Stanford | ENGINEERING
Electrical Engineering
Computer Science

PERVASIVE PARALLELISM LABORATORY PPL

Big Data Analytics in the Age of Accelerators

Kunle Olukotun
Cadence Design Systems Professor
Stanford University

Reconfigurable Computing for the Masses, Really?

September 4, 2015

Research Goals

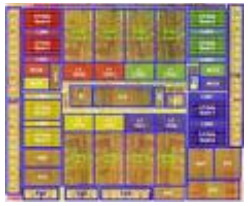
- Unleash full power of modern computing platforms
  - Heterogeneous parallelism
- Make parallel application development practical for the masses (Joe/Jane the programmer)
  - Parallelism is not for the average programmer
  - Too difficult to find parallelism, to debug, and get good performance
- Parallel applications without parallel programming

Modern Big Data Analytics

- Predictive Analytics ≈ Data Science
- Enable better decision making
  - Data is only as useful as the decisions it enables
- Deliver the capability to mine, search and analyze this data in real time
  - Requires the full power of modern computing platforms

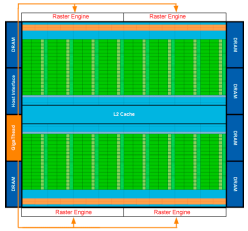
Data Center Computing Platforms

10s of cores



Multicore
Multi-socket

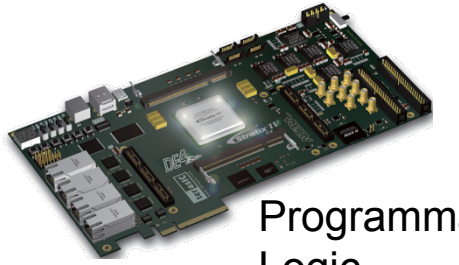
> 1 TFLOPS



Graphics
Processing
Unit (GPU)

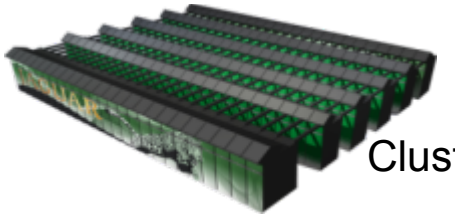
Accelerators

Reconfigurable comput.



Programmable
Logic

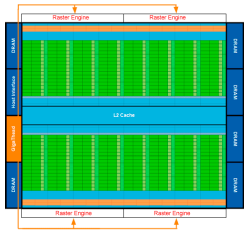
1000s of nodes



Cluster

Expert Parallel Programming

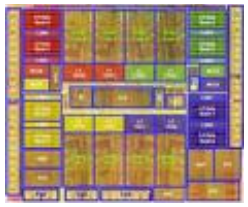
> 1 TFLOPS



Graphics Processing Unit (GPU)

CUDA
OpenCL

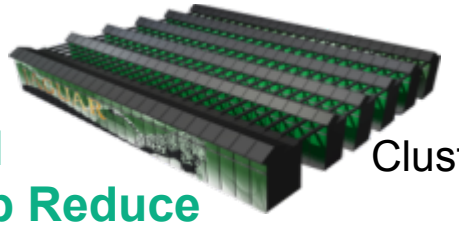
10s of cores



Multicore
Muti-socket

Threads
OpenMP

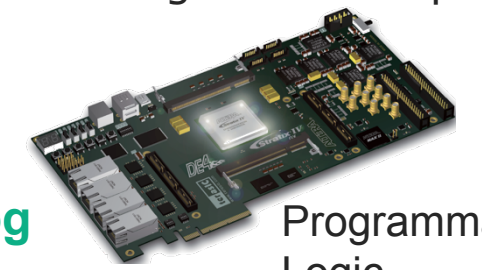
1000s of nodes



Cluster

MPI
Map Reduce

Reconfigurable comput.



Programmable Logic

Verilog
VHDL

MPI: Message Passing Interface

MapReduce vs CUDA

- **MapReduce: simplified data processing on large clusters**

  J Dean, S Ghemawat
  Communications of the ACM, 2008

  - Cited by 14764

- **Scalable parallel programming with CUDA**

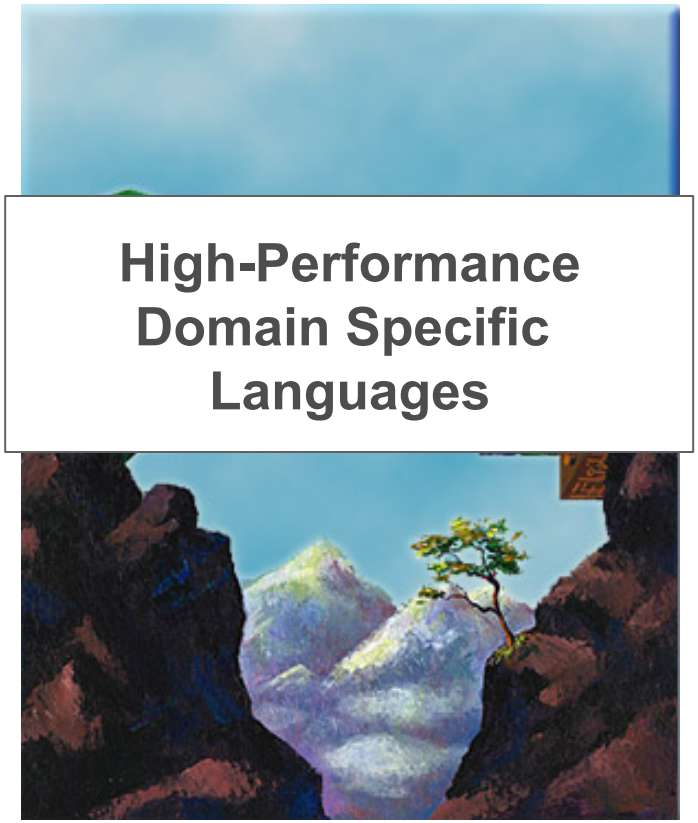
  J Nickolls, I Buck, M Garland, K Skadron
  ACM Queue, 2008

  - Cited by 1205

Big-Data Analytics Programming Challenge

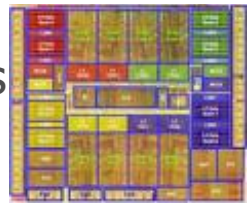
Data Analytics Application

- **Data Prep**
- **Data Transform**
- **Network Analysis**
- **Predictive Analytics**



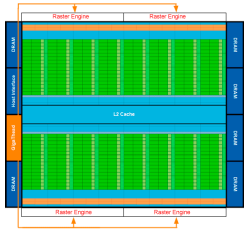
High-Performance Domain Specific Languages

Pthreads
OpenMP



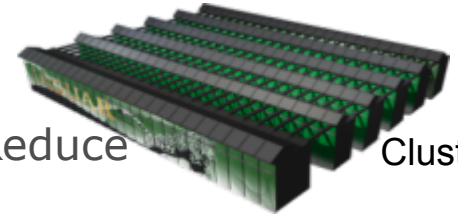
Multicore

CUDA
OpenCL



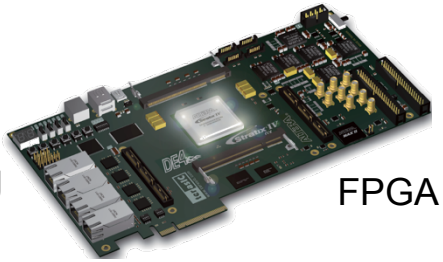
GPU

MPI
Map Reduce



Cluster

Verilog
VHDL

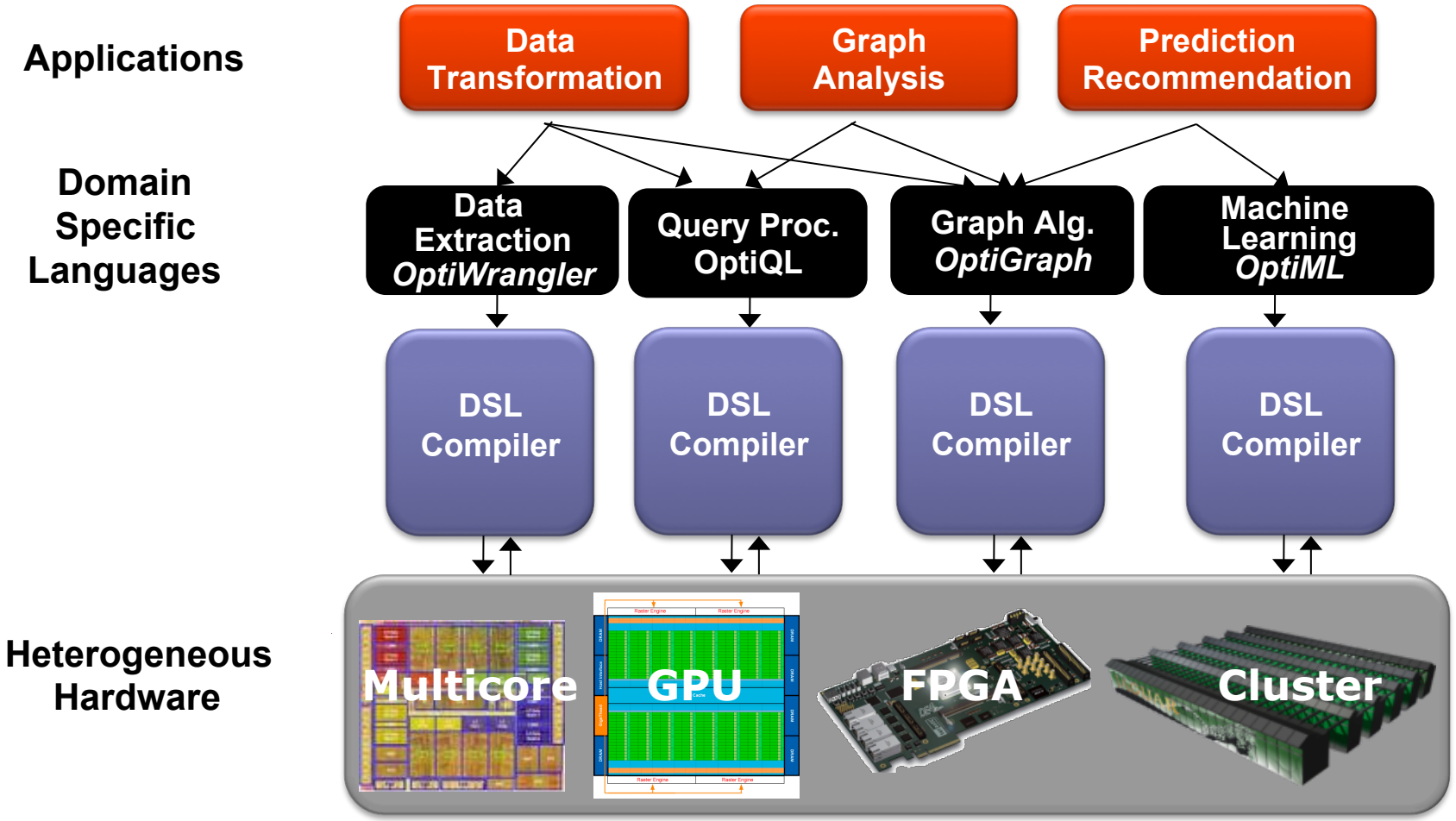


FPGA

Domain Specific Languages

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

High Performance DSLs for Data Analytics



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
  - Base types
    - Vector[T], Matrix[T], Graph[V,E], Stream[T]
  - Subtypes
    - TrainingSet, IndexVector, Image, …
- Implicitly parallel control structures
  - sum{…}, (0::end) {…}, gradient { … }, untilconverged { … }
  - Allow anonymous functions with restricted semantics to be passed as arguments of the control structures

