Compiler and runtime support for enabling generalized reduction computations on heterogeneous parallel configurations

10M37255 Nguyen Toan
Matsuoka Laboratory

Reference

- Compiler and runtime support for enabling generalized reduction computations on heterogeneous parallel configurations
  - **Vignesh T.Ravi**, Wenjing Ma, David Chiu, Gagan Agrawal
  - International Conference on Supercomputing (ICS) 2010

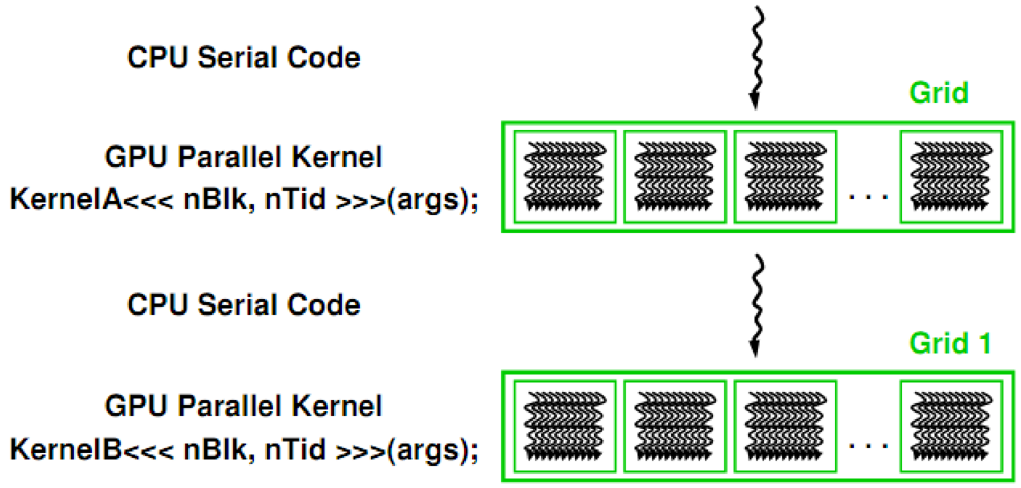
Outline

GPU

- Graphics Processing Unit
  - Dedicated processor for rendering graphics
  - Massively multithreaded manycore chip
    - Hundreds of scalar processors
    - Tens of thousands of concurrent threads
    - 1 TFLOP peak performance
    - Fine-grained data-parallel computation
  - Manyfold speedups can be achieved on GPUs

CUDA

- Compute Unified Device Architecture
- A scalable parallel programming model
  - Minimal extensions to C/C++
  - Heterogeneous serial-parallel programming model



A typical CUDA program

- Initialize some data on the host memory (CPU)
- Copy data from host memory to device memory (GPU)
- Execute GPU kernel functions
- Copy the computed results back from device to host memory

Background

- Heterogeneous computing platforms are increasing
- Challenges: how to exploit the aggregate computing power of multi-core CPUs and GPUs effectively
  - Programmability
  - Performance
  - Work distribution

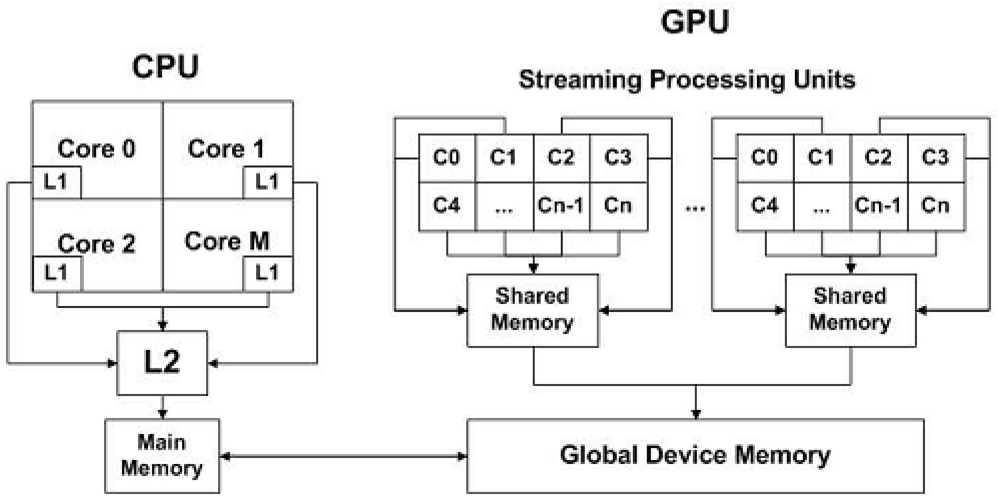


Figure 1: A Popular Heterogeneous Computing Platform

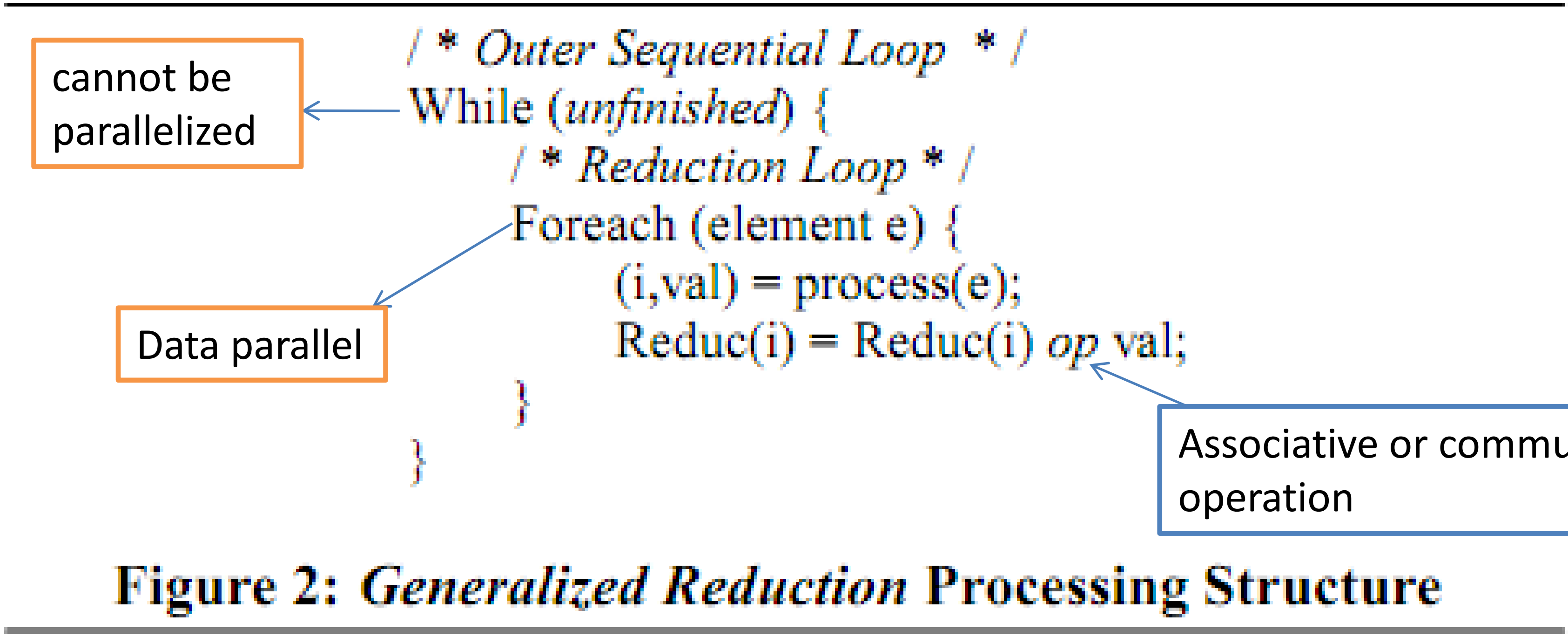
Purpose

- Develop a compiler and runtime support for heterogeneous multiprocessors
  - Focus on *generalized reduction computations*
  - Address three challenges:
    - Programmability
      - Applications which follow this structure can be written in a sequential C interface + some annotations
    - Performance
      - Show significant speedups over CPU-only and GPU-only results
    - Work distribution
      - Propose an effective dynamic work distribution scheme

