Taming Heterogeneous Parallelism with Domain Specific Languages

Kunle Olukotun
Pervasive Parallelism Laboratory
Stanford University

PPL Research Goal

- Unleash full power of future computing platforms
  - Make parallel application development practical for the masses (Joe the programmer)
  - Parallel applications without parallel programming

Guiding Observations

- Heterogeneous parallel hardware
  - Computing is energy constrained
  - Specialization: energy and area efficient parallel computing
- Must hide low-level issues from most programmers
  - Explicit parallelism won't work (10K- 100K threads)
  - Only way to get simple and portable programs
- No single discipline can solve all problems
  - Apps, PLs, runtime, OS, architecture
  - Need vertical integration
  - Hanrahan, Aiken, Rosenblum, Kozyrakis, Horowitz

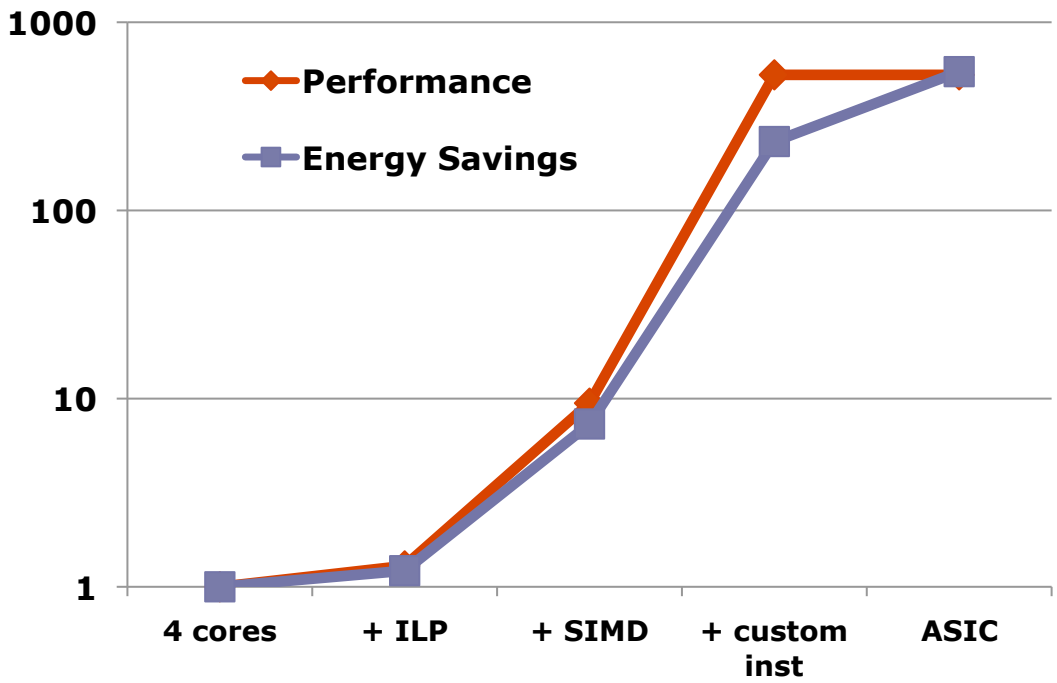
Computing System Power

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

= ↓ ↑

Heterogeneous Hardware

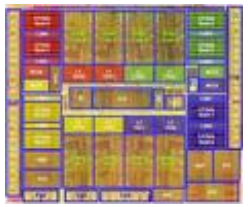
- Heterogeneous HW for energy efficiency
  - Multi-core, ILP, threads, data-parallel engines, custom engines
- H.264 encode study



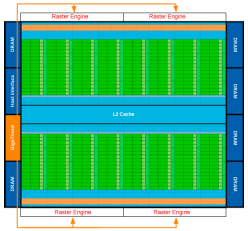
Source: Understanding Sources of Inefficiency in General-Purpose Chips (ISCA'10)

Heterogeneous Parallel Architectures

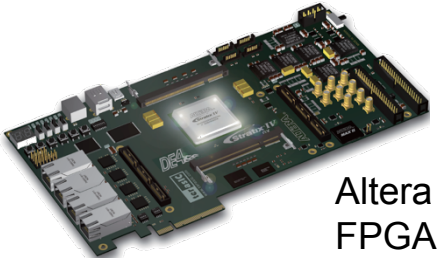
Driven by energy efficiency



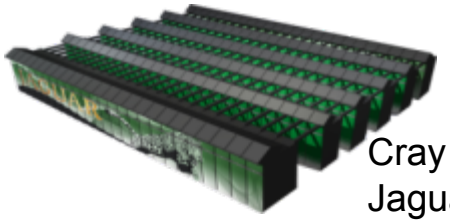
Sun T2



Nvidia Fermi



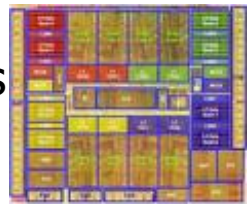
Altera FPGA



Cray Jaguar

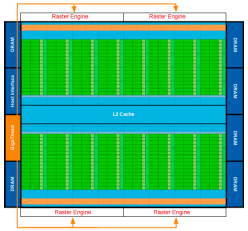
Heterogeneous Parallel Programming

Pthreads
OpenMP



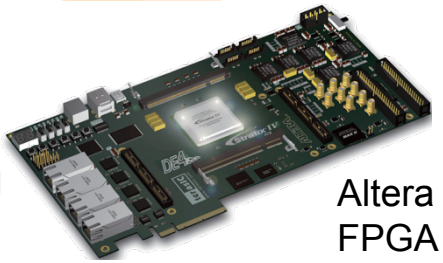
Sun
T2

CUDA
OpenCL



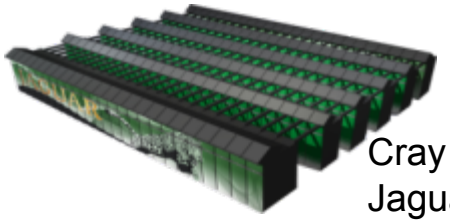
Nvidia
Fermi

Verilog
VHDL



Altera
FPGA

MPI



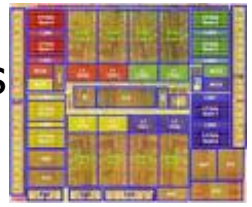
Cray
Jaguar

Programmability Chasm

Applications

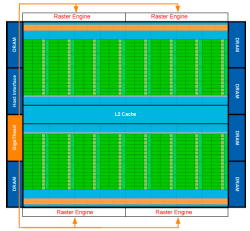
- Scientific Engineering
- Virtual Worlds
- Personal Robotics
- Data informatics