K-means Clustering in OptiM

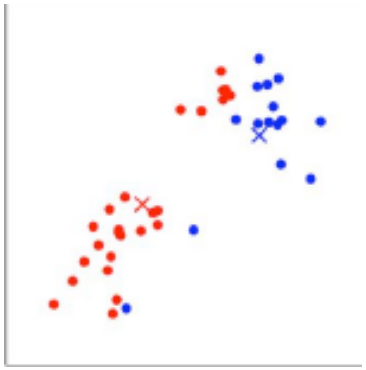
```
untilconverged(kMeans, tol){kMeans =>
  val clusters = samples.groupRowsBy { sample =>
       kMeans.mapRows(mean => dist(sample, mean)).minIndex
  }
  val newKmeans = clusters.map(e => e.sum /
  newKmeans
}
```

assign each sample to the closest mean

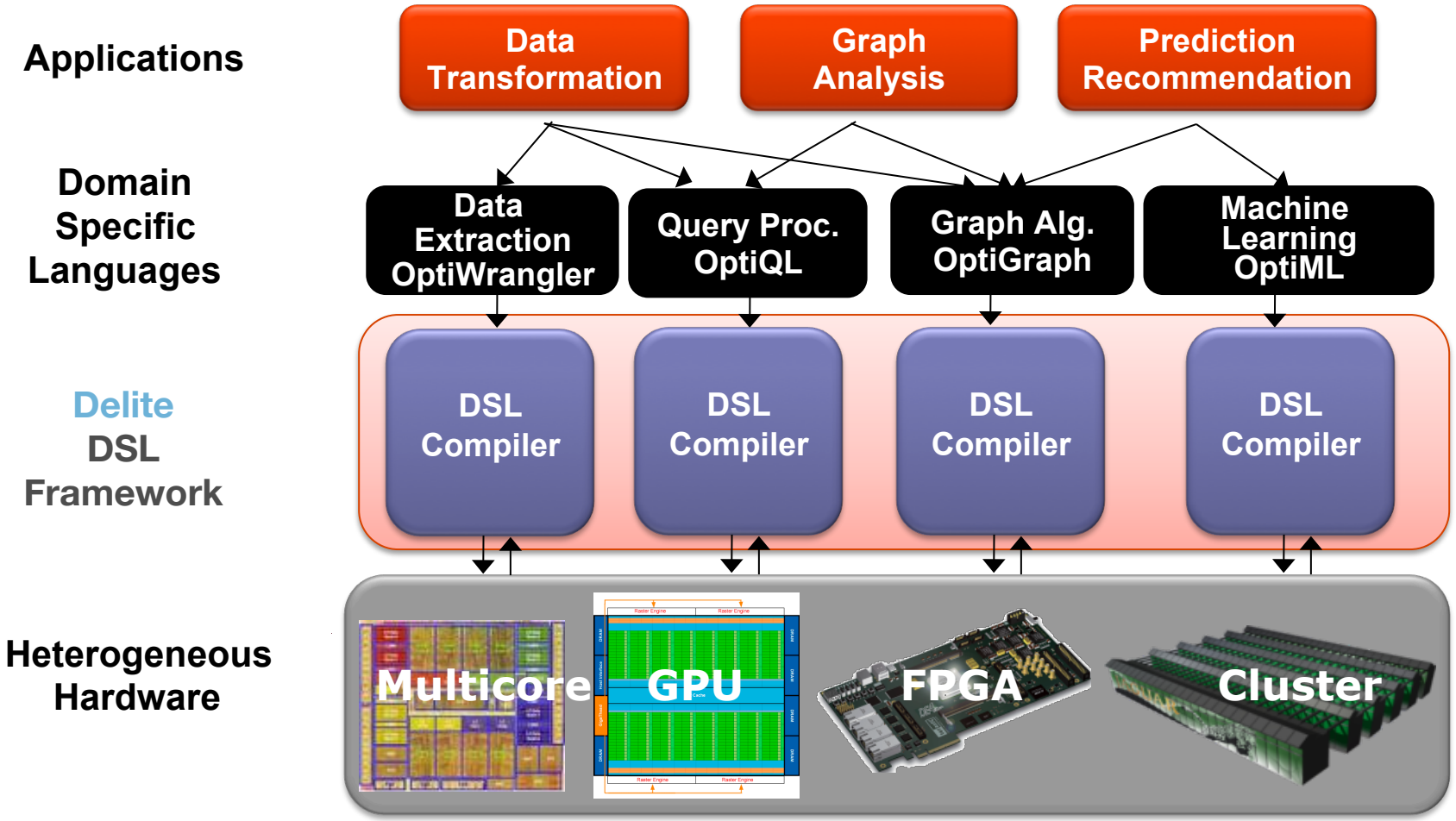
calculate distances to current means

move each cluster centroid to the mean of the points assigned to it

- No explicit map-reduce, no key-value pairs
- No distributed data structures (e.g. RDDs)
- No annotations for hardware design
- Efficient multicore and GPU execution
- Efficient cluster implementation
- Efficient FPGA hardware



High Performance DSLs for Data Analytics with Delite

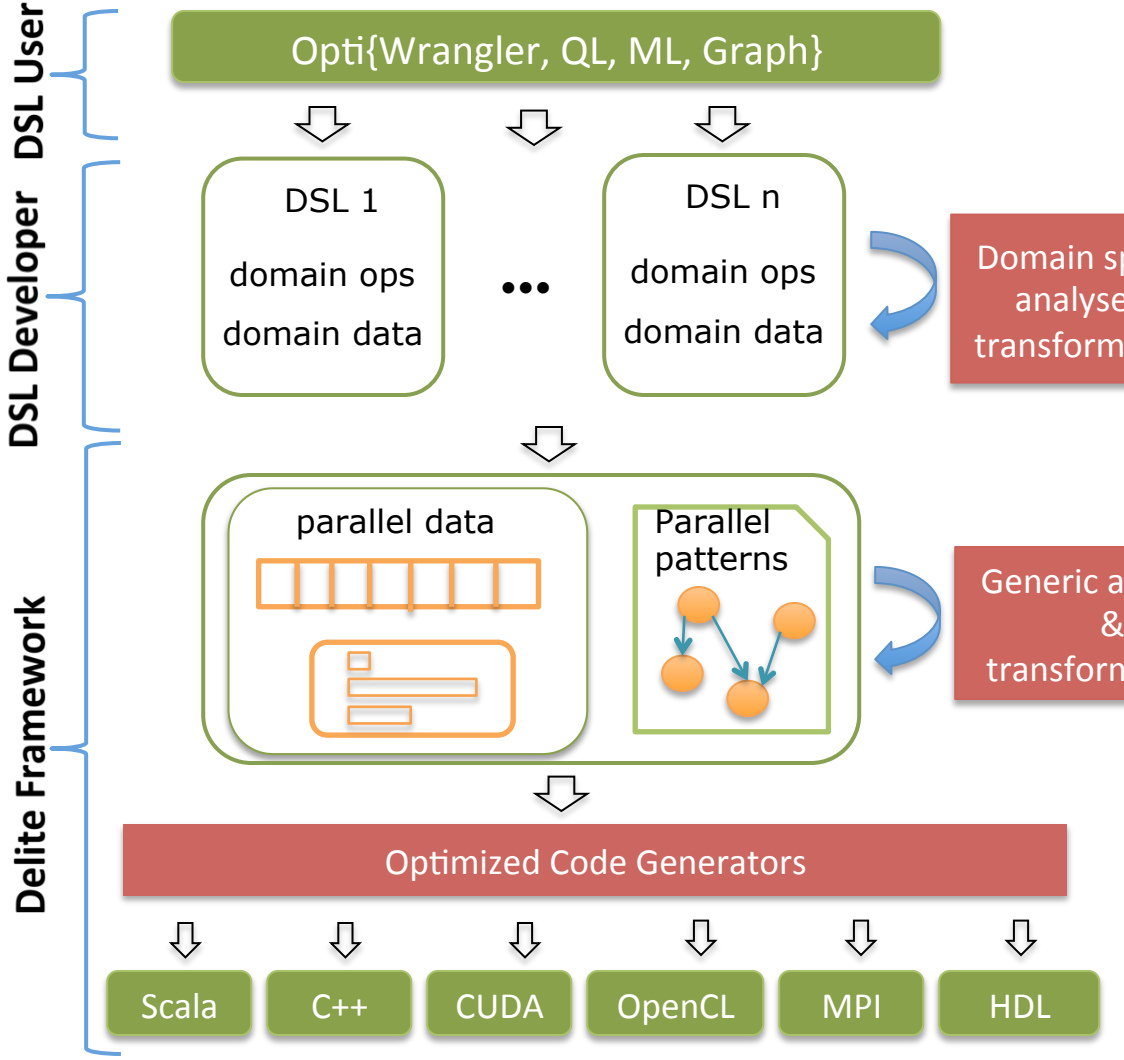


Delite: A Framework for High Performance DSLs

- Overall Approach: Generative Programming for "Abstraction without regret"
  - Embed compilers in Scala libraries
    - Scala does syntax and type checking
  - Use metaprogramming with LMS (type-directed staging) to build an intermediate representation (IR) of the user program
  - Optimize IR and map to multiple targets
- Goal: Make embedded DSL compilers easier to develop than stand alone DSLs
  - As easy as developing a library

