Techniques for Data-Parallel Searching for Duplicate Elements

Brenton Lessley* University of Oregon
Kenneth Moreland† Sandia Nat'l Lab
Matthew Larsen‡ Lawrence Livermore Nat'l Lab
Hank Childs§ University of Oregon

Abstract

We study effective shared-memory, data-parallel techniques for searching for duplicate elements. We consider several data-parallel approaches, and how hash function, machine architecture, and data set can affect performance. We conclude that most choices of algorithm and hash function are problematic for general usage. However, we demonstrate that the choice of the Hash-Fight algorithm with the FNV1a hash function has consistently good performance over all configurations.

1 Introduction

Searching for duplicate elements comes up in multiple visualization contexts, most notably external facelist calculation. There are two main approaches for identifying duplicates: (1) sorting all elements and looking for identical neighbors, and (2) hashing all elements and looking for collisions. These approaches were compared previously by Lessley et al. [5] in their study of external facelist calculation with data-parallel primitives (DPP). However, subsequent analysis has shown that the performance of the algorithm can vary unexpectedly with certain combinations of hash function, architecture, and data set. With this short paper, we run a study considering many test configurations, in an effort to better understand anomalous behavior. Using various metrics, we are able to understand the causes of unusual performance. We believe the contributions of this paper are two-fold. First, we contribute a better understanding of platform portable algorithms for identifying duplicates and their pitfalls, and specify recommendations for choices that will perform well over a variety of configurations. Second, we believe the result is useful for the community in identifying potential performance issues with DPP algorithms. Overall, we find that the Hash-Fight algorithm with the FNV1a hash function consistently achieves the best performance in identifying duplicates over all tested configurations.

2 Related Work

This work is a follow-on to the previous external facelist calculation study by Lessley et al. [5]. That work introduced two DPP-based algorithms for calculating the external facelist of three-dimensional unstructured grids. That said, the current work does make algorithmic contributions, in that it considers more variants of the algorithms via additional hash functions. Our study is again conducted within the VTK-m framework [9], which provides data parallel primitives as basic building blocks. Further, this study follows several previous studies in exploring the limits of portable performance for visualization algorithms in a DPP setting [8, 4, 3, 12, 7, 6], with the main difference being that we are exploring an algorithm of a very different nature — our algorithm is effectively a large search problem, where many of the others focused on iterating over cells in a mesh.

*e-mail: blessley@cs.uoregon.edu
†e-mail: kmorel@sandia.gov
‡e-mail: larsen30@llnl.gov
§e-mail: hank@cs.uoregon.edu

3 Experiment Overview

To better understand the behavior of external facelist calculation with respect to algorithm design choices, particularly for that of hash functions, we conducted experiments that varied three factors:

- Algorithm (7 options)
- Hardware architecture (3 options)
- Data set (34 options)

We did run the cross-product of tests ($714 = 7 \times 3 \times 34$), but our results section presents the relevant subset that capture the underlying behavior.

3.1 Algorithm

We studied three types of algorithms, which we refer to as **SortyById**, **Sort**, and **Hash-Fight**. Sort and Hash-Fight each need to be coupled with a hashing function. We considered three different hashing functions: **XOR**, **FNV1a**, and **Morton**. In total, we considered seven algorithms: Sort+FNV1a, Sort+XOR, Sort+Morton, Hash-Fight+FNV1a, Hash-Fight+XOR, Hash-Fight+Morton, and SortByID.

3.1.1 Algorithms

SortById: The idea behind this approach is to use sorting to identify duplicate faces. First, faces are placed in an array and sorted. Each face is identified by its indices. The sorting operation requires a way of comparing two faces (i.e., a "less-than" test); we order the vertices within a face, and then compare the vertices with the lowest index, proceeding to the next indices in cases of ties. The array can then be searched for duplicates in consecutive entries. Faces that repeat in consecutive entries are internal, and the rest are external.

SortById is likely not optimal, in that it requires storage for each index in the face (e.g., three locations for each point in a triangular face of a tetrahedron), resulting in a penalty for sorting extra memory. This motivates the next approach, which is to use hashing functions to reduce the amount of memory for each face in the sort.

