# A DOMAIN SPECIFIC APPROACH TO HETEROGENEOUS PARALLELISM

**Hassan Chafi**, Arvind Sujeeth, Kevin Brown, HyoukJoong Lee, Anand Atreya, Kunle Olukotun

Stanford University
Pervasive Parallelism Laboratory (PPL)

# Era of Power Limited Computing

- Mobile
  - Battery operated
  - Passively cooled
- Data center
  - Energy costs
  - Infrastructure costs



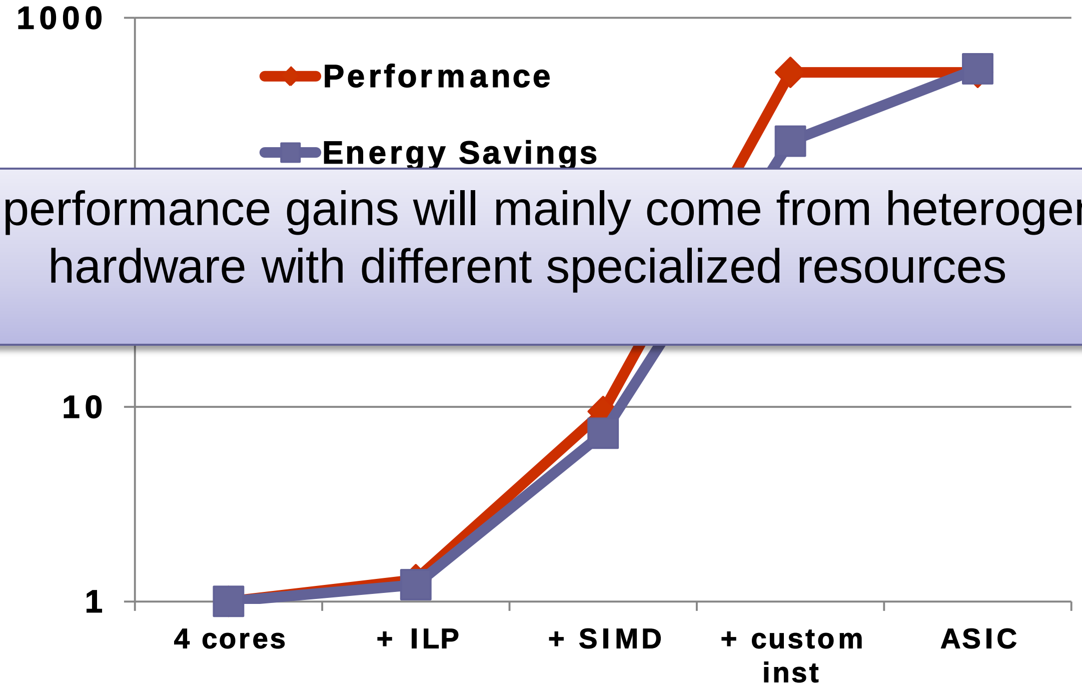
# Computing System Power

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

# Heterogeneous Hardware

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

- H.264 encode study

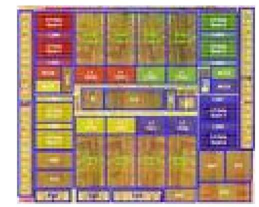



Future performance gains will mainly come from heterogeneous hardware with different specialized resources

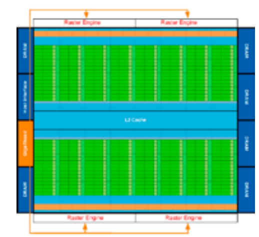
Source: Understanding Sources of Inefficiency in General-Purpose Chips (ISCA'10)

# Heterogeneous Parallel Architectures

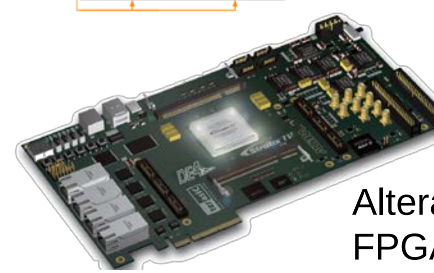
Driven by energy efficiency



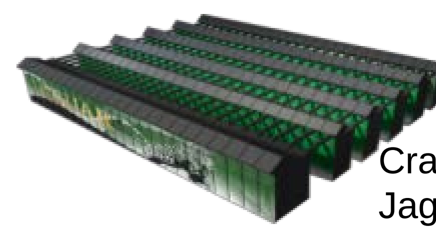
Sun T2



Nvidia Fermi



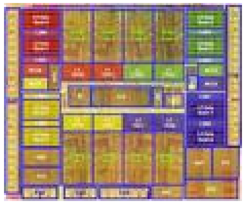
Altera FPGA



Cray Jaguar

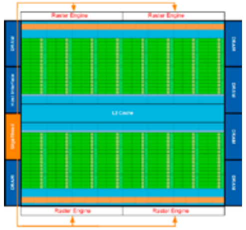
# Heterogeneous Parallel Programming

Pthreads
OpenMP



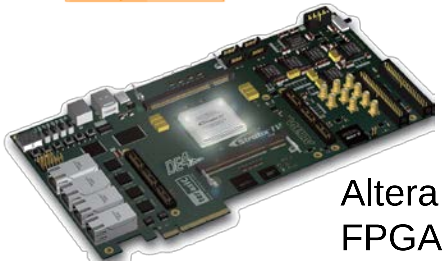
Sun
T2

CUDA
OpenCL



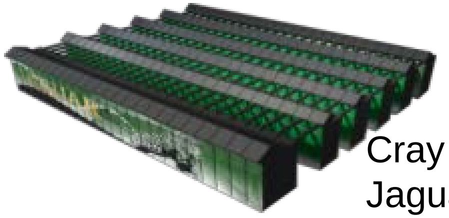
Nvidia
Fermi

Verilog
VHDL



Altera
FPGA

MPI



Cray
Jaguar

# Programmability Chasm

**Applications**

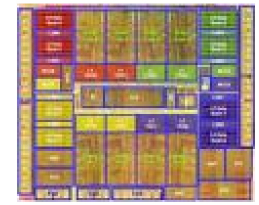
**Scientific Engineering**

**Virtual Worlds**

**Personal Robotics**

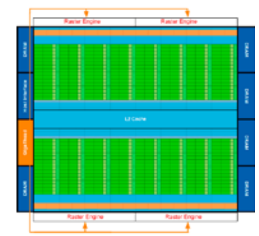
**Data informatics**

Pthreads
OpenMP



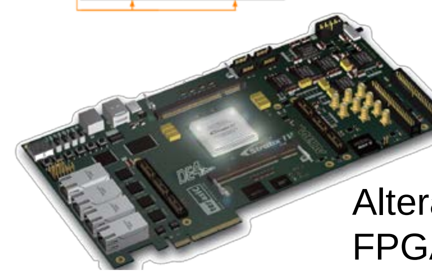
Sun T2

CUDA
OpenCL



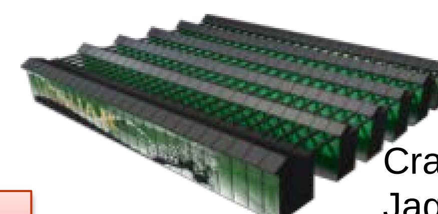
Nvidia Fermi

Verilog
VHDL



Altera FPGA

MPI



Cray Jaguar

**Too many different programming models**

IS IT POSSIBLE TO WRITE ONE PROGRAM

AND

RUN IT ON ALL THESE TARGETS?