Delite Overview

K. J. Brown et. al., "A heterogeneous parallel framework for domain-specific languages," *PACT, 2011*.



Key elements

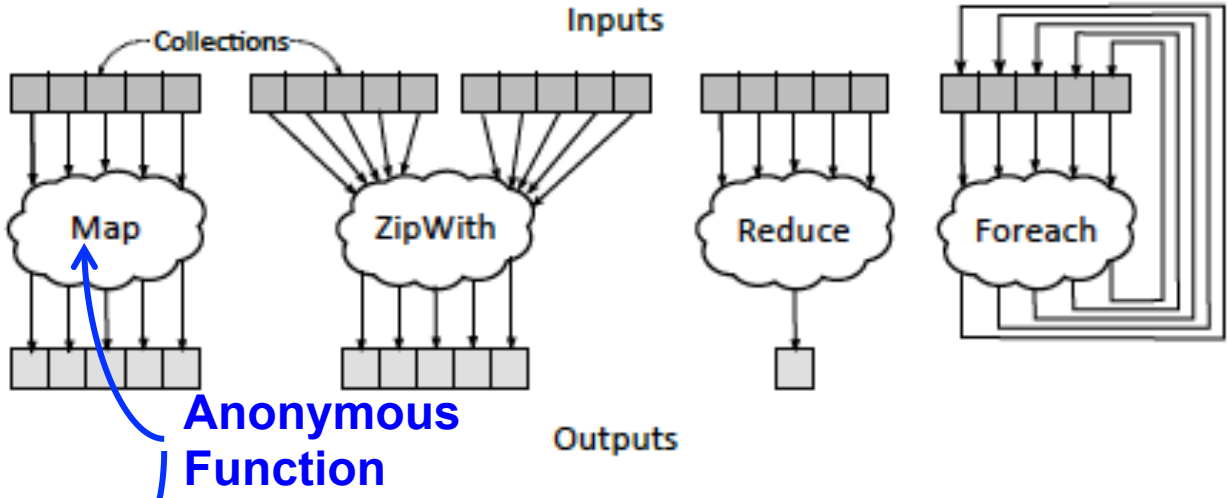
- DSLs embedded in Scala
- IR created using type-directed staging
- Domain specific optimization
- General parallelism and locality optimizations
- Optimized mapping to HW targets

Parallel Patterns: Delite Ops

- Parallel execution patterns
  - Functional: Map, FlatMap, ZipWith, Reduce, Filter, GroupBy, Sort, Join, union, intersection
  - Non-functional: Foreach, ForeachReduce, Sequential
  - Set of patterns can grow over time
- Provide high-level information about data access patterns and parallelism
- DSL author maps each domain operation to the appropriate pattern
  - Delite handles parallel optimization, code generation, and execution for all DSLs
- Delite provides implementations of these patterns for multiple hardware targets
  - High-level information creates straightforward and efficient implementations
  - Multi-core, GPU, clusters and FPGA

Parallel Patterns

Most data analytic computations can be expressed as parallel patterns on collections (e.g. sets, arrays, table)



Pattern	Example
map	`in map { e => e + 1 }`
zipwith	`inA zipWith(inB) { (eA,eB) => eA + eB }`
foreach	`inA foreach { e => if (e>0) inB(e) = true }`
filter	`in filter { e => e > 0}`
reduce	`in reduce { (e1,e2) => e1 + e2 }`
groupby	`in groupBy { e => e.id }`

Other patterns: sort, intersection, union

Parallel Patterns are Universal DSL Components

DSL	Parallel Patterns
OptiWrangler (data extraction)	Map, ZipWith, Reduce, Filter, Sort, GroupBy
OptiQL (query processing)	Map, Reduce, Filter, Sort, GroupBy, Intersection
OptiML (machine learning)	Map, ZipWith, Reduce, Foreach, GroupBy, Sort
OptiGraph (graph analytics)	Map, Reduce, Filter, GroupBy

Parallel Pattern Language (PPL)

- A data-parallel language that supports parallel patterns
- Example application: *k*-means

```
val clusters = samples groupBy { sample =>
  val dists = kMeans map { mean =>
    mean.zip(sample){ (a,b) => sq(a - b) } reduce { (a,b) => a + b }
  }
  Range(0, dists.length) reduce { (i,j) =>
    if (dists(i) < dists(j)) i else j
  }
}
val newKmeans = clusters map { e =>
  val sum = e reduce { (v1,v2) => v1.zip(v2){ (a,b) => a + b } }
  val count = e map { v => 1 } reduce { (a,b) => a + b }
  sum map { a => a / count }
}
```

Key Aspects of Delite IR

- Sea of Nodes
  - Data-flow graph
  - Explicit effects encoded as data dependencies
- Parallel Patterns (Delite Ops)
  - Sequential, Map, Reduce, Zip, Foreach, Filter, GroupBy, Sort, ForeachReduce, FlatMap
  - Skeletons that DSL authors extend
- Data Structures also in IR
  - Structs with restricted fields (scalars, arrays, structs)
  - Field access and struct instantiation is explicit and constructs IR nodes
- Multiple Views
  - Generic, Parallel, Domain-Specific
  - Can optimize at any level

Mapping Nested Parallel Patterns to GPUs

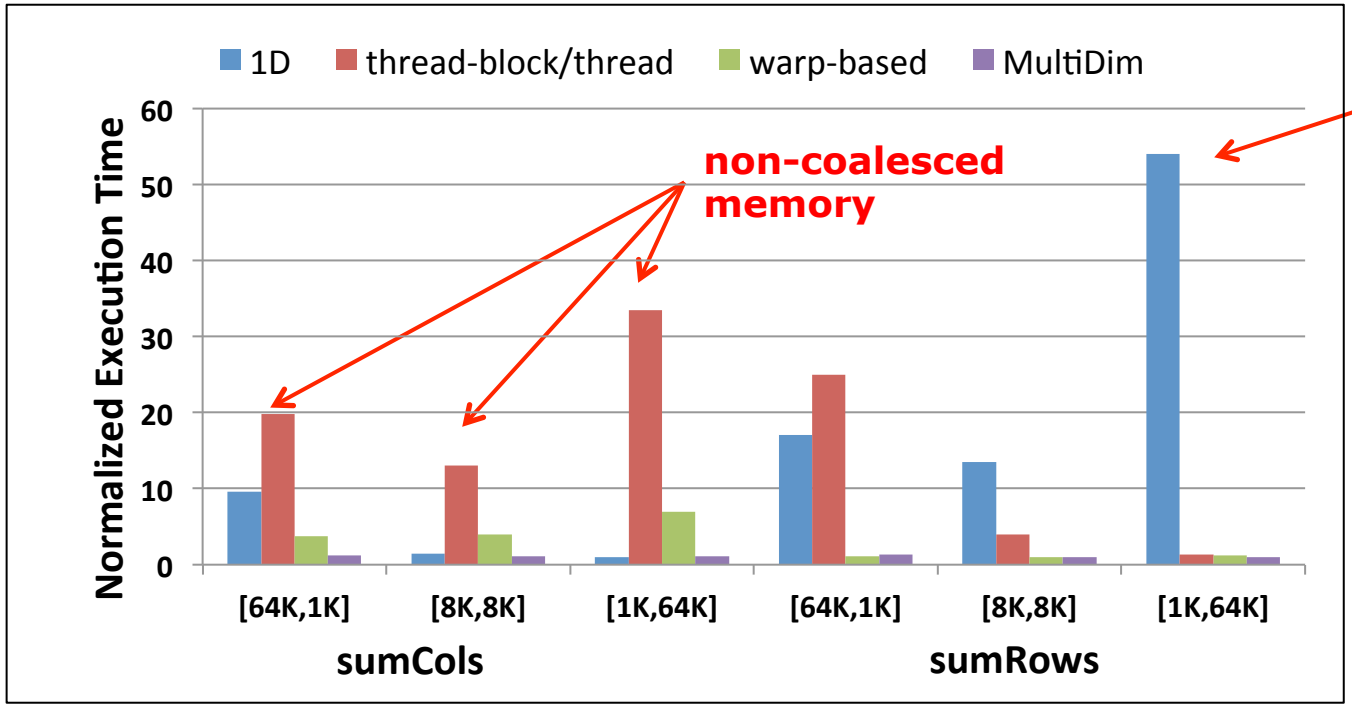
- Parallel patterns are often nested in applications
  - > 70% apps in Rodinia benchmark contain kernels with nested parallelism
- Efficiently mapping parallel patterns on GPUs becomes significantly more challenging when patterns are nested
  - Memory coalescing, divergence, dynamic allocations, …
  - Large space of possible mappings

