High Performance DSL Implementation Using Delite

Arvind K. Sujeeth, HyoukJoong Lee, Kevin J. Brown, Hassan Chafi, Kunle Olukotun
Stanford University
Pervasive Parallelism Laboratory (PPL)

Tiark Rompf, Martin Odersky
Ecole Polytechnique Federale de Lausanne (EPFL)
Programming Methods Laboratory
Outline

- High-level walk-through of major components of a Delite DSL
- DEMO: developing a very simple DSL with Delite
Components of Delite DSLs

1. Types
   - abstract, front-end

2. Operations
   - language operators and methods available on types; represented by IR nodes

3. Data Structures
   - platform-specific concrete implementation, back-end

4. Code Generators
   - Scala traits that define how to emit code as strings for various IR nodes and platforms

5. Analyses and Optimizations (Optional)
   - IR rewriting via pattern matching, traversals/transformations (e.g. fusion)
Types

- Front-end language types, defined as abstract classes
- Used for static type-checking and method dispatch
- Not tied to any back-end implementation or platform
Operations

- Language operators and methods dispatched on types

- Syntax encoded using Scala (implicit parameters, function currying, etc.)

- Implementation of operations constructs IR nodes
Data Structures

- Concrete, back-end implementation used to store state
  - Instantiated and manipulated by generated code
  - Requires an implementation per platform, e.g. Scala and CUDA
Code Generators

- Scala traits that define how to emit platform-specific code for IR nodes
- DSL author only has to define code generators for nodes that access back-end data structures
- Delite handles parallel code gen
Analyses and Transformations

- IR rewriting with pattern matching

- Traversals
  - Similar to code generators, but inspect the IR when visited

- Schedule modifications
  - Override Delite’s scheduler to transform it
    - e.g. fusion
Putting It All Together

- Everything in a Delite DSL is defined in Scala traits

- The different components are *mixed-in* to construct “packages”:
  - OpsPkg – contains all the abstract op methods
  - OpsExpPkg – contains all the op methods to construct IR nodes
  - CodeGenPkg – contains all the code generators for a platform

- Application objects are constructed from these packages
DEMO: A SIMPLE PROFILING DSL
abstract class Vector[T] extends DeliteCollection[T]

abstract class Matrix[T] extends DeliteCollection[T]

abstract class Image[T] extends Matrix[T]

placeholders for static type checking and method dispatch;
not bound to any implementation

Required interface for Delite ops
trait VectorOps {

// add an infix + operator to Rep[Vector[A]]
    def infix_+( lhs: Rep[Vector[A]], rhs: Rep[Vector[A]]) =
        vector_plus (lhs, rhs)

// abstract, applications cannot inspect what happens when
// methods are called
    def vector_length(lhs: Rep[Vector[A]]): Rep[Int]
}

The same abstract Vector we defined earlier
trait VectorOpsExp extends VectorOps with Expressions {
  // a Delite parallel op IR node
  case class VectorPlus (inA: Exp[Vector[A]], inB: Exp[Vector[A]])
      extends DeliteOpZipWith[Vector[A], Vector[A], Vector[A]] {
    // number of elements in the input collections
    def size = inA.length
    // the output collection
    def alloc = Vector[A](inA.length)
    // the ZipWith function
    def func = (a,b) => a + b
  }
  // construct IR nodes
  def vector_plus(lhs: Exp[Vector[A]], rhs: Exp[Vector[A]])
      = VectorPlus (lhs , rhs )
}
class Vector[T]( __length: Int ) {
    var _length = __length
    var _data: Array[T] = new Array[T]( _length )
}
trait ScalaGenVectorOps extends ScalaGen {
    val IR: VectorOpsExp
    import IR._

    override def emitNode (sym: Sym[Any], rhs: Def[Any]) (implicit stream: PrintWriter) =

    rhs match {
        case v@VectorNew (length) =>
            emitValDef (sym , " new " + remap("Vector ")+"(" + quote(length) + "]")
        case VectorLength(x) =>
            emitValDef (sym, quote(x) + ". _length")
        case _ => super.emitNode (sym, rhs)
    }
}
override def matrix_plus [A:Manifest:Arith]
  (x: Exp[Matrix[A]], y: Exp[Matrix[A]]) =

  (x, y) match {
    // (AB + AD) == A(B + D)
    case (Def(MatrixTimes (a, b)), Def(MatrixTimes(c, d)))
      if (a == c) =>
        matrix_times (a, matrix_plus (b,d))
    // ...
    case _ => super.matrix_plus (x, y)
  }

Putting It All Together

```scala
trait OptiML extends OptiMLScalaOpsPkg with VectorOps with MatrixOps \ with ...

trait OptiMLExp extends OptiMLScalaOpsPkgExp with VectorOpsExp with MatrixOpsExp \ with ...

trait OptiMLCodeGenScala extends OptiMLScalaCodeGenPkg with ScalaGenVectorOps with ScalaGenMatrixOps \ with ...

trait OptiMLCodeGenCuda extends OptiMLCudaCodeGenPkg with CudaGenVectorOps with CudaGenMatrixOps \ with ...
```