# Programmability Chasm

**Applications**

**Scientific Engineering**

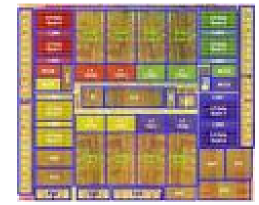
**Virtual Worlds**

**Personal Robotics**

**Data informatics**

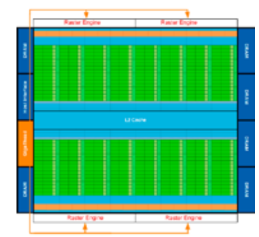
**Ideal Parallel Programming Language**

Pthreads
OpenMP



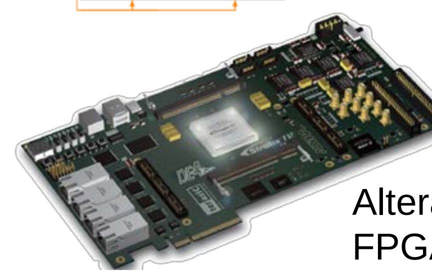
Sun
T2

CUDA
OpenCL



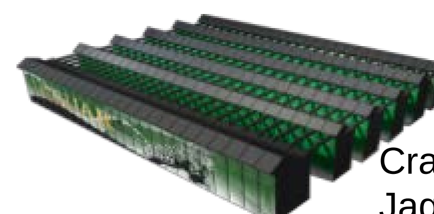
Nvidia
Fermi

Verilog
VHDL



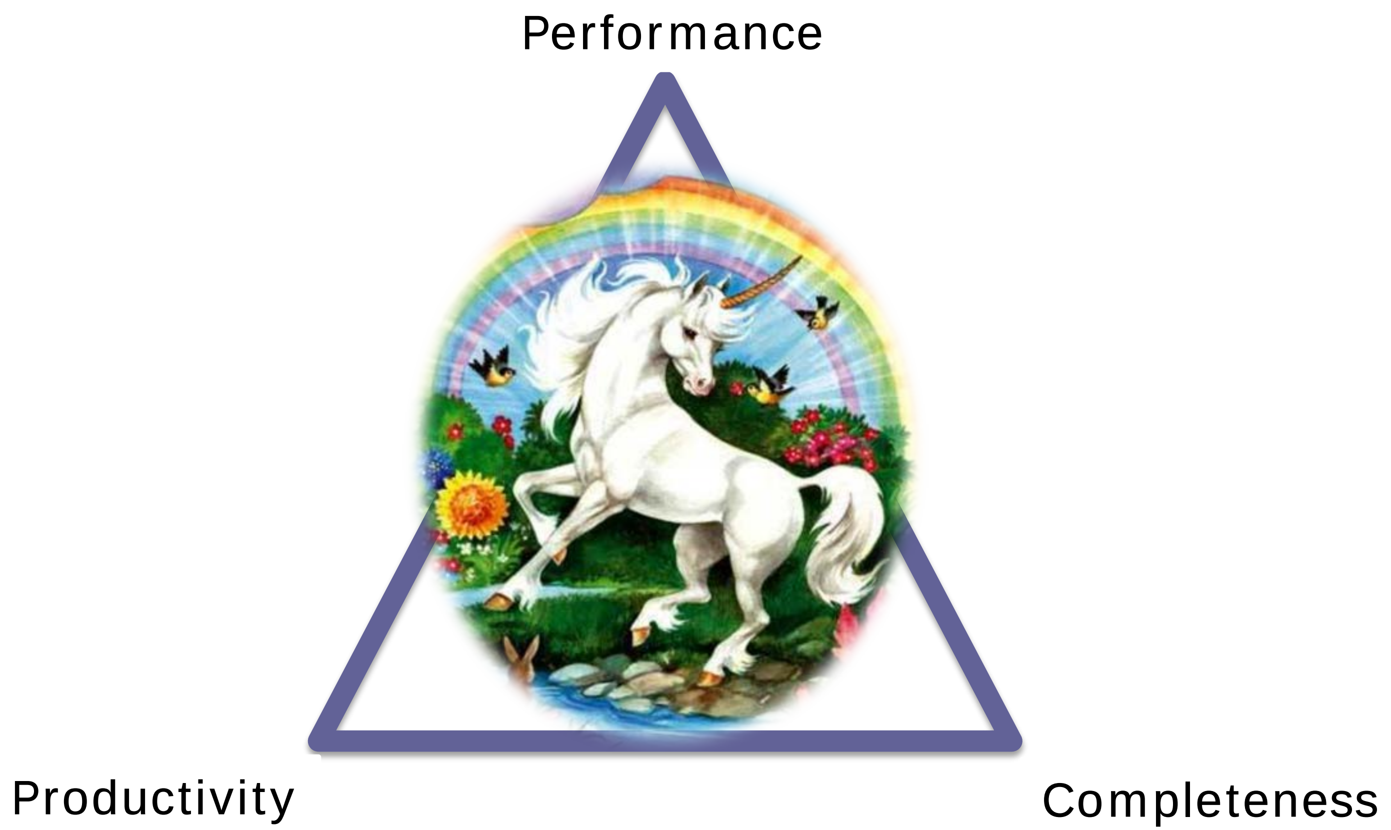
Altera
FPGA

MPI

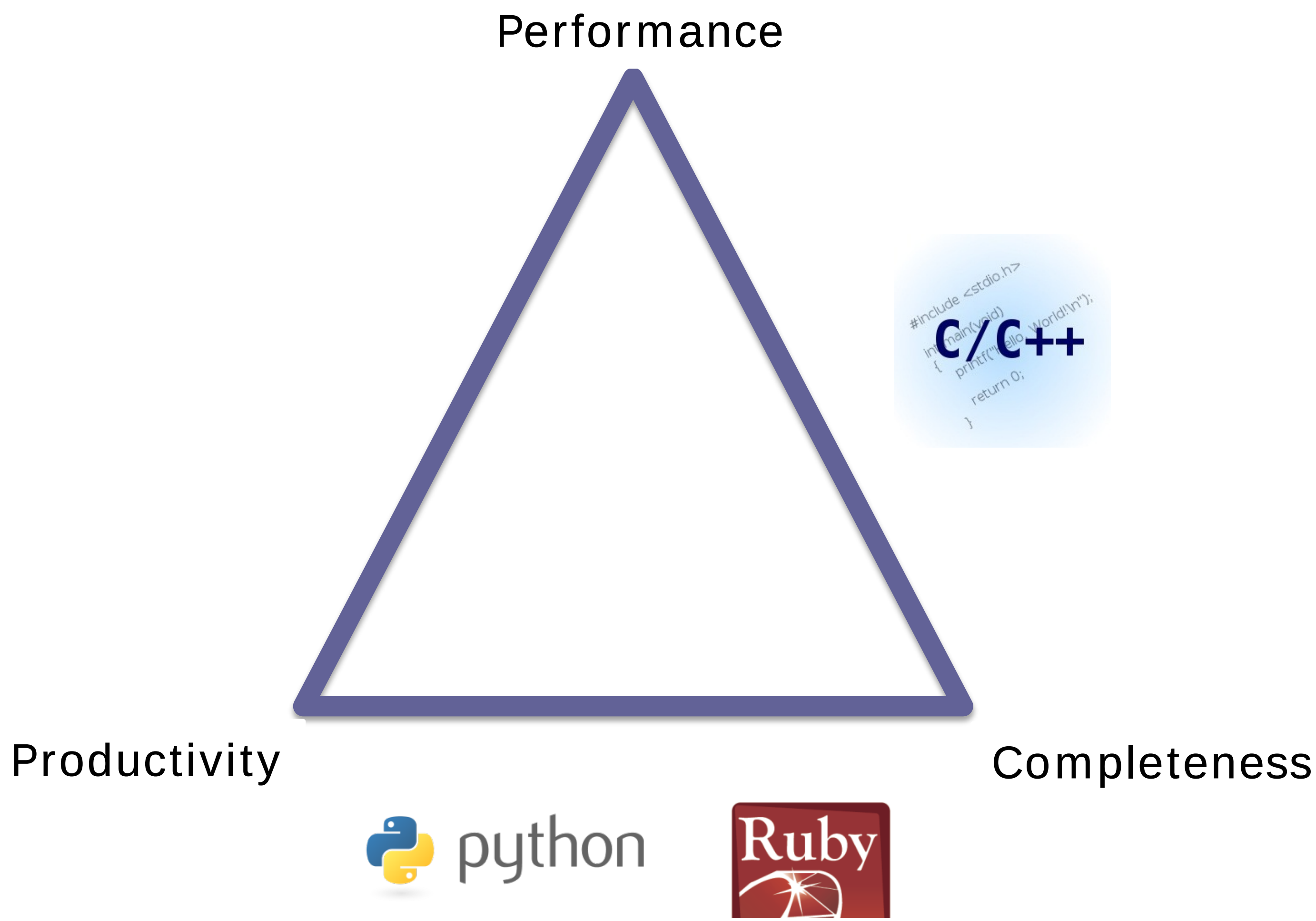


Cray
Jaguar

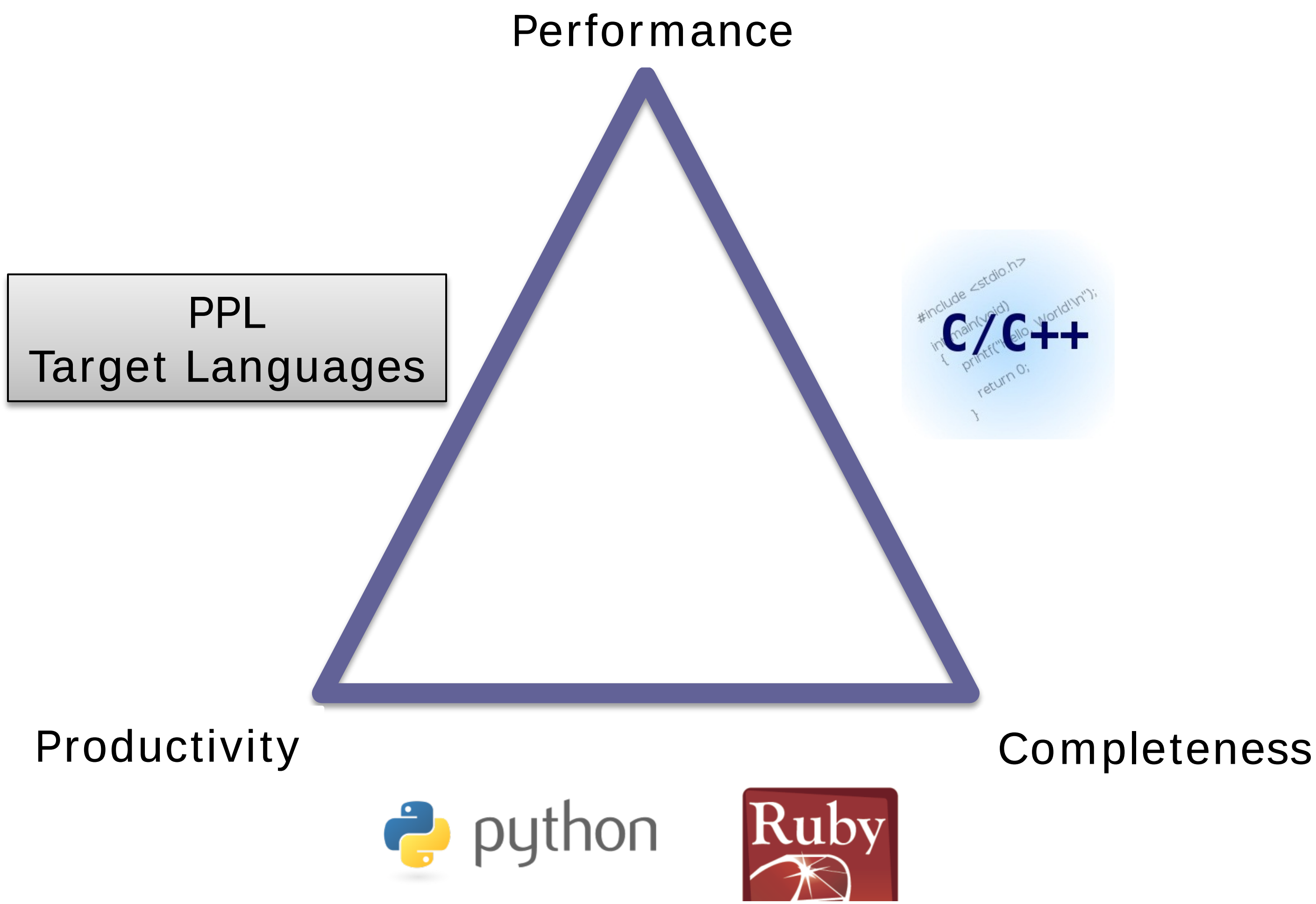
# The Ideal Parallel Programming Language



