Out-of-core GPU Memory Management for MapReduce-based Large-scale Graph Processing

Koichi Shirahata, Hitoshi Sato, Satoshi Matsuoka

Tokyo Institute of Technology
CREST, Japan Science and Technology Agency

Fast Large-scale Graph Processing using HPC

- Emergence of large-scale graphs
  - SNS, road network, smart grid, etc.
  - millions to trillions of vertices/edges
    - e.g.) a social friend network: 1.31 billion vertices, 170 billion edges
  - Need for fast graph processing on supercomputers

- Graph processing on supercomputers
  - A wide range of applications is accelerated using supercomputers (e.g. physical simulations)
  - Graph processing is also considered an important application on supercomputers
    - Graph500 benchmark is started from 2010

Large-scale Graph Processing on Heterogeneous Supercomputers

- GPU-based heterogeneous supercomputers
  - e.g.) Titan, TSUBAME2.5
  - High computing and memory performance

→ Fast large-scale graph processing on heterogeneous supercomputers

- **Problem: GPU memory capacity limits scalable large-scale graph processing**
  - Large-scale data, while GPU memory capacity is small
    - e.g.) TSUBAME2.5: GPU 6GB (x3), CPU 54GB

Contributions

- **Out-of-core GPU memory management for MapReduce-based graph processing**
  - Introduce out-of-core GPU data management techniques for GPU-MapReduce-based large-scale graph processing
  - Implement out-of-core GPU sorting
    - Incorporated in our GPU-MapReduce implementation
  - Investigate the balance of scale-up and scale-out approaches
    - Changing the number of GPUs per node for processing graph data

Performance on TSUBAME2.5 and TSUBAME-KFC

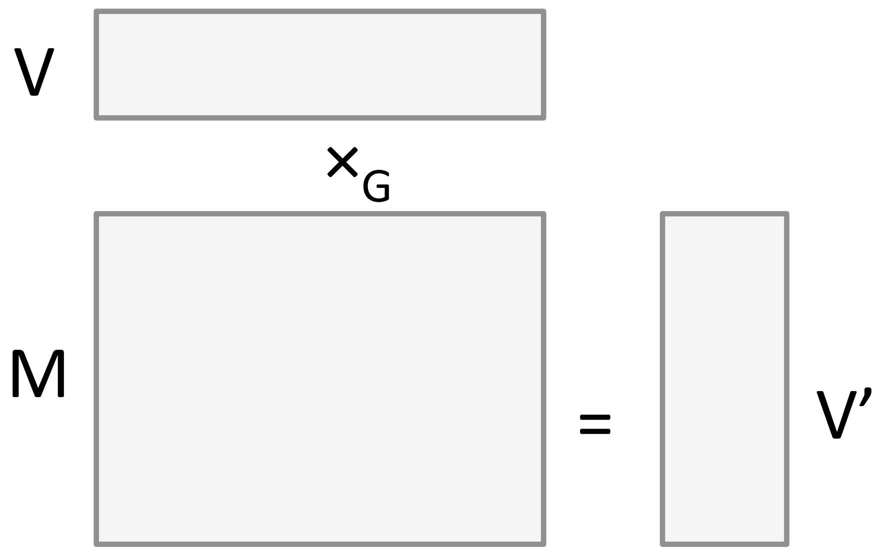
- **2.10x speedup than CPUs on 3072 GPUs**
- **1.71x power efficiency by scale-up strategy**

Table of Contents

Large graph processing algorithm GIM-V

- **Generalized Iterative Matrix-Vector multiplication**[*1]
  - Graph applications are implemented by defining 3 functions
    - PageRank, Random Walk with Restart, Connected Components etc.
  - $v' = M \times_G v$ where

$$v'_i = \text{Assign}(v_j, \text{CombineAll}_j(\{x_j \mid j = 1..n, x_j = \text{Combine2}(m_{i,j}, v_j)\})) \quad (i = 1..n)$$

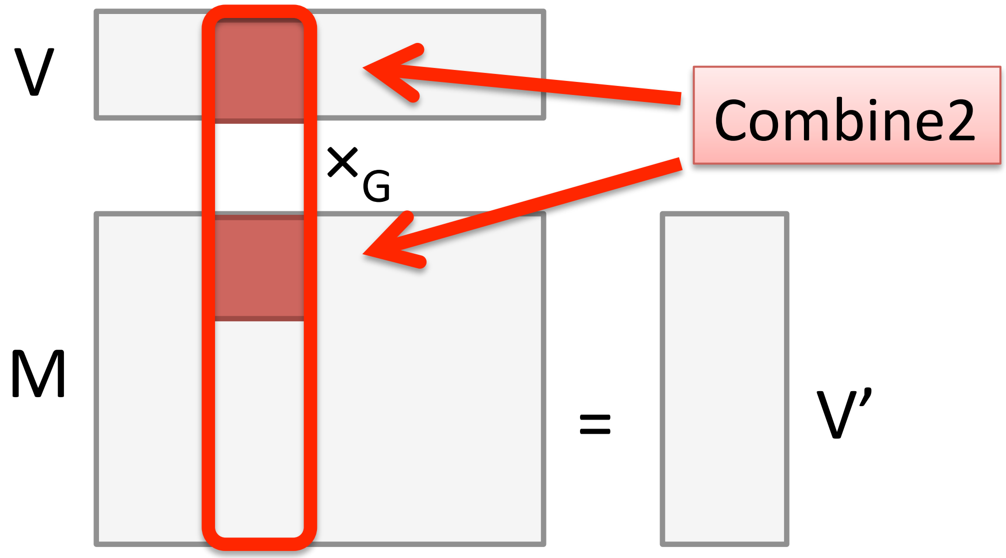


*1 : Kang, U. et al, “PEGASUS: A Peta-Scale Graph Mining System- Implementation and Observations”, IEEE INTERNATIONAL CONFERENCE ON DATA MINING 2009

Large graph processing algorithm GIM-V

- **Generalized Iterative Matrix-Vector multiplication** *1
  - Graph applications are implemented by defining 3 functions
    - PageRank, Random Walk with Restart, Connected Components etc.
  - $v' = M \times_G v$ where

v'_i = Assign(v_j , CombineAll$_j$ ({x_j | j = 1..n, x_j = Combine2($m_{i,j}$, v_j)})) (i = 1..n)

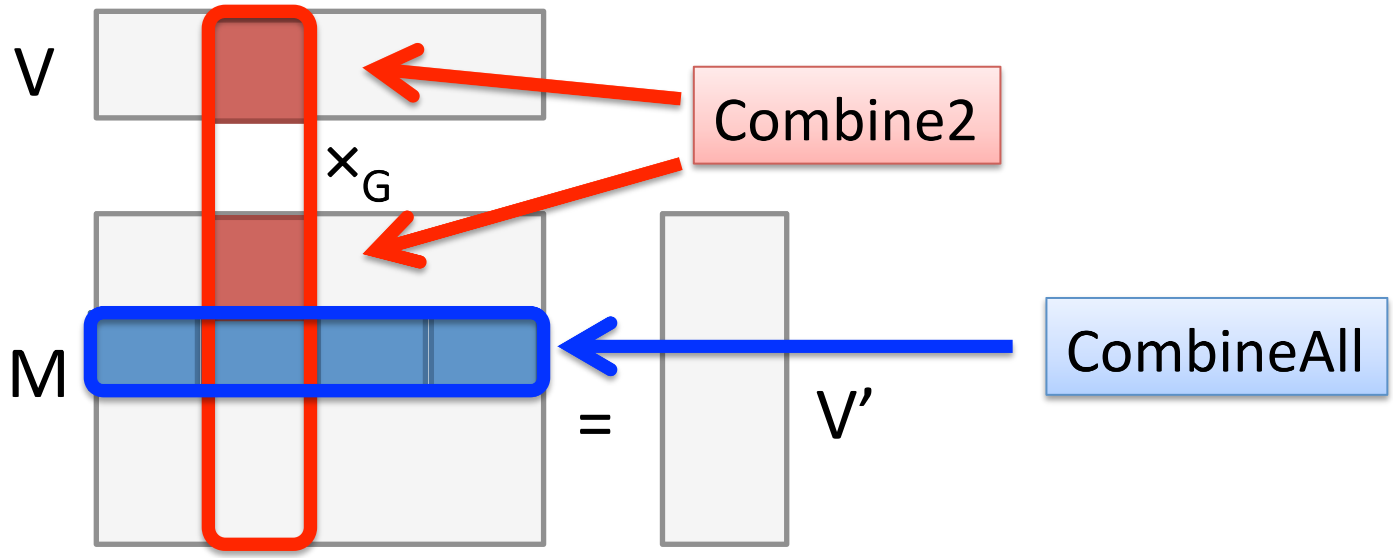


*1 : Kang, U. et al, "PEGASUS: A Peta-Scale Graph Mining System- Implementation and Observations", IEEE INTERNATIONAL CONFERENCE ON DATA MINING 2009

Large graph processing algorithm GIM-V

- **Generalized Iterative Matrix-Vector multiplication**[*1]
  - Graph applications are implemented by defining 3 functions
    - PageRank, Random Walk with Restart, Connected Components etc.
  - $v' = M \times_G v$ where

v'_i = Assign(v_j , CombineAll$_i$ ({x_j | j = 1..n, x_j = Combine2($m_{i,j}$, v_j)})) (i = 1..n)

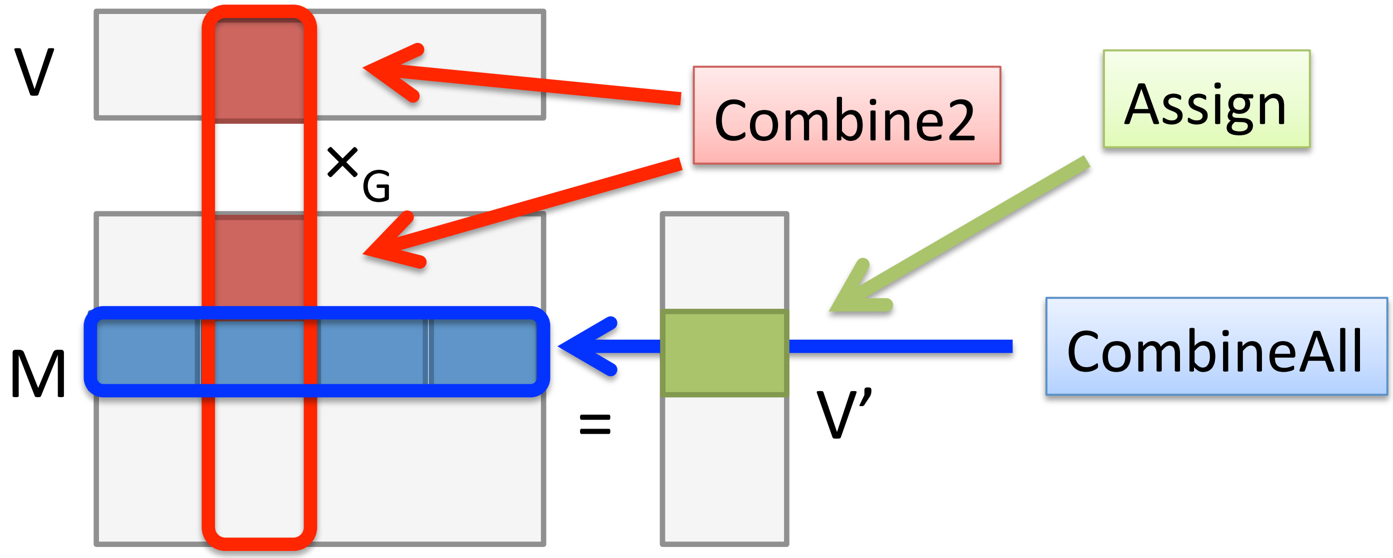


*1 : Kang, U. et al, "PEGASUS: A Peta-Scale Graph Mining System- Implementation and Observations", IEEE INTERNATIONAL CONFERENCE ON DATA MINING 2009

Large graph processing algorithm GIM-V

- **Generalized Iterative Matrix-Vector multiplication** *1
  - Graph applications are implemented by defining 3 functions
    - PageRank, Random Walk with Restart, Connected Components etc.
  - $v' = M \times_G v$ where

$v'_i = \text{Assign}(v_j, \text{CombineAll}_j(\{x_j \mid j = 1..n, x_j = \text{Combine2}(m_{i,j}, v_j)\}))\ \ (i = 1..n)$



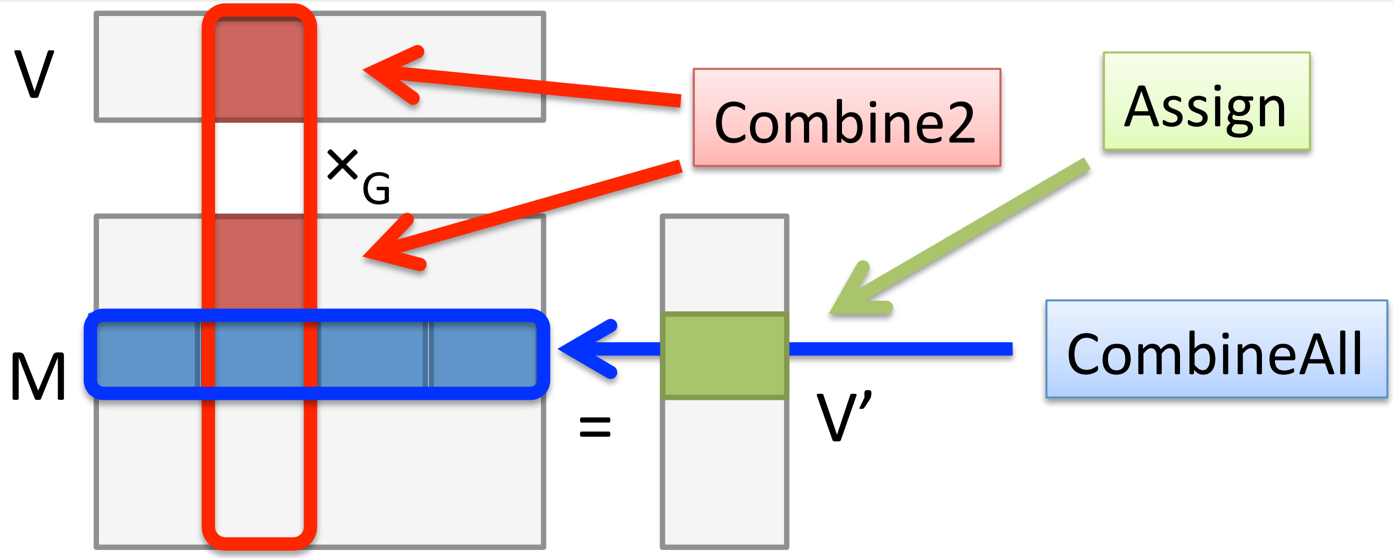
*1 : Kang, U. et al, "PEGASUS: A Peta-Scale Graph Mining System- Implementation and Observations", IEEE INTERNATIONAL CONFERENCE ON DATA MINING 2009

