Load Imbalance Mitigation Optimizations for GPU-Accelerated Similarity Joins

Benoit Gallet, Michael Gowanlock
benoit.gallet@nau.edu, michael.gowanlock@nau.edu

Northern Arizona University
School of Informatics, Computing and Cyber Systems

5th HPBDC Workshop, Rio de Janeiro, Brazil, May 20th, 2019

Introduction

Introduction

Given a dataset D in n dimensions

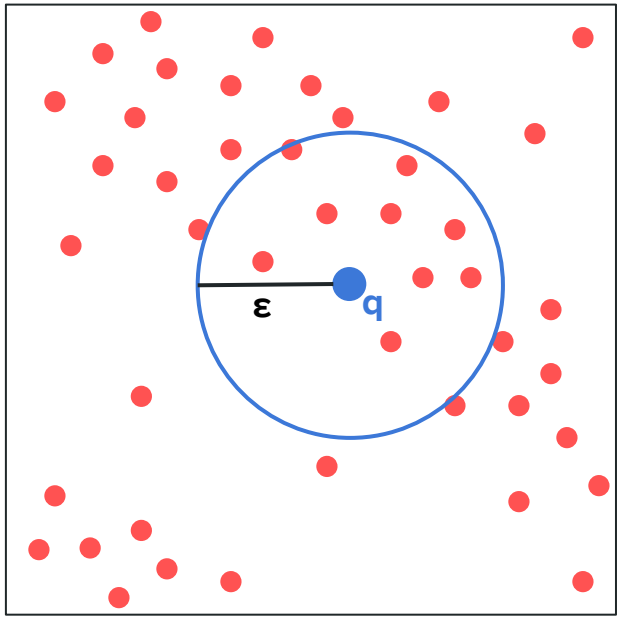
- Similarity self-join ➜ Find pairs of objects in D whose similarity is within a threshold ($D \bowtie D$)
- Similarity defined by a predicate or a metric

Introduction

Given a dataset *D* in *n* dimensions

- Similarity self-join ➔ Find pairs of objects in *D* whose similarity is within a threshold ($D \bowtie D$)
- Similarity defined by a predicate or a metric

- Distance similarity self-join ➔ Find pairs of object within a distance ε
  - e.g.: Euclidean distance ($D \bowtie_{\varepsilon} D$)
- Range Query: Compute distances between a query point *q* and its candidate points *c*
  - Distance similarity self-join = |*D*| range queries



Introduction

- Brute force method: nested for loops
  - Complexity ≃ O($|D|^2$)
- Use an indexing method to prune the search space
  - Complexity ≃ between O($|D| \times \log |D|$) and O($|D|^2$)

Introduction

- Brute force method: nested for loops
  - Complexity ≃ $O(|D|^2)$
- Use an indexing method to prune the search space
  - Complexity ≃ between $O(|D| \times \log |D|)$ and $O(|D|^2)$
- Hierarchical structures
  - R-Tree, X-Tree, k-D Tree, B-Tree, etc.
- Non-hierarchical structures
  - Grids, space filling curves, etc.

Introduction

- Brute force method: nested for loops
  - Complexity ≃ $O(|D|^2)$
- Use an indexing method to prune the search space
  - Complexity ≃ between $O(|D| \times \log |D|)$ and $O(|D|^2)$
- Hierarchical structures
  - R-Tree, X-Tree, k-D Tree, B-Tree, etc.
- Non-hierarchical structures
  - Grids, space filling curves, etc.
- Some better for high dimensions, some better for low dimensions
- Some better for the CPU, some better for the GPU
  - Recursion, branching, size, etc.

Background

Background

Reasons to use a GPU

- Range queries are independent
  - Can be performed in parallel
- Many memory operations
  - Benefits from high-bandwidth memory on the GPU

Background

Reasons to use a GPU

- Range queries are independent
  - Can be performed in parallel
- Many memory operations
  - Benefits from high-bandwidth memory
- Lot of cores, high-memory bandwidth
  - Intel Xeon E7-8894v4 ➜ 24 physical cores, up to 85 GB/s memory bandwidth
  - Nvidia Tesla V100 ➜ 5,120 CUDA cores, up to 900 GB/s memory bandwidth

➜ The GPU is well suited for this type of application

Background

However,

- Limited global memory size*
  - 512 GB of RAM per node (256 GB per CPU)
  - 96 GB of GPU memory per node (16 GB per GPU)
- Slow Host / Device communication bandwidth
  - 16 GB/s for PCIe 3.0
  - Known as a major bottleneck
- High chance of uneven workload between points
  - Uneven computation time between threads

➜ Necessary to consider these potential issues

** Specs of the Summit supercomputer (ranked 1 in TOP500), https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/*

Background

Leverage work from previous contribution [1]

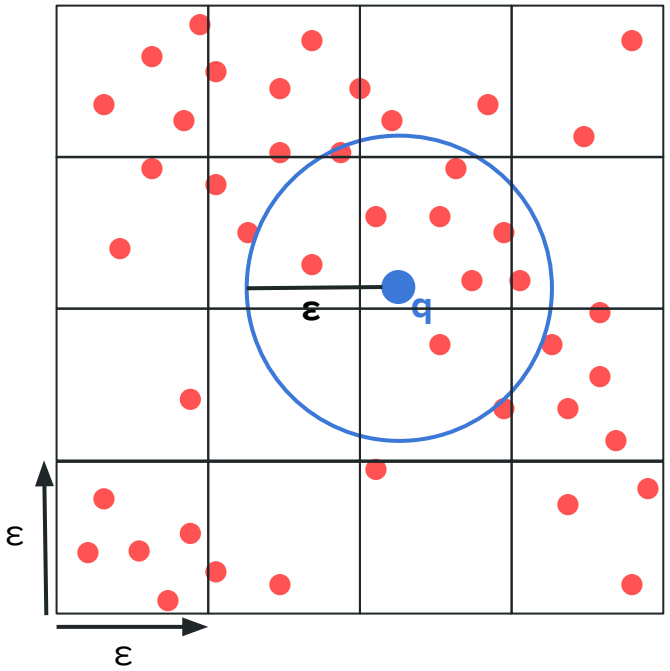
- Batching scheme
  - Splits the computation into smaller executions
  - Avoids memory overflow
  - Overlaps computation with memory transfers

[1] M. Gowanlock and B. Karsin, "GPU Accelerated Self-join for the Distance Similarity Metric," IEEE High-Performance Big Data, Deep Learning, and Cloud Computing, in Proc. of the 2018 IEEE Intl. Parallel and Distributed Processing Symposium Workshops, pp. 477–486, 2018.

Background

Leverage work from previous contribution [1]

- Batching scheme
  - Splits the computation into smaller executions
  - Avoids memory overflow
  - Overlaps computation with memory transfers
- Grid indexing
  - Cells of size ε^n
  - Only indexes non-empty cells
  - Bounds the search to 3^n adjacent cells
  - Threads check the same cell in lockstep
    - Reduces divergence

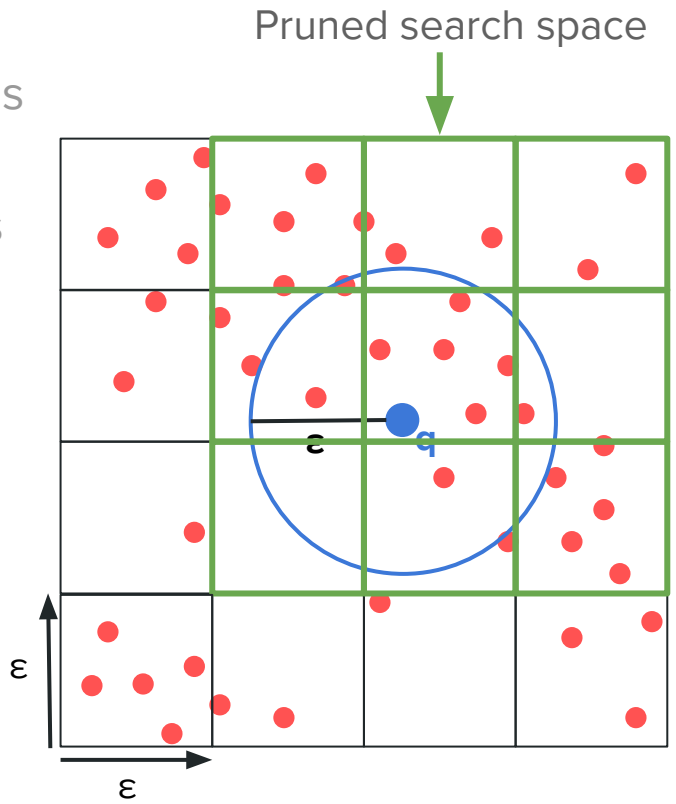


[1] M. Gowanlock and B. Karsin, "GPU Accelerated Self-join for the Distance Similarity Metric," IEEE High-Performance Big Data, Deep Learning, and Cloud Computing, in Proc. of the 2018 IEEE Intl. Parallel and Distributed Processing Symposium Workshops, pp. 477–486, 2018.

Background

Leverage work from previous contribution [1]

- Batching scheme
  - Splits the computation into smaller executions
  - Avoids memory overflow
  - Overlaps computation with memory transfers
- Grid indexing
  - Cells of size ε^n
  - Only indexes non-empty cells
  - Bounds the search to 3^n adjacent cells
  - Threads check the same cell in lockstep
    - Reduces divergence

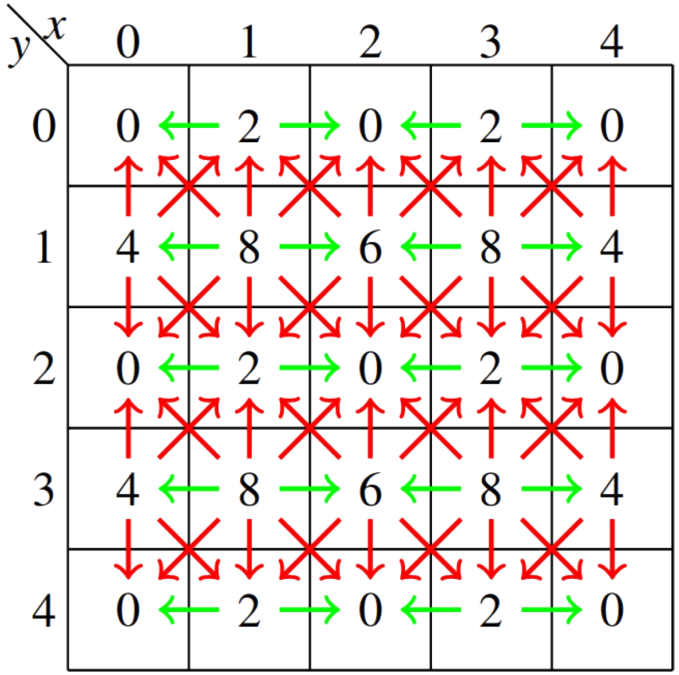


[1] M. Gowanlock and B. Karsin, "GPU Accelerated Self-join for the Distance Similarity Metric," IEEE High-Performance Big Data, Deep Learning, and Cloud Computing, in Proc. of the 2018 IEEE Intl. Parallel and Distributed Processing Symposium Workshops, pp. 477–486, 2018.

