Taming Heterogeneous Parallelism with Domain Specific Languages

WOLFHPC 2011

Kunle Olukotun
Pervasive Parallelism Laboratory
Stanford University

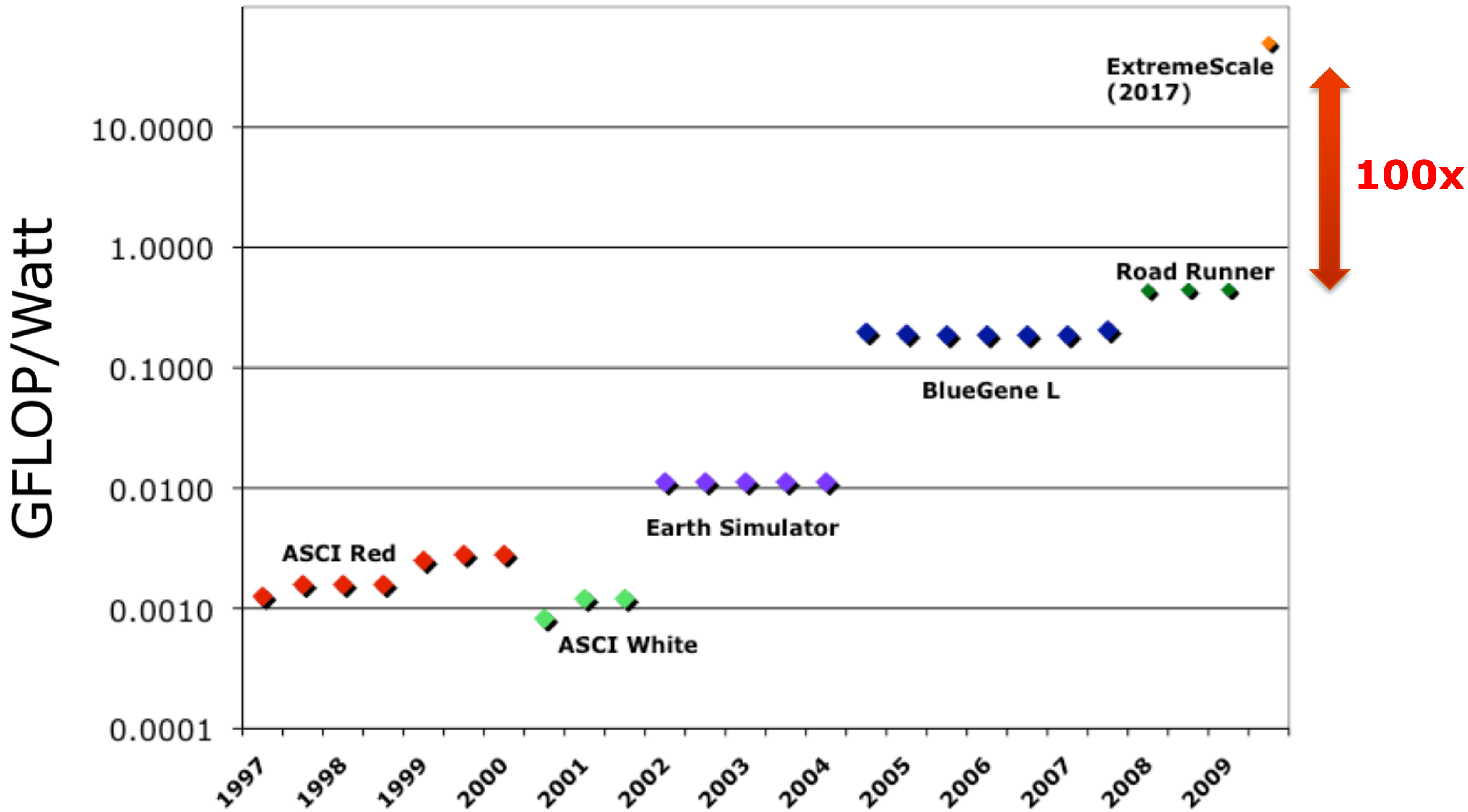
Outline

- Motivation for DSLs
- Liszt for mesh-based PDEs
- OptiML for machine learning
- Delite a framework for DSLs

Computing Goals: The 4 Ps

- Power
- Performance
- Productivity
- Portability

Exascale: 100:1 Improvement Needed



Source: DARPA Exascale Hardware and Software Studies

Computing System Power

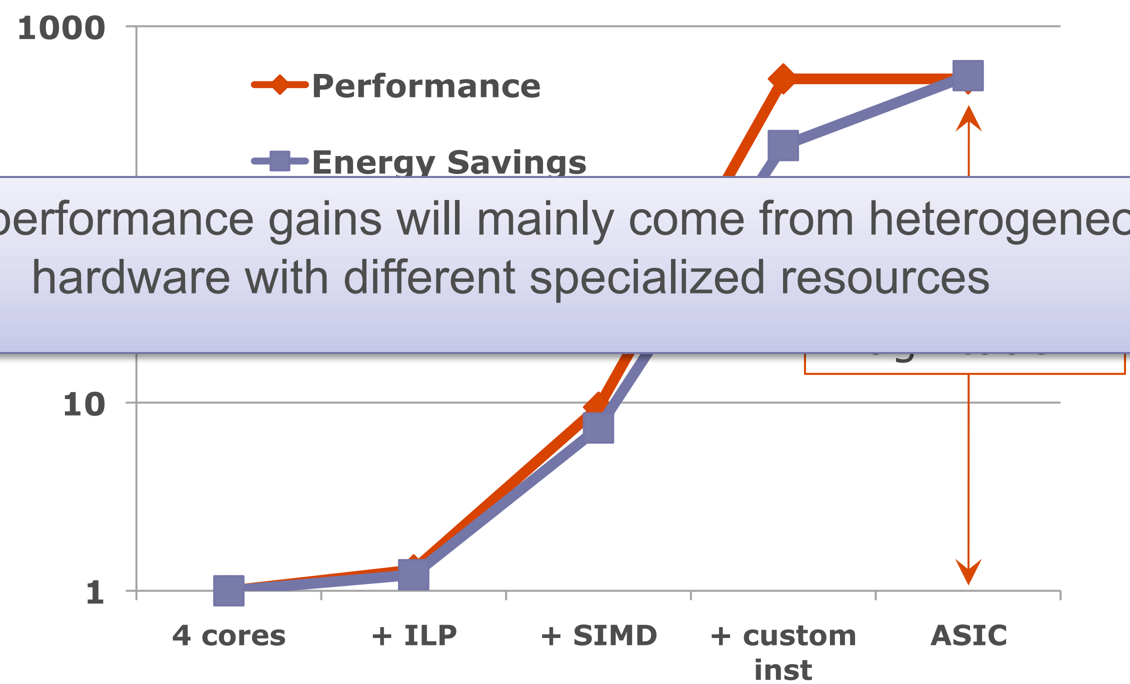
$$Power = Energy_{Op} \times \frac{Ops}{second}$$

FIXED

Heterogeneous Hardware

- Heterogeneous HW for energy efficiency
  - Multi-core, ILP, threads, data-parallel engines, custom engines

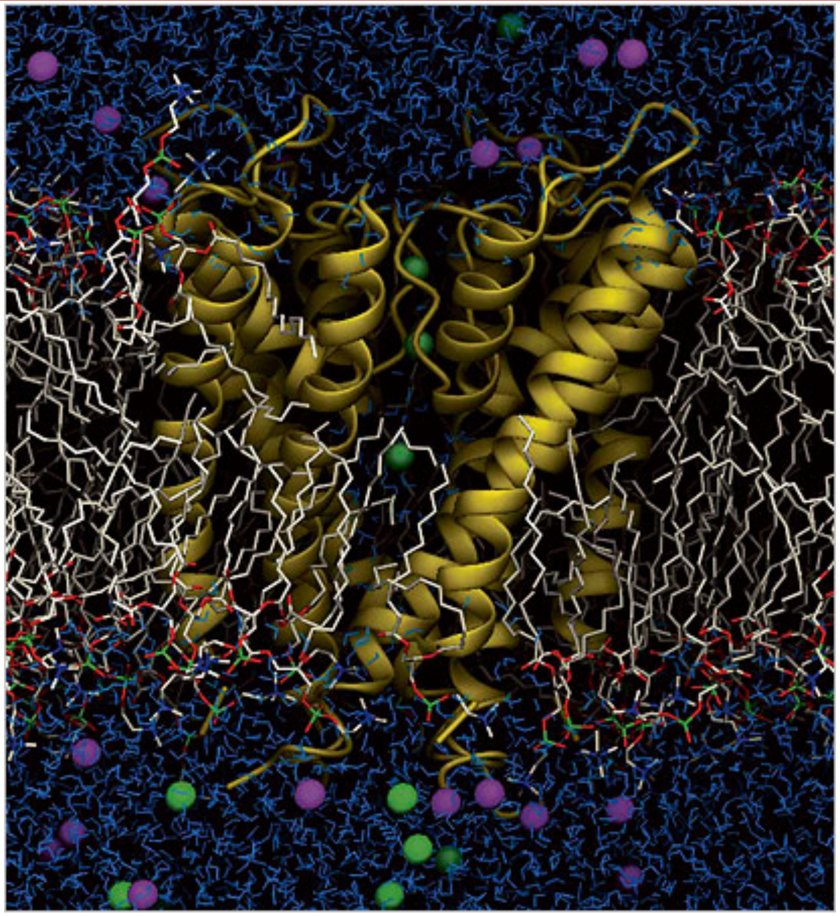
- H.264 encode study



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



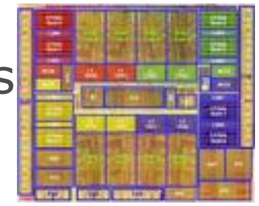
Molecular dynamics computer

100 times more power efficient

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

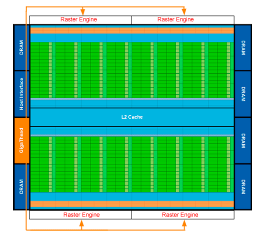
Heterogeneous Parallel Programming Today