Sort: We denote the algorithm that modifies SortById to sort hash values rather than indices. For each face, the three vertex indices are hashed, and the resulting integer value is used to represent the face. However, this creates additional work. The presence of collisions forces us to add a step to the algorithm that verifies whether matching hash values actually belong to the same face. In this study, we explore the tradeoff between sorting multiple values per face versus resolving collisions. Further, the specific choice of hash function may affect performance, and we explore this issue as well.

Hash-Fight: Traditionally, hash collisions are handled via a chaining or open-addressing approach. While these approaches are straight-forward to implement in a serial setting, they do not directly translate to a parallel setting. For example, if multiple threads on a GPU map to the same hash entry at the same time, then the behavior may be non-deterministic, unless atomics are employed.

In Lessley et al. [5], hash collisions are addressed in a parallel setting via a data-parallel hashing scheme that uses mutliple iterations. In this scheme, which we denote as Hash-Fight, no care is taken to detect collisions or prevent race conditions via atomics. Instead, every face is simultaneously written to the hash table, possibly overwriting previously-hashed faces. The final hash table will then contain the winners of this "last one in" approach. However, our next step is to check, for each face, whether it was actually placed in the hash table. If so, the face is included for calculations during that iteration. If not, then the face is saved for future iterations. All faces are eventually processed, with the number of iterations equal to the maximum number of faces hashed to a single index.

3.1.2 Hash Function

The following hash functions are considered in our study, each returning a hash value in the form of a 32-bit unsigned integer.

XOR: The XOR hashing function is a very simple bitwise exclusive-or operation on all the indices of a face. The XOR hash is very fast to compute but makes little effort to avoid collisions.

FNV1a: FNV1a hashing [2] starts with an offset value (2166136261 for this study) and then iteratively multiplies the current hash by a prime number (16777619 for this study) and performs an exclusive-or with the vertex index. The addition of the offset and prime number pseudo-randomize the hash values, which helps reduce collisions. However, FNV1a takes longer to compute because of its additional calculations.

Morton: The Morton z-order function maps a multi-dimensional point coordinate to a single Morton code value. It does this by interleaving and combining the binary representations of the coordinate values, while preserving the spatial locality of the points [10]. In this study, we compute a separate Morton code for each of the three vertex indices of a triangular cell face, and then add the three Morton codes together to form a hash value.

3.2 Hardware Architecture

We ran our tests on the following three architectures:

- Haswell: Dual Intel Xeon Haswell E5-2698 v3, each with 16 cores running at 2.30 GHz and 2 SMT hardware threads and a total of 512 GB of DDR4 memory.
- Knights Landing: An Intel Knights Landing Self Hosting Xeon Phi CPU with 72 cores, running in quadrant cluster mode and flat memory node. Each core has 4 threads and runs at a base clock frequency of 1.5 GHz. This processor also maintains 16 KB of on-package MCDRAM memory and 96 GB of DDR4 memory.
- Tesla: An NVIDIA Tesla P100 Accelerator with 3,584 processor cores, 16 GB memory, and 732 GB/sec memory bandwidth. Each core has a base frequency of 1,328 MHz, and a boost clock frequency of 1480 MHz.

3.3 Data set

For all of our experiments, we used variants of an unstructured tetrahedral mesh derived by tetrahedralizing a uniform grid. We considered 17 different data set sizes ranging from 4 million to 667 million cells, which translates to a range of 16 million to 2.6 billion faces. Additionally, we considered two types of mesh connectivity indexing schemes:

- Regular indexing: mesh connectivity was left unaltered. Mesh coordinates that are spatially close were generally located nearby in memory.
- Randomized indexing: mesh connectivity was randomized, i.e., mesh coordinates were scattered randomly in memory.

We considered randomized indexing to study the behavior of the different algorithms and hashing functions with different types of memory access patterns. We viewed these two patterns as ends of a spectrum (coherent access versus incoherent access), and we believe real world data sets will fall within these two extremes. Between the 17 data set sizes and two indexing schemes, there were 34 total data sets used in the study.