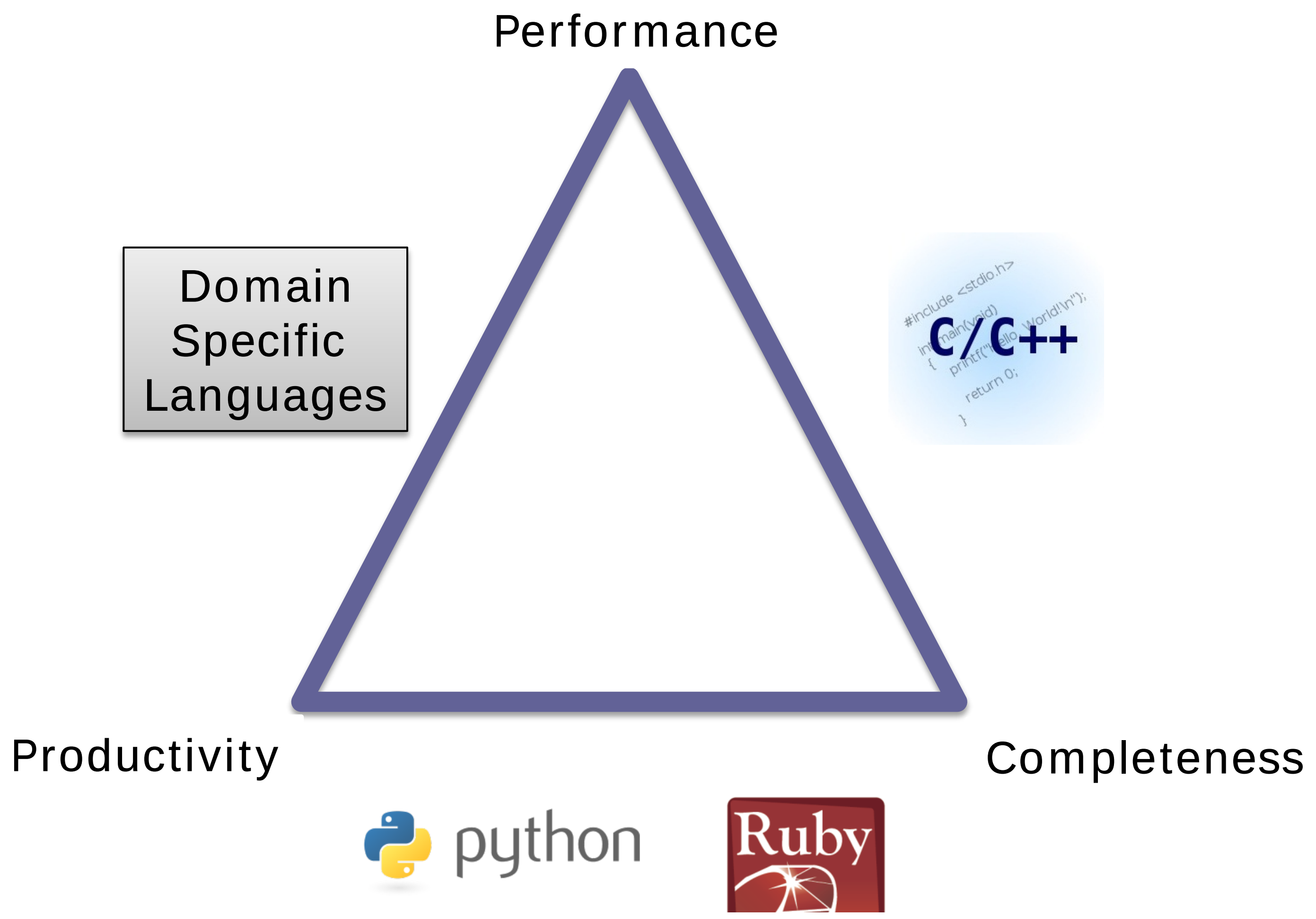
# Successful Languages



# Successful Languages



# Domain Specific  Languages



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

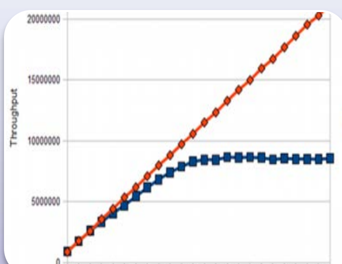
# Benefits of Using DSLs for Parallelism


