multi-stage Languages Using ASTs, Gensym, and Reflection. GPCE, 2003.

[7] J. Carette, O. Kiselyov, and C. chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.*, 19(5):509–543, 2009.

[8] B. Catanzaro, M. Garland, and K. Keutzer. Copperhead: compiling an embedded data parallel language. PPoPP, 2011.

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

[10] H. Chafi, A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya, and K. Olukotun. A domain-specific approach to heterogeneous parallelism. PPoPP, 2011.

[11] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. FlumeJava: easy, efficient data-parallel pipelines. PLDI, 2010.

[12] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon. *Parallel programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[13] A. Danial. CLOC–Count Lines of Code. *Open source*, 2009.

[14] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, and P. Hanrahan. Liszt: A Domain Specific Language for Building Portable Mesh-based PDE Solvers. SC, 2011.

[15] S. Erdweg, L. C. Kats, T. Rendel, C. Kästner, K. Ostermann, and E. Visser. SugarJ: library-based language extensibility. OOPSLA, 2011.

[16] M. Eysholdt and H. Behrens. Xtext: implement your language faster than the quick and dirty way. SPLASH '10, 2010.

[17] T. E. Foundation. Eclipse Modeling Framework Project (EMF). `http://www.eclipse.org/modeling/emf/`, 2013.

[18] Google. Protocol Buffers Data Interchange Format. `http://code.google.com/p/protobuf`, 2011.

[19] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. NSDI, 2011.

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

[21] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. Green-Marl: A DSL for Easy and Efficient Graph Analysis. ASPLOS, 2012.

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

[23] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. EuroSys, 2007.

[24] JetBrains. Meta Programming System. `http://www.jetbrains.com/mps/`, 2009.

[25] S. Kandel, A. Paepcke, J. Hellerstein, and J. Heer. Wrangler: interactive visual specification of data transformation scripts. CHI '11, 2011.

[26] L. C. Kats and E. Visser. The spoofax language workbench: rules for declarative specification of languages and IDEs. OOPSLA '10, 2010.

[27] E. Meijer, B. Beckman, and G. Bierman. LINQ: Reconciling Object, Relations and XML in the .NET framework. SIGMOD, 2006.

[28] P.-A. Muller, F. Fleurey, D. Vojtisek, Z. Drey, D. Pollet, F. Fondement, P. Studer, J.-M. Jézéquel, et al. On executable meta-languages applied to model transformations. MTiP, 2005.

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

[30] N. Nystrom, D. White, and K. Das. Firepile: run-time compilation for GPUs in scala. GPCE, 2011.

[31] T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. *Commun. ACM*, 55(6):121–130, 2012.

[32] T. Rompf, N. Amin, A. Moors, P. Haller, and M. Odersky. Scala-Virtualized: Linguistic Reuse for Deep Embeddings. Higher-Order and Symbolic Computation (Special issue for PEPM'12, to appear).

[33] T. Rompf, A. K. Sujeeth, N. Amin, K. Brown, V. Jovanovic, H. Lee, M. Jonnalagedda, K. Olukotun, and M. Odersky. Optimizing Data Structures in High-Level Programs. POPL, 2013.

[34] A. K. Sujeeth, H. Lee, K. J. Brown, T. Rompf, M. Wu, A. R. Atreya, M. Odersky, and K. Olukotun. OptiML: an Implicitly Parallel Domain-Specific Language for Machine Learning. ICML, 2011.

[35] A. K. Sujeeth, T. Rompf, K. J. Brown, H. Lee, H. Chafi, V. Popic, M. Wu, A. Prokopec, V. Jovanovic, M. Odersky, and K. Olukotun. Composition and Reuse with Compiled Domain-Specific Languages. ECOOP, 2013.

[36] H. Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, 30(3):202–210, 2005.

[37] W. Taha and T. Sheard. MetaML and multi-stage programming with explicit annotations. *Theor. Comput. Sci.*, 248(1-2):211–242, 2000.

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

[39] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen. Languages as libraries. PLDI '11, 2011.

[40] Typesafe. Simple Build Tool. `http://www.scala-sbt.org`.

[41] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. NSDI, 2011.