Pthreads
OpenMP



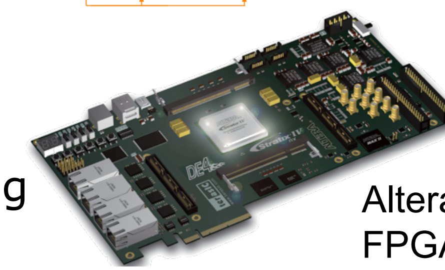
Sun T2

CUDA
OpenCL



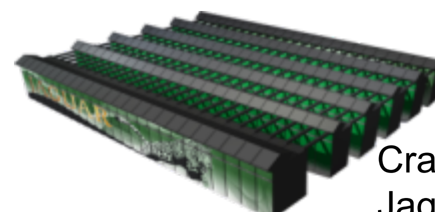
Nvidia Fermi

Verilog
VHDL



Altera FPGA

MPI
PGAS



Cray Jaguar

Programmability Chasm

Applications

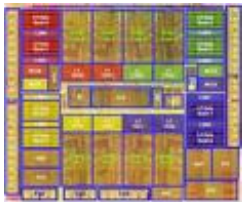
Scientific Engineering

Virtual Worlds

Personal Robotics

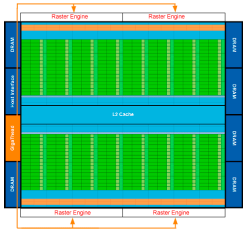
Data informatics

Pthreads
OpenMP



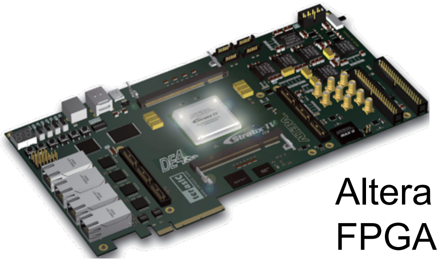
Sun T2

CUDA
OpenCL



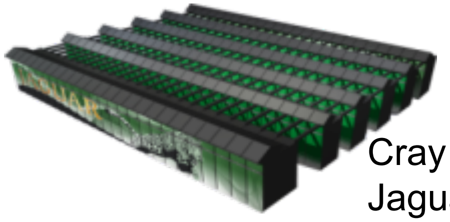
Nvidia Fermi

Verilog
VHDL



Altera FPGA

MPI
PGAS



Cray Jaguar

Too many different low-level programming models

Programmability Chasm

Applications

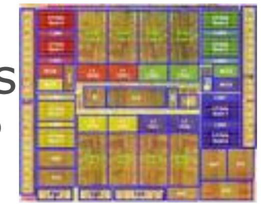
Scientific Engineering

Virtual Worlds

Personal Robotics

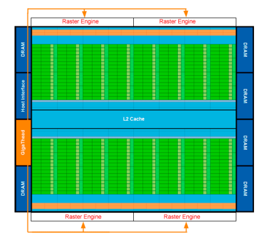
Data informatics

Pthreads
OpenMP



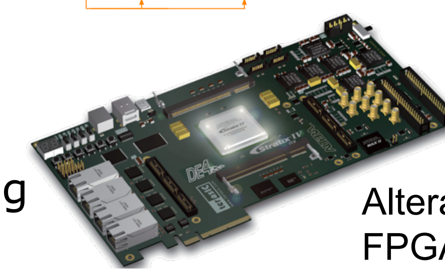
Sun T2

CUDA
OpenCL



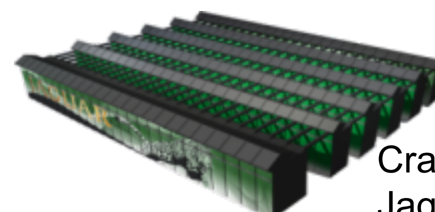
Nvidia Fermi

Verilog
VHDL



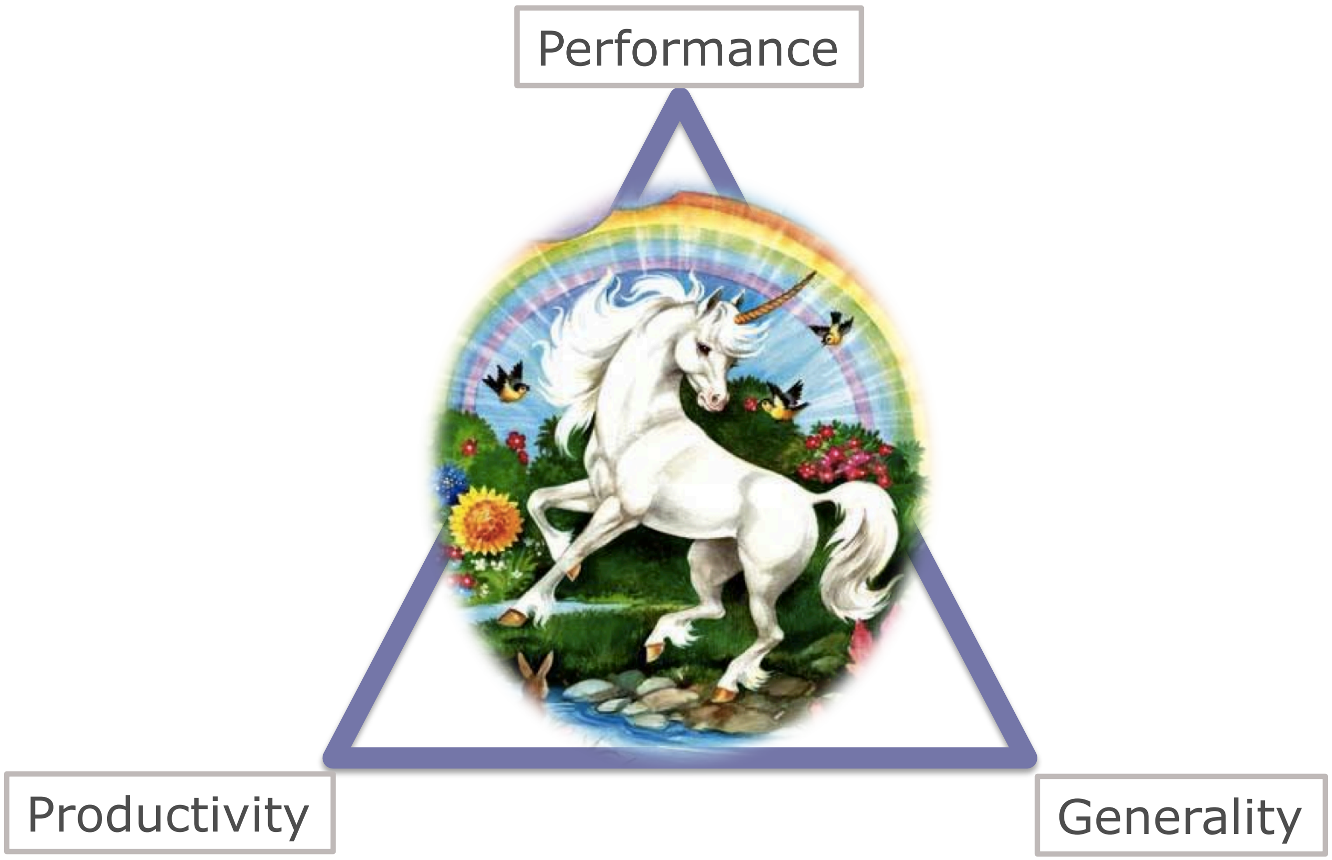
Altera FPGA

MPI
PGAS



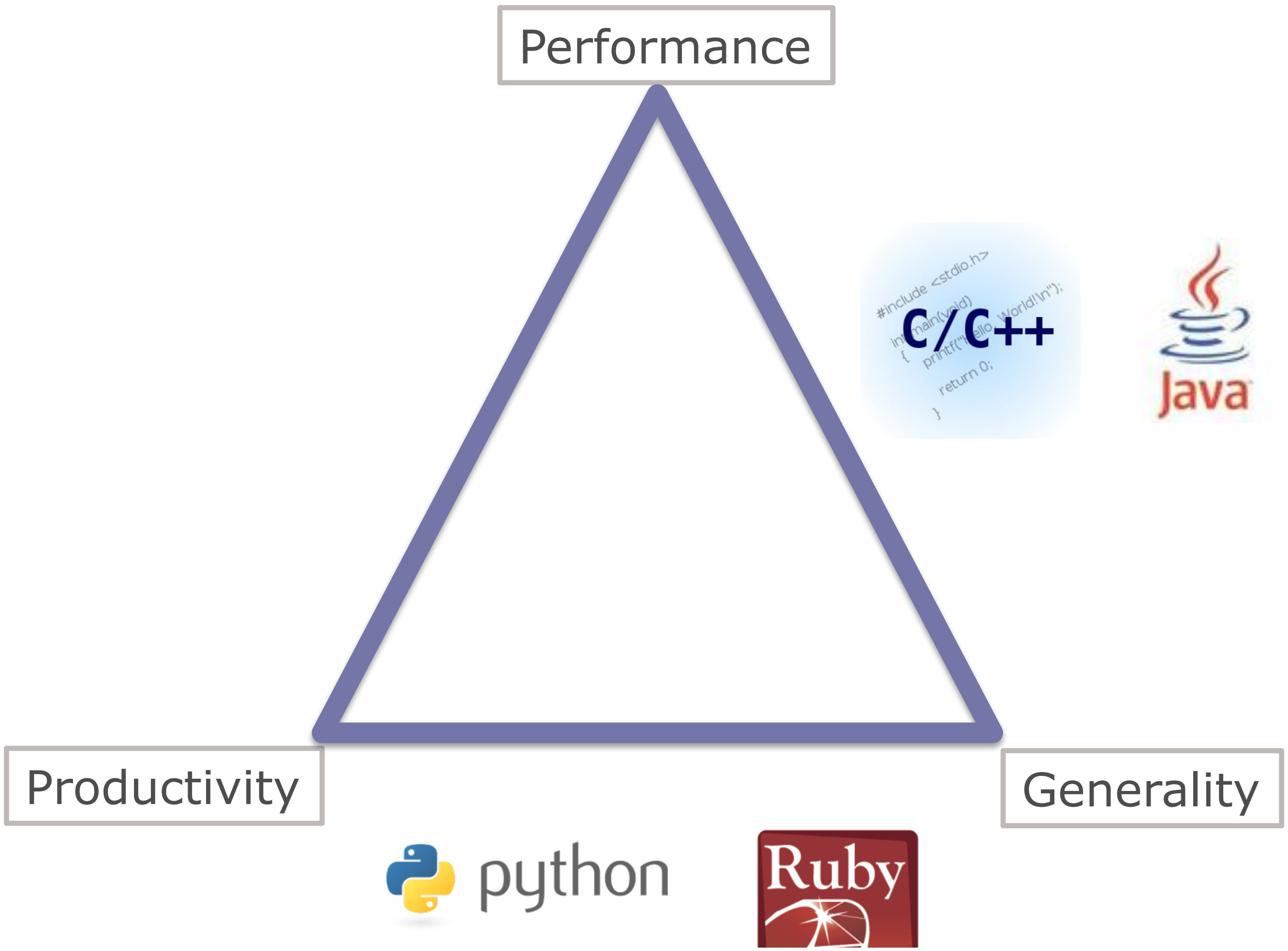
Cray Jaguar

The Ideal Parallel Programming Language

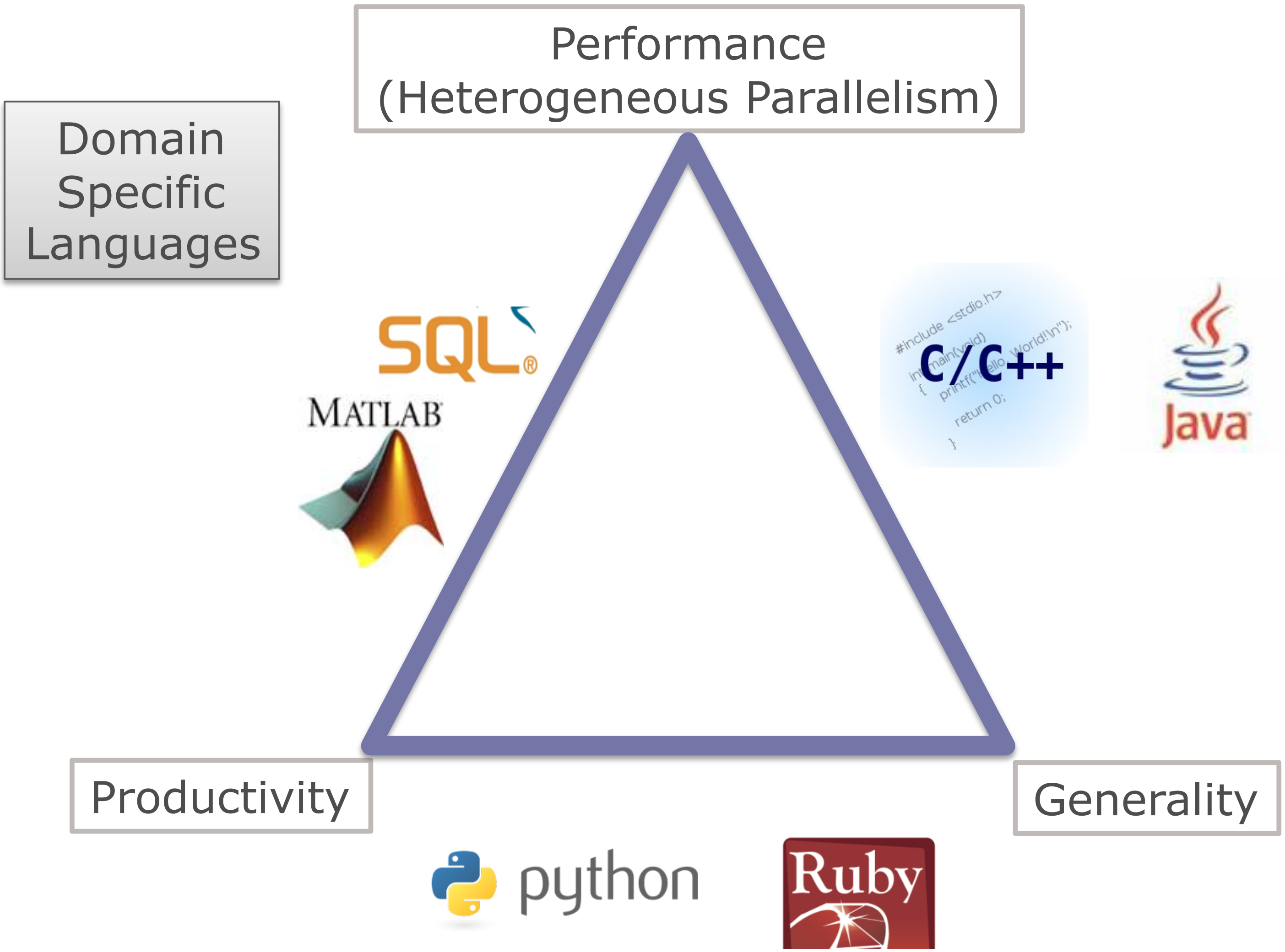


Successful Languages

PERVASIVE PARALLELISM LABORATORY



Way Forward ⇒ Domain Specific Languages



Domain Specific Languages

- Domain Specific Languages (DSLs)
  - Definition: A language or library with restrictive expessiveness that exploits domain knowledge for productivity and efficiency