## Productivity

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

## Performance

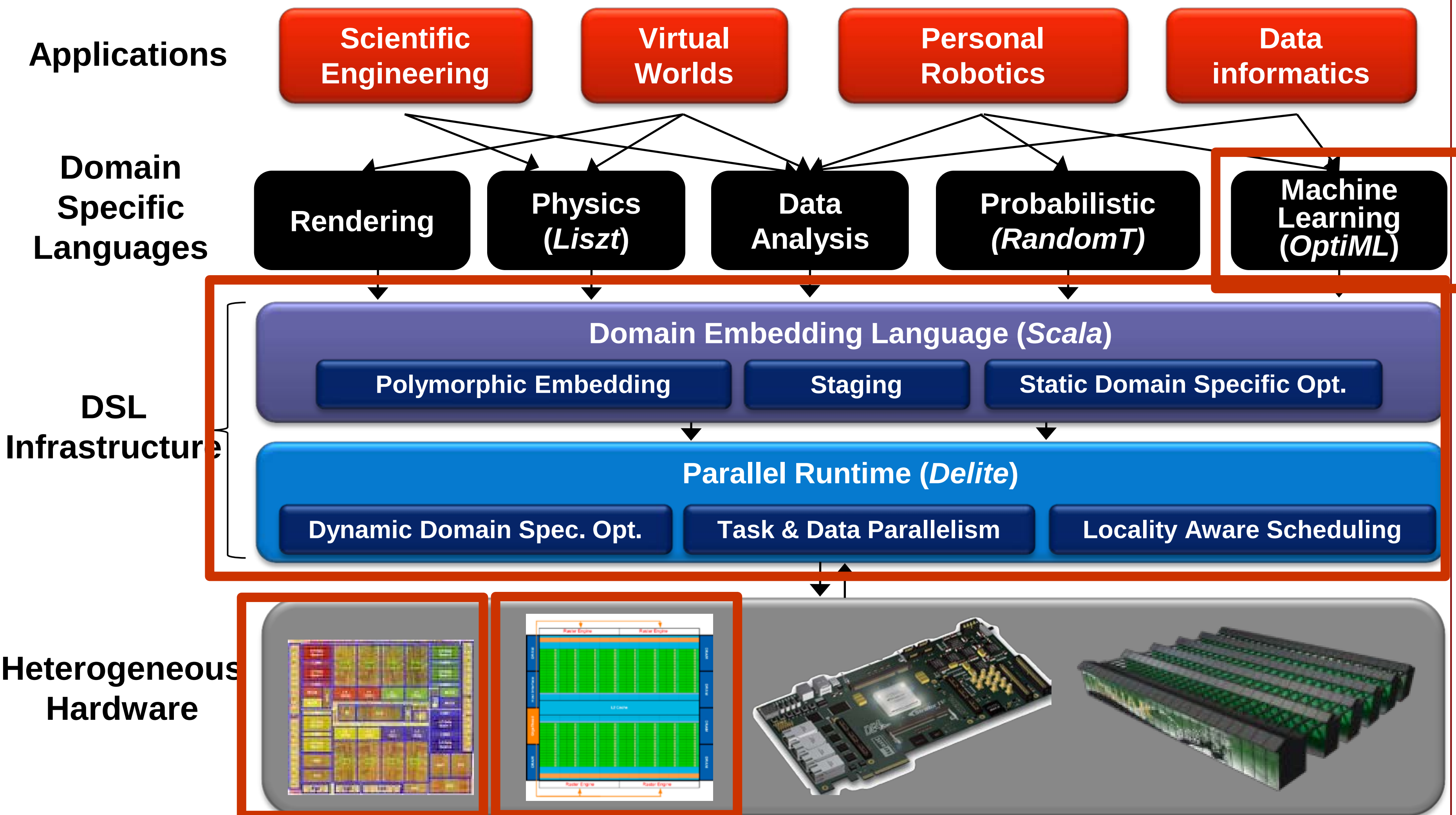
- Match generic parallel execution patterns to high level domain abstraction
- Restrict expressiveness to more easily and fully extract available parallelism
- Use domain knowledge for static/dynamic optimizations




## 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

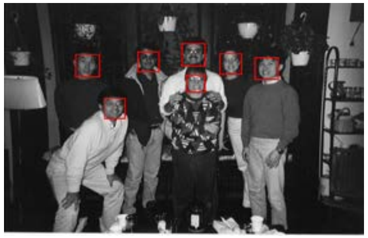
# Bridging the Programmability Chasm



# 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

reCAPTCHA™



- Characteristics of ML applications
  - Iterative algorithms on fixed structures
  - Large datasets with potential redundancy
  - Trade off between accuracy for performance
  - Large amount of data parallelism with varying granularity
  - Low arithmetic intensity

# OptiML: Motivation

- Raise the level of abstraction
  - Focus on algorithmic description, get parallel performance
- Use domain knowledge to identify coarse-grained parallelism
  - Identify parallel and sequential operations in the domain (e.g. ‘summations, batch gradient descent’)
- Single source => Multiple heterogeneous targets
  - Not possible with today’s MATLAB support
- Domain specific optimizations
  - Optimize data layout and operations using domain-specific semantics
- A driving example
  - Flesh out issues with the common framework, embedding etc.

# OptiML: Overview

- 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]
  - 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

# MATLAB implementation

- `parfor` is nice, but not always best
  - MATLAB uses heavy-weight MPI processes under the hood
  - Precludes vectorization, a common practice for best performance
  - GPU code requires different constructs
- The application developer must choose an implementation, and these details are all over the code

```
ind = sort(randsample(1:size(data,2),length(min_dist)));
data_tmp = data(:,ind);
all_dist = zeros(length(ind),size(data,2));
parfor i=1:size(data,2)
    all_dist(:,i) =
sum(abs(repmat(data(:,i),1,size(data_tmp,2)) -
data_tmp),1)';
end
all_dist(all_dist==0)=max(max(all_dist));
```

# Domain Specific Optimizations

- Relaxed dependencies
  - Iterative algorithms with inter-loop dependencies prohibit task parallelism
  - Dependencies can be relaxed at the cost of a marginal loss in accuracy
- Best effort computations
  - Some computations can be dropped and still generate acceptable results
  - Provide data structures with “best effort” semantics, along with policies that can be chosen by DSL users

S. Chakradhar, A. Raghunathan, and J. Meng. **Best-effort parallel execution framework for recognition and mining applications.** IPDPS’09

# Delite: a framework to help build parallel DSLs

- Building DSLs is hard
  - Building parallel DSLs is harder
  - For the DSL approach to parallelism to work, we need many DSLs
- Delite provides a common infrastructure that can be tailored to a DSL's needs
  - An interface for mapping domain operations to composable parallel patterns
  - Provides re-usable components: GPU manager, heterogeneous code generation, etc.

# Composable parallel patterns

- Delite view of a DSL: a collection of data(DeliteDSLTypes) and operations (OPs)
- Delite supports OP APIs that express parallel execution patterns
  - DeliteOP_Map, DeliteOP_Zipwith, DeliteOP_Reduce, etc.
  - Planning to add more specialized ops
  - DSL author maps each DSL operation to one of the patterns (can be difficult)
- OPs record their dependencies (both mutable and immutable)

# Example code for Delite OP

```
case class OP_+[A](val collA: Matrix[A],
                   val collB: Matrix[A],
                   val out: Matrix[A])
                  (implicit ops: ArithOps[A])
 extends DeliteOP_ZipWith2[A,A,A,Matrix]{
 def func = (a,b) => ops.+(a,b)
}
```

Dependencies

Execution pattern

Interface for this pattern

# Delite: a dynamic parallel runtime

- Executes a task graph on parallel, heterogeneous hardware
  - (paper) performs dynamic scheduling decisions
  - (soon) both static and dynamic scheduling
- Integrates task and data parallelism in a single environment
  - Task parallelism at the DSL operation granularity
  - Data parallelism by data decomposition of a single operation into multiple tasks
- Provides efficient implementations of the execution patterns

# Delite Execution Flow

Application

```
def example(a: Matrix[Int],
            b: Matrix[Int],
            c: Matrix[Int],
            d: Matrix[Int]) =
{

  val ab = a * b
  val cd = c * d
  return ab + cd

}
```