Large graph processing algorithm GIM-V

- **Generalized Iterative Matrix-Vector multiplication**[*1]

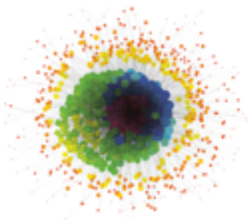
GIM-V can be implemented by 2-stage MapReduce
- **Stage 1: Combine2**
- **Stage 2: CombineAll, Assign**

→ Implement on our GPU MapReduce framework

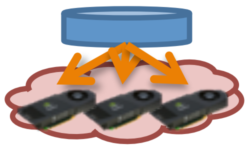


*1 : Kang, U. et al, "PEGASUS: A Peta-Scale Graph Mining System- Implementation and Observations", IEEE INTERNATIONAL CONFERENCE ON DATA MINING 2009

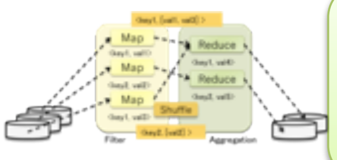
Previous work: Multi-GPU-MapReduce-based Graph Processing [1]



Graph Application
PageRank



Graph Algorithm
Multi-GPU GIM-V

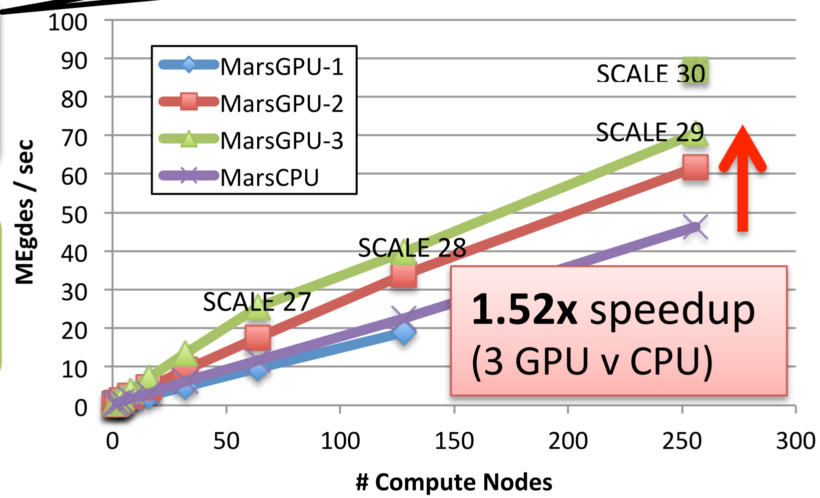


MapReduce Framework
Multi-GPU Mars

Platform
CUDA, MPI

Implement GIM-V on multi-GPUs MapReduce
- Optimization for GIM-V
- Load balance optimization

Extend existing GPU MapReduce framework (Mars) for multi-GPU



[1]: K. Shirahata et al., "A Scalable Implementation of a MapReduce-based Graph Processing Algorithm for Large-scale Heterogeneous Supercomputers", IEEE/ACM CCGrid, 2013

Problems on Large-scale Graph Processing on GPU

- **How to manage graph data whose size exceeds GPU memory capacity ?**
  - Handling memory overflow from GPU memory with minimal performance overhead
    - GPU memory capacity is smaller than CPU memory
    - Data transfers dominantly disturb efficient graph processing
      - e.g.) TSUBAME2.5: GPU 250 GB/sec, CPU-GPU 8 GB/sec
  - Efficient graph data assignment onto GPUs
    - Tradeoff between using single GPU on multiple nodes (scale-out) or using multiple GPUs per node (scale-up) in terms of performance and power efficiencies

Existing Solutions

- Handling memory overflow from GPU memory
  - Using multiple GPUs
    - GPU-MapReduce-based graph processing [Shirahata et al. 2013]
    - Breadth first search on Multi-GPU [Ueno et al. 2013]

    → Not consider memory overflow from GPU memory
  - Offloading graph data onto CPU memory
    - GPUfs: I/O from a GPU to file systems [Silberstein et al. 2013]
    - GPMR: a multi-GPU MapReduce library [Stuart et al. 2011]

    → Not experiment on realistic large-scale applications
- Analysis of tradeoff between scale-up and scale-out
  - Scale-up and Scale-out on CPUs [Michael et al. 2007]

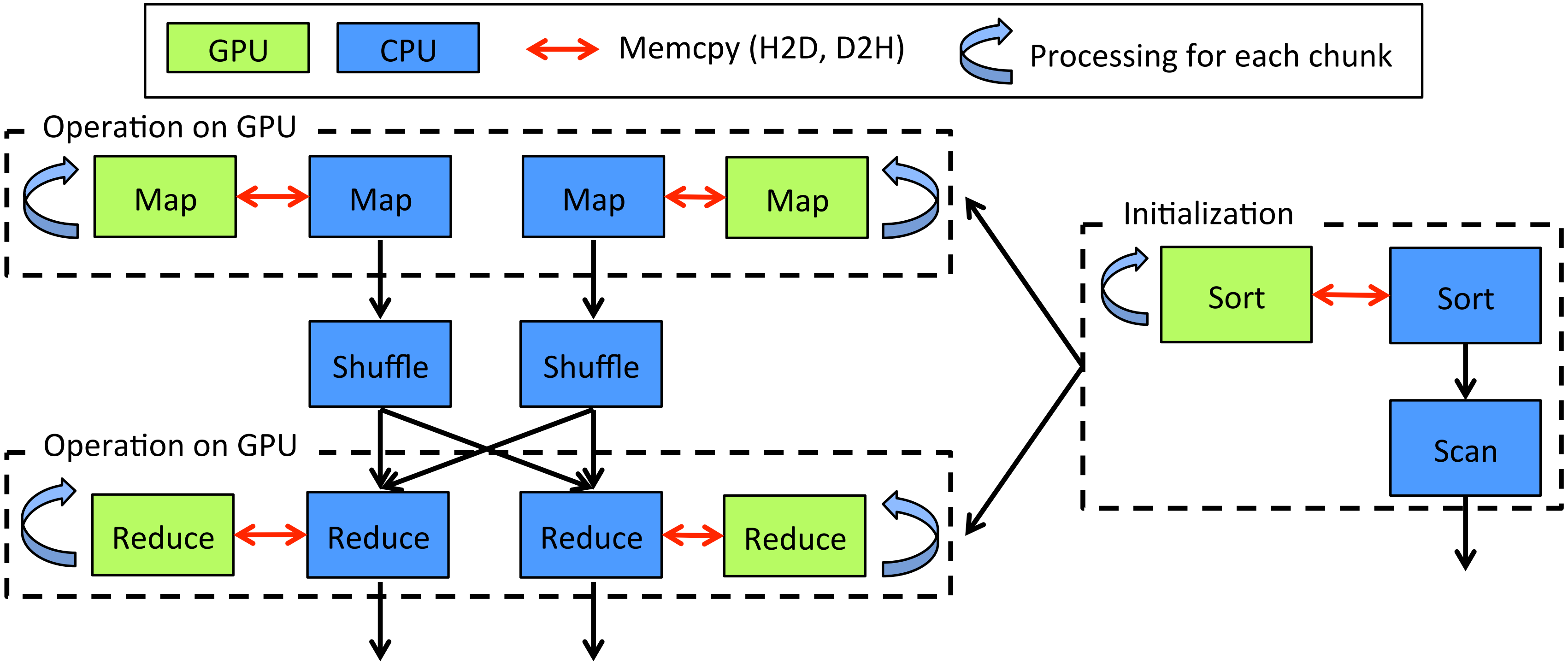
    → Not compare on GPUs

Idea: Streaming-based Out-of-core GPU Memory Management

- Streaming out-of-core GPU memory management
  - Divide graph data into multiple chunks and assigning each chunk one by one in each CUDA stream
  - Hide CPU-GPU data transfer by applying overlapping techniques between computation and data transfer
- GPU-based external sorting
  - Employ sample-based out-of-core GPU sorting
  - Out-of-core GPU sorting is conducted when graph data size exceeds GPU memory capacity

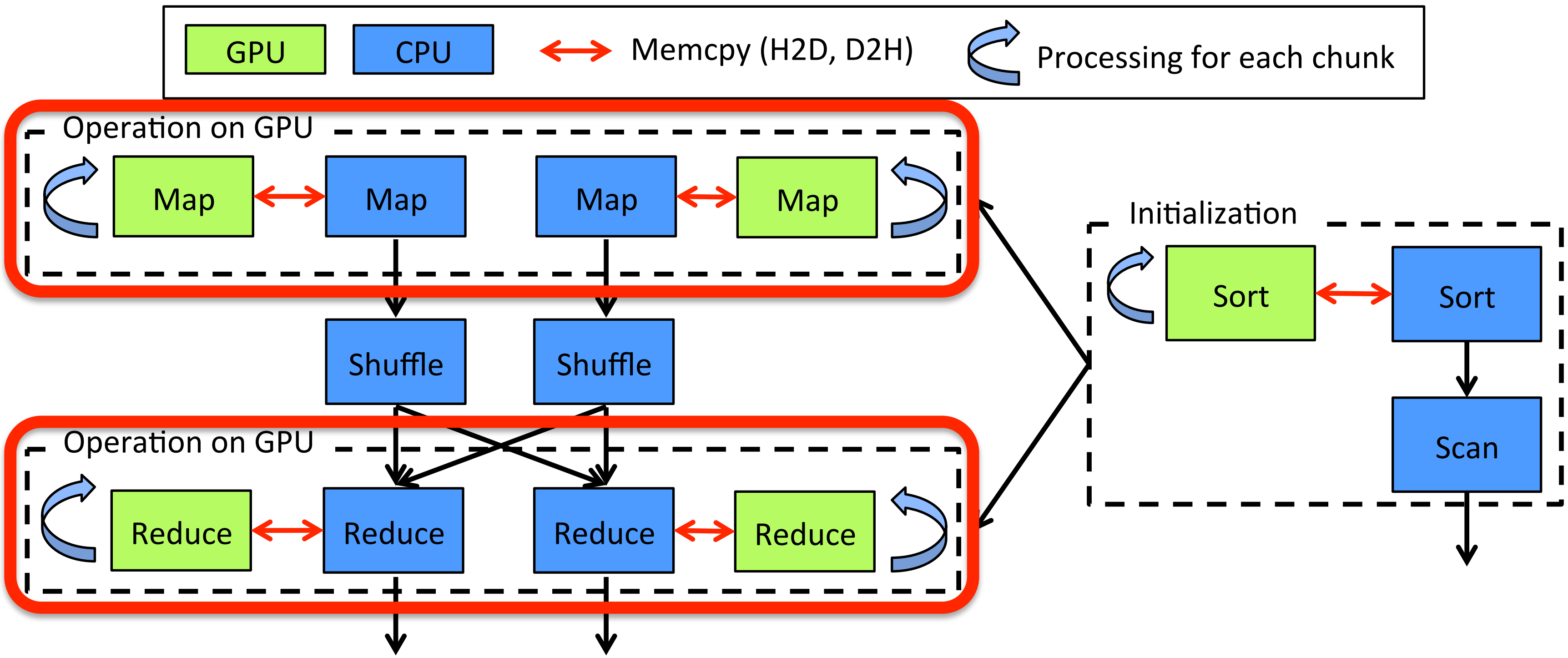
Out-of-core GPU Memory Management for MapReduce-based Graph Processing

- Out-of-core GPU memory management
  - Stream-based GPU MapReduce processing
  - Out-of-core GPU sorting



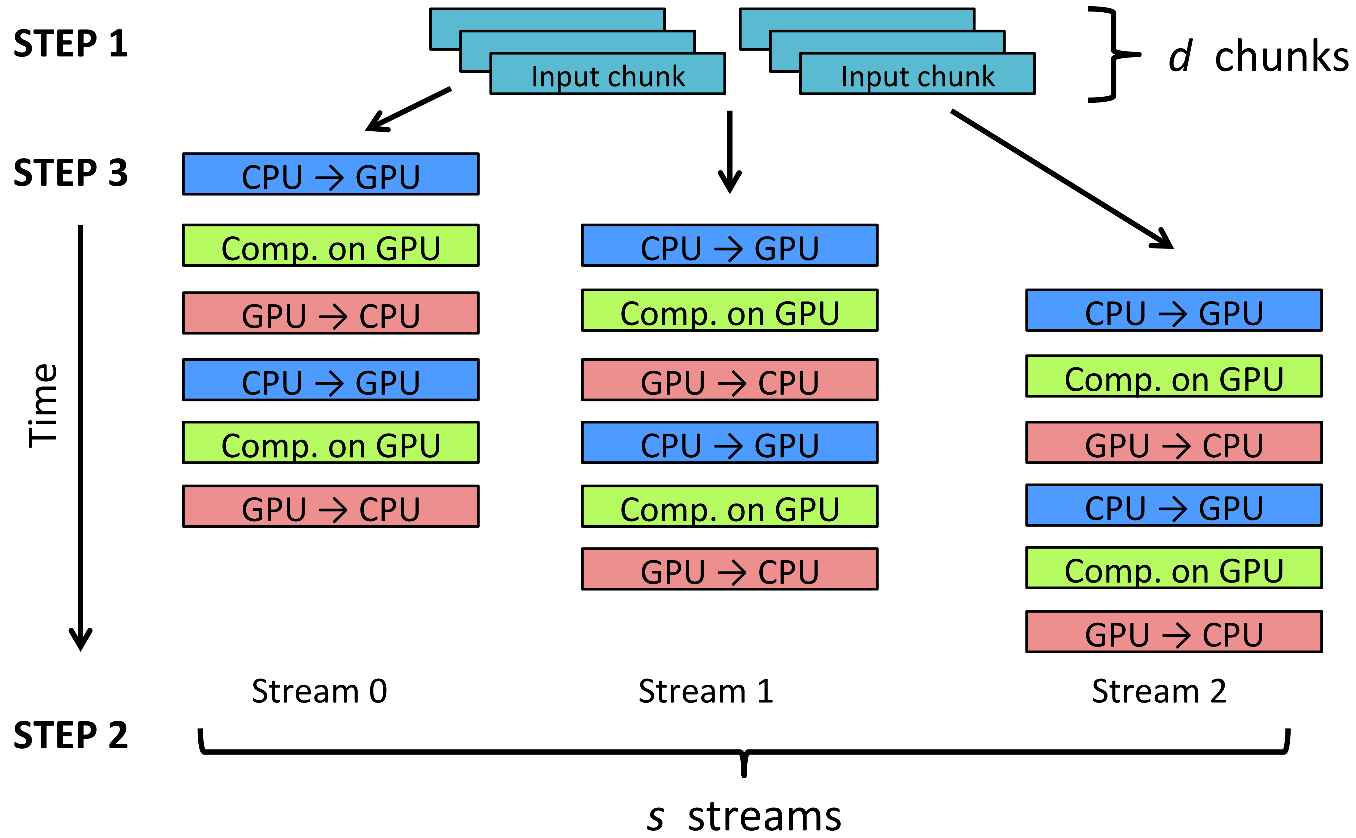
Out-of-core GPU Memory Management for MapReduce-based Graph Processing