Outline

- Introduction to GPU & CUDA
- Background
- Approach and System Design
- Language support and Code generation
- Experimental Results
- Related Work
- Conclusions
- Comments

Generalized Reduction Structure



```
/ * Outer Sequential Loop * /
While (unfinished) {
    / * Reduction Loop * /
    Foreach (element e) {
        (i,val) = process(e);
        Reduc(i) = Reduc(i) op val;
    }
}
```

Figure 2: *Generalized Reduction* Processing Structure

cannot be parallelized

Data parallel

Associative or commutative operation

Parallelization approach for multicore CPUs

- Divide data instances among processing threads
- The reduction object is shared among all processing threads
- Privatize the reduction object
  - Avoid data races
  - Each thread has its own copy of the reduction object
- Middleware system
  - Not discussed in this paper

GPU computing for Generalized Reductions

- Reduction objects are shared by all GPU threads

- Replicating reduction object for each thread
  - Avoid potential data races

- Data block is divided into small blocks
  - Each thread processes one block at a time
  - Implementing a reduction computation requires:
    - Read a data block
    - Compute the reduction object updates based on the data instance
    - Write back the reduction object update
  - Results from each thread are merged to form the final result

System Design

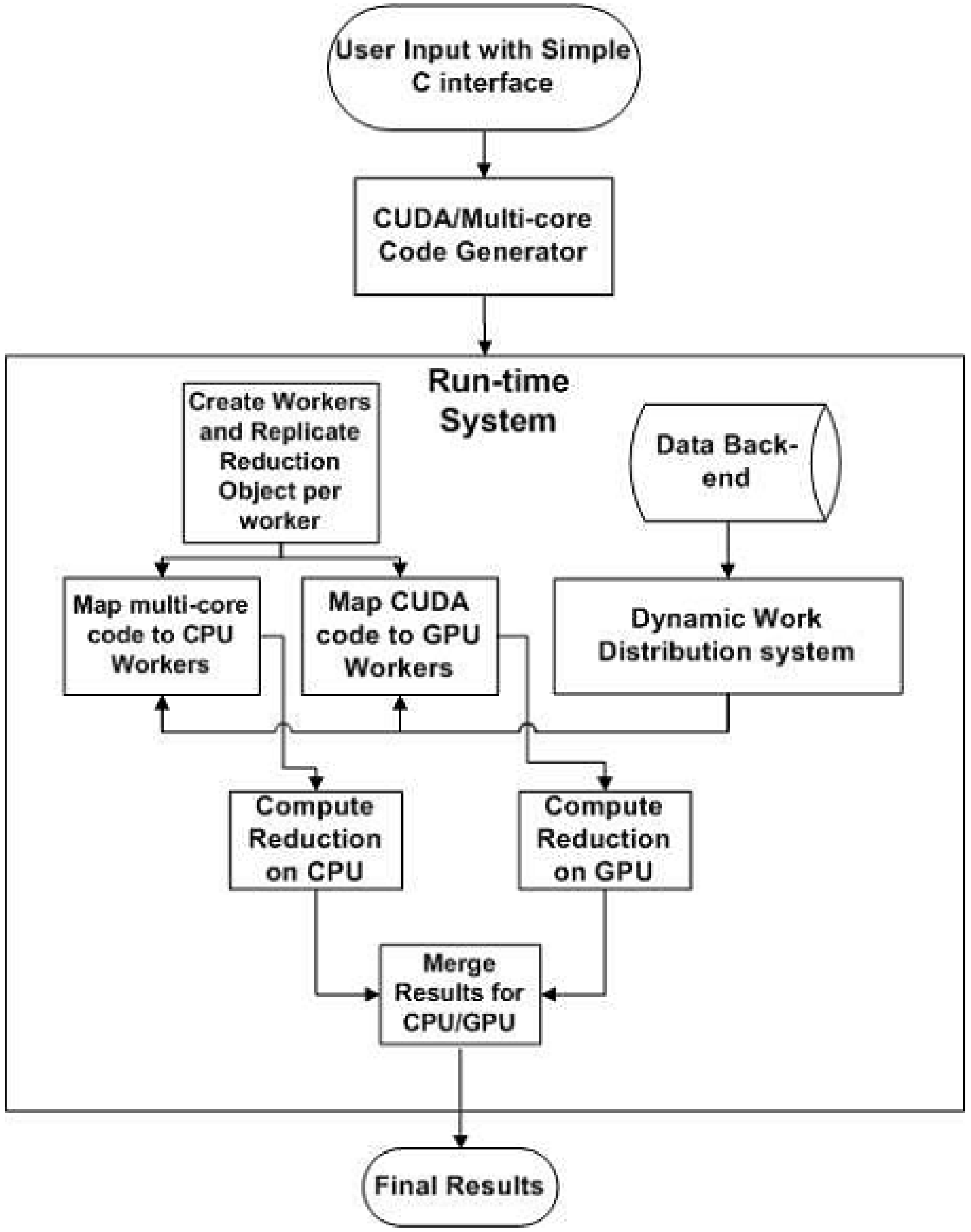

User Input with Simple C interface
CUDA/Multi-core Code Generator
Run-time System
Create Workers and Replicate Reduction Object per worker
Data Back-end
Map multi-core code to CPU Workers
Map CUDA code to GPU Workers
Dynamic Work Distribution system
Compute Reduction on CPU
Compute Reduction on GPU
Merge Results for CPU/GPU
Final Results

Figure 3: High-level Architecture of the System

Dynamic Work Distribution

- Dynamic distribution schemes have two possible approaches
  - Work sharing
    - The work is enqueued in a globally shared work list
    - An idle processor consumes work from the list
  - Work stealing
    - A private work list is maintained with each processor
    - An idle processor searched other busy processors for the work it could steal

- This work choose a work sharing approach
  - For data parallel applications
  - High latency in communication with GPU
  - GPU memory size limitation

