Building-Blocks for Performance Oriented DSLs

Tiark Rompf, Martin Odersky
EPFL

Arvind Sujeeth, HyoukJoong Lee, Kevin Brown, Hassan Chafi, Kunle Olukotun
Stanford University

DSL Benefits

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)
  - Sujeeth et al.: OptiML: An Implicitly Parallel Domain-Specific Language for Machine Learning. International Conference for Machine Learning (ICML) 2011
- 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

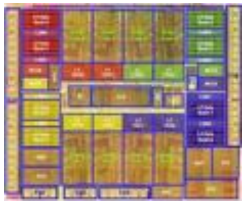
#1: DeliteOps

Heterogeneous Parallel Programming

Today:

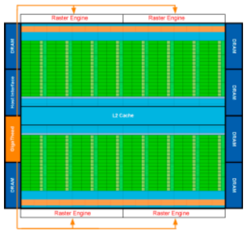
Performance
= heterogeneous
+ parallel

Pthreads
OpenMP



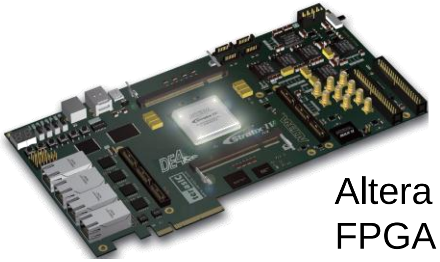
Sun
T2

CUDA
OpenCL



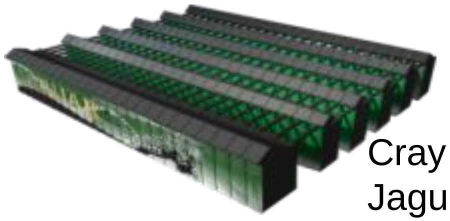
Nvidia
Fermi

Verilog
VHDL



Altera
FPGA

MPI



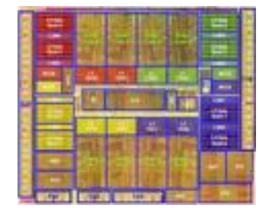
Cray
Jaguar

Heterogeneous Parallel Programming

Compilers have not kept pace!

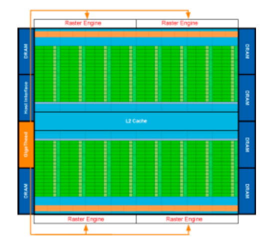
Your favourite Java, Haskell, Scala, C++ compiler will not generate code for these platforms.