To obtain our randomized indexing, we randomized the location of the vertices and cells in the regular data sets and adjusted the cell indices to match the new locations of the vertices. The randomized topology has the following two effects. First, indices are no longer near one another. For example, a tetrahedral cell in the regular topology might have indices 14, 21, 16 and 25, while, with the randomized topology, the same cell might have indices 512, 1, 73, and 1024. Second, accessing the vertices of a cell will exhibit poor memory access patterns since each point vertex is likely on a different cache line.

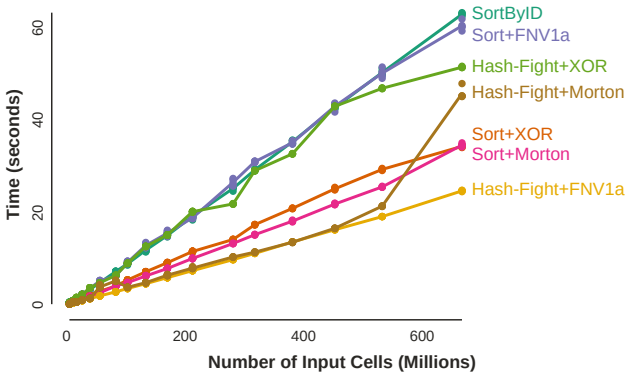

Figure 1: Runtime comparison of all algorithm types and all hash functions on Intel's Haswell architecture with regularly-indexed data sets. 10 trials were conducted per algorithm for each data set, and each trial is represented by a dot. The trendlines plot the average times of these trials.

4 Results

Our study contains three phases, each of which assesses the impact of different factors on performance: hash function, architecture, and data set regularity. In this section, we present and analyze the results of these phases.

4.1 Phase 1: Hash Functions

This phase examines the choice of hash function, and considers the performance over our 7 combinations of algorithm type and hash function. We also vary the data set size, specifically looking at all 17 regularly-indexed data sets. The only architecture considered in this phase is the Intel

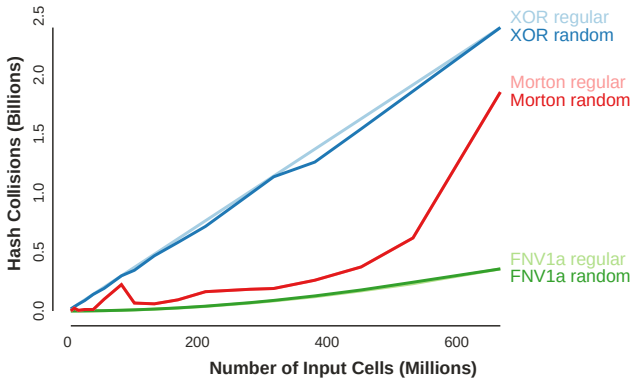

Figure 2: A comparison of the number of collisions produced by the 3 hash functions considered in this study, over both regular and random indexed data sets. The Morton and FNV1a functions have little variation in collisions between data set type and thus have overlapping plots.

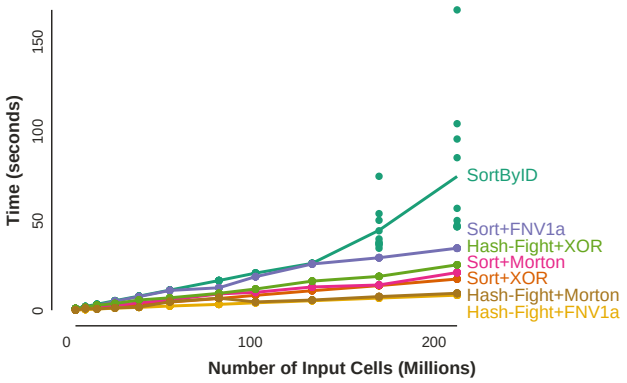

Figure 3: Runtime comparison of all algorithms and all hashing functions on Intel's Knights Landing architecture with regularly indexed data sets. 10 trials were conducted per algorithm for each data set, and each trial is represented by a dot. The trendlines plot the average times of these trials.

Operation	SortBy Id	Sort XOR	Sort FNV1a	Sort Morton
prepInputFaces	0.88	0.86	0.86	0.87
hashFaces	2.05	1.03	1.04	1.13
hashSortReduce	25.37	9.41	23.59	10.82
findInternalFaces	0.17	5.72	4.89	1.89
prepOutputFaces	0.56	0.29	0.46	0.41
Total Time	29.03	17.31	30.84	15.12

Table 1: Runtimes (sec) for each of the primary data-parallel operations of the Sort algorithms. Each time is the average of the 10 trials conducted on the 400^3 grid data set with approximately 318 million input cells.

