Language Virtualization for Heterogeneous Parallel Computing

Hassan Chafi, Arvind Sujeeth, Zach DeVito, Pat Hanrahan, Kunle Olukotun
Stanford University

Adriaan Moors, Tiark Rompf, Martin Odersky
EPFL

Era of Power Limited Computing

- Mobile
  - Battery operated
  - Passively cooled

- Data center
  - Energy costs
  - Infrastructure costs

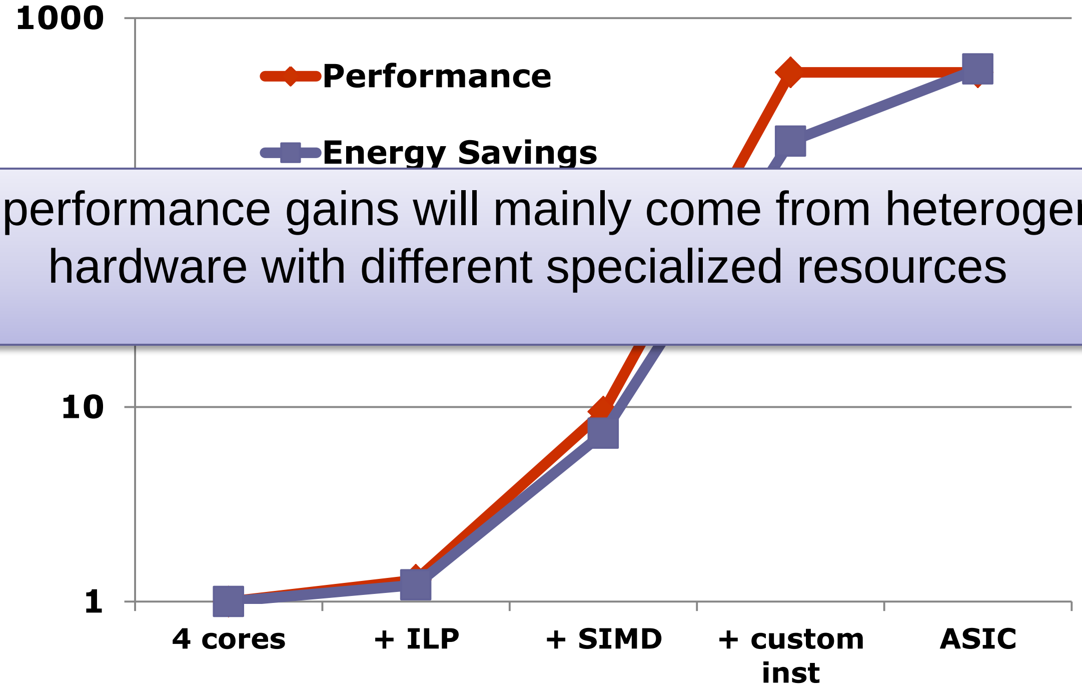
Computing System Power

$$Power = Energy_{Op} \times \frac{Ops}{second}$$

$=$ $\quad\quad$ $\downarrow$ $\quad\quad$ $\uparrow$

Heterogeneous Hardware

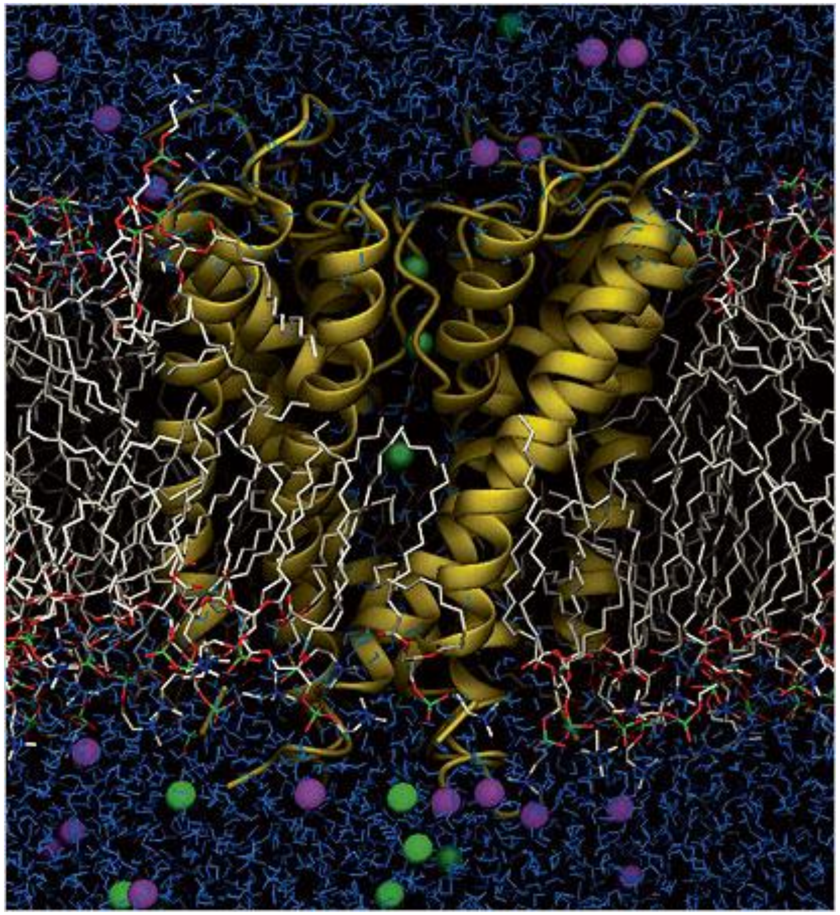
- Heterogeneous HW for energy efficiency
  - Multi-core, ILP, threads, data-parallel engines, custom engines
- H.264 encode study



Future performance gains will mainly come from heterogeneous hardware with different specialized resources

Source: Understanding Sources of Inefficiency in General-Purpose Chips (ISCA'10)

DE Shaw Research: Anton



Molecular dynamics computer

100 times more power efficient

D. E. Shaw et al. SC 2009, Best Paper and Gordon Bell Prize

Apple A4 in the i{Pad|Phone}

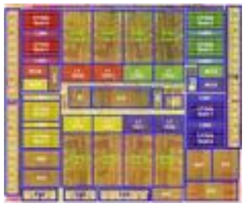
Contains CPU and GPU and …

Heterogeneous Parallel Computing

- Uniprocessor
  - Sequential programming
  - **C**

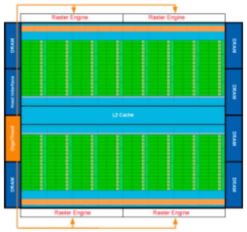
Intel
Pentium 4

- CMP (Multicore)
  - Threads and locks
  - C + **(Pthreads, OpenMP)**



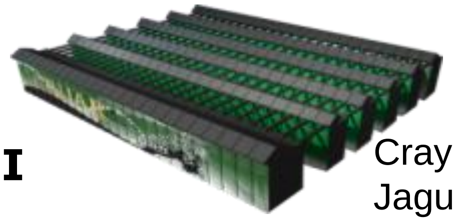
Sun
T2

- GPU
  - Data parallel programming
  - C + (Pthreads, OpenMP) + **(CUDA, OpenCL)**



