Katholieke
Universiteit
Leuven

**Department of
Computer Science**

ADVANCED COMPUTER ARCHITECTURE

project

Thibaut Beck
Wannes Paesschesoone

Academic year 2025–2026

Contents

1 Introduction

This report talks about the implementation of a point cloud to voxel grid filter. The reason for implementing this filter is to reduce the number of points in a point cloud, while preserving the overall structure and appearance of the original data. Doing this on a CPU takes a long time for large point clouds, so the filter is implemented on a GPU using CUDA to take advantage of the parallel processing capabilities of modern graphics hardware.

2 Implementation

In this chapter, the practical implementation details of the point cloud to voxel grid filter are presented. To leverage the massive parallelism of modern hardware, the algorithms were developed using CUDA for GPU acceleration. Two distinct parallel strategies were designed to solve the voxelization problem: a sorting-based approach using Morton encoding, and a scattering approach using a GPU-resident dynamic hash map.

2.1 Morton and Sort Voxelizer

The first approach utilizes the Thrust library to perform a sort-and-reduce operation. This method relies on the principle of organizing data such that points belonging to the same voxel are stored contiguously in memory.

The algorithm proceeds in the following stages:

1. Quantization: First, the global bounding box of the point cloud is calculated. For every point, discrete grid coordinates are computed based on the user-defined voxel size.
2. Encoding: The 3D grid coordinates are mapped to a 1D scalar value using Z-order curve encoding. This is achieved by bit-interleaving the integer coordinates to produce a 64-bit integer, which serves as the unique key for the voxel.
3. Sorting: The point indices are sorted based on their generated Morton codes. This ensures that all points residing in the same spatial voxel are grouped together in the linear array.
4. Reduction: A reduction-by-key operation is performed. This step iterates through the sorted array, identifies segments of identical Morton codes, and sums the coordinate and color data for each segment.
5. Centroid Calculation: Finally, the accumulated sums are divided by the point count per voxel to yield the final downsampled point cloud.

The theoretical complexity of this approach is dominated by the sorting phase. Given N input points, the time complexity is O(N log N). This method offers deterministic memory usage and stable performance regardless of the spatial distribution of the points.

2.2 Dynamic Hash Map Voxelizer

The second approach implements a custom open-addressing hash table directly in GPU global memory. This method is designed for maximum throughput, avoiding the global synchronization overhead required by sorting.

Crucially, this implementation re-purposes the Morton encoding scheme employed in the previous method. Instead of using the code for sorting, the 64-bit Morton integer serves as a compact, unique hash key. This allows a complex three-dimensional coordinate triplet to be treated as a single primitive type, enabling efficient atomic operations within the hardware registers.

The architecture operates through a high-throughput scatter-and-accumulate pipeline:

1. Initialization: A hash table is allocated in VRAM. To mitigate the risk of collisions inherent to hashing, the capacity is set significantly larger than the number of input points (typically a factor of 2.0 to 4.0).
2. Insertion and Accumulation: A custom kernel processes points in parallel. For each point, the grid coordinates are converted into a Morton code. This code is then hashed to find an initial slot in the table. The algorithm uses a linear probing strategy combined with lock-free atomic primitives:
   - A thread attempts to claim a bucket using an atomic Compare-and-Swap (`atomicCAS`).
   - If the bucket is empty, the thread successfully claims the voxel ownership.

- If the bucket holds the same Morton key, the thread accumulates its point data into the existing voxel using `atomicAdd`.
- If the bucket is occupied by a different key (a collision), the thread probes the subsequent memory slot until a valid location is found.

3. Compaction: Because the hashing process leaves gaps in the table, a final collection pass scans the memory to extract only the valid, populated voxels and writes them densely into the output buffer.

Ideally, this approach achieves O(N) complexity, processing each point in constant time. By treating the Morton code as a hash key, the algorithm creates a "race" where threads compete to update voxel data simultaneously. While this is generally faster than sorting, performance is sensitive to the load factor. As the table fills, threads must probe further to find open slots, which can degrade performance. However, for massive point clouds with sufficient memory available, this method typically yields the highest processing speeds.