Haswell CPU, although we vary the architecture in subsequent phases.

We first analyze the performance of the Hash-Fight algorithm. From Figure 1, we observe that Hash-Fight+FNV1a is consistently the fastest algorithm across the entire range of data set sizes. Hash-Fight+Morton is a close second, up until large data set sizes. When the data set size jumps from 530 million cells to 670 million cells, its execution time more than doubles. This jump in execution time is partially attributed to an increase in the number of hash-fight iterations, from 18 to 24. This increase in iterations is the result of the number of hash collisions tripling from 619 million to 1.9 billion, as seen in Figure 2 (Morton regular plot). We believe that the large increase in hash collisions occurred because the spatial locality of the Morton z-order curve deteriorates as the cell count significantly increases, due to the additive property of the Morton hash function. This increase in collisions was also seen with Sort+Morton.

Additionally, Hash-Fight coupled with the XOR hash function tends to perform poorly compared to the couplings with FNV1a and Morton. From Figure 2 (XOR regular plot), Hash-Fight+XOR consistently yields the largest number of hash collisions, by a wide margin, among the Hash-Fight algorithms. Overall, the trends for collisions in Figure 2 closely match the trends for Hash-Fight execution times in Figure 1.

We now consider the performance of the Sort algorithm. Although Hash-Fight generally performs better with FNV1a and worse with XOR, the opposite outcome tends to occur with Sort.

From Figure 1, we observe that Sort+XOR begins to significantly outperform Sort+FNV1a as the number of input cells increases. Although the XOR hash function produces many collisions, which increases the time to find internal faces, it also results in a markedly faster sort time (see the hashSortReduce row in Table 1). We discovered that the XOR hash function happens to create hash values that are already close to being in sorted order for these data set structures. The parallel sort algorithm, which comes from the TBB library [11], is a parallel version of quicksort, and this algorithm runs much faster on data that is pre-sorted or close to being sorted. The benefits from the faster sorting outweigh the extra time needed to resolve collisions.

Finally, the performance of SortById confirms our hypothesis from Section 3.1.1 that the cost of sorting multiple values per face (3 points per triangular face) is larger than the cost of resolving collisions from hashing the face. As Table 1 indicates, SortById consumes most of its total runtime ordering the 3 face points (hashFaces operation) and then sorting and reducing the faces (hashSortReduce operation).

4.2 Phase 2: Architectures

In this phase, we conduct the same set of experiments from Phase 1 on two additional architectures—Intel Knights Landing (KNL) CPU and Nvidia Tesla P100 GPU—and compare the results to that of the Intel Haswell CPU experiments from Phase 1. Figures 3 and 4 present the results of the KNL and Tesla experiments, respectively. Note that, due to machine memory restrictions, only the 11 datasets up to 213 million cells (350^3 grid) are considered for KNL, and only the 5 datasets up to 40 million cells (200^3 grid) are considered for Tesla.

Figures 3 and 4 reveal that the Sort and Hash-Fight algorithm types almost always run the fastest when coupled with the FNV1a or Morton hash functions on both the KNL and Tesla. Additionally, Hash-Fight+FNV1a and Hash-Fight+Morton are consistently the highest-performing algorithms for both architectures. From Phase 1, we discovered that the performance of an algorithm deteriorated for larger numbers of input cells with the Morton function and Haswell architecture. The analysis revealed that this lower performance was due to an increase in hash collisions. On the KNL and Tesla, this increase in collisions still occurs, but does not negatively affect the performance as it did on Haswell, at least for the size of data sets that fit within these architectures'