- Out-of-core GPU memory management
  - Stream-based GPU MapReduce processing
  - Out-of-core GPU sorting



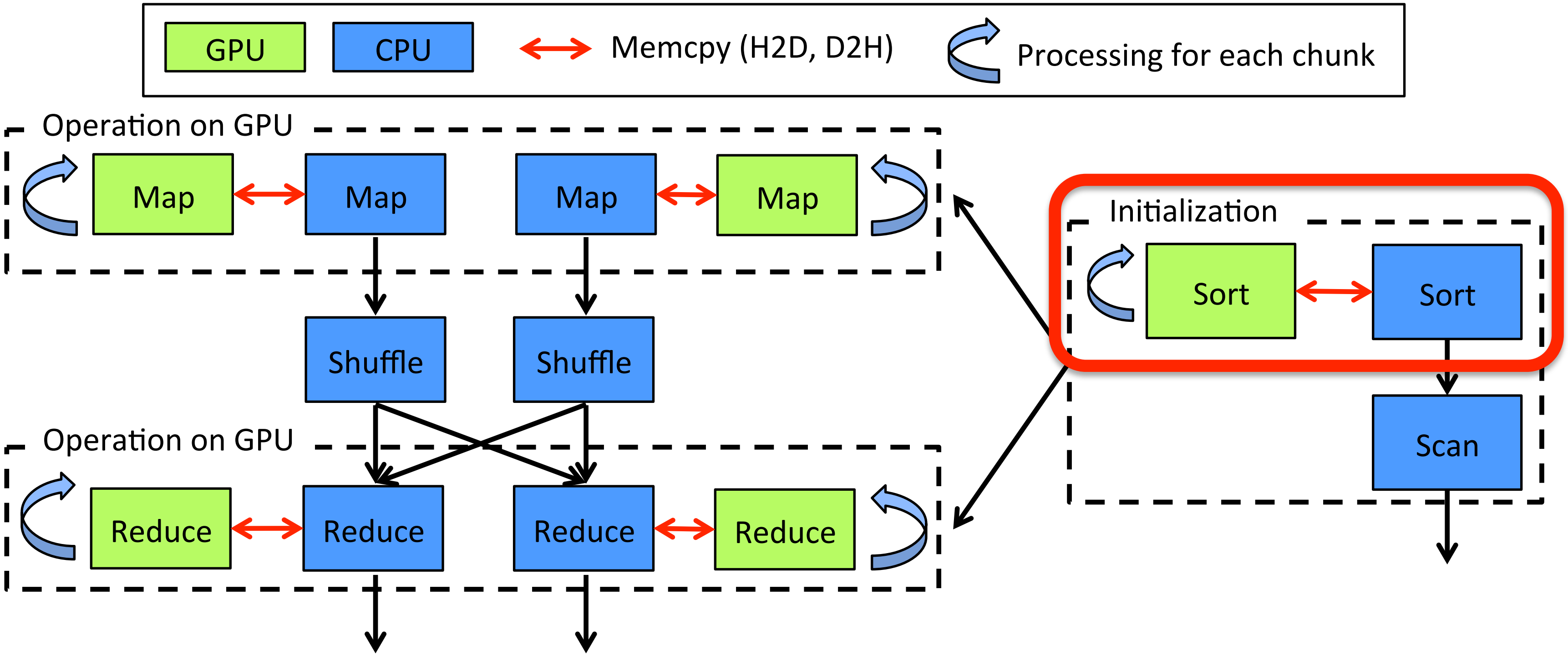
Stream-based GPU MapReduce Processing

- Overlap three operations
  - Copy CPU → GPU, Map/Reduce operation on GPU, Copy GPU → CPU
- Dynamically update the number of chunks (*d*) to fit on GPU memory



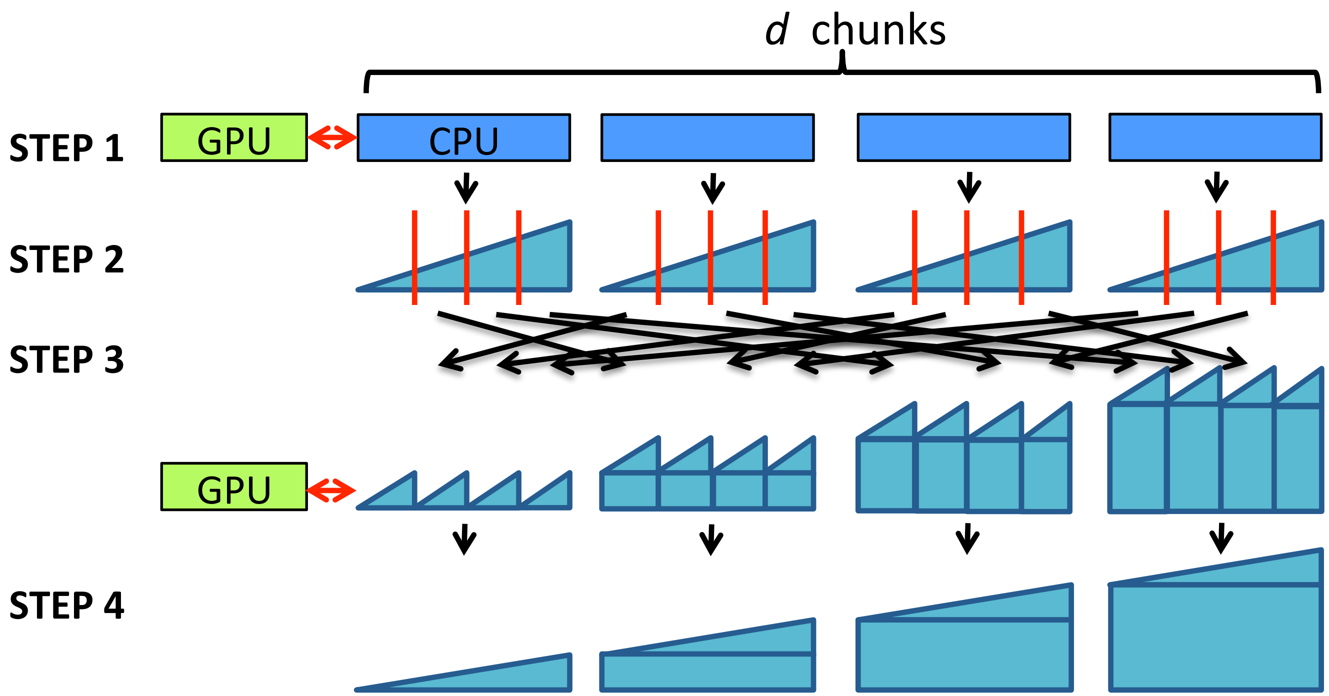
Out-of-core GPU Memory Management for MapReduce-based Graph Processing

- Out-of-core GPU memory management
  - Stream-based GPU MapReduce processing
  - Out-of-core GPU sorting



Out-of-core GPU Sorting

- Use sample-based out-of-core sorting [1]
  - Divide input data into chunks and split each chunk using splitters
  - Improve by decreasing the number of CPU-GPU data transfers
- Thrust radix sort is used for in-core sorting



[1]: Y. Ye et al., "GPUMemSort: A High Performance Graphics Co-processors Sorting Algorithm for Large Scale In-Memory Data", GSTF International Journal on Computing, 2011

Optimization Techniques

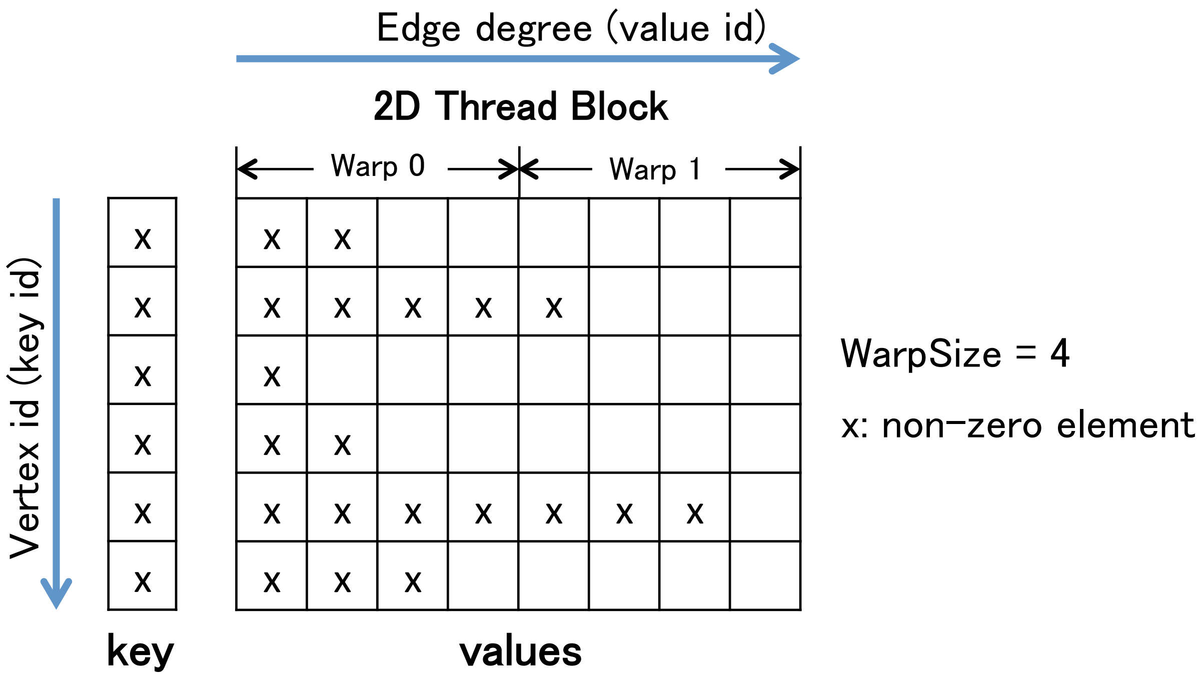
- Data structure
  - Employ a compact data structure similar to CSR for sparse matrix formats
    - Arrays of keys, values → arrays of unique keys, values
    - Compress duplicate keys to 1/{#edges per vertex}
    - Sort key-value → scan (prefix sum) → compact keys
- Shuffle
  - Implement range-based and hash-based splitters
  - Use range-based splitter, which performs good load balance by randomizing vertex indices
- Thread assignment policy on GPU
  - Apply warp-based assignment

Optimization Techniques

- Data structure
  - Employ a compact data structure similar to CSR for sparse matrix formats
    - Arrays of keys, values → arrays of unique keys, values
    - Compress duplicate keys to 1/{#edges per vertex}
    - Sort key-value → scan (prefix sum) → compact keys
- Shuffle
  - Implement range-based and hash-based splitters
  - Use range-based splitter, which performs good load balance by randomizing vertex indices
- Thread assignment policy on GPU
  - Apply warp-based assignment