  - High-level, usually declarative, and deterministic

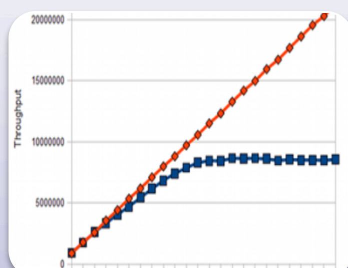
DSL Benifits

Productivity

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

Performance

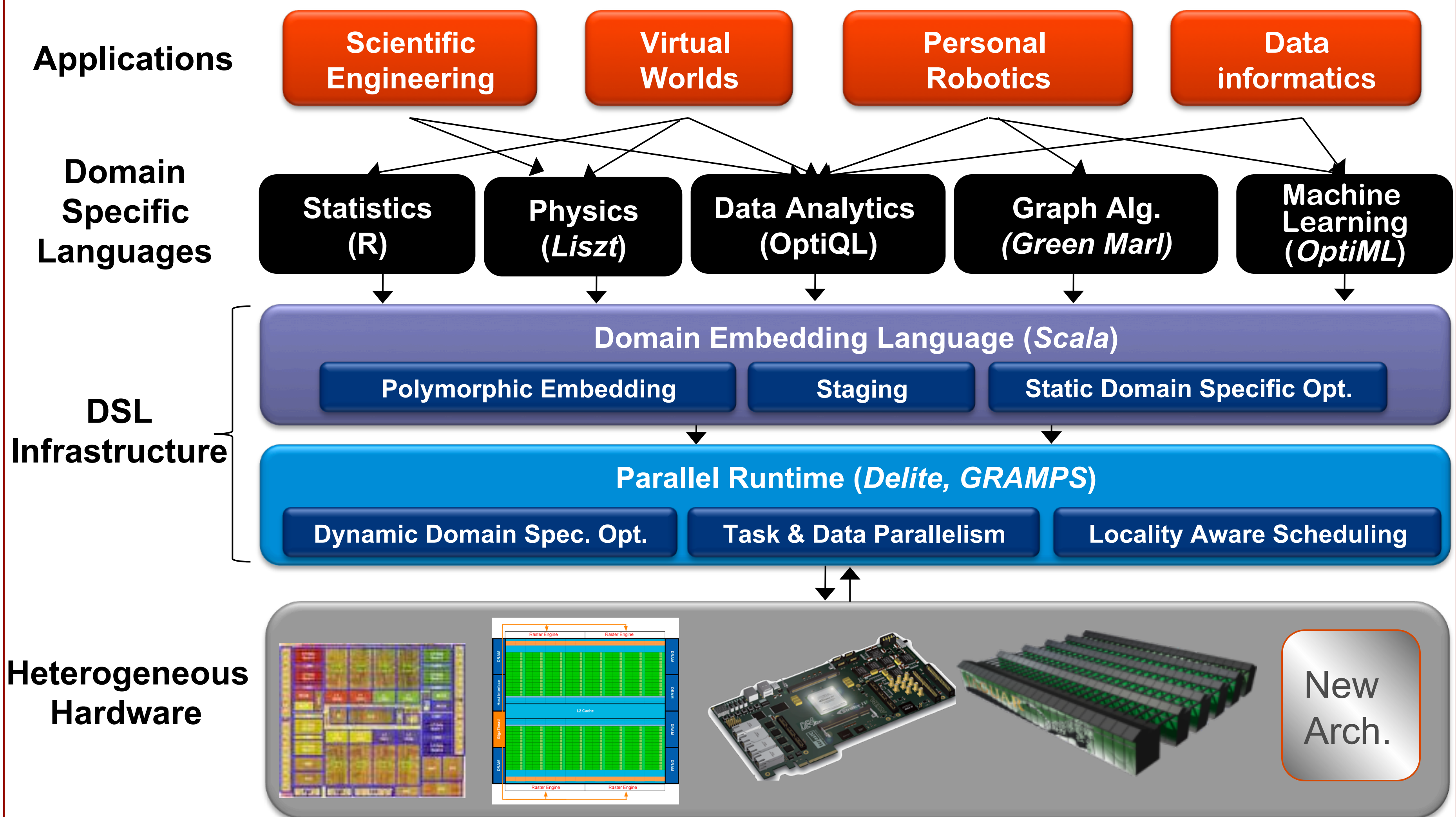
- Match high level domain abstraction to generic parallel execution patterns
- Restrict expressiveness to more easily and fully extract available parallelism
- Use domain knowledge for static/dynamic optimizations



Portability and forward scalability

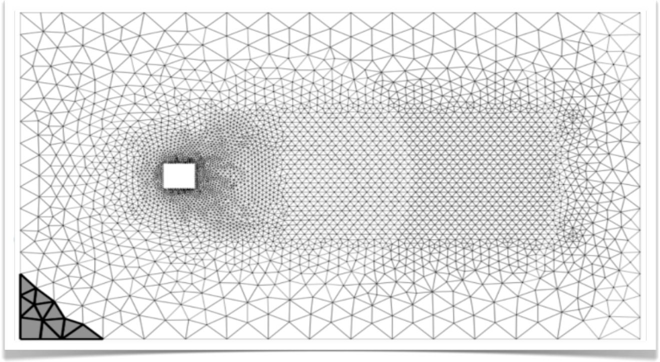
- DSL & Runtime can be evolved to take advantage of latest hardware features
- Applications remain unchanged
- Allows innovative HW without worrying about application portability

Bridging the Programmability Chasm



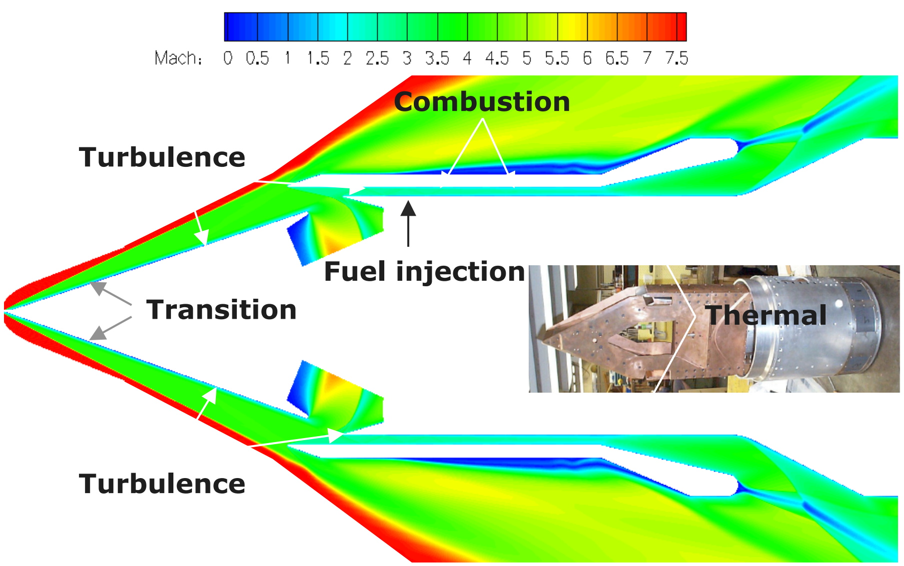
Liszt: DSL for Mesh PDEs

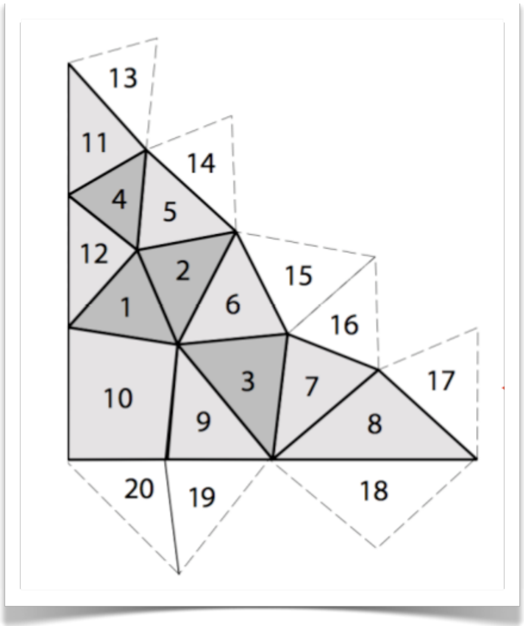
- Z. DeVito, N. Joubert, P. Hanrahan
- Solvers for mesh-based PDEs
  - Complex physical systems
  - Huge domains
  - millions of cells
  - Example: Unstructured Reynolds-averaged Navier Stokes (RANS) solver
- Goal: simplify code of mesh-based PDE solvers
  - Write once, run on any type of parallel machine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



PSAAP's Joe

- State-of-the-art unstructured Reynolds-averaged Navier Stokes (RANS) solver
- Main tool for system-level simulation
  - Highly optimized for MPI clusters
  - Fortran heritage





