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 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.

2.5 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.

2.6 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 Voxel size

4 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.

Both voxelizers are built on top of a small set of shared GPU primitives: (i) Morton code generation, (ii) a per-voxel accumulation structure, and (iii) a binary reduction functor. These components are declared once in a common header so that both implementations refer to the same definitions. This guarantees that voxel identities and accumulation semantics are consistent across all backends and avoids code duplication.

4.1 Shared GPU Primitives

Morton code generation. 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 mapping 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).

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 utility code as well as from device kernels in both voxelizers.

Per-voxel accumulation structure. The second shared component is a small aggregation structure, `PointAccum`, used to collect sums of positions and colors along with a point count. This abstraction decouples the accumulation logic from the particular parallelization strategy (sorting vs. hashing):

```
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 (shared).

Binary reduction functor. Finally, the binary functor `PointAccumOp` defines how two partial accumulators are merged. In the Morton-and-sort voxelizer it is passed to `thrust::reduce_by_key`, and the same semantics can be reused anywhere voxel-wise statistics need to be combined:

```cpp
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 (shared).

Declaring these primitives in a shared header ensures that both voxelizers operate on identical Morton keys and apply the same centroid and color averaging rules.

4.2 Morton and Sort Voxelizer

To implement the Morton sort voxelizer efficiently on the GPU, we decompose the pipeline into a small set of reusable building blocks: (i) Morton code generation via the shared `mortonEncode` helper, (ii) per-voxel accumulation of point attributes using `PointAccum` and `PointAccumOp`, and (iii) CUDA kernels that bridge raw device arrays with Thrust's sort-and-reduce primitives. This section focuses on the kernels specific to the sorting-based implementation and on how they are orchestrated from the outer host function.

Morton code computation kernel. The first CUDA kernel takes the input point positions and converts them into Morton codes. Each thread processes exactly one point, computes its voxel indices based on the global bounding box and voxel size, and then encodes these indices using the shared `mortonEncode`:

```cpp
__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.

The kernel assumes a structure-of-arrays (SoA) layout (`x`, `y`, `z` in separate buffers), which yields coalesced memory accesses on the GPU. The inverse voxel size `invVoxelSize` is precomputed on the host to avoid a division in the kernel.

Accumulator creation kernel. After sorting the points by their Morton codes, we construct a `PointAccum` object for each point in sorted order. Instead of physically reordering all coordinate and color arrays, we maintain a separate index array `indices` that encodes the permutation induced by the sort:

```cpp
__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.

This design keeps all device arrays in their original layout and uses a single indirection to access the sorted points. It avoids unnecessary data movement and allows Thrust to operate directly on the Morton code and index arrays.

Integration in the outer loop. The outer host function (omitted here for brevity) orchestrates the Morton-and-sort voxelizer as follows:

1. **Preprocessing:** Compute the global bounding box and `invVoxelSize` on the CPU.
2. **Device setup:** Upload point coordinates and colors to device vectors; initialize an index array with `thrust::sequence`.
3. **Morton computation:** Launch `computeMortonCodesKernel` to fill the Morton code buffer.
4. **Sorting:** Call `thrust::sort_by_key` on the pair `(d_mortonCodes, d_indices)`, grouping points with identical Morton codes into contiguous segments.
5. **Accumulator creation:** Launch `createPointAccumKernel` to build a `PointAccum` per sorted point.
6. **Reduction by voxel:** Invoke `thrust::reduce_by_key` with the sorted keys and accumulator values, using `thrust::equal_to<uint64_t>` and the shared `PointAccumOp`. This yields one accumulator per occupied voxel.
7. **Final averaging:** Copy the reduced accumulators back to the host and divide the sums by `count` to obtain the centroid position and average color per voxel.

Because the Morton code computation kernel and the `PointAccumOp` functor are defined in a shared header, the same primitives can be reused by other backends (such as the hash-map voxelizer) without redefinition.

4.3 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 overhead required by sorting. Instead of ordering points, it scatters them into a large hash table and accumulates voxel statistics in-place.

As in the previous approach, the 64-bit Morton code generated by the shared `mortonEncode` function is used as a compact, unique key. Here, the code acts as the hash key for open addressing, allowing a three-dimensional voxel coordinate to be treated as a single scalar that can be compared and updated atomically.

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.

The `__align__(16)` qualifier ensures that each bucket starts on a 16-byte boundary, which improves memory coalescing and reduces bank conflicts. The special value `EMPTY_KEY` acts as a sentinel for uninitialized slots; virtually no valid Morton code will take this all-ones pattern.