Uniform-chunk distribution scheme

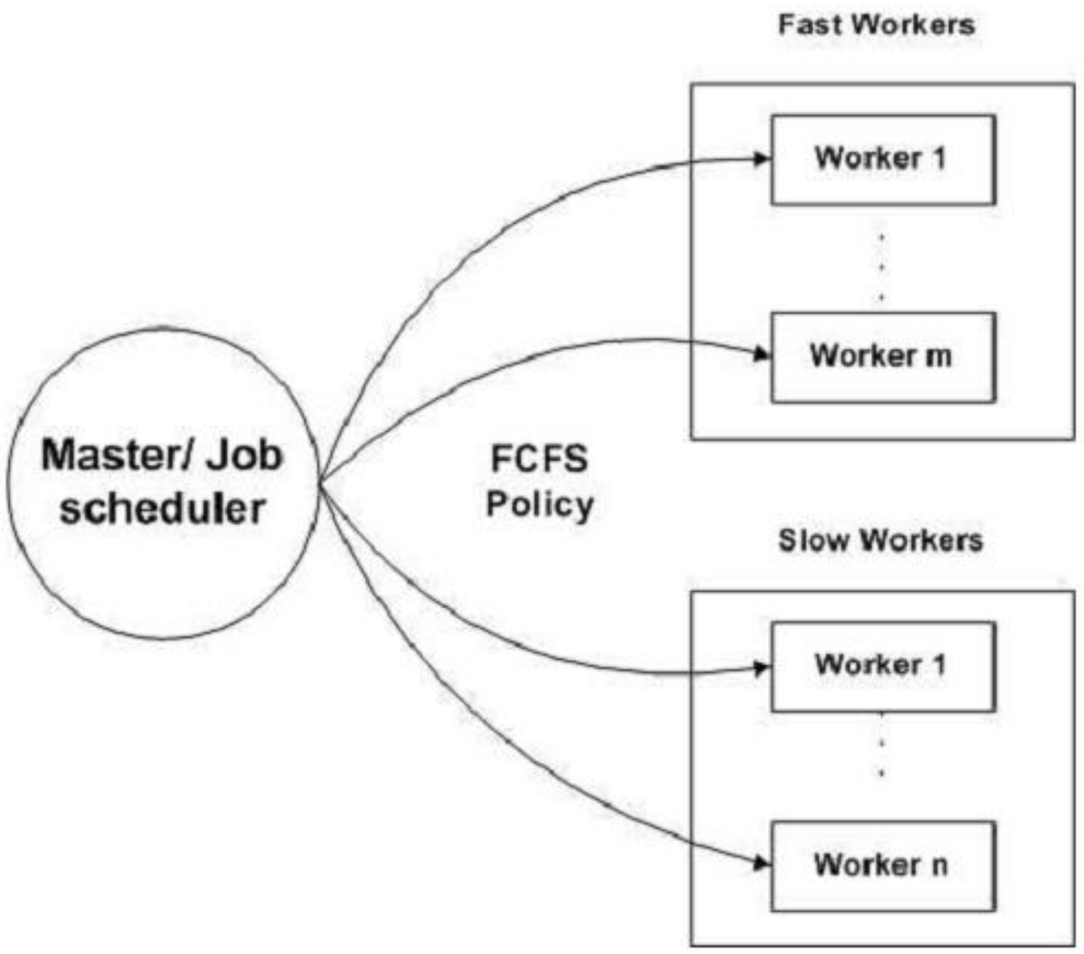


Figure 4: Uniform Chunk-Size Distribution Scheme

Non-uniform-chunk distribution scheme

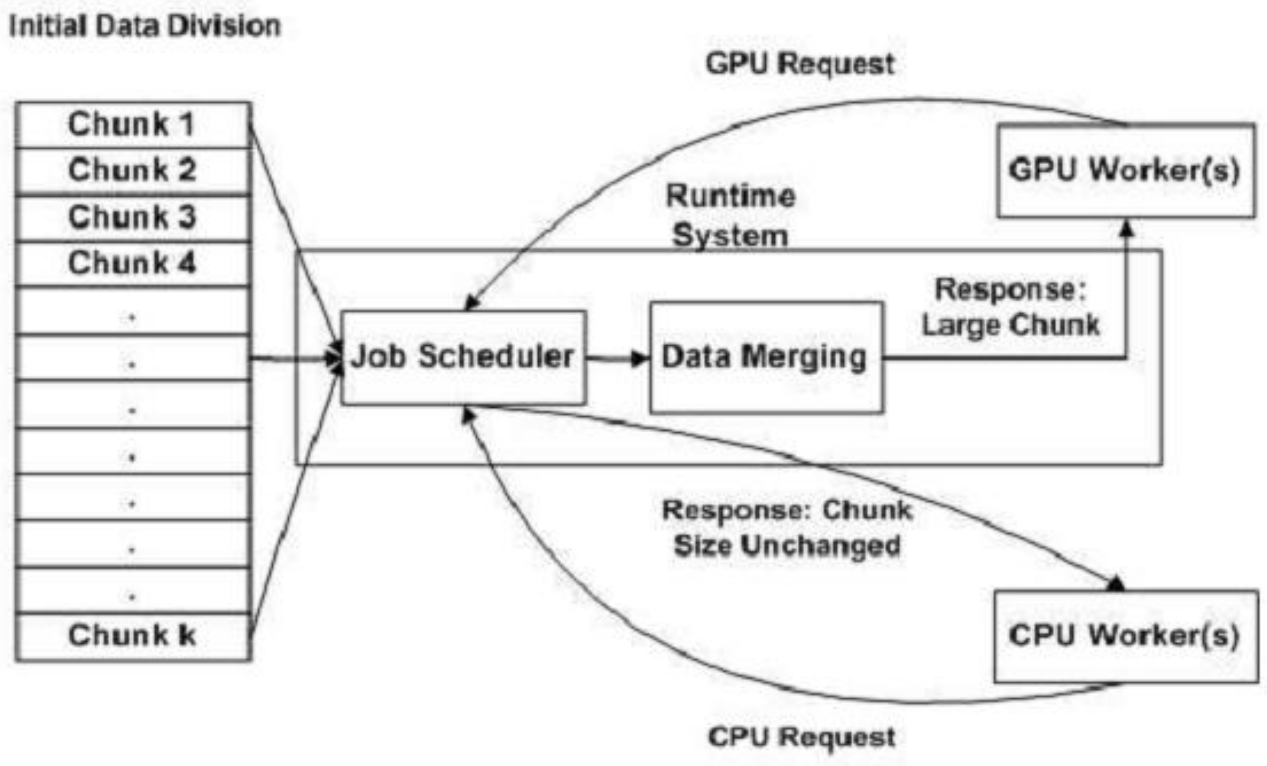


Figure 5: Non-Uniform Chunk-Size Distribution Scheme

Outline

- Introduction to GPU & CUDA
- Background
- Approach and System Design
- Language support and Code generation
- Experimental Results
- Related Work
- Conclusions
- Comments

User Input

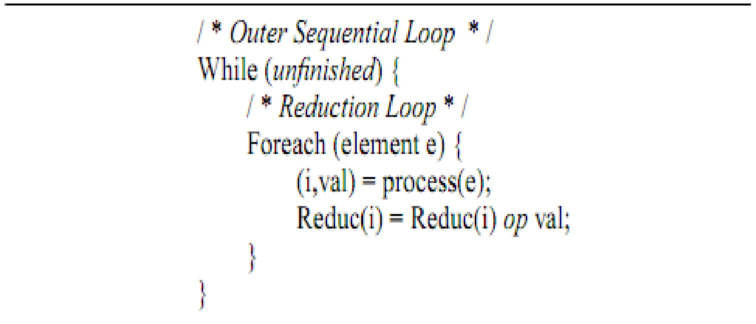
- The user has to identify the generalized reduction structure and uses *reduction* functions to express an application
- Information required in each function
  - Variable list for computing
    - Variable format: `name, type, size[value]`
    - Example: `update_centers float* 5 K`
  - Sequential reduction function
    - A reduction loop can be expresses as a function

Program Analysis (1/2)

