Forge: Generating High-Performance DSL Implementations from a High-Level Specification

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

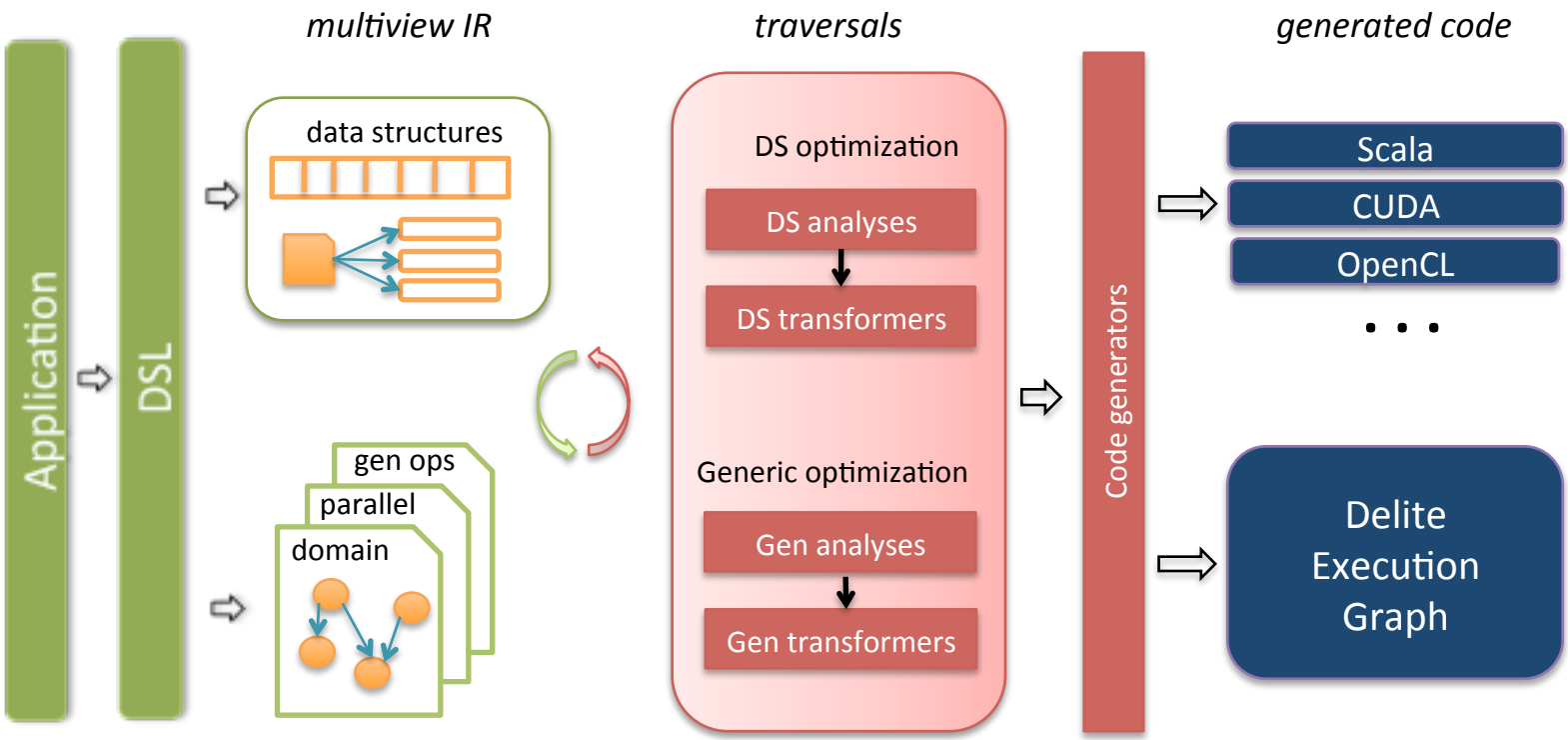
Stanford University, EPFL

DSLDI 2013
July 1, 2013

Delite: a Framework for High Performance DSLs

- A compiler toolchain for high performance embedded DSLs
- Built on top of Lightweight Modular Staging (LMS) to build an *intermediate representation* (IR) from Scala application code
- Provides extensible reusable components
  - *Parallel patterns* for structured computation
  - *Delite structs* for structured data
  - Transformers for domain-specific optimizations
- Delite optimizes DSL code and generates target code for C++, CUDA, OpenCL, and clusters