Pthreads
OpenMP



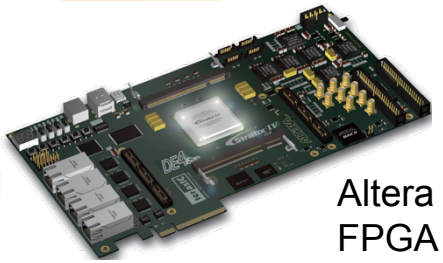
Sun T2

CUDA
OpenCL



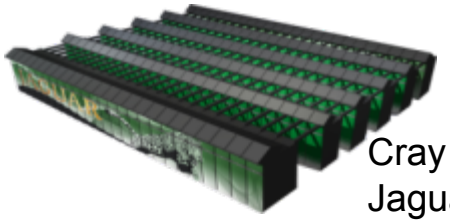
Nvidia Fermi

Verilog
VHDL



Altera FPGA

MPI



Cray Jaguar

Too many different programming models

Hypothesis

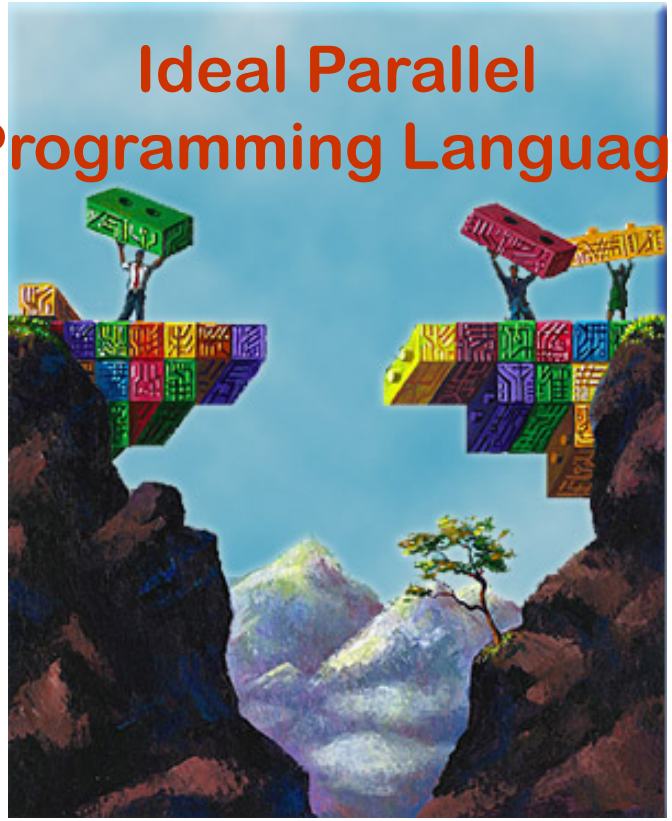
PERVASIVE PARALLELISM LABORATORY PPL

It is possible to write one program
and
run it on all these machines

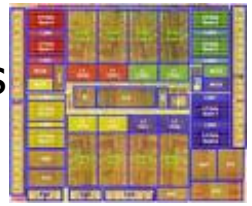
Programmability Chasm

Applications

- Scientific Engineering
- Virtual Worlds
- Personal Robotics
- Data informatics

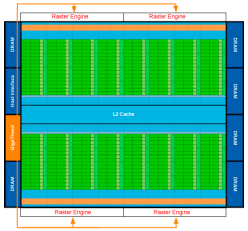


Pthreads
OpenMP



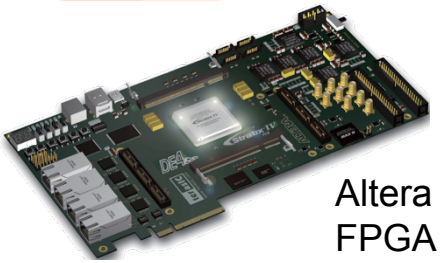
Sun
T2

CUDA
OpenCL



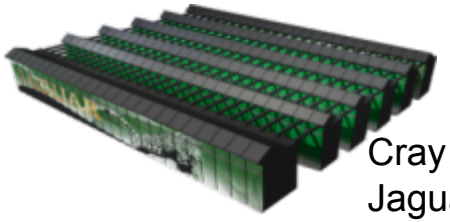
Nvidia
Fermi

Verilog
VHDL



Altera
FPGA

MPI

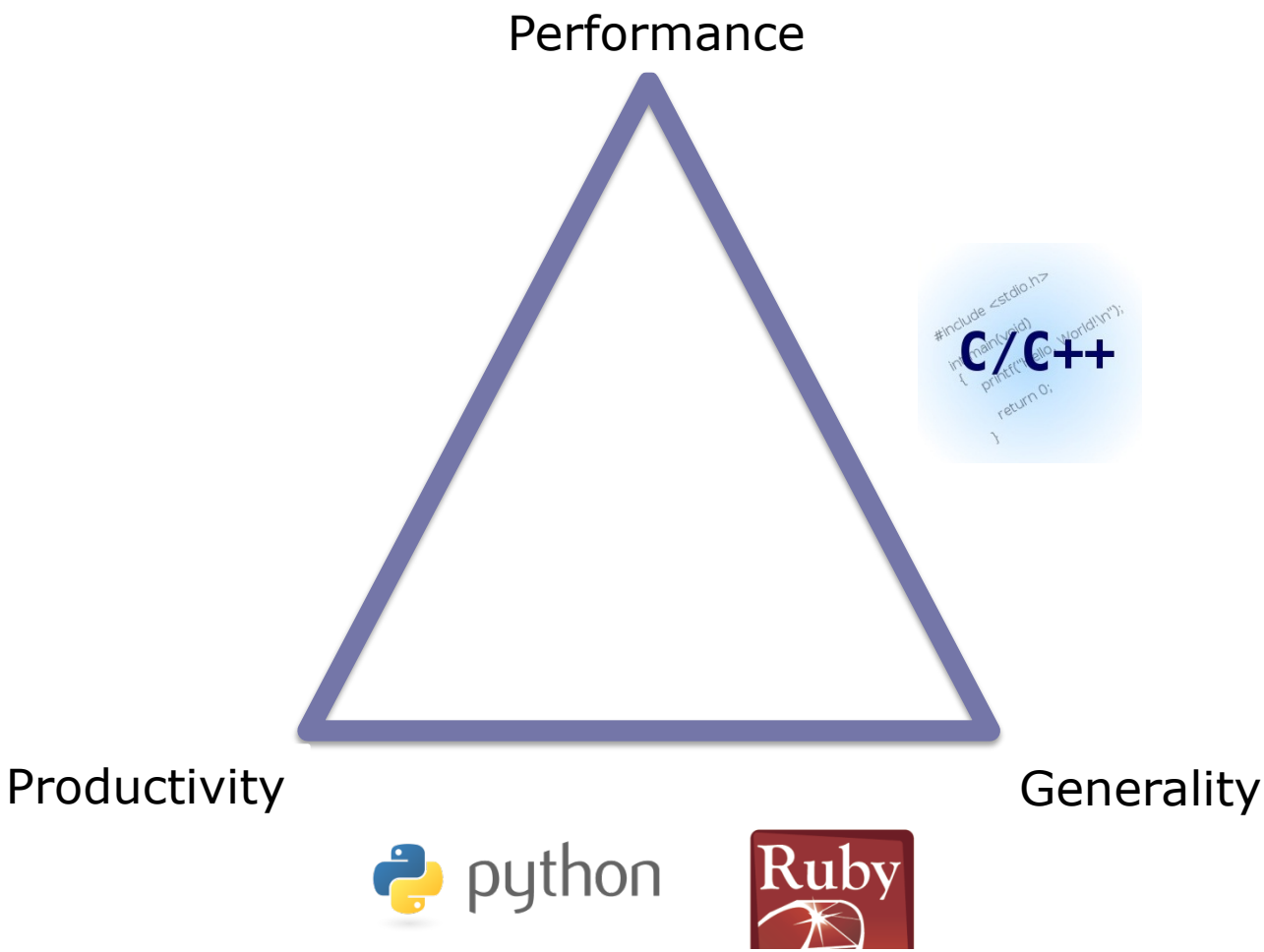