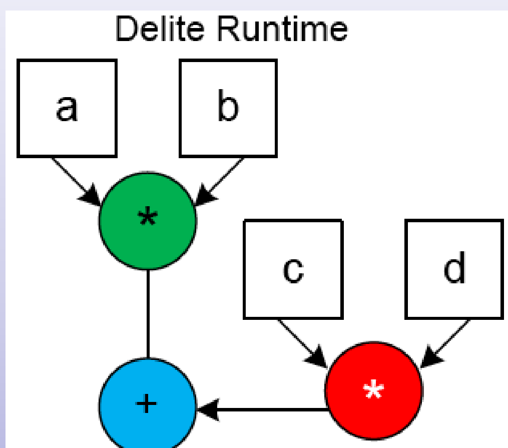
**Calls Matrix DSL methods**

Matrix DSL

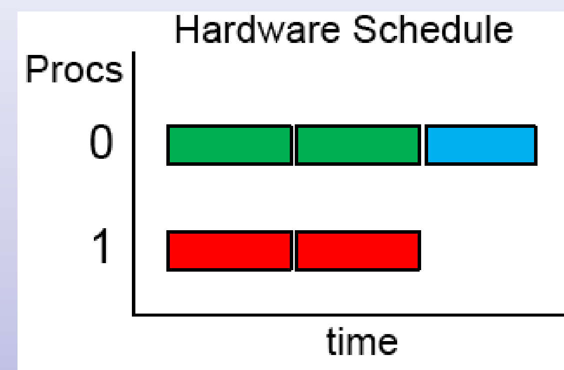
```
def *(m: Matrix[Int]) =
  delite.defer(OP_mult(this, m))

def +(m: Matrix[Int]) =
  delite.defer(OP_plus(this, m))
```

**DSL defers OP execution to Delite R.T.**




**Delite applies generic & domain transformations and generates mapping**



# Using GPUs with MATLAB

- MATLAB Parallel Computing Toolbox

```
sigma = gpuArray(zeros(n,n));
for i=1:m
    if (y(i) == 0)
        sigma = sigma + gpuArray(x(i,:)-mu0)'*gpuArray(x(i,:-mu0);
    else
        sigma = sigma + gpuArray(x(i,:)-mu1)'*gpuArray(x(i,:-mu1);
    end
end
```

- AccelerEyes Jacket

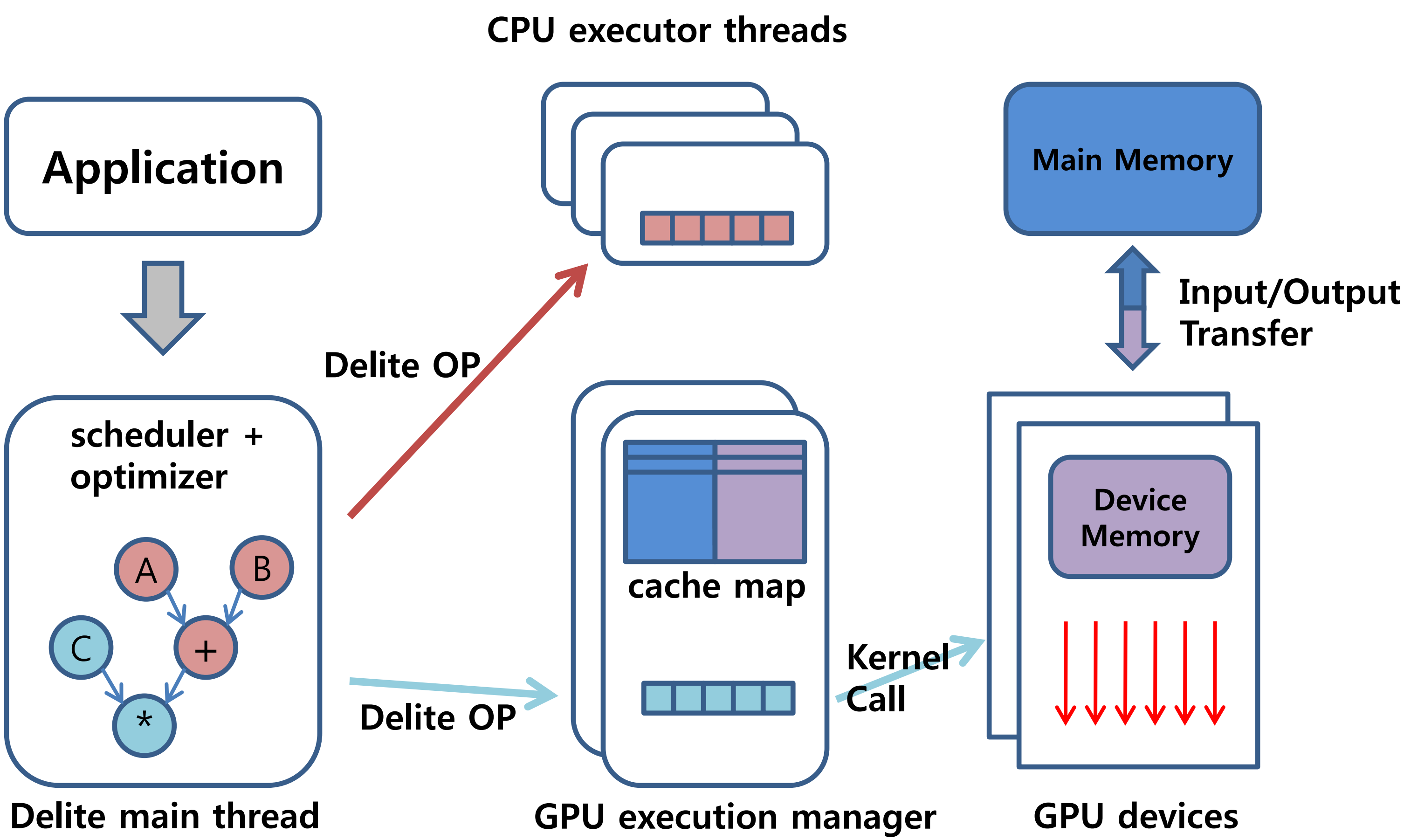
```
sigma = gzeros(n,n);
y = gdouble(y);
x = gdouble(x);
for i=1:m
    if (y(i) == 0)
        sigma = sigma + (x(i,:)-mu0)'* (x(i,:-mu0);
    else
        sigma = sigma + (x(i,:)-mu1)'* (x(i,:-mu1);
    end
end
```

# Using GPUs with Delite

- No change in the application source code
  - Same application code runs on any kind of heterogeneous system
    - Good for portability
  - Runtime (not the DSL user) dynamically determines whether to ship the operation to GPU or not
    - Good for productivity