Delite Architecture



Keys To Delite Performance

- Compilers use high-level semantics
  - `MatrixTimesVector` *instead of*

```
for i <- 0 until m.numRows {
  for j <- 0 until m.numCols {
    ...
  }
}
```

- Staging programmatically removes abstraction
  - Method calls inlined, modularity structures (traits, classes) compiled away
  - Generated code is low-level and first order
- Data structures are represented in the IR
  - Struct wrappers, fields, are statically eliminated when possible
- Optimizations
  - Producer-consumer (vertical) and sibling (horizontal) fusion

More DSLs more problems

- DSL developers must be comfortable with advanced Scala and modular Delite APIs
- Significant boilerplate required to:
  - emulate DSL syntax
  - isolate the DSL program from the DSL compiler
  - extend parallel patterns
  - implement new data structures
- Debugging is painful
  - Compilation is slow
  - Interactive programming is hard (see recent work in ECOOP '13)

Forging a new way forward

- Delite is a highly-flexible compiler and IR for high performance in heterogeneous environments
- But it is too low-level for the average DSL developer
- Idea: let's raise the level of abstraction and apply DSL principles to DSL development

Forge: a "meta" DSL for DSL development

Hello, Forge

```
trait HelloWorldDSL extends ForgeApplication {
  def dslName = ”HelloWorld”

  def specification() {
    val Hello = grp(“Hello”)
    direct (Hello) (“helloWorld”, Nil :: MUnit, effect = simple)
      implements codegen(scala, ${ println(“hello, world!”) })
  }
}
```

```
trait HelloWorldApp extends HelloWorldDSLApplication {
  def main() {
    helloWorld() // what a DSL, we can only do 1 thing
  }
}
```

Hello, Forge

```
trait HelloWorldDSL extends ForgeApplication {
  def dslName = ”HelloWorld”

  def specification() {
    val Hello = grp(“Hello”)
    direct (Hello) (“helloWorld”, Nil :: MUnit, effect = simple)
      implements codegen(scala, ${ println(“hello, world!”) })
  }
}
```

Front-end syntax

```
trait HelloWorldApp extends HelloWorldDSLApplication {
  def main() {
    helloWorld() // what a DSL, we can only do 1 thing
  }
}
```

Hello, Forge

```
trait HelloWorldDSL extends ForgeApplication {
  def dslName = ”HelloWorld”

  def specification() {
    val Hello = grp(“Hello”)
    direct (Hello) (“helloWorld”, Nil :: MUnit, effect = simple)
      implements codegen(scala, ${ println(“hello, world!”) })
  }
}
```

Method signature

Front-end syntax

```
trait HelloWorldApp extends HelloWorldDSLApplication {
  def main() {
    helloWorld() // what a DSL, we can only do 1 thing
  }
}
```

Hello, Forge

```
trait HelloWorldDSL extends ForgeApplication {
  def dslName = ”HelloWorld”

  def specification() {
    val Hello = grp(“Hello”)
    direct (Hello) (“helloWorld”, Nil :: MUnit, effect = simple)
      implements codegen(scala, ${ println(“hello, world!”) })
  }
}

trait HelloWorldApp extends HelloWorldDSLApplication {
  def main() {
    helloWorld() // what a DSL, we can only do 1 thing
  }
}
```

Method signature

Compiler annotations

Front-end syntax

Hello, Forge

```
trait HelloWorldDSL extends ForgeApplication {
  def dslName = ”HelloWorld”

  def specification() {
    val Hello = grp(“Hello”)
    direct (Hello) (“helloWorld”, Nil :: MUnit, effect = simple)
      implements codegen(scala, ${ println(“hello, world!”) })
  }
}
```

