Simplifying Parallel Programming with Domain Specific Languages

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

Stanford University
Pervasive Parallelism Laboratory (PPL)

GPU Technology Conference 2010

Era of Power Limited Computing

- Mobile
  - Battery operated
  - Passively cooled

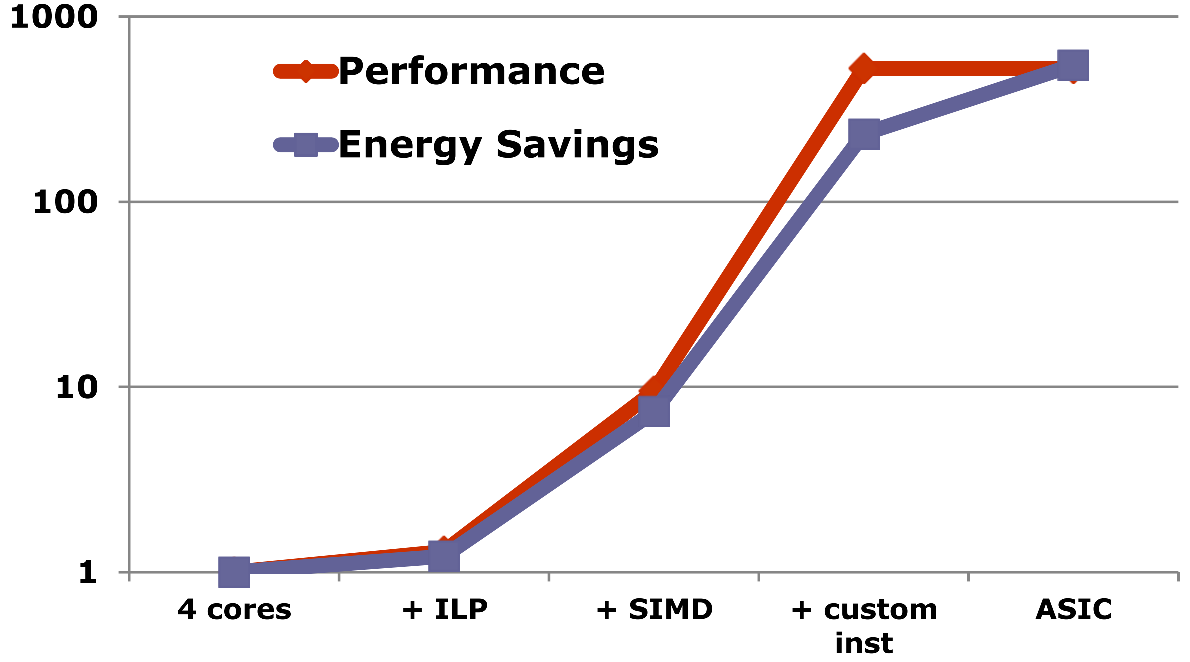
- Data center
  - Energy costs
  - Infrastructure costs

Computing System Power

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

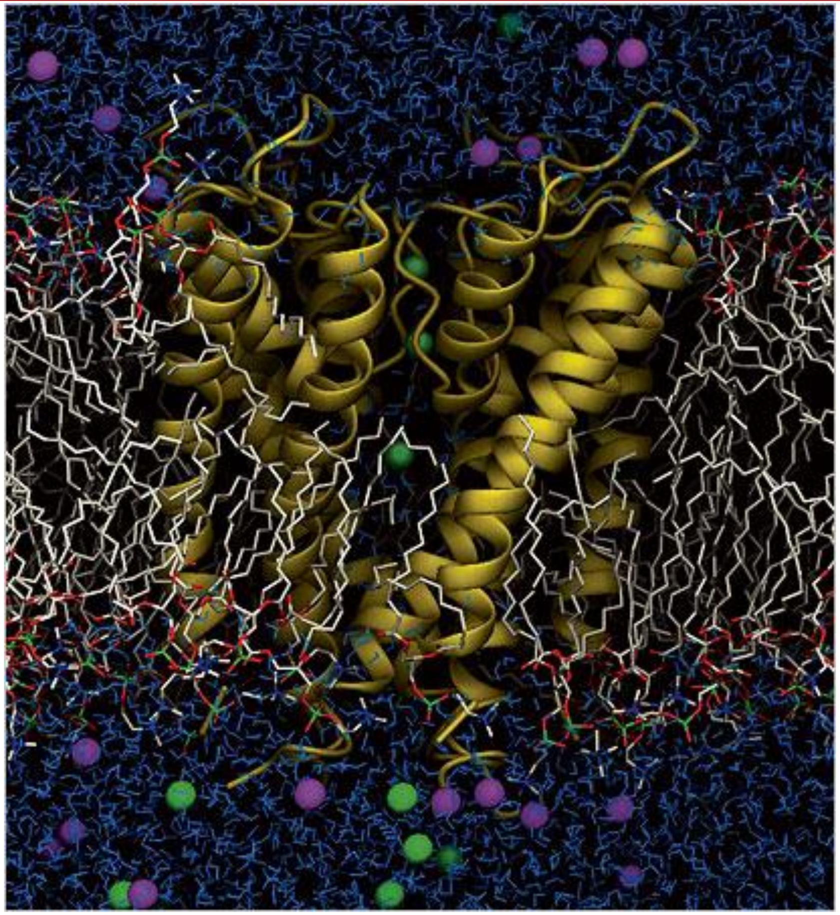
Heterogeneous Hardware

- 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)

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 iP{ad|hone}

PERVASIVE PARALLELISM LABORATORY

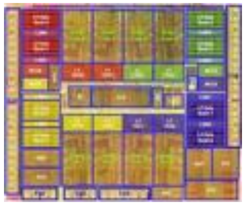
Contains CPU and GPU and …

Heterogeneous Parallel Computing

- Uniprocessor
  - Sequential programming
  - **C**

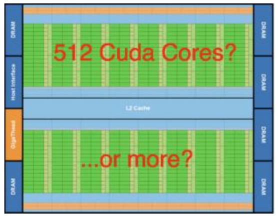
Intel Pentium 4

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



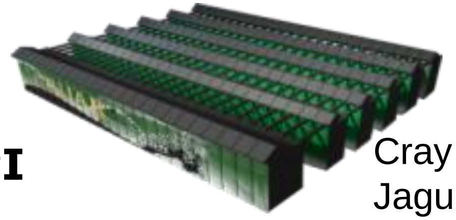
Sun T2

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



Nvidia Fermi

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



Cray Jaguar

Multiple incompatible programming models

IS IT POSSIBLE TO WRITE ONE PROGRAM

AND

RUN IT ON ALL THESE MACHINES?