Nvidia
Fermi

- Cluster
  - Message passing
  - C + (Pthreads, OpenMP) + (CUDA, OpenCL) + **MPI**



Cray
Jaguar

Too many different programming models

It's all About Energy
(Ultimately: Money)

- Human effort just like electrical power
- Aim: reduce development effort, increase performance
- Increase performance now means:
  - reduce energy per op
  - increase # of targets
- Need to reduce effort per target!

IS IT POSSIBLE TO WRITE
ONE PROGRAM

AND

RUN IT ON ALL THESE TARGETS?

HYPOTHESIS: YES, BUT NEED

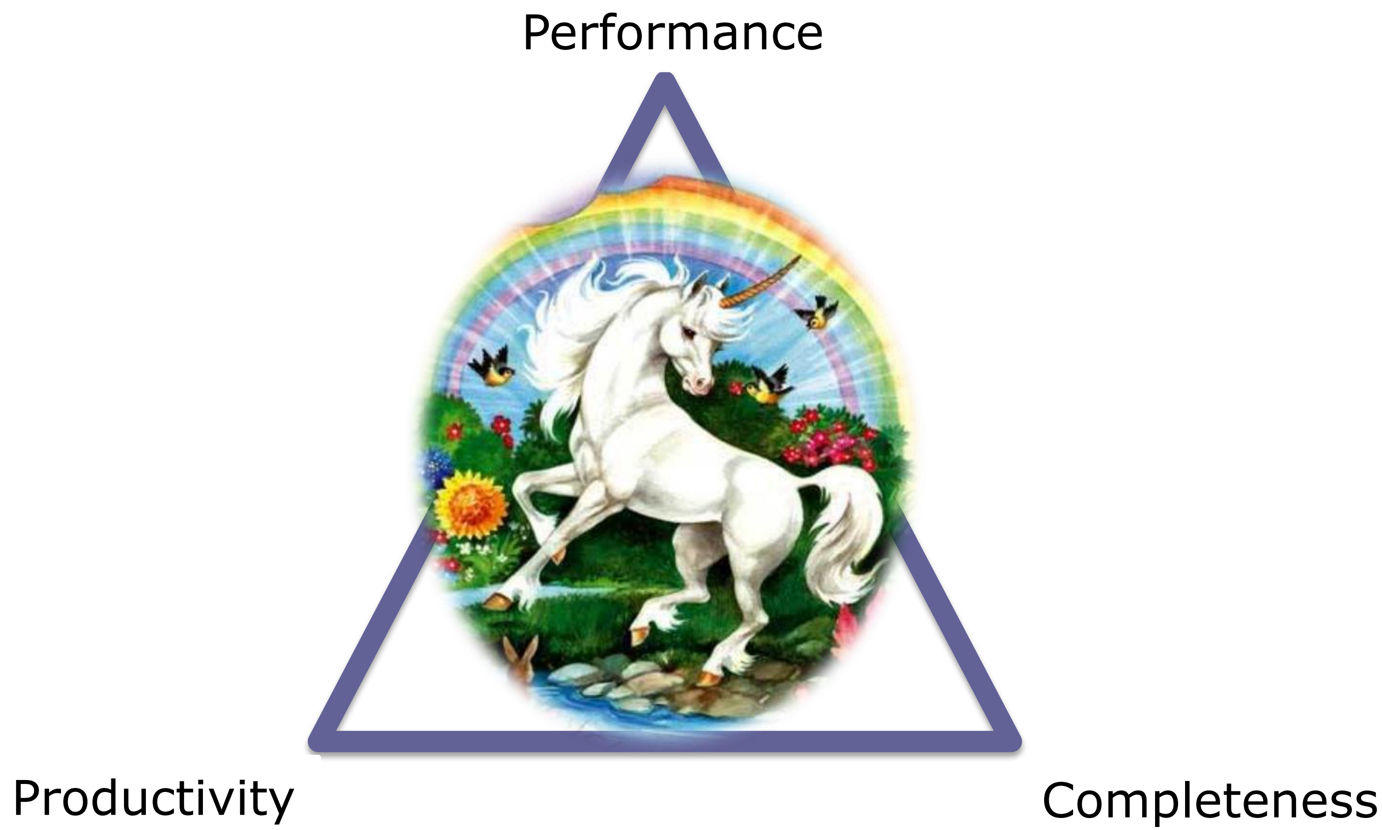
DOMAIN-SPECIFIC

LIBRARIES AND LANGUAGES

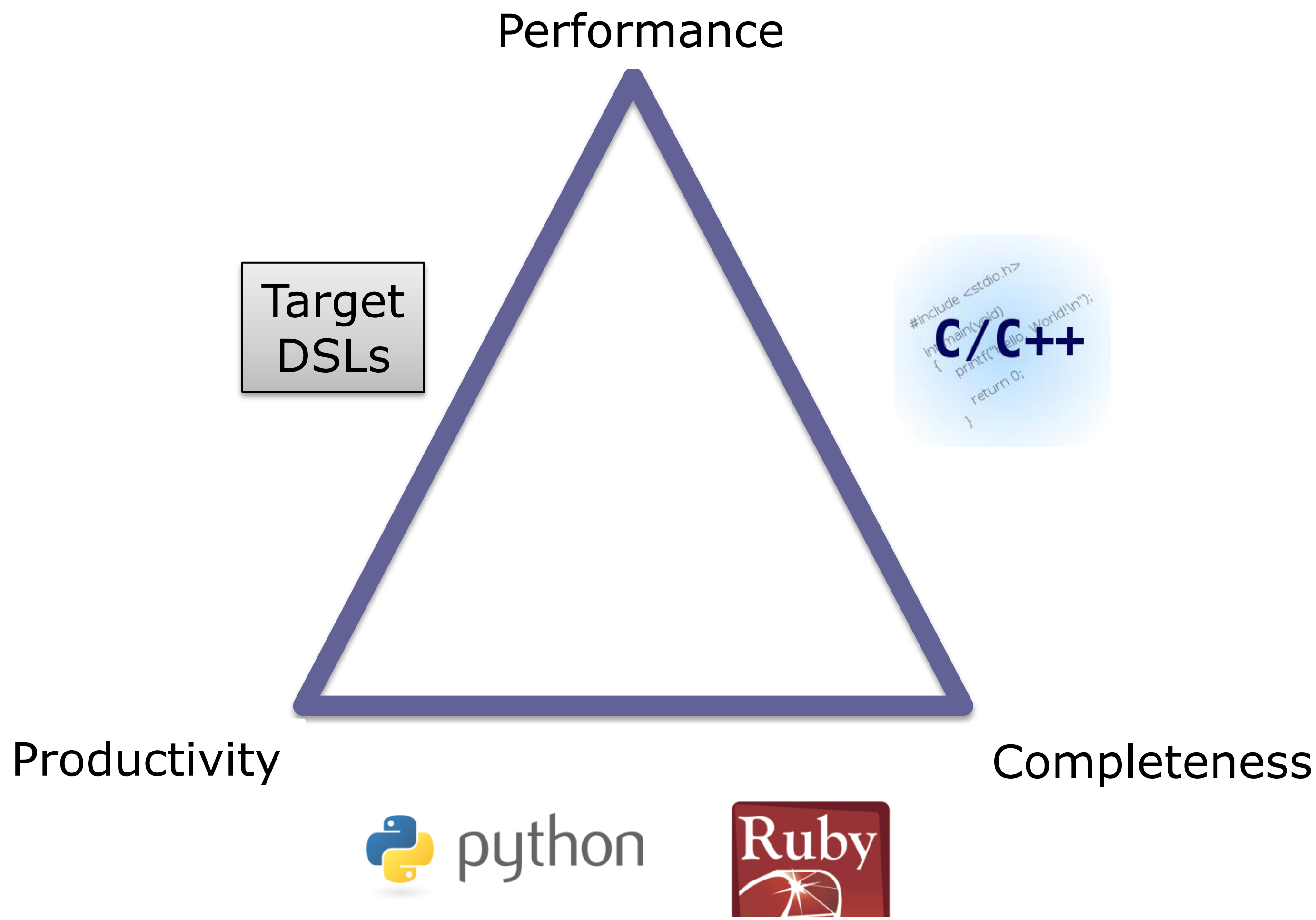
A Solution For Pervasive Parallelism

- Domain Specific Languages (DSLs)
  - Programming language with restricted expressiveness for a particular domain

The Holy Grail of Performance Oriented Languages



The Holy Grail of Performance Oriented Languages



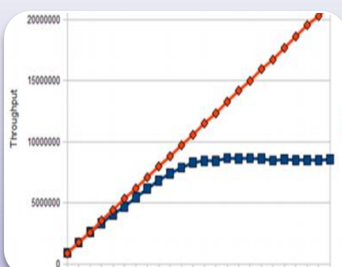
Benefits of Using DSLs for Parallelism

Productivity

- Shield average programmers from the difficulty of parallel programming
- Focus on developing algorithms and applications and not on low level implementation details

Performance

- Match generic parallel execution patterns to high level domain abstraction
- Restrict expressiveness to more easily and fully extract available parallelism
- Use domain knowledge for static/dynamic optimizations



Portability and forward scalability

- DSL & Runtime can be evolved to take advantage of latest hardware features
- Applications remain unchanged
- Allows HW vendors to innovate without worrying about application portability

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

Need to Do Better

- Goal: Develop embedded DSLs that perform as well as stand-alone ones
- Intuition: General-purpose languages should be designed with DSL embedding in mind