Features of high performance PDE solvers

- Find Parallelism
  - Data-parallelism on mesh elements
- Expose Data Locality
  - PDE Operators have local support
  - Stencil captures exact region of support
- Reason about Synchronization
  - Iterative solvers
  - Read old values to calculate new values

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) ) { … }`

Example: Heat Conduction on Grid

```
val Position = FieldWithLabel[Vertex,Float3](“position”)
val Temperature = FieldWithConst[Vertex,Float](0.0f)
val Flux = FieldWithConst [Vertex,Float](0.0f)
val JacobiStep = FieldWithConst[Vertex,Float](0.0f)
var i = 0;
while (i < 1000) {
  for (e <- edges(mesh)) {
    val v1 = head(e)
    val v2 = tail(e)
    val dP = Position(v1) - Position(v2)
    val dT = Temperature(v1) - Temperature(v2)
    val step = 1.0f/(length(dP))
    Flux(v1) += dT*step
    Flux(v2) -= dT*step
    JacobiStep(v1) += step
    JacobiStep(v2) += step
  }
  for (p <- vertices(mesh)) {
    Temperature(p) += 0.01f*Flux(p)/JacobiStep(p)
  }
  for (p <- vertices(mesh)) {
    Flux(p) = 0.f; JacobiStep(p) = 0.f;
  }
  i += 1
}
```

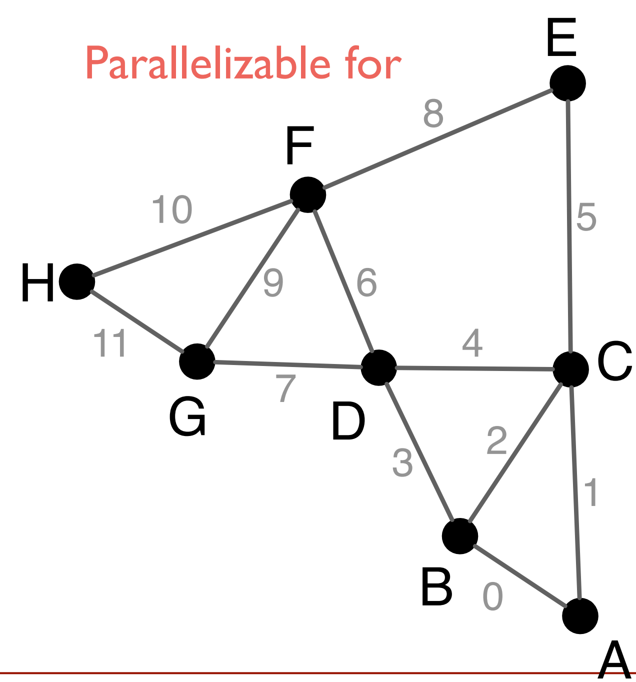
Mesh Elements

Topology Functions

Sets

Fields (Data storage)

Parallelizable for



Infer Data Accesses from Liszt

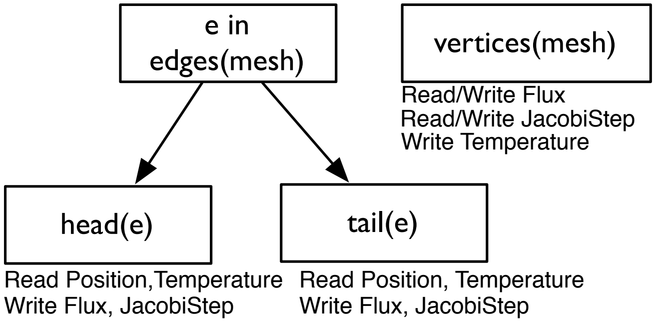
- "Stencil" of a piece of code:
  - Captures just the memory accesses it performs

- Infer stencil for each for-comprehension in Liszt

Domain Specific Transform: Stencil Detection

- Analyze code to detect memory access stencil of each top-level for-all comprehension
  - Extract nested mesh element reads
  - Extract field operations
  - Difficult with a traditional library

```
for (e <- edges(mesh)) {
  val v1 = head(e)
  val v2 = tail(e)
  val dP = Position(v1) - Position(v2)
  val dT = Temperature(v1) - Temperature(v2)
  val step = 1.0f/(length(dP))
  Flux(v1) += dT*step
  Flux(v2) -= dT*step
  JacobiStep(v1) += step
  JacobiStep(v2) += step
}
```



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

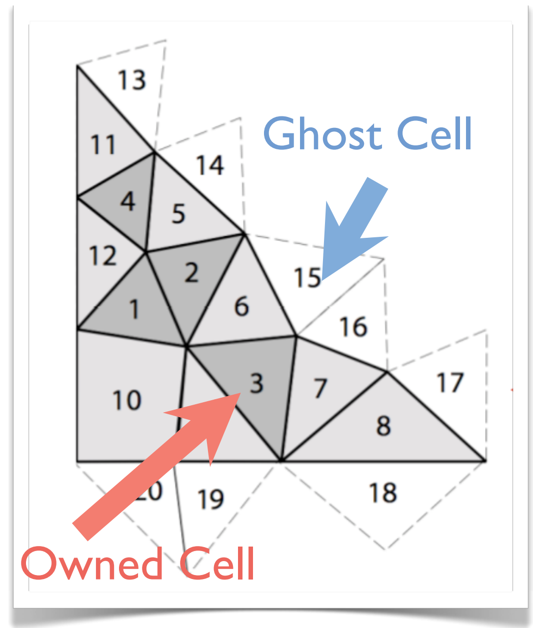
Code contains possible write conflicts!

We use architecture specific strategies guided by domain knowledge

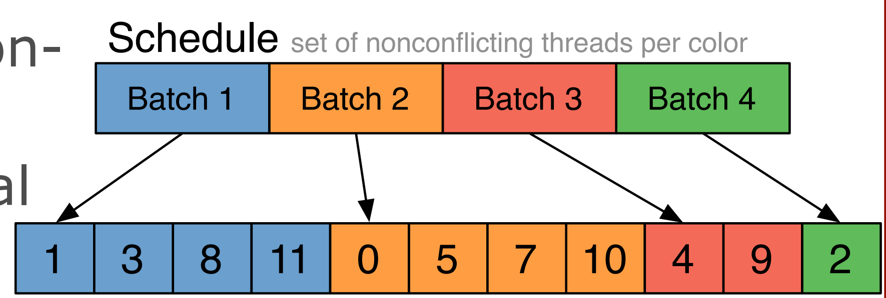
- MPI: Ghost cell-based message passing
- GPU: Coloring-based use of shared memory

Execution Strategies

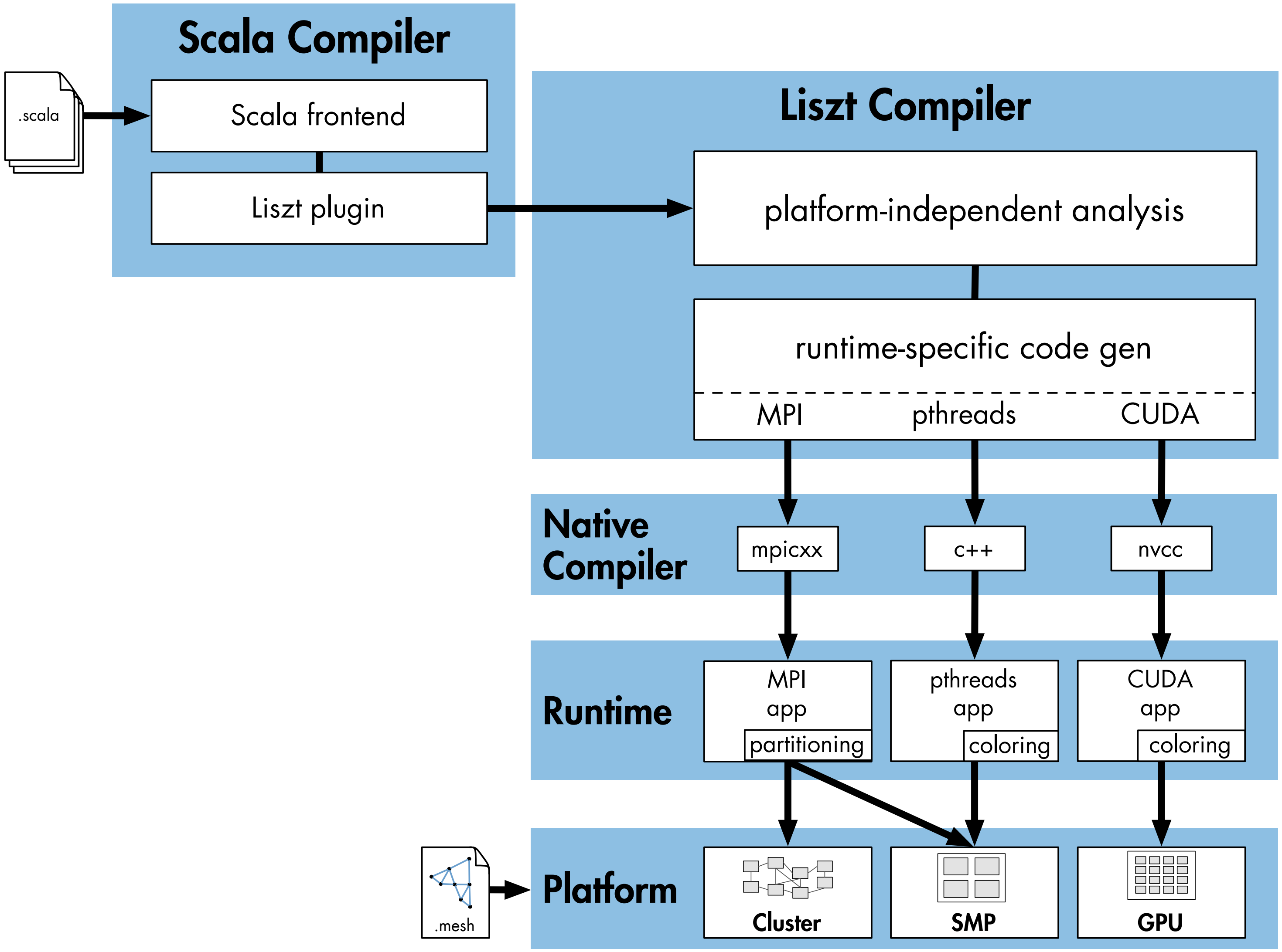
- Partitioning
  - Assign partition to each computational unit
  - Use ghost elements to coordinate cross-boundary communication.
  - Ideal for single computational unit per memory space



- Coloring
  - Calculate interference between work items on domain
  - Schedule work-items into non-interfering batches
  - Ideal for many computational units per memory space



Architecture



Results

- 4 example codes with Liszt and C++ implementations:
  - Euler solver from Joe
  - Navier-Stokes solver from Joe
  - Shallow Water simulator
    - Free-surface simulation on globe as per Drake et al.
    - Second order accurate spatial scheme
  - Linear FEM
    - Hexahedral mesh
    - Trilinear basis functions with support at vertices
    - CG solver

