Exploring the Design-Space of GPU-Efficient Similarity Self-Join Kernels

Benoît Gallet[1,2]
benoit.gallet@etu.univ-orleans.fr

Michael Gowanlock[2]
Michael.Gowanlock@nau.edu

[1]Université d'Orléans, Département d'Informatique
[2]Northern Arizona University, School of Informatics, Computing and Cyber Systems

1. Introduction

- Given a dataset, self-join finds all objects within a range of each other defined by a similarity metric.
- Focus on the distance similarity self-joins, finding all points within a distance ϵ of each other.
- Use a grid-based index designed for the GPU to prune the search for nearby points.

2. Background

- The use of the GPU is justified by its high parallelism and memory bandwidth in comparison to a CPU.
- In previous work [1], Unicomp avoids redundant computations: given a point p and its neighbor q, if q is within a distance ϵ of p, then p is within a distance ϵ of q, and it is possible to add both (p, q) and (q, p) to the result set.
- Figure 2 shows the comparison between grid cells that occur when a given query point falls within a respective origin cell.
- Each thread processes a point. Some points are located in cells with many or few comparisons to adjacent cells.
- Assuming that the data is uniformly distributed, then the computational work is not balanced between all of the threads, as some threads will execute longer than others.

3. Solution

- A new computational pattern for Unicomp is advanced for 2D and 3D datasets called B-Unicomp (Figure 1 shows for 2D). The computational load is theoretically evenly balanced for non-border cells of the dataset, as well as the GPU resources.
- Another optimization is to increase the granularity of the parallel computation of the distance calculations by using multiple threads to compute the distance between a point and its neighbors as shown in Figure 2.

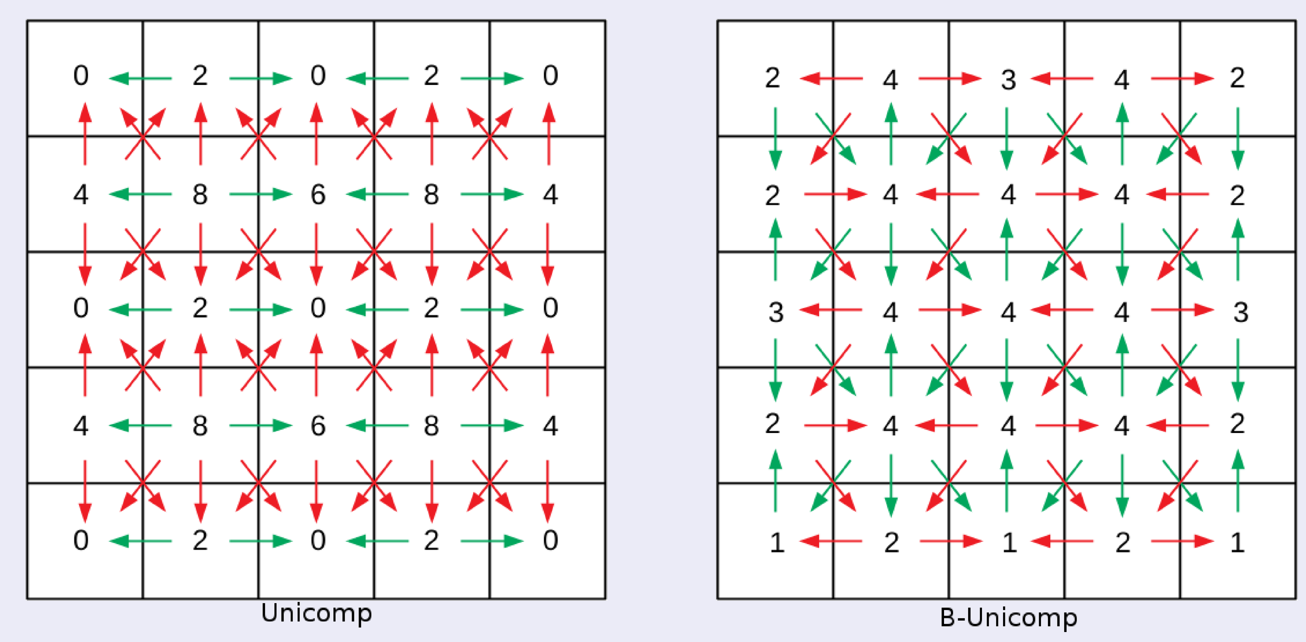

Figure 1: Unicomp and B-Unicomp computational patterns.