- Can we make this intuition more tangible?

Virtualization Analogy

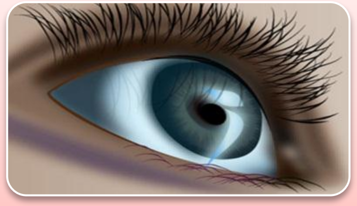
Want to have a range of differently configured machines

- Not practical to run as many physical machines
- Hardware Virtualization: run the logical machines on virtualizable physical hardware

Want to have a range of different languages

- Not practical to implement as many compilers
- Language Virtualization: embed the logical languages into a virtualizable host language

Language Virtualization Requirements

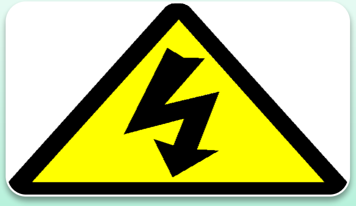


Expressiveness

- Encompasses syntax, semantics and general ease of use for domain experts

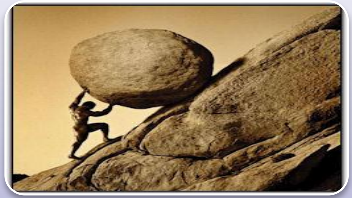
Performance

- Embedded language must me amenable to extensive static and dynamic analysis, optimization and code generation



Safety

- Preserve type safety of embedded language
- No loosened guarantees about program behavior



Modest Effort

- Virtualization is only useful if it reduces effort to embed high performance DSL

Achieving Virtualization: Expressiveness

- OOP allowed higher level of abstractions
  - Add your own types and define operations on them
  - But how about custom type interaction with language features

- Overload all relevant embedding language constructs

```
for (x <- elems if x % 2 == 0) p(x)
```

maps to

```
elems.withFilter(x => x % 2 == 0).foreach(x => p(x))
```

- DSL developer can control how loops over domain collection should be represented and executed by implementing withFilter and foreach for their DSL type

Achieving Virtualization: Expressiveness

- For full virtualization, need to apply similar techniques to all other relevant constructs of the embedding language (for example)

```
if (cond) something else somethingElse
```