Hash table initialization. Before insertion, the hash table must be cleared. The initialization kernel marks all buckets as empty and resets their counters and sums:

```
__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.

Scatter-and-accumulate kernel. The core of the hash-based voxelizer is the `populateHashMapKernel`. Each thread processes one point: it computes voxel indices, converts them to a Morton code using the shared `mortonEncode`, and then inserts or accumulates into the hash table using open addressing and atomic operations:

```
__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) {
                // Successfully claimed the slot; it now belongs to this Morton key
```

```
                oldKey = mortonCode;
            } else {
                // Another thread claimed it in the meantime
                oldKey = assumed;
            }
        }

        // Either we found our key, or we collided with another key
        if (oldKey == mortonCode) {
            // Accumulate position
            atomicAdd(&table[currentSlot].sumX, x[idx]);
            atomicAdd(&table[currentSlot].sumY, y[idx]);
            atomicAdd(&table[currentSlot].sumZ, z[idx]);

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

            // Increment point count
            atomicAdd(&table[currentSlot].count, 1);
            return; // Done with this point
        }

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

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

This kernel realizes a lock-free scatter-and-accumulate pattern: threads compete to claim buckets with `atomicCAS`, and once ownership is established, they use `atomicAdd` to update voxel statistics. The shared Morton encoder guarantees that both voxelizers use exactly the same voxel indexing scheme.

Counting valid voxels. Because open addressing leaves empty gaps in the hash table, a compaction pass is required. First, the number of occupied buckets is counted:

```
__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.

This provides the exact number of output voxels, allowing the host to allocate a tightly packed output array.

Collecting results. Finally, a second pass walks over the hash table and writes the averaged voxel representatives into a compact output array. An atomic counter is used to obtain a unique output index for each valid bucket:

```
__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.

This kernel mirrors the final averaging step of the Morton-and-sort voxelizer, but operates directly on hash buckets instead of reduced accumulators. Since voxel statistics are accumulated using the same semantics as `PointAccumOp`, the resulting centroids and colors are consistent across both implementations.

Integration in the outer loop. The host-side function that drives the dynamic hash map voxelizer follows these steps:

1. **Capacity selection:** Choose a hash table capacity as a multiple of the number of points (e.g. a factor between 2 and 4) to keep the load factor low and reduce probing lengths.
2. **Bounding box and quantization parameters:** Compute the global bounding box on the CPU and derive `invVoxelSize`.
3. **Device allocation:** Allocate device arrays for point coordinates and colors and copy the input data to the GPU.
4. **Hash table allocation and initialization:** Allocate a device buffer of `HashBucket`s and launch `initHashMapKernel` to mark all buckets as empty.
5. **Scatter-and-accumulate:** Launch `populateHashMapKernel`, which computes Morton keys (via the shared `mortonEncode`) and accumulates point statistics into hash buckets using atomic operations.
6. **Voxel counting:** Allocate a single device-side counter, set it to zero, and call `countValidBucketsKernel` to determine the number of occupied buckets, i.e. the number of output voxels.
7. **Output allocation and compaction:** Allocate a device array of `Point` with `numVoxels` entries, reset the counter, and launch `collectResultsKernel` to fill the array with centroid and color averages.
8. **Host transfer and cleanup:** Copy the compact output array back to the host, then free all device buffers.

Compared to the Morton-and-sort voxelizer, this hash-based approach trades the $O(N \log N)$ sorting phase for an $O(N)$ scatter-and-accumulate phase driven by atomic operations. Because both implementations share the same Morton encoding and accumulation definitions, their outputs are directly comparable: they differ only in how voxel membership is discovered (sorting versus hashing), not in how voxel identities or voxel-averaged attributes are defined.

5 Results and Analysis

5.1 Timing remarks and methodology

All the following results were obtained on relatively old hardware: a NVIDIA Geforce GTX 1080ti with 11GB of GDDR5X memory, paired with an Intel Core i7-8700K CPU @ 4.8 GHz and 32GB of DDR4 RAM and the pc runs ubuntu 24.04.3 LTS. The point cloud used for testing contains 648433 points, representing a dense scan of an urban environment.

5.2 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.

5.3 Performance Analysis per Method

5.3.1 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.

5.3.2 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).

5.4 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.

5.5 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.

5.6 Timing Breakdown

5.6.1 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

5.6.2 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)

5.7 Conclusions

5.7.1 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.

5.7.2 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.

5.7.3 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.

