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 Theory

2.1 Point Cloud

A **point cloud** is a collection of data points defined in a three-dimensional coordinate system. Each point represents a position in space, typically described by (x, y, z) coordinates. Point clouds are commonly acquired using 3D scanners, LiDAR sensors, or photogrammetry systems. They are used in applications such as 3D modeling, robotics, mapping, and computer vision.

2.2 Voxel

A **voxel** (volumetric pixel) is the smallest unit of a 3D grid, similar to how a pixel is the smallest unit of a 2D image. Voxels divide 3D space into uniform cubes, allowing volumetric representations of shapes or environments. They are used in 3D reconstruction, simulation, gaming, and medical imaging.

2.3 Pointcloud to Voxel Grid Filtering

Pointcloud to voxel grid filtering converts an unstructured point cloud into a regular 3D grid. Space is divided into equal-sized voxels, and each point is assigned to its corresponding voxel. This reduces data density, removes redundant points, and creates a structured representation that is easier and faster to process for tasks such as downsampling and spatial queries.

2.4 Voxel Size

The **voxel size** is a critical parameter in pointcloud to voxel grid filtering. It determines the dimensions of each cubic cell in the 3D grid. A larger voxel size groups more points together, resulting in a coarser, more downsampled representation of the original point cloud. Conversely, a smaller voxel size produces a finer representation with higher spatial resolution but increased memory and computational requirements. The choice of voxel size depends on the application and the desired balance between accuracy and efficiency.

2.5 Morton Codes

Morton codes (also called Z-order curves) are a method of encoding multi-dimensional coordinates into a single integer value. They work by bit-interleaving the binary coordinates (e.g., x, y, z). This encoding preserves spatial locality, making it useful for data structures such as octrees and for accelerating spatial queries on GPUs.

2.6 CUDA Thrust Library

The **CUDA Thrust library** is a parallel algorithms library for NVIDIA GPUs. It provides high-level abstractions similar to the C++ Standard Template Library (STL), including parallel sorting, scanning, reduction, and vector operations. Thrust simplifies GPU programming by offering ready-to-use, highly optimized parallel primitives.

2.7 LAS Files

LAS files are a standardized file format used for storing LiDAR point cloud data. The format supports 3D coordinates, intensity values, classification labels, GPS time, color information, and other metadata. LAS is widely used in geospatial applications, surveying, and remote sensing due to its efficiency and interoperability.

3 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. This hashmap uses the same Morton encoding for voxel indexing, ensuring consistency between both methods. Both voxelizers share the same Morton encoding utilities so that a 3D voxel index is always mapped to the same 64-bit key, independent of the backend. On top of that, the Morton-and-sort voxelizer introduces its own per-voxel accumulation structure and reduction functor, while the hash-based implementation uses a dedicated hash bucket structure.

3.1 Shared Morton Encoding Utilities

Bit expansion helper. The first shared building block is the mapping from 3D integer voxel coordinates (i_x, i_y, i_z) to a single 64-bit Morton code. This is reused in *both* voxelizers: in the sorting-based pipeline it is used to generate keys for `thrust::sort_by_key`, while in the hash-based pipeline the same 64-bit Morton code serves directly as the hash key for open addressing.

The helper function `splitBy3` expands a 21-bit integer into a 64-bit value in which each original bit is separated by two zero bits:

```
__device__ __host__ inline uint64_t splitBy3(uint64_t v) {
    v = (v | (v << 32)) & 0x1f00000000ffffULL;
    v = (v | (v << 16)) & 0x1f0000ff0000ffULL;
    v = (v | (v <<  8)) & 0x100f00f00f00f00fULL;
    v = (v | (v <<  4)) & 0x10c30c30c30c30c3ULL;
    v = (v | (v <<  2)) & 0x1249249249249249ULL;
    return v;
}
```

Listing 1: Bit expansion of a 21-bit coordinate into Morton layout (shared).

Morton code generation. Using `splitBy3`, the actual Morton code is formed by interleaving the expanded bits of the three coordinates:

```
__device__ __host__ inline uint64_t mortonEncode(uint32_t x,
                                                 uint32_t y,
                                                 uint32_t z) {
    return (splitBy3((uint64_t)x) << 2) |
           (splitBy3((uint64_t)y) << 1) |
            splitBy3((uint64_t)z);
}
```

Listing 2: Morton encoding of 3D voxel indices (shared).