Pthreads
OpenMP



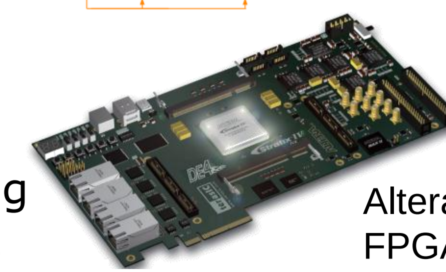
Sun
T2

CUDA
OpenCL



Nvidia
Fermi

Verilog
VHDL



Altera
FPGA

MPI

Cray
Jaguar

Programmability Chasm

Applications

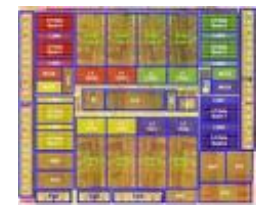
Scientific Engineering

Virtual Worlds

Personal Robotics

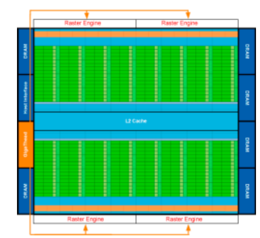
Data informatics

Pthreads
OpenMP



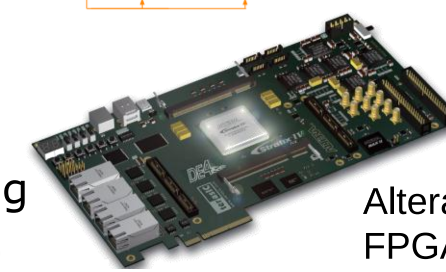
Sun
T2

CUDA
OpenCL



Nvidia
Fermi

Verilog
VHDL



Altera
FPGA

MPI

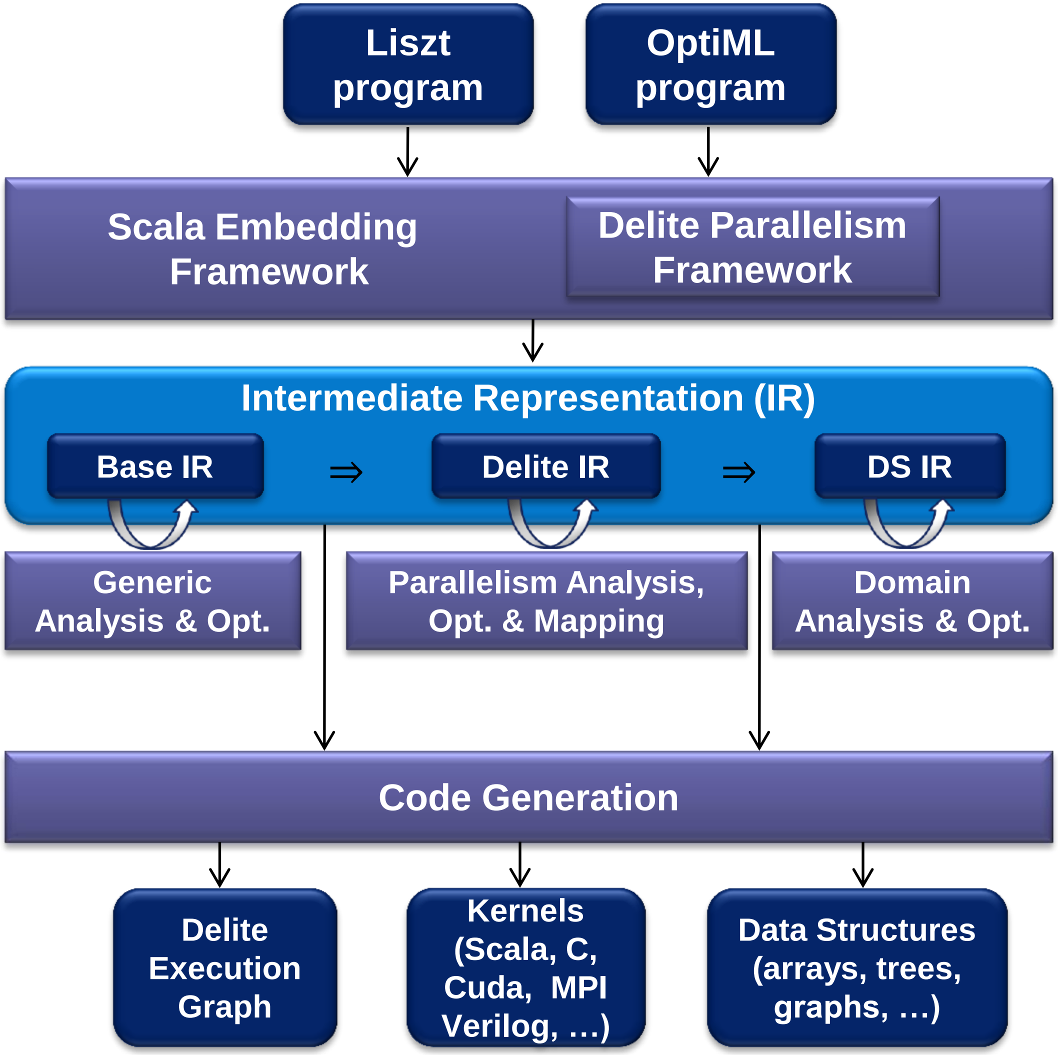
Cray
Jaguar

Too many different programming models

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



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
  - size(loop1) == size(loop2)
  - No mutual dependencies (which aren't removed by fusing)

Delite Op Fusion

```
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(me


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

```
val v = Vector.rand(100)

println("today's lucky number is: ")
println(v.sum)
```

DSL interface

```
abstract class Vector[T]

def vector_rand(n: Rep[Int]): Rep[Vector[Double]]

def infix_sum[T:Numeric](v: Rep[Vector[T]]): Rep[T]
```

```
type Rep[T]
```

DSL imlpl.