maps to

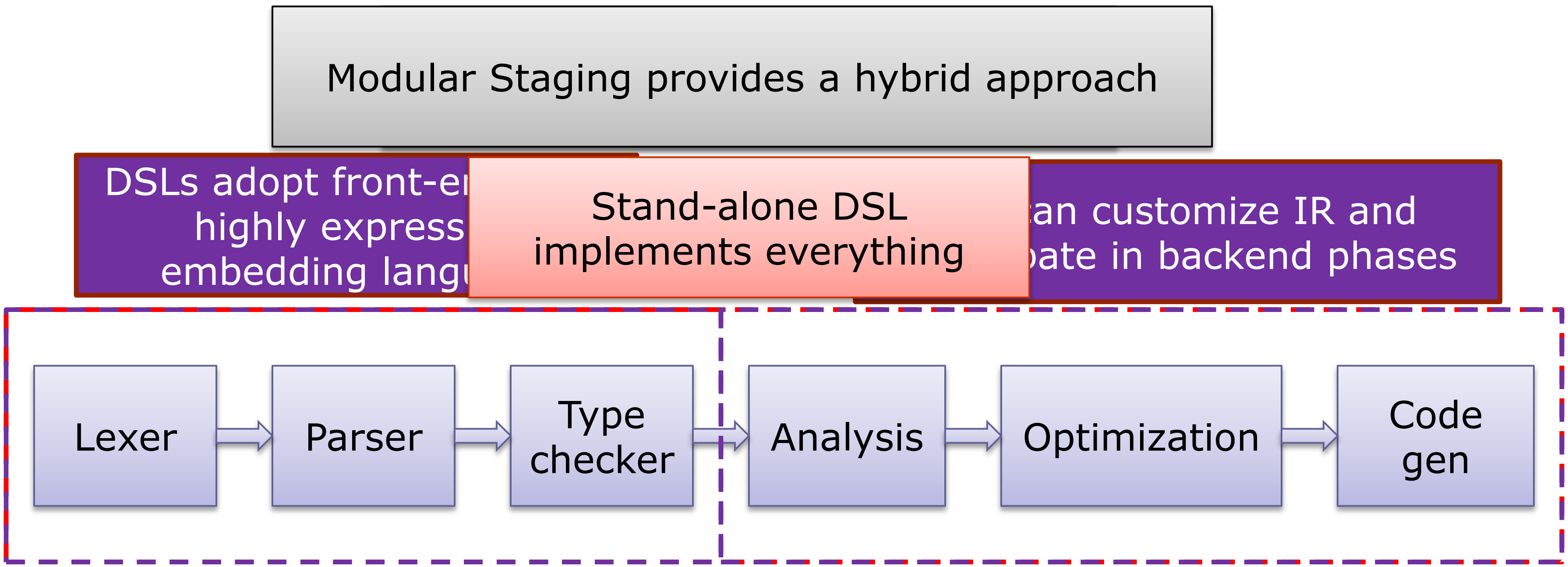
```
__ifThenElse(cond, something, somethingElse)
```

- DSL developer can control the meaning of conditionals by providing overloaded variants specialized to DSL types

Outline

- Introduction
  - Using DSLs for parallel programming
- Language Virtualization
  - Enhancing the power of DSL embedding languages
- Polymorphic Embedding and Modular Staging
  - Enhancing the power of embedded DSLs
- Example DSLs
  - OptiML – targets machine learning applications
  - Liszt – targets scientific computing simulations
- Conclusion

Lightweight Modular Staging Approach



Typical Compiler

GPCE'10: Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs

Linear Algebra Example

```
trait TestMatrix {

  def example(a: Matrix, b: Matrix, c: Matrix, d: Matrix) = {
    val x = a*b + a*c
    val y = a*c + a*d
    println(x+y)
  }

}
```

$$a*b + a*c + a*c + a*d$$
$$=$$
$$a * (b + c + c + d)$$

Abstract Matrix Usage

```
trait TestMatrix {this: MatrixArith =>


  def example(a: Rep[Matrix], b: Rep[Matrix],
         c: Rep[Matrix] , d: Rep[Matrix]) = {
    val x = a*b + a*c
    val y = a*c + a*d
    println(x+y)
  }

}
```

- Rep[Matrix]: abstract type constructor ⇒ range of possible implementations of Matrix
- Operations on Rep[Matrix] defined in MatrixArith trait

Lifting Matrix to Abstract Representation

- DSL interface building blocks structured as traits
  - Expressions of type Rep[T] *represent* expressions of type T
  - Can plug in different representation
- Need to be able to convert (lift) Matrix to abstract representation
- Need to define an interface for our DSL type

```
trait MatrixArith {

    type Rep[T]

    implicit def liftMatrixToRep(x: Matrix): Rep[Matrix]

    def infix_+(x:Rep[Matrix], y: Rep[Matrix]): Rep[Matrix]
    def infix_*(x:Rep[Matrix] , y: Rep[Matrix]): Rep[Matrix]

  }
```

- Now can plugin different implementations and representations for the DSL

Now Can Build an IR

- Start with common IR structure to be shared among DSLs

```
trait Expressions {
     // constants/symbols (atomic)
    abstract class Exp[T]
    case class Const[T](x: T) extends Exp[T]
    case class Sym[T](n: Int) extends Exp[T]

    // operations (composite, defined in subtraits)
    abstract class Op[T]

    // additional members for managing encountered definitions
    def findOrCreateDefinition[T](op: Op[T]): Sym[T]

    implicit def toExp[T](d: Op[T]): Exp[T] = findOrCreateDefinition(d)
  }
```

- Generic optimizations (e.g. common subexpression and dead code elimination) handled once and for all

Customize IR with Domain Info

```
trait MatrixArithRepExp extends MatrixArith with Expressions {
  type Rep[T] = Exp[T]
  implicit def liftMatrixToRep(x: Matrix) = Const(x)
  case class Plus(x: Exp[Matrix],y: Exp[Matrix]) extends Op[Matrix]
  case class Times(x: Exp[Matrix],y: Exp[Matrix]) extends Op[Matrix]
  def infix_+(x: Exp[Matrix],y: Exp[Matrix]) = Plus(x, y)
  def infix_*(x: Exp[Matrix],y: Exp[Matrix]) = Times(x, y)
}
```

- Choose Exp as representation for the DSL types
- Define Lifting function to create expressions
- Extend generic IR with domain-specific node types
- DSL methods build IR as program runs

DSL Optimization

- Use domain-specific knowledge to make optimizations in a modular fashion

```
trait MatrixArithRepExpOpt extends MatrixArithRepExp {
  override def infix_+(x: Exp[Matrix], y: Exp[Matrix]) = (x, y) match {
    case (Times(a, b), Times(c, d)) if (a == c) => infix_*(a, infix_+(b,d))
    case _ => super.plus(x, y)
}}
```

- Override IR node creation
- Construct Optimized IR nodes if possible
- Construct default otherwise

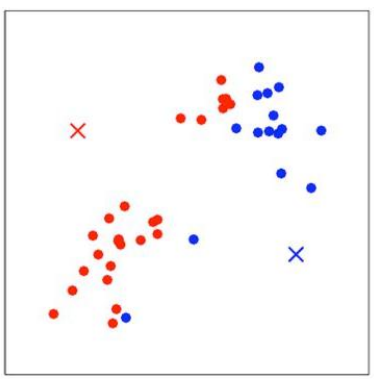
- Rewrite rules are simple, yet powerful optimization mechanism
- Access to the full domain specific IR allows for application of much more complex optimizations