Both functions are marked `__device__ __host__ inline`, so they can be called from host-side test code as well as from all CUDA kernels in both voxelization backends.

3.2 Morton and Sort Voxelizer

To implement the Morton-and-sort voxelizer efficiently on the GPU, the pipeline is decomposed into: (i) Morton code generation via the shared utilities, (ii) per-voxel accumulation of point attributes, and (iii) CUDA kernels that bridge raw device arrays with Thrust's sort-and-reduce primitives. The accumulation structures and reduction functor described below are specific to this implementation and are *not* used by the hash-based voxelizer.

Per-voxel accumulation structure. To compute voxel centroids and average colors in a single pass after sorting, we define a compact accumulation structure `PointAccum`. It stores the running sums of coordinates and RGB components, as well as the number of points that contributed to the voxel:

```
struct PointAccum {
    float    sumX, sumY, sumZ;
    uint32_t sumR, sumG, sumB;
    uint32_t count;
```

```

    __device__ __host__ PointAccum()
        : sumX(0), sumY(0), sumZ(0),
          sumR(0), sumG(0), sumB(0),
          count(0) {}

    __device__ __host__ PointAccum(float x, float y, float z,
                                    uint8_t r, uint8_t g, uint8_t b)
        : sumX(x), sumY(y), sumZ(z),
          sumR(r), sumG(g), sumB(b),
          count(1) {}
};
```

Listing 3: Accumulation structure for voxel-wise reduction (Morton-and-sort only).

This structure is used as the value type in the subsequent reduction-by-key stage and is therefore only required by the sorting-based voxelizer.

Binary reduction functor. To use `PointAccum` with `thrust::reduce_by_key`, we define a binary functor `PointAccumOp`. It describes how two partial voxel accumulators are merged:

```
struct PointAccumOp {
    __device__ __host__
    PointAccum operator()(const PointAccum& a,
                          const PointAccum& b) const {
        PointAccum result;
        result.sumX  = a.sumX  + b.sumX;
        result.sumY  = a.sumY  + b.sumY;
        result.sumZ  = a.sumZ  + b.sumZ;
        result.sumR  = a.sumR  + b.sumR;
        result.sumG  = a.sumG  + b.sumG;
        result.sumB  = a.sumB  + b.sumB;
        result.count = a.count + b.count;
        return result;
    }
};
```

Listing 4: Binary operator for reducing `PointAccum` values (Morton-and-sort only).

This functor is specific to the reduction-based pipeline and is not needed by the dynamic hash map implementation.

Morton code computation kernel. The first CUDA kernel in this voxelizer takes the input point positions and converts them into Morton codes using the shared `mortonEncode` helper. Each thread processes exactly one point:

```
__global__ void computeMortonCodesKernel(
    const float* x,
    const float* y,
    const float* z,
    uint64_t* mortonCodes,
    float minX, float minY, float minZ,
    float invVoxelSize,
    size_t numPoints)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx >= numPoints) return;

    // Compute voxel indices (offset by min to ensure non-negative indices)
    uint32_t ix = (uint32_t)floorf((x[idx] - minX) * invVoxelSize);
    uint32_t iy = (uint32_t)floorf((y[idx] - minY) * invVoxelSize);
    uint32_t iz = (uint32_t)floorf((z[idx] - minZ) * invVoxelSize);

    mortonCodes[idx] = mortonEncode(ix, iy, iz);
}
```

Listing 5: Kernel computing Morton codes for each point (Morton-and-sort).

The kernel assumes a structure-of-arrays (SoA) layout (`x`, `y`, `z` in separate buffers) for coalesced memory access. The inverse voxel size is precomputed on the host to avoid divisions in device code.