```
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)
```

```
type Rep[T] = Exp[T]

class Exp[T]

class Def[T]
```

- “Finally Tagless” / Polymorphic embedding
  - Carette, Kiselyov, Shan: Finally Tagless, Partially Evaluated: Tagless Staged Interpreters for Simpler Typed Languages. APLAS’07/J. Funct. Prog. 2009.
  - Hofer, Ostermann, Rendel, Moors: Polymorphic Embeddings of DSLs. GPCE’08.

- Lightweight Modular Staging (LMS)
  - Rompf, Odersky: Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. GPCE’10.

- 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

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

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

```
def infix_count[A](v: Rep[Vector[A]])(f: R
  var c: Rep[Int] = 0
  v foreach { x => if (f(x)) c += 1 }
  c
}
```

```
val v: Array[Int] = ...
var 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*a + b*b == c*c)
  println("found:")
  println(a,b,c)
}
```

```
while (…) {
  while (…) {
    while (…) {
      if (…) {
        println("found:")
        println(a,b,c)
      }
    }
  }
}
```

```
def amb[T](xs: Rep[Vector[T]]): Rep[T] @cps[R
  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

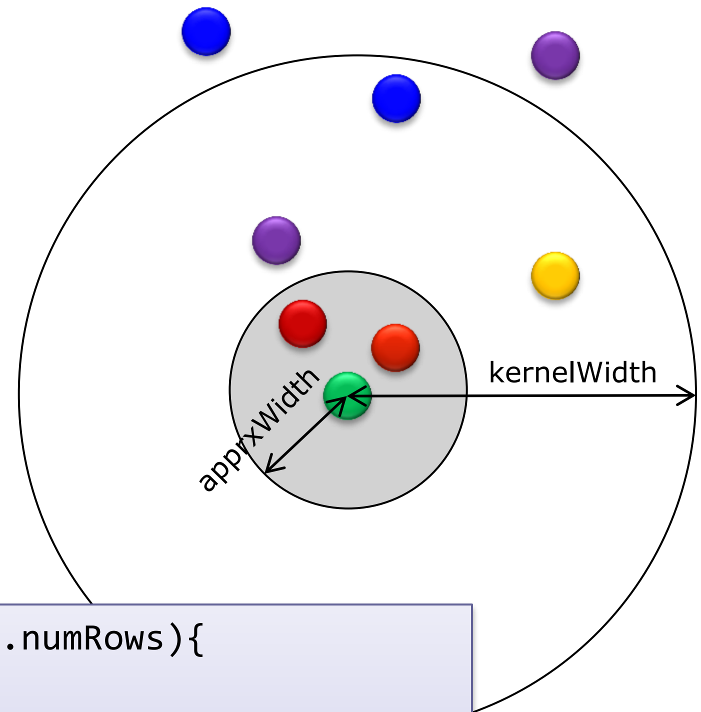
OptiML: A DSL For Machine Learning

- 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




```
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

```
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!