Background

Leverage work from previous contribution [1]

- Unidirectional Comparison: Unicomp
  - Euclidean distance is a symmetric function
  - $p, q \in D$, distance(p, q) = distance(q, p)
  - Only look at some of the neighboring cells
  - ➔ Only computes the distance once

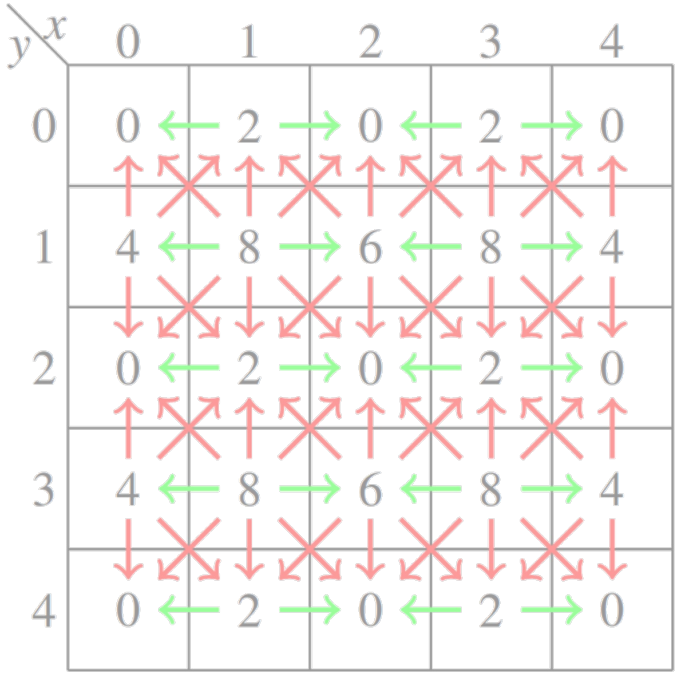


[1] M. Gowanlock and B. Karsin, "GPU Accelerated Self-join for the Distance Similarity Metric," IEEE High-Performance Big Data, Deep Learning, and Cloud Computing, in Proc. of the 2018 IEEE Intl. Parallel and Distributed Processing Symposium Workshops, pp. 477–486, 2018.

Background

Leverage work from previous contribution [1]

- Unidirectional Comparison: Unicomp
  - Euclidean distance is a symmetric function
  - $p, q \in D$, distance(p, q) = distance(q, p)
  - Only look at some of the neighboring cells
  - ➔ Only computes the distance once

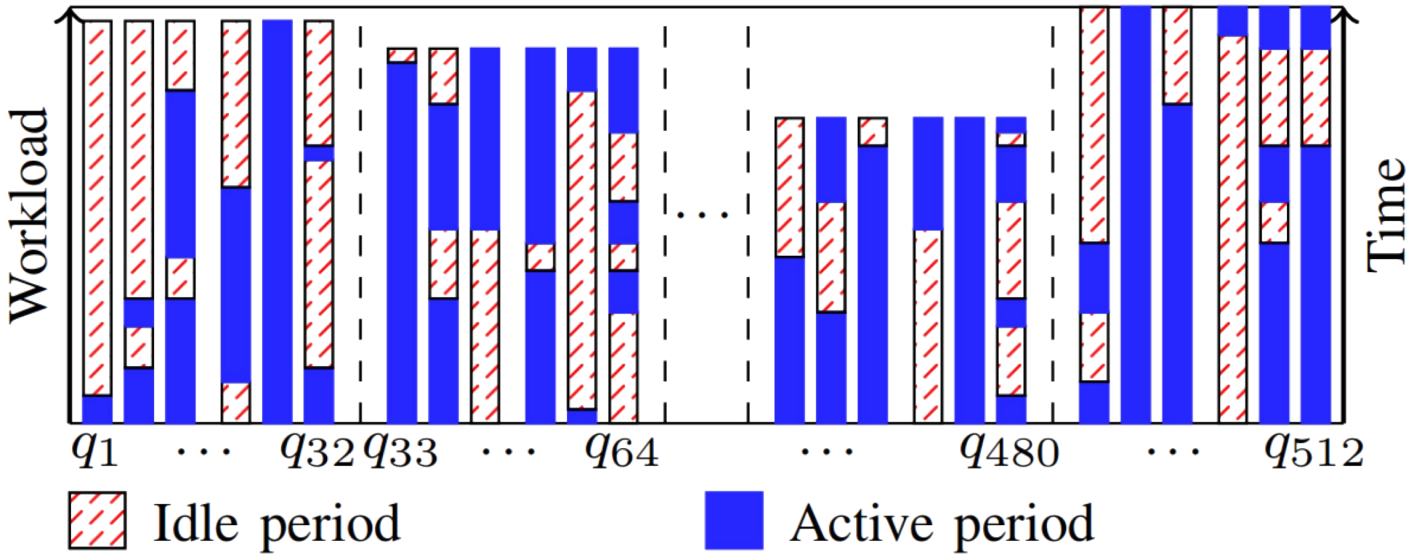


- GPU Kernel
  - Computes the ε-neighborhood of each query point
  - A thread is assigned a single query point
  - $|D|$ threads in total

[1] M. Gowanlock and B. Karsin, "GPU Accelerated Self-join for the Distance Similarity Metric," IEEE High-Performance Big Data, Deep Learning, and Cloud Computing, in Proc. of the 2018 IEEE Intl. Parallel and Distributed Processing Symposium Workshops, pp. 477–486, 2018.

Issue

- Depending on data characteristics ➜ different workload between threads
  - SIMT architecture of the GPU ➜ threads executed by groups of 32 (warps)
  - Different workloads ➜ idle time for some of the threads within a warp

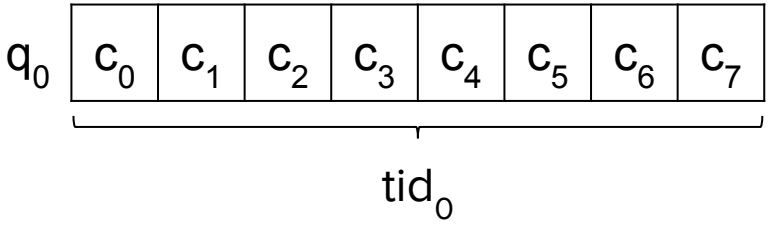


Optimizations

- Range Query Granularity Increase
- Local and Global Load Balancing
- Cell Access Pattern
- Warp Execution Scheduling

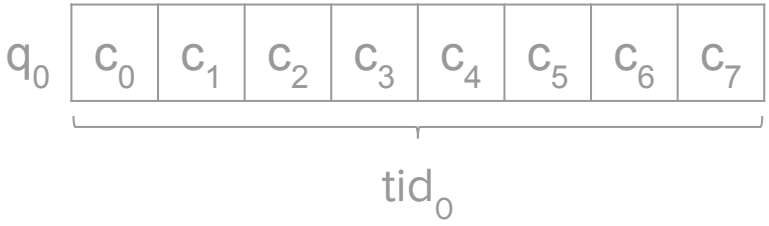
Range Query Granularity Increase: $k > 1$

- Original kernel → 1 thread per query point

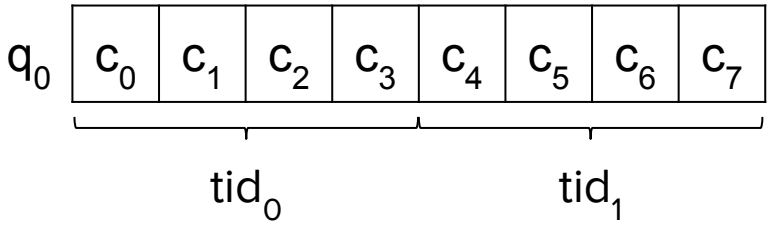


Range Query Granularity Increase: $k > 1$

- Original kernel → 1 thread per query point

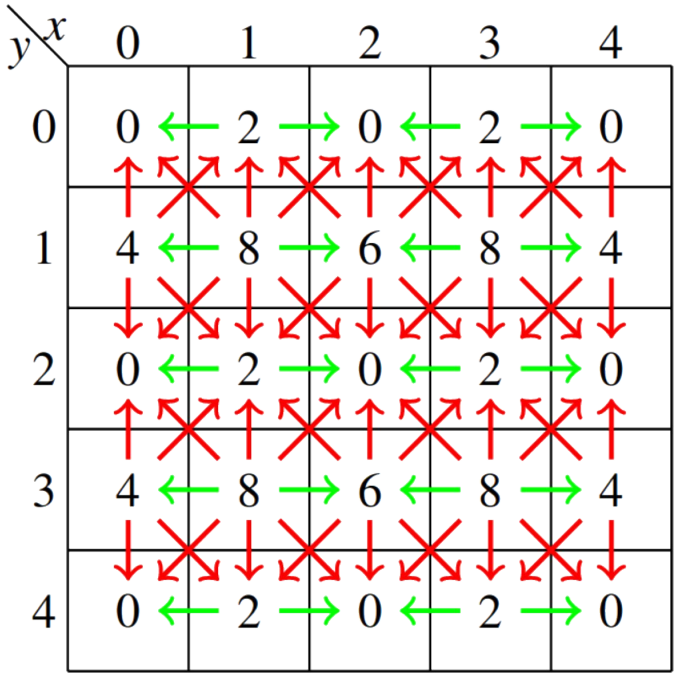


- Use multiple threads per query point
  - Each thread assigned to the query point *q* computes a fraction of the candidate points *c*
  - *k* =number of threads assigned to each query point