- Performance optimizations under the hood
  - Memory transfer between CPU and GPU
  - On-chip device memory utilization
  - Concurrent kernel executions

# Optimized GPU Runtime Diagram



# GPU Code generation

- DSL OPs require implementations of GPU kernels
  - (paper) DSL provides optimized implementations
    - Libraries (CUBLAS, CUFFT, etc) can be used
  - (now) GPU kernels generated from Scala kernels
    - Write once, run anywhere, libraries can still be used

- What about DSL constructs with anonymous functions?
  - The GPU task is given by DSL user, not DSL writer
  - Impossible to pre-generate kernels
  - Solution: Automatically generate corresponding GPU kernels at compile time

# GPU Code Generation Flow

```
val a = Vector[Double](n)
val b = 3.28
val c = (0::n) { i =>  i * b * a(i) }
```

Original Code

Scala compiler plugin / embedding
(AST manipulation)

```
__global__ kernel0(double *input, double *output, int length, double *a, double b) {
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if(i < length)
        output[i] = input[i] * b * a[input[i]];
}
```

Generated CUDA Code

+

```
val a = Vector[Double](n)
val b = 3.28
val c = (0::n) { DeliteGPUFunc( {i =>  i * b * a(i)}, 0, List(a,b) ) }
```

Transformed Code

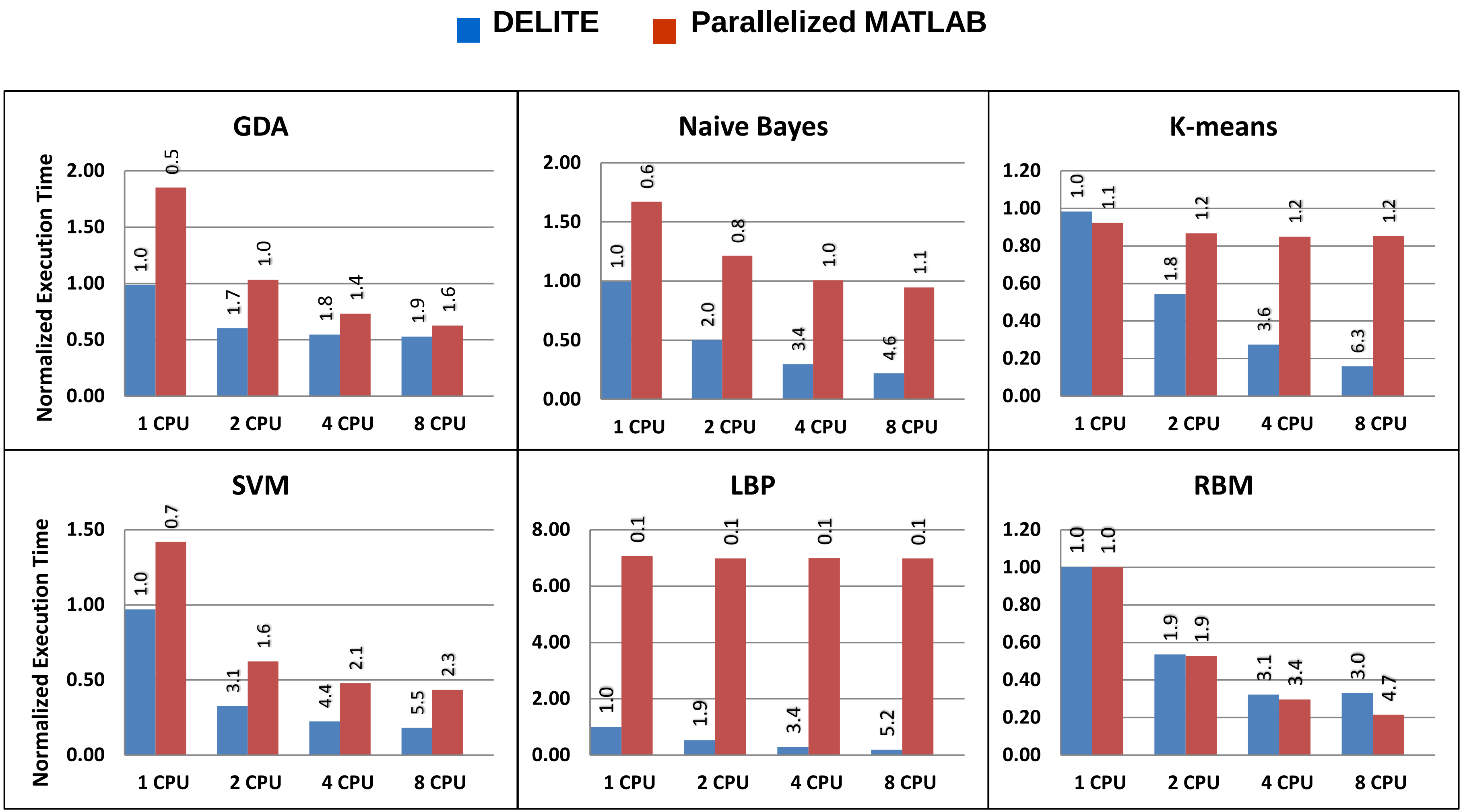
# Experimental Setup

- 4 Different implementations
  - OptiML+Delite
  - MATLAB (Original, GPU, Jacket)
- System:
  - Intel Nehalem
  - 2 sockets, 8 cores, 16 threads
  - 24 GB DRAM
  - NVIDIA GTX 275 GPU