Mapping Nested Ops to GPUs

```
m = Matrix.rand(nR,nC)
v = m.sumCols
```

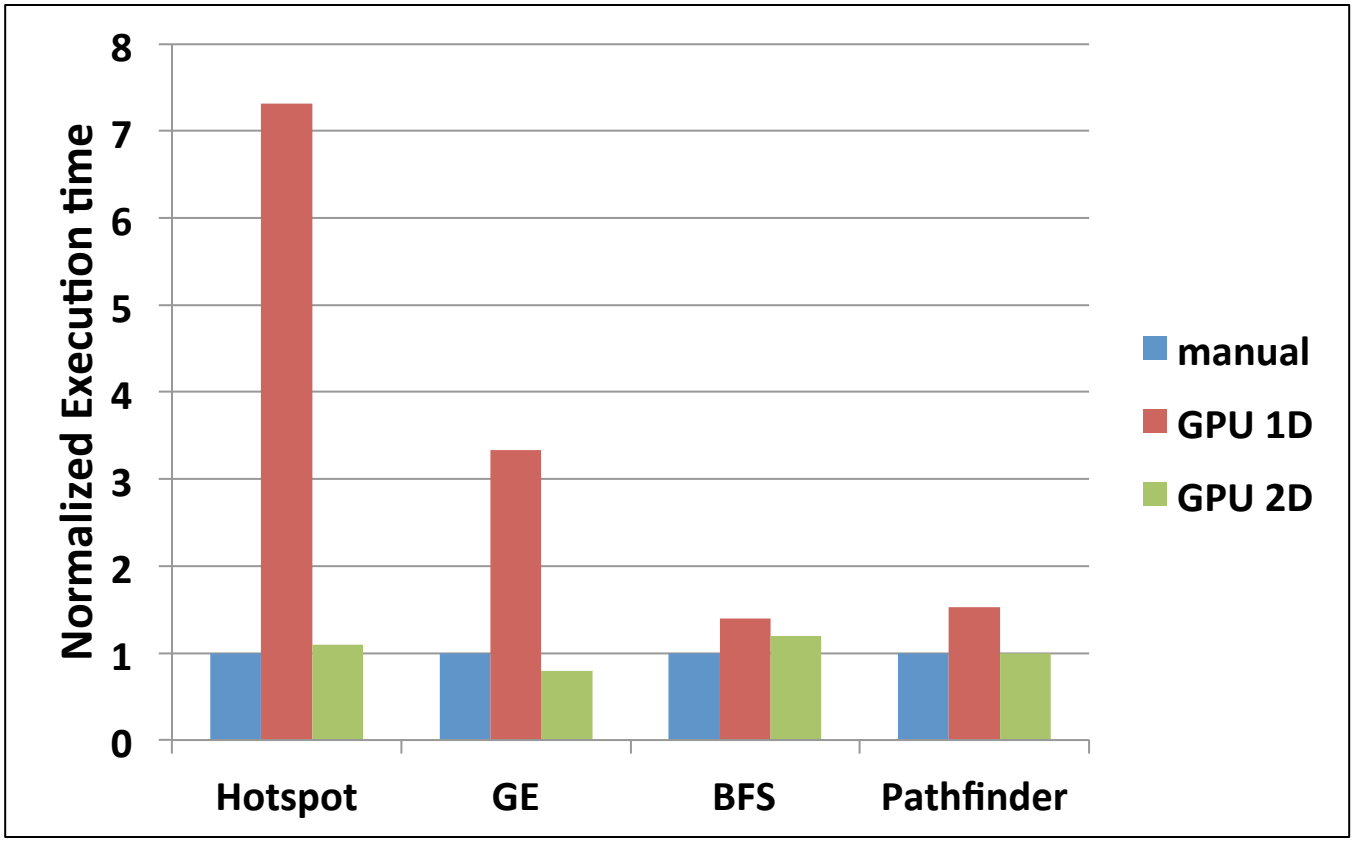
```
map (i)
  reduce(j)
```

```
m = Matrix.rand(nR,nC)
v = m.sumRows
```



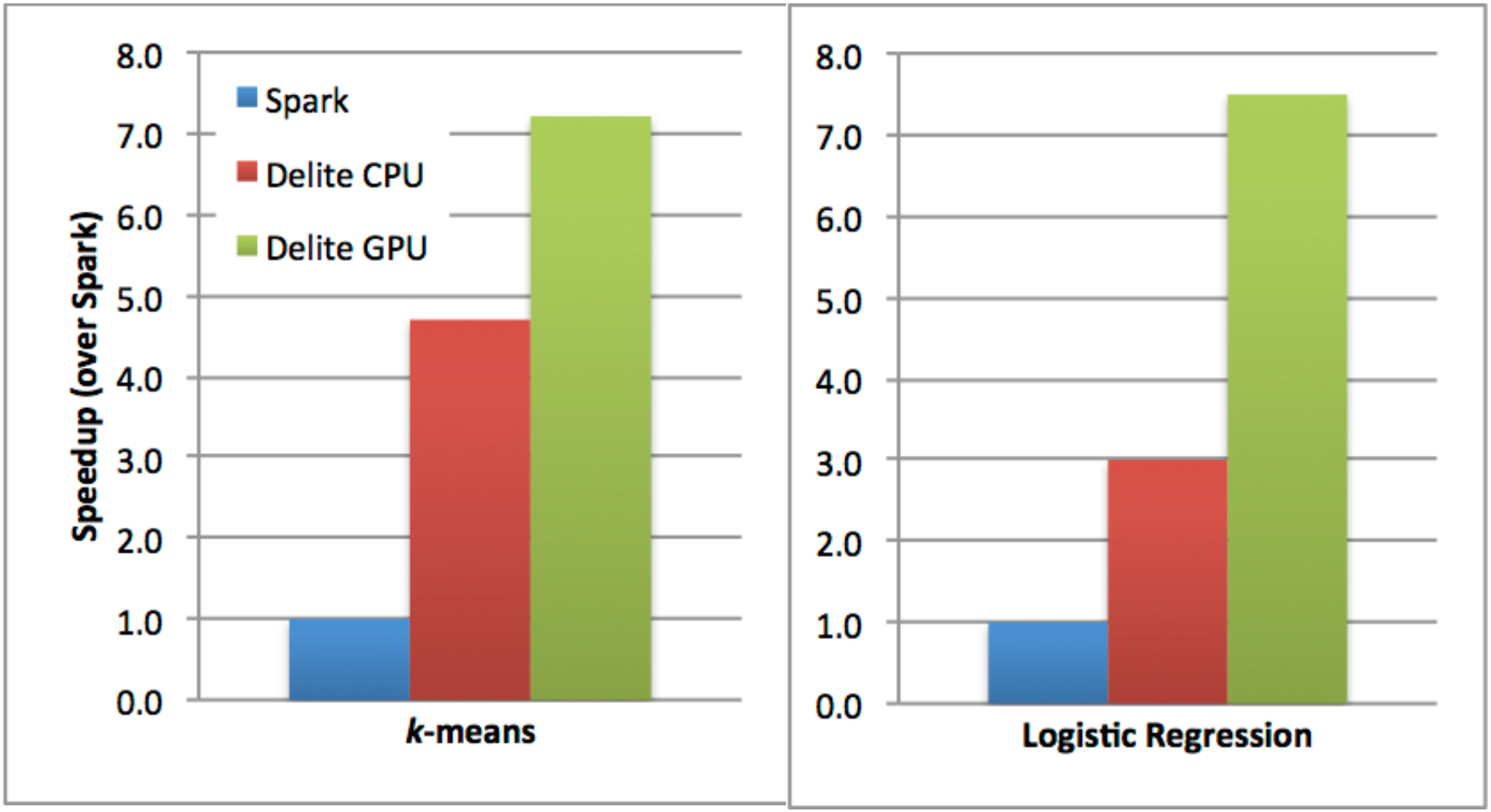
HyoukJoong Lee et. al, "Locality-Aware Mapping of Nested Parallel Patterns on GPUs," *MICRO'14*

Nested Delite Ops on Rodinia Apps



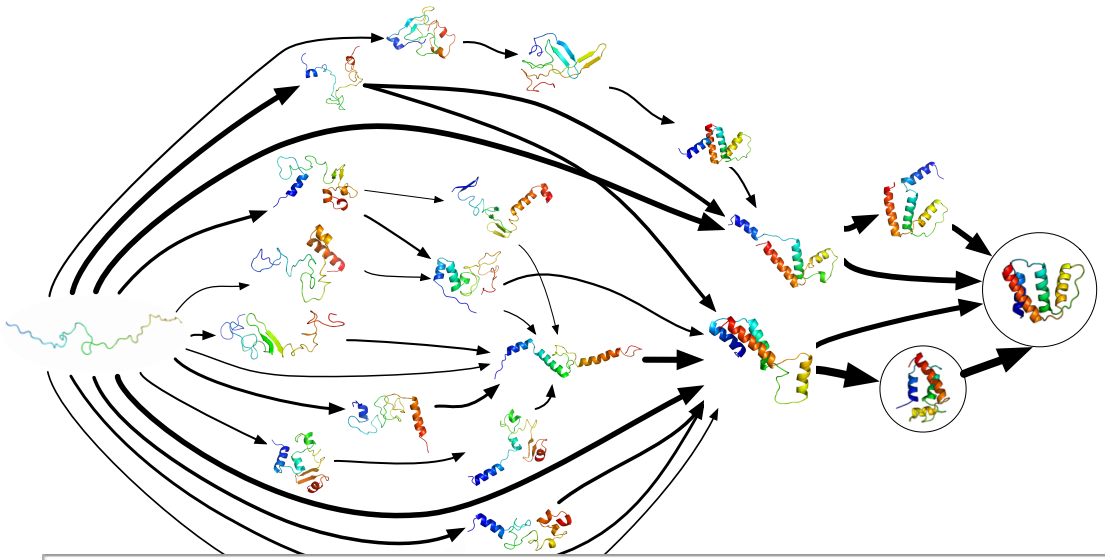
- 2D mapping exposes more parallelism
- 2D mapping enables coalesced memory accesses