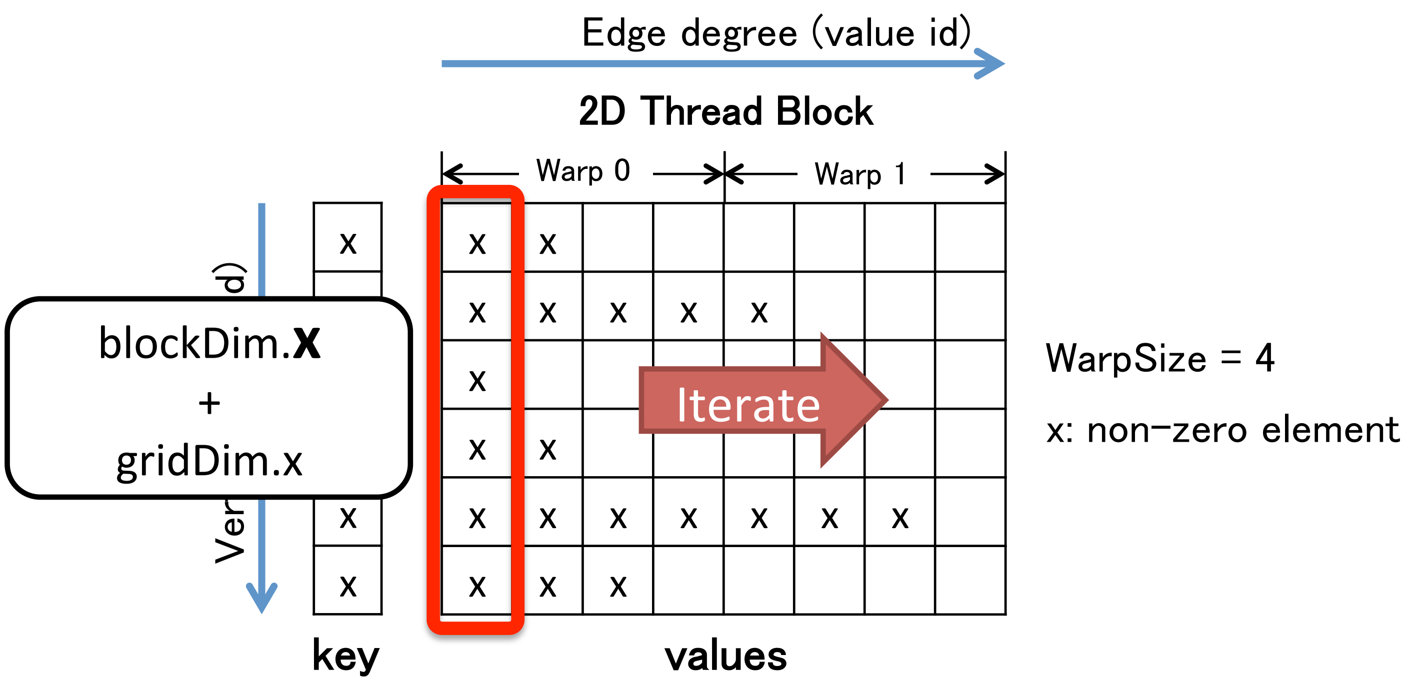
Thread Assignment Optimization

- Three thread assignment policies



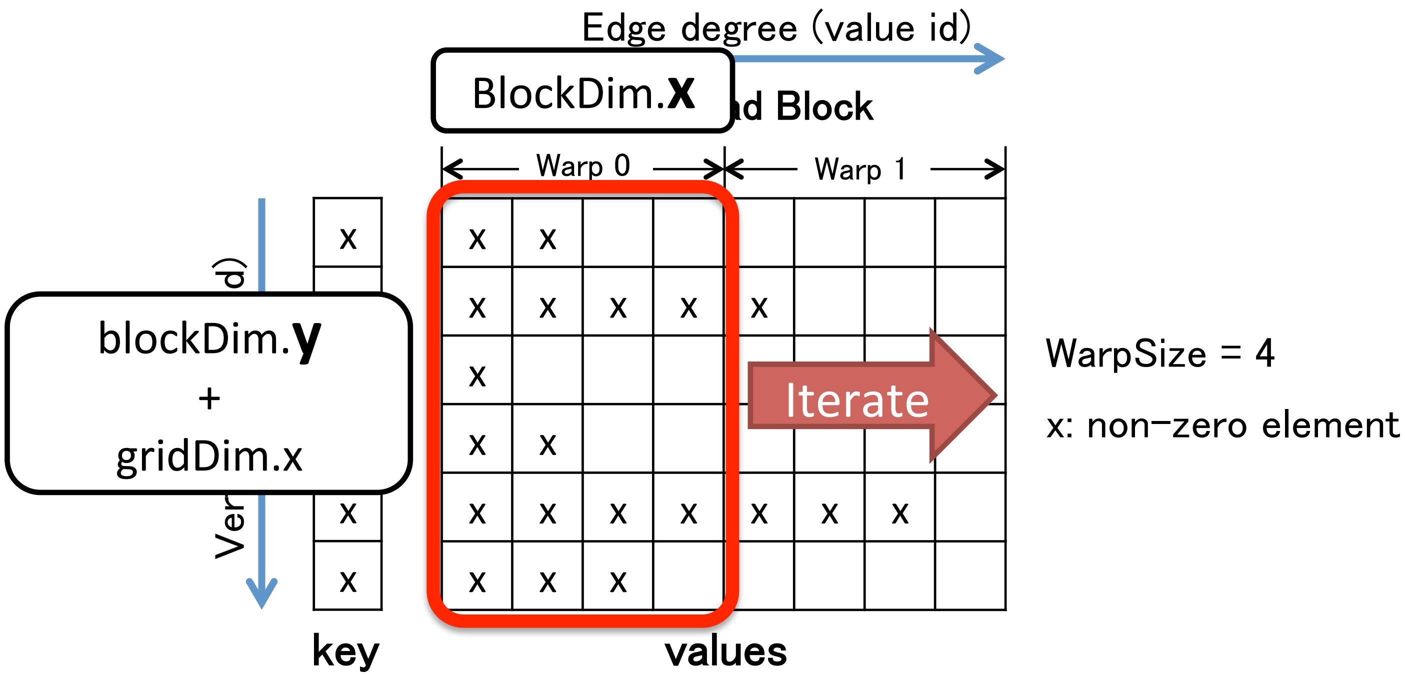
Thread Assignment Optimization

- Three thread assignment policies
  - Thread-based assignment: assign one thread per vertex



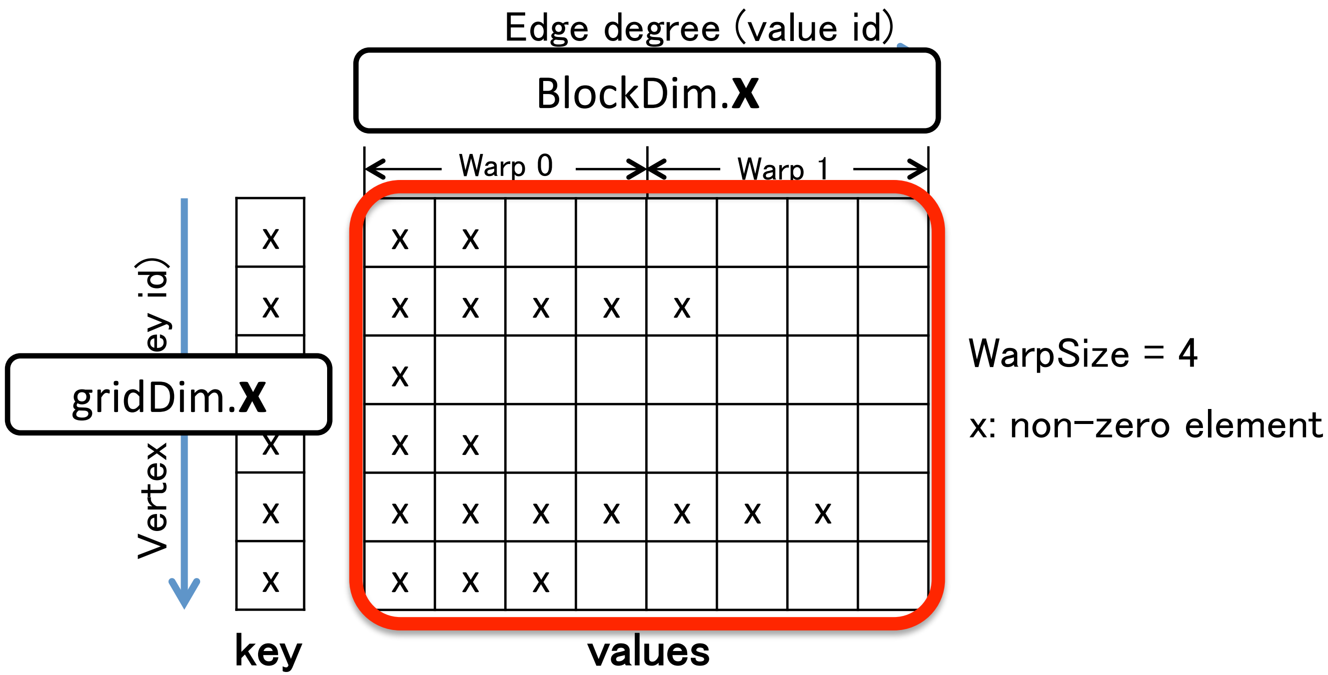
Thread Assignment Optimization

- Three thread assignment policies
  - Thread-based assignment: assign one thread per vertex
  - Warp-based assignment: assign one warp per vertex (32 on K20x GPU)



Thread Assignment Optimization

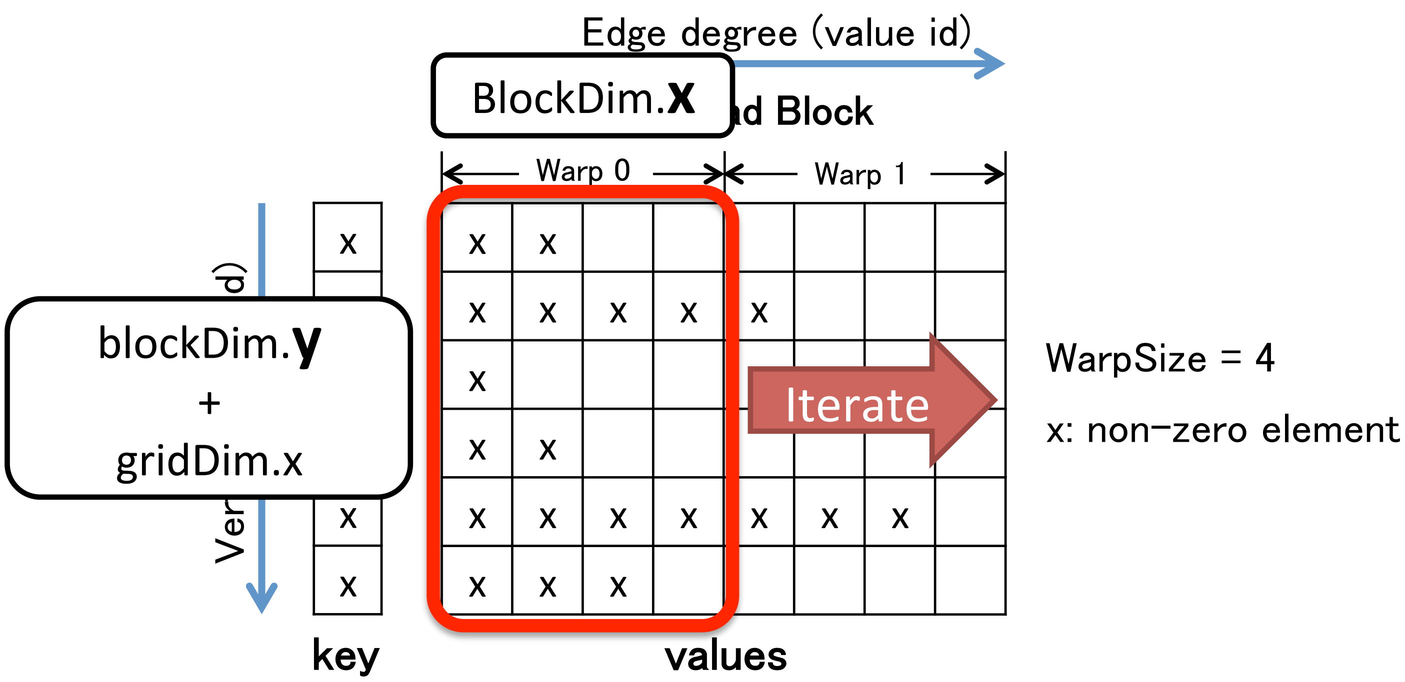
- Three thread assignment policies
  - Thread-based assignment: assign one thread per vertex
  - Warp-based assignment: assign one warp per vertex (32 on K20x GPU)
  - Thread block-based assignment: assign one thread block per vertex (1024 on K20x GPU)



Thread Assignment Optimization

- Three thread assignment policies
  - Thread-based assignment: assign one thread per vertex
  - Warp-based assignment: assign one warp per vertex (32 on K20x GPU)
  - Thread block-based assignment: assign one thread block per vertex (1024 on K20x GPU)

→ Apply warp-based 2D thread mapping, since warp size is expected to be close to the average number of edges per vertex

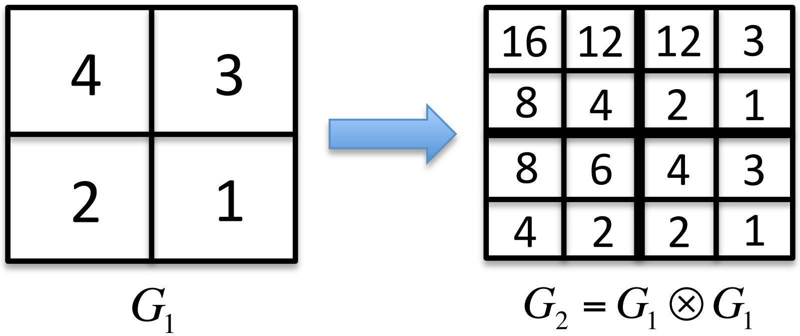


Experiments

Study the performance of our multi-GPU GIM-V

- **Comparison with a CPU-based implementation**
- **Analysis of performance and power efficiencies**

- Methods
  - A single round of iterations (w/o Preprocessing)
  - PageRank application
    - Measures relative importance of web pages
  - Input data
    - Artificial Kronecker graphs
      - Generated by generator in Graph500
    - Parameters
      - *SCALE*: log 2 of #vertices (#vertices = 2^{SCALE})
      - *Edge_factor*: 16 (#edges = *Edge_factor* × #vertices)



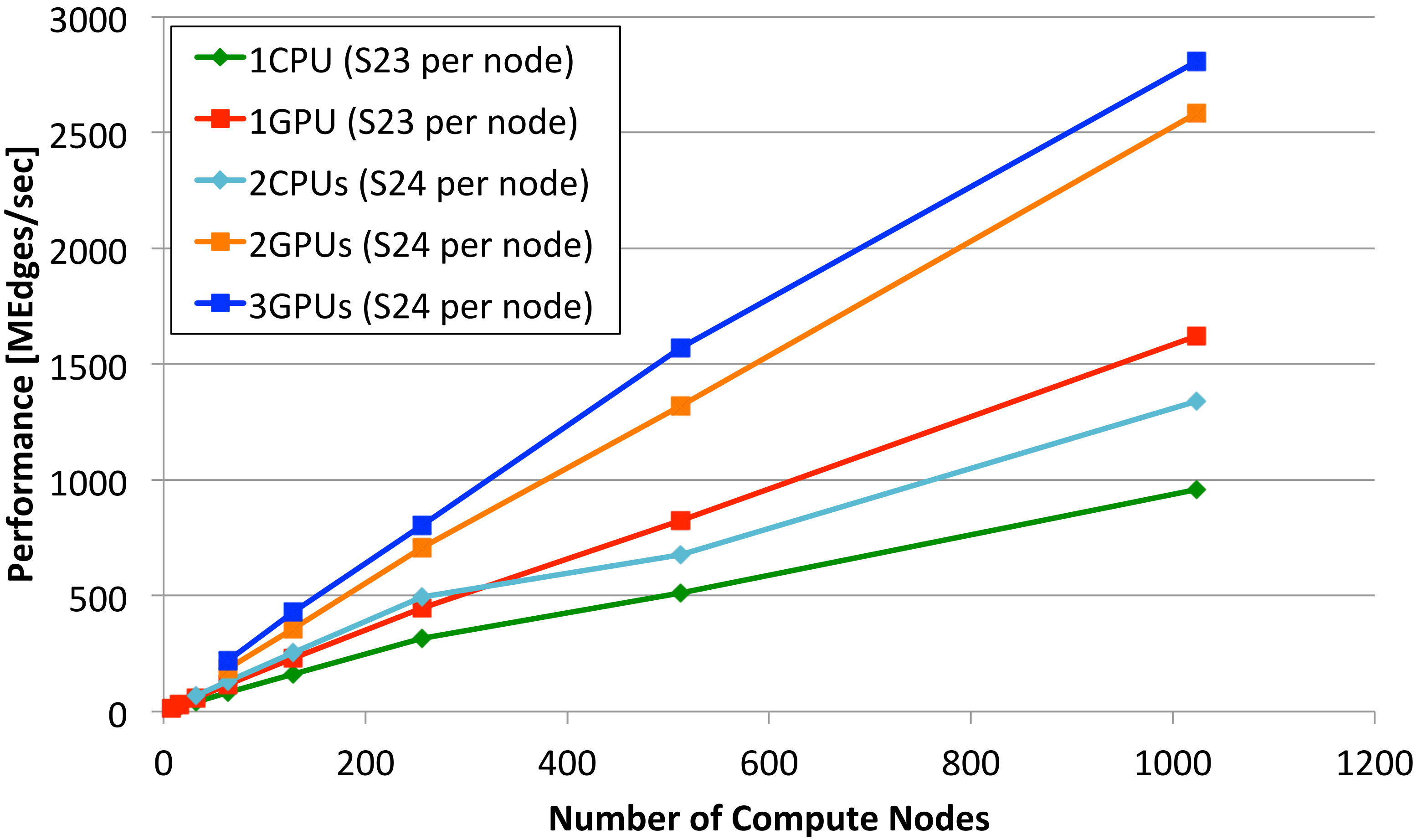
Experimental environments

- TSUBAME2.5 supercomputer
  - Use up to 1024 nodes (3072 GPUs)
    - CPU-GPU: PCI-E 2.0 x16 (8 GB/sec)
    - Internode: QDR IB dual rail (10 GB/sec)
- Setup
  - *n GPU(s)*
    - n GPUs / node (n: 1, 2, 3)
  - *n CPU(s)*
    - 12 threads / node
    - MPI and OpenMP
    - Thrust OpenMP Sort