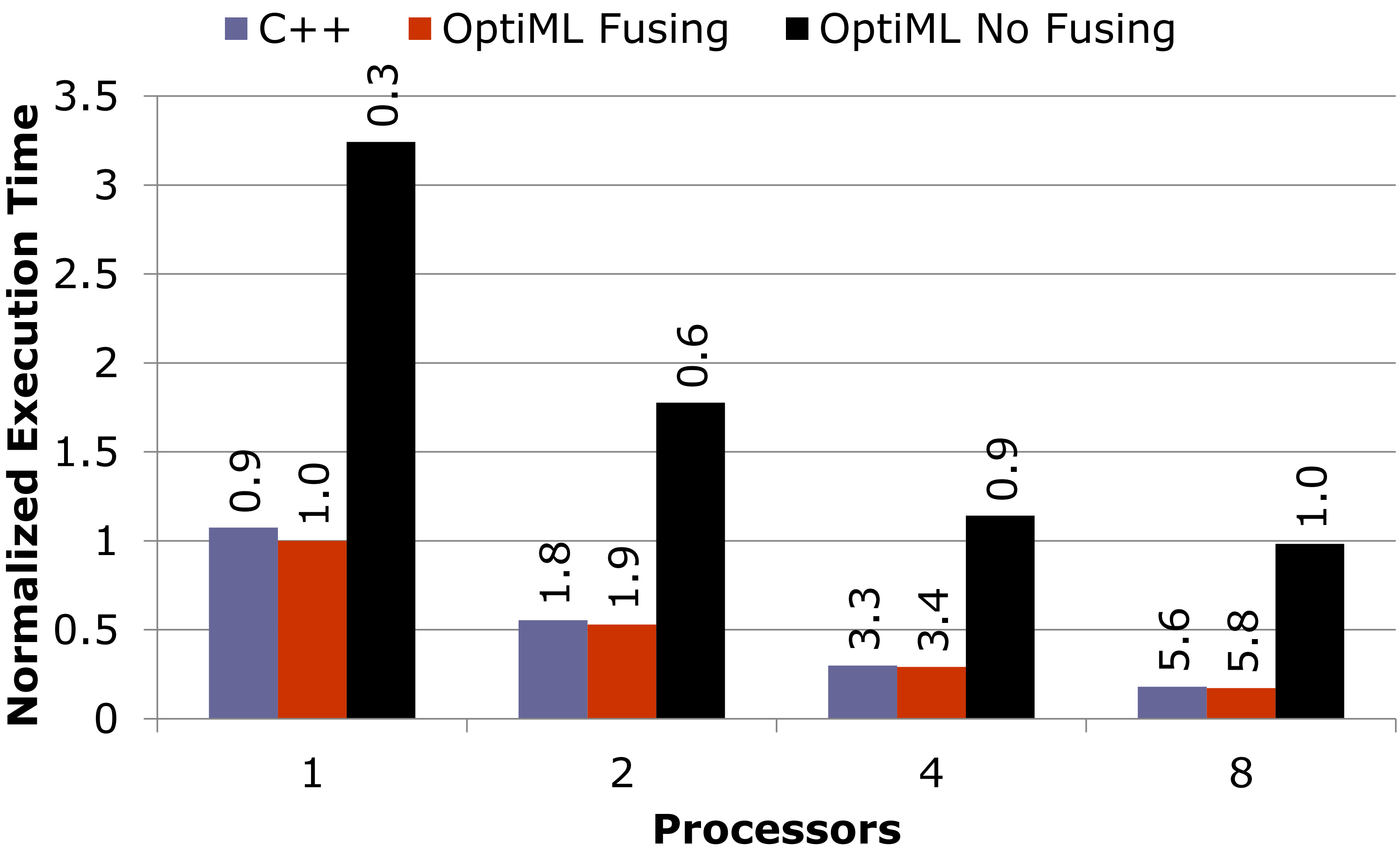
SPADE generated code

```
// 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
}
```

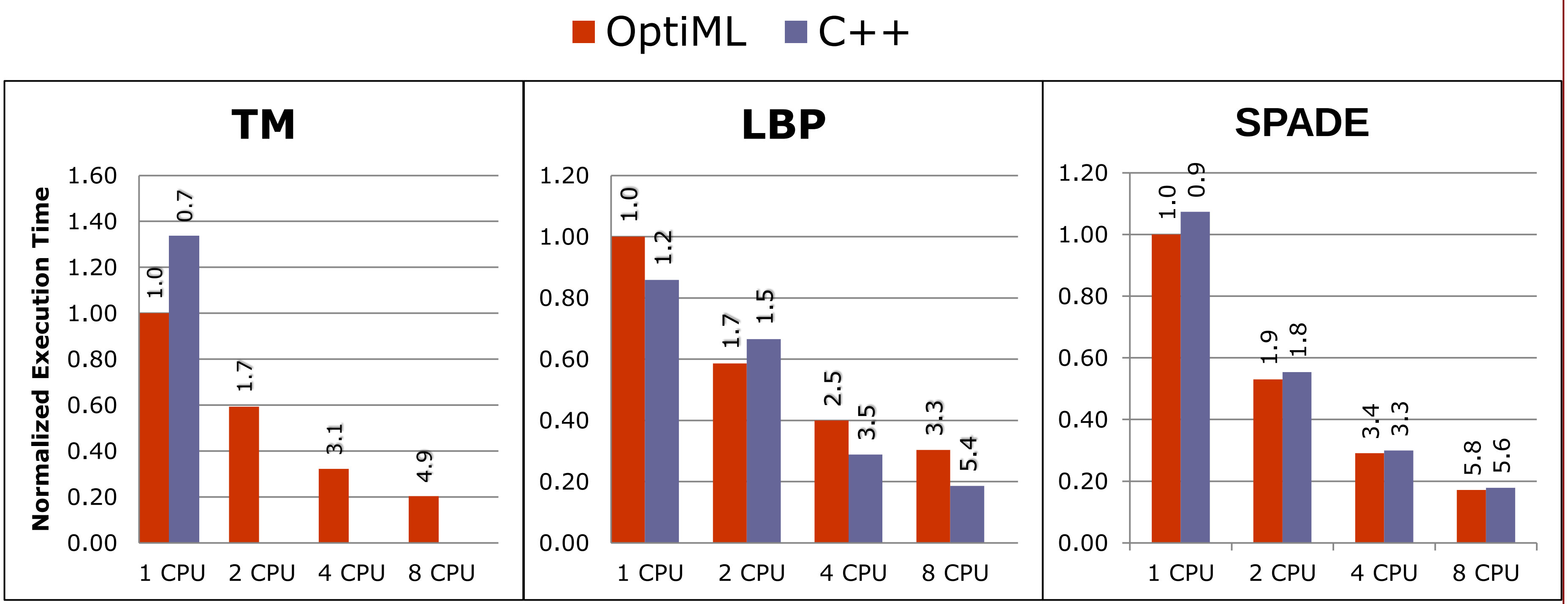
From a ~5 line algorithm description in OptiML