Method signature

Compiler annotations

Front-end syntax

Implementation

```
trait HelloWorldApp extends HelloWorldDSLApplication {
  def main() {
    helloWorld() // what a DSL, we can only do 1 thing
  }
}
```

Hello, Forge

```
trait HelloWorldDSL extends ForgeApplication {
  def dslName = ”HelloWorld”

  def specification() {
    val Hello = grp(“Hello”)
    direct (Hello) (“helloWorld”, Nil :: MUnit, effect = simple)
      implements codegen(scala, ${ println(“hello, world!”) })
  }
}
```

Method signature

Compiler annotations

Front-end syntax

Implementation

```
trait HelloWorldApp extends HelloWorldDSLApplication {
  def main() {
    helloWorld() // what a DSL, we can only do 1 thing
  }
}
```

A Simple Vector DSL

```
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) })

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

A Simple Vector DSL

```
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) })

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

A "Simple" Vector DSL in Delite

```
trait VectorOps extends Variables {
  this: SimpleVector =>

  trait Vector[A] extends DeliteCollection[A]
   object Vector {
     def apply[A:Manifest](length: Rep[Int]) = vectorNew(length)
   }

  implicit class VectorOpsCls[A:Manifest](x: Rep[Vector[A]]) {
    def +(y: Rep[Vector[A]])(implicit n: Numeric[A]) = vectorPlus(x,y)
    def apply(idx: Rep[Int]) = vectorApply(x, idx)
    // …
  }

  def vectorNew[A:Manifest](length: Rep[Int]): Rep[Vector[A]]
  def vectorApply[A:Manifest](x: Rep[Vector[A]], idx: Rep[Int]): Rep[A]
}


trait VectorOpsExp extends VectorOps with DeliteCollectionOpsExp with DeliteStructsExp {
  this: SimpleVectorExp =>

  case class VectorPlus[A:Manifest:Numeric](inA: Exp[Vector[A]], inB: Exp[Vector[A]])
    extends DeliteOpZipWith[A,A,A,Vector[A]] {

    override def alloc(len: Exp[Int]) = Vector[A](len)
    val size = copyTransformedOrElse(_.size)(inA.length)

    def func = (a,b) => a + b
  }

  case class VectorNew[A:Manifest](len: Exp[Int]) extends DeliteStruct[Vector[A]] {
    val elems = copyTransformedElems(Seq("data" -> var_new(DeliteArrayBuffer[A](len)).e))
    val mA = manifest[A]
  }

  def vectorNew[A:Manifest](length: Exp[Int]) = reflectMutable(VectorNew(length))
  def vectorApply[A:Manifest](x: Exp[Vector[A]], idx: Exp[Int]) = x.data.apply(idx)
  def vectorPlus[A:Manifest:Numeric](x: Exp[Vector[A]], y: Exp[Vector[A]]) = reflectPure(VectorPlus(x,y))
}
```