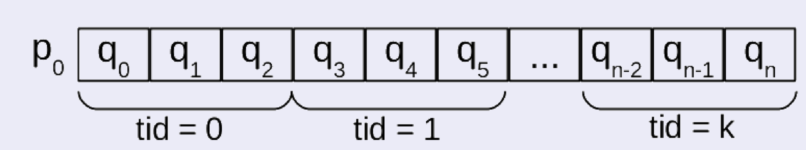

Figure 2: p_i the computed point, q_j its neighbor points, *tid* the threads, $k = (n+1)/T$, T the number of threads per point.

4. Results

Implementations:

- SuperEGO: parallel CPU algorithm [3].
- GPU: global memory kernel advanced in [2].
- Unicomp: solution proposed in [1].
- B-Unicomp: balanced pattern for Unicomp.
- Unicomp/2: Unicomp using 2 threads per point.
- B-Unicomp/2: B-Unicomp using 2 threads per point.

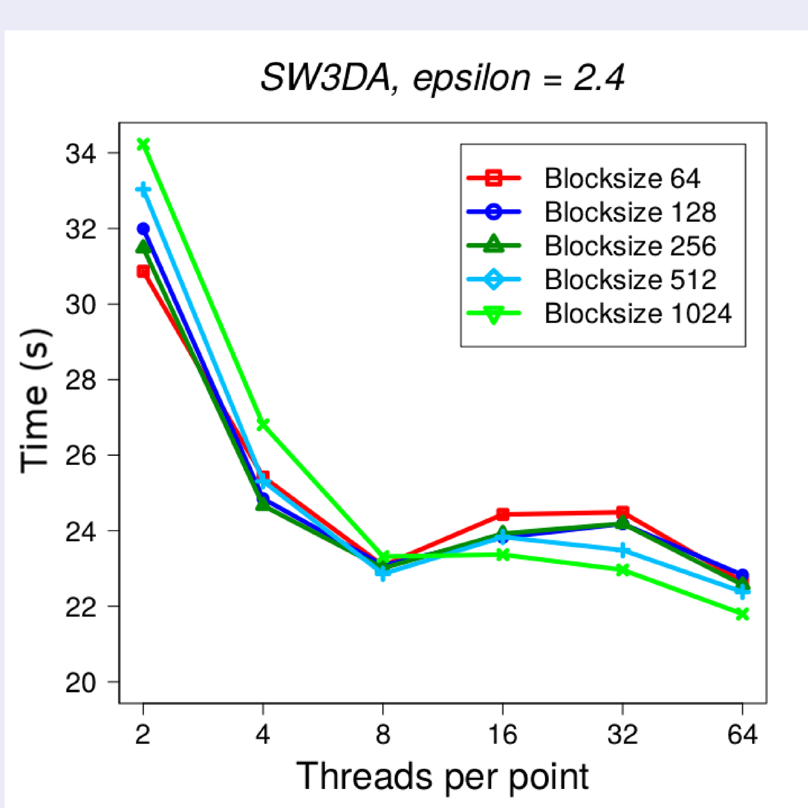

Figure 3: Response time for different block sizes and threads per point.

- Figure 4 plots the response time for real-world datasets: SW and SDSS in 2 and 3 dimensions. These datasets are the same that were used in performance results of [1]. We only consider up to 2 threads per point for Unicomp and B-Unicomp.
- While Unicomp and B-Unicomp achieve similar performance on 2D datasets, B-Unicomp outperforms Unicomp on 3D datasets.
- The load balancing of B-Unicomp is demonstrated when there is a sufficient workload for the GPU to execute.
- Using 2 threads to compute a single point is only advantageous on large workloads (SW datasets), and even outperforms the SuperEGO implementation on the SW3DA dataset, where SuperEGO outperforms Unicomp.
- Figure 3 shows the impact of the block size and the number of threads used per point on the SW3DA dataset for an $\epsilon = 2.4$. While the block size does not significantly impact the performance, using 8 or 64 threads per point offers the best performance improvement in this scenario.
- We leave the exploration of more than 2 threads per point on the other datasets for future work.