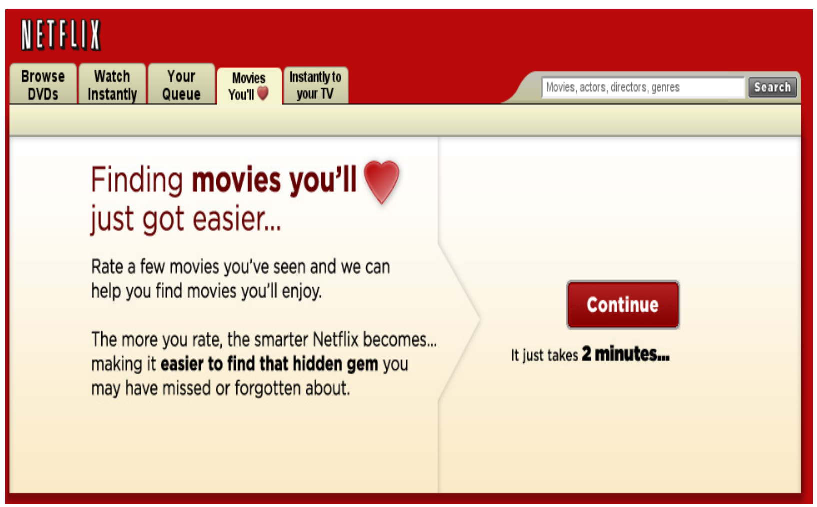
Outline

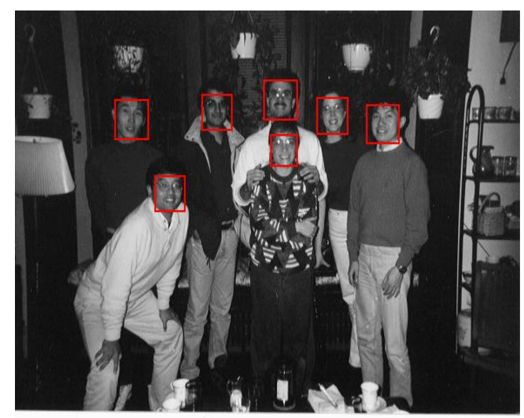
- Introduction
  - Using DSLs for parallel programming
- Language Virtualization
  - Enhancing the power of DSL embedding languages
- Polymorphic Embedding and Modular Staging
  - Enhancing the power of embedded DSLs
- Example DSLs
  - OptiML – targets machine learning applications
  - Liszt – targets scientific computing simulations
- Conclusion

OptiML: A DSL for Machine Learning

- Learning patterns from data
  - Regression
  - Classification (e.g. SVMs)
  - Clustering (e.g. K-Means)
  - Density estimation (e.g. Expectation Maxi
  - Inference (e.g. Loopy Belief Propagation)
  - Adaptive (e.g. Reinforcement Learning)







Why Machine Learning

- A good domain for studying parallelism
  - Many applications and datasets are time-bound in practice
  - A combination of regular and irregular parallelism at varying granularities
  - At the core of many emerging applications (speech recognition, robotic control, data mining etc.)

OptiML Language Features

- Implicitly parallel data structures
  - General linear algebra data types : Vector[T], Matrix[T]
    - Independent from the underlying implementation
  - Special data types : TrainingSet, TestSet, IndexVector, Image, Video ..
    - Encode semantic information
- Implicitly parallel control structures
  - Sum{…}, (0::end) {…}, gradient { … }, untilconverged { … }
  - Encode restricted semantics within passed in code block
- Domain specific optimizations
  - Trade off a small amount accuracy for a large amount of performance
    - Relaxed dependencies
    - Best effort computing

OptiML Code Example

- Gaussian Discriminant Analysis

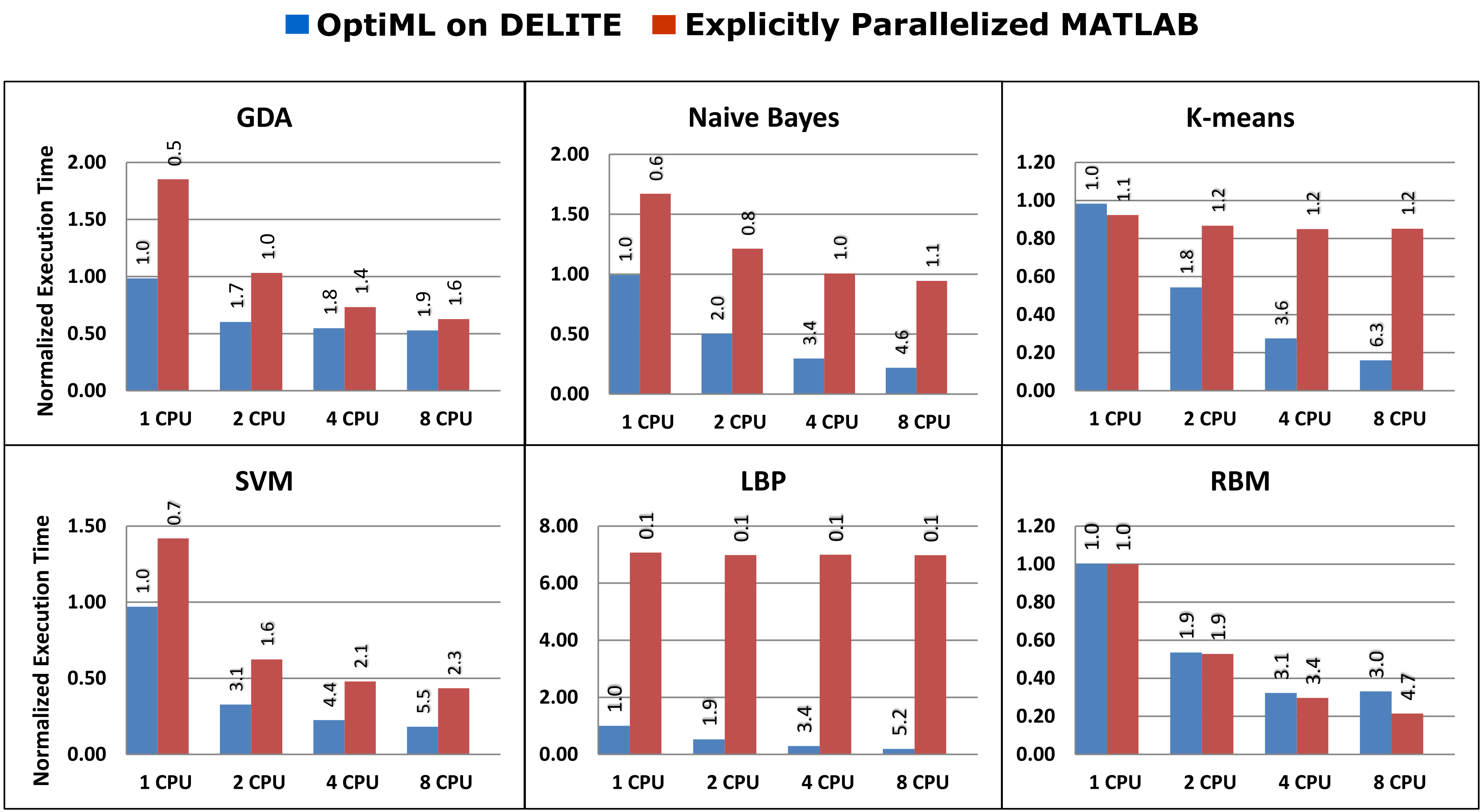
```
// x : TrainingSet[Double]
// mu0, mu1 : Vector[Double]

val sigma = sum(0,x.numSamples) {
  if (x.labels(_) == false)
      (x(_)-mu0).trans.outer(x(_)-mu0)
  else
     (x(_)-mu1).trans.outer(x(_)-mu1)
}
```

