Future Directions in DSL Research

Hassan Chafi, Arvind Sujeeth, Kevin Brown, Sungpack Hong, Zach Devito and Kunle Olukotun
Oracle Labs
Stanford University
Required Support For Multiple DSLs

- Each new DSL supported by Delite introduces some new requirement
  - New Ops
  - New optimization requirements
- The good news is that we are finding a lot of re-use between different DSLs which validates the need for an infrastructure
DSLs Implemented So Far

- OptiML
  - Original DSL
- OptiQL
  - LINQ like on Delite
- OptiGraph
  - Green-Marl on Delite
- OptiMesh
  - Liszt on Delite
- OptiCollections
  - Scala Collections on Delite
## Re-Use Across DSLs

<table>
<thead>
<tr>
<th>DSL</th>
<th>Delite Ops</th>
<th>Generic Optimizations</th>
</tr>
</thead>
<tbody>
<tr>
<td>OptiML</td>
<td>Map, ZipWith, Reduce, Foreach, Hash, Sort</td>
<td>CSE, DCE, code motion, fusion</td>
</tr>
<tr>
<td>OptiQL</td>
<td>Map, Reduce, Filter, Sort, Hash, Join</td>
<td>CSE, DCE, code motion, fusion</td>
</tr>
<tr>
<td>OptiGraph</td>
<td>ForEachReduce, Map, Reduce, Filter</td>
<td>CSE, DCE, code motion</td>
</tr>
<tr>
<td>OptiMesh</td>
<td>ForEachReduce</td>
<td>CSE, DCE, code motion</td>
</tr>
<tr>
<td>OptiCollections</td>
<td>Map, Reduce, Filter, Sort, Hash, ZipWith</td>
<td>CSE, DCE, code motion, fusion</td>
</tr>
</tbody>
</table>
Restrictions & Semantics

- The domain-specific restrictions and semantics are currently implementing in an adhoc fashion by DSL author.
- Some restrictions and semantics of DSL operations are currently provided by the infrastructure.
  - Side-effect management
- Can we generalize and provide a framework for DSL authors to specify the semantics and restrictions imposed by a DSL declaratively?
**DSL Extensibility**

- Need to be able to start with a DSL or part of a DSL and extend it with some extra domain-specific optimizations and constructs.
- How does extensibility interact with the restrictions and semantics of the DSLs?
OptiML extends OptiLA

- \texttt{trait} OptiLA extends OptiLAScalaOpsPkg with LanguageOps with ArithOps with VectorOps // with ...

- \texttt{trait} OptiML extends OptiLA with OptiMLScalaOpsPkg with OptiMLVectorOps with StreamOps // with ...
How does the DSL interoperate with the host language?

- Currently we lift the host language into the DSL
- The non-lifted parts are staged away during the first part of compilation

Interesting applications will also use multiple DSLs

How should these DSLs interoperate?

At what level of granularity?
OptiML and OptiQL interoperability

```scala
val orderData = DeliteRef[Array[Record]]()
val theta = DeliteRef[Array[Double]]()
OptiQL {
    // customers: Array[Customer], orders: Array[Order]
    val orders = customers Join(orders)
    WhereEq(_.c_custkey, _.o_custkey)
    Select((c,o) => new Result {
        val nationKey = c.c_nationkey
        val acctBalance = c.c_acctbal
        val price = o.o_totalprice
    })
    orderData.set(orders)
}
OptiML {
    // run linear regression on price
    val data = Matrix(orderData.get.map(t =>
        Array(t.getDouble(1), t.getDouble(2), t.getDouble(3))))
    val x = data.sliceCols(0,1)
    val y = data.getCol(2)
    theta.set(linreg.weighted(x,y).toArray)
}
println("theta: " + theta.get)
```
Abstracting Data Structures

- Seen how to abstract code in Delite and retarget it to different hw resources
- Need to do the same for data representation
  - current status of stable Delite version is that the DSL author is responsible for providing concrete implementations of the back-end data structures
Data Structures in Delite

- **Current:** DSL developers implement their own data structures for each target device

- **New:** everything is a Struct/Record, specified programmatically
  - instantiation and field access lifted into IR

```scala
def Complex(re: Rep[Double], im: Rep[Double]) =
  new Record { val real = re; val imag = im }
  //lifted to __new and forwarded to Delite

val x = Complex(0,0)
x.real //type-checked that field “real” exists, then forwarded to Delite
```
Why Records?

- We can auto-generate the back-end implementation to different platforms
  - Supported by large number of target languages

- We can reason about which part of the record is actually used
  - Unwrap the record and just pass around required fields in generated code
  - Unused fields can be eliminated all together

- We can perform automatic AoS -> SoA conversion
  - Instantiate only arrays of primitives in the generated code
AoS -> SoA Optimization

- Provide familiar AoS form to the DSL developer
- Perform SoA transformations transparently for DeliteOps
  - Functions returning record types split into result for each component
  - Create separate loop to compute each component
  - Unused components are dead-code-eliminated
  - Loop fusion recombines live components into single loop
Integrating with Control Flow

```scala
val b = if (x > 7) a.map(conj) else a
```

- ‘a’ exists in IR as (“real” -> Rep[Array[Double]], “imag” -> Rep[Array[Double]])
- a.map is split into map for each component
  - map over real component is optimized away (identity function)
- real array is unaffected by conditional, simply re-used in generated code
Abstracting DSL Analysis and Transformation

- Delite currently supports abstractions for parallel execution patterns
- Delite supports optimizations on IR nodes it understands
  - This benefits DSL nodes as well
- Support for Domain-Specific analysis and IR transform in Delite is limited to IR creation time
- More complex use cases are handled by the DSL author in an adhoc fashion
- Need to support and abstract more use cases for optimization and transformation
Planned Analyses and Transformations Framework

- DSL author defines transformation rules as pattern match on IR nodes
  - As we have done when IR nodes are created
- Delite provides IR traversal patterns
  - bottom-up
  - top-down
  - ...

DSL Tooling and IDEs

- With embedding can re-use host language but...
  - Very cryptic error messages
  - Refactoring support and other IDE goodies are missing
  - Difficult to debug generated code and trace it back to original DSL code
Debugging Support for DSLs

- With DSLs, you can code at very high-level, it should be possible to debug at a very high level
  - Domain-specific debuggers
  - Domain-specific visualizer
Debugging (2)

Highlighted DSL Source:

```scala
y_zeros = y count { _ == false }
y_ones = y count { _ == true }
mu0_num = sumIf[Vector[Double]](0,m)
m1_num = sumIf[Vector[Double]](0,m)
```

- **Start time:** 139751us
- **Duration:** 26410us
- **OType:** OP_MultiLoop
- **Kernel:** mu1_num206
- **Source:** GDA.scala:43
Host Languages for DSLs

- Shown how we can modify Scala Compiler and make the language more overloadable and useful for high-performance embedding

- Still some issues
  - Can’t overload class definition
  - Can’t overload exceptions

- What would a host language built from the ground up for high-performance DSL embedding look like?