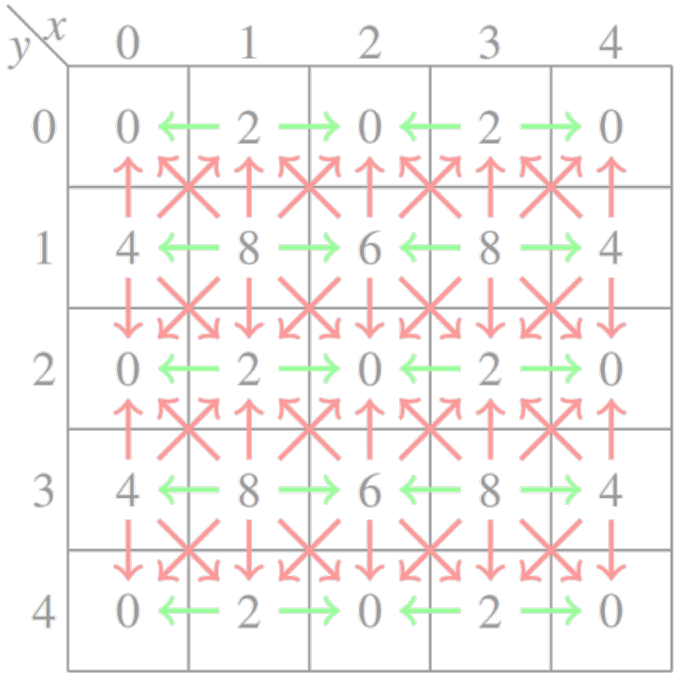
Cell Access Pattern: Lid-Unicomp

- Unidirectional comparison (Unicomp)
  - Potential load imbalance between cells

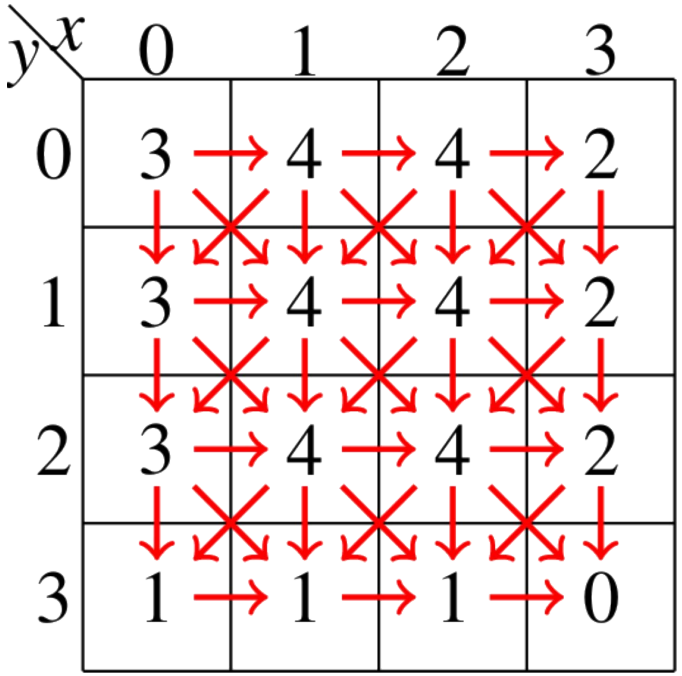


Cell Access Pattern: Lid-Unicomp

- Unidirectional comparison (Unicomp)
  - Potential load imbalance between cells

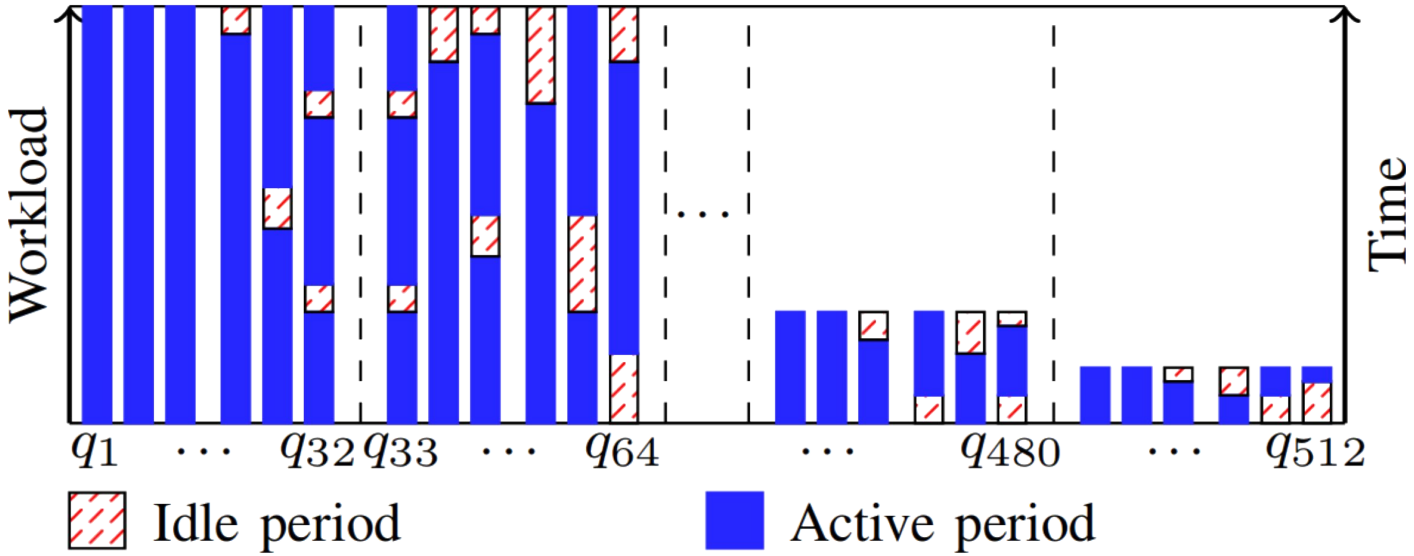


- Linear ID unidirectional comparison (Lid-Unicomp)
  - Based on cells' linear id
  - Compare to cells with greater linear id



Local and Global Load Balancing: SortByWL

- Sort the points from most to least workload
  - Reduces intra-warp load imbalance
  - Reduces block-level load imbalance



Warp Execution Scheduling: WorkQueue

- Sorting points does not guarantee their execution order
  - GPU's physical scheduler
- Force warp execution order with a work queue
  - Each thread atomically takes the available point with the most work

D the original dataset

Warp Execution Scheduling: WorkQueue

- Sorting points does not guarantee their execution order
  - GPU's physical scheduler
- Force warp execution order with a work queue
  - Each thread atomically takes the available point with the most work

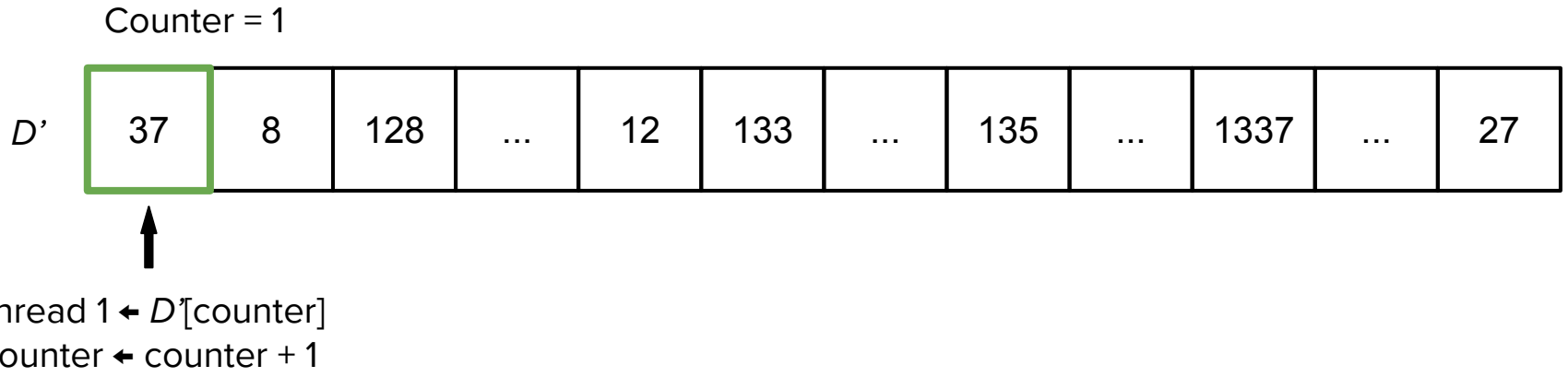
D' the original dataset sorted by workload

D'	37	8	128	...	12	133	...	135	...	1337	...	27

Warp Execution Scheduling: WorkQueue

- Sorting points does not guarantee their execution order
  - GPU's physical scheduler
- Force warp execution order with a work queue
  - Each thread atomically takes the available point with the most work

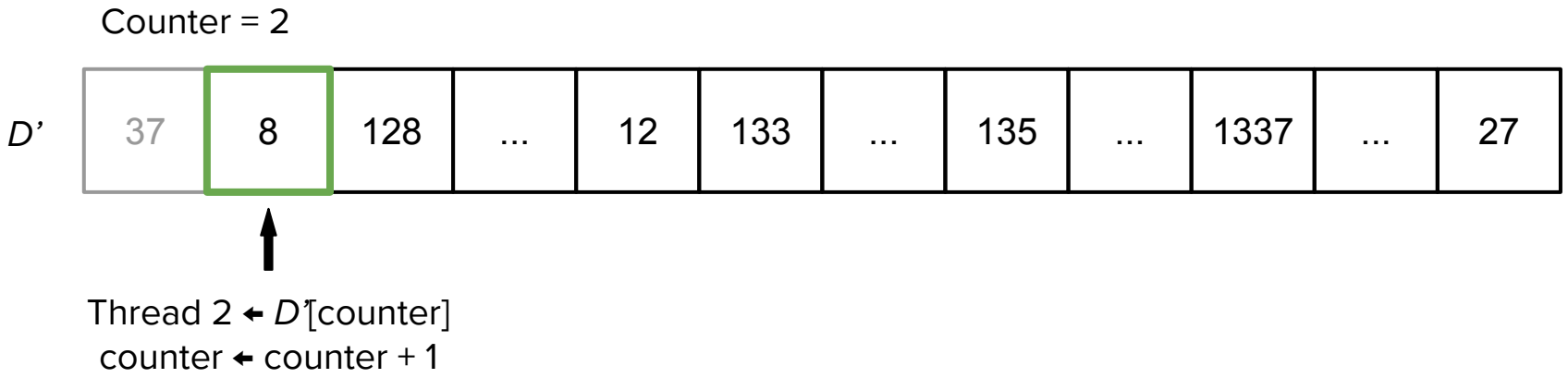
Thread *i* → *i*th thread to be executed



Warp Execution Scheduling: WorkQueue

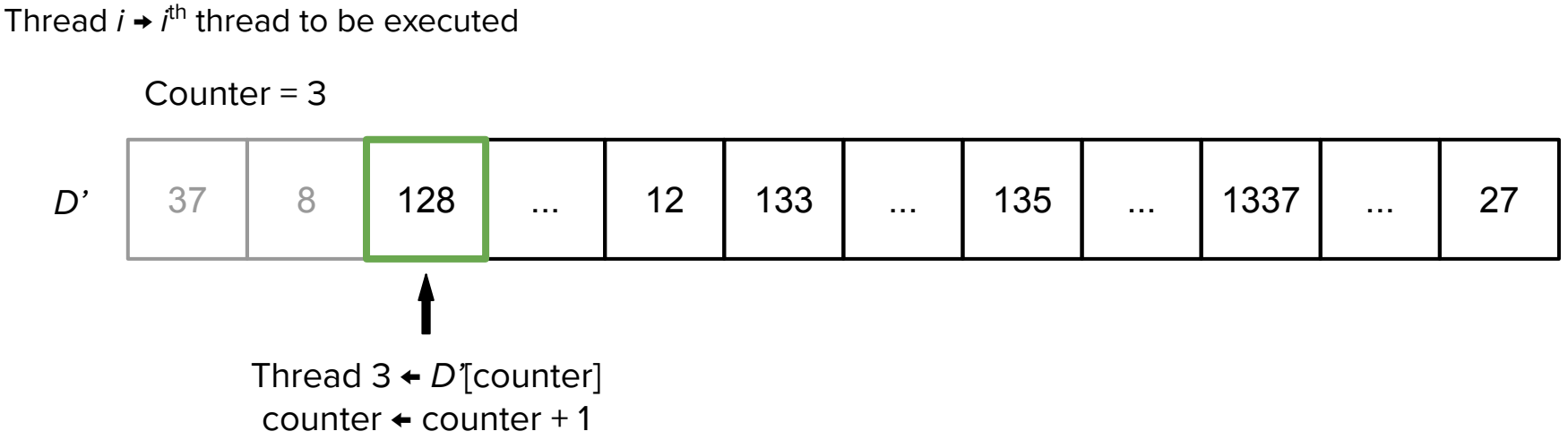
- Sorting points does not guarantee their execution order
  - GPU's physical scheduler
- Force warp execution order with a work queue
  - Each thread atomically takes the available point with the most work

Thread *i* → *i*th thread to be executed



Warp Execution Scheduling: WorkQueue

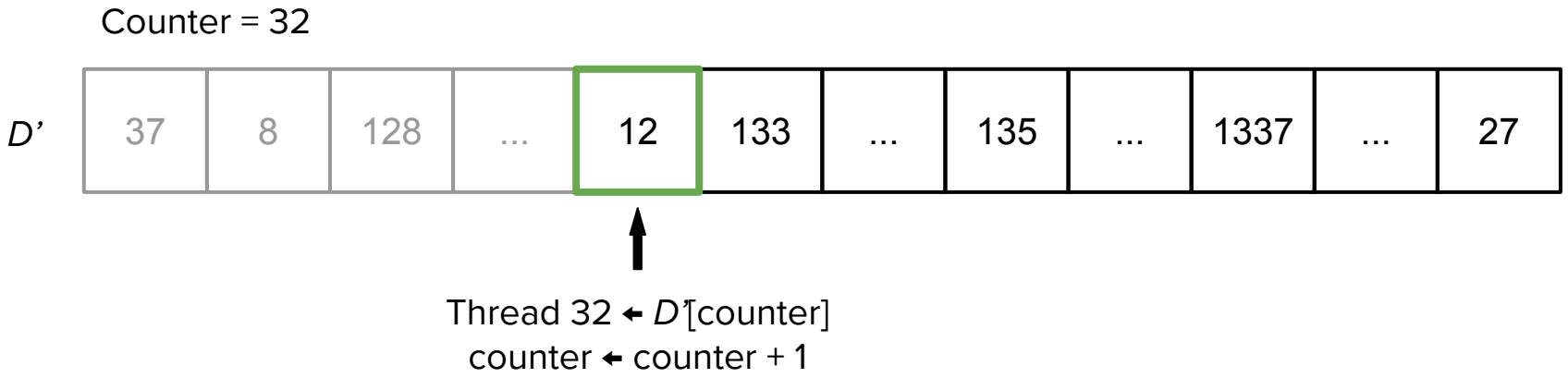
- Sorting points does not guarantee their execution order
  - GPU's physical scheduler
- Force warp execution order with a work queue
  - Each thread atomically takes the available point with the most work