ML-specific data types

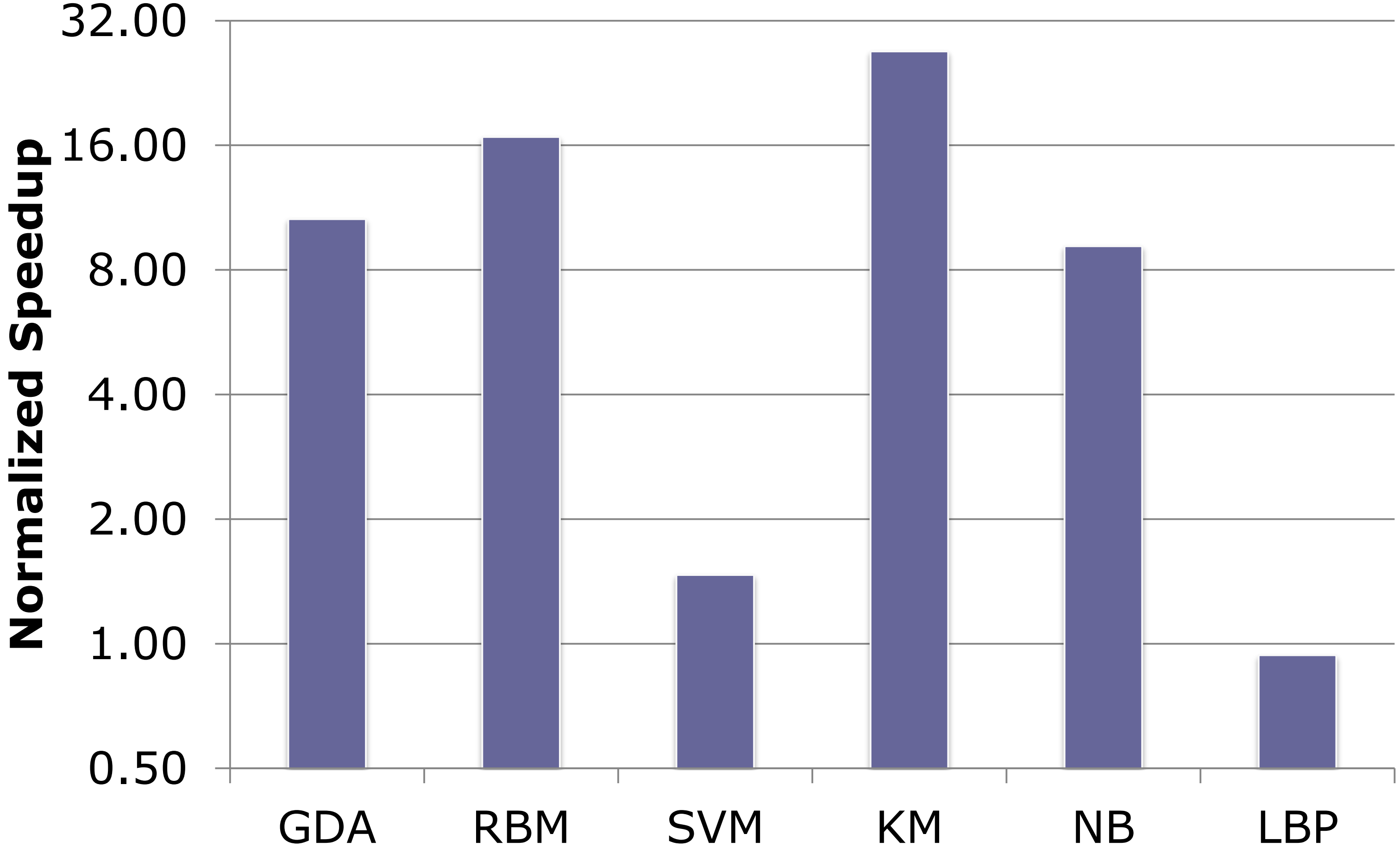
Parallel Control structures

Restricted index semantics

Performance Study (CPU)

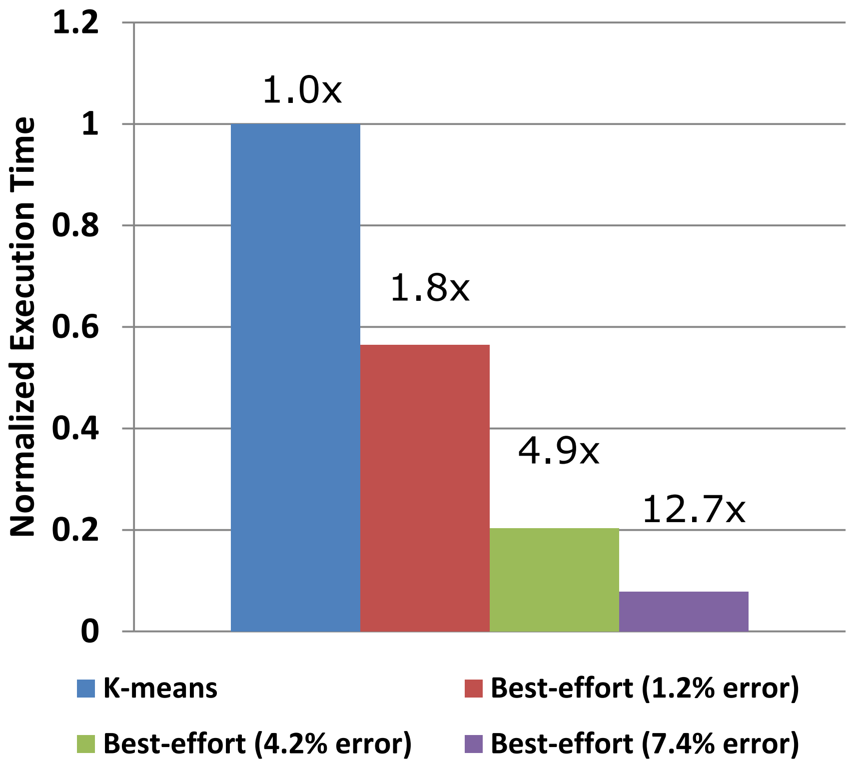


Performance Study (GPU)