embedded DSL implementation artifacts leaking through

A "Simple" Vector DSL in Delite

```
trait SimpleVectorScalaOpsPkg extends Base
  with Equal with IfThenElse with Variables with While with Functions
  with ImplicitOps with NumericOps with OrderingOps with StringOps
  with BooleanOps with PrimitiveOps with MiscOps with TupleOps
  with MathOps with CastingOps with ObjectOps with ArrayOps
  with DeliteArrayOps

//Exps version
trait SimpleVectorScalaOpsPkgExp extends SimpleVectorScalaOpsPkg with DSLOpsExp
  with EqualExp with IfThenElseExp with VariablesExp with WhileExp with FunctionsExp
  with ImplicitOpsExp with NumericOpsExp with OrderingOpsExp with StringOpsExp
  with BooleanOpsExp with PrimitiveOpsExp with MiscOpsExp with TupleOpsExp
  with MathOpsExp with CastingOpsExp with ObjectOpsExp with ArrayOpsExp with RangeOpsExp
  with DeliteArrayFatExp with DeliteArrayBufferOpsExp with DeliteOpsExp with StructExp

//Scala codegen version
trait SimpleVectorScalaCodeGenPkg extends ScalaGenDSLOps
  with ScalaGenEqual with ScalaGenIfThenElse with ScalaGenVariables with ScalaGenWhile with ScalaGenFunctions
  with ScalaGenImplicitOps with ScalaGenNumericOps with ScalaGenOrderingOps with ScalaGenStringOps
  with ScalaGenBooleanOps with ScalaGenPrimitiveOps with ScalaGenMiscOps with ScalaGenTupleOps
  with ScalaGenMathOps with ScalaGenCastingOps with ScalaGenObjectOps with ScalaGenArrayOps with ScalaGenRangeOps
  with ScalaGenDeliteArrayOps with ScalaGenDeliteArrayBufferOps with ScalaGenDeliteStruct
  { val IR: SimpleVectorScalaOpsPkgExp }

trait SimpleVectorCudaCodeGenPkg extends CudaGenDSLOps
  with CudaGenEqual with CudaGenIfThenElse with CudaGenVariables with CudaGenWhile with CudaGenFunctions
  with CudaGenImplicitOps with CudaGenNumericOps with CudaGenOrderingOps with CudaGenStringOps
  with CudaGenBooleanOps with CudaGenPrimitiveOps with CudaGenMiscOps
  with CudaGenMathOps with CudaGenCastingOps with CudaGenArrayOps with CudaGenRangeOps
  { val IR: SimpleVectorScalaOpsPkgExp }

trait SimpleVectorCCodeGenPkg extends CGenDSLOps
  with CGenEqual with CGenIfThenElse with CGenVariables with CGenWhile with CGenFunctions
  with CGenImplicitOps with CGenNumericOps with CGenOrderingOps with CGenStringOps
  with CGenBooleanOps with CGenPrimitiveOps with CGenMiscOps
  with CGenArrayOps with CGenRangeOps
  { val IR: SimpleVectorScalaOpsPkgExp }

/**
 * add SimpleVector functionality
 */

trait SimpleVector extends SimpleVectorScalaOpsPkg with VectorOps { this: SimpleVectorApplication => }

//additional functionality I want available in the compiler, but not in the applications
trait SimpleVectorCompiler extends SimpleVector with RangeOps {
  this: SimpleVectorApplication with SimpleVectorExp =>
}
```