Cray
Jaguar

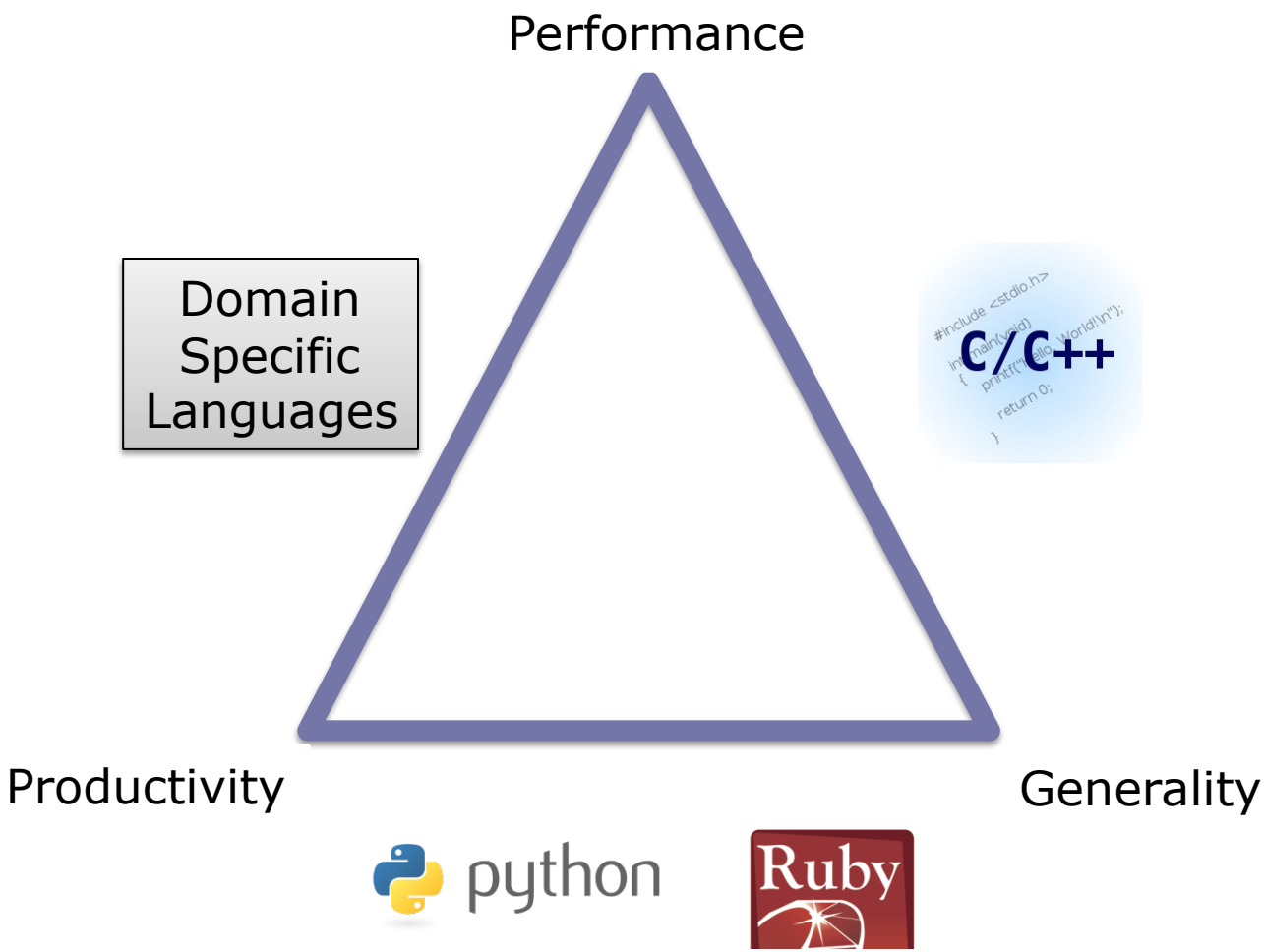
Too many different programming models

The Ideal Parallel Programming Language

Successful Languages



True Hypothesis ⇒ Domain Specific Languages



Domain Specific Languages

- Domain Specific Languages (DSLs)
  - Programming language with restricted expressiveness for a particular domain
  - High-level and usually declarative

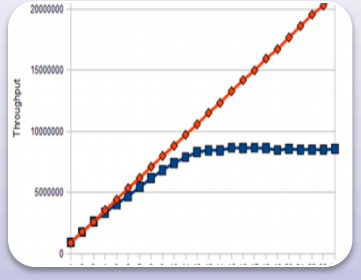
Benefits of Using DSLs for Parallelism

Productivity

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

Performance

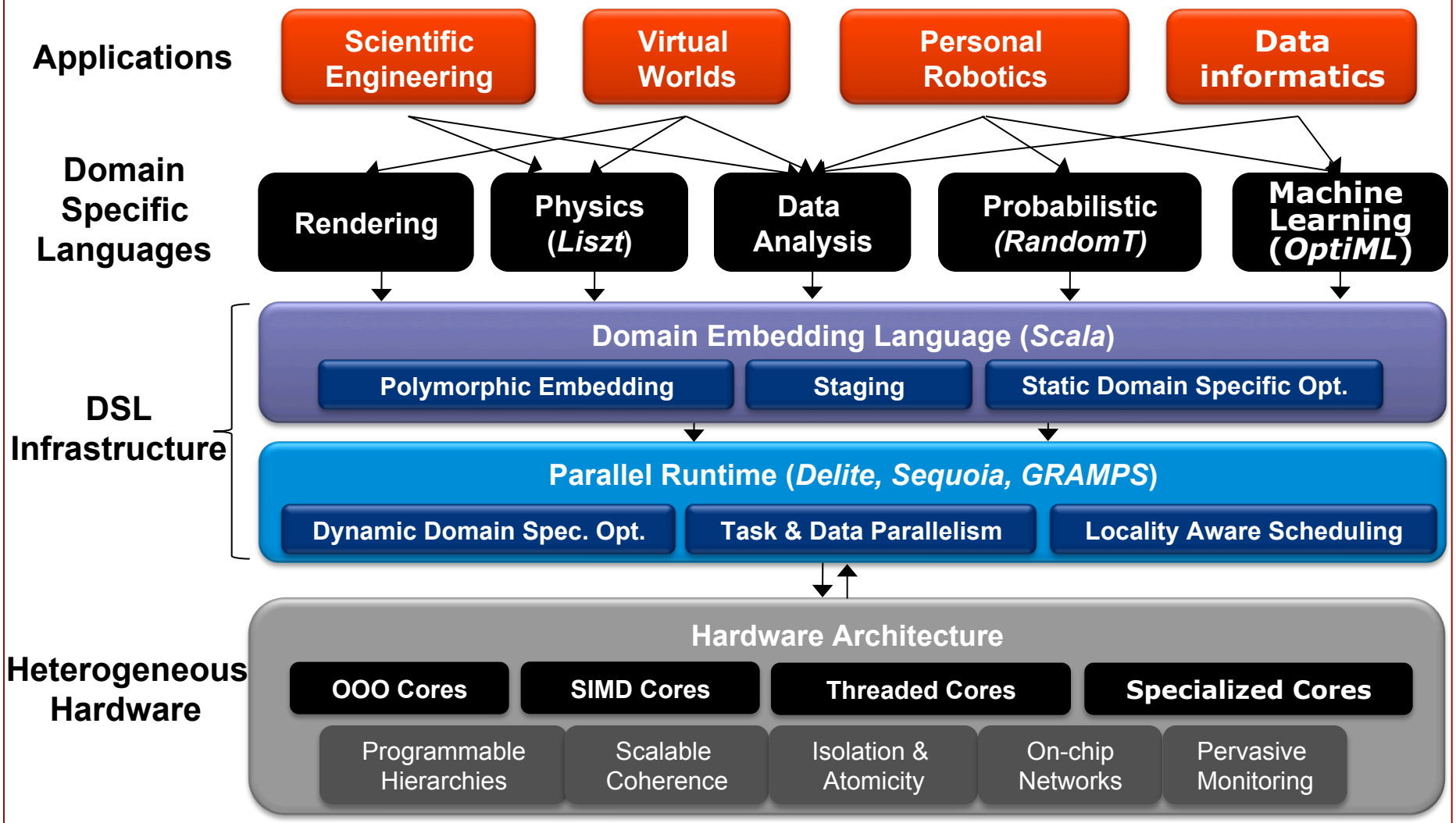
- Match high level domain abstraction to generic parallel execution patterns
- Restrict expressiveness to more easily and fully extract available parallelism
- Use domain knowledge for static/dynamic optimizations