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
    - OpenGL, MATLAB, SQL, VHDL, ..
- Benefit of using DSLs for parallelism
  - Productivity
    - Shield average programmers from the difficulty of parallel programming
  - 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

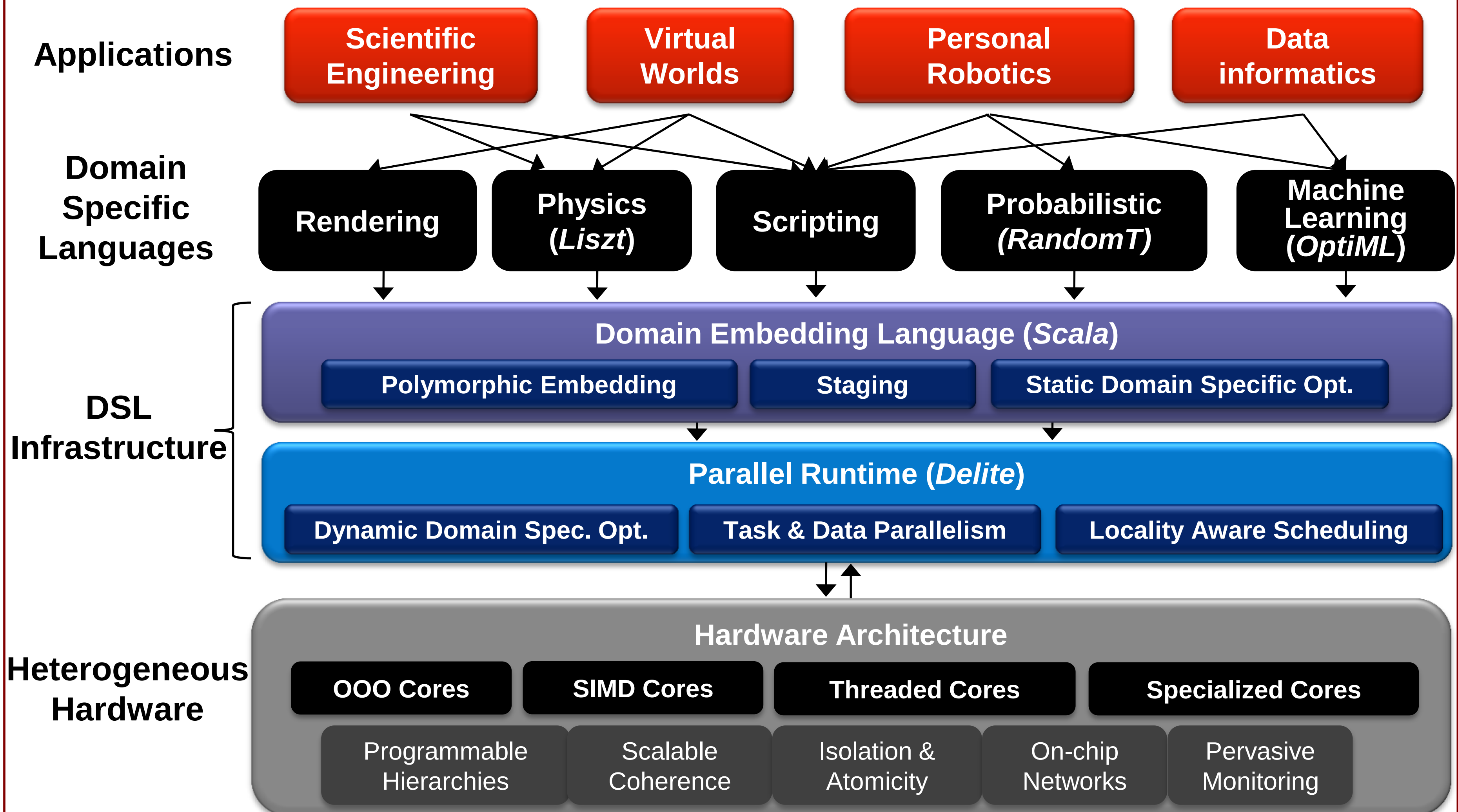
PPL Goals and Organization

- Goal: the parallel computing platform for the masses
  - Parallel applications without parallel programming

- PPL is a collaboration of
  - Leading Stanford researchers across multiple domains
    - Applications, languages, software systems, architecture
  - Leading companies in computer systems and software
    - NVIDIA, Oracle(Sun), AMD, IBM, Intel, NEC, HP

- PPL is open
  - Any company can join; all results in the public domain

The PPL Vision



Outline

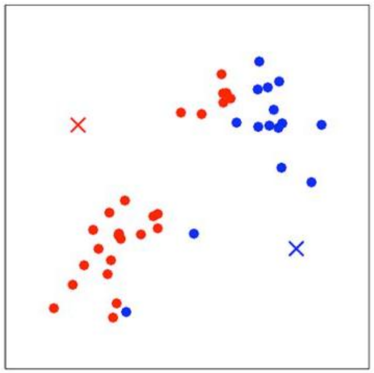
- Introduction
  - Using DSL for parallel programming
- OptiML
  - An example DSL for machine learning
- Delite
  - Runtime and framework for DSL approach
- Delite with GPU
  - Optimizations and automatic code generation
- Experimental Results
- Conclusion