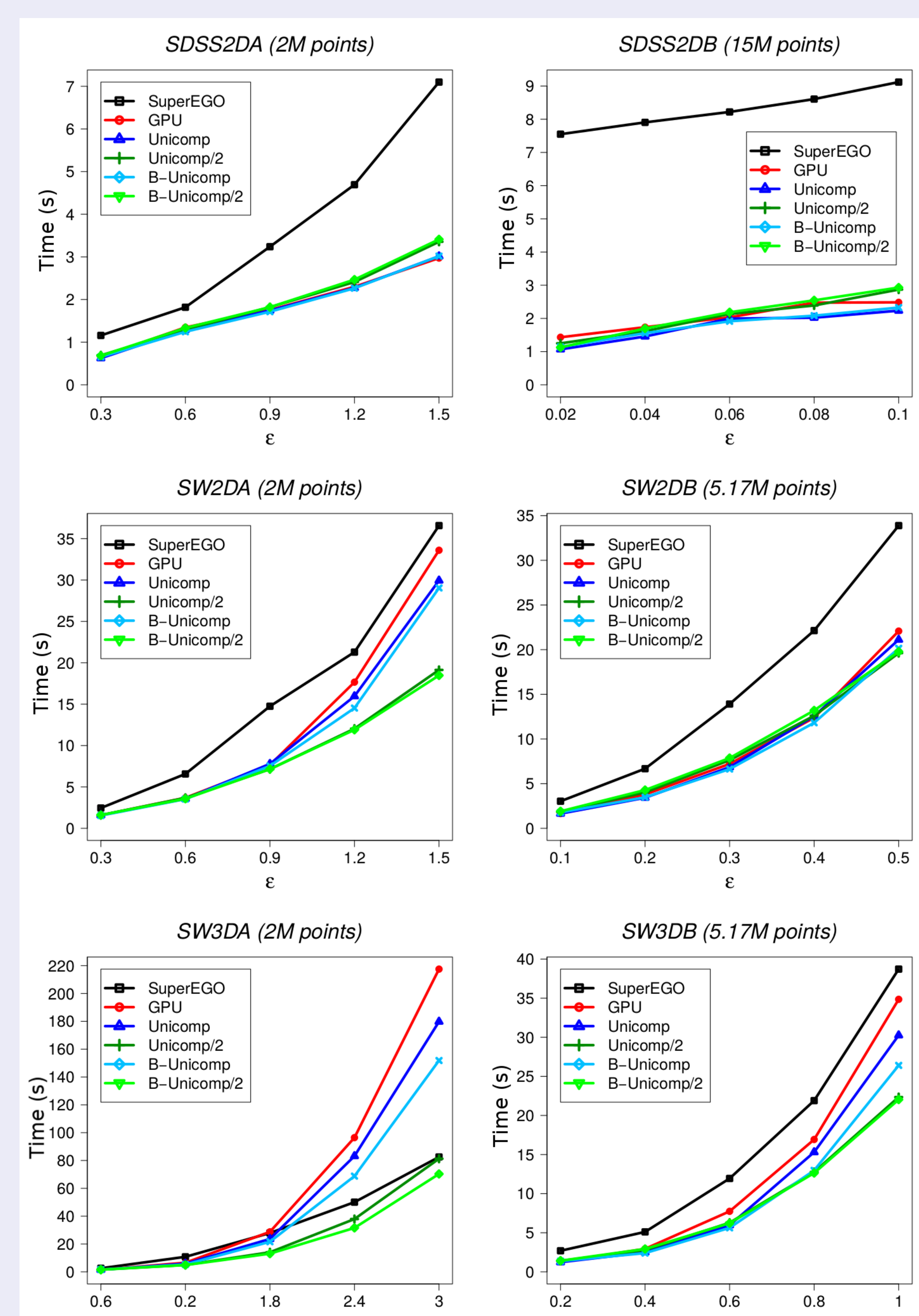

Figure 4: Response time of SW and SDSS on 2 and 3 dimensions.

5. Conclusion

- In each scenario, B-Unicomp outperforms or achieves the same performance as Unicomp.
- Using multiple threads per point is useful when the computational workload is high.
- Future work includes generalizing B-Unicomp to higher dimensions, avoiding redundant operations when using multiple threads per point and a performance model that can be used to select the best configuration given an ϵ value and a dataset.

References

[1] Michael Gowanlock, Ben Karsin GPU Accelerated Self-join for the Distance Similarity Metric *In High-Performance Big Data, Deep Learning, and Cloud Computing, Workshop of the 32nd IEEE International Parallel & Distributed Processing Symposium*, Vancouver, 2018.

[2] Michael Gowanlock, Cody M. Rude, David M. Blair, Justin D. Li, Victor Pankratius Clustering Throughput Optimization on the GPU *In Proc. of the 31st IEEE International Parallel & Distributed Processing Symposium*, Orlando, 2017.

[3] Dmitri V. Kalashnikov Super-EGO: fast multi-dimensional similarity join *In The VLDB Journal*, vol. 22, no. 4, pp. 561-585, 2013.