Building-Blocks for Performance Oriented DSLs

Tiark Rompf, Martin Odersky
EPFL

Arvind Sujeeth, HyoukJoong Lee, Kevin Brown, Hassan Chafi, Kunle Olukotun
Stanford University
Make **programmers** more productive

- Raise the level of abstraction
- Easier to reason about programs
- Maintenance, verification, etc
Performance Oriented DSLs

Make **compiler** more productive, too!

- Generate better code
- Optimize using domain knowledge
- Target heterogeneous + parallel hardware
DSLs under Development

- Liszt (mesh based PDE solvers)
  - DeVito et al.: Liszt: A Domain-Specific Language for Building Portable Mesh-based PDE solvers. Supercomputing (SC) 2011

- OptiML (machine learning)

- OptiQL (data query)

- all embedded in Scala
- heterogeneous compilation (multi core CPU/GPU)
- good absolute performance and speedups
Common DSL Infrastructure

- Don’t start from scratch for each new DSL
  - It’s just too hard ...

- Delite Framework + Runtime
  - See also Brown et al.: A Heterogeneous Parallel Framework for Domain-Specific Languages. PACT’11

- This Talk/Paper: Building blocks that work together in new or interesting ways
Focus on 2 things:

#1: DeliteOps
- high-level view of common execution patterns (i.e. loops)
- parallelism and heterogeneous targets

#2: Staging
- DSL programs are program *generators*
- move (costly) abstraction to generating stage

Case study: SPADE app in OptiML
#1: DeliteOps
Heterogeneous Parallel Programming

Today:

Performance = heterogeneous + parallel
Compilers have not kept pace!

Your favourite Java, Haskell, Scala, C++ compiler will not generate code for these platforms.
Programmability Chasm

Too many different programming models

Applications

Scientific Engineering
Virtual Worlds
Personal Robotics
Data informatics

Pthreads
OpenMP
CUDA
OpenCL
Verilog
VHDL
MPI

Sun T2
Nvidia Fermi
Altera FPGA
Cray Jaguar

Virtual Worlds
Personal Robotics
Data informatics

Scientific Engineering
DeliteOps

- Capture common parallel execution patterns
  - map, filter, reduce, ... join, bfs, ...

- Map them efficiently to a variety of target platforms
  - Multi core CPU, GPU

- Express your DSL as DeliteOps
  - => Parallelism for free!
Delite DSL Compiler

- Provide a common IR that can be extended while still benefiting from generic analysis and opt.
- Extend common IR and provide IR nodes that encode data parallel execution patterns
- Now can do parallel optimizations and mapping
- DSL extends appropriate data parallel nodes for their operations
- Now can do domain-specific analysis and opt.
- Generate an execution graph, kernels and data structures

Scala Embedding Framework

Delite Parallelism Framework

Intermediate Representation (IR)

- Base IR
- Delite IR
- DS IR

Generic Analysis & Opt.
- Parallelism Analysis, Opt. & Mapping
- Domain Analysis & Opt.

Code Generation

- Delite Execution Graph
- Kernels (Scala, C, Cuda, MPI, Verilog, …)
- Data Structures (arrays, trees, graphs, …)
Delite Op Fusion

- Operates on all loop-based ops
- Reduces op overhead and improves locality
  - Elimination of temporary data structures
  - Merging loop bodies may enable further optimizations
- Fuse both dependent and side-by-side operations
  - Fused ops can have multiple inputs + outputs
- Algorithm: fuse two loops if
  - \( \text{size(loop1)} == \text{size(loop2)} \)
  - No mutual dependencies (which aren’t removed by fusing)
def square(x: Rep[Double]) = x*x

def mean(xs: Rep[Array[Double]]) = xs.sum / xs.length

def variance(xs: Rep[Array[Double]]) = xs.map(square) / xs.length - square(mean(xs))

val array1 = Array.fill(n) { i => 1 }
val array2 = Array.fill(n) { i => 2*i }
val array3 = Array.fill(n) { i => array1(i) + array2(i) }

val m = mean(array3)
val v = variance(array3)

println(m)
println(v)

3+1+(1+1) = 6 traversals, 4 arrays

// begin reduce x47,x51,x11
var x47 = 0
var x51 = 0
var x11 = 0
while (x11 < x0) {
  val x44 = 2.0*x11
  val x45 = 1.0+x44
  val x50 = x45*x45
  x47 += x45
  x51 += x50
  x11 += 1
}
// end reduce
val x48 = x47/x0
val x49 = println(x48)
val x52 = x51/x0
val x53 = x48*x48
val x54 = x52-x53
val x55 = println(x54)

1 traversal, 0 arrays
#2: Staging

How do we go from DSL source to DeliteOps?
2 Challenges:

- #1: generate intermediate representation (IR) from DSL code embedded in Scala
- #2: do it in such a way that the IR is free from unnecessary abstraction
- Avoid abstraction penalty!
Example

DSL program

```scala
val v = Vector.rand(100)
println("today's lucky number is: ")
println(v.sum)
```

DSL interface

```scala
abstract class Vector[T]
def vector_rand(n: Rep[Int]): Rep[Vector[Double]]
def infix_sum[T:Numeric](v: Rep[Vector[T]]): Rep[T]
```

dsl imple.

```scala
case class VectorRand(n: Exp[Int]) extends Def[Vector[Double]]
case class VectorSum[T:Numeric](in: Exp[Vector[T]]) extends DeliteOpReduce[Exp[T]] {
def func = (a,b) => a + b
}
def vector_rand(n: Exp[Int]) = new VectorRand(n)
def infix_sum[T:Numeric](v: Exp[Vector[T]]) = new VectorSum(v)
```
“Finally Tagless” / Polymorphic embedding

- Hofer, Ostermann, Rendel, Moors: Polymorphic Embeddings of DSLs. GPCE’08.

Lightweight Modular Staging (LMS)

Can use the full host language to compose DSL program fragments!

Move (costly) abstraction to the generating stage
Example

- Use higher order functions in DSL programs
- While keeping the DSL first order!
Higher-Order functions

```scala
val xs: Rep[Vector[Int]] = ...
println(xs.count(x => x > 7))
```

```scala
def infix_foreach[A](v: Rep[Vector[A]])(f: Rep[A] => Rep[Unit]) = {
  var i: Rep[Int] = 0
  while (i < v.length) {
    f(v(i))
    i += 1
  }
}
```

```scala
  var c: Rep[Int] = 0
  v foreach { x => if (f(x)) c += 1 }
  c
}
```

```scala
val v: Array[Int] = ...
val c = 0
var i = 0
while (i < v.length) {
  val x = v(i)
  if (x > 7) {
    c += 1
    i += 1
  }
}
println(c)
```
Continuations

```
val u, v, w: Rep[Vector[Int]] = ...
nondet {
  val a = amb(u)
  val b = amb(v)
  val c = amb(w)
  require(a^2 + b^2 == c^2)
  println("found:")
  println(a, b, c)
}
```

```
def amb[T](xs: Rep[Vector[T]]): Rep[T] @cps[Rep[Unit]]
  xs foreach k
}
def require(x: Rep[Boolean]): Rep[Unit] @cps[Rep[Unit]] = shift {
  k =>
  if (x) k() else ()
}
```
Result

- Function values and continuations translated away by staging

- Control flow strictly first order

- Much simpler analysis for other optimizations
Regular Compiler optimizations

- Common subexpression and dead code elimination
- Global code motion
- Symbolic execution / pattern rewrites

Coarse-grained: optimizations can happen on vectors, matrices or whole loops
In the Paper:

- Removing data structure abstraction
- Partial evaluation/symbolic execution of staged IR
- Effect abstractions
- Extending the framework/modularity
Case Study: OptiML

A DSL For Machine Learning
OptiML: A DSL For Machine Learning

- Provides a familiar (MATLAB-like) language and API for writing ML applications
  - Ex. `val c = a * b` (a, b are `Matrix[Double]`)

- Implicitly parallel data structures
  - General data types: `Vector[T]`, `Matrix[T]`, `Graph[V,E]`
    - Independent from the underlying implementation
  - Specialized data types: `Stream`, `TrainingSet`, `TestSet`, `IndexVector`, `Image`, `Video` ..
    - Encode semantic information & structured, synchronized communication

- Implicitly parallel control structures
  - `sum{...}`, `(0::end) {...}`, `gradient { ... }`, `untilconverged { ... }
  - Allow anonymous functions with restricted semantics to be passed as arguments of the control structures
Putting it all together: SPADE

Downsample:

L1 distances between all $10^6$ events in 13D space... reduce to 50,000 events

```scala
val distances = Stream[Double](data.numRows, data.numRows){
  (i,j) => dist(data(i),data(j))
}
for (row <- distances.rows) {
  if(densities(row.index) == 0) {
    val neighbors = row find { _ < apprxWidth }
    densities(neighbors) = row count { _ < kernelWidth }
  }
}
```
SPADE transformations

```scala
val distances = Stream[Double](data.numRows, data.numRows){
    (i,j) => dist(data(i),data(j))
}

for (row <- distances.rows) {
    row.init  // expensive! part of the stream foreach operation
    if(densities(row.index) == 0) {
        val neighbors = row find { _ < apprxWidth }
        densities(neighbors) = row count { _ < kernelWidth }
    }
}
```

row is 235,000 elements in one typical dataset – fusing is a big win!
// FOR EACH ELEMENT IN ROW
while (x155 < x61) {
    val x168 = x155 * x64
    var x180 = 0

    // INITIALIZE STREAM VALUE (dist(i,j))
    while (x180 < x64) {
        val x248 = x164 + x180
        // ...
    }

    // VECTOR FIND
    if (x245) x201.insert(x201.length, x155)

    // VECTOR COUNT
    if (x246) {
        val x207 = x208 + 1
        x208 = x207
    }
    x155 += 1
}

From a ~5 line algorithm description in OptiML...