Accumulator creation kernel. After sorting the points by their Morton codes, a `PointAccum` object is created for each point in sorted order. Rather than physically reordering all input arrays, a separate index array encodes the permutation induced by the sort:

```
__global__ void createPointAccumKernel(
    const float*  x,
    const float*  y,
    const float*  z,
    const uint8_t* r,
    const uint8_t* g,
    const uint8_t* b,
    const uint32_t* indices,
    PointAccum* accums,
    size_t numPoints)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx >= numPoints) return;

    uint32_t origIdx = indices[idx];
    accums[idx] = PointAccum(
        x[origIdx], y[origIdx], z[origIdx],
        r[origIdx], g[origIdx], b[origIdx]
    );
}
```

Listing 6: Kernel creating `PointAccum` objects after sorting (Morton-and-sort).

Integration in the outer loop. The outer host function for the Morton-and-sort voxelizer (not shown in full) orchestrates the following steps:

1. **Preprocessing:** Compute the global bounding box and `invVoxelSize` on the CPU.
2. **Device setup:** Upload positions and colors to device vectors; initialize an index array with `thrust::sequence`.
3. **Morton computation:** Launch `computeMortonCodesKernel` to fill the Morton code buffer using the shared `mortonEncode`.
4. **Sorting:** Call `thrust::sort_by_key` on the pair `(d_mortonCodes, d_indices)`.
5. **Accumulator creation:** Launch `createPointAccumKernel` to build a `PointAccum` per sorted point.
6. **Reduction by voxel:** Invoke `thrust::reduce_by_key` with `PointAccum` values and `PointAccumOp` to obtain one accumulator per occupied voxel.
7. **Final averaging:** Copy the reduced accumulators back to the host and divide sums by `count` to obtain centroid and color per voxel.

3.3 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 and $O(N \log N)$ complexity of the sorting-based pipeline. Instead of ordering points, it scatters them into a large hash table and accumulates voxel statistics in-place using atomics.

This implementation *reuses* only the shared Morton encoding helpers (`splitBy3` and `mortonEncode`). The per-voxel accumulation is performed directly inside each hash bucket, so there is no need for `PointAccum` or `PointAccumOp`.

Hash bucket layout. Each entry in the GPU hash table is represented by a tightly packed `HashBucket` structure:

```
struct __align__(16) HashBucket {
    unsigned long long key;  // 64-bit Morton code
    float sumX, sumY, sumZ;
    uint32_t sumR, sumG, sumB;
    uint32_t count;
};
```

```

#define EMPTY_KEY 0xFFFFFFFFFFFFFFFFULL
```

Listing 7: Aligned hash bucket structure and empty key sentinel (hash voxelizer).

The `__align__(16)` qualifier improves memory coalescing, and `EMPTY_KEY` serves as a sentinel to identify unused slots. The 64-bit `key` field stores the Morton code computed by the shared `mortonEncode` function.

Hash table initialization. Before insertion, the hash table is cleared by setting all keys to `EMPTY_KEY` and zeroing the accumulators:

```
__global__ void initHashMapKernel(HashBucket* table, size_t capacity) {
    size_t idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < capacity) {
        table[idx].key   = EMPTY_KEY;
        table[idx].count = 0;

        table[idx].sumX = 0.0f;
        table[idx].sumY = 0.0f;
        table[idx].sumZ = 0.0f;
        table[idx].sumR = 0;
        table[idx].sumG = 0;
        table[idx].sumB = 0;
    }
}
```

Listing 8: Kernel initializing the GPU hash table (hash voxelizer).