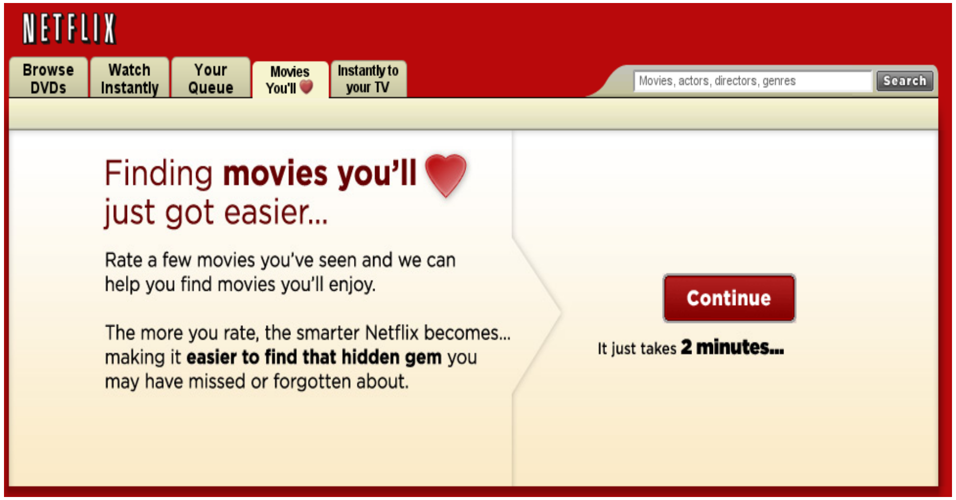
Machine Learning

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

Machine Learning Examples

PERVASIVE PARALLELISM LABORATORY

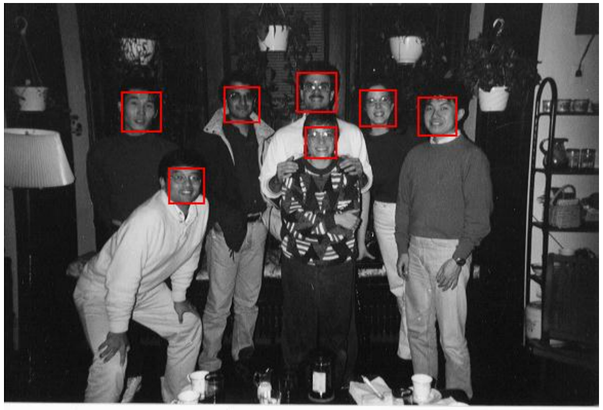




reCAPTCHA™

Google translate

Report Spam



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. ‘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
  - Provide an easy syntax for operations
  - 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, ..
    - Encode semantic information
- Implicitly parallel control structures
  - Sum{…}, (0::end) {…}
  - Allow anonymous functions to be passed as arguments of the control structures

Example OptiML / MATLAB code (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)
    }
}
```

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

OptiML vs. MATLAB

OptiML

- Statically typed
- Implicit parallelization
- Automatic GPU data management via run-time support
- Inherits Scala features and tool-chain
  - Still experimenting with: "what, if any, Scala features do we want to disallow, and how should we do that?"

MATLAB

- Dynamically typed
- Applications must explicitly choose between vectorization or parallelization
- Explicit GPU data management
- Widely used, efficient

Dynamic 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
  - Relaxation percentage is run-time configurable
- Best effort computations
  - Some computations can be dropped and still generates acceptable results
  - Provide data structures with "best effort" semantics, along with policies that can be chosen by DSL users

Potential Static Optimizations

- Efficient data representation
  - Same abstract data types can have multiple underlying optimized implementations
  - Matrix[Double] can be implemented as a dense matrix or a sparse matrix

- Transparent compression
  - Use knowledge of ML data types (image, video, audio, etc) to automatically insert efficient compression routines before transferring data across address spaces

Outline

- Introduction
  - Using DSL for parallel programming
- OptiML
  - An example DSL for machine learning
- Delite
  - Runtime and framework for DSL approach
- Delite with GPU
  - Optimizations and automatic code generation
- Experimental Results
- Conclusion

Delite: A DSL Design Framework

- Delite provides a common infrastructure for exposing implicit task and data parallelism
  - OPs to automate building of execution task graph (task-level parallelism)
    - Extended to provide implicitly parallelized DSL operations
  - OP archetypes that simplify exposing data-parallelism
    - DeliteOP_Map, DeliteOP_Zipwith, DeliteOP_Reduce, etc.
- *DSL author free to package work into Delite OPs however they deem best*
  - Method call mapped to a deferred OP is a good starting point
  - Sum control structure in OptiML creates two Delite OPs
    - Generate temp results
    - Perform final summation

Delite OPs

```scala
protected[optiml] case class OP_subtract[A]
  (v1: Vector[A], v2: Vector[A])
  extends DeliteOP_SingleTask[Vector[A]](v1,v2) {

  def task = {
    val result = Vector[A](v1.length)
    for (k <- 0 until v1.length)
      result(k) = v1(k) - v2(k)
    result
}}
```