5.8 gpu vs cpu performance

TODOOOOOOOOOOOOOOOOOOOOOOOOO

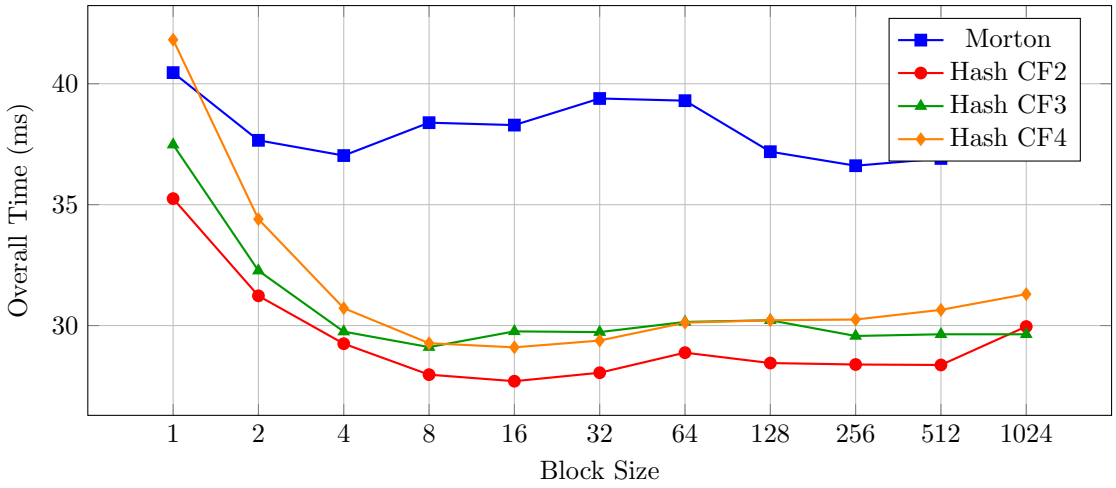

Figure 1: Overall execution time for voxel size 0.25

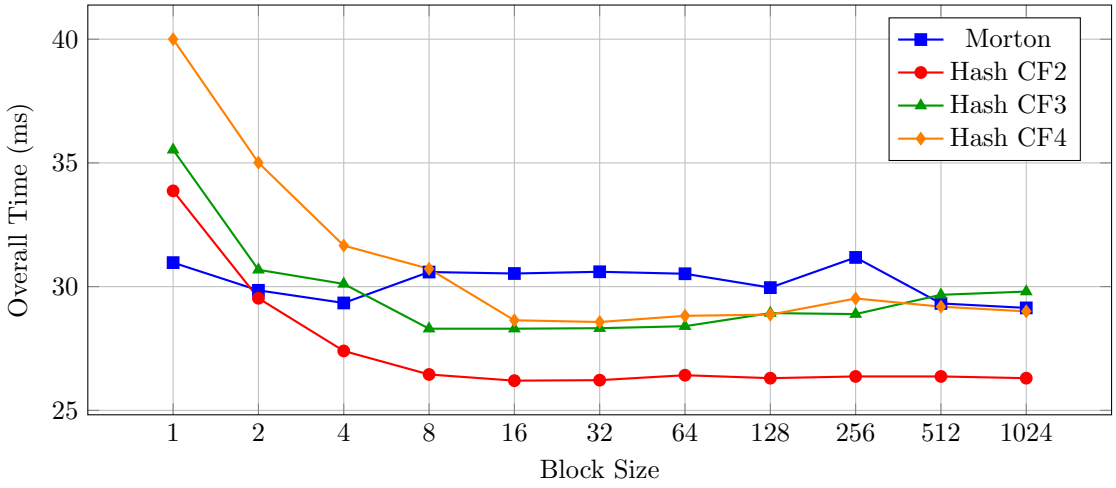

Figure 2: Overall execution time for voxel size 0.5

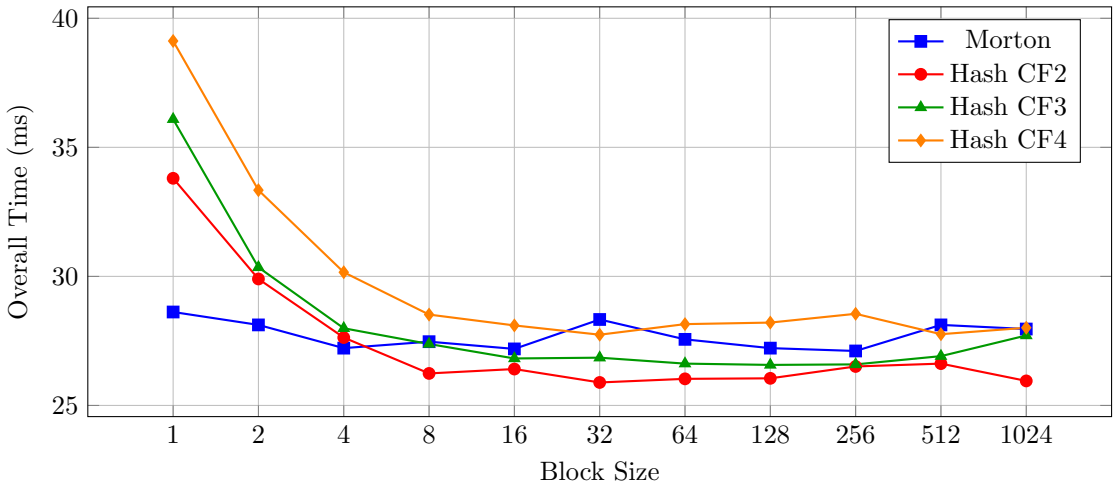

Figure 3: Overall execution time for voxel size 0.75