3 Results and Analysis

3.1 General Observations

The voxelizers are compared among five voxel sizes (0.25, 0.5, 0.75, 1.0, 1.25). For each voxel size, eight block sizes were tested (1, 2, 4, 8, 16, 32, 64, 256, 512, 1024 threads per block). The hash-table voxelizer was evaluated with three different capacity factors (2, 3, and 4 times the number of input points). The performance metrics recorded include total execution time and a breakdown of time spent in key phases of each algorithm. Every configuration was run 100 times to obtain an average execution time, minimizing the impact of transient system load variations.

3.2 Performance Analysis per Method

3.2.1 Morton-Code Voxelizer

The Morton-based approach demonstrates consistent and predictable performance. Key findings include:

- **Optimal block size:** 4–256 threads per block.
- **Total execution time:** Ranging from 25.18 ms (voxel size 1.25, block size 64) to 40.46 ms (voxel size 0.25, block size 1).
- **Primary bottleneck:** The sorting stage dominates runtime, requiring approximately 0.90–1.02 ms, nearly constant across all tests.

Very small block sizes (1–4 threads) significantly slow down Morton-code generation (0.27–1.07 ms), while larger block sizes stabilize this step to around 0.04 ms.

3.2.2 Hash-Table Voxelizer

The hash-based method demonstrates greater variability, largely influenced by the selected capacity factor of the hash table.

Capacity Factor 2 (highest efficiency)

- **Best overall performance:** 25.67 ms (voxel size 1.0, block size 32).
- **Optimal block sizes:** 8–32 threads per block.
- **Advantages:** Lowest memory overhead and fastest device-to-host transfer times.

Capacity Factor 3 (balanced)

- Execution times are **2–4 ms slower** compared to capacity factor 2.
- Reduced collisions during accumulation, at the cost of increased memory usage.

Capacity Factor 4 (highest overhead)

- **5–10 ms slower** than capacity factor 2.
- Initialization and memory operations increase disproportionately.
- Device-to-host + cleanup time rises up to 5.26 ms (compared to 2.71 ms for CF=2).

3.3 Impact of Voxel Size

Smaller voxel sizes generate a significantly larger number of unique voxel entries. This favors the hash-table method with a low capacity factor, as the Morton approach becomes less efficient when spatial data is highly fragmented.

For larger voxels (e.g., voxel size 1.25), the Morton method benefits from predictable spatial coherence and memory access patterns, making it more efficient than the hash-based approach.

3.4 Impact of Block Size

- **Block sizes 1–4:** Strong performance penalties due to insufficient warp utilization.
- **Block sizes 8–256:** Optimal performance range with minimal overhead.
- **Block sizes 512–1024:** No significant improvements; potentially limited by register pressure.

3.5 Timing Breakdown

3.5.1 Morton Method

Dominant phases:

1. **Sorting:** 0.90–1.02 ms (consistent across tests)
2. **Morton code computation:** 0.04–1.07 ms (strongly dependent on block size)
3. **Point accumulation:** 0.24–1.08 ms

3.5.2 Hash Method

Dominant phases:

1. **Device-to-host transfer + cleanup:** 2.71–5.26 ms (increases with capacity factor)
2. **Initialization:** 0.33–3.71 ms (proportional to hash-table size)
3. **Populate phase:** 0.31–2.46 ms (collision-sensitive)

3.6 Conclusions

3.6.1 Optimal Configurations

The analysis indicates that different voxel sizes benefit from different GPU strategies:

- **Small voxel sizes (0.25–0.5):** Hash-based voxelizer with capacity factor 2 and a block size of 16–32 threads.
- **Medium voxel sizes (0.75–1.0):** Hash-based voxelizer with capacity factor 2 and a block size of 32 threads.
- **Large voxel sizes (1.25 and above):** Morton-code voxelizer with a block size between 64 and 256 threads.