```scala
protected[optiml] case class OP_subtract[A]
  (val collA: Vector[A], val collB: Vector[A],
  val out: Vector[A])
  extends DeliteOP_ZipWith2[A,A,A,Vector] {

  def func = (a,b) => a - b
}
```

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

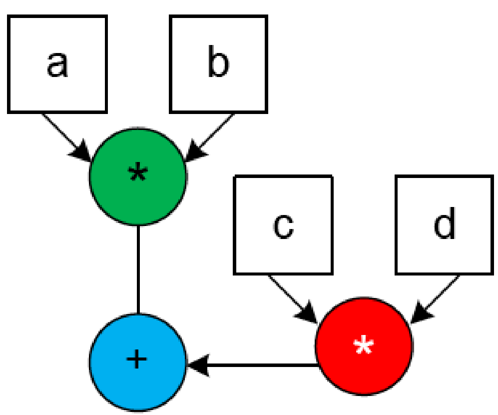
Matrix DSL

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

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

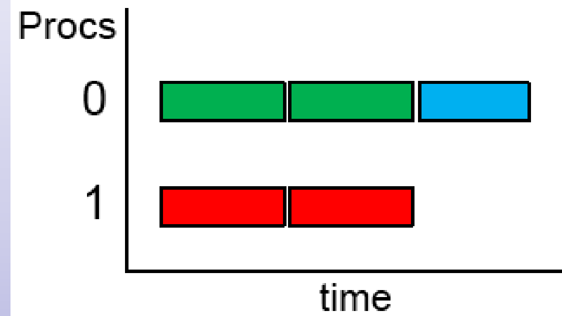
DSL defers OP execution to Delite R.T.

Delite Runtime



Delite applies generic & domain transformations and generates mapping

Hardware Schedule



Delite: A Heterogeneous Parallel Runtime

- Delite schedules OPs to run from the window of currently deferred OPs, honoring the dependencies and anti-dependencies present in the task graph
- OPs are scheduled using a low-cost clustering heuristic in order to minimize communication costs among OPs as well as scheduling overhead
- Data-parallel OPs are submitted to the runtime as a single OP and later split into the desired number of OP chunks.
  - The number of chunks is chosen at scheduling time based on the size of the collection and the availability of hardware resources in the system

Outline

- Introduction
  - Using DSL for parallel programming
- OptiML
  - An example DSL for machine learning
- Delite
  - Runtime and framework for DSL approach
- Delite with GPU
  - Optimizations and automatic code generation
- Experimental Results
- Conclusion

Using GPUs with MATLAB

- MATLAB GPU code

```
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
```

- Jacket GPU code

```
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

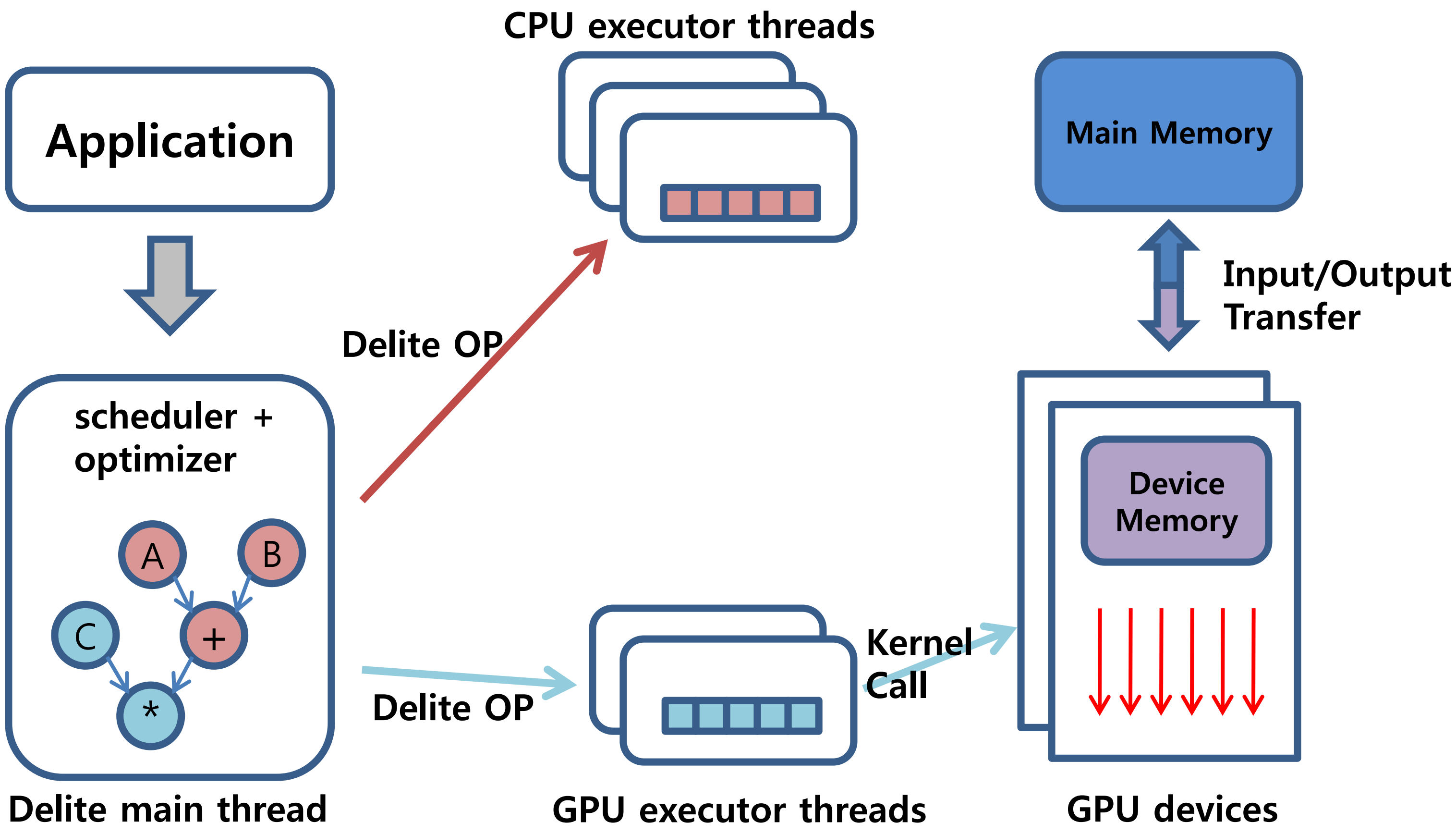
- No change in the application source code
  - Same application code also runs on systems with GPUs
  - Runtime and DSL (not DSL user) dynamically make scheduling decisions (CPU or GPU)
  - Good for portability / productivity
- Performance optimizations under the hood
  - Memory transfers between CPU and GPU
  - On-chip device memory allocation
  - Concurrent kernel executions