	2 CPUs / node	**3 GPUs / node**
Model	Intel® Xeon® X5670	Tesla K20X
# Cores	6	2688
Frequency	2.93 GHz	0.732 GHz
Memory	54 GB	6 GB
Memory BW	32 GB/sec	250 GB/sec
Compiler	gcc 4.3.4	Nvcc 5.0

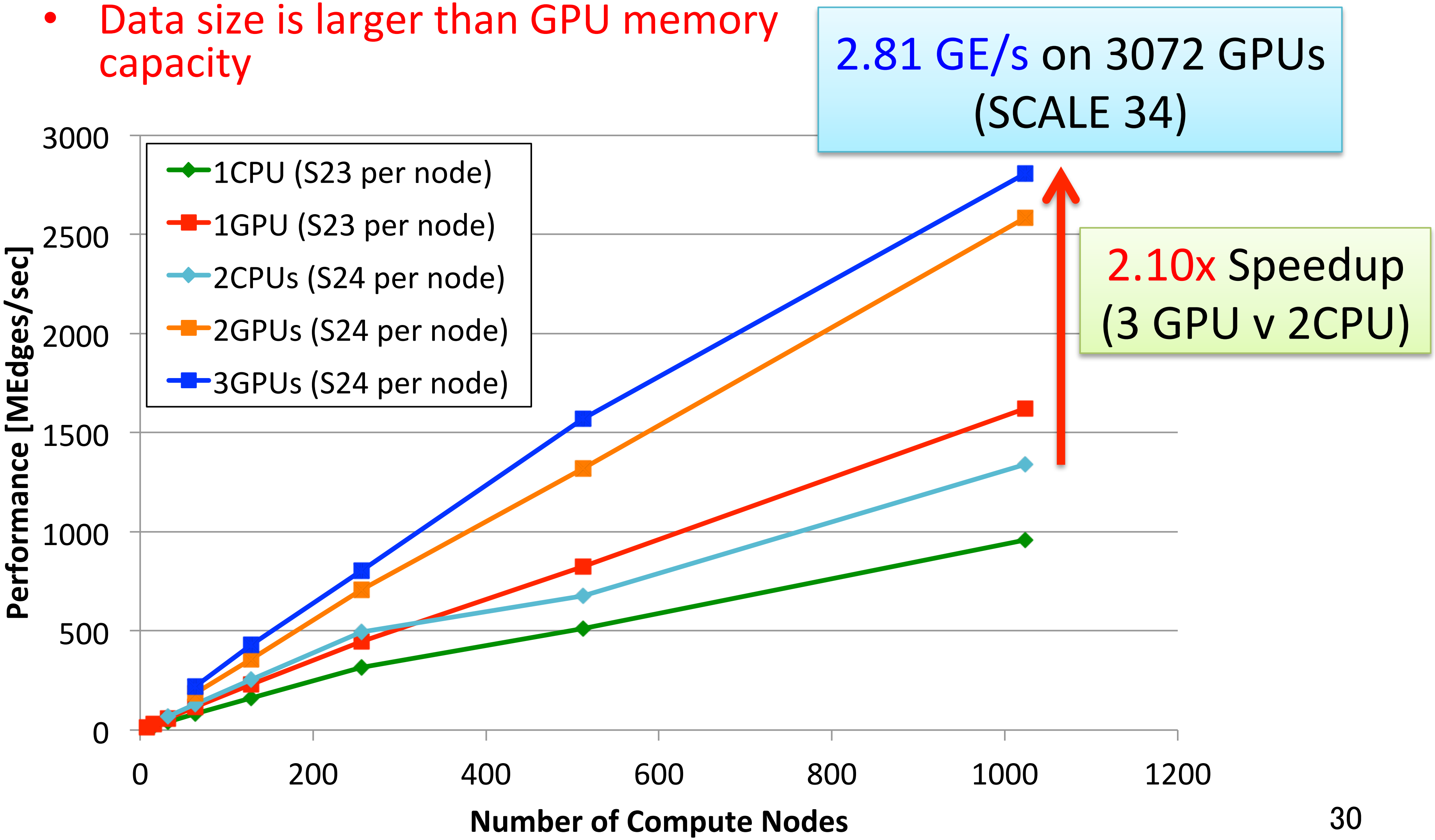
Weak Scaling Performance

- PageRank application
- Data size is larger than GPU memory capacity



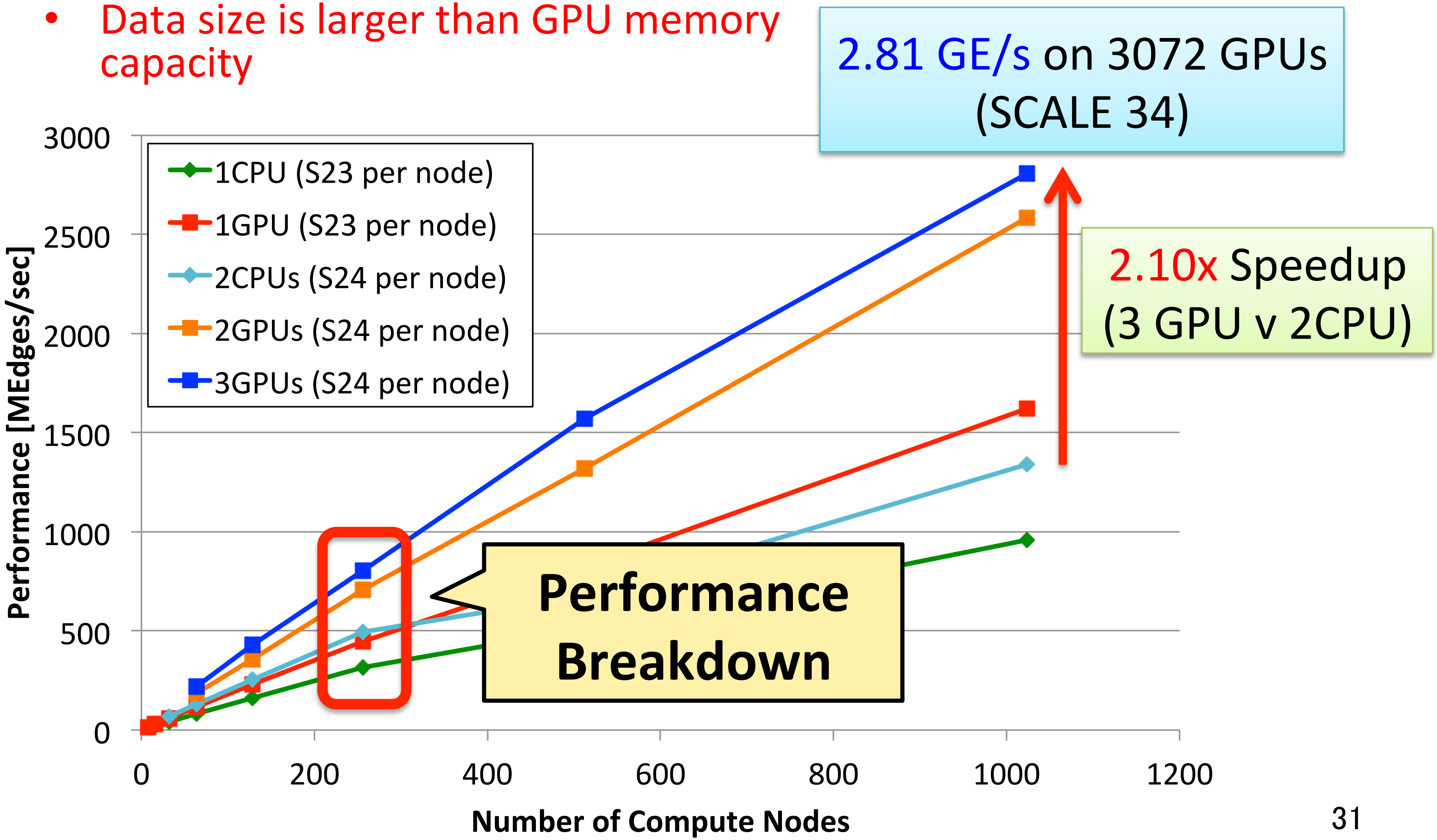
Weak Scaling Performance

- PageRank application
- Data size is larger than GPU memory capacity



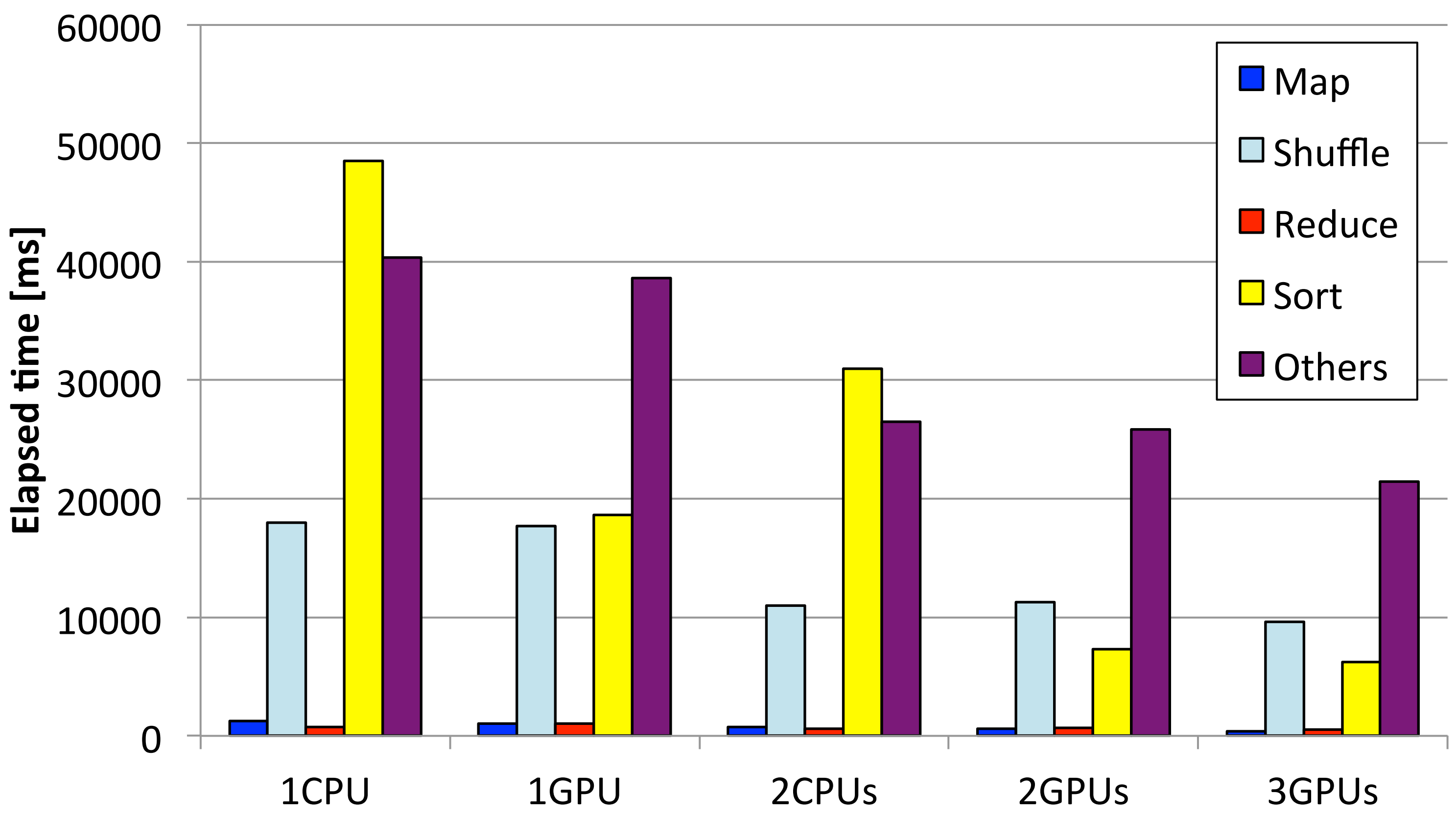
Weak Scaling Performance

- PageRank application
- Data size is larger than GPU memory capacity



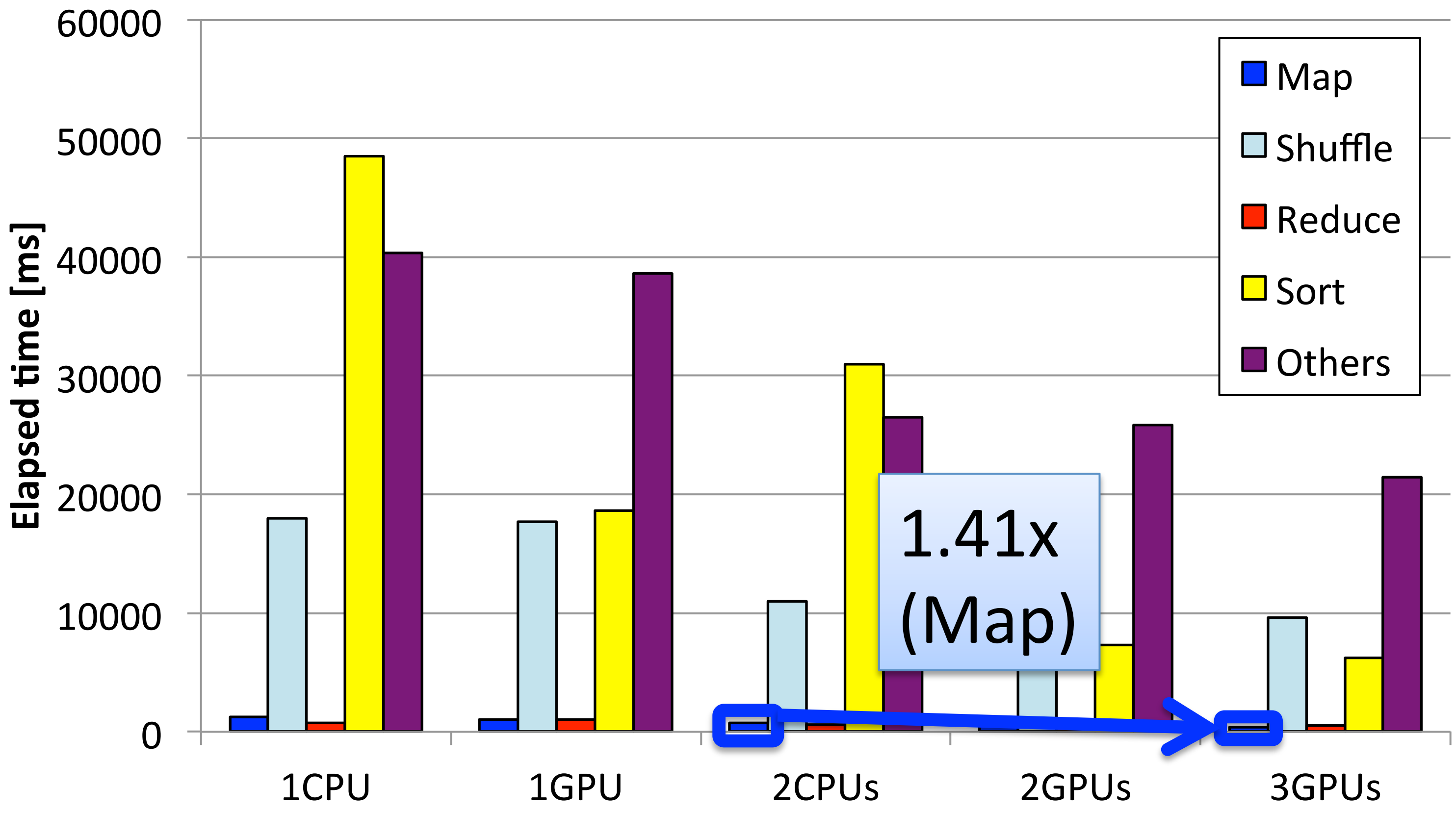
Breakdown

- Performance on 3 GPUs compared with 2 CPUs
  - SCALE 31, 256 nodes
  - Map: 1.41x, Reduce: 1.49x, Sort: 4.95x speedup
    - Overlapping communication effectively