...to an efficient, fused, imperative version that closely resembles a hand-optimized C++ baseline!
Impact of Op Fusion

![Impact of Op Fusion Chart](chart.jpg)

- C++
- OptiML Fusing
- OptiML No Fusing

Normalized Execution Time

<table>
<thead>
<tr>
<th>Processors</th>
<th>C++</th>
<th>OptiML Fusing</th>
<th>OptiML No Fusing</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.9</td>
<td>1.0</td>
<td>0.3</td>
</tr>
<tr>
<td>2</td>
<td>1.8</td>
<td>1.9</td>
<td>0.6</td>
</tr>
<tr>
<td>4</td>
<td>3.3</td>
<td>3.4</td>
<td>0.9</td>
</tr>
<tr>
<td>8</td>
<td>5.6</td>
<td>5.8</td>
<td>1.0</td>
</tr>
</tbody>
</table>
Experiments on larger apps

<table>
<thead>
<tr>
<th></th>
<th>OptiML</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>TM</td>
<td>1.0</td>
<td>0.7</td>
</tr>
<tr>
<td></td>
<td>1.7</td>
<td>3.1</td>
</tr>
<tr>
<td></td>
<td>2.5</td>
<td>4.9</td>
</tr>
<tr>
<td>LBP</td>
<td>1.2</td>
<td>1.5</td>
</tr>
<tr>
<td></td>
<td>1.7</td>
<td>2.5</td>
</tr>
<tr>
<td></td>
<td>3.5</td>
<td>3.3</td>
</tr>
<tr>
<td>SPADE</td>
<td>1.0</td>
<td>0.9</td>
</tr>
<tr>
<td></td>
<td>1.9</td>
<td>1.8</td>
</tr>
<tr>
<td></td>
<td>3.4</td>
<td>3.3</td>
</tr>
<tr>
<td></td>
<td>5.8</td>
<td>5.6</td>
</tr>
</tbody>
</table>
Experiments on ML kernels

- **GDA**
- **Naive Bayes**
- **K-means**
- **SVM**
- **Linear Regression**
- **RBM**

**Normalized Execution Time**

- **OptiML**
- **Parallelized MATLAB**
- **MATLAB + Jacket**

### GDA

<table>
<thead>
<tr>
<th>CPU</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>CPU + GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.0</td>
<td>1.6</td>
<td>1.8</td>
<td>1.9</td>
<td>4.1</td>
</tr>
<tr>
<td>2</td>
<td>1.0</td>
<td>1.9</td>
<td>3.8</td>
<td>5.8</td>
<td>11.0</td>
</tr>
<tr>
<td>4</td>
<td>1.3</td>
<td>2.6</td>
<td>2.6</td>
<td>2.6</td>
<td>13.2</td>
</tr>
<tr>
<td>8</td>
<td>4.1</td>
<td>4.1</td>
<td>4.1</td>
<td>4.1</td>
<td>4.1</td>
</tr>
<tr>
<td>CPU + GPU</td>
<td>0.5</td>
<td>0.1</td>
<td>0.2</td>
<td>0.3</td>
<td>0.3</td>
</tr>
</tbody>
</table>

### Naive Bayes

<table>
<thead>
<tr>
<th>CPU</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>CPU + GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.0</td>
<td>1.9</td>
<td>3.8</td>
<td>5.8</td>
<td>11.0</td>
</tr>
<tr>
<td>2</td>
<td>1.9</td>
<td>0.2</td>
<td>0.2</td>
<td>0.3</td>
<td>0.3</td>
</tr>
<tr>
<td>4</td>
<td>4.1</td>
<td>4.1</td>
<td>4.1</td>
<td>4.1</td>
<td>4.1</td>
</tr>
<tr>
<td>8</td>
<td>5.8</td>
<td>5.8</td>
<td>5.8</td>
<td>5.8</td>
<td>5.8</td>
</tr>
<tr>
<td>CPU + GPU</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
</tr>
</tbody>
</table>