Runtime Implementation

- Portion of the task graph (Delite OPs) scheduled on GPU is sent to a dedicated GPU executor
  - 1 GPU executor thread for 1 GPU device

- GPU executor identifies the OP and launches corresponding GPU kernel on GPU device
  - Use asynchronous calls of CUDA Driver APIs
  - Transfer input data from main memory to GPU memory
  - Check timestamps to determine kernel termination
    - Pinned host memory is allocated for timestamps, and each kernel updates the timestamp value after execution
  - Copy back the result data when CPU needs it

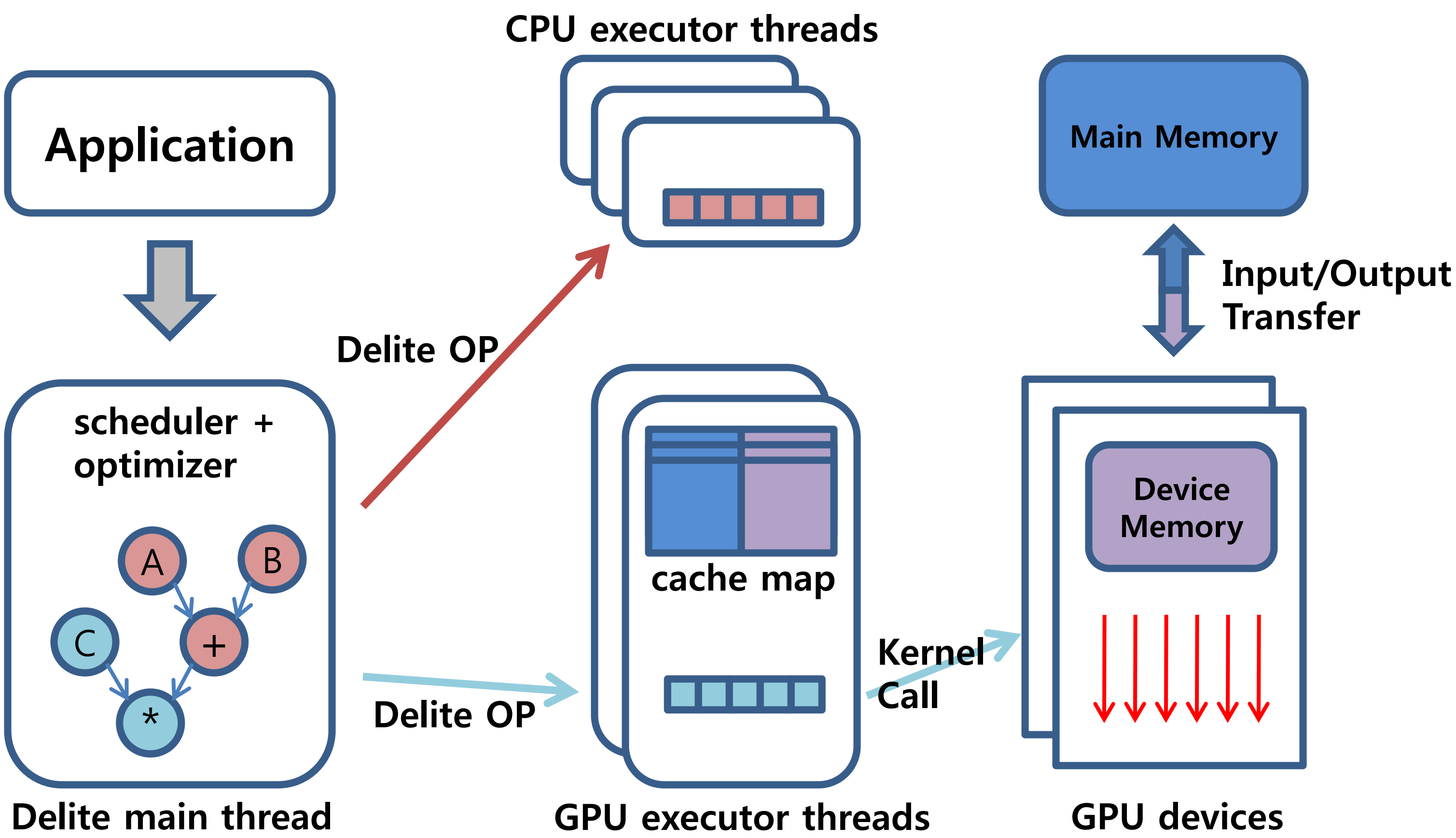
GPU Runtime Diagram



GPU Runtime Optimizations

- High communication cost between CPU/GPU
  - PCI Express 2.0 (x16) bandwidth: 8GB/s max
- Reuse data in GPU device memory
  - Keep input/output data of GPU kernels in GPU memory as long as possible
    - Likely to reuse recently touched data in subsequent kernels
  - Evict only when needed
    - Limited GPU device memory size
- Encourage bulk transfer
  - Transfer entire data structures even when only portions are used

Optimized GPU Runtime Diagram



GPU Memory Coherency

- Problem: DSL OPs with side effects
  - Using GPU device memory as a cache inherently results in the coherency problem between main memory and GPU device memory
- Solution: Use runtime information (list of true/anti dependencies) of OPs to keep correct order of executions with synchronization
  - Generates necessary data transfers
  - When GPU mutates the data
    - CPU worker asks GPU for the updated data
  - When CPU mutates the data
    - GPU invalidates corresponding cache line