Portability and forward scalability

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

PPL Vision


Applications
Scientific Engineering
Virtual Worlds
Personal Robotics
Data informatics
Domain Specific Languages
Rendering
Physics (Liszt)
Data Analysis
Probabilistic (RandomT)
Machine Learning (OptiML)
DSL Infrastructure
Domain Embedding Language (Scala)
Polymorphic Embedding
Staging
Static Domain Specific Opt.
Parallel Runtime (Delite, Sequoia, GRAMPS)
Dynamic Domain Spec. Opt.
Task & Data Parallelism
Locality Aware Scheduling
Heterogeneous Hardware
Hardware Architecture
OOO Cores
SIMD Cores
Threaded Cores
Specialized Cores
Programmable Hierarchies
Scalable Coherence
Isolation & Atomicity
On-chip Networks
Pervasive Monitoring

New Problem

- We need to develop all of 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

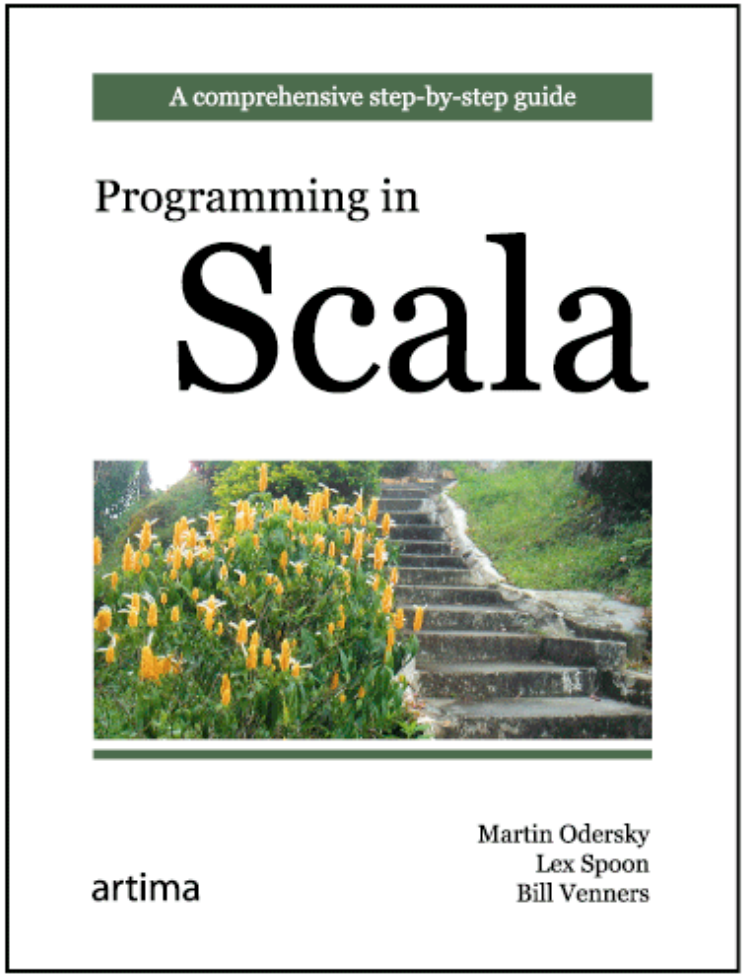
- Purely embedded DSLs ⇒ "just a library"
  - Easy to develop (can reuse full host language)
  - Easier to learn DSL
  - Can Combine multiple DSLs in one program
  - Can Share DSL infrastructure among several DSLs
  - Hard to optimize using domain knowledge
  - Target same architecture as host language

Need to do better

Need to Do Better

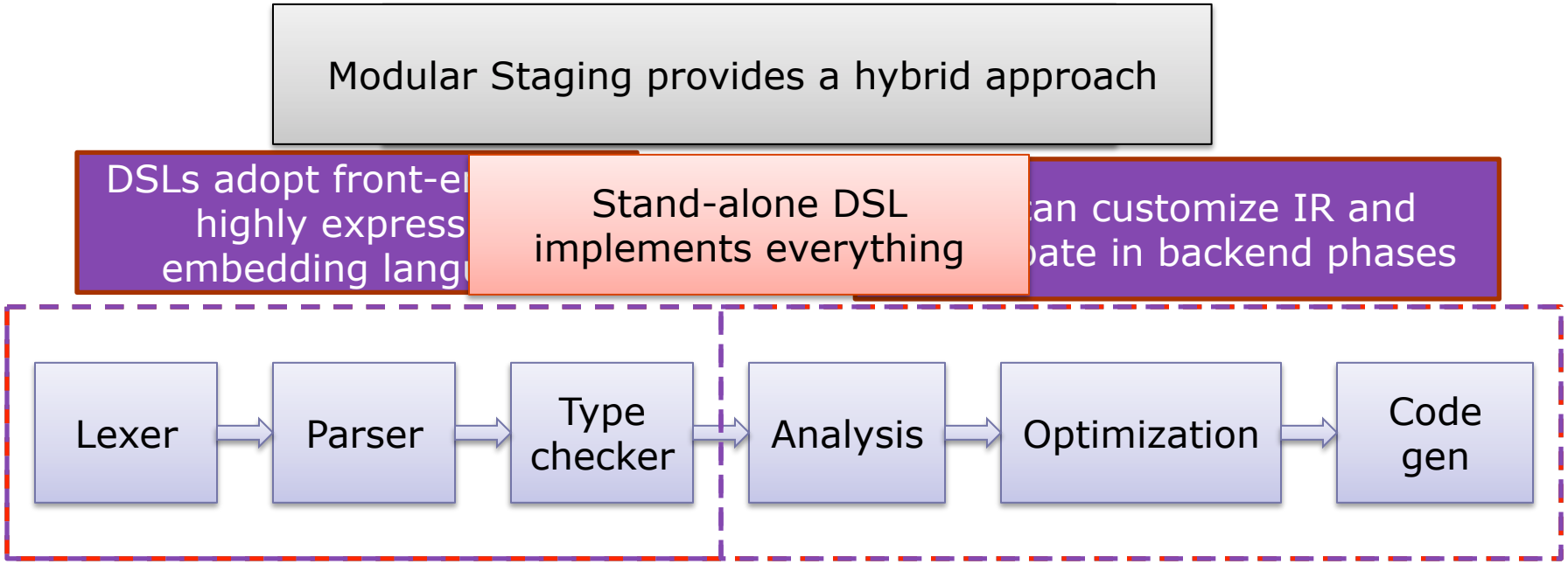
- Goal: Develop embedded DSLs that perform as well as stand-alone ones
- Intuition: General-purpose languages should be designed with DSL embedding in mind

DSL Embedding Language



- Mixes OO and FP paradigms
  - Targets JVM
- Expressive type system allows powerful abstraction
- Scalable language
- Stanford/EPFL collaboration on leveraging Scala for parallelism
- "Language Virtualization for Heterogeneous Parallel Computing" Onward 2010, Reno