- Variable Analysis and Classification
  - Variables are classified into: `Input`,`Output`, `Temporary`
  - Variable analysis
    - Use LLVM to generate intermediate representations (IR) of reduction functions
    - Apply Andersen's points-to analysis to get the set of pointers `points-to`
    - Trace the entire function in IR form and when encounter a `store` operation, classify variables:
      - If the destination of `store` is a `points-to` set of any varibale in the argument's list and the source is not in the same set, it is an `output`
      - Otherwise, all variables in the argument list are `input`
      - Not in the argument list are `temporary`

Program Analysis (2/2)

- Code Analysis
  - Extract the reduction objects with their combination operation
    - `output` variables are reduction objects
    - Identify operator being used for updateing reduction object variables
  - Extract the parallel loops
    - Extract loop variables in `foreach` loop
      - `num_iter:` number of iterations
      - *loop variable:* accessed only with an affine subscript of the loop index



```
/ * Outer Sequential Loop * /
While (unfinished) {
      / * Reduction Loop * /
      Foreach (element e) {
            (i,val) = process(e);
            Reduc(i) = Reduc(i) op val;
      }
}
```

Figure 2: ***Generalized Reduction*** **Processing Structure**

Code Generation for CUDA (1/2)

- Generating Host function
  - Declare and Copy
    - Allocate device memory for variables, except `temporary`, used in kernel function
    - Disjoint portions of `loop` variables are distributed across threads
    - Other read/write variables are updated my multiple threads simultaneously and are replicated for each thread
  - Compute
    - Execution configuration (thread/block config.) is defined
    - Invoke the kernel function
  - Copy Updates
    - Results from kernel computation are copied back to host
    - `Output` variables from each block are combined to produce the final result

Code Generation for CUDA (2/2)

- Generating Kernel Code
  - Divide the loop to be parallelized by # of thread blocks and # of threads in a block
  - Regroup the array index
    - Ex: `data[i]`→`data[i+index_n]`
      - `index_n`: offset for each thread in the entire grid
  - Optimize the use of GPU's shared memory
    - Shared memory is allocated in increasing order to capitalize fast accesses of GPU's shared memory

  - `combine()` performs merging all threads' results
  - `_synchthreads()` provides synchronization

Outline

- Introduction to GPU & CUDA
- Background
- Approach and System Design
- Language support and Code generation
- Experimental Results
- Related Work
- Conclusions
- Comments

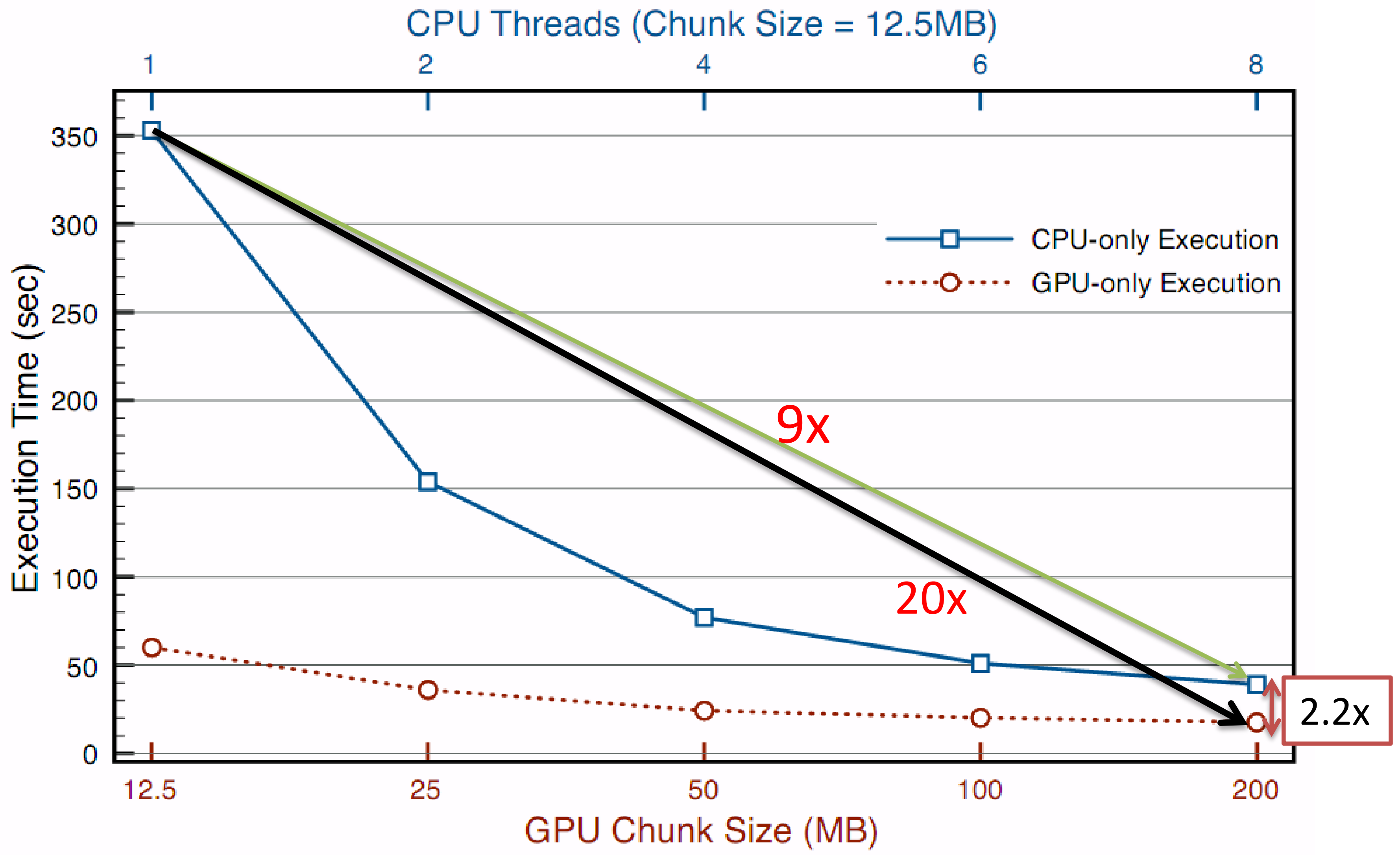
Experiment Environment

Machine	AMD Opteron 8350
OS	Redhat Linux
CPU	8 CPU cores, 16 GB main memory
GPU	GeForce 9800 GTX, 512 MB memory