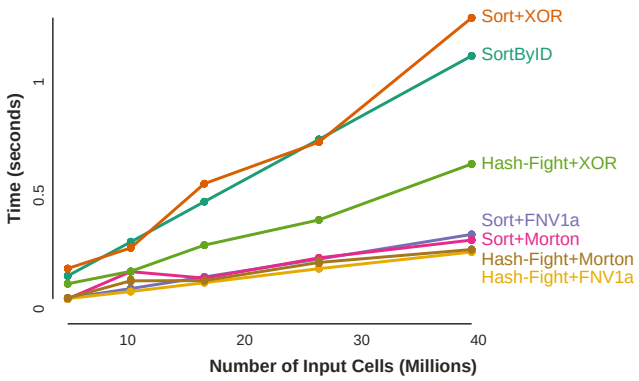

Figure 4: Runtime comparison of all algorithms and all hashing functions on NVIDIA's Tesla P100 GPU with regularly indexed data sets. 10 trials were conducted per algorithm for each data set, and each trial is represented by a dot. The trendlines plot the average times of these trials.

memory. This finding may also be due to architecturally-specific traits in the memory hierarchies (e.g., NUMA, cache infrastructure, etc.); we will investigate this further in future work.

Similar to the findings of Phase 1 on Haswell, the Sort algorithm type performs very well on KNL when coupled with the XOR function. However, Sort+XOR is the worst-performing algorithm, along with SortById, on the Tesla architecture. This reversal in performance of Sort+XOR on the Tesla can be attributed to the sub-routine that finds all internal faces. From Table 2, we observe that XOR, FNV1a, and Morton all take approximately the same amount of time (0.15 seconds) to sort and reduce the hash values, but differ greatly in the time needed to find internal faces, with XOR requiring almost a whole second longer to complete. This is because XOR yielded a substantially larger number of hash collisions, which generated more neighbor searches to resolve collisions and find the internal faces. This matches the finding from Phase 1. However, unlike in Phase 1, the XOR algorithm did not yield a significantly faster sort time to compensate for the increase in time caused by the added collisions. This is because the GPU's parallel sort algorithm, which comes from the Thrust library [1], is based on radix sorting, which is both faster in general and much less sensitive to initial ordering than quicksort.

Contrarily, Sort+FNV1a and Sort+Morton perform very well on the Tesla because the majority of the work involves sorting operations, which are very suitable for massive thread and data-parallelism. Using the CUDA Thrust radix sort, the runtime needed to sort the larger arrays of unique hash values for Sort+FNV1a and Sort+Morton was significantly faster than that of Intel TBB quicksort, which is used in our Haswell and KNL experiments. Replacing the Thrust radix sort with a known-to-be slower Thrust merge sort revealed the same sorting pattern from Phase 1, in which Sort+FNV1A takes longer to perform the hash value sorting than Sort+XOR and Sort+Morton. This indicates that the choice of the sort algorithm matters a lot, with the radix sort being faster in general.

4.3 Phase 3: Irregular Data Sets

In this phase of the study, we evaluate the performance of the algorithms when using data sets with randomized, irregular topologies. Figures 5, 6, and 7 compare algorithm runtimes

Operation	Sort XOR	Sort FNV1a	Sort Morton
prepInputFaces	0.02	0.02	0.02
hashFaces	0.01	0.01	0.01
hashSortReduce	0.13	0.15	0.15
findInternalFaces	1.09	0.13	0.11
prepOutputFaces	0.03	0.02	0.01
Total Time	1.28	0.33	0.30

Table 2: Runtimes (sec) for each of the primary data-parallel operations of the Sort algoritms on an Nvidia Tesla P100 GPU. Each time is the average of the 10 trials conducted on the 200^3 grid data set with approximately 40 million input cells.

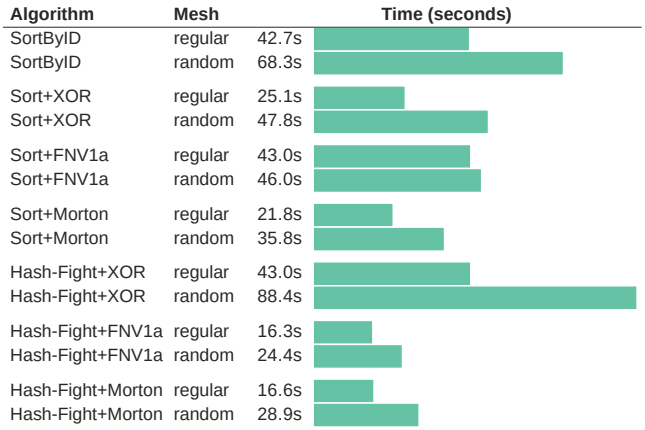

Figure 5: A comparison of the performance between regular indexing and randomized indices on Haswell with data sets of 453 million tetrahedral cells (450^3 grid).

of regular and randomized topologies on the Haswell, KNL, and Tesla architectures, respectively.