Heterogeneous Cluster Performance



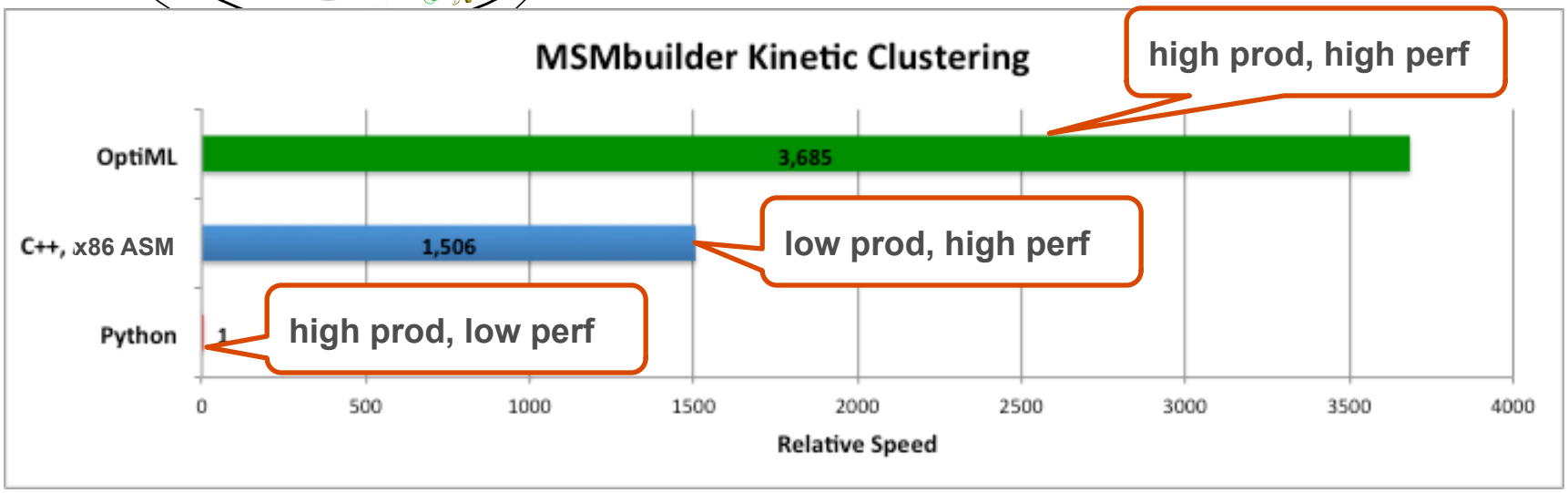
4 node local cluster: 3.4 GB dataset

MSM Builder Using OptiML
with Vijay Pande

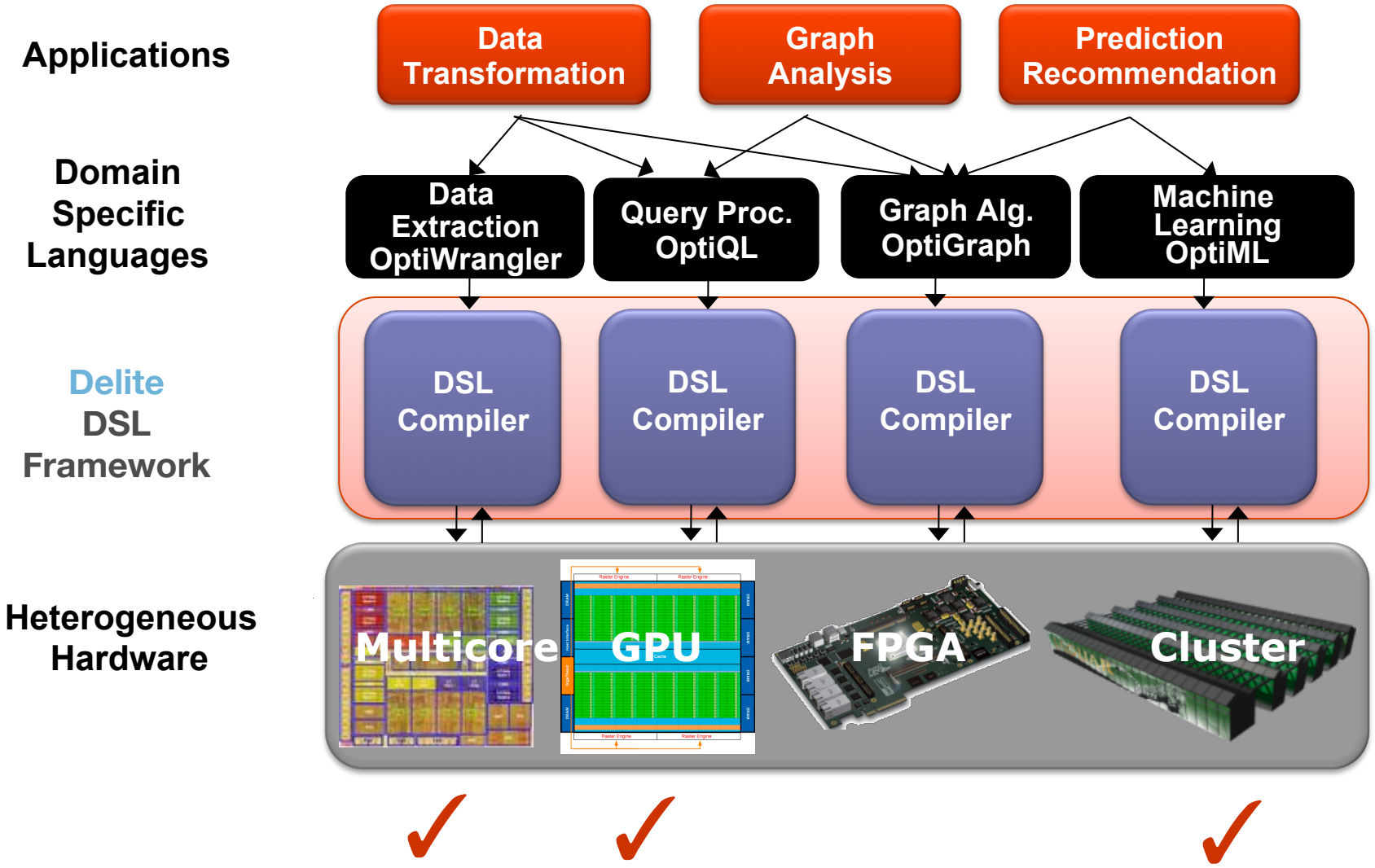


Markov State Models (MSMs)

MSMs are a powerful means of modeling the structure and dynamics of molecular systems, like proteins



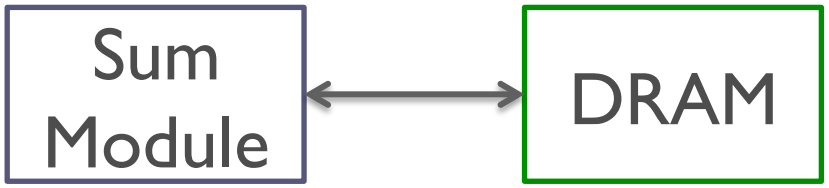
High Performance Data Analytics with Delite



FPGAs in the Datacenter?

- FPGAs based accelerators
  - Recent commercial interest from Baidu, Microsoft, and Intel
  - Key advantage: Performance, Performance/Watt
  - Key disadvantage: lousy programming model
- Verilog and VHDL poor match for software developers
  - High quality designs
- High level synthesis (HLS) tools with C interface
  - Medium/low quality designs
  - Need architectural knowledge to build good accelerators
  - Not enough information in compiler IR to perform access pattern and data layout optimizations
  - Cannot synthesize complex data paths with nested parallelism

Hardware Design with HLS is Easy



Add 512 integers stored in external DRAM

```
void(int* mem){

        mem[512] = 0;

        for(int i=0; i<512; i++){
                mem[512] += mem[i];
        }
}
```

27,236 clock cycles for computation

Two-orders of magnitude too long!

High Quality Hardware Design with HLS is Still Difficult

```
#define ChunkSize (sizeof(MPort)/sizeof(int))
#define LoopCount (512/ChunkSize)

void(MPort* mem){

        MPort buff[LoopCount];
        memcpy(buff, mem, LoopCount);

        int sum=0;
        for(int i=1; i<LoopCount; i++){
                #pragma PIPELINE
                for(int j=0; j<ChunkSize; j++){
                        #pragma UNROLL
                        sum+=(int)(buff[i]>>j*sizeof(int)*8);
                }
        }
        mem[512]=sum;
}
```

Width of the DRAM controller interface

Burst access

Use local variable

Special compiler directives

Reformat code

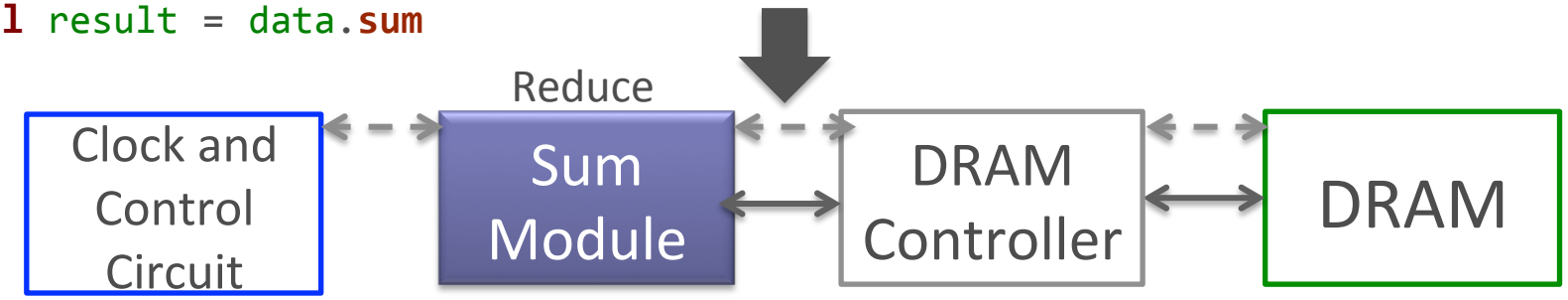
302 clock cycles for computation

Make HLS Easier with Delite

Nithin George et. al. "Hardware system synthesis from Domain-Specific Languages," *FPL 2014*

- Delite generates HLS code for each parallel pattern in the application
  - Currently targets Xilinx Vivado HLS
  - Optimizations for burst DRAM access

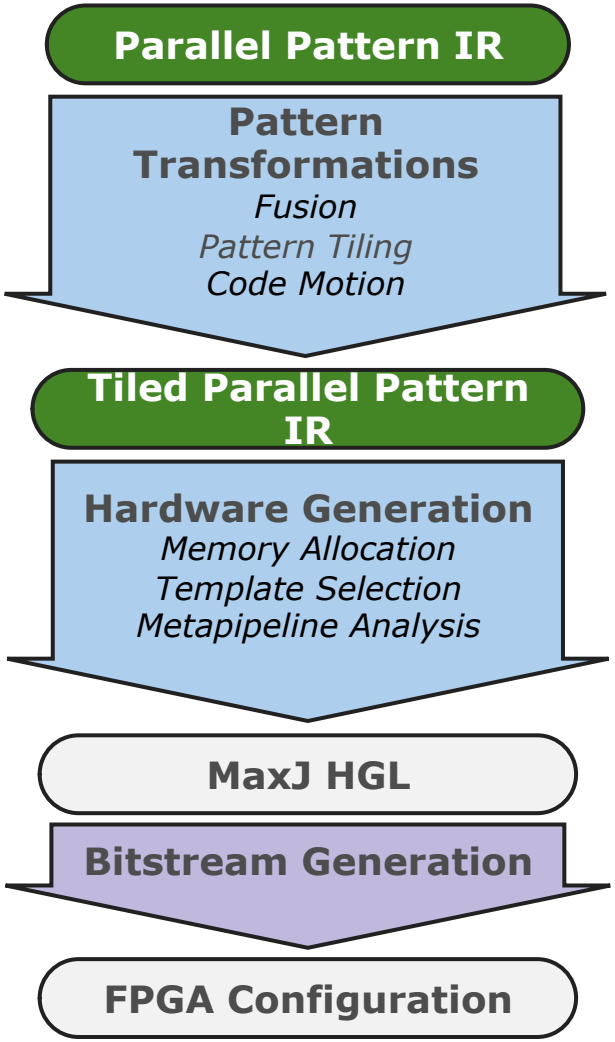
```
…
// data is an array of 512 elements that has been declared and initialized.
val result = data.sum
…
```