Warp Execution Scheduling: WorkQueue

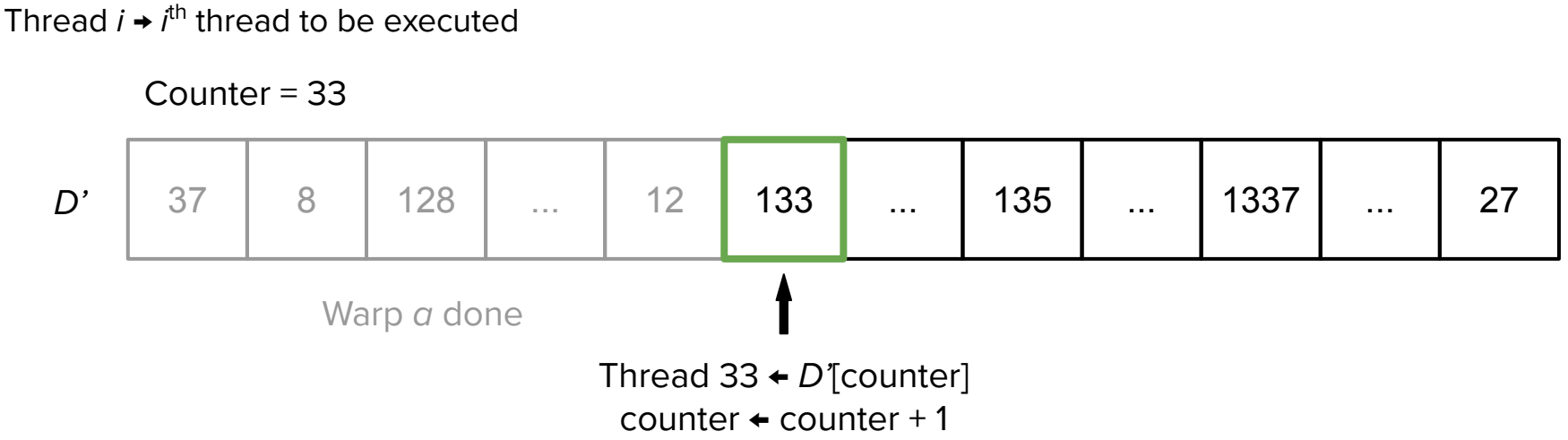
- Sorting points does not guarantee their execution order
  - GPU's physical scheduler
- Force warp execution order with a work queue
  - Each thread atomically takes the available point with the most work

Thread *i* → *i*th thread to be executed



Warp Execution Scheduling: WorkQueue

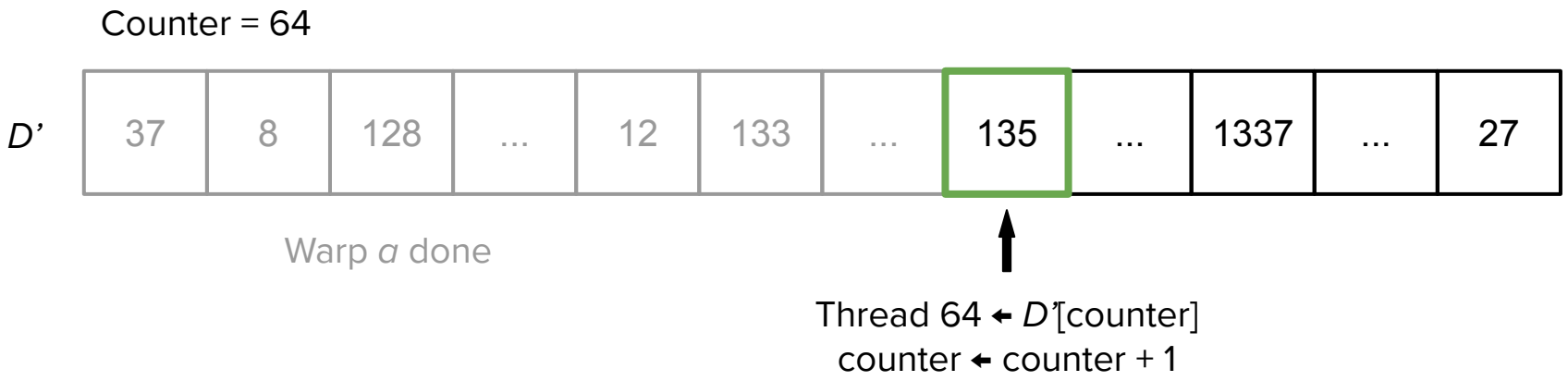
- Sorting points does not guarantee their execution order
  - GPU's physical scheduler
- Force warp execution order with a work queue
  - Each thread atomically takes the available point with the most work



Warp Execution Scheduling: WorkQueue

- Sorting points does not guarantee their execution order
  - GPU's physical scheduler
- Force warp execution order with a work queue
  - Each thread atomically takes the available point with the most work

Thread *i* → *i*th thread to be executed



Warp Execution Scheduling: WorkQueue

- Sorting points does not guarantee their execution order
  - GPU's physical scheduler
- Force warp execution order with a work queue
  - Each thread atomically takes the available point with the most work

Thread *i* → *i*th thread to be executed

Warp Execution Scheduling: WorkQueue

- Sorting points does not guarantee their execution order
  - GPU's physical scheduler
- Force warp execution order with a work queue
  - Each thread atomically takes the available point with the most work

Thread *i* → *i*th thread to be executed

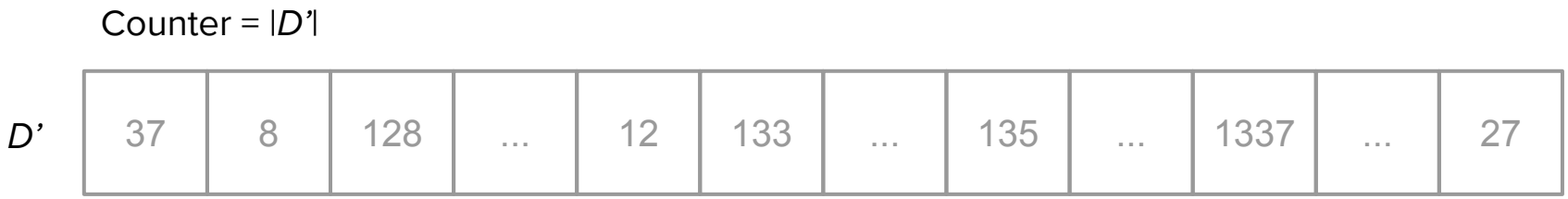
Warp Execution Scheduling: WorkQueue

- Sorting points does not guarantee their execution order
  - GPU's physical scheduler
- Force warp execution order with a work queue
  - Each thread atomically takes the available point with the most work

Thread i → i^{th} thread to be executed

Warp Execution Scheduling: WorkQueue

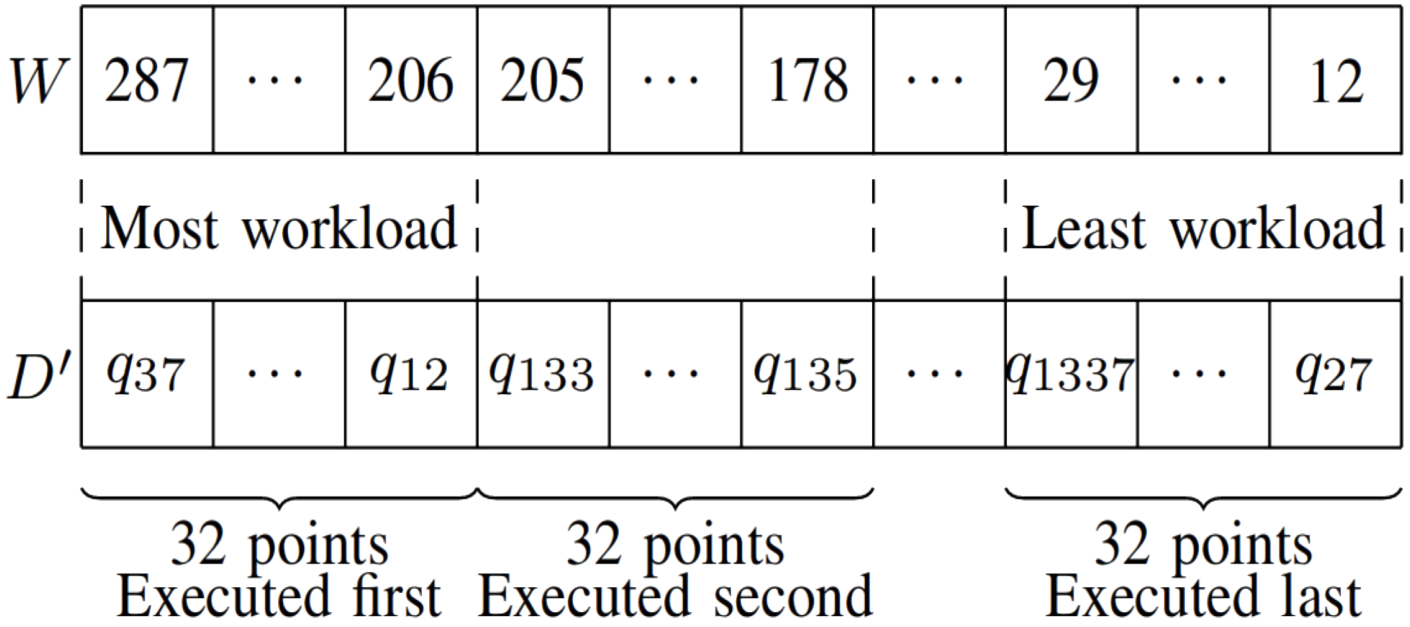
- Sorting points does not guarantee their execution order
  - GPU's physical scheduler
- Force warp execution order with a work queue
  - Each thread atomically takes the available point with the most work



Computation done

Warp Execution Scheduling: WorkQueue

- Use a work queue to assign query points to threads
  - Ensures a similar workload within a warp
  - Ensures a similar workload within a batch



Further explained during the poster session of the IPDPS PhD Forum, Wednesday

Experimental Evaluation

Experimental Evaluation

- Uniformly and exponentially distributed synthetic datasets
  - 2 to 6 dimensions
  - 2M points
  - Represent uniform and very different workloads
- Real world datasets
  - Space Weather: 2 and 3 dimensions, 1.86M and 5M points
  - 50M points from the Gaia catalog
- Platform used
  - 2 x Intel Xeon E5-2620v4@2.10 GHz (16 physical cores) + 128 GB of RAM
  - Nvidia Quadro P100 (16 GB of global memory)

Experimental Evaluation

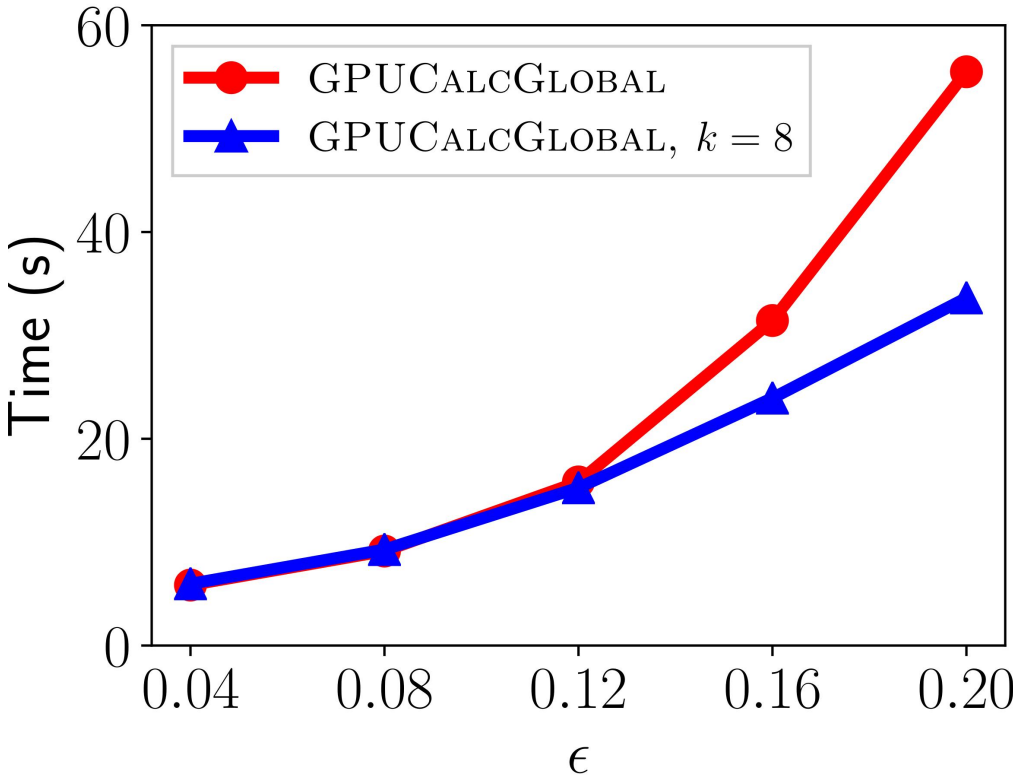
- Uniformly and exponentially distributed synthetic datasets
  - 2 to 6 dimensions
  - 2M points
  - Represent uniform and very different workloads
- Real world datasets
  - Space Weather: 2 and 3 dimensions, 1.86M and 5M points
  - 50M points from the Gaia catalog
- Platform used
  - 2 x Intel Xeon E5-2620v4@2.10 GHz (16 physical cores) + 128 GB of RAM
  - Nvidia Quadro P100 (16 GB of global memory)
- Code in C/C++ and CUDA,compiled with O3 flag
- GPU implementations: 256 threads per block, 64-bit floating point values
  - GPUCalcGlobal: original GPU kernel from previous work
- CPU implementation: 16 threads, 32-bit floating point values
  - Super-EGO: state-of-the-art parallel CPU algorithm