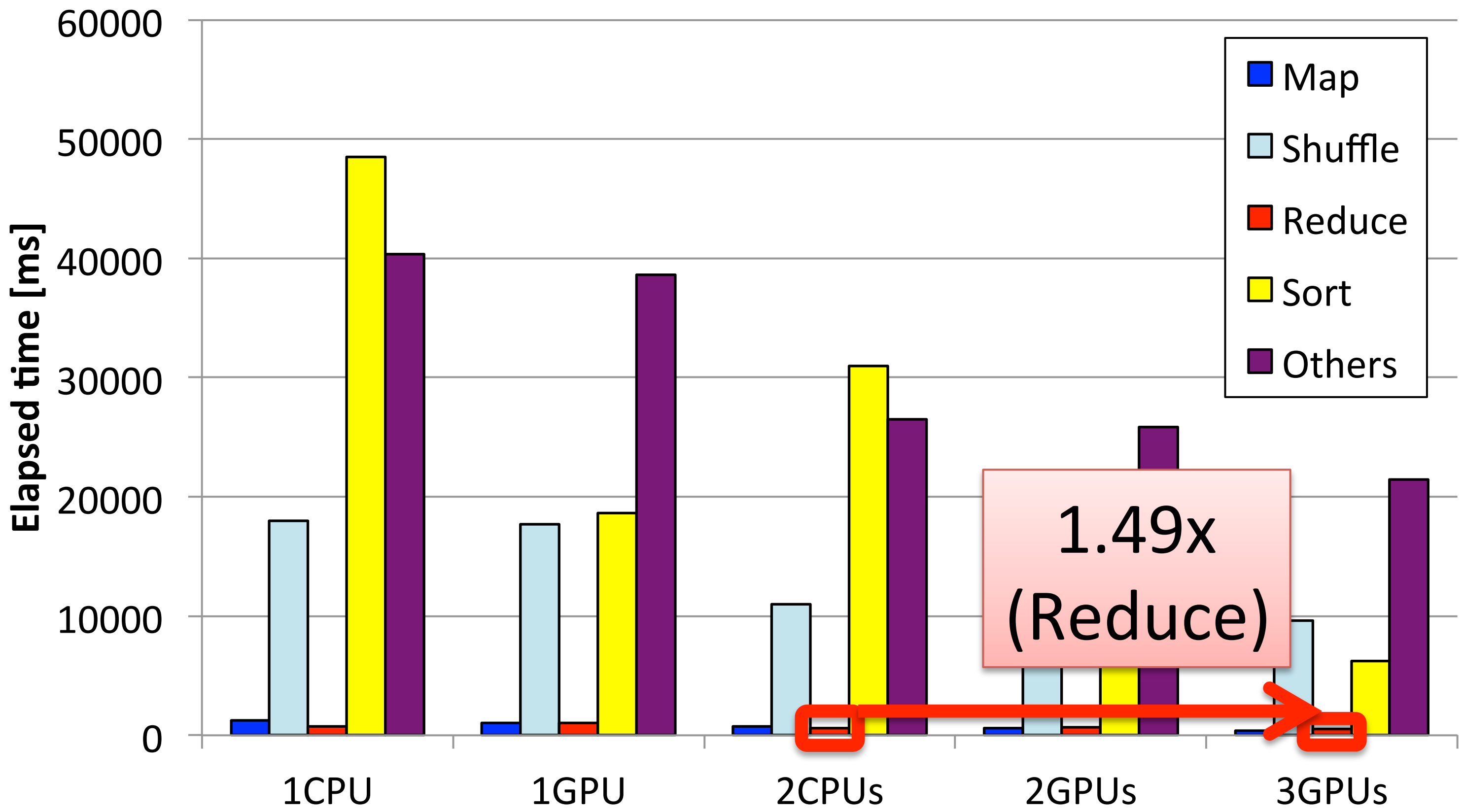
Breakdown

- Performance on 3 GPUs compared with 2 CPUs
  - SCALE 31, 256 nodes
  - Map: 1.41x, Reduce: 1.49x, Sort: 4.95x speedup
    - Overlapping communication effectively



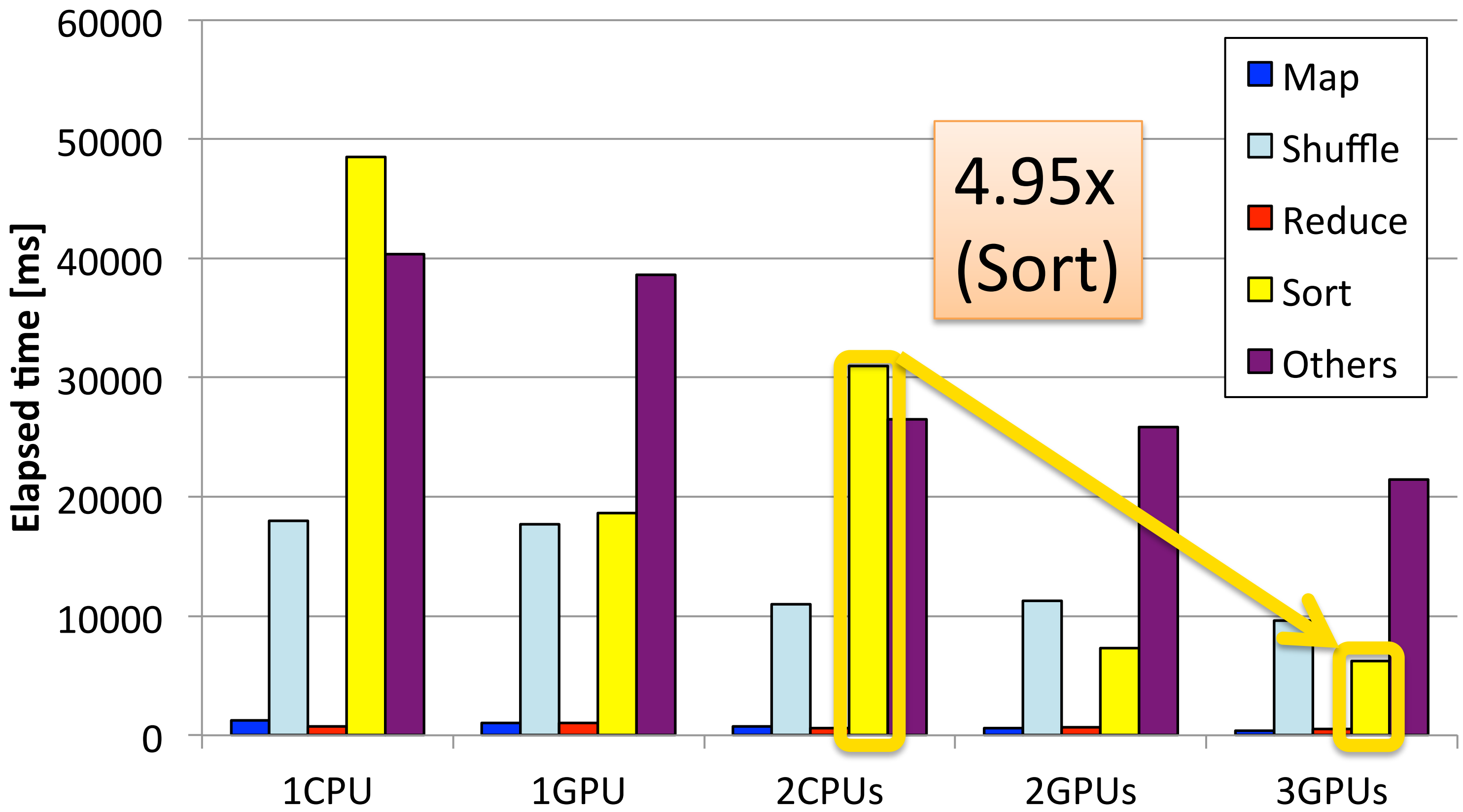
Breakdown

- Performance on 3 GPUs compared with 2 CPUs
  - SCALE 31, 256 nodes
  - Map: 1.41x, Reduce: 1.49x, Sort: 4.95x speedup
    - Overlapping communication effectively



Breakdown

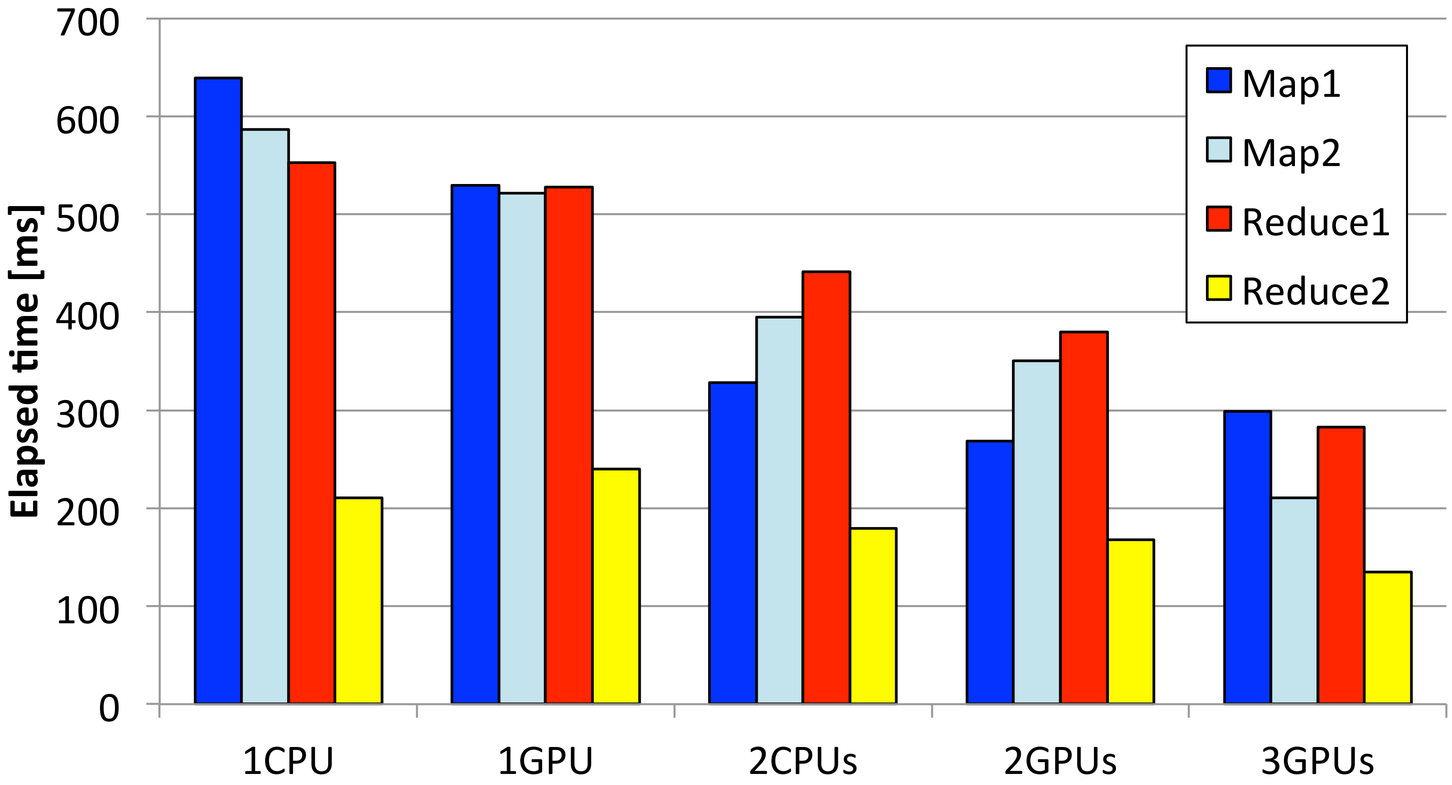
- Performance on 3 GPUs compared with 2 CPUs
  - SCALE 31, 256 nodes
  - Map: 1.41x, Reduce: 1.49x, Sort: 4.95x speedup
    - Overlapping communication effectively



Breakdown of Map/Reduce

- Map1, Map2 (Pass) 1.41x
  - Speedup by overlapping communication efficiently
- Reduce1 (Combine2) 1.56x, Reduce2 (CombineAll, Assign) 1.33x
  - Speedup by overlapping communication and parallel reduction