…to an efficient, fused, imperative version that closely resembles a hand-optimized C++ baseline!

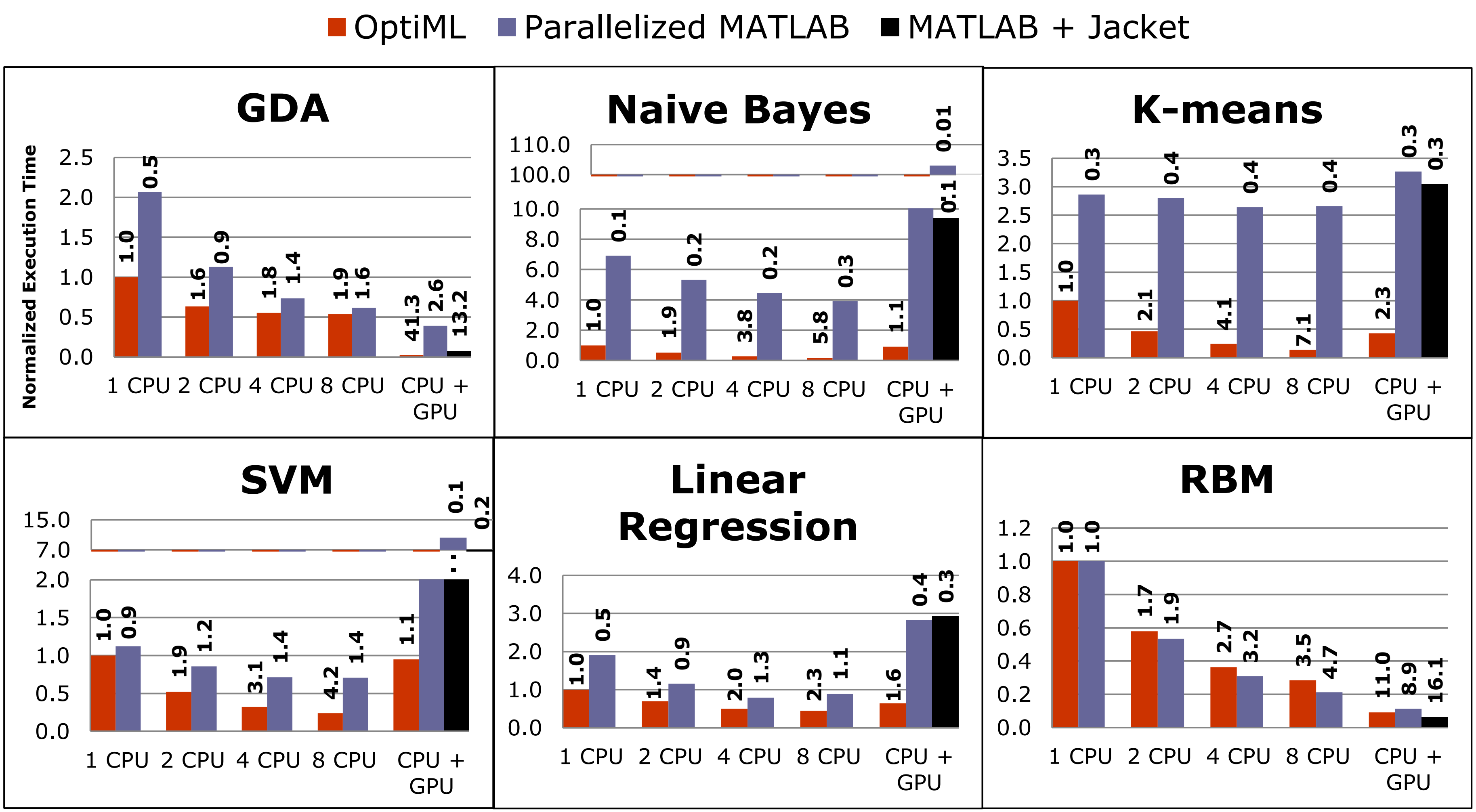
Impact of Op Fusion



Experiments on larger apps



Experiments on ML kernels