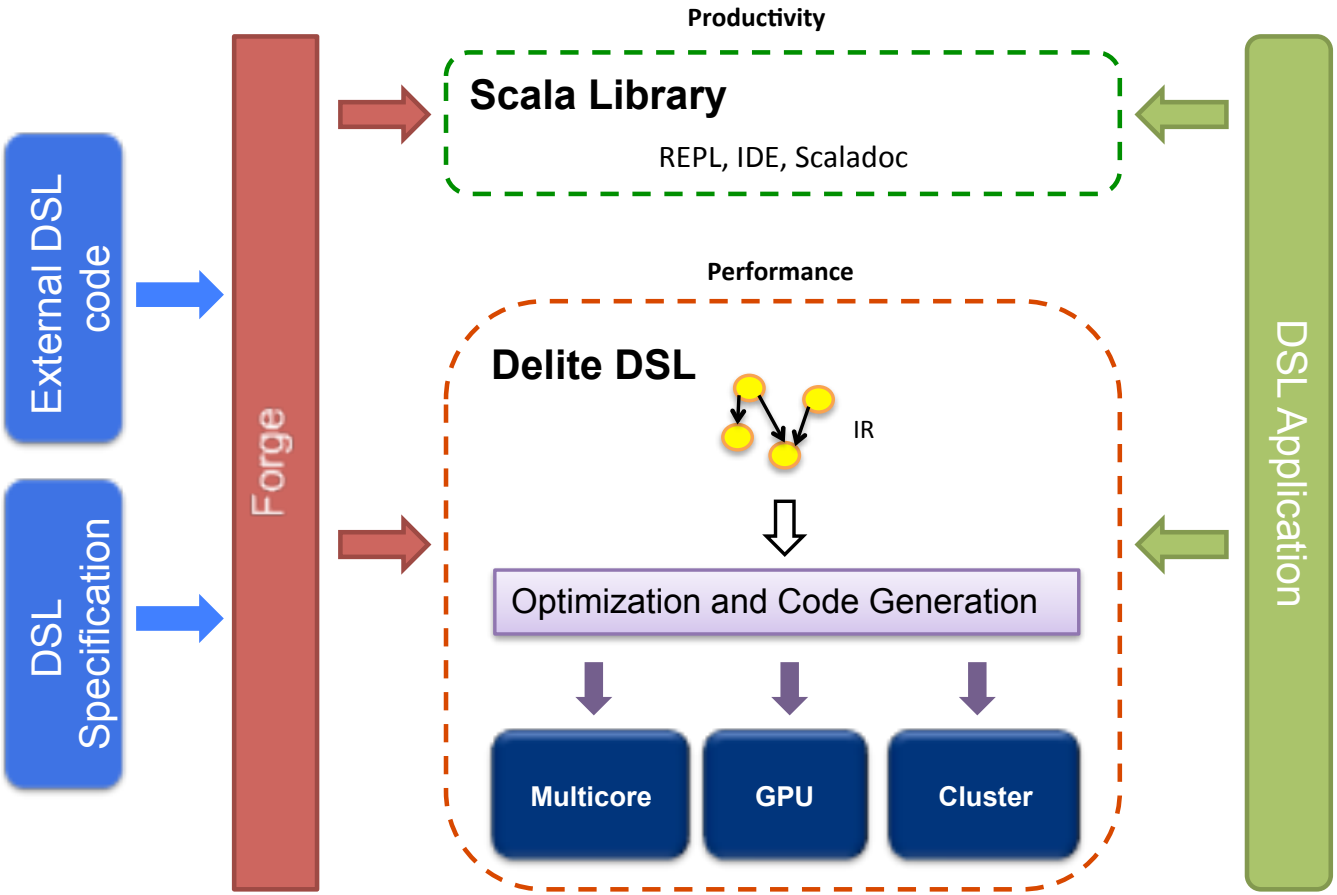
and that's not all..

A “Simple” Vector DSL in Delite

```
trait SimpleVectorExp extends SimpleVectorCompiler with SimpleVectorScalaOpsPkgExp with VectorOpsExp with VectorImplOpsStandard with DeliteOpsExp with DeliteAllOverridesExp {
  this: DeliteApplication with SimpleVectorApplication =>

  def getCodeGenPkg(t: Target{val IR: SimpleVectorExp.this.type}) : GenericFatCodegen{val IR: SimpleVectorExp.this.type} = {
    t match {
      case _:TargetScala => new SimpleVectorCodegenScala{val IR: SimpleVectorExp.this.type = SimpleVectorExp.this}
      case _:TargetCuda => new SimpleVectorCodegenCuda{val IR: SimpleVectorExp.this.type = SimpleVectorExp.this}
      case _ => throw new RuntimeException("simple vector does not support this target")
    }
  }
}

trait SimpleVectorLift extends LiftVariables with LiftEquals with LiftString with LiftNumeric with LiftBoolean {
  this: SimpleVector =>
}

//the trait all SimpleVector applications must extend
trait SimpleVectorApplication extends SimpleVector with SimpleVectorLift {
  var args: Rep[Array[String]]
  def main()
}

//the runner for SimpleVector applications
trait SimpleVectorApplicationRunner extends SimpleVectorApplication with DeliteApplication with SimpleVectorExp


trait SimpleVectorCodegenBase extends GenericFatCodegen {
  val IR: DeliteApplication with SimpleVectorExp
  override def initialDefs = IR.deliteGenerator.availableDefs

}

trait SimpleVectorCodegenScala extends SimpleVectorCodegenBase with SimpleVectorScalaCodeGenPkg
  with ScalaGenDeliteOps with ScalaGenVariantsOps with ScalaGenDeliteCollectionOps with DeliteScalaGenAllOverrides {

  val IR: DeliteApplication with SimpleVectorExp
}

trait SimpleVectorCodegenCuda extends SimpleVectorCodegenBase with SimpleVectorCudaCodeGenPkg
  with CudaGenDeliteOps with CudaGenVariantsOps with DeliteCudaGenAllOverrides {

  val IR: DeliteApplication with SimpleVectorExp
}

trait SimpleVectorCodegenC extends SimpleVectorCodegenBase with SimpleVectorCCodeGenPkg
  with CGenDeliteOps with CGenVariantsOps with DeliteCGenAllOverrides {

  val IR: DeliteApplication with SimpleVectorExp
}
```