3.6.2 Explanation

These optimal configurations follow from the observed behaviour of both voxelization methods:

1. For small voxel sizes, the hash method with capacity factor 2 achieves the highest throughput due to low memory overhead and efficient device-to-host transfers.
2. For larger voxel sizes, the Morton-based approach becomes more efficient thanks to spatial coherence and predictable memory access patterns.
3. Block sizes between 16 and 64 threads provide the most balanced performance in terms of occupancy and register usage.

3.6.3 Performance Limitations

Despite their strengths, both methods exhibit certain limitations:

- The hash-based method is primarily constrained by memory bandwidth, especially during device-to-host transfers.
- The Morton-based method is strongly dominated by the sorting stage, which forms its main computational bottleneck.
- For both approaches, increasing block sizes beyond 256 threads yields minimal benefits due to hardware saturation.

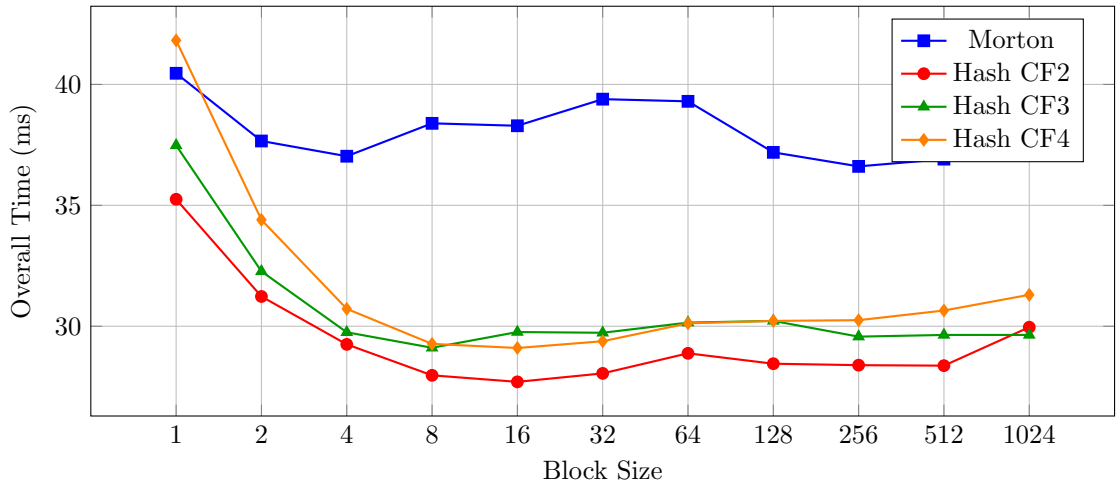

Figure 1: Overall execution time for voxel size 0.25

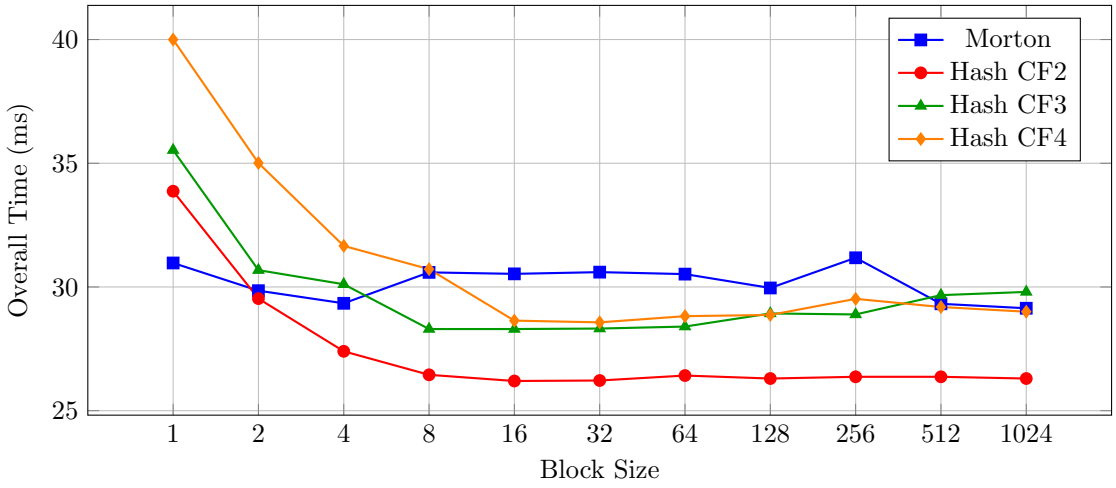

Figure 2: Overall execution time for voxel size 0.5

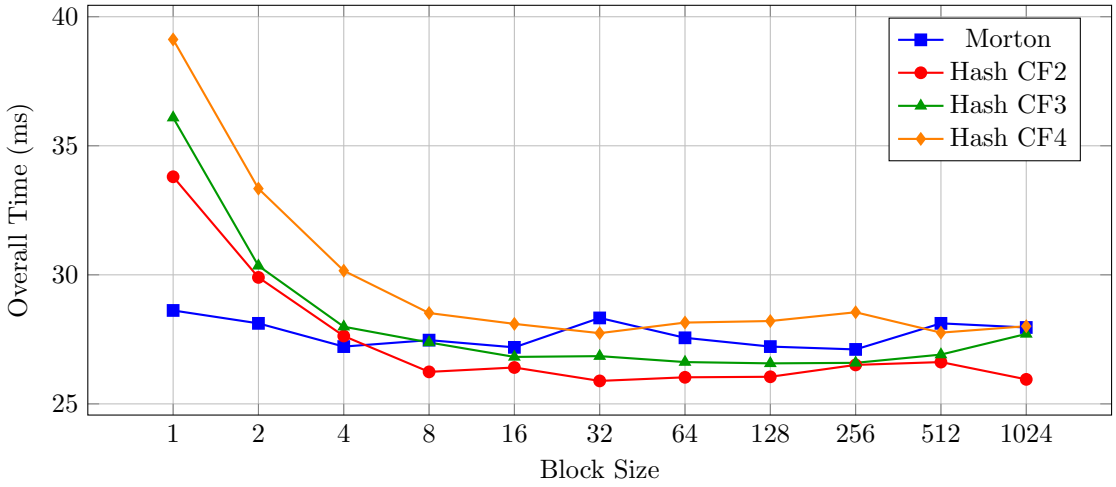

Figure 3: Overall execution time for voxel size 0.75

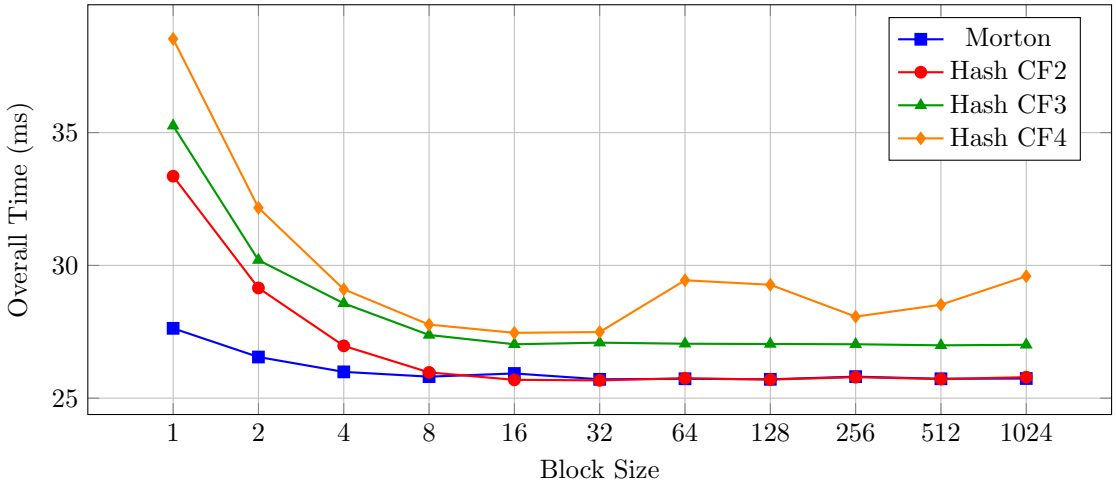

Figure 4: Overall execution time for voxel size 1.0

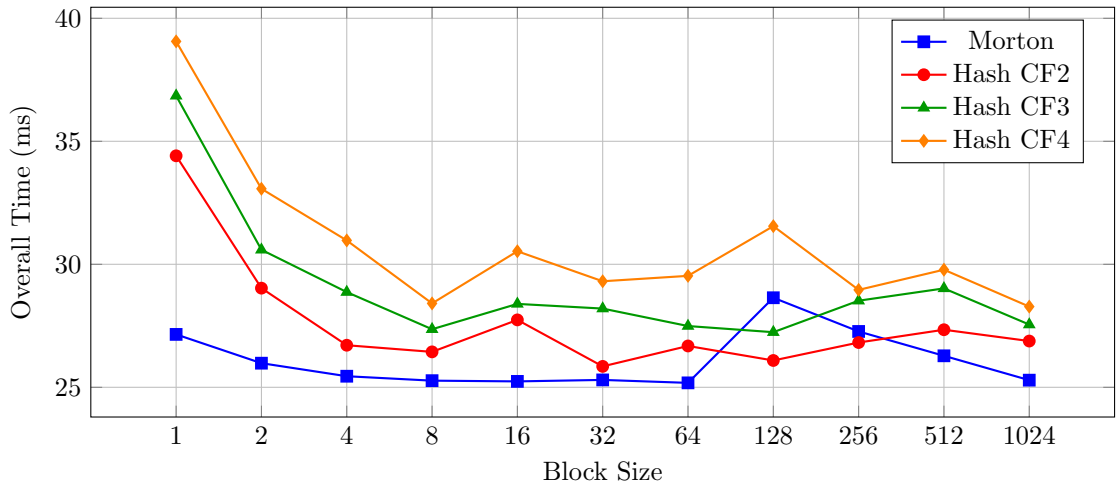

Figure 5: Overall execution time for voxel size 1.25

4 Possible Improvements

The main problem is that both implementations work with a fixed voxel size. This means that if the point cloud is very large but sparse, a lot of memory is wasted on empty space. A possible improvement would be to implement an octree structure, which would allow for variable voxel sizes depending on the density of points in a given area.

References

[1] M. Nießner, M. Zollhöfer, S. Izadi & M. Stamminger, "Real-time 3D Reconstruction at Scale using Voxel Hashing," ACM Transactions on Graphics (TOG), vol. 32, no. 6, 2013.

[2] "Morton encoding/decoding through bit interleaving: Implementations", Forceflow — Jeroen Baert's Blog, 7 October 2013. Available online: `https://www.forceflow.be/2013/10/07/morton-encodingdecoding-through-bit-interleaving-implementations/` (Accessed: 28 November 2025).

[3] NVIDIA Corporation, *"Thrust — CUDA Core Compute Libraries (CCCL)"*. Available at: `https://nvidia.github.io/cccl/thrust/` (Accessed: 28 November 2025).

[4] Wikipedia contributors, *"LAS file format"*, Wikipedia, The Free Encyclopedia. Available at: `https://en.wikipedia.org/wiki/LAS_file_format` (Accessed: 28 November 2025).

[5] Ayushi Sharma, *"From Point Clouds to Voxel Grids: A Practical Guide to 3D Data Voxelization"*, Medium, 2021. Available at: `https://medium.com/@ayushi.sharma.3536/from-point-clouds-to-voxel-grids-a-practical-guide-to-3d-data-voxelization-cf5991c1e7bb` (Accessed: 28 November 2025).