368 clock cycles for computation

Optimized Approach to HW Generation

- Key optimizations:
  - Parallel pattern tiling to maximize on-chip data reuse
  - *Metapipelines* to exploit nested parallelism
- Generate MaxJ code
  - Use Maxeler's MaxCompiler to generate FPGA bitstream



Generalized Parallel Pattern Language (GPPL)

- Enable use of general pattern matching rules for automatic tiling
  - Polyhedral modeling limits array accesses to affine functions of loop indices
  - Pattern matching rules can be run on any input program, even those with random and data-dependent accesses

	Pattern	Description	Application Usage Example
N-Dim dense	Map	Generates one element per loop index, aggregates result into fixed size output collection	`x.map{e => 2*e}` `x.zip(y){(a,b) => a + b}`
	MultiFold	Reduces one partial result per loop index into a subsection of a fixed size accumulator	`mat.map{row =>` `  row.fold{(a,b) => a + b}}`
1-Dim sparse	FlatMap	Concatenates arbitrary number of elements per loop index into dynamic output collection	`data.filter{e => e > 0}`
	GroupByFold	Reduces arbitrary number of partial results per loop index into buckets based on generated keys	`img.histogram`

PPL Fusion of *k*-means

Fused

Fusion creates MultiFold

```
val clusters = samples groupBy { sample =>
  val dists = kMeans map { mean =>
    mean.zip(sample){ (a,b) => sq(a - b) } reduce { (a,b) => a + b }
  }
  Range(0, dists.length) reduce { (i,j) =>
    if (dists(i) < dists(j)) i else j
  }
}
val newKmeans = clusters map { e =>
  val sum = e reduce { (v1,v2) => v1.zip(v2){ (a,b) => a + b } }
  val count = e map { v => 1 } reduce { (a,b) => a + b }
  sum map { a => a / count }
}
```

Core of *k*-means using GPPL

```
sums = multiFold(n){i =>
  pt1 = points.slice(i, *)
  minDistWithIndex = multiFold(k){j =>
    pt2 = centroids.slice(j, *)
    dist = distance(pt1, pt2)
    (0, (dist, j))
  }{(a,b) => if (a._1 < b._1) a else b }
  minIndex = minDistWithIndex._2
  (minIndex, pt1)
}{(a,b) => map(d){k => a(k) + b(k) }
```

For each point in a set of *n* points

Get point *pt1* from points set

For each centroid in set of *k* centroids

Get centroid *pt2* from centroids set

Calculate distance between point & centroid

Take closer of current (distance,index) pair & previously found closest (distance,index)

Extract index of closest centroid

At index of closest centroid, **add point (with dimension *d*) to accumulator (non-affine access)**

Parallel Pattern Tiling 1

- **Strip mining**: Chunk parallel patterns into nested patterns of known size, chunk predictable array accesses with copies
  - **Copy** becomes local memory with hardware prefetching
  - Strip mined patterns enable computation reordering

Example	Parallel Patterns	Strip Mined Patterns

Simple Map
x: Array, size d
`x.map{e => 2*e}`
	`map(d){i => 2*x(i)}`	`multiFold(d/b){ii =>`
		`  xTile = x.copy(b + ii)`
		`  (i, map(b){i => 2*xTile(i) }) }`

Sum through matrix rows
mat: Matrix, size m x n
`mat.map{row =>`
`  row.fold{(a,b) => a + b}}`
	`multiFold(m,n){i,j =>`
	`  (i, mat(i,j))`
	`}{(a,b) => a + b}`
		`multiFold(m/b0,n/b1){ii,jj =>`
		`  matTile = mat.copy(b0+ii,b1+jj)`
		`  (ii, multiFold(b0,b1){i,j =>`
		`  (i, matTile(i,j))`
		`  }{(a,b) => a + b})`
		`}{(a,b) => a + b}`

Simple data filter
data: Array, size d
`data.filter{e => e > 0}`
	`flatMap(d){i =>`
	`  if (x(i) > 0) x(i)`
	`  else []`
	`}`
		`flatMap(d/b){ii =>`
		`  xTile = x.copy(b + ii)`
		`  flatMap(b){i =>`
		`  if (xTile(i) > 0) xTile(i)`
		`  else []`
		`}}`

Parallel Pattern Tiling 2

- **Pattern interchange:** Reorder nested patterns and split imperfectly nested patterns when intermediate data created is statically known to fit on chip
  - Reordering improves locality and reuse of on-chip memory
  - Reduces number of main memory reads and writes

Example	Strip Mined Patterns	Interchanged Patterns

Matrix multiplication
x: Matrix, size m x p
y: Matrix, size p x n
x.mapRows{row =>
  y.mapCols{col =>
  row.zip(col){(a,b)=>
  a*b
  }.sum
}}
	multiFold(m/b0,n/b1){ii,jj =>
	  xTl = x.copy(b0+ii, b1+jj)
	  ((ii,jj), map(b0,b1){i,j =>
	  multiFold(p/b2){kk =>
	  yTl dy.copy(b1+jj, b2+kk)
	  (0, multiFold(b2){ k =>
	  (0, xTl(i,j)* yTl(j,k))
	  }{(a,b) => a + b})
	  }{(a,b) => a + b}
	  })
	}
		multiFold(m/b0,n/b1){ii,jj =>
		  xTl = x.copy(b0+ii, b1+jj)
		  ((ii,jj), multiFold(p/b2){kk =>
		  yTl = y.copy(b1+jj, b2+kk)
		  (0, map(b0,b1){i,j =>
		  (0, multiFold(b2){ k =>
		  (0, xTl(i,j)* yTl(j,k))
		  }{(a,b) => a + b})
		  })
		  }{(a,b) =>
		  map(b0,b1){i,j =>
		  a(i,j) + b(i,j) }
		  })
		}

Hardware (MaxJ code) Generation

- Parallel patterns mapped to library of hardware templates
- Each template exploits one or more kinds of parallelism or memory access pattern
- Templates coded in MaxJ: Java based hardware generation language from Maxeler

Hardware Templates

Pipe. Exec. Units	Description	IR Construct
Vector	SIMD parallelism	Map over scalars
Reduction tree	Parallel reduction of associative operations	MultiFold over scalars
Parallel FIFO	Buffer ordered outputs of dynamic size	FlatMap over scalars
CAM	Fully associative key-value store	GroupByFold over scalars

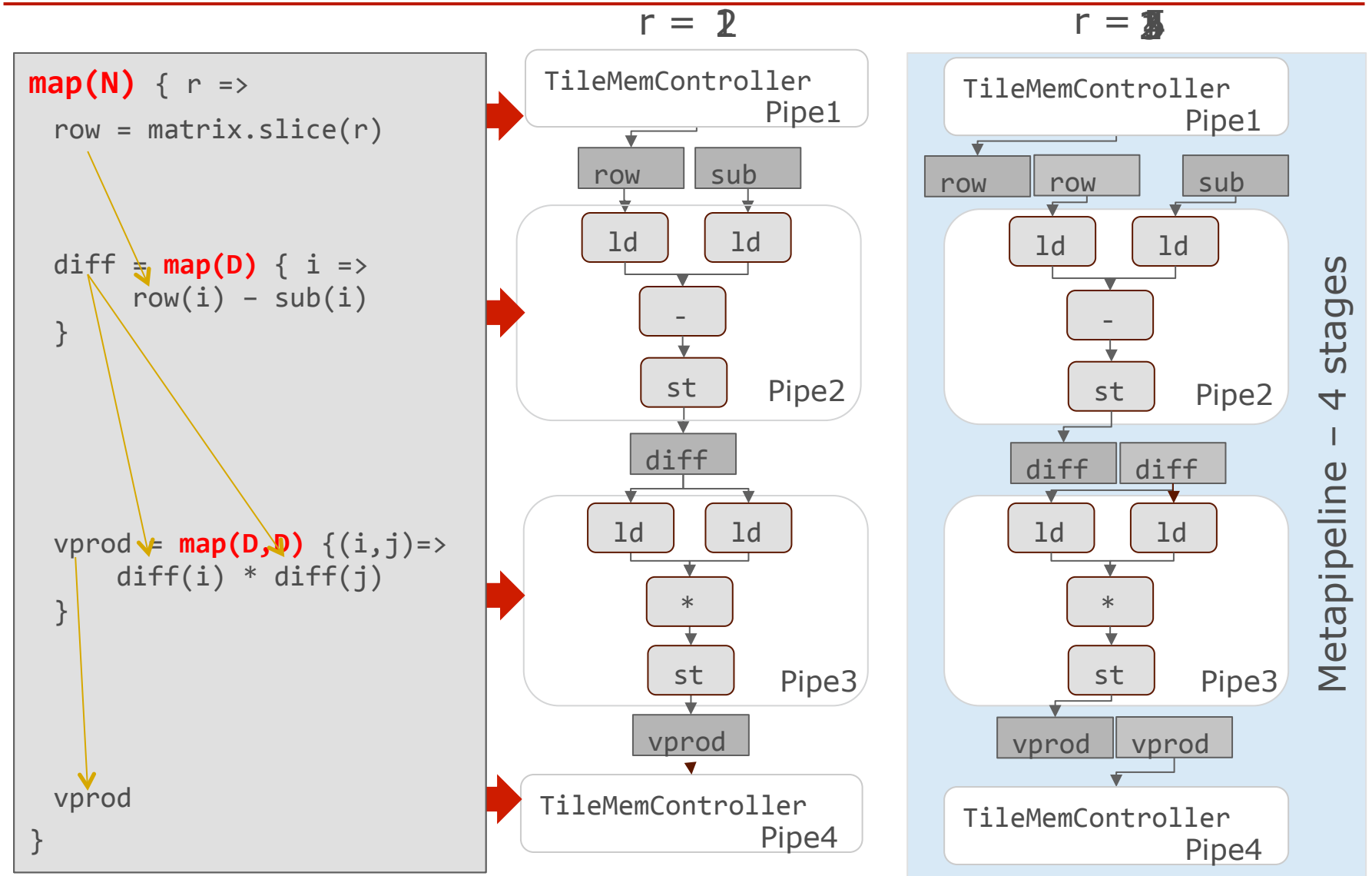
Memories	Description	IR Construct
Buffer	Scratchpad memory	Statically sized array
Double buffer	Buffer coupling two stages in a metapipeline	Metapipeline
Cache	Tagged memory exploits locality in random accesses	Non-affine accesses