Scalar Performance Comparisons

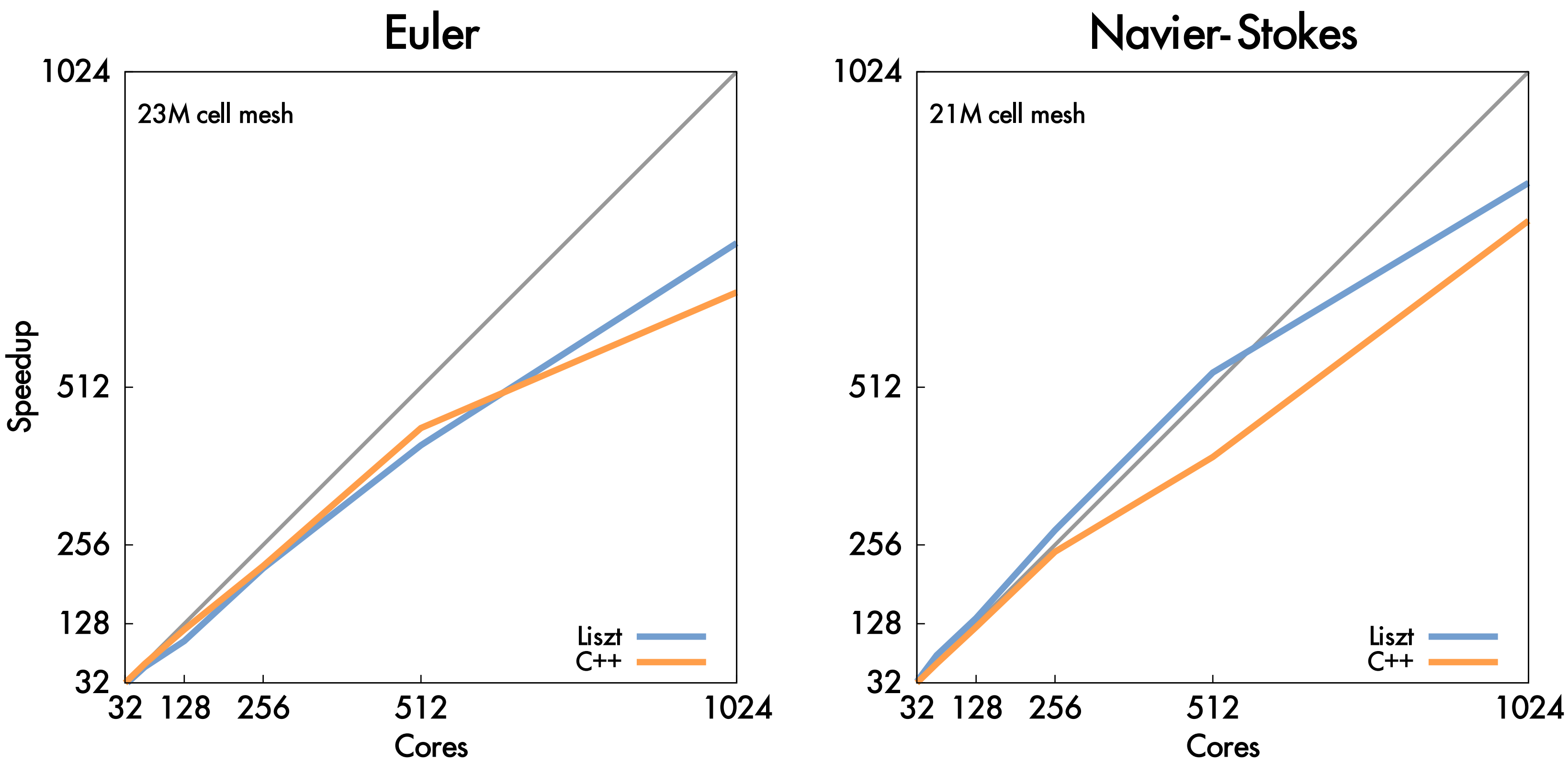
- Runtime comparisons between hand-tuned C++ and Liszt

- Liszt performance within 12% of C++

	Euler	Navier-Stokes	FEM	Shallow Water
Mesh size	367k	668k	216k	327k
Liszt	0.37s	1.31s	0.22s	3.30s
C++	0.39s	1.55s	0.19s	3.34s

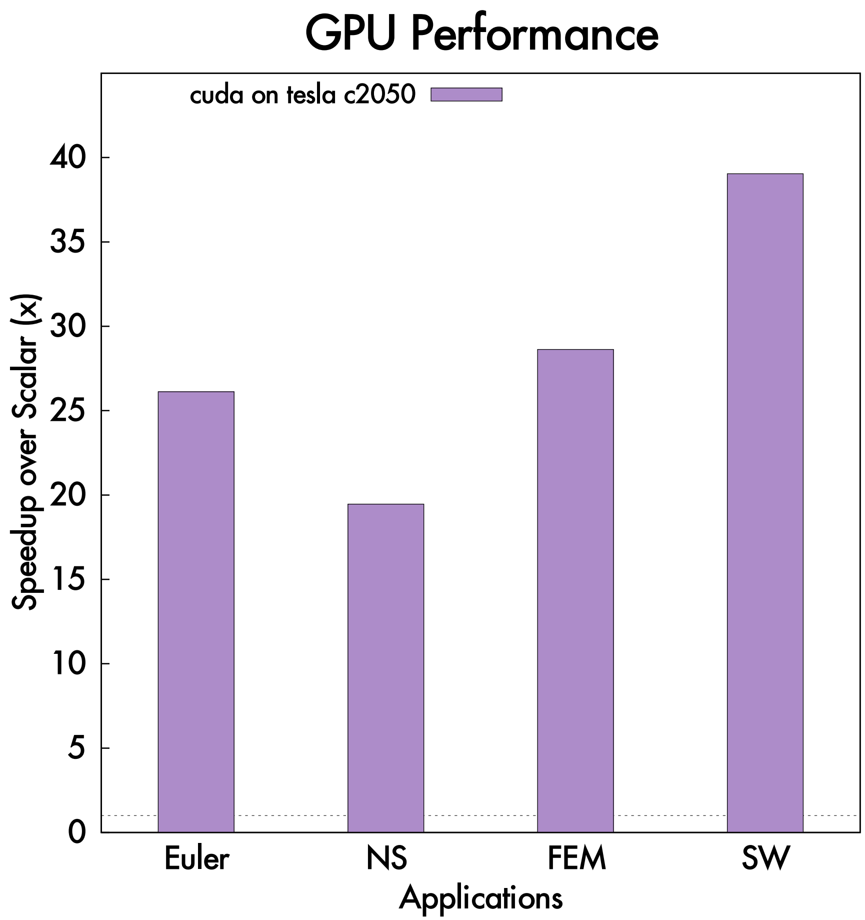
MPI Performance

- 4-socket 6-core 2.66Ghz Xeon CPU per node (24 cores), 16GB RAM per node. 256 nodes, 8 cores per node



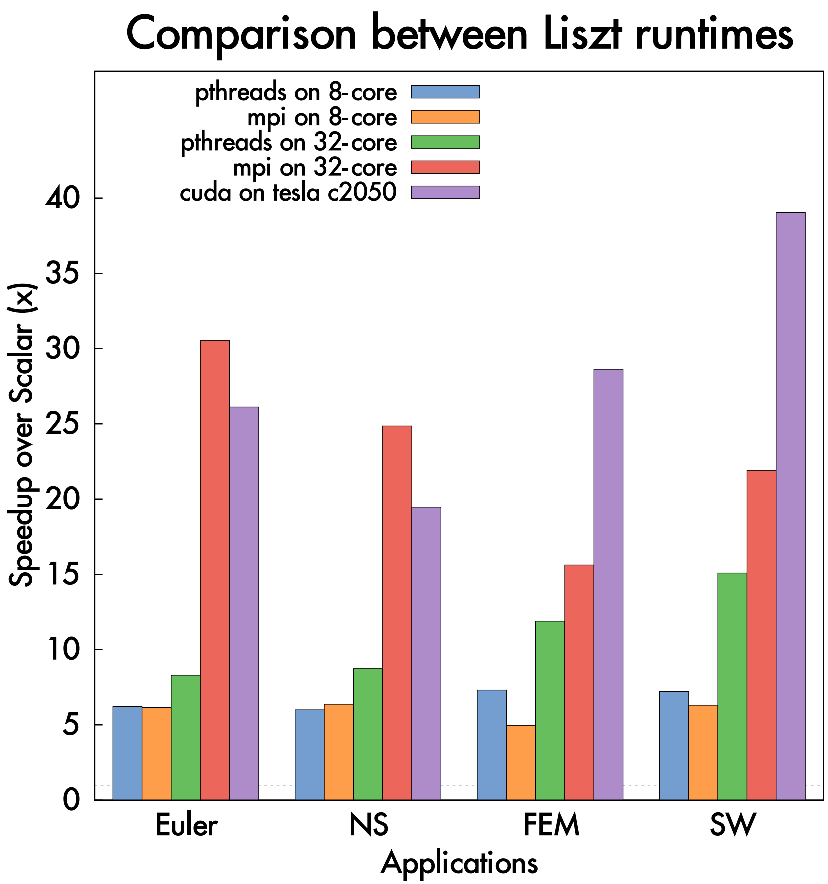
GPU Performance

- Tesla C2050, Double Precision, compared to single core, Nehalem E5520 2.26Ghz, 8GB RAM



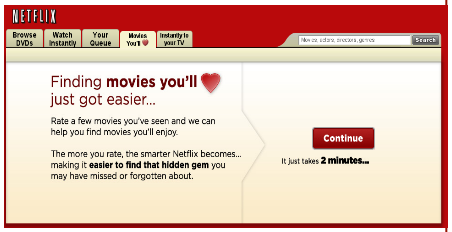
Portability

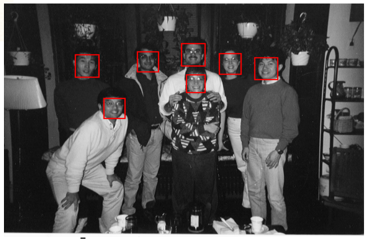
- Tested both pthreads (coloring) and MPI (partitioning) runtime on:
  - 8-core Nehalem E5520 2.26Ghz, 8GB RAM
  - 32-core Nehalem-EX X7560 2.26GHz, 128GB RAM



OptiML: A DSL for ML

- A. Sujeeth and H. Chafi
- 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