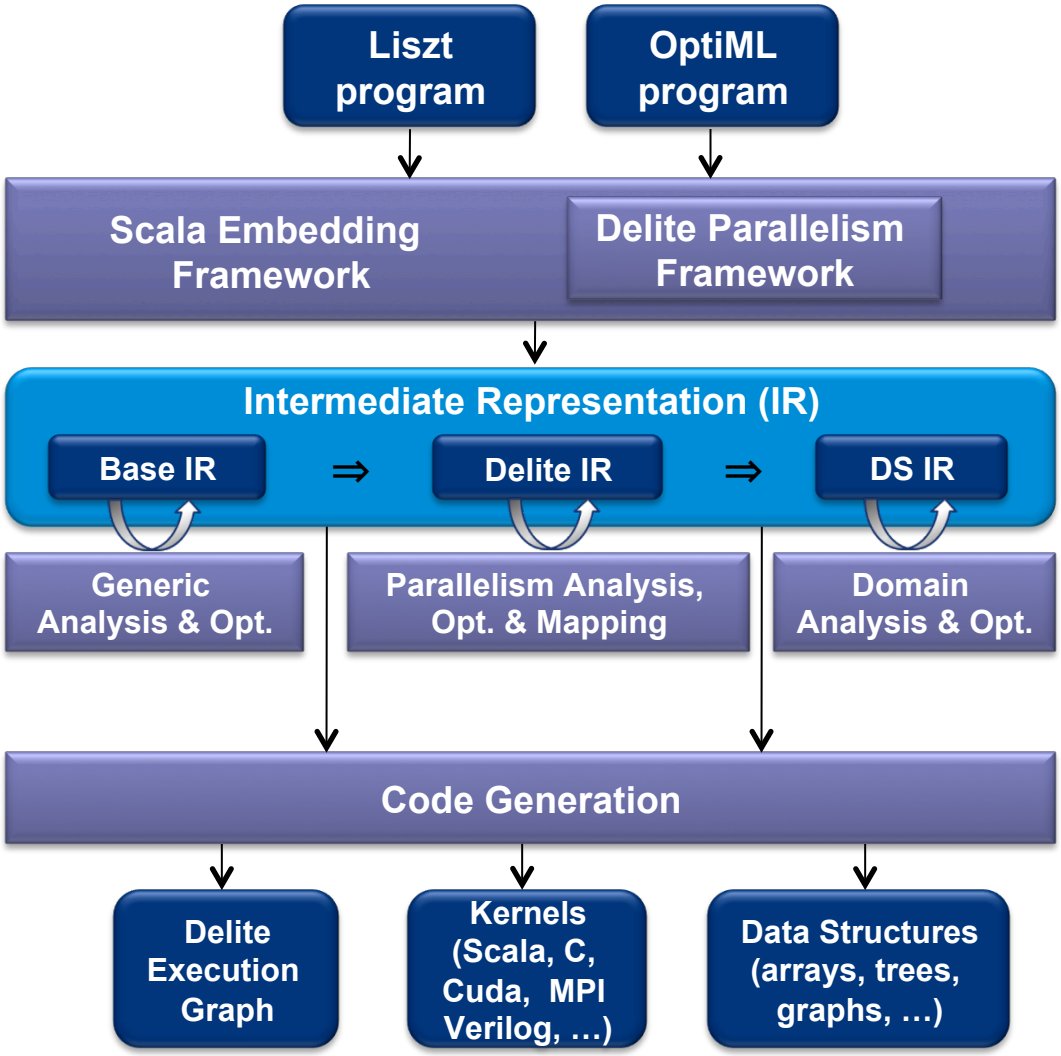
Lightweight Modular Staging Approach



Typical Compiler

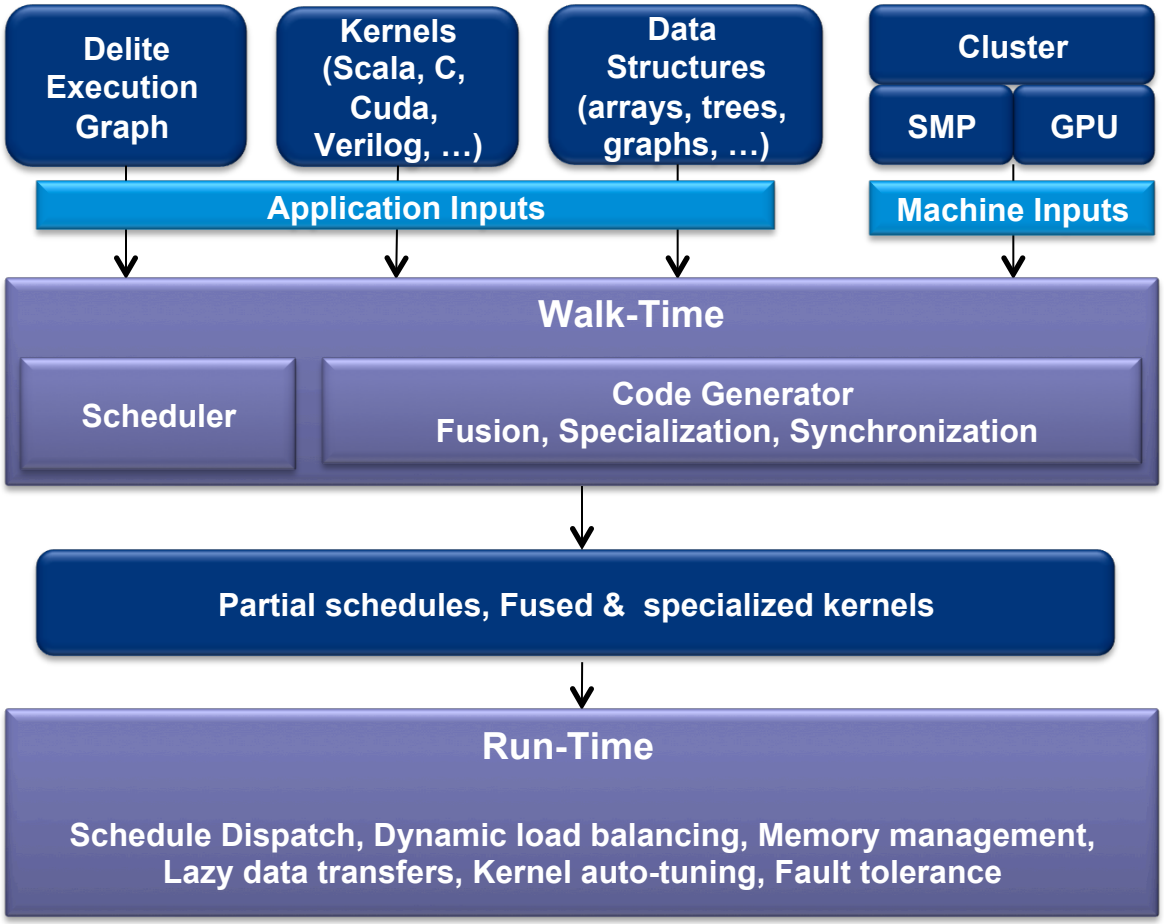
GPCE'10: Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs

Delite DSL Infrastructure



- Provide a common IR that can be extended while still benefitting 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 most appropriate data parallel nodes for their operations
  - Now can do domain-specific analysis and opt.
- Generate an execution graph, kernels and data structures

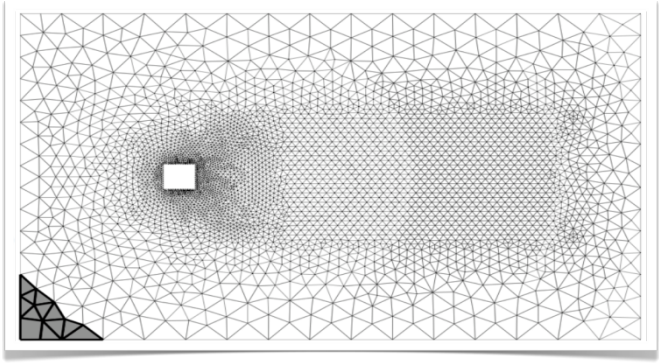
Delite Execution

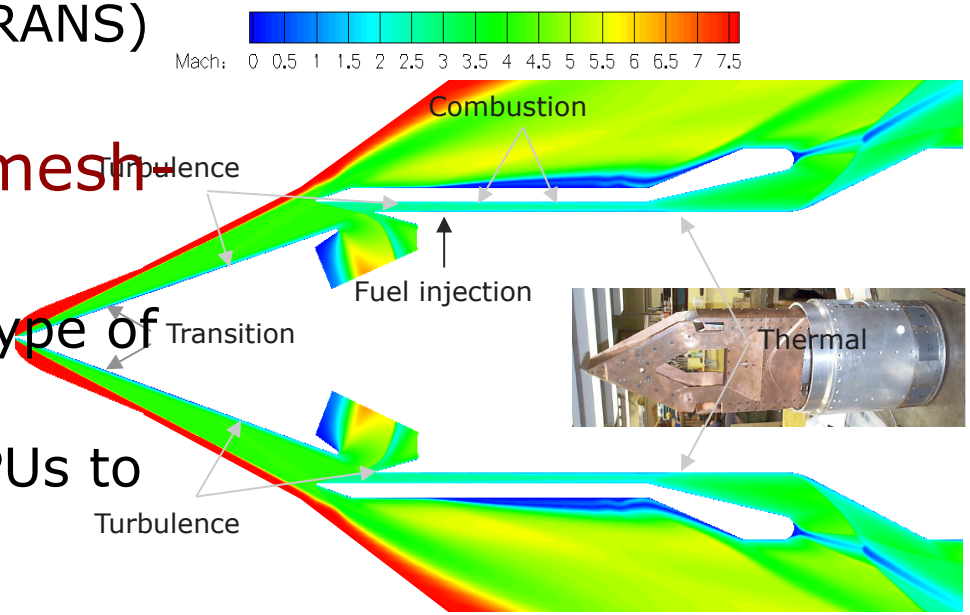


- Maps the machine-agnostic DSL compiler output onto the machine configuration for execution
- Walk-time scheduling produces partial schedules
- Code generation produces fused, specialized kernels to be launched on each resource
- Run-time executor controls and optimizes execution