The XOR and FVN1a hash functions are both based on creating an index-based hash and, for CPU architectures, SortById, Sort+XOR, and Hash-Fight+XOR pay significant penalties with the randomized topology. For SortById and Sort+XOR, the initial positions of the keys are much closer to their sorted positions with the regular topologies than with the randomized versions. Thus, the sorting algorithm has to perform more work, leading to increased randomized topology runtimes. On the GPU, the VTK-m thrust back-end uses an optimized radix sort when keys are single 32-bit values and a merge sort for all other value types. Consequently, Sort+XOR with a randomized topology pays far less of a penalty as compared to the CPU version, while SortById actually pays a higher penalty. Conversely, the hashing properties of FNV1a distribute keys evenly with both regular and randomized topologies, leading to better performance on the KNL architecture using the randomized topology.

As seen in Figure 2 (XOR random plot), XOR has the largest number of collisions since it is a poor hash function, and the number of collisions increases linearly with the size of the data set. FNV1a and Morton perform much better as the data set size increases, as the increase in the number of duplicates is less than linear. With Morton, the number of collisions increased significantly for the largest data set. For all three hash functions, there is no significant difference in the number of collisions between the regular and random meshes.

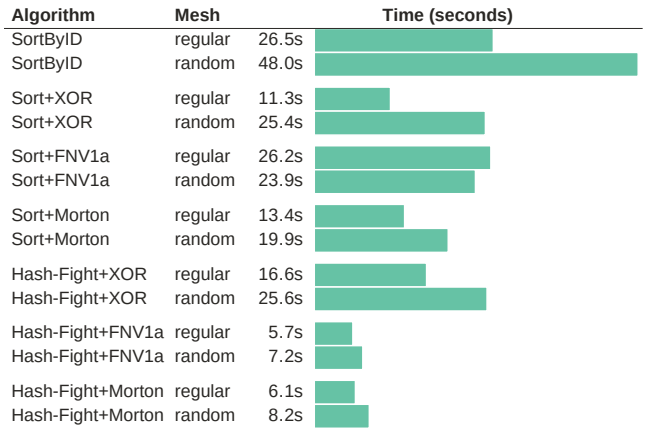

Figure 6: A comparison of the performance between regular indexing and randomized indices on KNL with data sets of 134 million tetrahedral cells (300^3 grid).

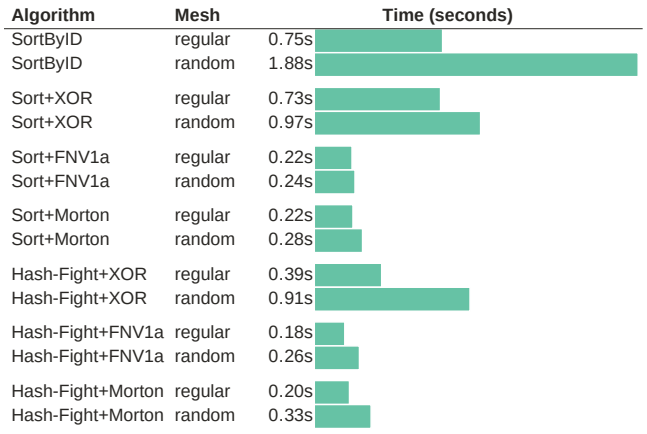

Figure 7: A comparison of the performance between regular indexing and randomized indices on Tesla with data sets of 26 million tetrahedral cells (175^3 grid).

5 Conclusion

We summarize our findings and best practices by phase. From Phase 1, we conclude the following:

- Use the Hash-Fight+FNV1a algorithm for consistently-optimal performance for regularly-indexed data sets.
- Avoid the Morton hash function for large data set sizes, as it does not demonstrate robustness to hash collisions.