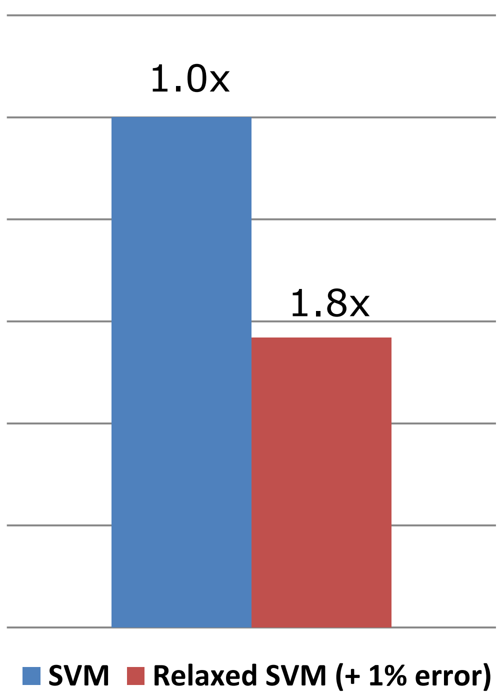


Domain Specific Optimizations

- Best Effort Computation



- Relaxed Dependencies

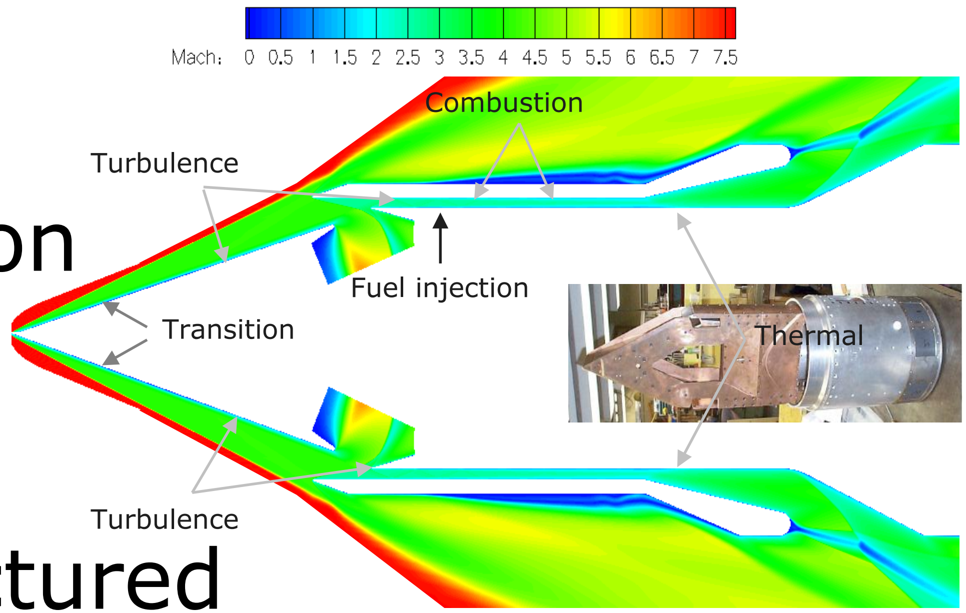


Outline

- Introduction
  - Using DSLs for parallel programming
- Language Virtualization
  - Enhancing the power of DSL embedding languages
- Polymorphic Embedding and Modular Staging
  - Enhancing the power of embedded DSLs
- Example DSLs
  - OptiML – targets machine learning applications
  - Liszt – targets scientific computing simulations
- Conclusion

Liszt: A DSL for PDEs

- Mesh-based
- Numeric Simulation
- Huge domains
  - millions of cells
- Example: Unstructured Reynolds-averaged Navier Stokes (RANS) solver



Liszt Language Features

- Built-in mesh interface for arbitrary polyhedra
  - Vertex, Edge, Face, Cell
- Collections of mesh elements
  - Element Sets: faces(c:Cell), edgesCCW(f:Face)
- Mesh-based data storage
  - Fields: val vert_position = position(v)
- Parallelizable iteration
  - forall statements: for(f <- faces(cell)) { … }

Liszt Code Example

```
for(edge <- edges(mesh)) {
    val flux = flux_calc(edge)
    val v0 = head(edge)
    val v1 = tail(edge)
    Flux(v0) += flux
    Flux(v1) -= flux
}
```

Simple Set Comprehension

Functions, Function Calls

Mesh Topology Operators

Field Data Storage

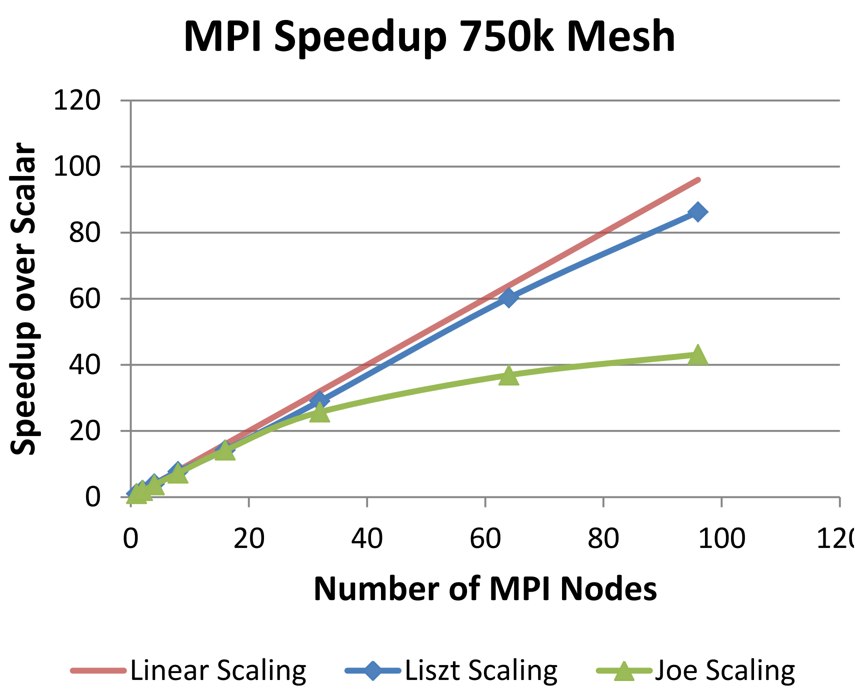
Code contains possible write conflicts!