Hash-Based Voxel Accumulation using Atomic Open Addressing The core function of the hash-based voxelizer is the `populateHashMapKernel` CUDA kernel, which implements a scatter-and-accumulate pattern using a 64-bit Morton code derived from $3D$ integer coordinates (i_x, i_y, i_z) via `mortonEncode` as the key. Each thread calculates an initial hash slot $H =$ `mortonCode` $\pmod{\texttt{capacity}}$ and uses linear probing to find the correct `HashBucket`. Concurrency is managed by atomically claiming an empty slot using `atomicCAS` (Compare-and-Swap) with `EMPTY_KEY`, or by finding a slot already containing its key. Once the correct slot is secured, the thread uses atomic additions (`atomicAdd`) to safely accumulate the point's properties (position sums, color sums, and `count`) into the shared `HashBucket` structure, ensuring thread-safe data aggregation despite concurrent writes.

```
__global__ void populateHashMapKernel(
    const float*   x, const float*   y, const float*   z,
    const uint8_t* r, const uint8_t* g, const uint8_t* b,
    HashBucket* table,
    size_t capacity,
    size_t numPoints,
    float minX, float minY, float minZ,
    float invVoxelSize)
{
    size_t idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx >= numPoints) return;

    // 1. Quantize position to voxel grid and compute Morton key
    uint32_t ix = (uint32_t)floorf((x[idx] - minX) * invVoxelSize);
    uint32_t iy = (uint32_t)floorf((y[idx] - minY) * invVoxelSize);
    uint32_t iz = (uint32_t)floorf((z[idx] - minZ) * invVoxelSize);

    uint64_t mortonCode = mortonEncode(ix, iy, iz);

    // 2. Initial hash slot
    size_t hashIdx = mortonCode % capacity;

    // 3. Linear probing with atomic CAS and atomic adds
    for (size_t i = 0; i < capacity; ++i) {
        size_t currentSlot = (hashIdx + i) % capacity;

        unsigned long long oldKey = table[currentSlot].key;

        // Try to claim an empty slot
        if (oldKey == EMPTY_KEY) {
            unsigned long long assumed =
```

```
                atomicCAS((unsigned long long*)&table[currentSlot].key,
                          EMPTY_KEY,
                          (unsigned long long)mortonCode);
            if (assumed == EMPTY_KEY) {
                oldKey = mortonCode; // we now own this bucket
            } else {
                oldKey = assumed;    // another thread claimed it
            }
        }

        // If this bucket belongs to our Morton key, accumulate data
        if (oldKey == mortonCode) {
            atomicAdd(&table[currentSlot].sumX, x[idx]);
            atomicAdd(&table[currentSlot].sumY, y[idx]);
            atomicAdd(&table[currentSlot].sumZ, z[idx]);

            atomicAdd(&table[currentSlot].sumR, (uint32_t)r[idx]);
            atomicAdd(&table[currentSlot].sumG, (uint32_t)g[idx]);
            atomicAdd(&table[currentSlot].sumB, (uint32_t)b[idx]);

            atomicAdd(&table[currentSlot].count, 1);
            return;
        }

        // Otherwise: collision with a different key, continue probing
    }
}
```

Listing 9: Kernel inserting points into the GPU hash map (hash voxelizer).

Note that this kernel does *not* use `PointAccum` or `PointAccumOp`; all aggregation happens directly inside the `HashBucket`.

Counting valid voxels. Because open addressing leaves gaps in the hash table, a compaction pass is required. The first step is to count how many buckets are actually occupied:

```
__global__ void countValidBucketsKernel(
    HashBucket* table,
    size_t capacity,
    uint32_t* counter)
{
    size_t idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < capacity) {
        if (table[idx].key != EMPTY_KEY) {
            atomicAdd(counter, 1);
        }
    }
}
```

Listing 10: Kernel counting the number of occupied buckets (hash voxelizer).

Collecting results. A second pass converts the populated buckets into a dense array of voxel representatives. As in the sorting-based implementation, each voxel is represented by its centroid and average color:

```
__global__ void collectResultsKernel(
    HashBucket* table,
    size_t capacity,
    Point* output,
    uint32_t* globalCounter)
{
    size_t idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx >= capacity) return;

    HashBucket bucket = table[idx];

    if (bucket.key != EMPTY_KEY && bucket.count > 0) {
        uint32_t outIdx = atomicAdd(globalCounter, 1);

        float c = (float)bucket.count;

```

```
        Point p;
        p.x = bucket.sumX / c;
        p.y = bucket.sumY / c;
        p.z = bucket.sumZ / c;
        p.r = (uint8_t)(bucket.sumR / bucket.count);
        p.g = (uint8_t)(bucket.sumG / bucket.count);
        p.b = (uint8_t)(bucket.sumB / bucket.count);

        output[outIdx] = p;
    }
}
```

Listing 11: Kernel converting hash buckets into final voxel points (hash voxelizer).