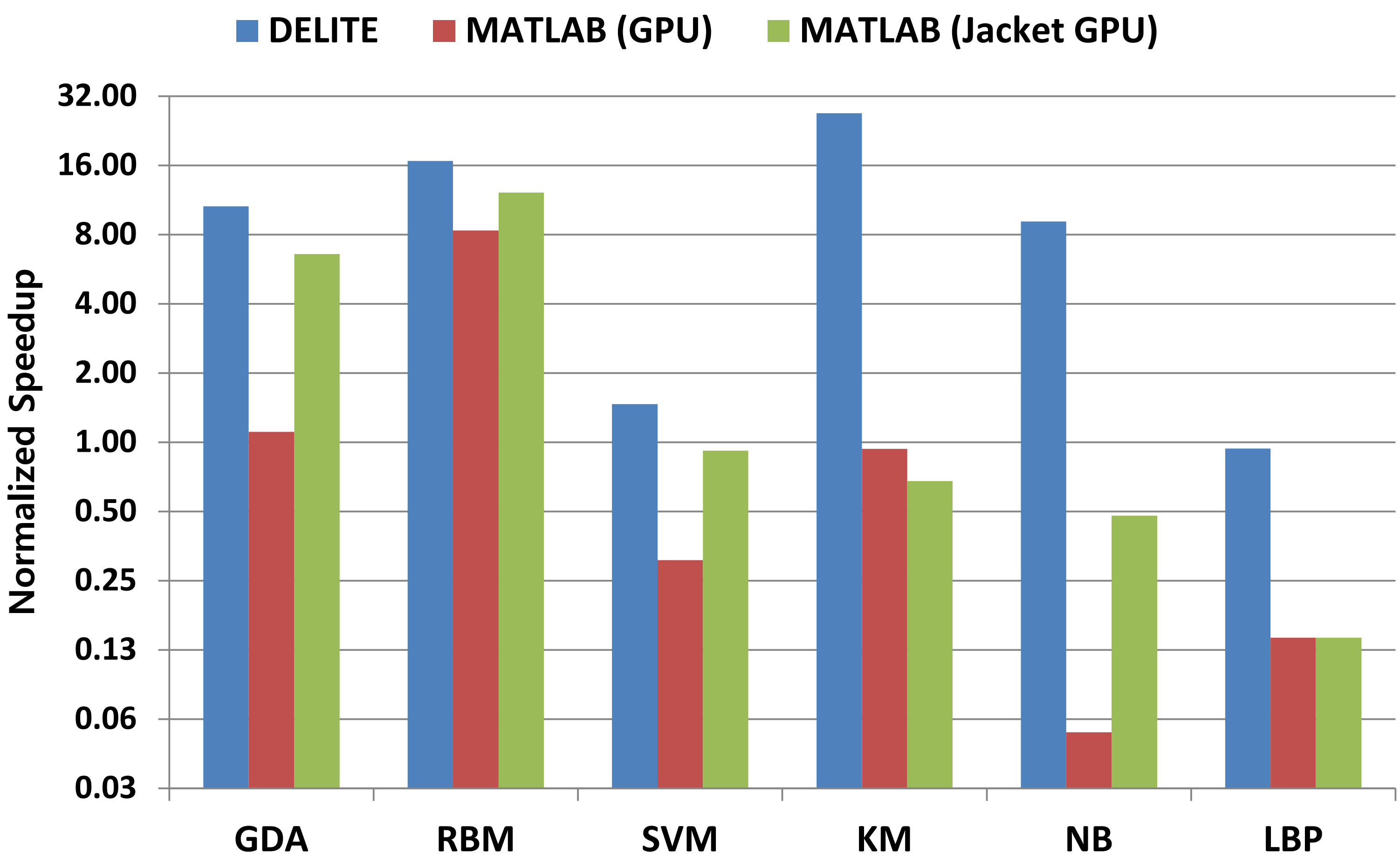
# Benchmark Applications

- 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

# Performance Study (CPU)

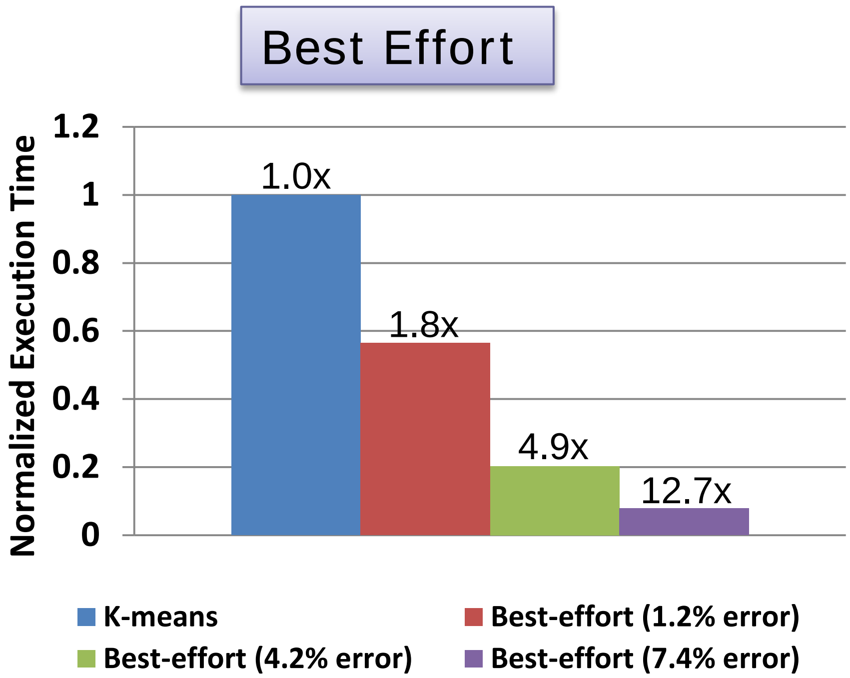


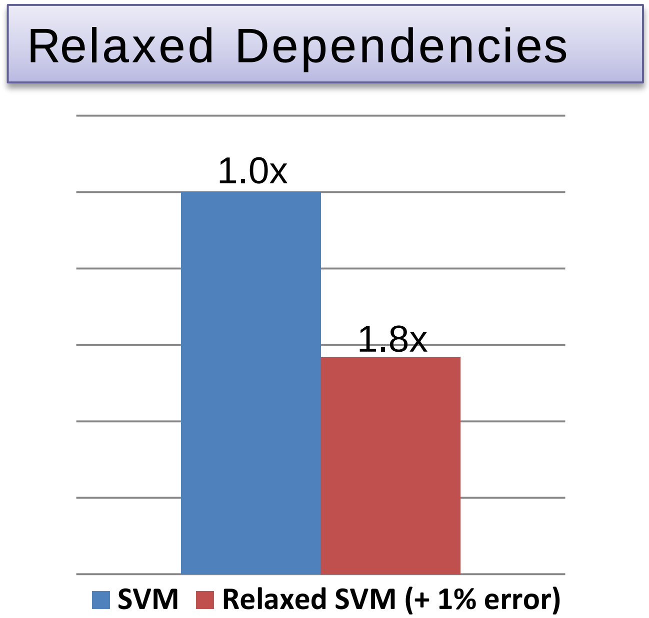
# Performance Study (GPU)




Speedup relative to single core execution time on Nehalem system

# Domain Specific Optimizations




Speedup relative to 8 core execution time on Nehalem system

# Conclusion

- Using Domain Specific Languages (DSLs) is a potential solution for heterogeneous parallelism
  - OptiML is a proof-of-concept DSL for ML
    - Productive, portable, performant
  - Delite is a framework for building DSLs and a parallel runtime
    - Simplifies developing implicitly parallel DSLs
    - Maps DSL to heterogeneous devices
    - Performs GPU specific optimizations and automatic code generation
  - Experimental results show that OptiML+Delite outperforms various MATLAB implementations