still going

Forge Compilation Pipeline



Forge Language Overview

- Data Types
  - `ftpe`
  - `tpe`
  - `tpePar`
  - `tpeClass`
  - `tpeClassInst`
- Data structures:
  - `data`
  - `[op] allocates [data]`
  - `[op] getter [field]`
  - `[op] setter [field]`
- Methods
  - `arg`
  - `op` `[static, direct, infix, compiler, implicit]`
  - `impl` `[single, composite, map, filter, reduce, zip, foreach]`
  - *`effect`* `[simple, mutable, write]`
  - *`frequency`* `[normal, hot, cold]`
  - *`aliasHint`* `[copies]`
- Miscellaneous:
  - `grp`
  - `extern`
  - `lift`
  - `lookup`

Forge Language Overview

- Data Types
  - `ftpe`
  - `tpe`
  - `tpePar`
  - `tpeClass`
  - `tpeClassInst`

Delite parallel patterns

- Methods
  - `arg`
  - `op` [static, direct, infix, compiler, implicit]
  - `impl` [single, composite, map, filter, reduce, zip, foreach]
  - *`effect`* [simple, mutable, write]
  - *`frequency`* [normal, hot, cold]
  - *`aliasHint`* [copies]

- Data structures:
  - `data`
  - `[op] allocates [data]`
  - `[op] getter [field]`
  - `[op] setter [field]`

Delite structs

- Miscellaneous:
  - `grp`
  - `extern`
  - `lift`
  - `lookup`

Forge Implementation

- Forge itself is an embedded DSL in Scala and LMS
- Builds its own IR representing the DSL, and traverses it using different code generators (library, compiler)
- Preprocessor auto quotes “next-stage” code
- A collection of shell scripts combine and package the resulting artifacts for DSL users

Simple Vector Application

```
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:")
    println(sum(v))

    println(”v:")
    v.pprint

    val vc = v.map(e => e*(e-1))
    println("vc:")
    vc.pprint
  }
}
```

Interactive programming

```
[info] Starting scala interpreter...
[info]
import optiml.library._
OptiML: optiml.library.OptiMLApplicationInterpreter = $anon$1@56c6580c
import OptiML._
Welcome to Scala version 2.10.0 (Java HotSpot(TM) 64-Bit Server VM, Java 1.6.0_45).
Type in expressions to have them evaluated.
Type :help for more information.

scala> val v = DenseVector.rand(10)
v: OptiML.Rep[OptiML.DenseVector[Double]] = [0.7220 0.1950 0.6672 0.7784 0.6186 0.6231 0.2368 0.4872 0.680

scala> sum(v)
res0: OptiML.Rep[Double] = 5.533237117591888

scala> val vc = v.map(e => e*(e-1))
vc: OptiML.Rep[OptiML.DenseVector[Double]] = [-0.2007 -0.1570 -0.2221 -0.1725 -0.2359 -0.2348 -0.1807 -0.2

scala> vc.t
res1: OptiML.DenseVector[Double] =
[-0.2007]
[-0.1570]
[-0.2221]
[-0.1725]
[-0.2359]
[-0.2348]
[-0.1807]
[-0.2498]
[-0.2174]
-0.2494

scala>
```