Liszt: DSL for Mesh PDEs

- Solvers for mesh-based PDEs
  - Complex physical systems
  - Huge domains
  - millions of cells
  - Example: Unstructured Reynolds-averaged Navier Stokes (RANS) solver
- Goal: simplify code of mesh based PDE solvers
  - Write once, run on any type of parallel machine
  - From multi-cores and GPUs to clusters





Liszt Language Features

- Minimal Programming language
  - Aritmetic, short vectors, functions, control flow
- Built-in mesh interface for arbitrary polyhedra
  - `Vertex, Edge, Face, Cell`
  - `Optimized memory representation of mesh`
- Collections of mesh elements
  - `Element Sets: faces(c:Cell), edgesCCW(f:Face)`
- Mapping mesh elements to fields
  - `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

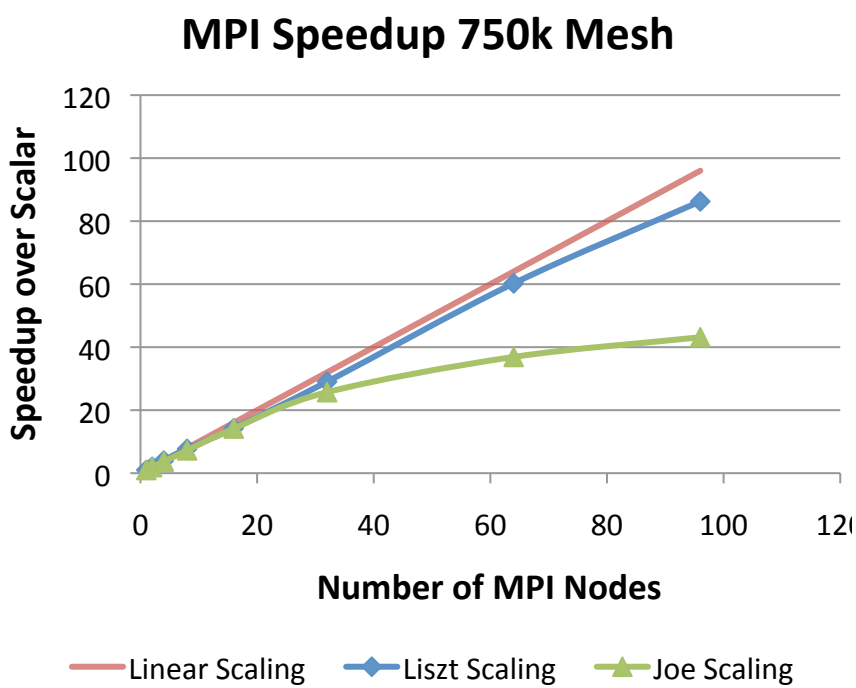
Code contains possible write conflicts!

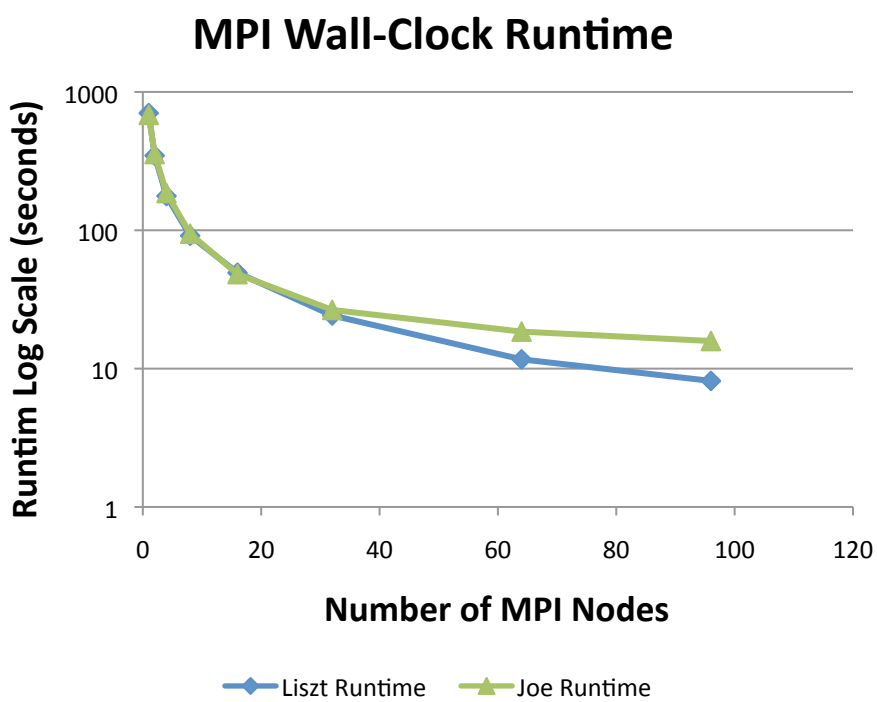
We use architecture specific strategies guided by domain knowledge

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

MPI Performance

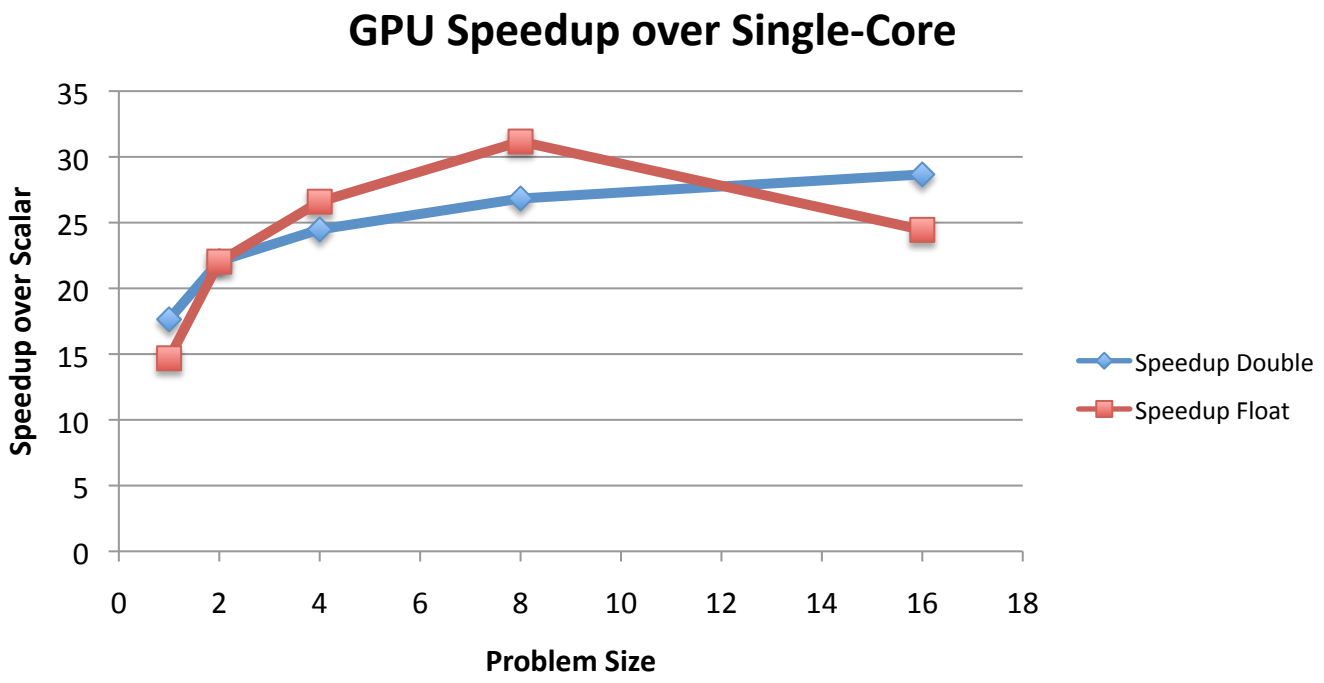
- Using 8 cores per node, scaling up to 96 cores (12 nodes, 8 cores per node, all communication using MPI)