GPU Code generation

- GPU kernels for DSL OPs
  - DSL OPs have optimized GPU kernels for the task
  - DSL author provides the GPU kernels
  - Libraries (CUBLAS, CUFFT, ..) can be used

- What about DSL OPs with anonymous functions?
  - The task behavior is not determined by OP itself
    - Given by DSL user, not DSL author
    - Function is passed to the OP as an argument
  - Ex) map{..}, sum(0,n){..}, (0::n){..}

GPU Code generation

<Example Code>

```
val a = Vector.randn(n)
val tau = 3.28
val b = (0::n) { i =>  i * tau / a(i) }
```

- DSL author cannot provide GPU kernels
- Automatically generate corresponding GPU kernels at compile time
  - Use Scala compiler plugin
  - Traverse the application's AST and generate CUDA source code
  - Transform the AST for runtime information

GPU Code Generation Flow

```
val a = Vector.randn(n)
val tau = 3.28
val b = (0::n) { i =>  i * tau / a(i) }
```

Original Application Code

Scala compiler plugin
(AST traversal / transformation)

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

Generated CUDA Code

+

Closure

Kernel ID

Input List

```
val a = Vector.randn(n)
val tau = 3.28
val b = (0::n) { DeliteGPUFunc( {i =>  i * tau / a(i)}, 0, List(a,tau) ) }
```

Transformed Application Code

Outline

- Introduction
  - Using DSL for parallel programming
- OptiML
  - An example DSL for machine learning
- Delite
  - Runtime and framework for DSL approach
- Delite with GPU
  - Optimizations and automatic code generation
- Experimental Results
- Conclusion

Experiments Setup

- 4 Different implementations
  - OptiML+Delite
  - MATLAB (Parallel CPU, GPU, Jacket GPU)

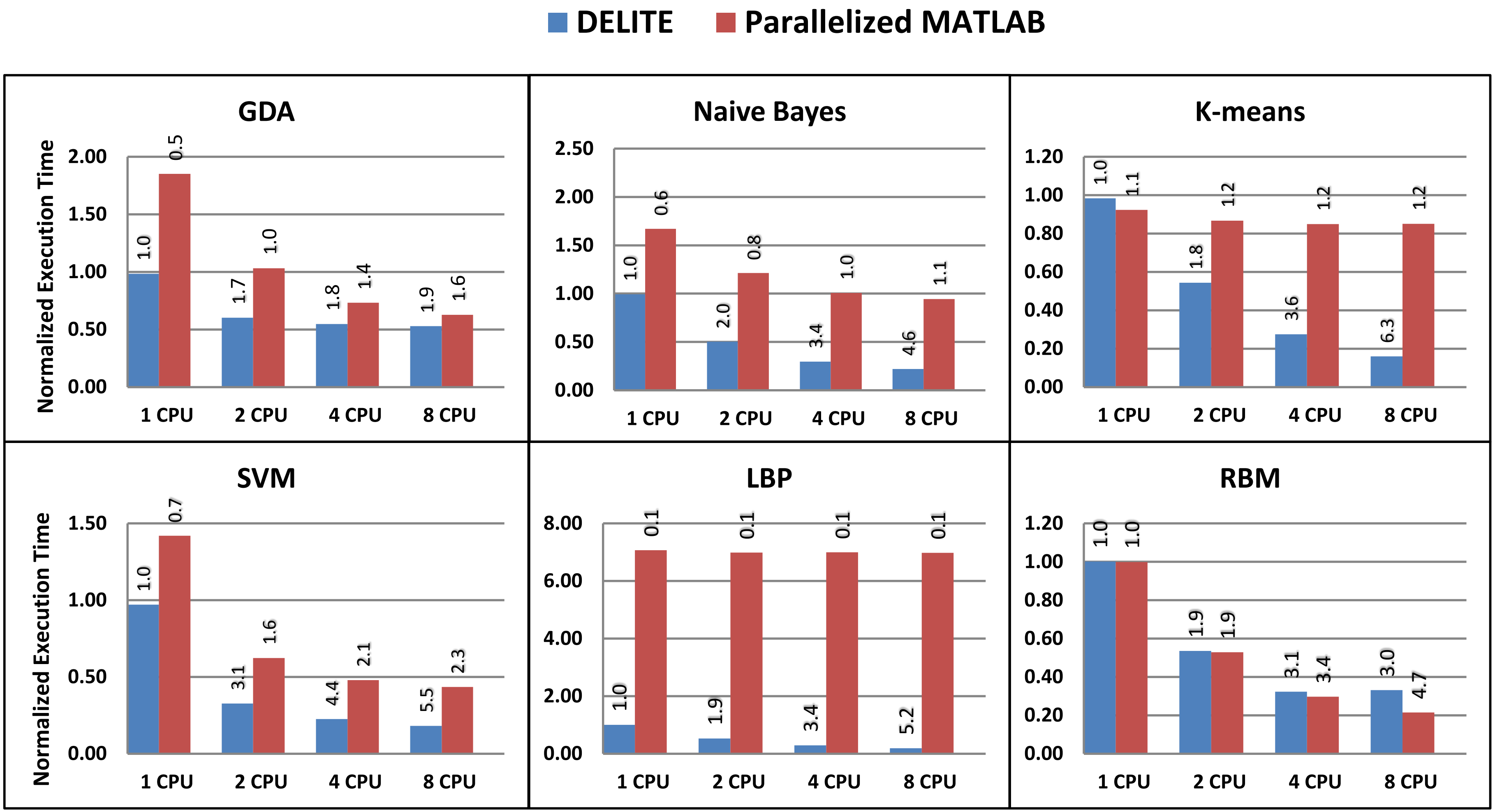
- System 1: Performance Tests
  - Intel Xeon X5550 (2.67GHz)
  - 2 sockets, 8 cores, 16 threads
  - 24 GB DRAM
  - GPU: NVIDIA GTX 275 GPU