Controllers	Description	IR Construct
Sequential	Coordinates sequential execution	Sequential IR node
Parallel	Coordinates parallel execution	Independent IR nodes
Metapipeline	Execute nested parallel patterns in a pipelined fashion	Outer parallel pattern with multiple inner patterns
Tile memory	Fetch tiles of data from off-chip memory	Transformer-inserted array copy

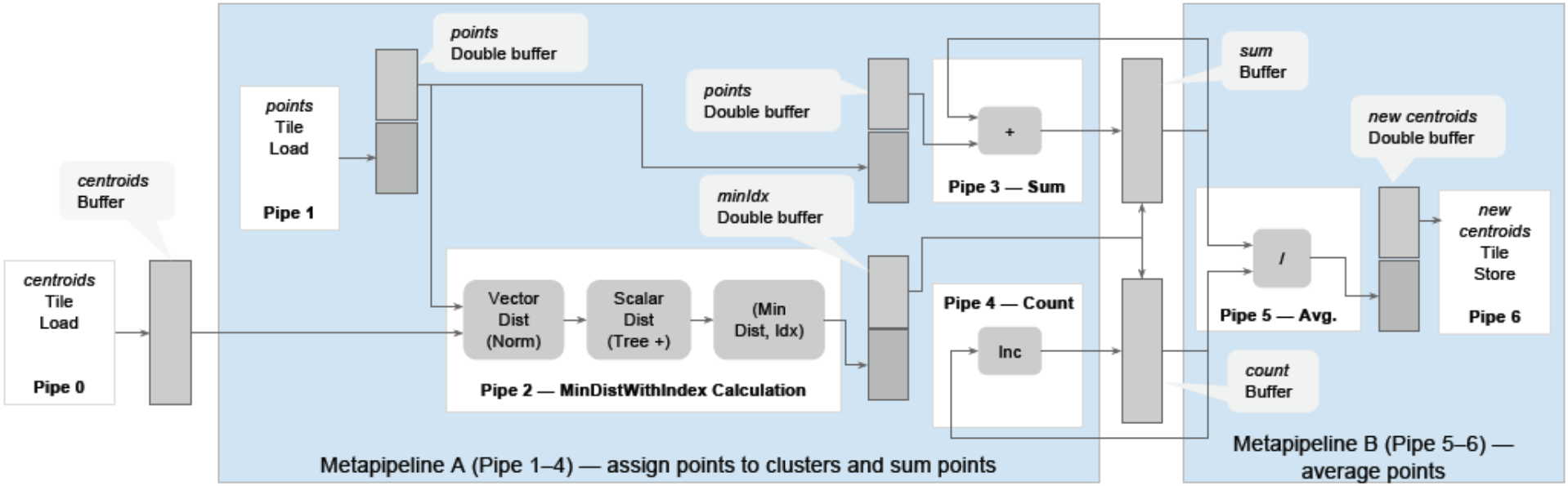
Metapipelining

- Hierarchical pipeline: "pipeline of pipelines"
  - Exploits nested parallelism
- Stages could be other nested patterns or combinational logic
  - Does not require iteration space to be known statically
  - Does not require complete unrolling of inner patterns
- Intermediate data from each stage stored in double buffers
  - No need for lockstep execution

Metapipeline – Simple Example

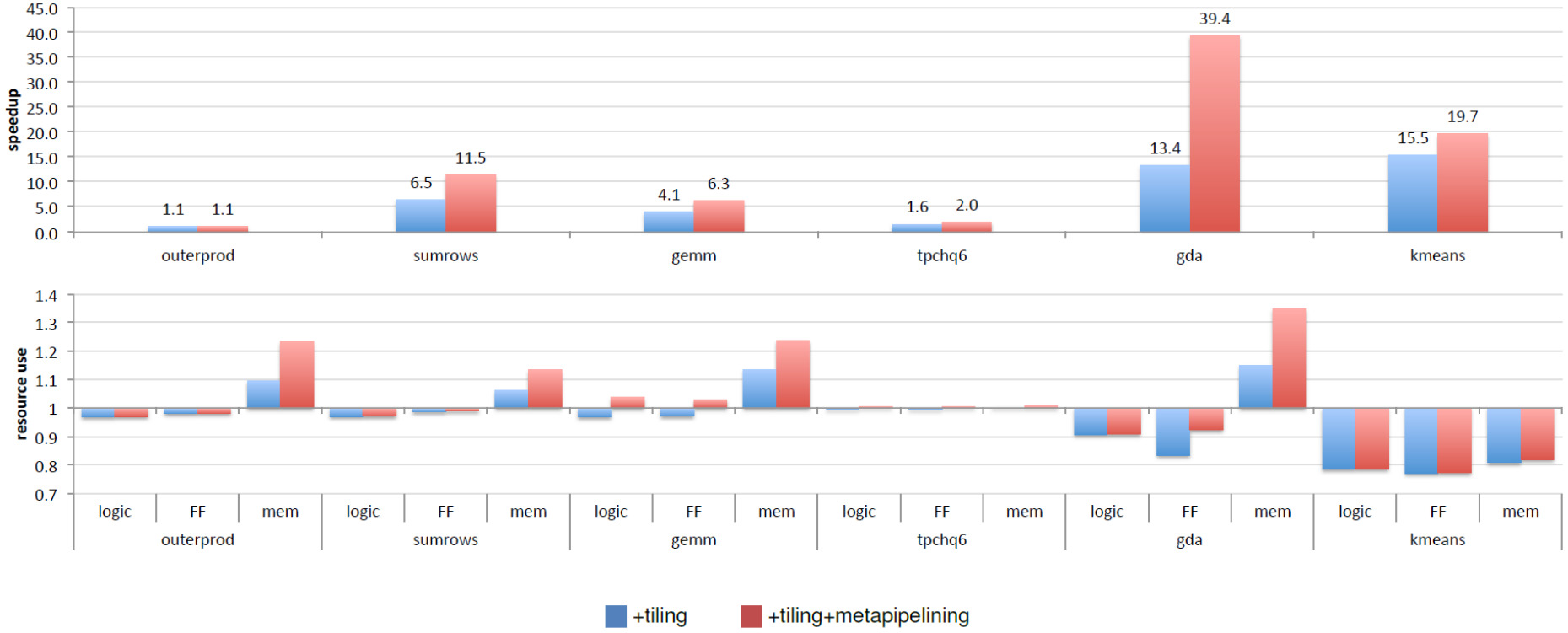


Generated *k*-means Hardware



- High quality hardware design
  - Hardware similar to Hussain et al. *Adapt. HW & Syst. 2011*
    - "FPGA implementation of k-means algorithm for bioinformatics application"
    - Implements a fixed number of clusters and a small input dataset
  - Tiling analysis automatically generates buffers and tile load units to handle arbitrarily sized data
  - Parallelizes across centroids and vectorizes the point distance calculations

Impact of Tiling and Metapipelining



- Base design uses burst access
- Speedup with tiling alone: up to **15.5x**
- Speedup with tiling and metapipelining: up to **39.4x**
- Minimal (often negative!) impact on resource usage
  - Tiled designs have fewer off-chip data loaders and storers

Summary

- In the age of heterogeneous architectures
  - Power limited computing ⇒ parallelism and accelerators
- Need parallelism and acceleration for the masses
  - DSLs let programmers operate at high-levels of abstraction
  - Need one DSL for all architectures
  - Semantic information enables compiler to do coarse-grained domain-specific optimization and translation
- Need a parallelism and accelerator friendly IR
  - Parallel pattern IR structures computation and data
  - Allows aggressive parallelism and locality optimizations through transformations
  - Provides efficient mapping to heterogeneous architectures
- DSL tools for FPGAs need to be improved
  - Better performance prediction
  - More optimization
  - Shorter compile times (place and route)

Big Data Analytics
In the Age of Accelerators

- Power
- Performance
- Productivity
- Portability

Accelerators
(GPU, FPGA, …)

Parallel Patterns

High Performance DSLs
(OptiML, OptiQL, …)

Colaborators & Funding

- Faculty
  - Pat Hanrahan
  - Martin Odersky (EPFL)
  - Chris Ré
  - Tiark Rompf (Purdue/EPFL)
- PhD students
  - Chris Aberger
  - Kevin Brown
  - Hassan Chafi
  - Zach DeVito
  - Chris De Sa
  - Nithin George (EPFL)
  - David Koeplinger
  - HyoukJoong Lee
  - Victoria Popic
  - Raghu Prabhakar
  - Aleksander Prokopec (EPFL)
  - Vojin Jovanovic (EPFL)
  - Vera Salvisberg (EPFL)
  - Arvind Sujeeth
- Funding
  - PPL : Oracle Labs, Nvidia, Intel, AMD, Huawei, SAP
  - NSF
  - DARPA

Extra Slides

Comparing Programming Models of Recent Systems For Data ANALYTICs

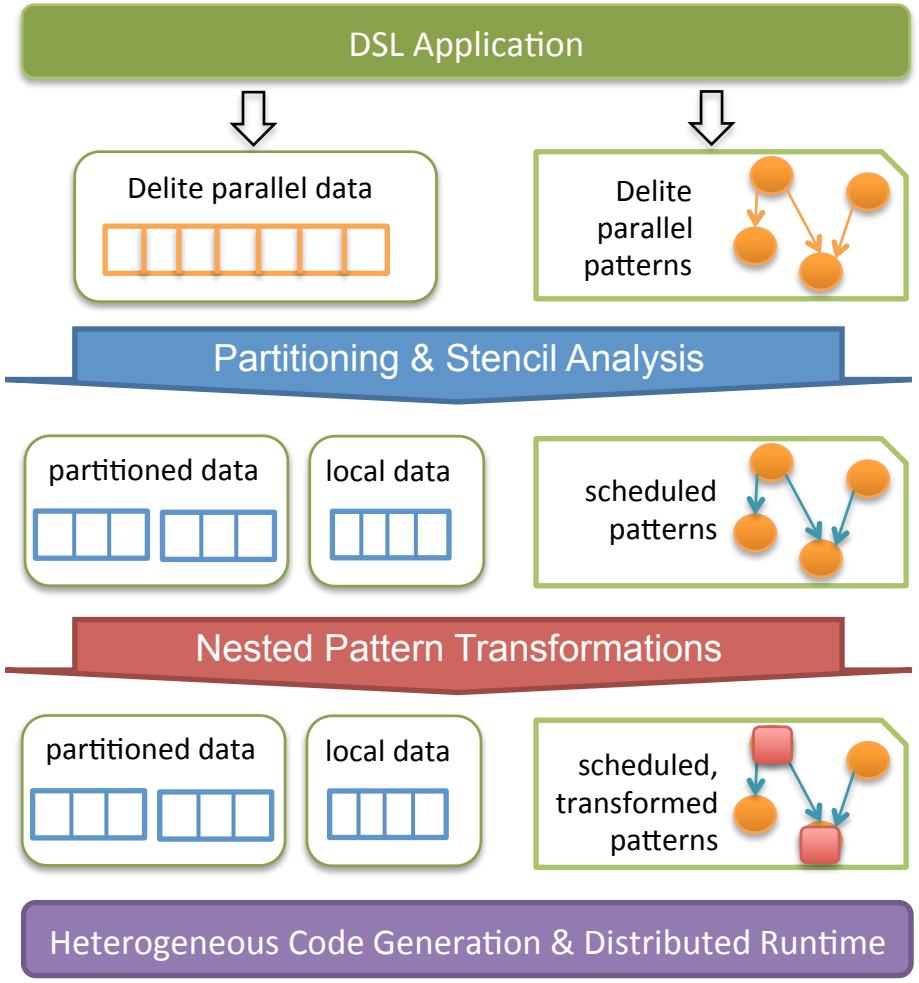
System	Programming Model Features					Supported Hardware				
	Rich Data Parallelism	Nested Prog.	Nested Parallelism	Multiple Collections	Random Reads	Multi-core	NUMA	Clusters	GPU	FPGA
MapReduce								✓		
DryadLINQ	✓	✓						✓		
Thrust					✓				✓	
Scala Collections	✓	✓	✓	✓	✓	✓				
Delite	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Spark	✓					✓		✓		
Lime		✓	✓	✓	✓	✓			✓	✓
PowerGraph					✓	✓		✓		
Dandelion	✓	✓				✓		✓	✓	