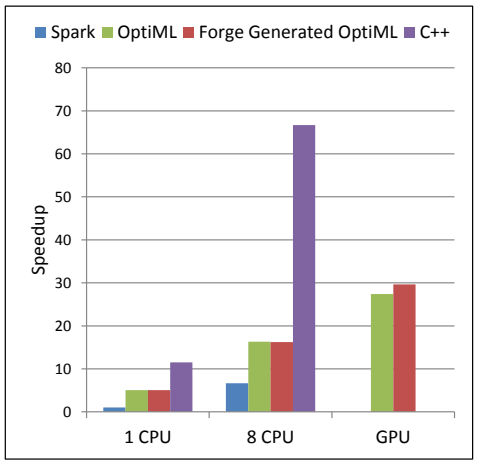
EVALUATION

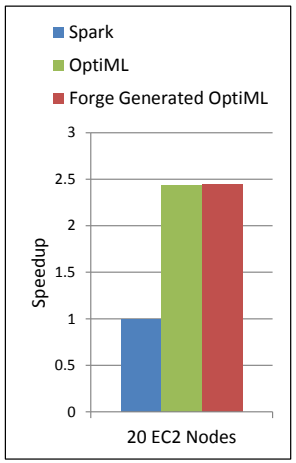
Lines of Code

DSL	Forge specification	Delite (manual)	Delite (generated)	Spark library
OptiML	1322	7416	11743	n/a
OptiQL	301	862	1287	n/a
OptiWrangler	343	n/a	1814	253