- System 2: Scalability Tests
  - Sun UltraSPARC T2+ (1.16GHz)
  - 4 sockets, 32 cores, 256 threads
  - 128 GB DRAM

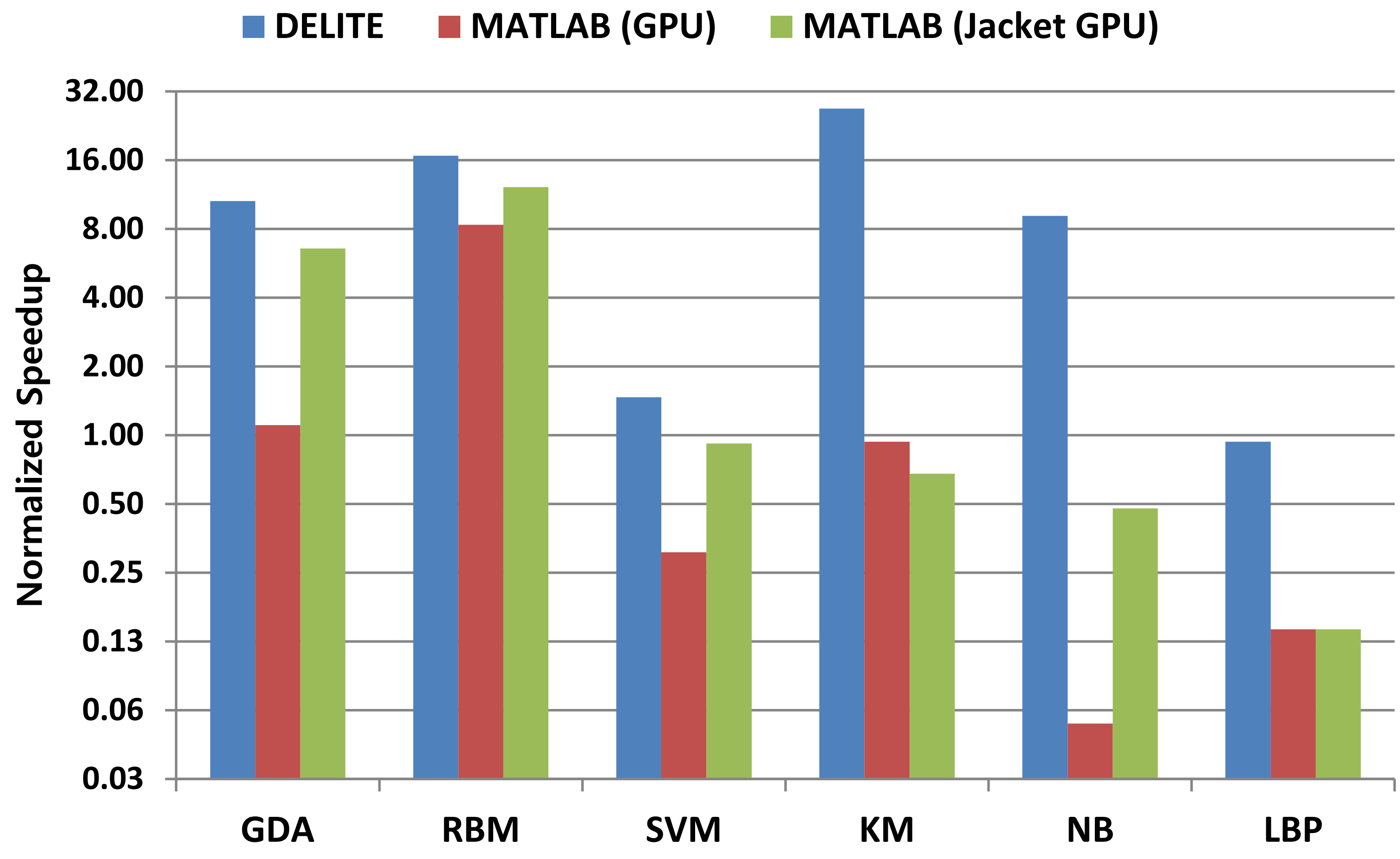
Applications for Experiments

- 6 machine learning domain 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