### K-means

<table>
<thead>
<tr>
<th>CPU</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>CPU + GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.0</td>
<td>2.1</td>
<td>4.1</td>
<td>7.1</td>
<td>2.3</td>
</tr>
<tr>
<td>2</td>
<td>2.1</td>
<td>0.4</td>
<td>0.4</td>
<td>0.4</td>
<td>0.4</td>
</tr>
<tr>
<td>4</td>
<td>4.1</td>
<td>4.1</td>
<td>4.1</td>
<td>4.1</td>
<td>4.1</td>
</tr>
<tr>
<td>8</td>
<td>7.1</td>
<td>7.1</td>
<td>7.1</td>
<td>7.1</td>
<td>7.1</td>
</tr>
<tr>
<td>CPU + GPU</td>
<td>0.3</td>
<td>0.3</td>
<td>0.3</td>
<td>0.3</td>
<td>0.3</td>
</tr>
</tbody>
</table>

### SVM

<table>
<thead>
<tr>
<th>CPU</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>CPU + GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.0</td>
<td>1.9</td>
<td>3.1</td>
<td>4.2</td>
<td>4.2</td>
</tr>
<tr>
<td>2</td>
<td>1.9</td>
<td>1.4</td>
<td>1.4</td>
<td>1.4</td>
<td>1.4</td>
</tr>
<tr>
<td>4</td>
<td>3.1</td>
<td>4.2</td>
<td>4.2</td>
<td>4.2</td>
<td>4.2</td>
</tr>
<tr>
<td>8</td>
<td>4.2</td>
<td>4.2</td>
<td>4.2</td>
<td>4.2</td>
<td>4.2</td>
</tr>
<tr>
<td>CPU + GPU</td>
<td>0.1</td>
<td>0.2</td>
<td>0.2</td>
<td>0.2</td>
<td>0.2</td>
</tr>
</tbody>
</table>

### Linear Regression

<table>
<thead>
<tr>
<th>CPU</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>CPU + GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.0</td>
<td>1.4</td>
<td>1.9</td>
<td>2.3</td>
<td>1.6</td>
</tr>
<tr>
<td>2</td>
<td>1.4</td>
<td>0.9</td>
<td>0.9</td>
<td>1.1</td>
<td>1.1</td>
</tr>
<tr>
<td>4</td>
<td>1.9</td>
<td>1.3</td>
<td>1.3</td>
<td>1.1</td>
<td>1.1</td>
</tr>
<tr>
<td>8</td>
<td>2.3</td>
<td>2.3</td>
<td>2.3</td>
<td>2.3</td>
<td>2.3</td>
</tr>
<tr>
<td>CPU + GPU</td>
<td>0.4</td>
<td>0.4</td>
<td>0.4</td>
<td>0.4</td>
<td>0.4</td>
</tr>
</tbody>
</table>

### RBM

<table>
<thead>
<tr>
<th>CPU</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>CPU + GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.0</td>
<td>1.7</td>
<td>2.7</td>
<td>3.5</td>
<td>4.7</td>
</tr>
<tr>
<td>2</td>
<td>1.7</td>
<td>1.9</td>
<td>1.9</td>
<td>3.5</td>
<td>4.7</td>
</tr>
<tr>
<td>4</td>
<td>2.7</td>
<td>3.2</td>
<td>3.2</td>
<td>3.2</td>
<td>3.2</td>
</tr>
<tr>
<td>8</td>
<td>3.5</td>
<td>3.5</td>
<td>3.5</td>
<td>3.5</td>
<td>3.5</td>
</tr>
<tr>
<td>CPU + GPU</td>
<td>8.9</td>
<td>8.9</td>
<td>8.9</td>
<td>8.9</td>
<td>8.9</td>
</tr>
</tbody>
</table>
Summary

- Performance oriented DSLs are a promising parallel programming platform
  - Capable of achieving portability, productivity, and high performance

- Delite can simplify the task of implementing DSLs

- OptiML outperforms MATLAB and C++ on a set of well known machine learning applications, with expressive code
Questions?
Programming Language Design Space

Performance

Productivity

Generality
Programming Language Design Space

Performance

Productivity

Generality
General Purpose Languages

- Performance
- Productivity
- Generality

Performance-oriented DSLs

Languages:
- C/C++
- Java
- Python
- Ruby
We need to develop all these DSLs

Current DSL methods are unsatisfactory
Current DSL Development Approaches

- Stand-alone DSLs
  - Can include extensive optimizations
  - Enormous effort to develop to a sufficient degree of maturity
    - Actual Compiler/Optimizations
    - Tooling (IDE, Debuggers,...)
  - Interoperation between multiple DSLs is very difficult

- Purely embedded DSLs ⇒ “just a library”
  - Easy to develop (can reuse full host language)
  - Easier to learn DSL
  - Can Combine multiple DSLs in one program
  - Can Share DSL infrastructure among several DSLs
  - Hard to optimize using domain knowledge
  - Target same architecture as host language

Need to do better
- DSLs: trade off generality for productivity and performance

- DSL embedding:
  - Combine benefits of pure embedding with analyzability of external dsls