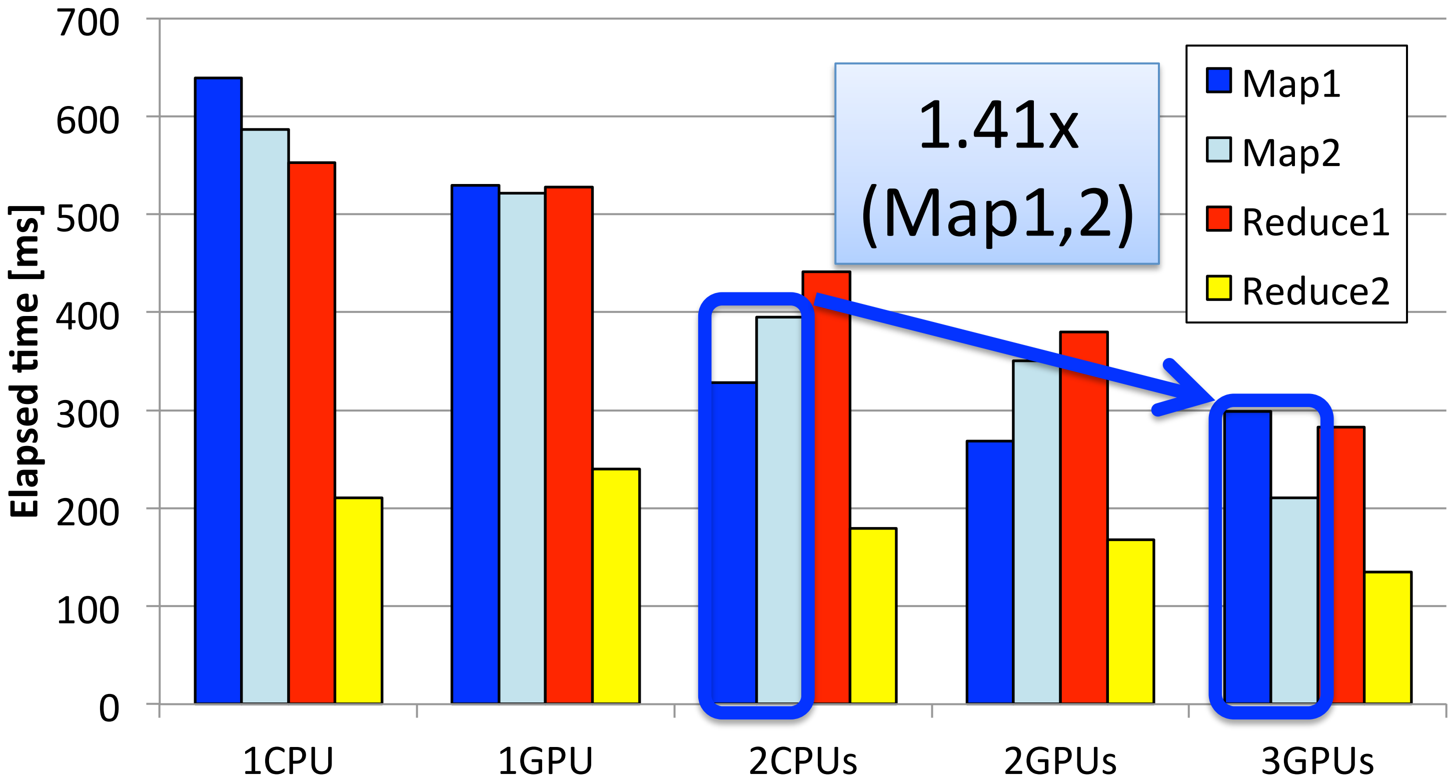
→ heavier operation is more accelerated



Breakdown of Map/Reduce

- Map1, Map2 (Pass) 1.41x
  - Speedup by overlapping communication efficiently
- Reduce1 (Combine2) 1.56x, Reduce2 (CombineAll, Assign) 1.33x
  - Speedup by overlapping communication and parallel reduction

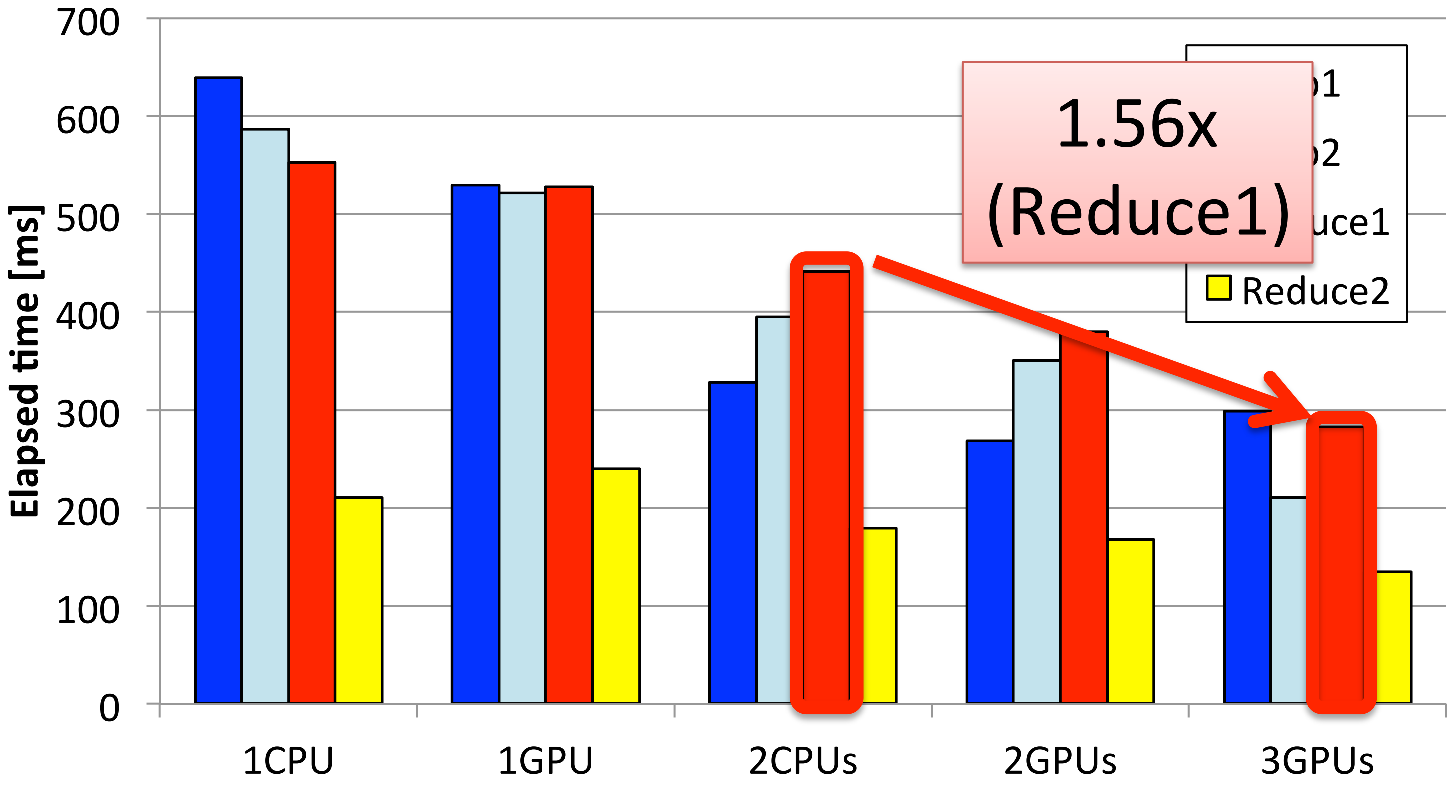
→ heavier operation is more accelerated



Breakdown of Map/Reduce

- Map1, Map2 (Pass) 1.41x
  - Speedup by overlapping communication efficiently
- Reduce1 (Combine2) 1.56x, Reduce2 (CombineAll, Assign) 1.33x
  - Speedup by overlapping communication and parallel reduction

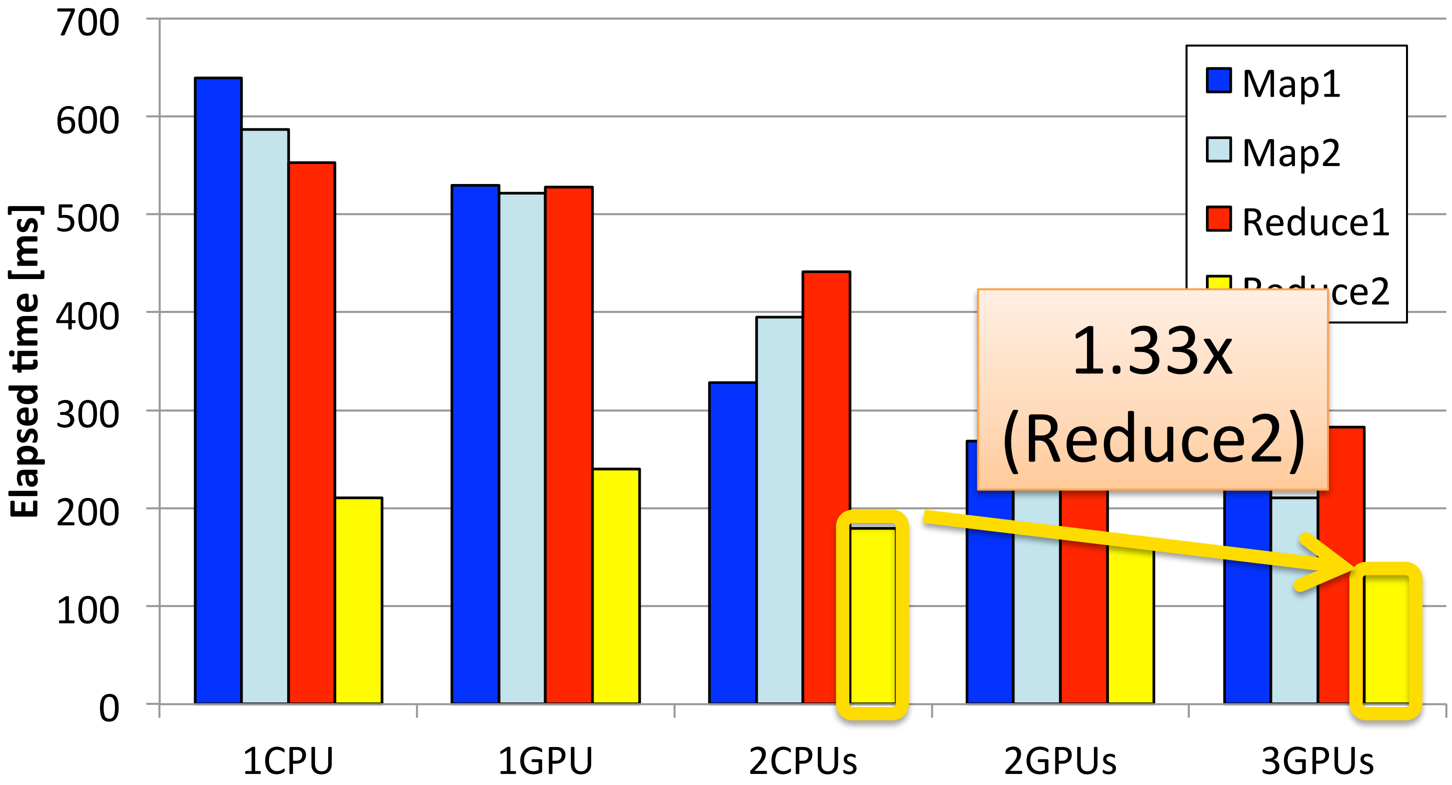
→ heavier operation is more accelerated



Breakdown of Map/Reduce

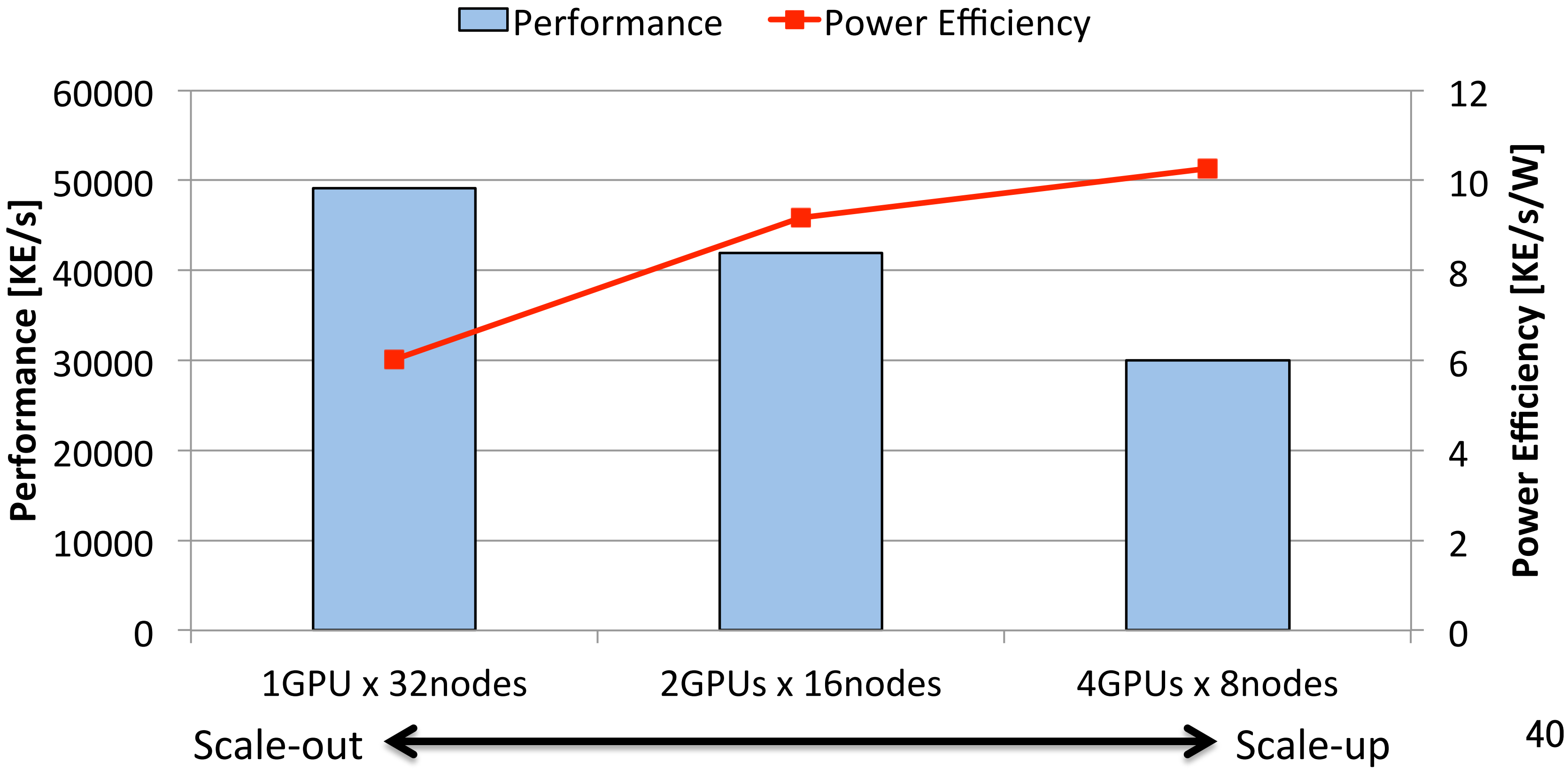
- Map1, Map2 (Pass) 1.41x
  - Speedup by overlapping communication efficiently
- Reduce1 (Combine2) 1.56x, Reduce2 (CombineAll, Assign) 1.33x
  - Speedup by overlapping communication and parallel reduction