Target applications

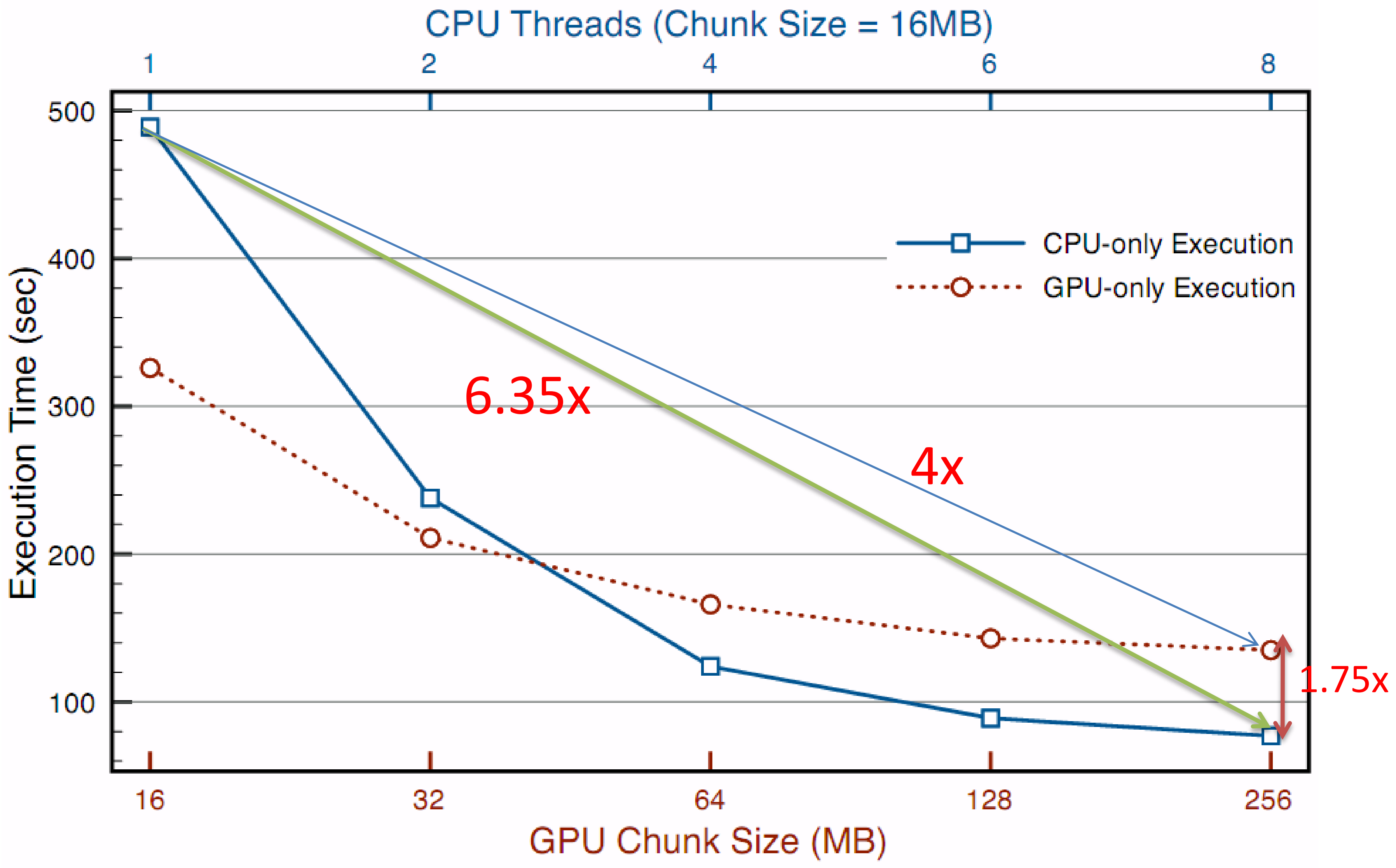
Application	K-means	PCA
Data Size	6.4 GB	8.5 GB
	# of clusters : 125	# of columns: 64

- Goals
  - Evaluate performance of a multi-core CPU and GPU independently and study how chunk-size impacts performance
  - Study performance gain while simultaneously exploiting both multi-core CPU and GPU
  - Provide elaborate evaluation with two dynamic distribution schemes



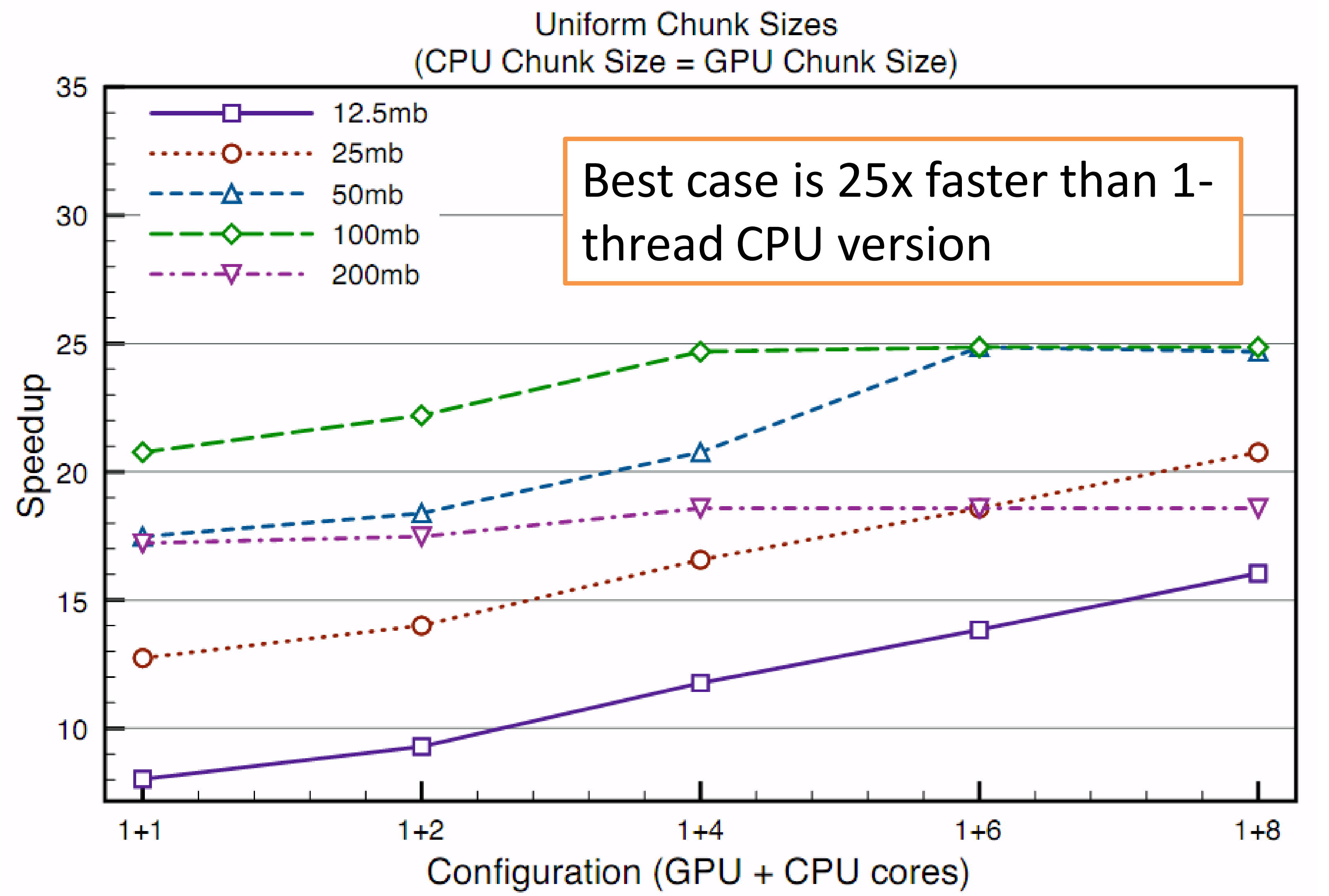
GPU-only version is 2.2x faster than 8-core CPU-only version