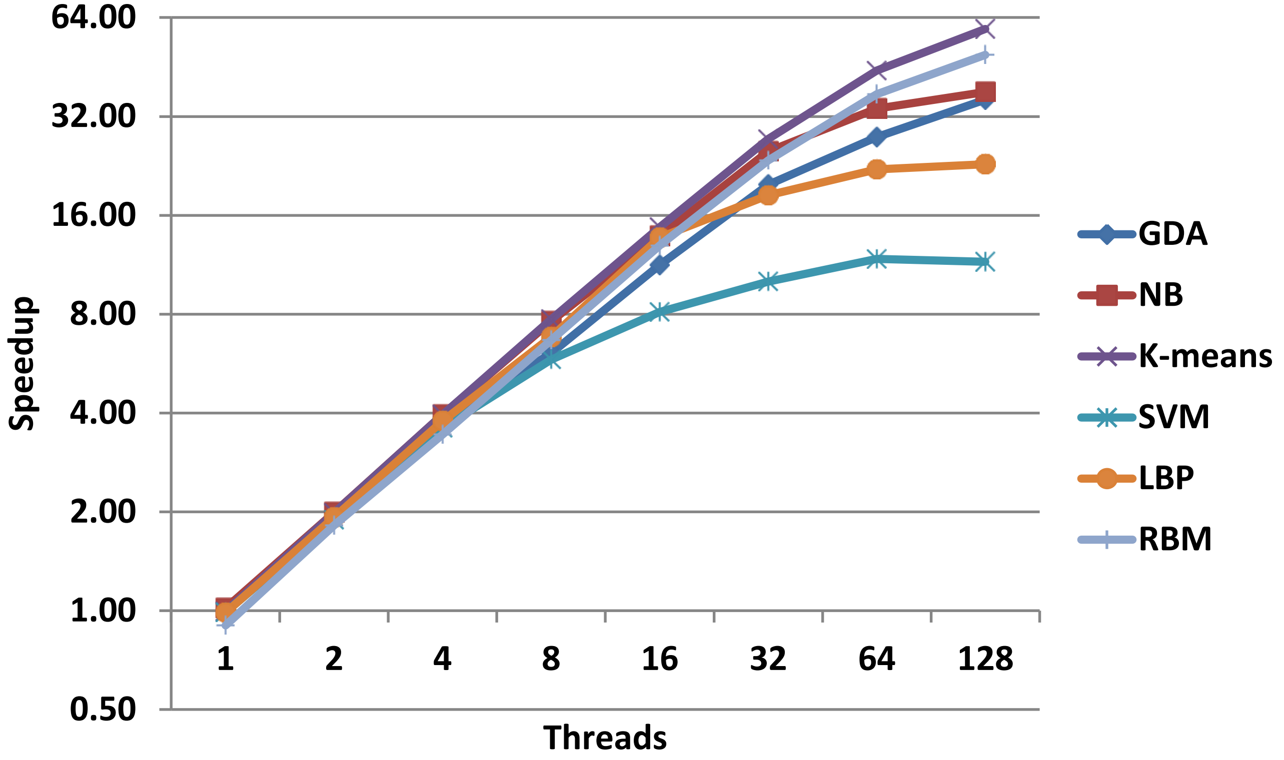
Performance Study (CPU)



Performance Study (GPU)

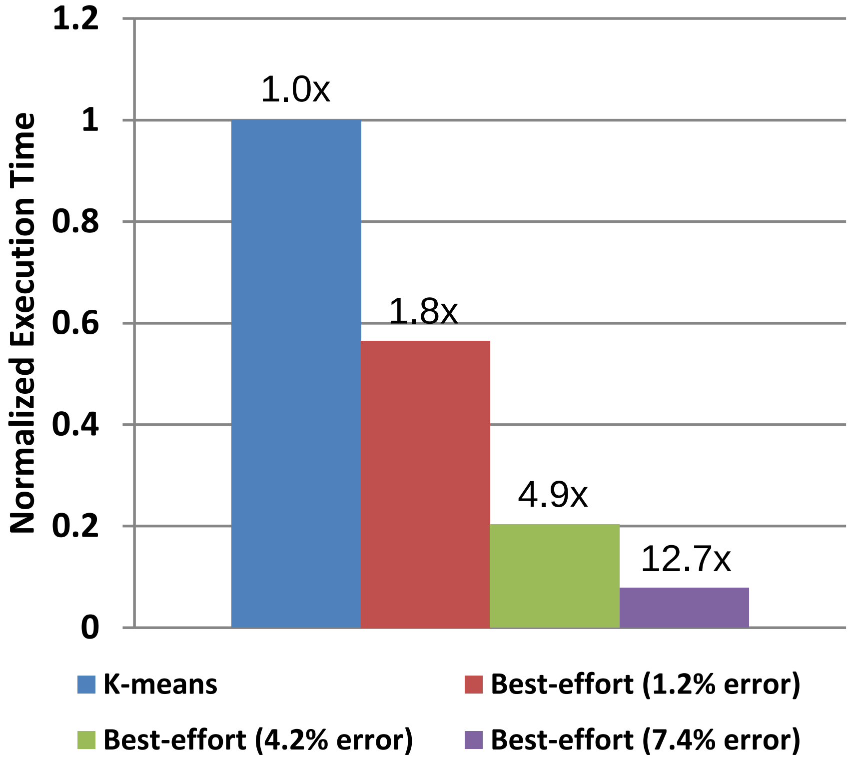


Scalability Study

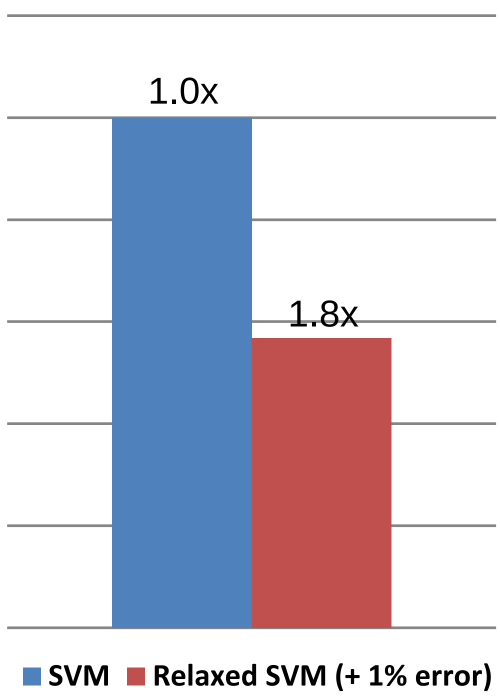


Domain Specific Optimizations

- **Best Effort Computation**



- **Relaxed Dependencies**



Conclusion

- Using Domain Specific Languages (DSLs) is a potential solution for heterogeneous parallelism
  - OptiML, an example DSL for ML demonstrates productivity, portability and performance
  - Delite, as a framework, simplifies developing implicitly parallel DSLs that target heterogeneous platforms
  - Delite, as a runtime, maximizes performance through dynamic optimizations and scheduling decisions
  - GPU specific optimizations and automatic CUDA code generation allows efficient use of GPU devices with Delite runtime
  - Experimental results show that OptiML+Delite outperforms various MATLAB implementations

THANK YOU