Frameworks are listed in chronological order

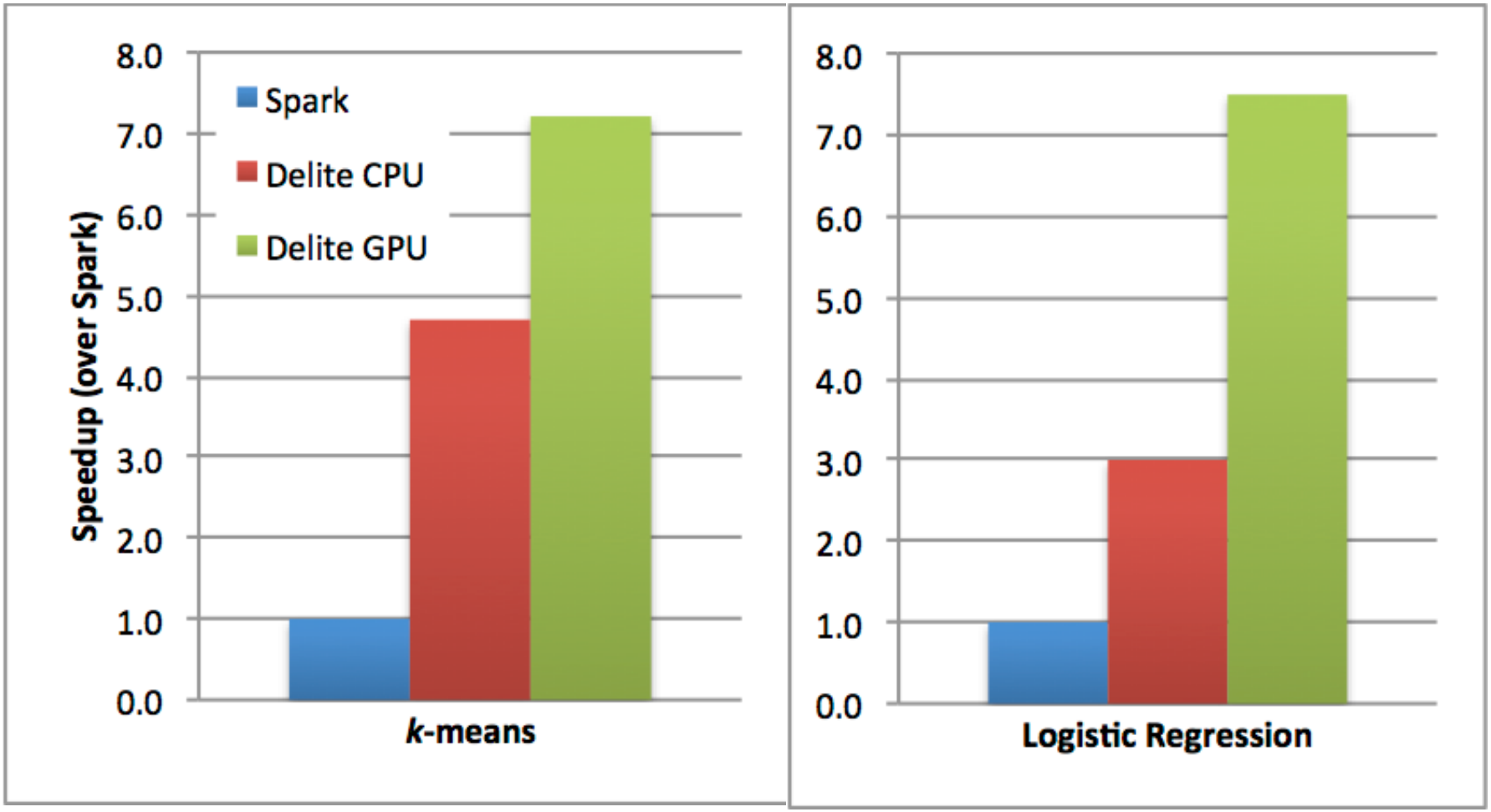
Requirement: expressive programing model and support for all platforms

Distributed Heterogeneous Execution

- Separate Memory Regions
  - NUMA
  - Clusters
  - FPGAs
- Partitioning Analysis
  - Multidimensional arrays
  - Decide which data structures / parallel ops to partition across abstract memory regions
- Nested Pattern Transformations
  - Optimize patterns for distributed and heterogeneous architectures

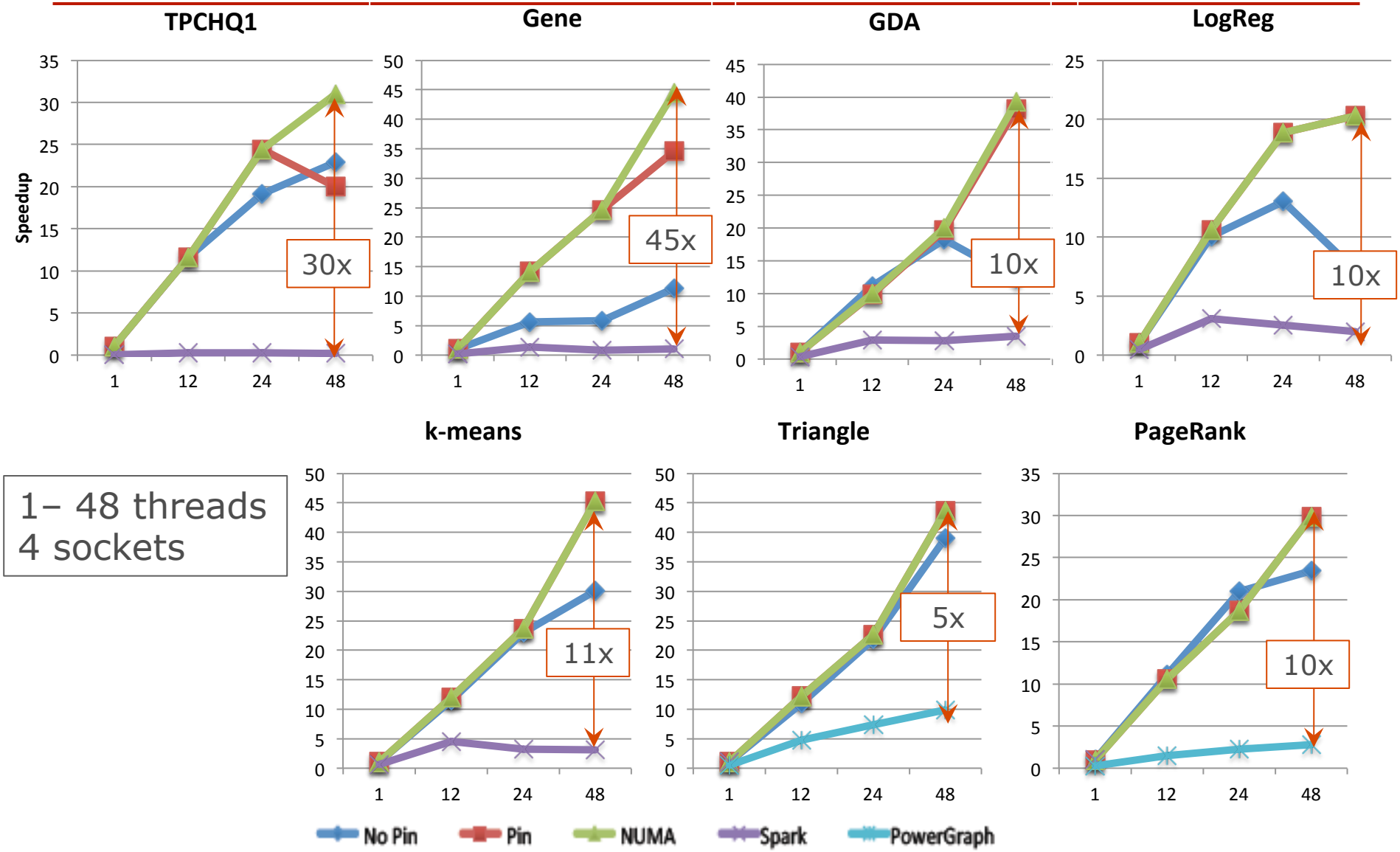


OptiML on Heterogeneous Cluster



4 node local cluster: 3.4 GB dataset

Multi-socket NUMA Performance



Parallel Pattern Language

- Implemented a data-parallel language that supports parallel patterns
- Structured computations and data structures
  - Computations : map, zipwith, foreach, filter, reduce, groupby, …
  - Data structures: scalars, array, structs
- Example application: PageRank

```
Graph.nodes map { n =>
    nbrsWeights = n.nbrs map { w =>
        getPrevPageRank(w) / w.degree
    }
    sumWeights = nbrsWeights reduce { (a,b) => a + b }
    ((1 - damp) / numNodes + damp * sumWeights
}
```