From Phase 2, we conclude the following:

- Use the Hash-Fight+FNV1a algorithm for optimal portable performance across varying architectures.
- Avoid the use of Sort+XOR on a GPU architecture; instead, use either Sort+FNV1a or Sort+Morton, in combination with the CUDA Thrust radix sort.

From Phase 3, we conclude the following:

- Both Sort and Hash-Fight perform best with the FNV1a hash function, which is robust to both regular and randomized mesh topologies across multiple architectures.
- Radix sort performs best overall, while quicksort performs poorly with heavily shuffled input. Further, the best-performing hash functions produce heavily shuffled input.

Overall, we believe these findings inform best practices for searching for duplicate elements within data-parallel primitives. The Hash-Fight+FNV1a algorithm consistently performed as a top choice in all configurations; all other algorithms suffered slowdowns in at least some configurations. The fact that the Hash-Fight configurations often beat the more traditional sorting-based algorithm is particularly interesting considering that it intentionally invokes write after write hazards. Although writing to the same or nearby memory locations from multiple threads can degrade cache performance, Hash-Fight still performs efficiently. In terms of future work, we are interested in expanding our analyses of the Haswell and KNL CPU-based performance, particularly in regard to the behavior of hash collisions and caching.

References

[1] N. Bell and J. Hoberock. *GPU Computing Gems, Jade Edition*, chap. Thrust: A Productivity-Oriented Library for CUDA, pp. 359–371. Morgan Kaufmann, October 2011.

[2] G. Fowler, L. C. Noll, K.-P. Vo, D. Eastlake, and T. Hansen. The fnv non-cryptographic hash algorithm. Technical report, Network Working Group, 2017. `https://tools.ietf.org/html/draft-eastlake-fnv-13`.

[3] M. Larsen, S. Labasan, P. Navrátil, J. Meredith, and H. Childs. Volume Rendering Via Data-Parallel Primitives. In *Proceedings of EuroGraphics Symposium on Parallel Graphics and Visualization (EGPGV)*, pp. 53–62. Cagliari, Italy, May 2015.

[4] M. Larsen, J. Meredith, P. Navrátil, and H. Childs. Ray-Tracing Within a Data Parallel Framework. In *Proceedings of the IEEE Pacific Visualization Symposium*, pp. 279–286. Hangzhou, China, Apr. 2015.

[5] B. Lessley, R. Binyahib, R. Maynard, and H. Childs. External Facelist Calculation with Data-Parallel Primitives. In *Proceedings of EuroGraphics Symposium on Parallel Graphics and Visualization (EGPGV)*, pp. 10–20. Groningen, The Netherlands, June 2016.

[6] S. Li, N. Marsaglia, V. Chen, C. Sewell, J. Clyne, and H. Childs. Achieving Portable Performance For Wavelet Compression Using Data Parallel Primitives. In *Eurographics Symposium on Parallel Graphics and Visualization*. The Eurographics Association, 2017.

[7] L.-t. Lo, C. Sewell, and J. P. Ahrens. Piston: A portable cross-platform framework for data-parallel visualization operators. In *EGPGV*, pp. 11–20, 2012.

[8] R. Maynard, K. Moreland, U. Atyachit, B. Geveci, and K.-L. Ma. Optimizing threshold for extreme scale analysis. In *IS&T/SPIE Electronic Imaging*, pp. 86540Y–86540Y. International Society for Optics and Photonics, 2013.

[9] K. Moreland, C. Sewell, W. Usher, L. Lo, J. Meredith, D. Pugmire, J. Kress, H. Schroots, K.-L. Ma, H. Childs, M. Larsen, C.-M. Chen, R. Maynard, and B. Geveci. VTK-m: Accelerating the Visualization Toolkit for Massively Threaded Architectures. *IEEE Computer Graphics and Applications (CG&A)*, 36(3):48–58, May/June 2016.

[10] G. Morton. A computer oriented geodetic data base and a new technique in file sequencing. Technical Report Ottawa, Ontario, Canada, 1966.

[11] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly, July 2007.

[12] H. A. Schroots and K.-L. Ma. Volume Rendering with Data Parallel Visualization Frameworks for Emerging High Performance Computing Architectures. In *SIGGRAPH Asia 2015 Visualization in High Performance Computing*, SA '15, pp. 3:1–3:4. ACM, 2015.