remote sensing

MDPI

Article

NanoMap: A GPU-Accelerated OpenVDB-Based Mapping and Simulation Package for Robotic Agents

Violet Walker *,†, Fernando Vanegas † and Felipe Gonzalez †

School of Electrical Engineering and Robotics, Queensland University of Technology, 2 George St., Brisbane, QLD 4000, Australia

* Correspondence: nanomap.dev@gmail.com

† These authors contributed equally to this work.

Abstract: Encoding sensor data into a map is a problem that must be undertaken by any robotic agent operating in unknown or uncertain environments, and real-time updates are crucial to safe planning and control. Most modern robotic sensors produce some form of depth data or point cloud information that is only useful to the agent after being processed into the appropriate data structure, oftentimes an occupancy map. However, as the quality of sensor technology improves, so does the magnitude of the input data, which can creates a problem when trying to construct occupancy maps in real-time. Populating such an occupancy map using these dense point clouds can quickly become an expensive process, and many robotic agents have limited onboard computational bandwidth and memory. This results in delayed map updates and reduced operational performance in dynamic environments where real-time information is crucial for safe operation. However, while many modern robotic agents are still relatively limited by the power of onboard central processing units (CPUs), many platforms are gaining access to onboard graphics processing units (GPUs), and these resources remain underutilised with respect to the problem of occupancy mapping. We propose a novel probabilistic mapping solution that leverages a combination of OpenVDB, NanoVDB, and Nvidia's Compute Unified Device Architecture (CUDA) to encode dense point clouds into OpenVDB data structures, leveraging the parallel compute strength of GPUs to provide significant speed advantages and further free up resources for tasks that cannot as easily be performed in parallel. An evaluation of our solution is provided, with performance benchmarks provided for both a laptop and a low power single board computer with onboard GPU. Similar performance improvements should be accessible on any system with access to a CUDA-compatible GPU. Additionally, our library provides the means to simulate one or more sensors on an agent operating within a randomly generated 3D-grid environment and create a live map for the purposes of evaluating planning and control techniques and for training agents via deep reinforcement learning. We also provide interface packages for the Robotic Operating System (ROS1) and the Robotic Operating System 2 (ROS2), and a ROS2 visualisation (RVIZ2) plugin for the underlying OpenVDB data structure.

Keywords: occupancy; mapping; simulation; ROS; ROS2; OpenVDB; NanoVDB; GPU; CUDA

Citation: Walker, V.; Vanegas, F.; Gonzalez, F. NanoMap: A GPU-Accelerated OpenVDB-Based Mapping and Simulation Package for Robotic Agents. *Remote Sens.* **2022**, *14*, 5463. https://doi.org/10.3390/rs14215463

Academic Editors: Kourosh Khoshelham, Martin Weinmann, Johannes Otepka and Di Wang

Received: 28 August 2022
Accepted: 25 October 2022
Published: 30 October 2022

1. Introduction

The fields of planning and control are areas of continued exploration within robotics research. A problem shared by these fields is the need for accurate, on-time information. Since the early days of robotics, occupancy grids have played a crucial role in providing robotic agents information about an environment [1]. While 2-dimensional (2D) occupancy grids have been predominately superseded by their 3-dimensional (3D) counterparts, occupancy-map-based solutions remain a core component of robotic solutions. Many planning and control solutions still utilise some form of occupancy map for determining the safe regions of an environment [2]. However, with the development of higher-fidelity planning and control software, the need for more detailed maps has grown. This increase in desired map resolution combined with an increase in the resolution of sensors can cause

even modern occupancy map solutions to become information bottlenecks as populating occupancy grids with dense point clouds is computationally expensive and is a problem still being explored today [3].

This is an issue further compounded in fields where agent size and computing power are restricted. These low-powered robotic agents are often forced to sacrifice some combination of map resolution, effective sensor resolution, or sensor update rate in order to process the sensor information in real-time. Within the growing field of Aerial Robotics, such compromises are routinely made in both software and hardware to enable operation as a result of considerable platform limitations [4–7].

A common 3D occupancy mapping solution used in modern robotics is Octomap, proposed by Hornung et al. [8]. The Octomap data structure has become a well-known and often-used mapping solution for robotics problems because of its probabilistic nature and its performance. Octomap enables the construction of 3D maps that contain measures of the uncertainty of the observed environment and by leveraging octrees to reduce the memory footprint of its maps. Unfortunately, the memory benefits of the hierarchical octree data structure are offset by the speed of operations on the data, creating bottlenecks for sensors with high update rates.