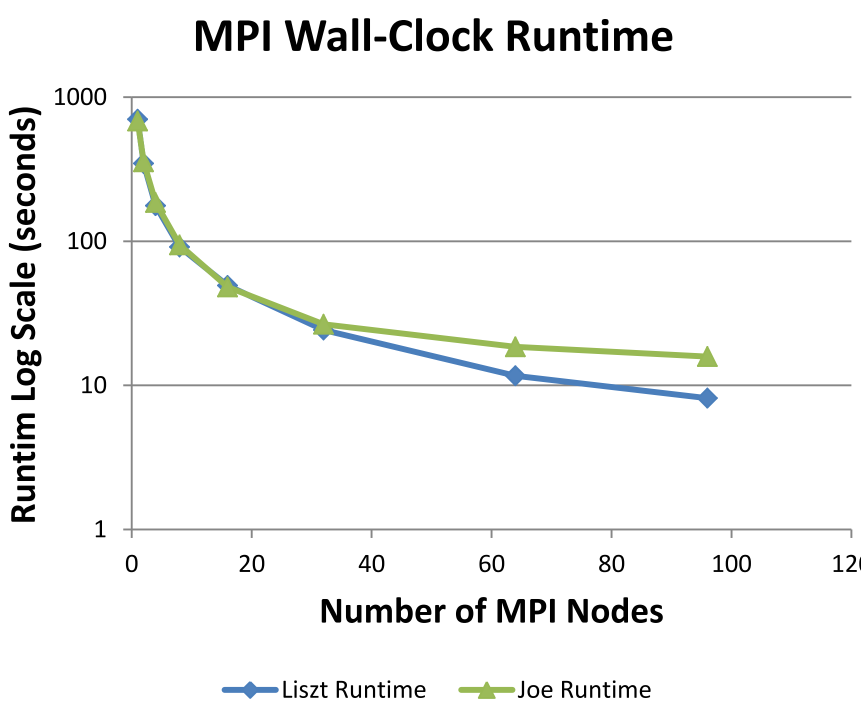
We use architecture specific strategies guided by domain knowledge

- MPI: Ghost cell-based message passing
- GPU: Coloring-based use of shared memory

MPI Performance

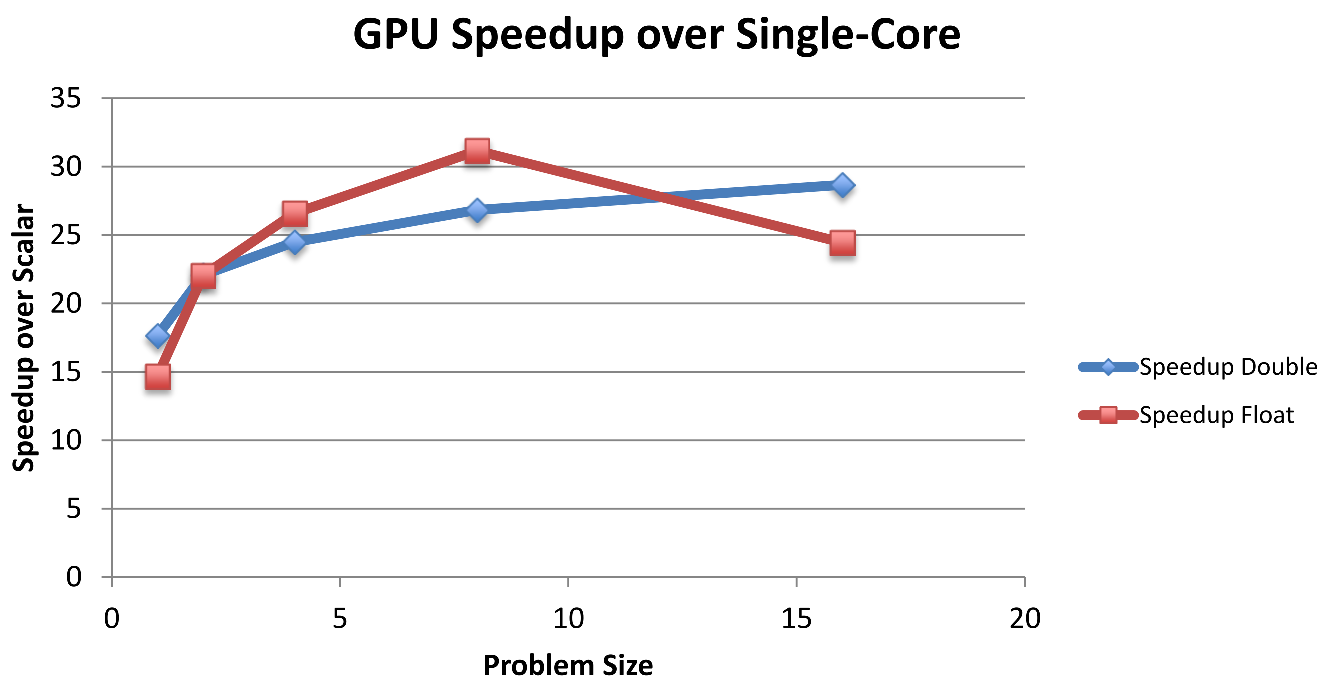
- Using 8 cores per node, scaling up to 96 cores (12 nodes, 8 cores per node, all communication using MPI)





GPU Performance

- Scaling mesh size from 50k (unit-sized) cells to 750k (16x) on a Tesla C2050. Comparison is against single threaded runtime on host CPU (Core 2 Quad 2.66Ghz)



Single-Precision: 31.5x, Double-precision: 28x

Conclusions

- DSLs can be an answer to the heterogeneous parallel programming problem
- Need embedding languages to be more virtualizable
- First steps in virtualizing Scala
- Lightweight modular staging allows for more powerful embedded DSLs
- Early embedded DSL results are promising
- No unicorns were harmed during production