GPU Performance

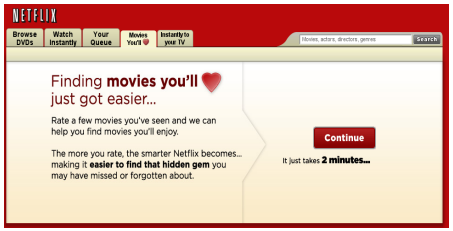
- Scaling mesh size from 50K (unit-sized) cells to 750K (16x) on a Tesla C2050. Comparison is against single threaded runtime on host CPU (Core 2 Quad 2.66Ghz)



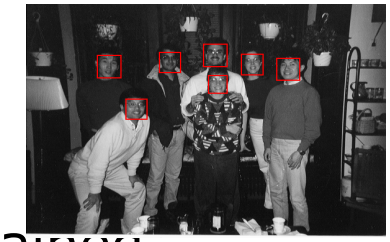
Single-Precision: 31.5x, Double-precision: 28x

OptiML: A DSL for ML

- Machine Learning domain
  - Learning patterns from data
  - Applying the learned models to tasks
    - Regression, classification, clustering, estimation
  - Computationally expensive
  - Regular and irregular parallelism



- Motivation for OptiML
  - Raise the level of abstraction
  - Use domain knowledge to identify coarse-grained parallelism
  - Single source ⇒ multiple heterogeneous targets
  - Domain specific optimizations



OptiML Language Features

- 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]
    - 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 { … }
  - Allow anonymous functions with restricted semantics to be passed as arguments of the control structures

Example OptiML / MATLAB code (Gaussian Discriminant Analysis)

ML-specific data types

```
// 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)
    }
}
```

Implicitly parallel control structures

Restricted index semantics

OptiML code

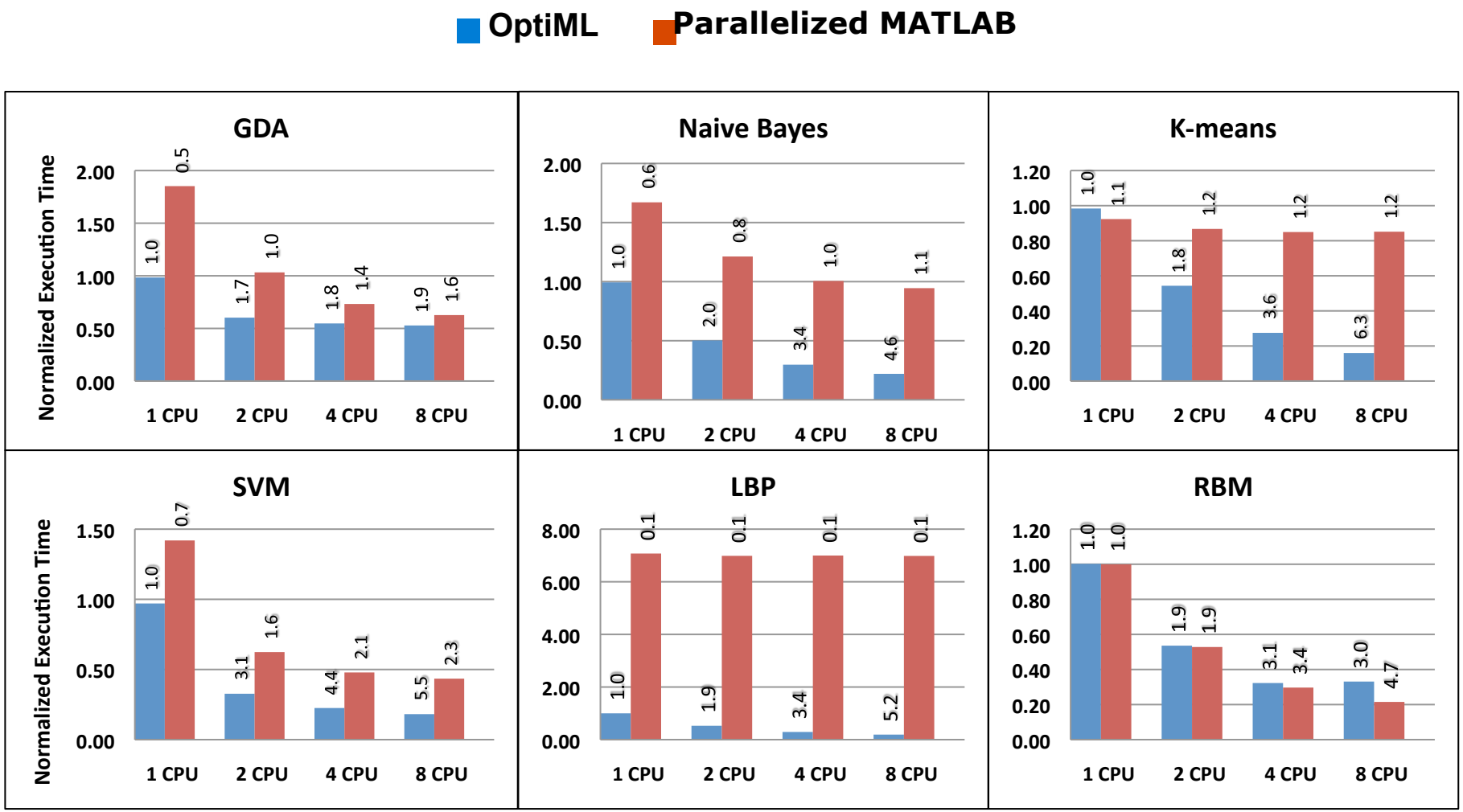
```
% x : Matrix, y: Vector
% mu0, mu1: Vector

n = size(x,2);
sigma = zeros(n,n);

parfor i=1:length(y)
    if (y(i) == 0)
        sigma = sigma + (x(i,:)-mu0)'*(x(i,:)-mu0);
    else
        sigma = sigma + (x(i,:)-mu1)'*(x(i,:)-mu1);
    end
end
```