Integration in the outer loop. The host-side driver for the dynamic hash map voxelizer can then be summarized as:

1. **Capacity selection:** Choose a hash table capacity as a multiple of the number of points (e.g. factor 2–4).
2. **Preprocessing:** Compute the bounding box and `inverse VoxelSize` on the CPU.
3. **Device setup:** Allocate device arrays for point attributes and copy input data.
4. **Hash table initialization:** Allocate and clear `HashBucket` array via `initHashMapKernel`.
5. **Scatter-and-accumulate:** Launch `populateHashMapKernel`, which uses the shared `mortonEncode` to derive keys and accumulates directly into buckets.
6. **Voxel counting:** Use `countValidBucketsKernel` to determine the number of occupied buckets and allocate a dense output array.
7. **Compaction:** Reset the counter and call `collectResultsKernel` to write voxel centroids and colors into the dense output array.
8. **Host transfer:** Copy the compact output back to the CPU and release GPU memory.

In this way, the two voxelizers share a single, consistent Morton encoding implementation, while each maintains its own accumulation and reduction strategy tailored to its parallelization scheme.

4 Results and Analysis

4.1 Methodology and Experimental Setup

All results were obtained on an NVIDIA GeForce GTX 1080 Ti (11GB GDDR5X) paired with an Intel Core i7-8700K CPU @ 4.8 GHz and 32GB DDR4 RAM, running Ubuntu 24.04.3 LTS. The test dataset consists of a dense point cloud scan of an urban environment containing 648,433 points.

To ensure robust results, the voxelizers were evaluated across a wide parameter space:

- **Voxel Sizes:** 0.25, 0.5, 0.75, 1.0, 1.25.
- **Block Sizes:** 1, 2, 4, 8, 16, 32, 64, 256, 512, 1024 threads per block.
- **Hash-Table Capacity Factors:** 2, 3, and 4 times the input point count.

Every configuration was executed 100 times to obtain average execution times and minimize transient system load variations.

4.2 Overall Performance: CPU vs. GPU

The transition to GPU-based voxelization yields substantial performance improvements. As shown in Tables 1 and 2, the GPU implementations achieve speedup factors ranging from **9.5× to 15.5×** compared to the CPU.

The most significant gains occur at smaller voxel sizes (e.g., 0.25), where the massive parallelism of the GPU is fully exploited. Even in worst-case GPU scenarios, the speedup remains between 6.1× and 12.2×.

Voxel Size	CPU (ms)	Morton (ms)	Hash CF2 (ms)	Hash CF3 (ms)	Hash CF4 (ms)
0.25	429.09	36.61	27.70	29.57	29.10
0.5	335.89	29.14	26.20	28.30	28.57
0.75	272.39	27.11	25.89	26.57	27.74
1.0	253.67	25.71	25.67	26.99	27.46
1.25	240.00	25.18	25.85	27.24	28.28

Table 1: CPU vs Minimum GPU Voxelization Time

Voxel Size	CPU (ms)	Morton (ms)	Hash CF2 (ms)	Hash CF3 (ms)	Hash CF4 (ms)
0.25	429.09	40.46	35.25	37.48	41.82
0.5	335.89	31.18	33.87	35.53	40.00
0.75	272.39	28.62	33.80	36.09	39.12
1.0	253.67	27.63	33.36	35.26	38.53
1.25	240.00	28.64	34.41	36.85	39.06

Table 2: CPU vs Maximum GPU Voxelization Time

4.3 Detailed Analysis of GPU Methods

4.3.1 Morton-Code Voxelizer

The Morton-based approach demonstrates consistent and predictable performance. It excels at larger voxel sizes (1.25 and above) where spatial coherence is high.

- **Timing Breakdown:** The runtime is dominated by the **sorting stage** (0.90–1.02 ms), which is constant across tests. Point accumulation takes 0.24–1.08 ms.
- **Sensitivity:** Very small block sizes (1–4 threads) significantly slow down Morton-code generation (up to 1.07 ms), whereas larger blocks stabilize this step to ≈0.04 ms.

4.3.2 Hash-Table Voxelizer

The hash-based method is generally faster for small-to-medium voxel sizes but is highly sensitive to the **Capacity Factor (CF)**.