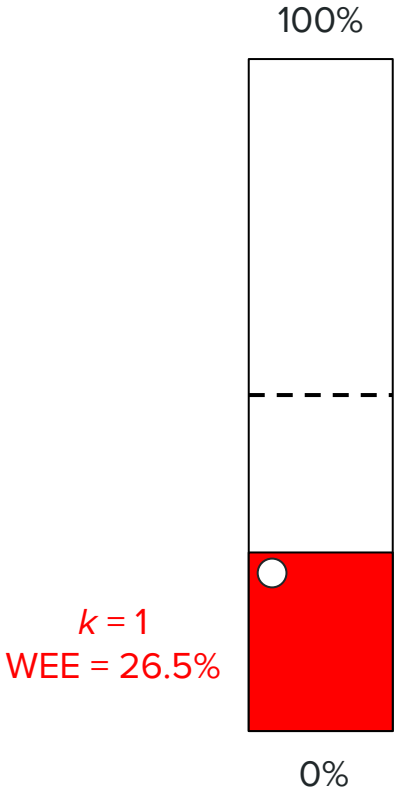
Experimental Evaluation

- Metrics used: time and warp execution efficiency (WEE)
  - Time: execution time of the application
    - Includes memory allocations, transfers, computation, etc.
    - Does not include index construction time (not the focus of this work)
  - Warp execution efficiency
    - Average percentage of active threads in each executed warp
    - Increasing it increases utilization of the GPU's compute resources
    - Lowered by divergent branches
    - Good indicator of workload balancing
      - Higher percentage means similar workload

Experimental Evaluation: $k = 8$

- Comparison between $k = 1$ and $k = 8$ using GPUCalcGlobal

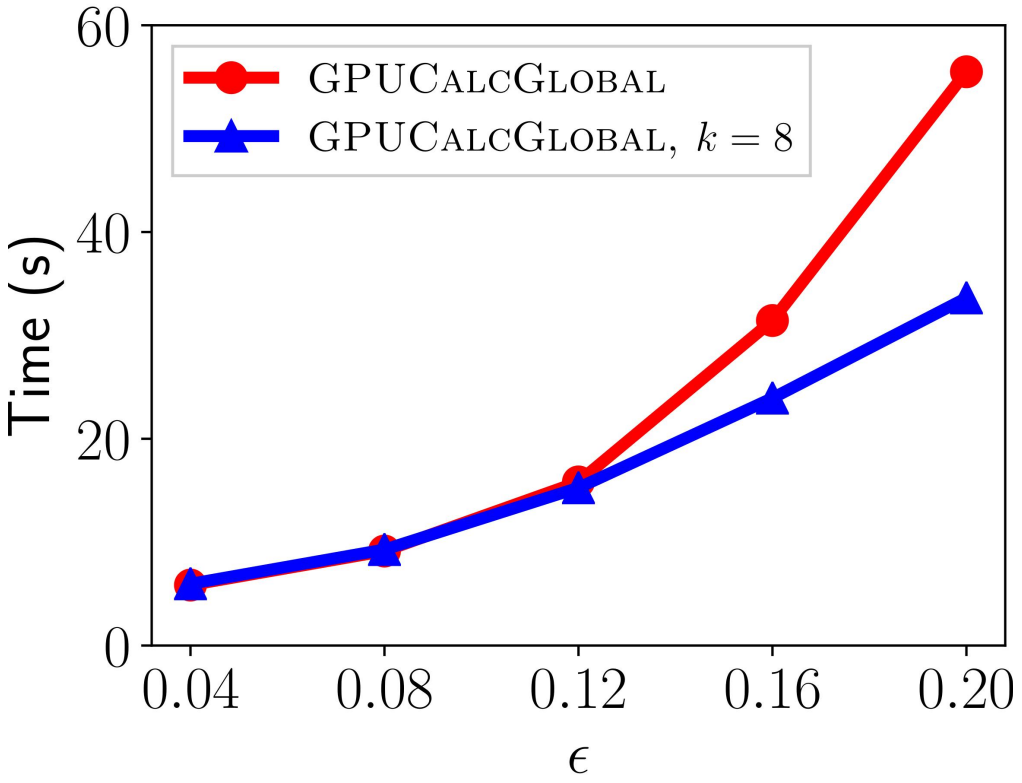


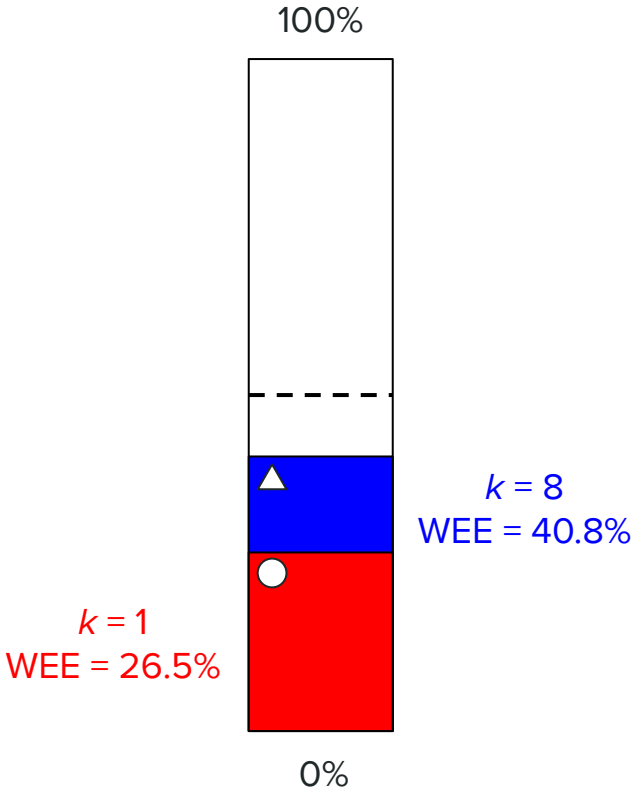


Dataset: Expo2D2M, Warp execution efficiency: ε = 0.2

Experimental Evaluation: $k = 8$

- Comparison between $k = 1$ and $k = 8$ using GPUCalcGlobal

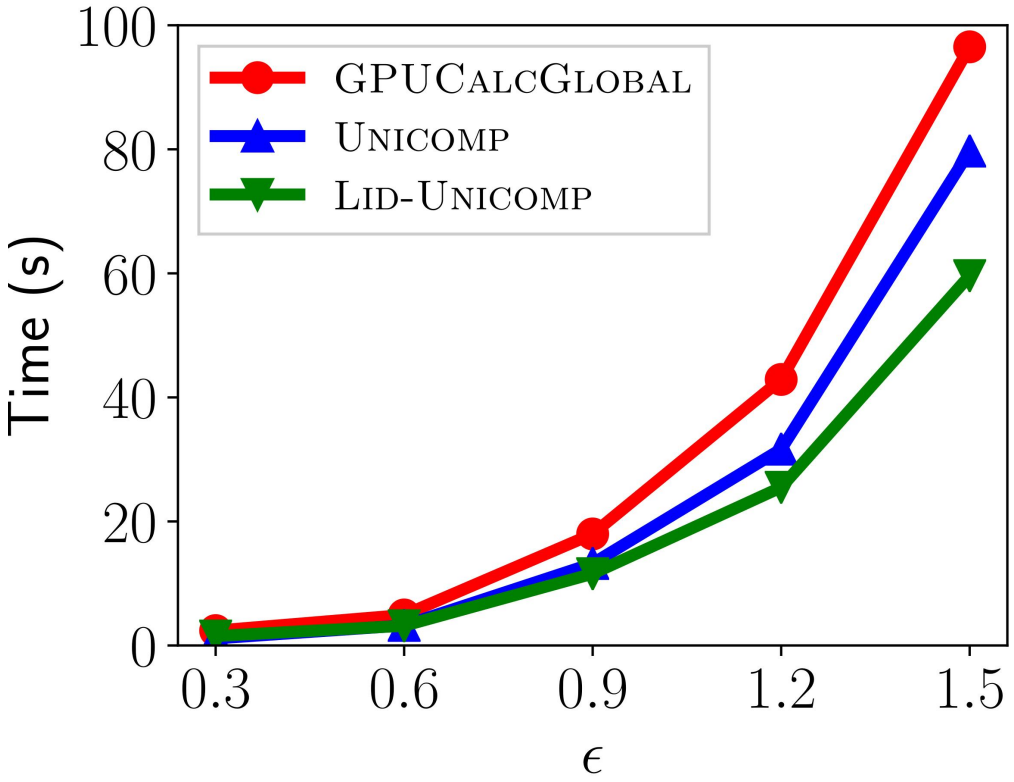


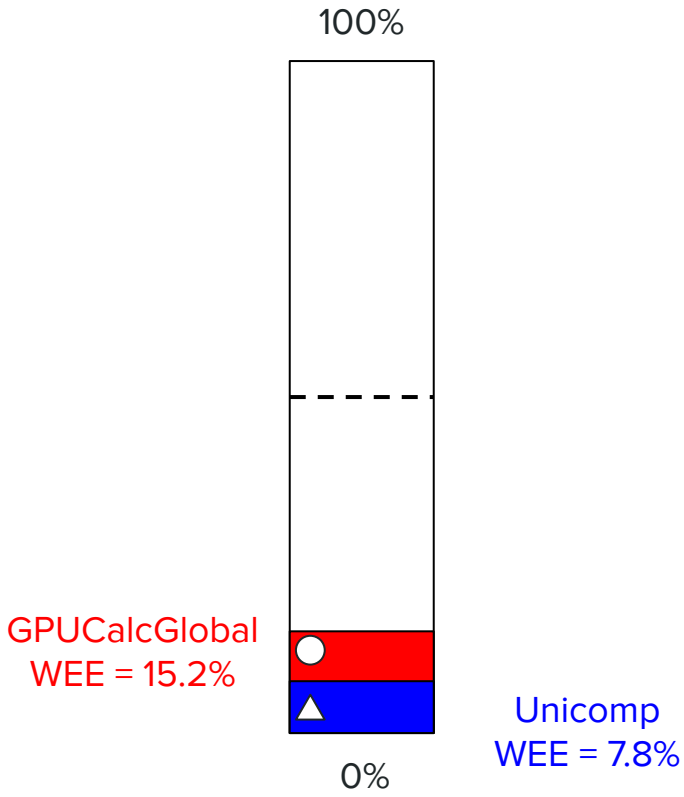


Dataset: Expo2D2M, Warp execution efficiency: ε = 0.2

Experimental Evaluation: Lid-Unicomp

- Comparison between GPUCalcGlobal, Unicomp and Lid-Unicomp
  - Note: Unicomp and Lid-Unicomp perform half the distance calculations of GPUCalcGlobal