Figure 7: Scalability of K-Means with CPU-only and GPU-only



8-thread CPU-only version is 1.75x faster than the GPU-only version

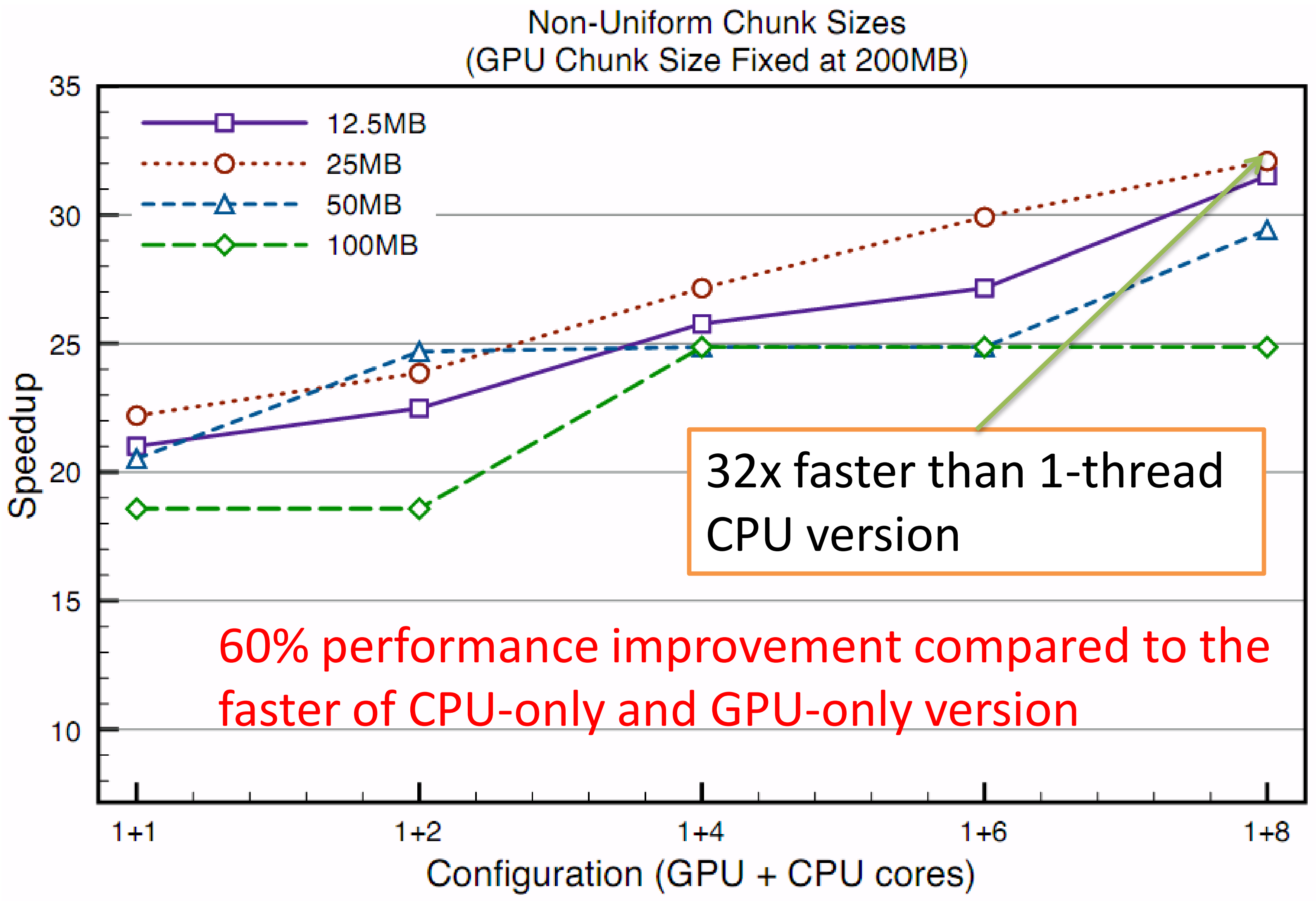
Figure 8: Scalability of PCA with CPU-only and GPU-only



Best case is 25x faster than 1-thread CPU version

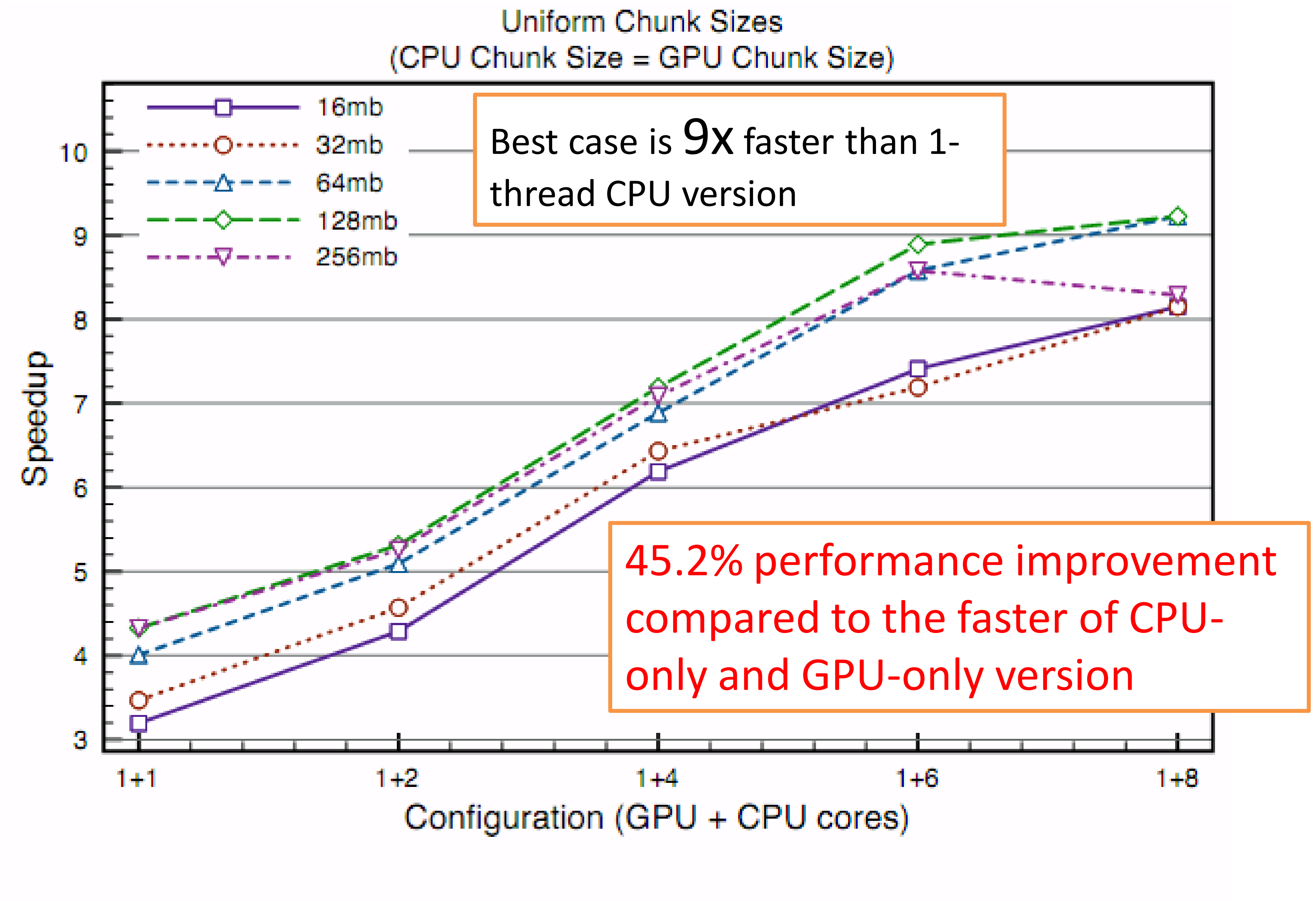
23.9% performance improvement compared to the faster of CPU-only and GPU-only version