- **CF 2 (Highest Efficiency):** Offers the best performance (25.67 ms at voxel size 1.0) due to low memory overhead and fast device-to-host transfers (≈2.71 ms).
- **CF 3 & 4 (Higher Overhead):** Increasing the table size reduces collisions but incurs significant penalties in initialization and memory transfer. CF 4 is 5–10 ms slower than CF 2, with transfer times rising to 5.26 ms.

4.4 Parameter Sensitivity and Bottlenecks

4.4.1 Impact of Voxel Size

Smaller voxel sizes generate a larger number of unique entries. This favors the **Hash-Table (CF 2)** method, which handles high fragmentation efficiently. Conversely, larger voxel sizes favor the **Morton** method, which benefits from predictable memory access patterns when spatial data is less fragmented.

4.4.2 Impact of Block Size

Both methods exhibit similar responses to thread block sizing:

- **1–4 threads (Inefficient):** Severe performance penalties due to insufficient warp utilization.
- **8–256 threads (Optimal):** The "sweet spot" with minimal overhead.
- **512–1024 threads (Saturated):** No significant improvement; performance is likely limited by register pressure.

4.5 Conclusions and Optimal Configurations

Based on the analysis, the optimal strategy depends on the target voxel resolution:

1. **Small Voxel Sizes (0.25–0.5):** Use the **Hash-based voxelizer (CF 2)** with a block size of 16–32 threads. It maximizes throughput by minimizing memory overhead.
2. **Large Voxel Sizes (1.25+):** Use the **Morton-code voxelizer** with a block size of 64–256 threads. It leverages spatial coherence to outperform the hash method.

Limitations: The Hash method is primarily bound by memory bandwidth (device-to-host transfer), while the Morton method is strictly bottlenecked by the sorting phase.