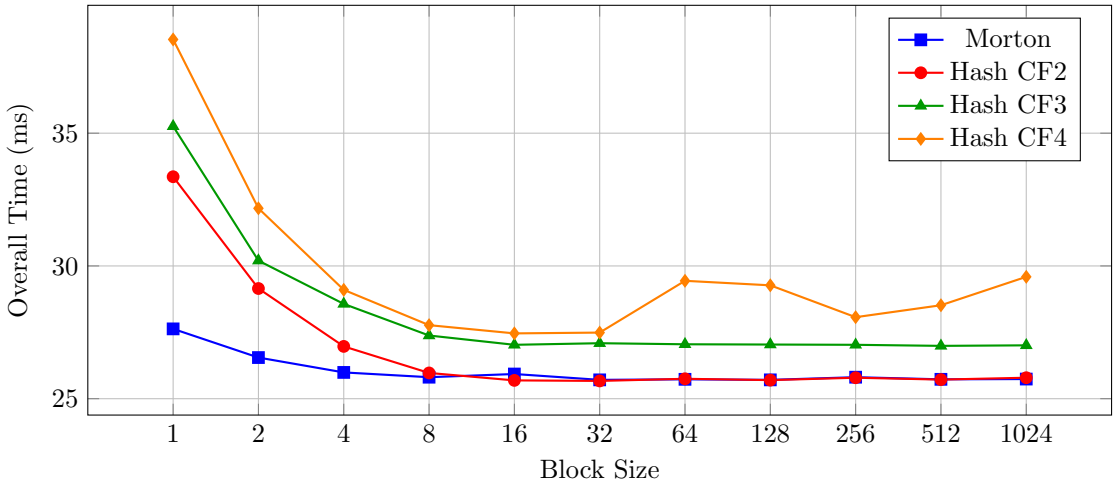

Figure 4: Overall execution time for voxel size 1.0

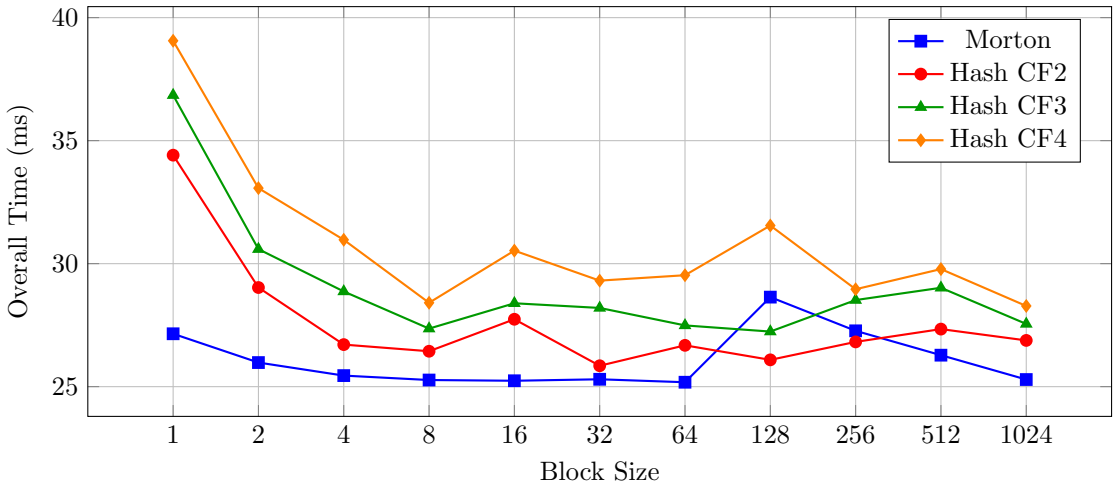

Figure 5: Overall execution time for voxel size 1.25

6 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**

7 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).