Figure 9: K-Means Using Heterogeneous Version with Uniform Chunk Size



36% performance improvement over UCS

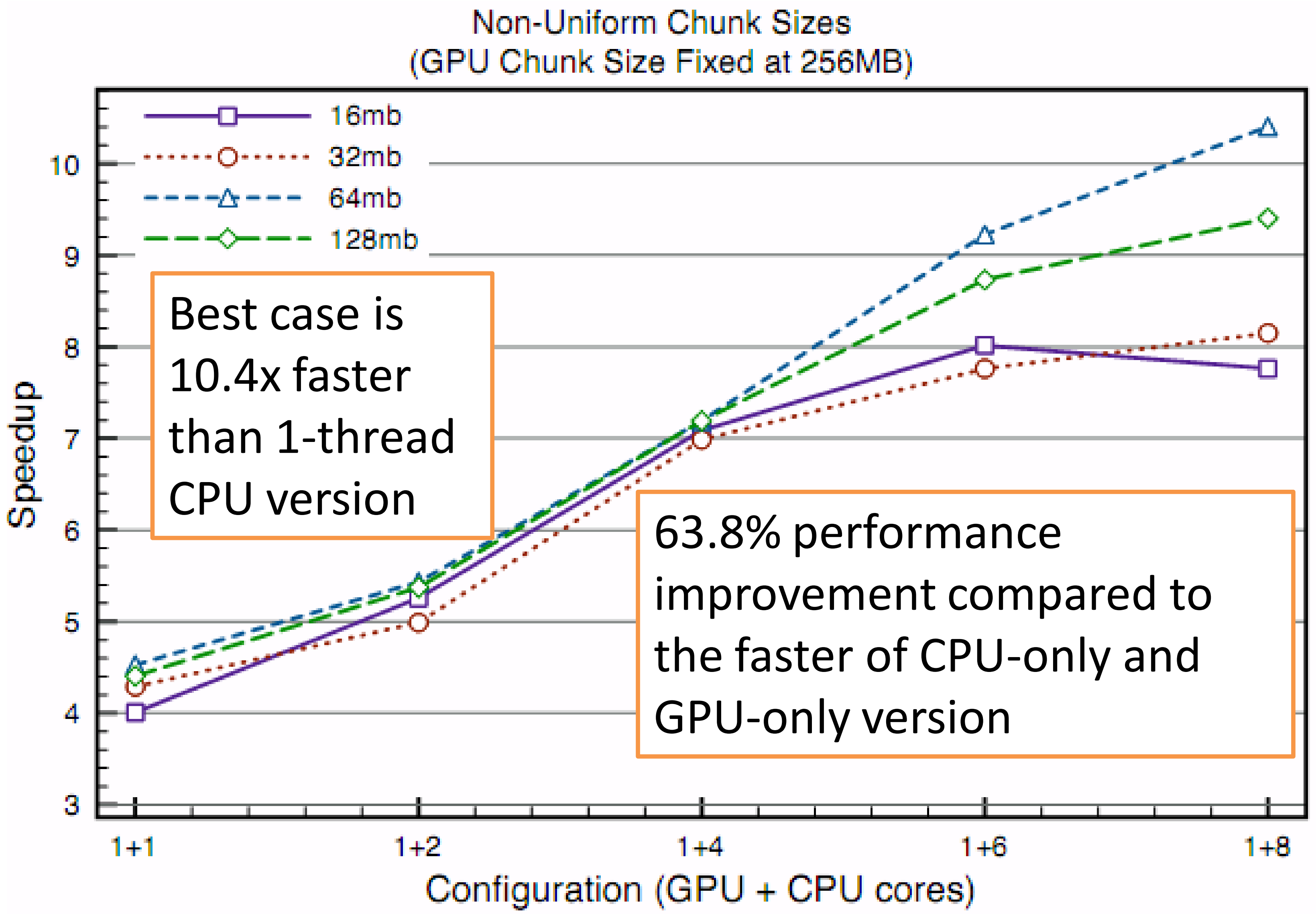
Figure 10: K-Means Using Heterogeneous Version with Non-Uniform Chunk Size



Best case is 9x faster than 1-thread CPU version

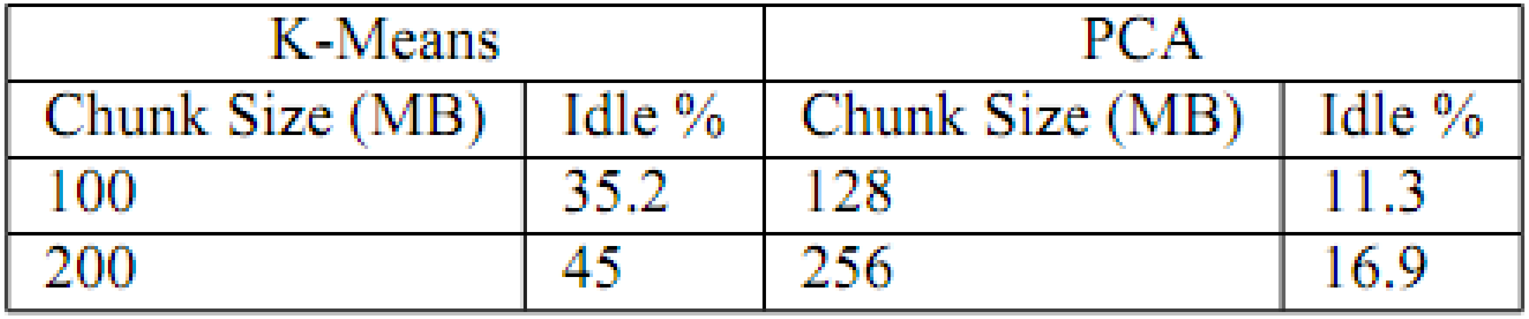
45.2% performance improvement compared to the faster of CPU-only and GPU-only version

Figure 11: PCA Using Heterogeneous Version with Uniform Chunk Size



18.6% performance improvement over UCS

Figure 12: PCA Using Heterogeneous Version with Non-Uniform Chunk Size



K-Means		PCA	
Chunk Size (MB)	Idle %	Chunk Size (MB)	Idle %
100	35.2	128	11.3
200	45	256	16.9

Table 1: % Idle with Uniform Chunk Size

- Consider 1+8 thread configuration with large chunk size
  - Contention between CPU threads is highest