→ heavier operation is more accelerated



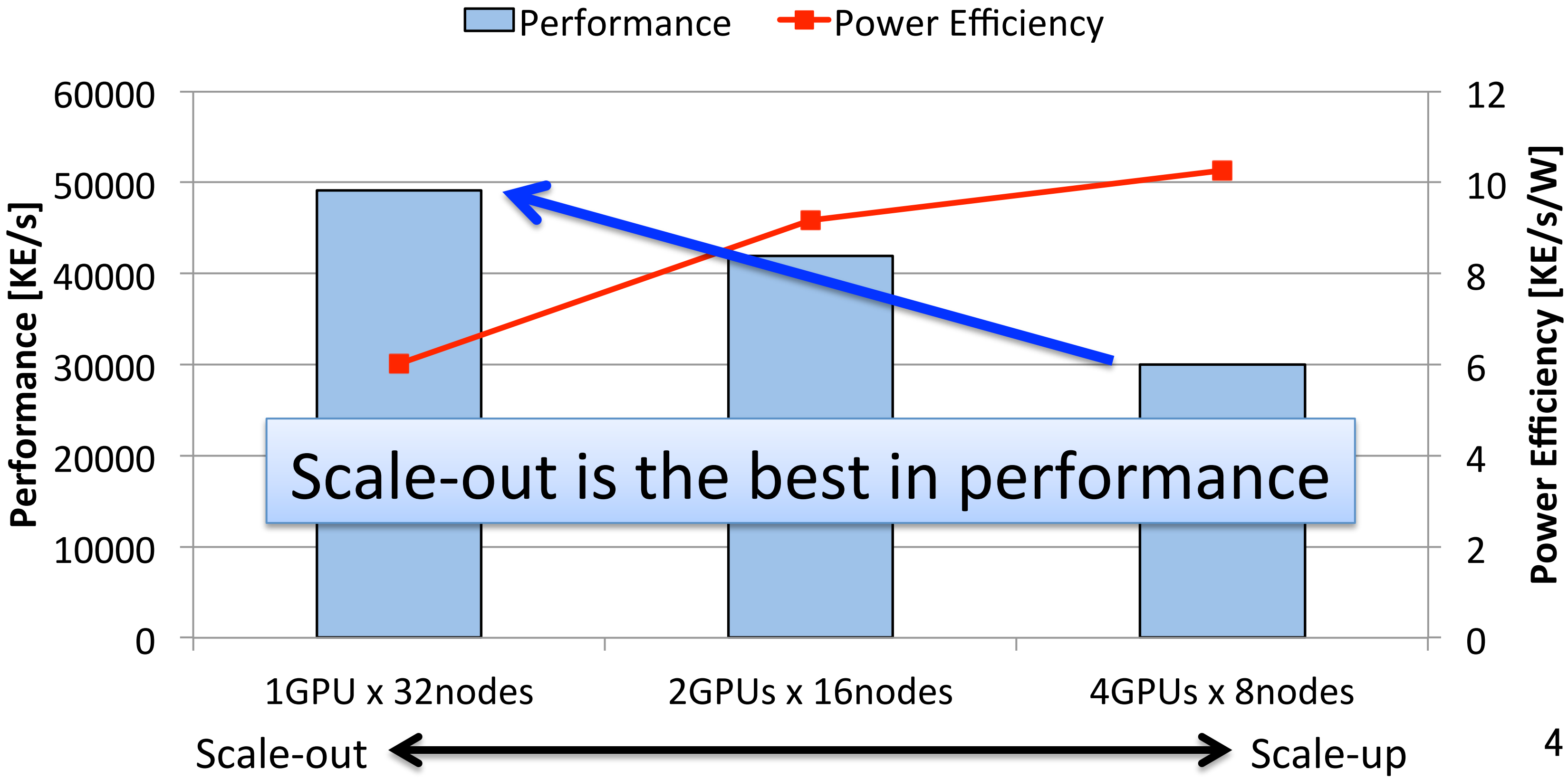
Performance and Power Efficiency

- Experiments on **TSUBAME-KFC**
  - Scale-out: 1 GPU x 32 nodes
    - better performance
  - Scale-up: 2 GPUs x 16 nodes, 4 GPUs x 8 nodes
    - better power efficiency (1.53x on 2 GPUs, 1.71x on 4 GPUs)



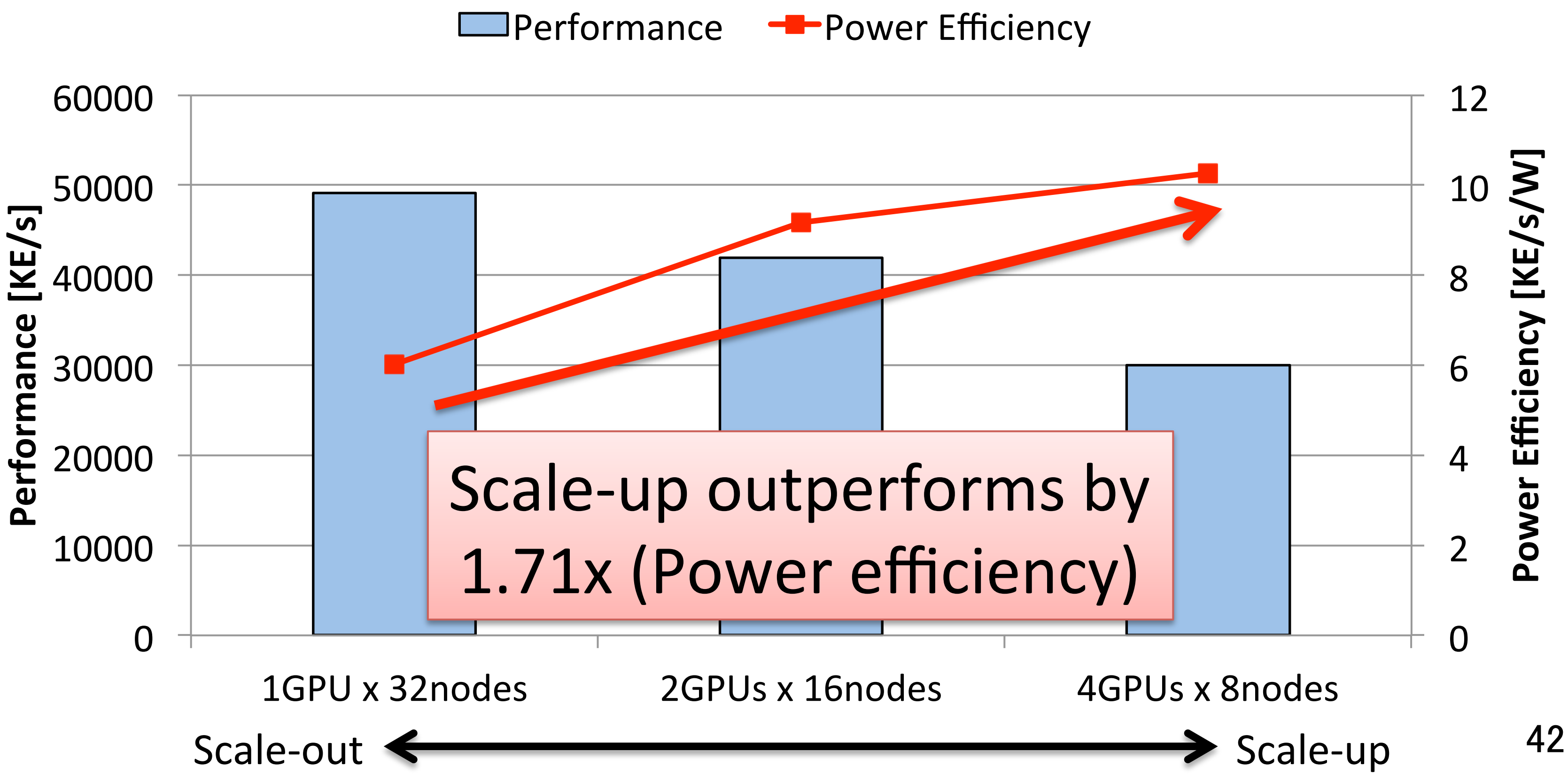
Performance and Power Efficiency

- Experiments on **TSUBAME-KFC**
  - Scale-out: 1 GPU x 32 nodes
    - better performance
  - Scale-up: 2 GPUs x 16 nodes, 4 GPUs x 8 nodes
    - better power efficiency (1.53x on 2 GPUs, 1.71x on 4 GPUs)



Performance and Power Efficiency

- Experiments on **TSUBAME-KFC**
  - Scale-out: 1 GPU x 32 nodes
    - better performance
  - Scale-up: 2 GPUs x 16 nodes, 4 GPUs x 8 nodes
    - better power efficiency (1.53x on 2 GPUs, 1.71x on 4 GPUs)



Summary of Experiments

- Performance
  - Scales well up to 1024 nodes (3072 GPUs) when data size is larger than GPU memory capacity
  - 2.10x speedup using 3GPUs per node compared with 2CPUs per node

  → **Out-of-core GPU memory management can accelerate by fully overlapping CPU-GPU data transfer and applying several optimizations**
- Efficiency
  - 1.71x better power efficiency by scale-up strategy (using 4GPUs per node)

  → **Scale-up approach performs better power efficiency than simple scale-out approach**
- Limitation
  - May not perform efficiently on graphs with different characteristics
    - e.g.) road network (only 4 edges per vertex)

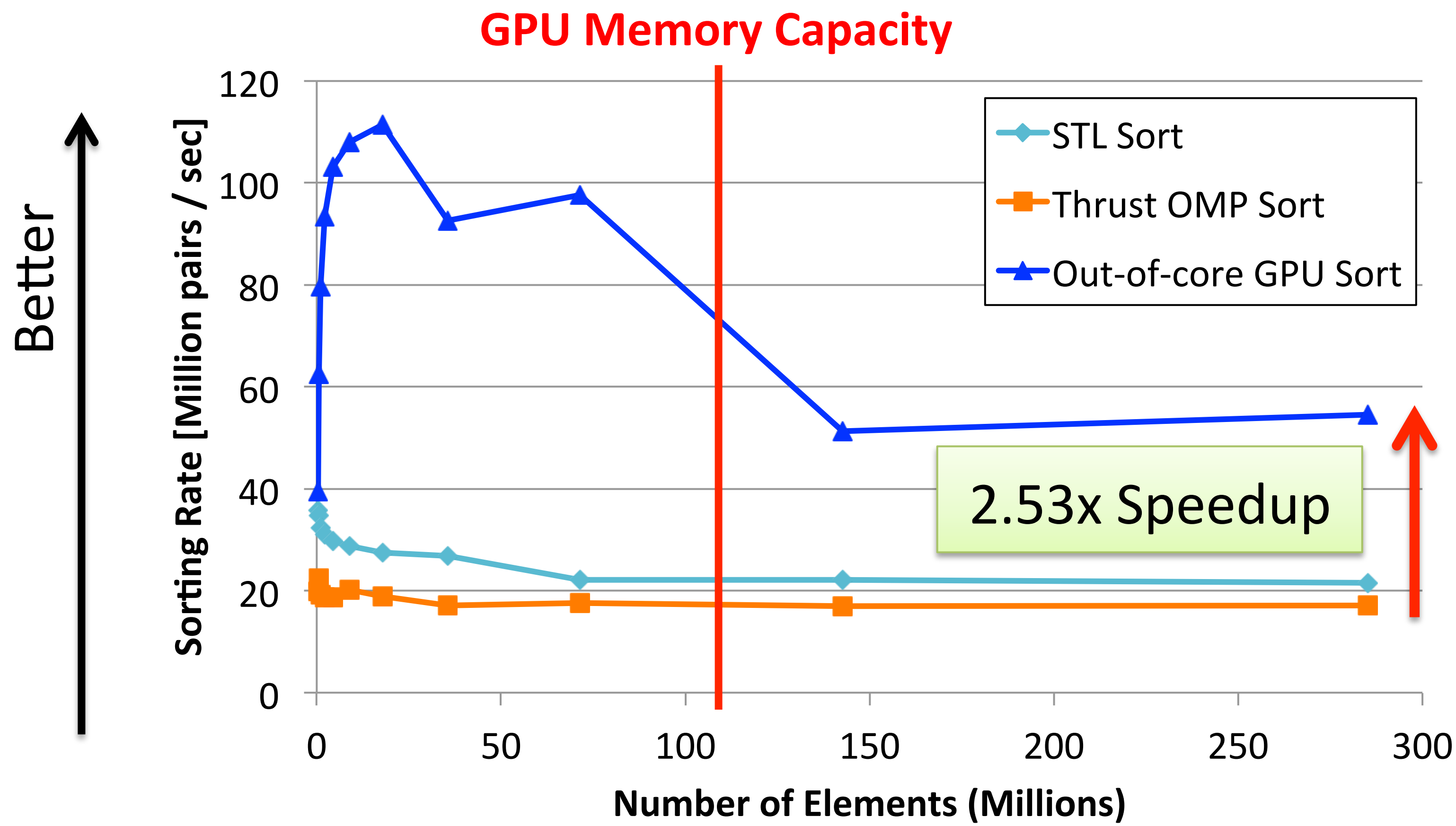
Conclusions

- **Out-of-core GPU memory management for MapReduce-based graph processing**
  - Methodology
    - Out-of-core GPU data management for GPU-MapReduce-based large-scale graph processing
    - Implement out-of-core GPU sorting
    - Investigate the balance of scale-up and scale-out approaches
  - Performance
    - 2.10x speedup than CPU on SCALE 34 (1024 nodes, 3072 GPUs)
    - 1.71x power efficiency by scale-up strategy
- **Future work**
  - Handling host memory overflow by utilizing I/O from Non-Volatile Memory

- backup

Result of Out-of-core GPU Sorting

- Comparison of our out-of-core sorting on 1 GPU with OpenMP sorting on 1 CPU
- 2.53x speedup compared with CPU when data size is larger than GPU memory capacity



Balance between Scale-up and Scale-out

- Performance difference by number of GPUs per node
  - 1 GPU x 1024 nodes, 2 GPUs x 512 nodes, 3 GPUs x 512 nodes
  - 2 GPUs performs 81.3 %, 3 GPUs performs 96.9 % of 1 GPU with double number of nodes