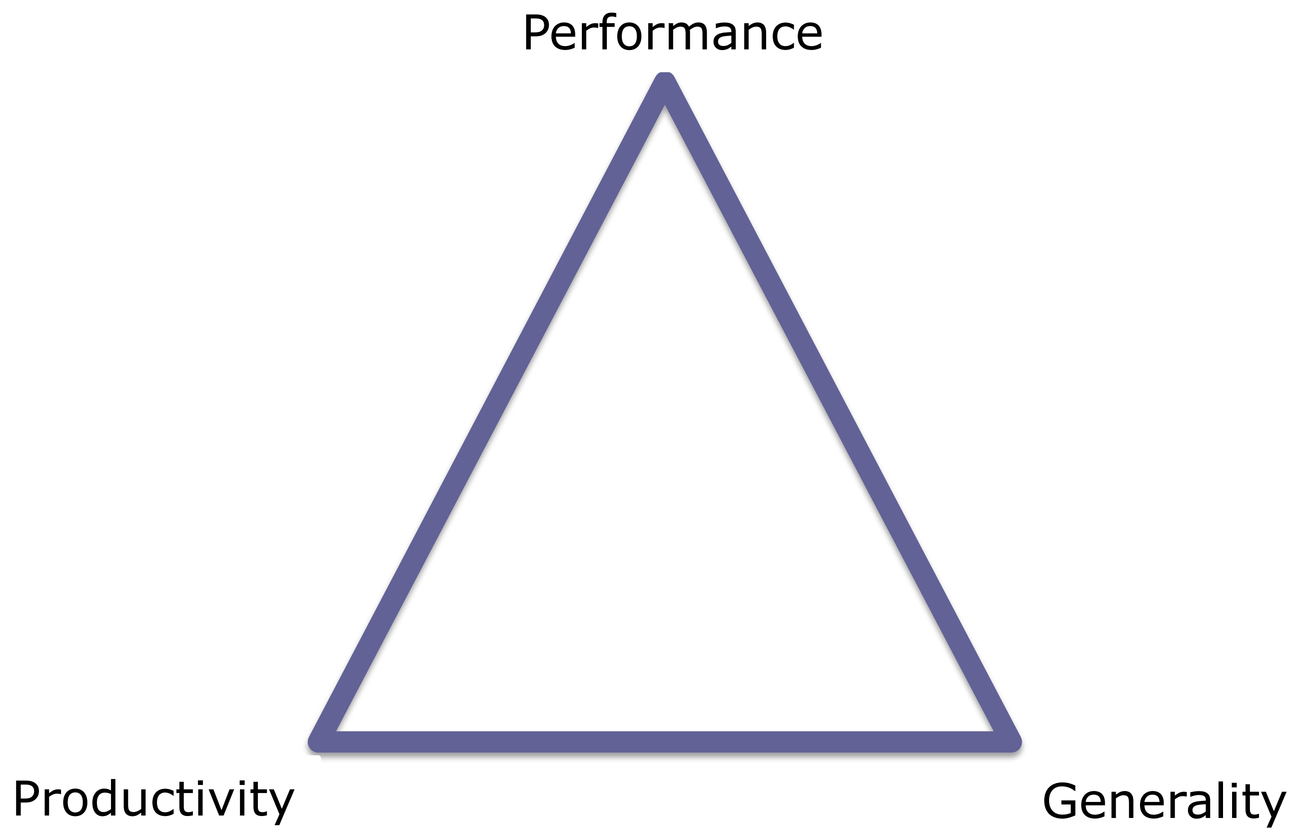


Summary

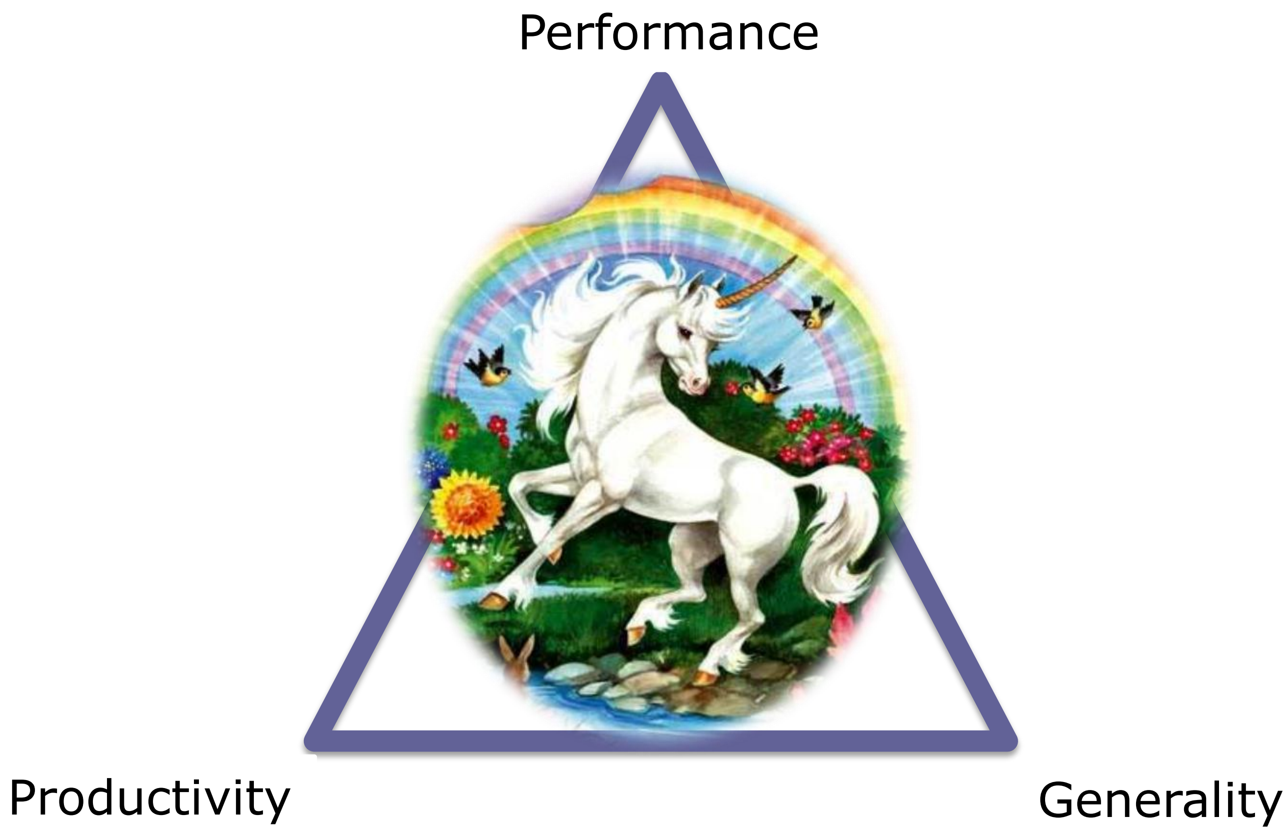
- Performance oriented DSLs are a promising parallel programming platform
  - Capable of achieving portability, productivity, and high performance
- Delite can simplify the task of implementing DSLs
- OptiML outperforms MATLAB and C++ on a set of well known machine learning applications, with expressive code

Questions?