(parallel) MATLAB code

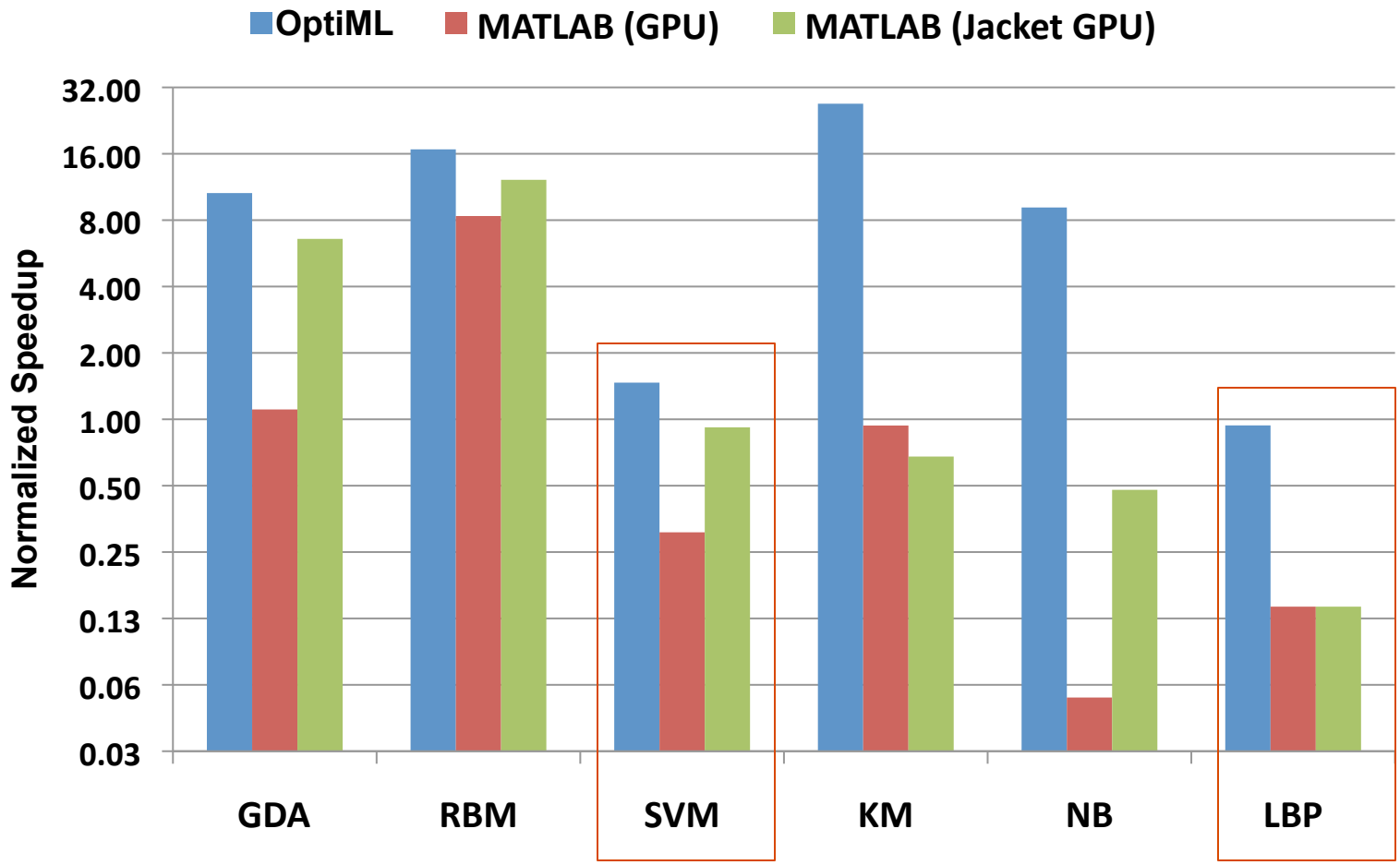
Experimental Setup

- 4 Different implementations
  - OptiML+Delite
  - MATLAB (Original, GPU, Jacket)
- System 1: Performance Tests
  - Intel Nehalem
  - 2 sockets, 8 cores, 16 threads
  - 24 GB DRAM
  - NVIDIA GTX 275 GPU
- System 2: Scalability Tests
  - Sun Niagara T2+
  - 4 sockets, 32 cores, 256 threads
  - 128 GB DRAM

Performance Study (CPU)

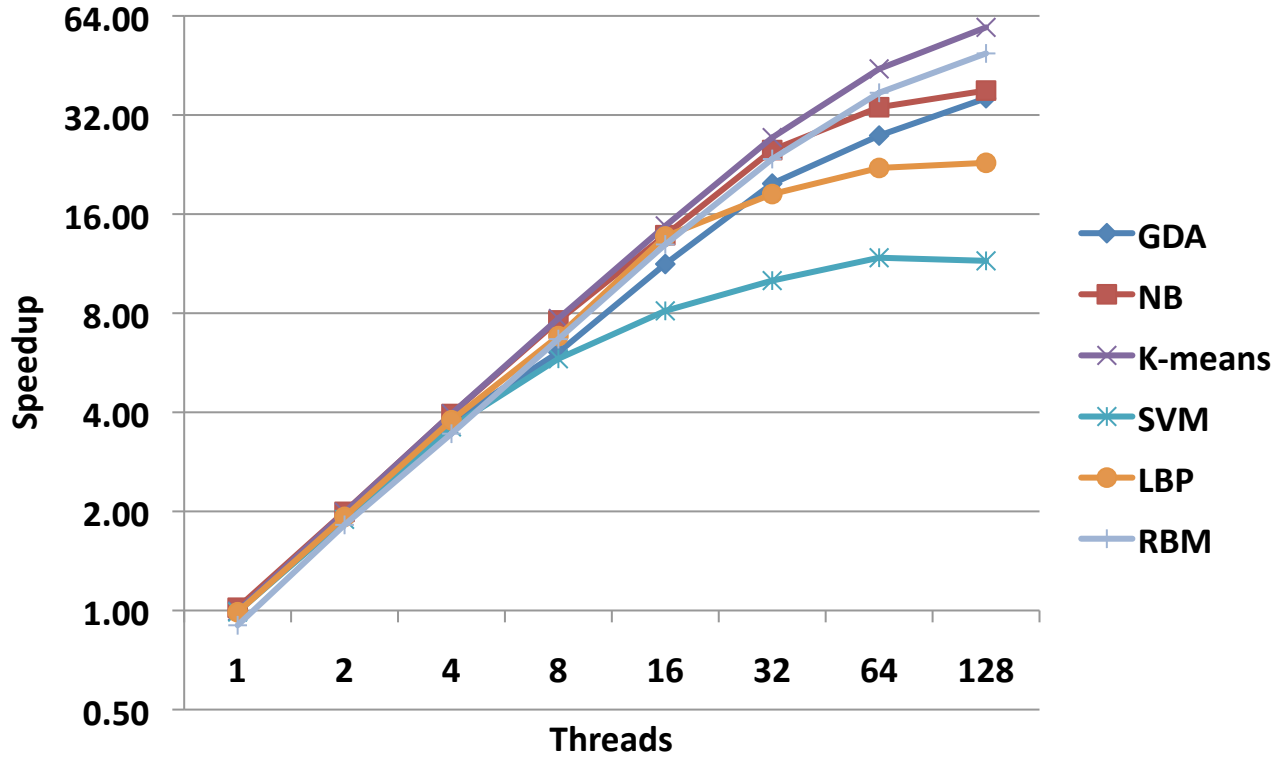


Performance Study (GPU)

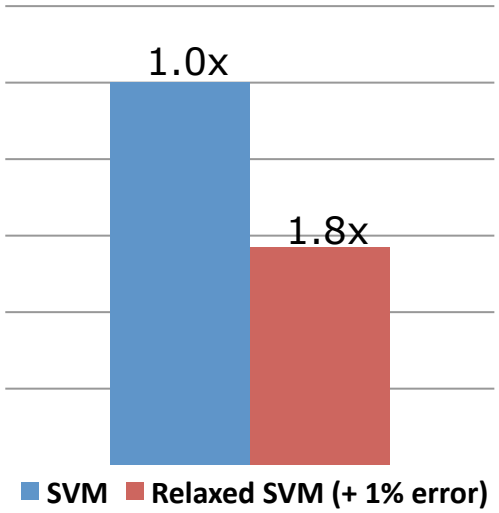


Speedup relative to 1 core OptiML execution time on Nehalem system

Scalability Study (T2+)



Domain Specific Optimizations



Speedup relative to 8 core execution time on Nehalem system

Conclusions

- DSLs can provide the answer to the heterogeneous parallel programming problem
- Need to simplify the process of generating DSLs for parallelism
  - Need programming languages to be designed for flexible embedding
  - Lightweight modular staging in Scala allows for more powerful embedded DSLs
  - Delite provides a framework for adding parallelism
- Early embedded DSL results are very promising