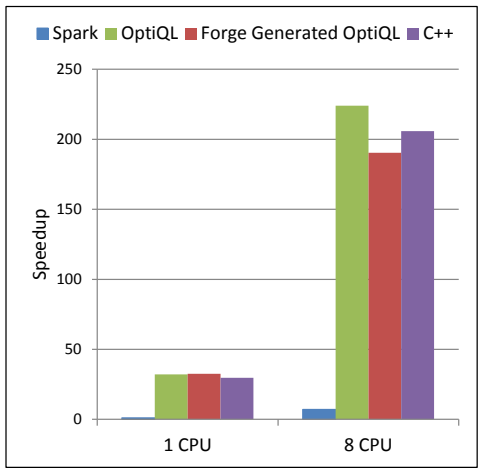
Performance

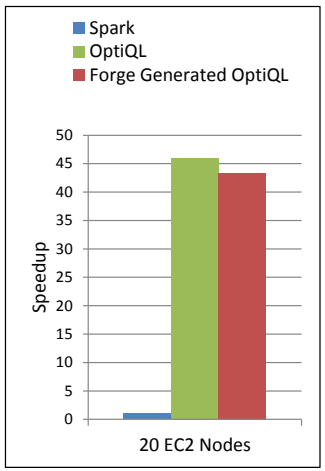
OptiML



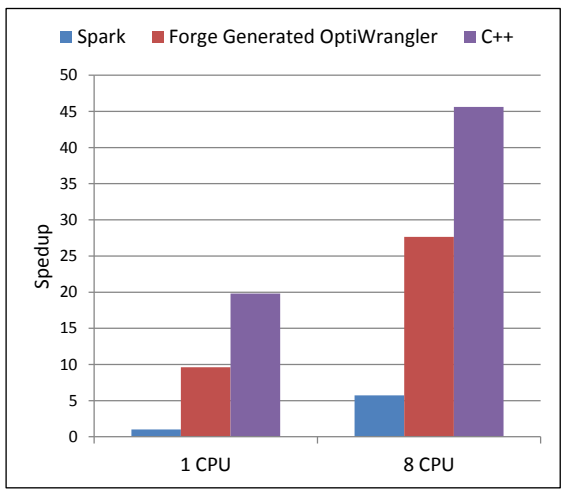


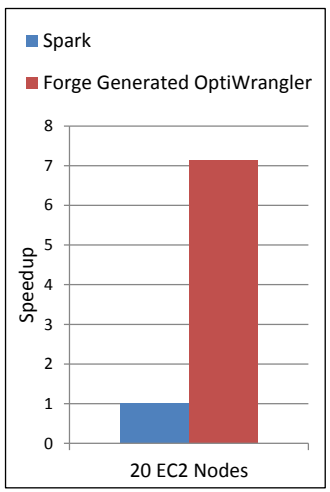
OptiQL





OptiWrangler





Conclusion

- Forge is a new "meta" DSL for high performance DSL development
- Generates a high productive (library) implementation and a high performance (Delite) implementation
  - Enables interactive programming with Delite DSLs
  - Huge reduction in LOC compared to hand-implemented Delite
- DSL users target both with the same application code using polymorphic embedding
  - Users can flip a switch to go from local sequential execution to parallel / GPU / cluster execution
- Declarative specification enables new opportunities
  - Easy to retarget to new implementations

Thank you!

Try it out and let us know what you think:

https://github.com/stanford-ppl/Forge

forge-dsl@googlegroups.com

BACKUP

Do we really need another stage?

- a.k.a., why isn't the Delite interface the Forge interface?
- Answer: the Delite interface is part of a fully functional implementation
  - Applications can compile and run against it
  - A Forge specification is abstract
- Delite shields DSL developers from changes in hardware
  - Forge shields DSL developers from changes in Delite (who shields DSL developers from changes in Forge?!)

Adding External Code

- Forge was designed to be used as a scaffolding tool with the common cases
  - Fully functional DSL can be implemented concisely
- Domain-specific optimizations (rewrites and transformations) are added externally
  - Ordinary Delite code, type-checked when compiling the generated DSL
  - Rewrites and transformations in Delite are already high-level APIs
- Forge includes a set of publishing scripts
  - Combines the external and generated code, compiles, stages, and executes
  - Uses rsync for incremental multi-stage recompilation

Syntactic Sugar with Language Virtualization

- Forge was implemented as an embedded DSL for convenience
  - but we need an expressive syntax to limit boilerplate
- A new use-case for Scala-Virtualized *scopes*: type-safe syntactic scoping

```
// outside syntactic scope
infix (Vector) (“foo”, T, ((Vector, MInt) :: Vector)) ...

val VectorOps = withTpe (Vector)  // creates a new scope
VectorOps {
  // convenience methods made available inside scope
  // with implicit self arg
  infix ("apply") (MInt :: T) implements composite $
    { vector_raw_data($self)($1) }
}
```

Scala-Virtualized scope is desugared to create an anonymous object containing the new methods

Forge Preprocessor

- A Forge spec is a multi-stage program
- It includes code that uses the (yet to be generated) DSL interface
- The Forge preprocessor enables DSL developers to write in an (almost) normal syntax
  - Blocks are quoted as strings and passed to Forge
  - Type-checked when the generated DSL is compiled

Preprocessor Example

```
infix ("slice") ((("start",MInt),("end",MInt)) :: Vector(T))
  implements single ${
    val out = Vector[T]($end - $start)
    var i = $start
    while (i < $end) {
      out(i-$start) = $self(i)
      i += 1
    }
    out
}
```

Preprocessor Example

```
infix ("slice") ((("start",MInt),("end",MInt)) :: Vector(T))
  implements single ${
    val out = Vector[T]($end - $start)
    var i = $start
    while (i < $end) {
      out(i-$start) = $self(i)
      i += 1
    }
    out
}

infix ("slice") ((("start",MInt),("end",MInt)) :: Vector(T))) implements single {
          val end = quotedArg("end")
          val start = quotedArg("start")
          val self = quotedArg("self")
          s"""val out = Vector[T]($end - $start)
var i = $start
while (i < $end) {
  out(i-$start) = $self(i)
  i += 1
}
out"""”
}
```