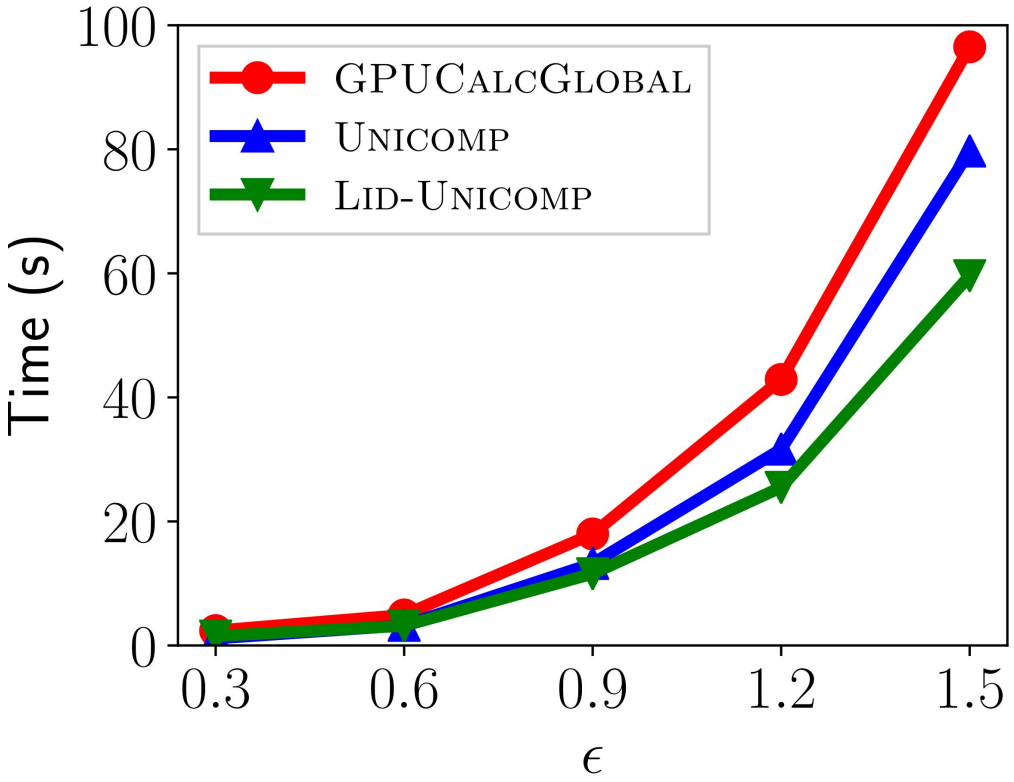


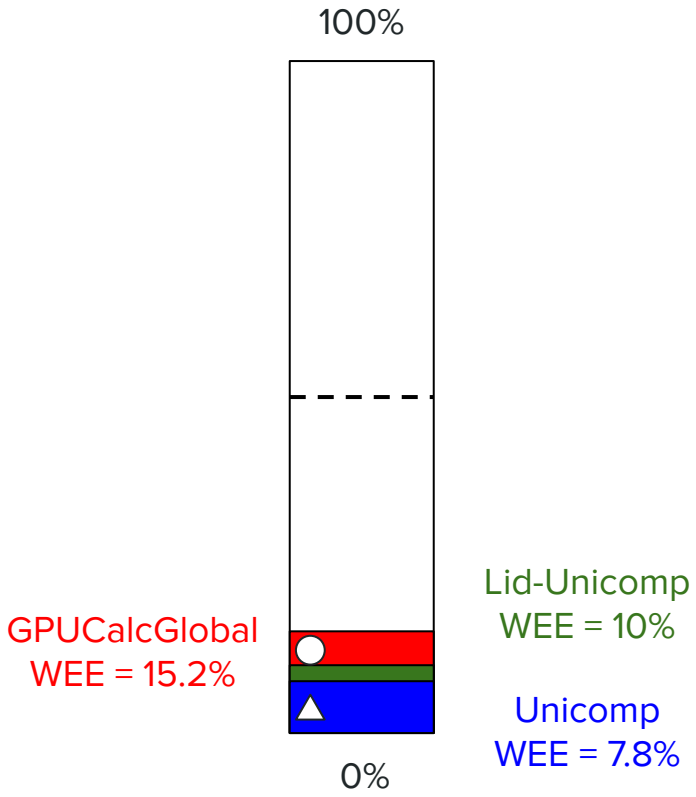


Dataset: Expo6D2M, Warp execution efficiency: ε = 1.2

Experimental Evaluation: Lid-Unicomp

- Comparison between GPUCalcGlobal, Unicomp and Lid-Unicomp
  - Note: Unicomp and Lid-Unicomp perform half the distance calculations of GPUCalcGlobal