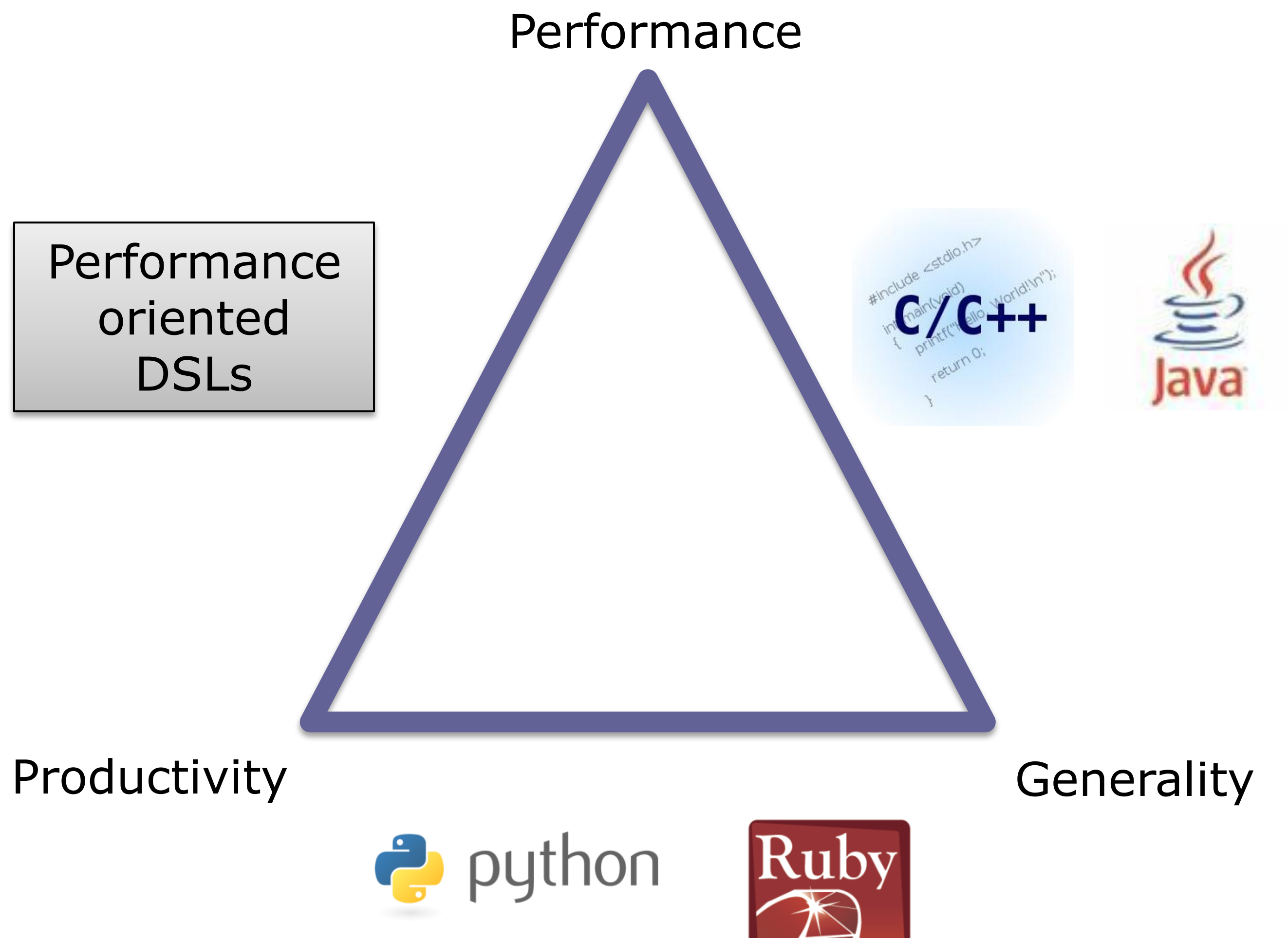
Programming Language Design Space



Programming Language Design Space



General Purpose Languages



DSLs Present New Problem

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