4.6 Visualizing Global Performance

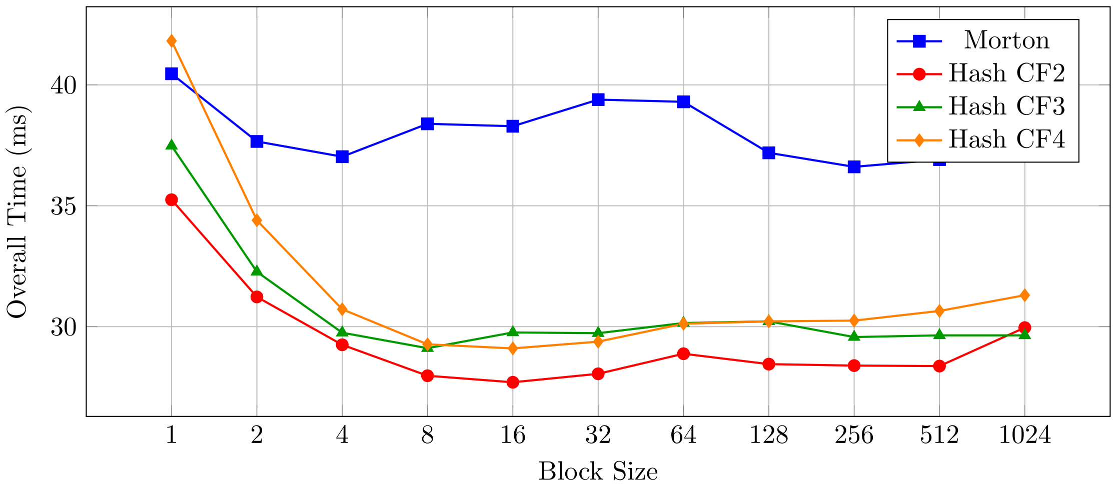

Figure 1: Overall execution time for voxel size 0.25

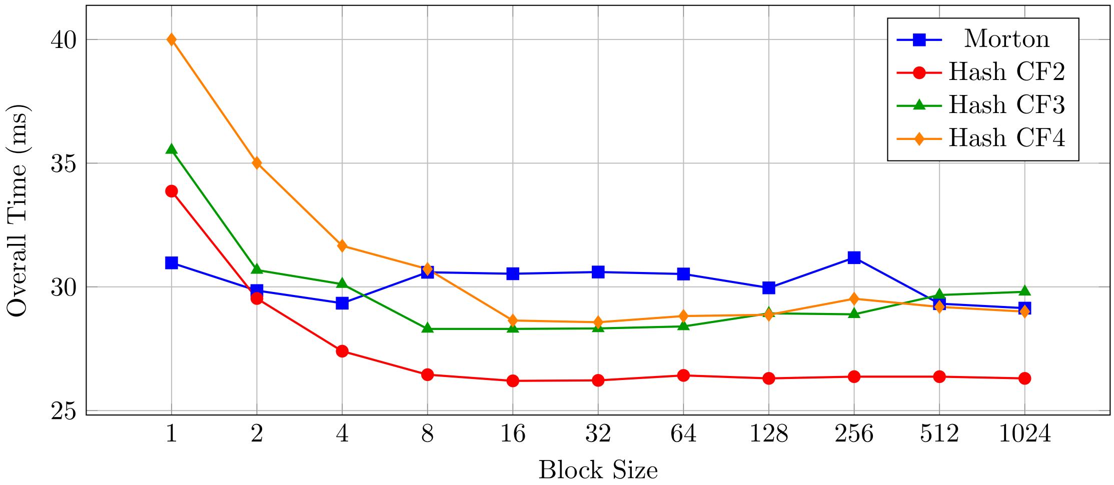

Figure 2: Overall execution time for voxel size 0.5

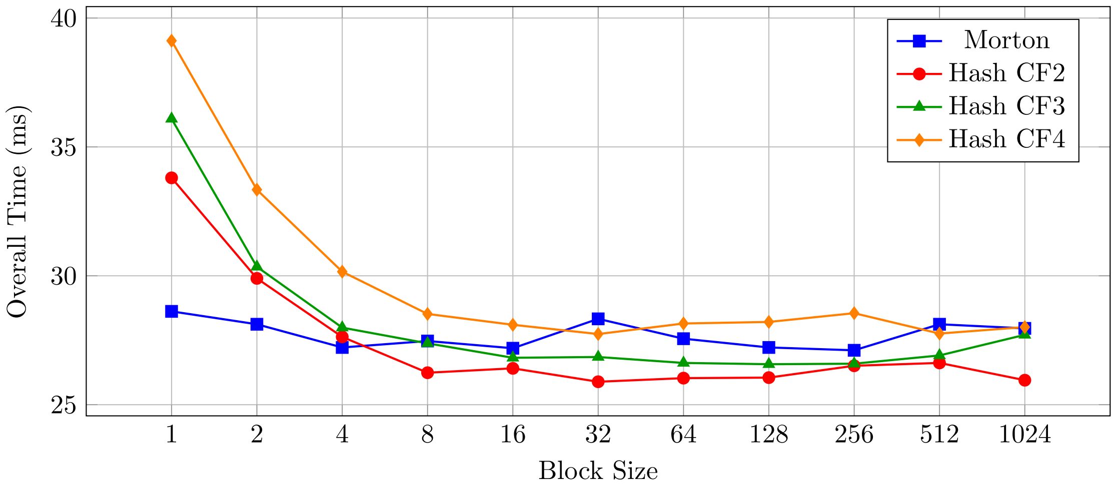

Figure 3: Overall execution time for voxel size 0.75

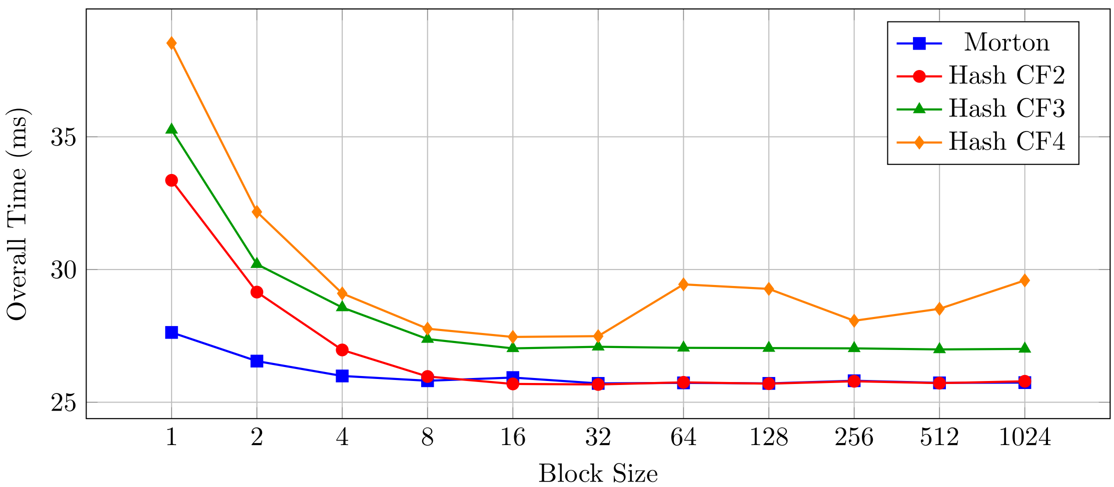

Figure 4: Overall execution time for voxel size 1.0

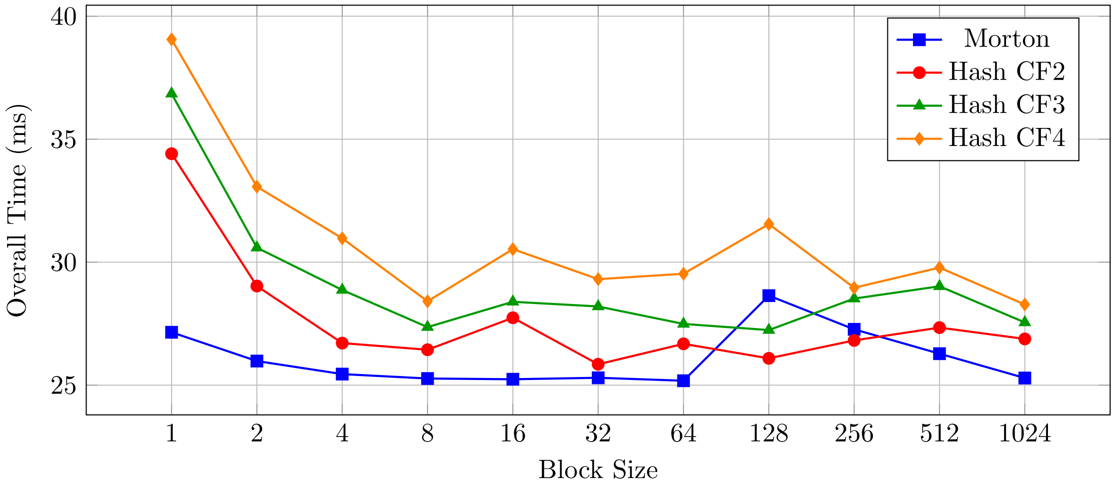

Figure 5: Overall execution time for voxel size 1.25

5 Visualisation

Everything was visualised using the python open3d library. This library allows for easy loading and displaying of point clouds. The original point cloud and the voxelized point cloud were displayed side by side for comparison. **TODOOOO : paste figures**

6 Conclusion and Future Work

This study confirms that GPU-based voxelization offers a significant performance advantage over CPU-based implementations, achieving speedups between 9.5× and 15.5×. While both the Morton-code and Hash-table approaches effectively parallelize the workload, their performance characteristics differ: the Hash method (Capacity Factor 2) excels at handling high-resolution grids (small voxels) due to lower overhead, whereas the Morton method provides stability and efficiency for lower-resolution grids (large voxels) via spatial coherence.

Despite these gains, two primary limitations remain:

1. **Data Transfer Bottleneck:** A substantial portion of the runtime is consumed by transferring data between host (CPU) and device (GPU). Reducing the volume of data transmission remains a key area for optimization.

2. **Memory Inefficiency:** Both implementations rely on a fixed voxel size. For large, sparse point clouds, the hash-table approach allocates memory for a global grid regardless of local point density, leading to wasted VRAM on empty space.

Future Improvements To address these issues, future work should focus on implementing an Octree data structure. Unlike fixed grids, an octree allows for variable voxel sizes, dynamically adapting resolution based on local point density. This would significantly reduce memory usage in sparse regions and potentially minimize the data transfer footprint, addressing the two main bottlenecks identified in this analysis.

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).