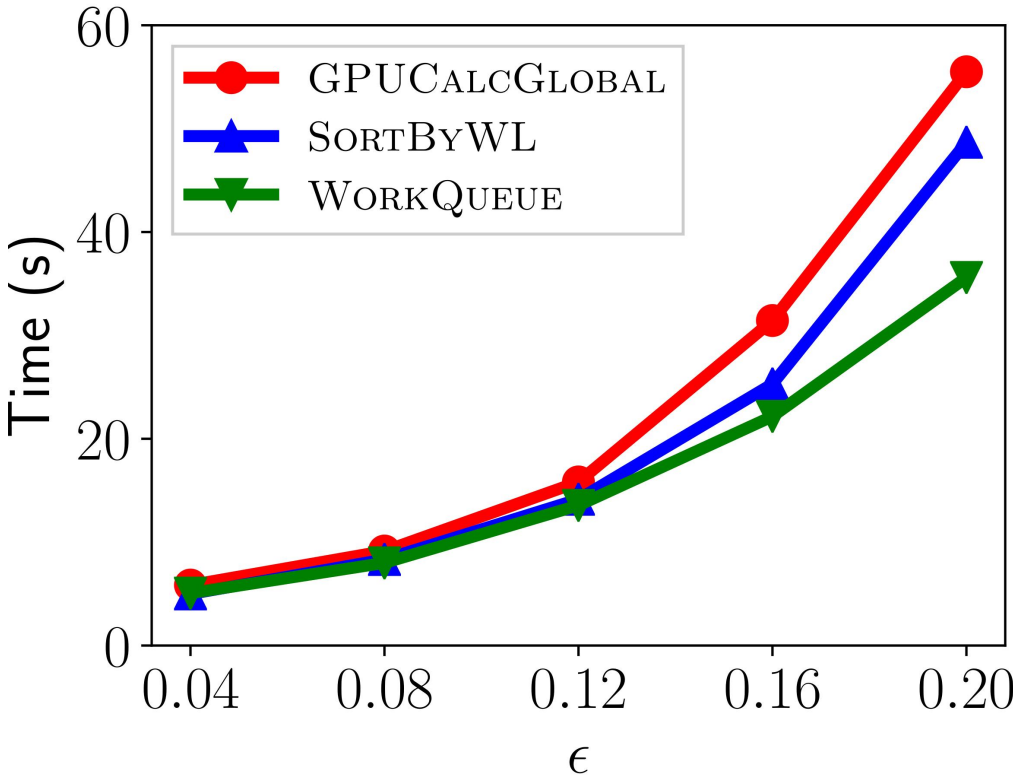


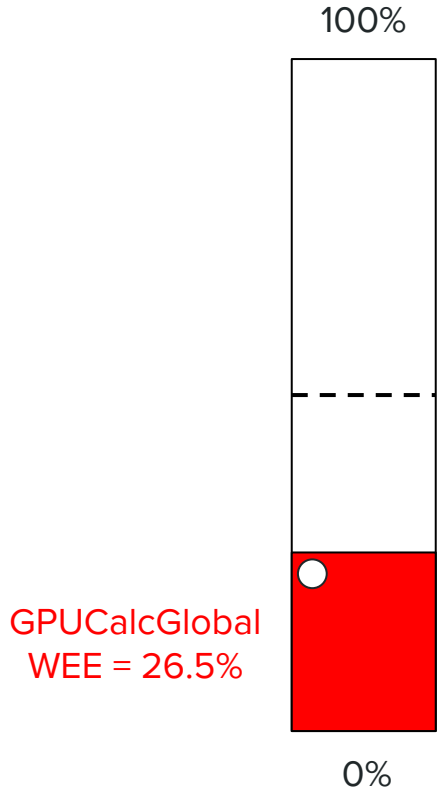


Dataset: Expo6D2M, Warp execution efficiency: ε = 1.2

Experimental Evaluation: SortByWL and WorkQueue

- Comparison between GPUCalcGlobal, SortByWL and WorkQueue

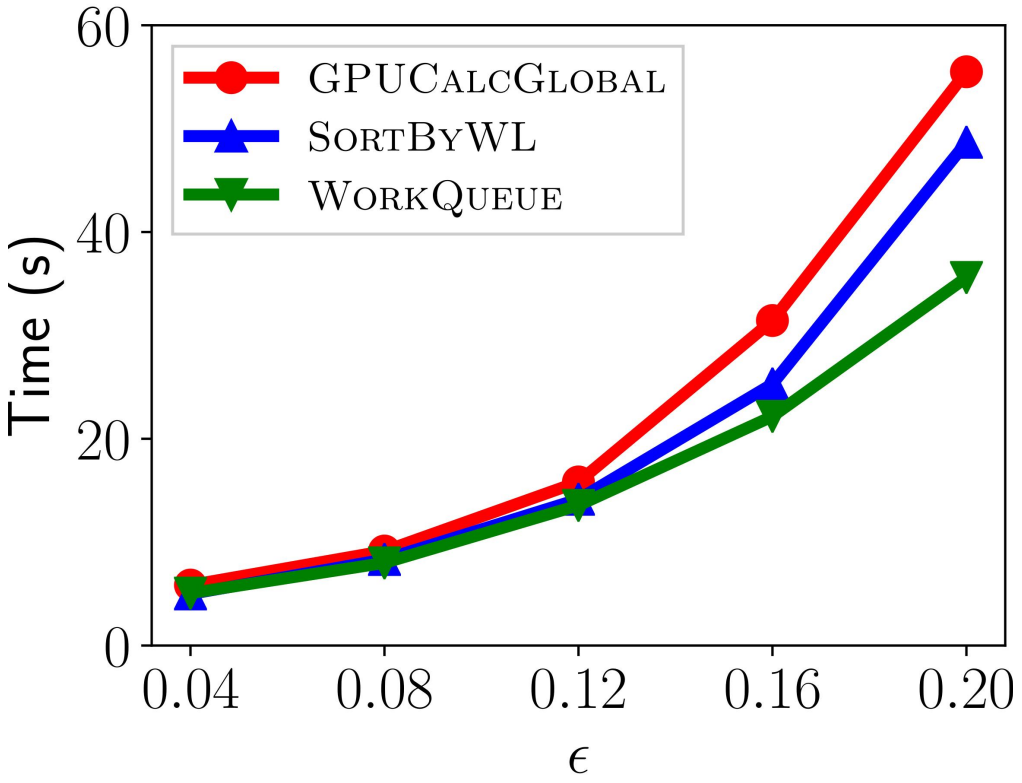


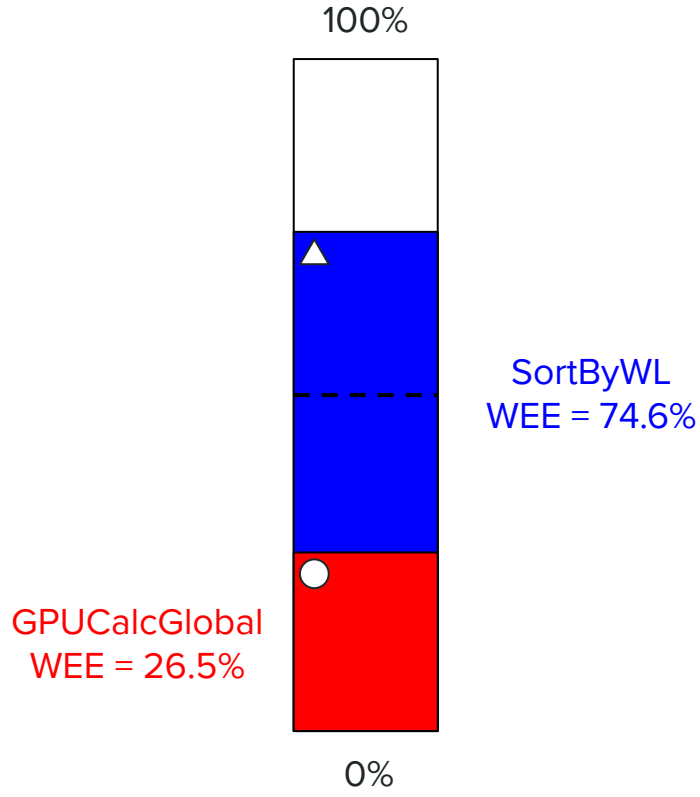


Dataset: Expo2D2M, Warp execution efficiency: ε = 0.2

Experimental Evaluation: SortByWL and WorkQueue

- Comparison between GPUCalcGlobal, SortByWL and WorkQueue

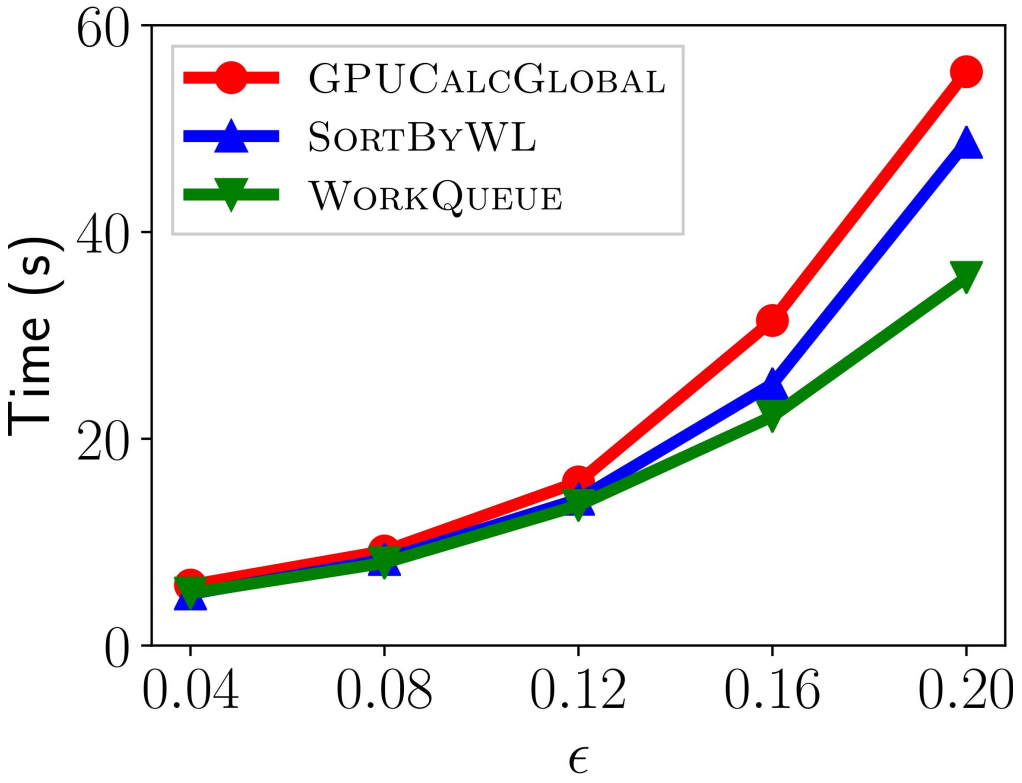


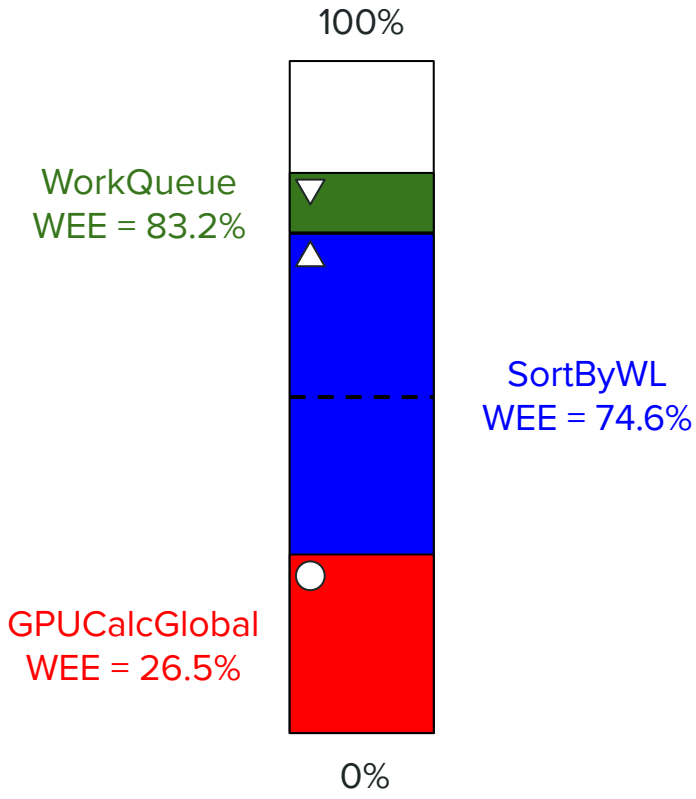


Dataset: Expo2D2M, Warp execution efficiency: ε = 0.2

Experimental Evaluation: SortByWL and WorkQueue

- Comparison between GPUCalcGlobal, SortByWL and WorkQueue

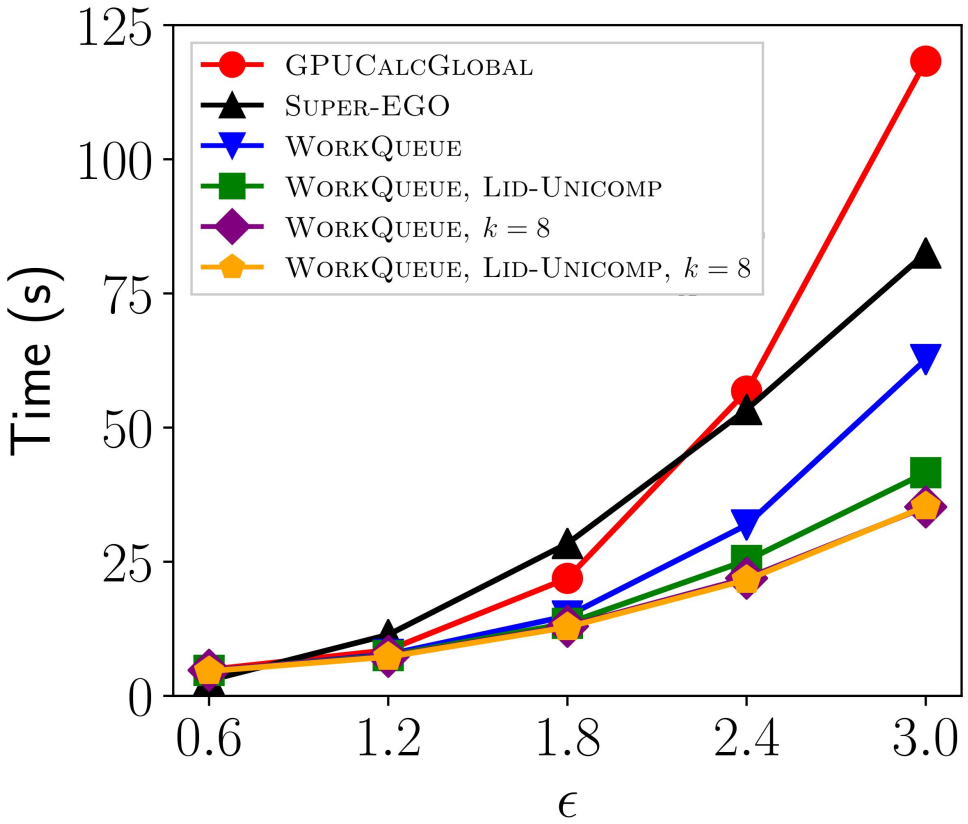


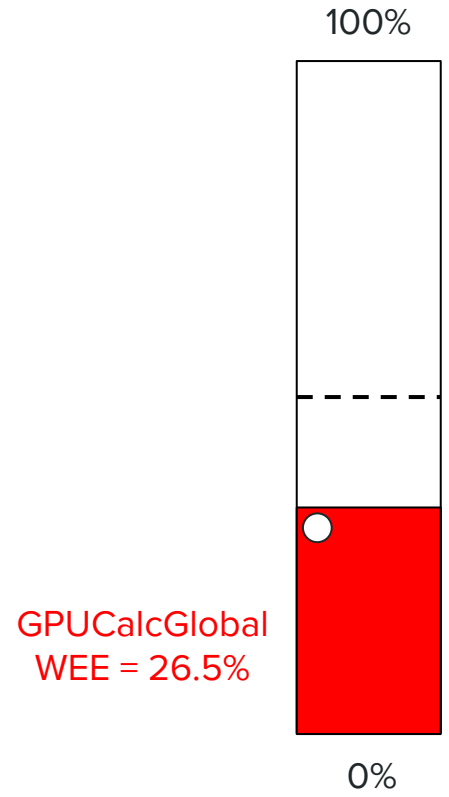


Dataset: Expo2D2M, Warp execution efficiency: ε = 0.2

Experimental Evaluation: Combination

- Comparison between GPUCalcGlobal, Super-EGO, WorkQueue, WorkQueue + Lid-Unicomp, WorkQueue + k = 8 and WorkQueue + Lid-Unicomp + k = 8