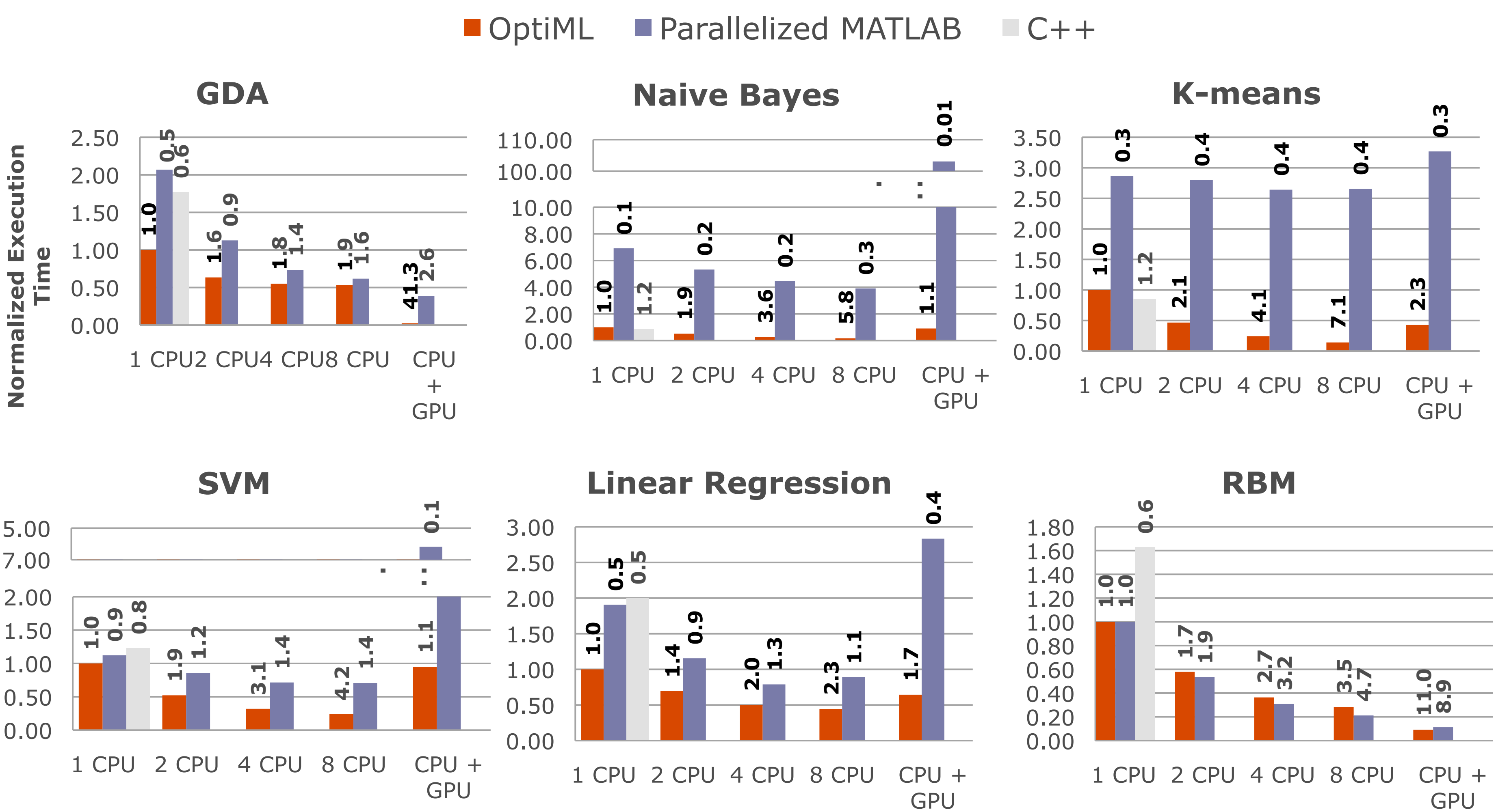
Restricted index semantics

```
% 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
```

OptiML code

(parallel) MATLAB code

OptiML vs. Matlab vs. C++



More DSLs …

- Graphs and graph algorithms
  - BFS, maximum flow, matching, assignment, components and connectivity, . . .
  - Social networks, data analysis
- Bio-simulation
  - Molecular dynamics, cells & viruses , drug-design, prosthetics
- Query Language
  - Relations, data analytics, financial trading
- Computational Geometry
  - Arbitrary polyhedra, convex hull, delauny triangulation, . . .
- Visualization
  - Protovis, Data wrangler

- Your DSL goes here

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

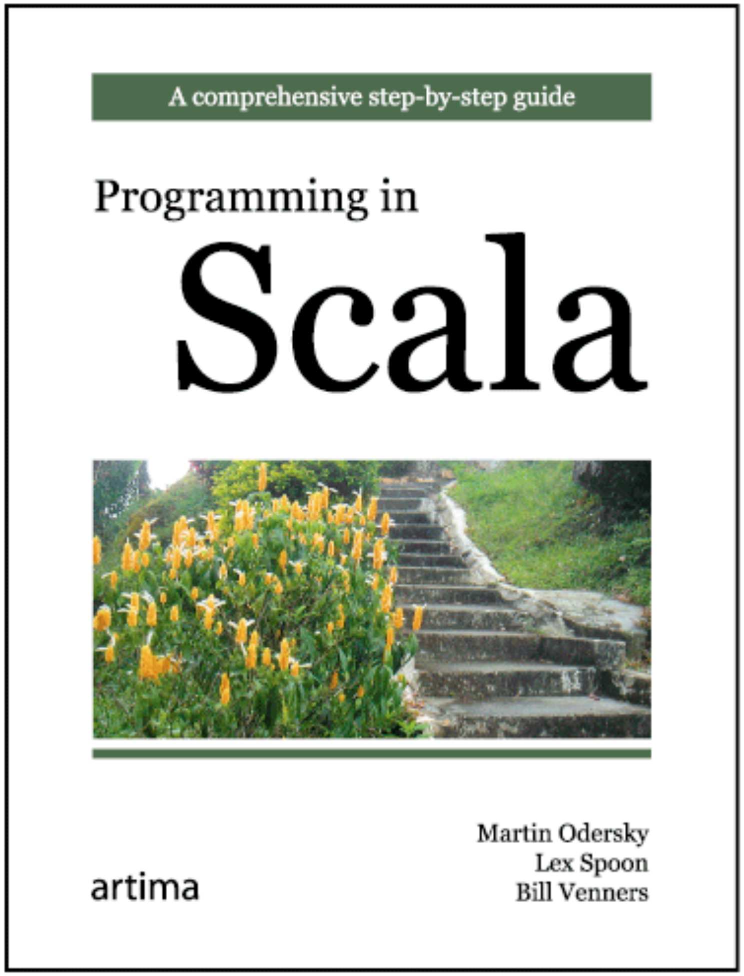
- Purely embedded DSLs ⇒ “just a library”
  - Easy to develop (can reuse full host language)
  - Easier to learn DSL
  - Can Combine multiple DSLs in one program
  - Can Share DSL infrastructure among several DSLs
  - Hard to optimize using domain knowledge
  - Target same architecture as host language

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

More Powerful Embedded DSLs

- Constructs of the embedding language can be overriden by the DSL:

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

maps to

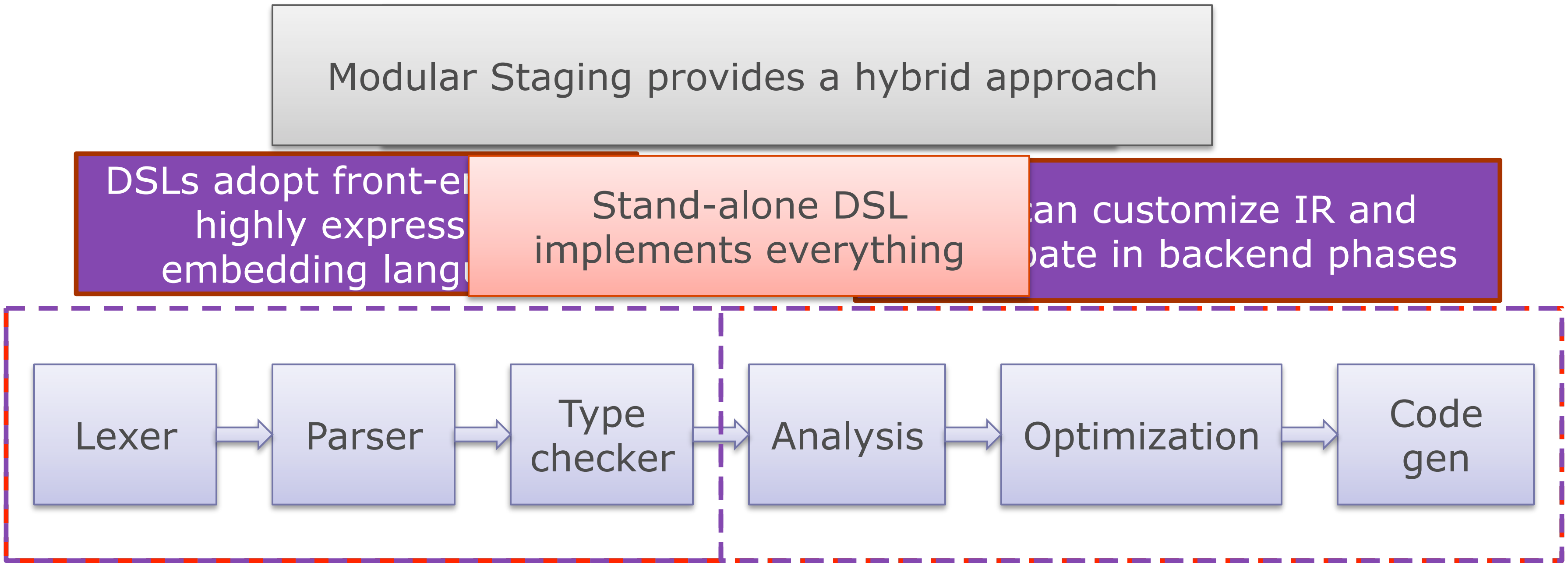
```
__ifThenElse(cond, something, somethingElse)
```

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

Lifting Scala to IR

- What we lift into the embedding world
  - DSL-defined methods
  - Basic Scala types (primitives, Arrays, Lists, Tuples, etc.)
  - Control structures (If, For, While, …)
  - Equality
  - Variable declaration and assignment
  - Functions
- What we don't lift (yet)
  - Classes
  - Methods