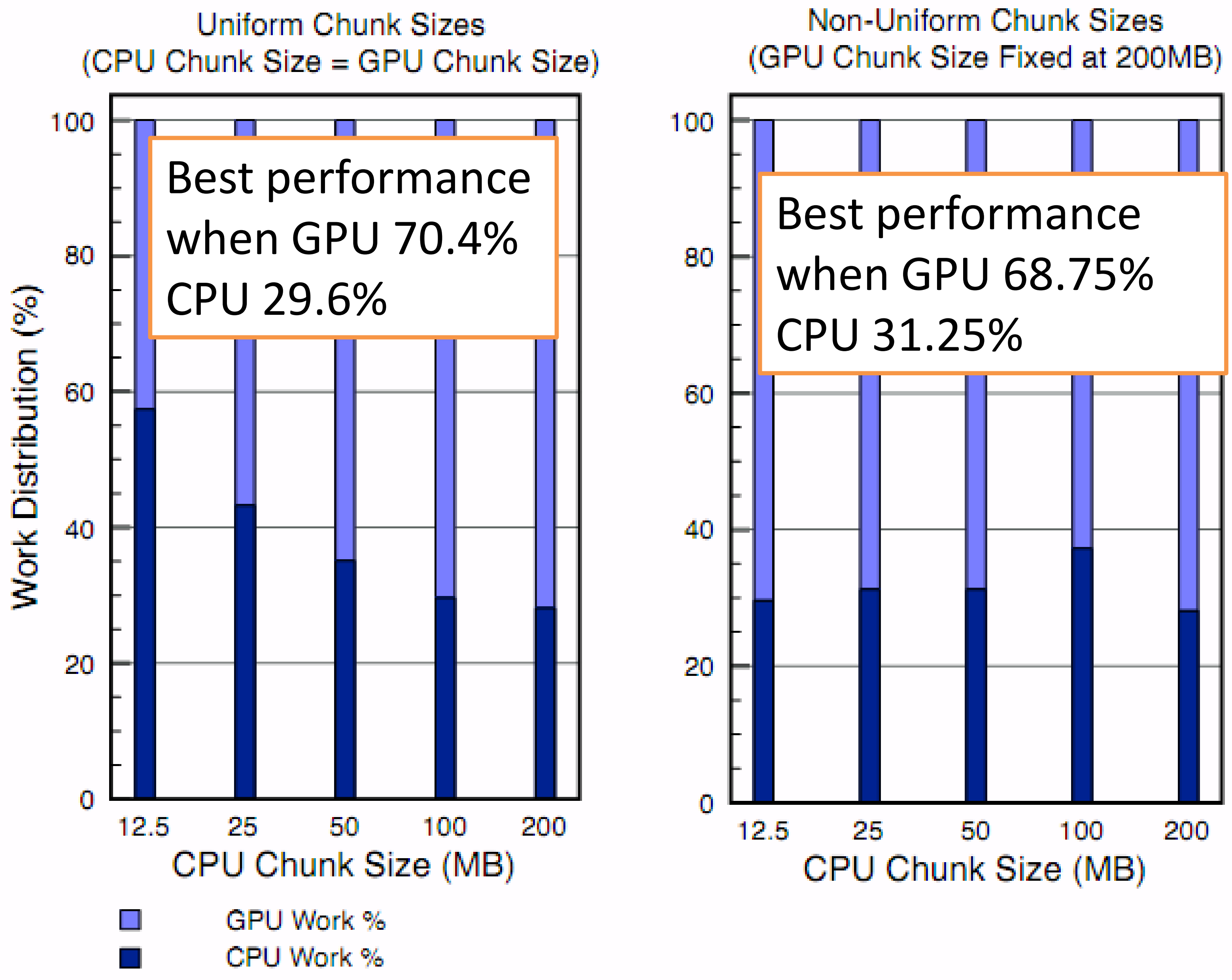


Figure 13: Work Distribution (K-Means)

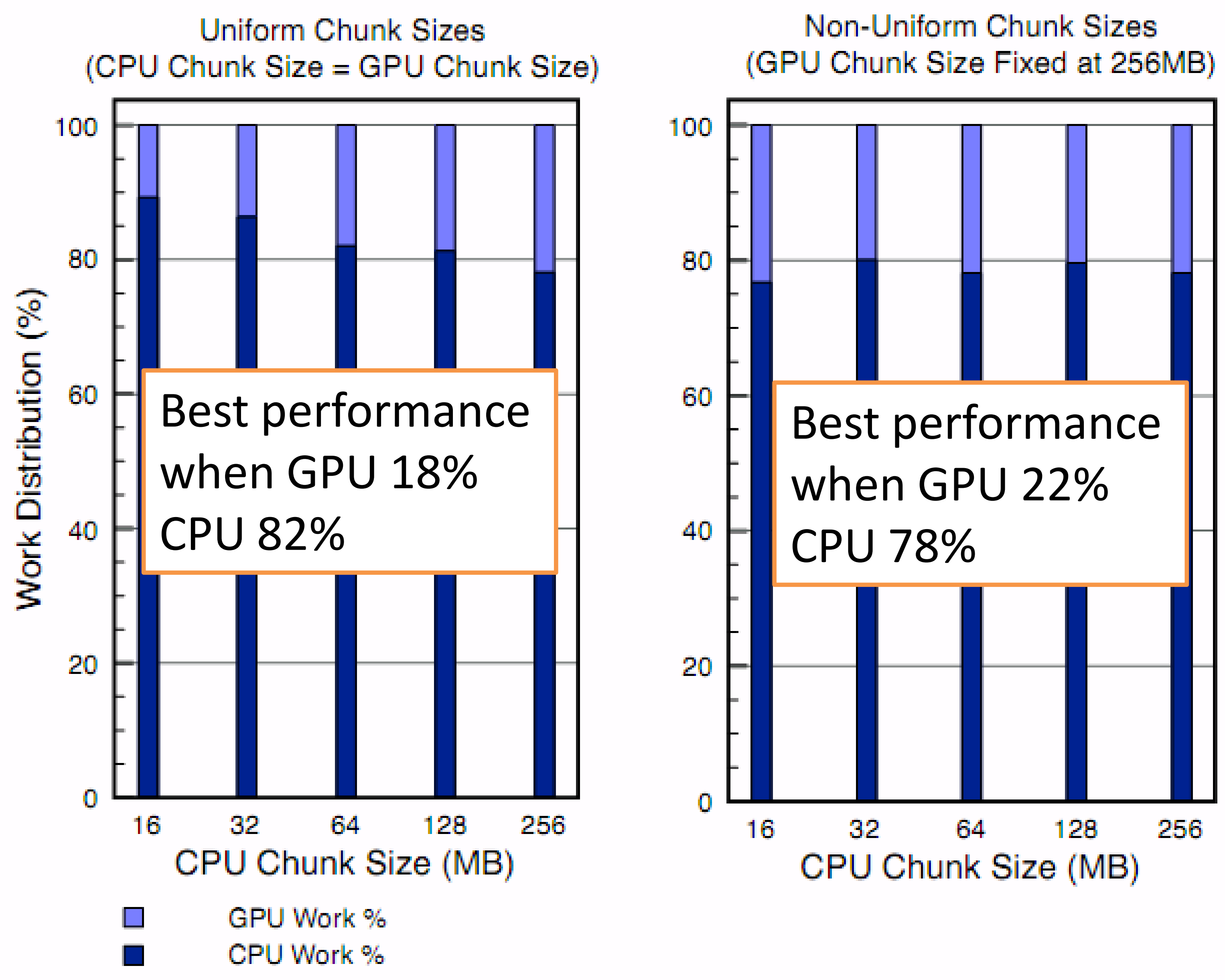


Figure 14: Work Distribution (PCA)

Related Work

- OpenCL
  - Enables data parallel programming on CPU, GPU and any other device
- Other efforts
  - Exochi, Venkatasubramanian, Kuzman, Helios,Qilin
- Diffrences from existing works
  - Map sequential code to data parallel code
  - Support very high-level programming API
  - Improve performance through a dynamic work distribution scheme
- In last few years
  - CUDA-lite, Pycuda
  - Translating OpenMP to CUDA
  - Automatic CUDA code generation

Conclusions

- Developed compiler and runtime support targeting generalized reduction computations on a heterogeneous system
- Discuss two dynamic work distribution schemes that can effectively improve performance
- Gain considerable speedups on K-means and PCA applications

Comments

- The performance speedup is quite impressive
- The specific input for generalized reduction applications may be troublesome for users
- No user guide for this framework
- Little detail on the implementation