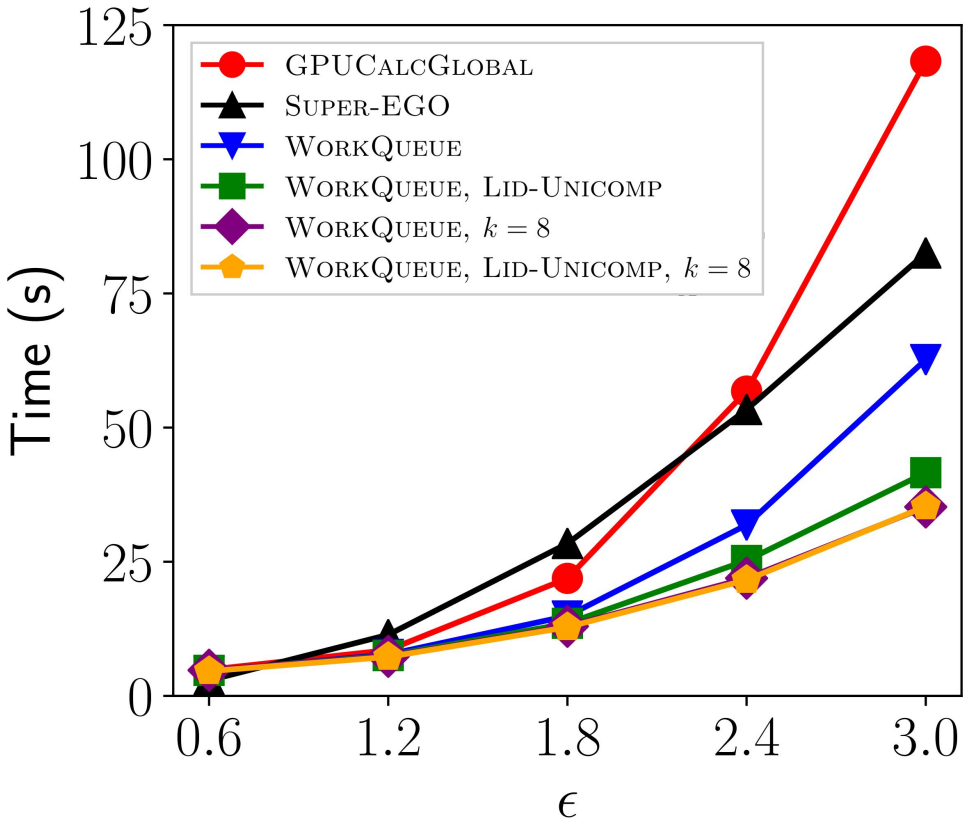


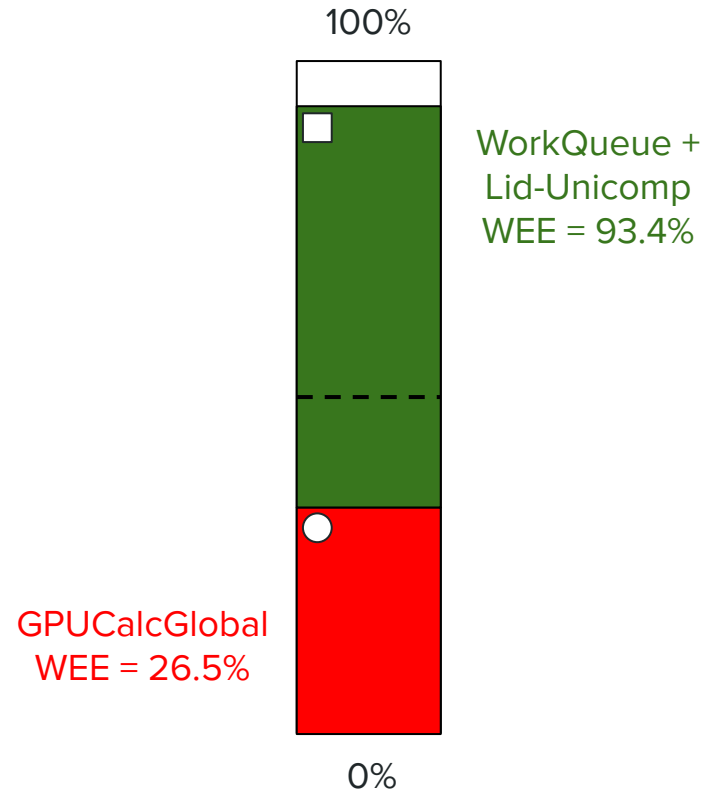


Dataset: SW3DA, Warp execution efficiency: ε = 2.4

Experimental Evaluation: Combination

- Comparison between GPUCalcGlobal, Super-EGO, WorkQueue, WorkQueue + Lid-Unicomp, WorkQueue + k = 8 and WorkQueue + Lid-Unicomp + k = 8

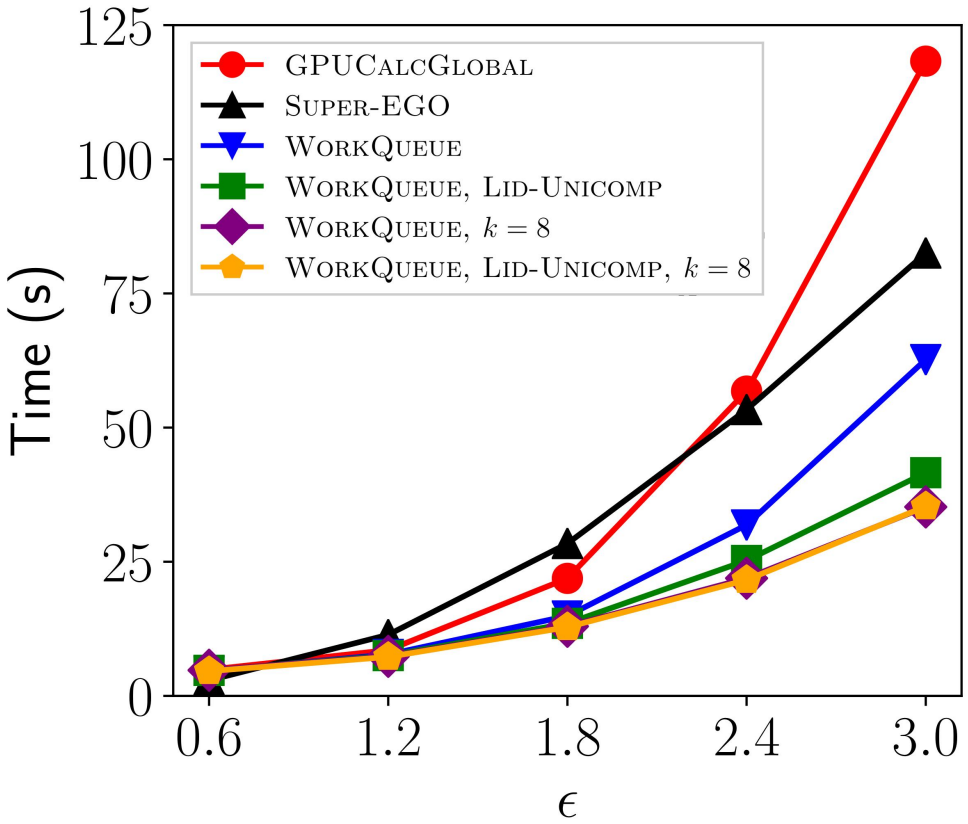


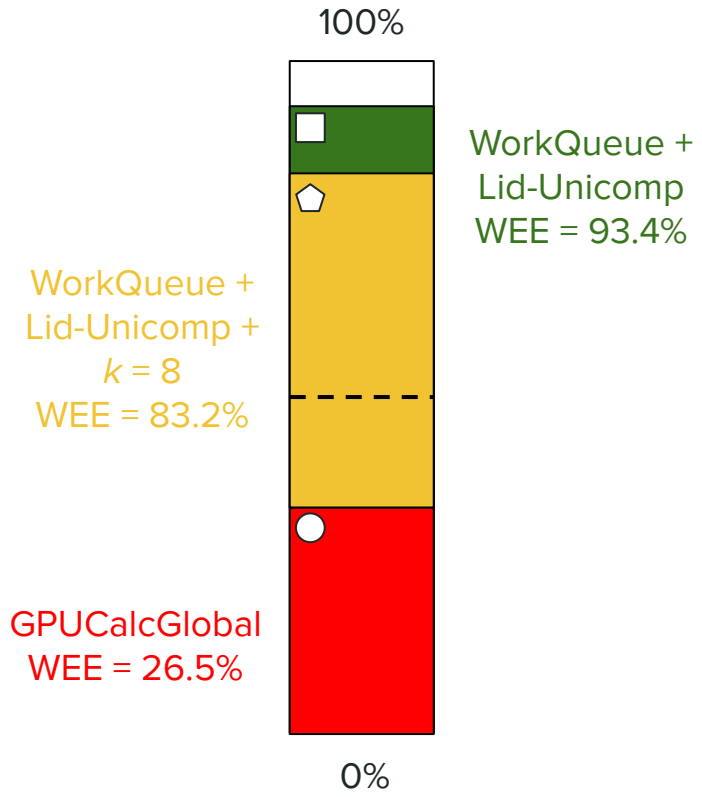


Dataset: SW3DA, Warp execution efficiency: ε = 2.4

Experimental Evaluation: Combination

- Comparison between GPUCalcGlobal, Super-EGO, WorkQueue, WorkQueue + Lid-Unicomp, WorkQueue + k = 8 and WorkQueue + Lid-Unicomp + k = 8

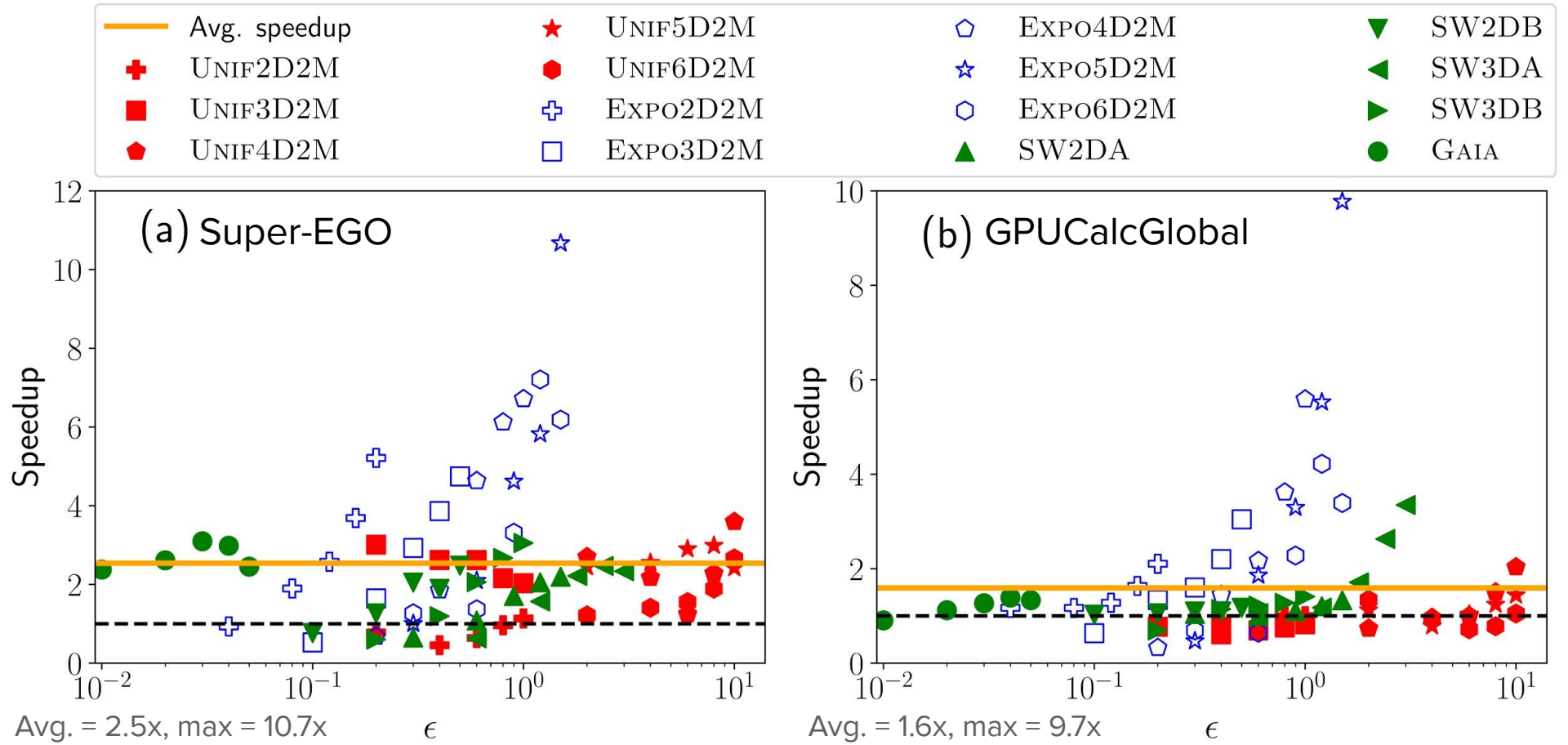




Dataset: SW3DA, Warp execution efficiency: ε = 2.4

Experimental Evaluation: Speedup

- Speedup of all our optimizations combined versus (a) Super-EGO and (b) GPUCalcGlobal



Conclusion and Future Work

Conclusion

- Intra-warp and inter-warp load balancing improves performance
  - Similar workloads in a warp
    - Fewer idling threads
  - Similar workloads between warps
    - Less waiting for the last executing warp

- May be used for other algorithms with data-dependent performance characteristics

- High warp execution efficiency improves GPU's utilization
  - May indicate one of the potential boundaries for further performance optimizations (cannot go beyond 32 active threads over 32)

Future Work

- Improve Lid-Unicomp execution
  - Currently iterates over every neighboring cells, then checks the linear id
  - ➔ Remove unnecessary loop iterations

- Improve the work queue
  - Memory allocation suited to the firsts large batches
  - Many small batches towards the end of the computation
  - ➔ Group the last batches together

- Complete this work with a parallel CPU implementation
  - ➔ Splits the work between the CPU and the GPU