Lightweight Modular Staging Approach

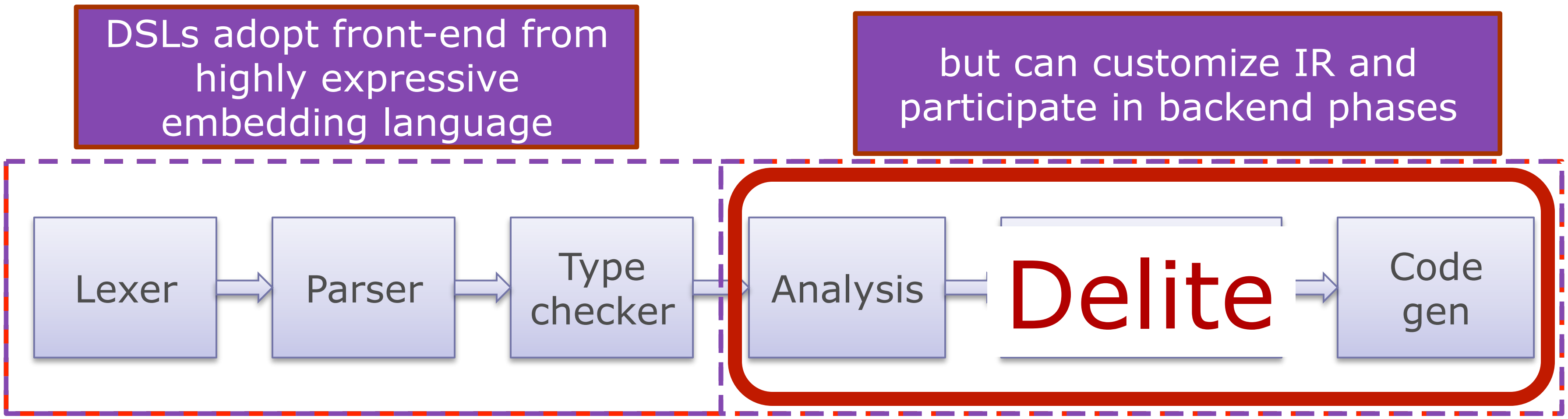


Typical Compiler

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

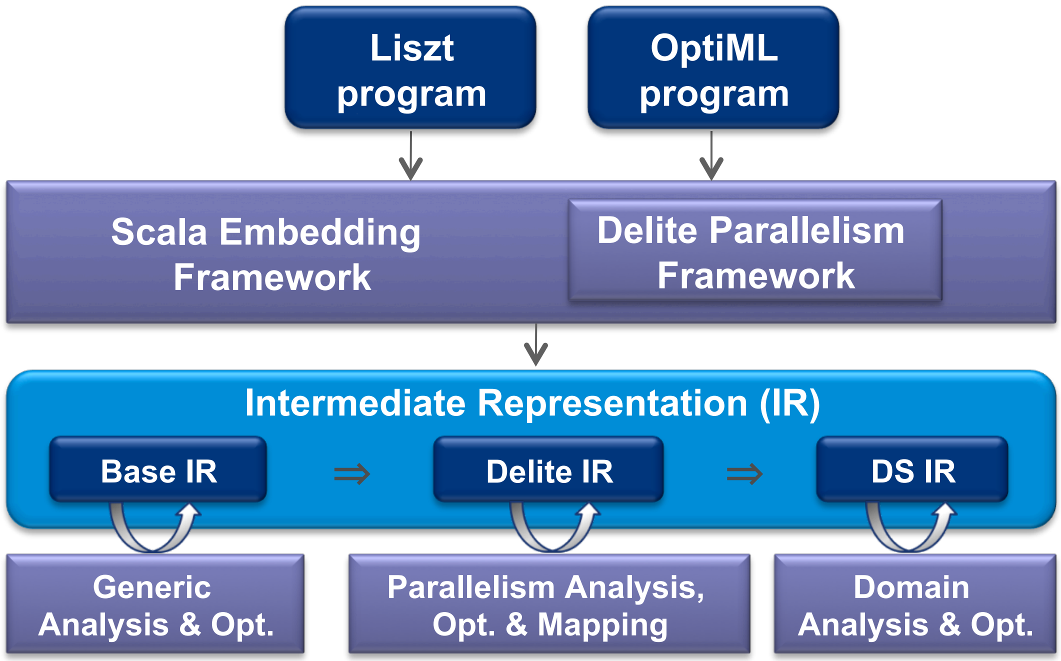
Delite: A Framework for DSL Parallelism

H. Chafi, A. Sujeeth, K. Brown, H. Lee



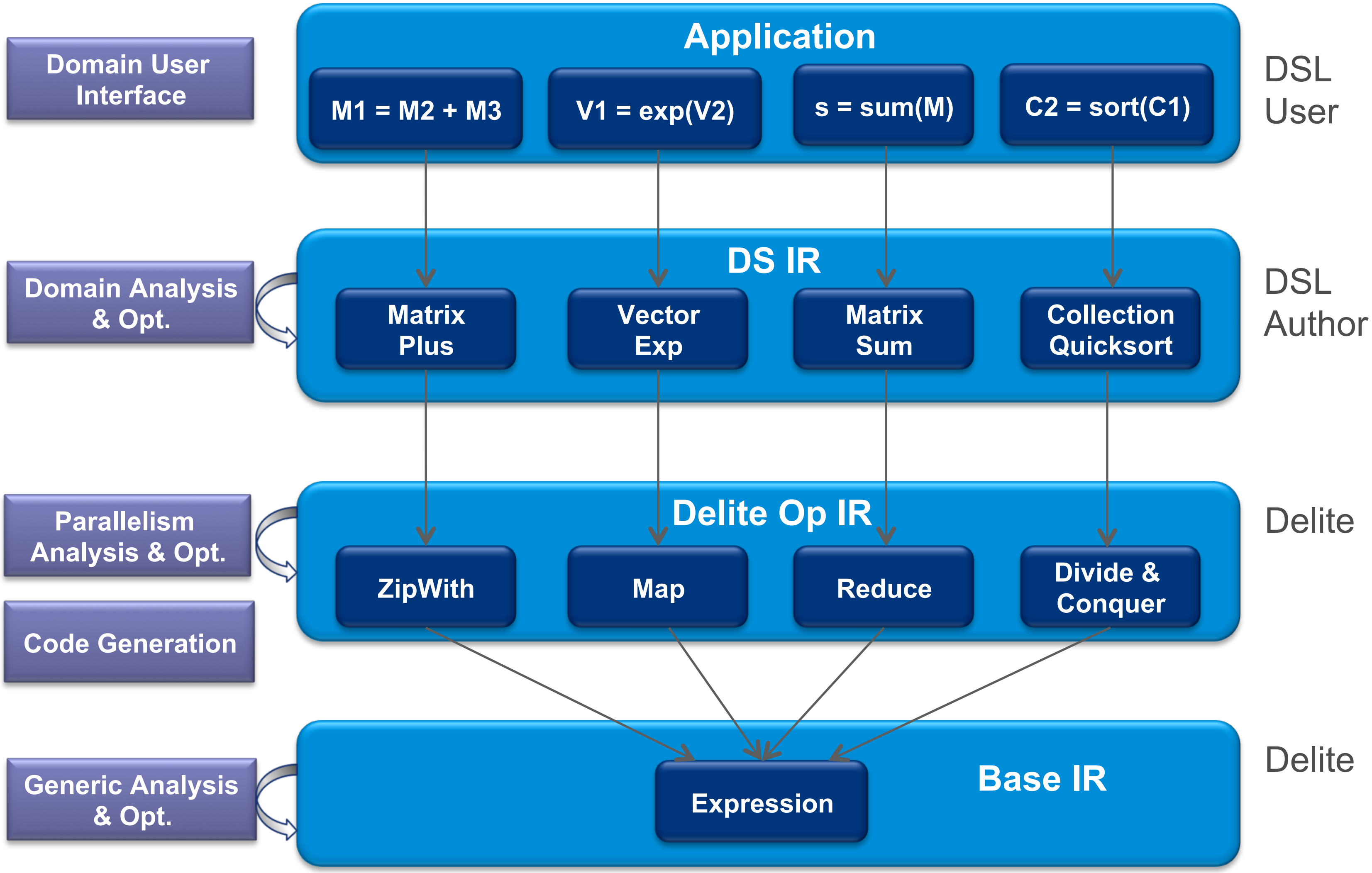
Need a framework to simplify development of DSL backends

Delite DSL Compiler



- 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 appropriate data parallel nodes for their operations
  - Now can do domain-specific analysis and opt.

The Delite Multiview IR



Generic Optimizations

- Common subexpression elimination
  - Global dictionary tracks what's been seen before
- Dead code elimination
  - All code is emitted due to dependencies on computing a required result
  - Dead code is never encountered in this process
- Constant folding
  - Constants are lifted into the IR lazily
  - Operations on constants are computed as program runs
- Code motion
  - Pull computation out of loops
  - Push computation into conditionals

DSL Optimizations

- Use domain-specific knowledge to make optimizations in a modular fashion
- Override IR node creation
  - Construct Optimized IR nodes if possible
  - A * B + A *C = A * (B + C) // Matrix A, B, C
  - 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

OptiML Linear Algebra Rewrites

- A straightforward translation of the Gaussian Discriminant Analysis (GDA) algorithm from the mathematical description produces the following code:

```
val sigma = sum(0,m) { i =>
  if (x.labels(i) == false) {
    ((x(i) - mu0).t) ** (x(i) - mu0)
  else
    ((x(i) - mu1).t) ** (x(i) - mu1)
  }
}
```

- A much more efficient implementation recognizes that

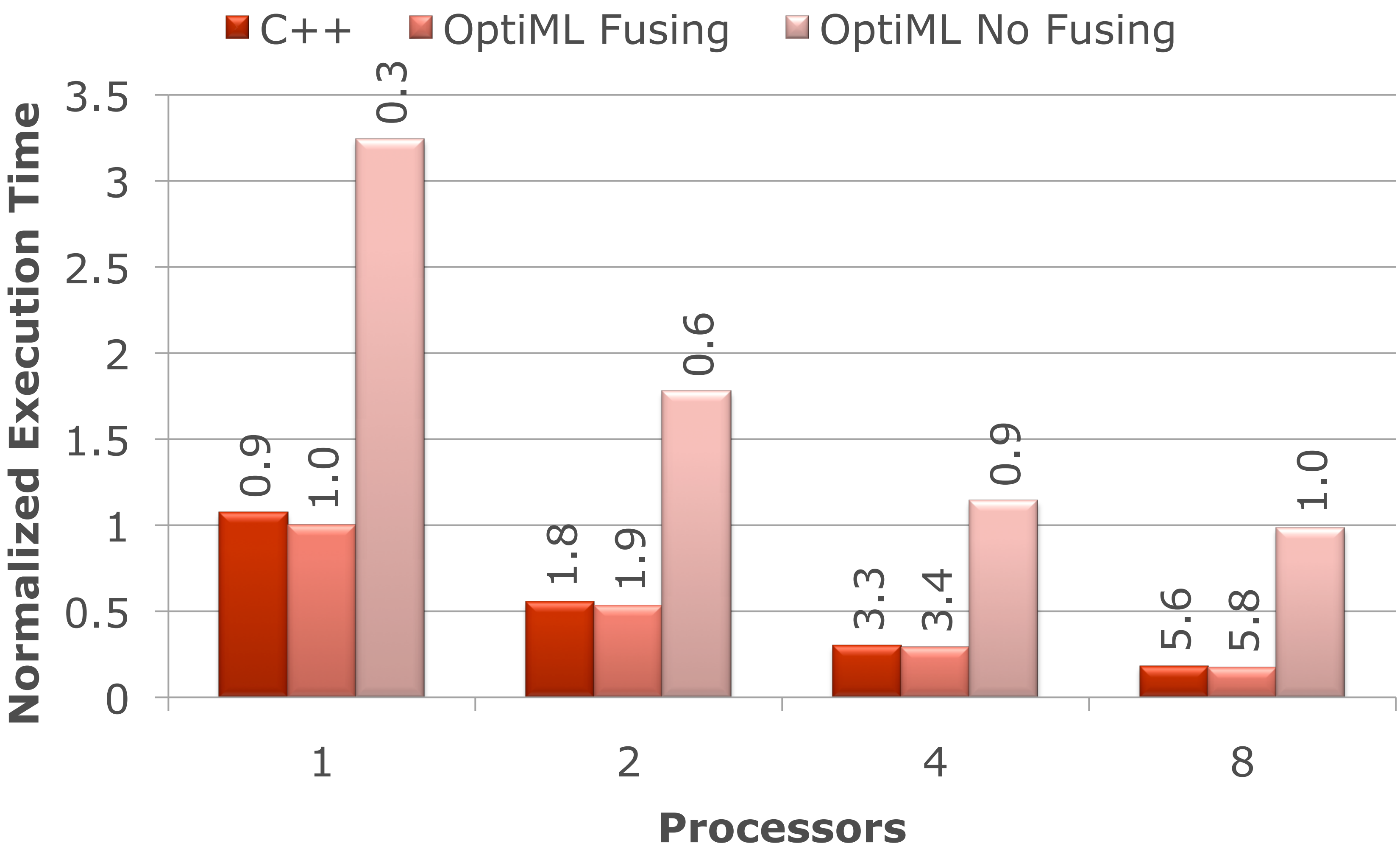
$$\sum_{i=0}^{n} \vec{x_i} * \vec{y_i} \rightarrow \sum_{i=0}^{n} X(:,i) * Y(i,:) = X * Y$$

- Transformed code was 20.4x faster with 1 thread and 48.3x faster with 8 threads.

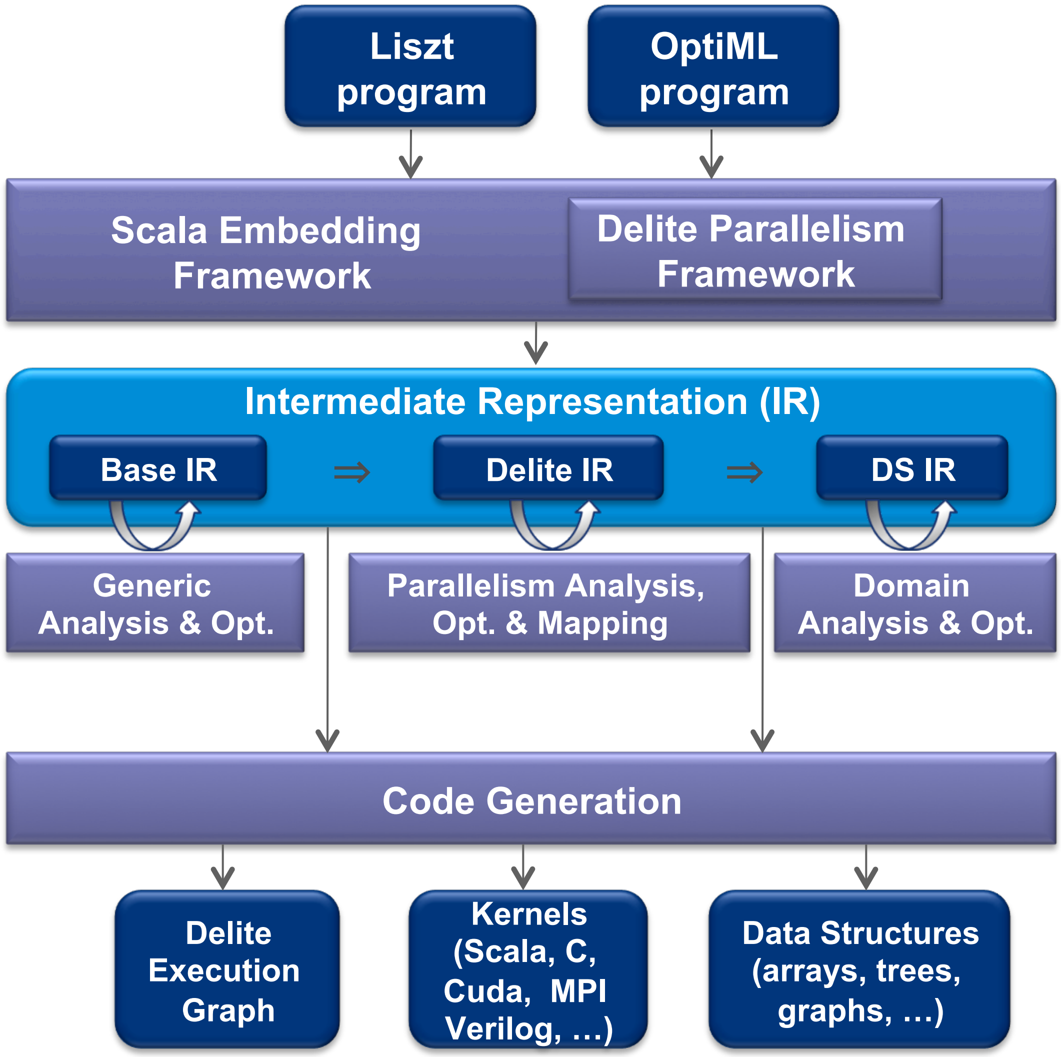
Delite Op Fusing

- Fuse Parallel Ops
- Reduces Op overhead
  - Op setup
  - Loop overhead
- Improves locality
  - Fused Op communication through registers

Benefits of Fusing



Delite DSL Compiler



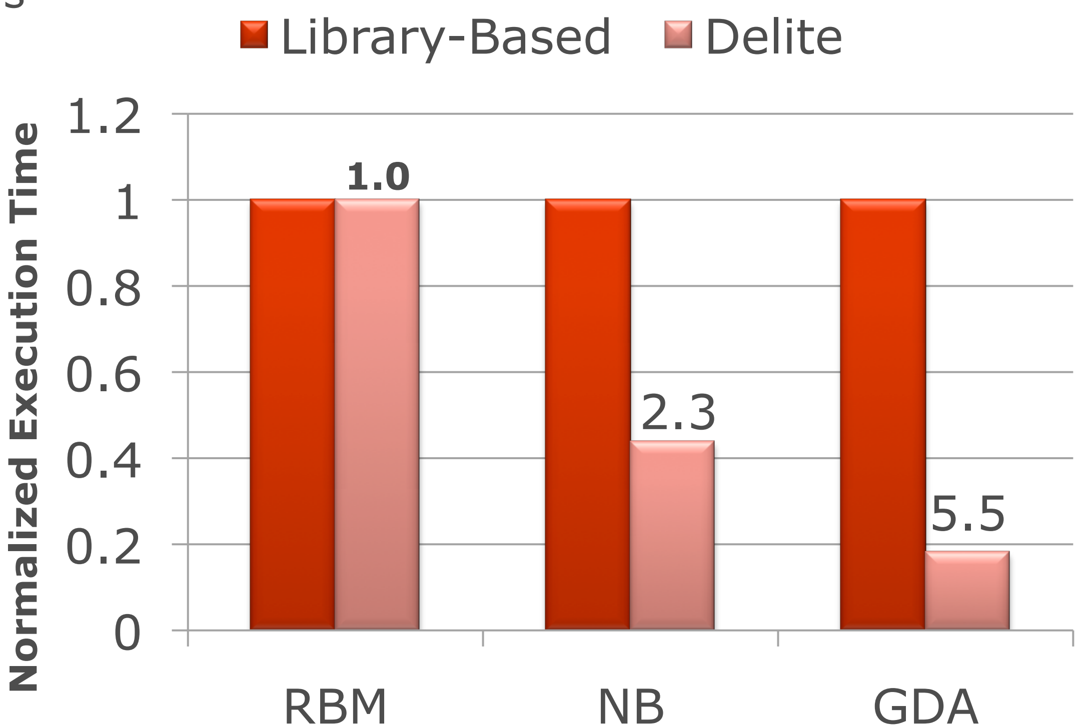
- Provide a common IR that can be extended while still benefitting from generic analysis and opt.
- Extend common IR and provide IR nodes that encode data parallel execution patterns
  - Now can do parallel optimizations and mapping
- DSL extends appropriate data parallel nodes for their operations
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

Delite Code Generator

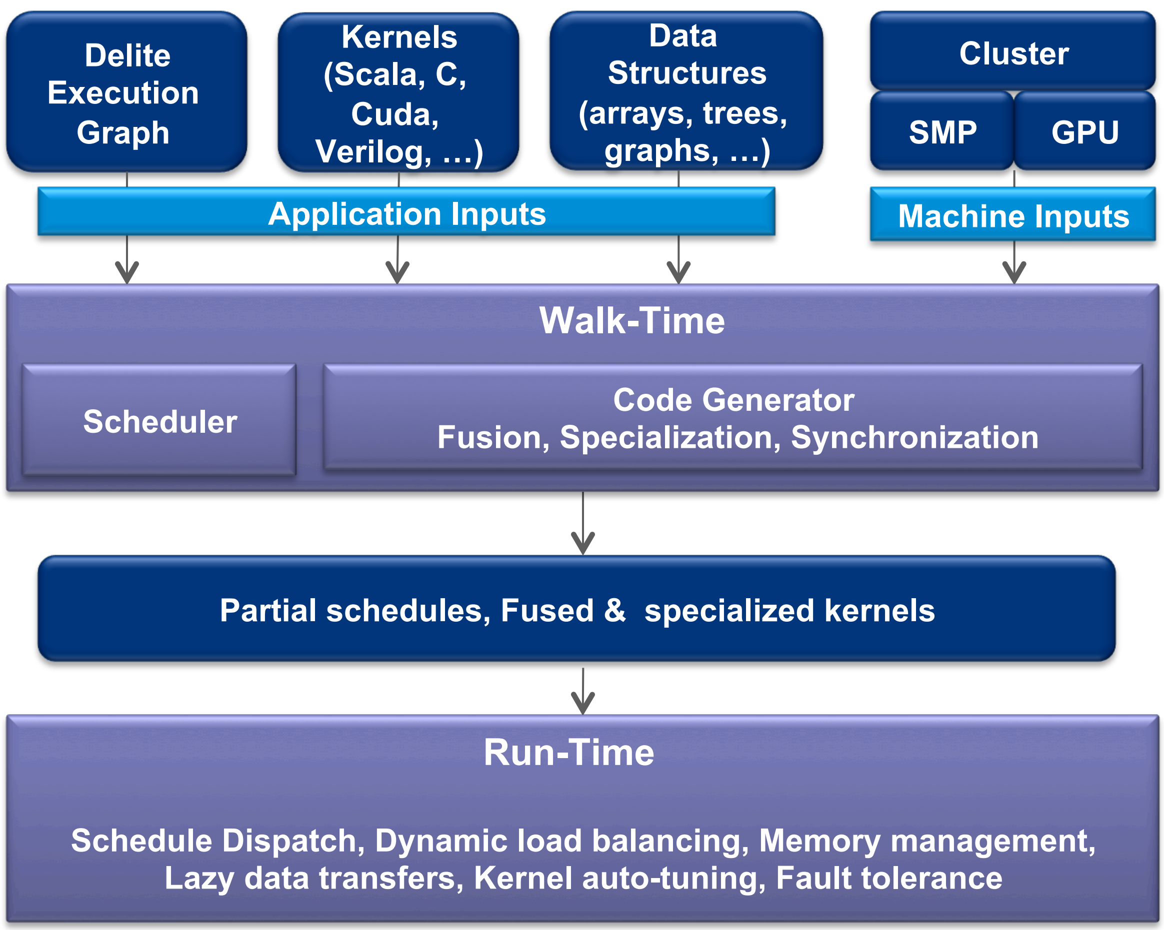
- Generates an execution graph with all DeliteOps and their dependencies
- Calls all registered code generators (Scala, Cuda, …) for each Op to create kernels
  - Only 1 generator (currently Scala) has to succeed
- Every Op at top-level of program is emitted as a kernel
  - Creates a file and object header then calls emitNode() on the Op
  - Nested calls to emitNode() result in implementation being inlined in current kernel

Cuda Code Generation

- With a library approach we can only launch pre-written kernels
- Code generation enables kernels containing user-defined functions and optimization opportunities
  - e.g., fuse operations into one kernel and keep intermediate results in registers



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

Specialization and the 4 Ps

PERVASIVE PARALLELISM LABORATORY

- Power
- Performance
- Productivity
- Portability

Application Specific Hardware

Domain Specific Languages

Conclusions

- DSLs have potential to solve the heterogeneous parallel programming problem
  - Don’t expose programmers to explicit parallelism
- Need to simplify the process of developing 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

Performance Results

- Machine
  - Two quad-core Nehalem 2.67 GHz processors
  - NVidia Tesla C2050 GPU

- Application Versions
  - OptiML + Delite
  - MATLAB
    - version 1: multi-core (parallelization using “parfor” construct and BLAS)
    - version 2: GPU
  - C++
    - used Armadillo linear algebra library for a sequential baseline
    - Algorithmically identical to OptiML version

Benchmark Applications

- 6 machine learning applications
  - Gaussian Discriminant Analysis (GDA)
    - Generative learning algorithm for probability distribution
  - Loopy Belief Propagation (LBP)
    - Graph based inference algorithm
  - Naïve Bayes (NB)
    - Supervised learning algorithm for classification
  - K-means Clustering (K-means)
    - Unsupervised learning algorithm for clustering
  - Support Vector Machine (SVM)
    - Optimal margin classifier using SMO algorithm
  - Restricted Boltzmann Machine (RBM)
    - Stochastic recurrent neural network