De Gregorio and Di Stefano [9] produced SkiMap in an attempt to improve performance of 3D occupancy mapping by parallelising data operations. It succeeds in cases using dense sensors such as red–green–blue-depth(RGB-D) sensors, but struggles compared to Octomap when dealing with sensors that produce large numbers of sparse data.

Besselman et al. [10] propose VDB-Mapping as a 3D occupancy mapping solution that utilizes the OpenVDB data structure to accelerate 3D occupancy mapping. It outperforms both Octomap and SkiMap and is a promising approach for robotic mapping, with the OpenVDB data structure having been used for years for managing and working on sparse data sets in the visual effects industry. Despite its promise, it still struggles on low-powered platforms, as it is a serial, CPU-bound approach. Macenski et al. [11] also propose the use of the OpenVDB data structure to improve the information available in a 3D occupancy map to improve robotic planning and control outcomes.

Finally, Jia et al. [12] propose a hardware-accelerated approach. They developed a dedicated 3D mapping accelerator to drastically accelerate mapping operations when using Octomap as the base data structure. While promising, this solution is not easily accessible for the wider robotics community.

This work extends the concept of probabilistic mapping using the OpenVDB data structure presented by Besselmann et al. [10], off-loading the computationally expensive component of filtering and ray-casting points to the highly parallel GPU while additionally leveraging useful aspects of the OpenVDB data structure to further accelerate CPU performance during map insertion. The work is evaluated on multiple platforms against the existing VDB-Mapping code provided by Besselmann et al. and Octomap.

Additionally the proposed NanoMap library provides methods to very quickly simulate the construction of a map-using, user-defined sensor views within an existing OpenVDB grid; this enables training and validation of planning and control solutions prior to implementation on hardware.

The structure of this paper is as follows. Section 2 describes the NanoMap software solution and details working with a graphics processing unit (GPU)-based occupancy map approach. This section also outlines the simulation component of the solution. Section 3 contains the evaluation and analysis carried out on a number of computing platforms, comparing the overall performance of the solutions. Section 4 concludes, outlining goals for future works and additions to be made to the library.

2. NanoMap

The type and quality of sensors are key considerations when outfitting robotic agents. While some robotic agents use light detection and ranging (LIDAR) systems, many mobile robotic agents use RGB-D or stereo cameras as they are often less expensive and more lightweight than their equivalent LIDAR counterparts. The Nanomap library provides

functionality for both kinds of sensors; however, the most significant performance advantages are provided when used with dense sensors, referred to within the library and this work as Frustum Sensors. Like many other occupancy-mapping solutions NanoMap requires an accurate pose, and the corresponding point cloud for processing, it additionally requires the user to supply some sensor information when creating the mapping instance. This information is used to allocate the necessary memory for interacting with the GPU prior to operation.

The core operation of NanoMap is very similar to that of its predecessors. Take an input point cloud in the form of points and an agent pose. For each point in the cloud, create a ray with a start location at the sensor pose, and an end location at the point. Then, step along each ray over a temporary grid and encode into the occupancy map all the cells encountered along the ray as empty and the cell occupied by the end of the ray as occupied.

Obviously, accessing these cell locations within a map would be time-consuming if not using some form of accelerated data structure. Octomap as mentioned in the introduction solves this issue using octrees. Within an octree, each node of the tree contains 2^3 or eight children. While this is sufficient for reducing the memory usage of a volumetric grid, this can be problematic when dealing with high-resolution grids as the number of layers in the tree can grow to a size that makes insertion and lookup comparatively costly. OpenVDB side-steps this issue with its state-of-the-art data structure designed for placing a hard limit on data manipulation times for sparse grids. Additionally NanoVDB is a module of the OpenVDB library that focuses on working with and using OpenVDB structures on the GPU.

Consequently this work follows in the footsteps of Besselmann et al. [10] and Macenski et al. [11] with our use of the OpenVDB data structure as the core data structure for our occupancy grids. It also parallelises the data processing operations as is the case with SkiMap [9] but leverages access to the hugely parallel GPUs now available to many robotics platforms; this prevents the need for the dedicated hardware accelerator proposed by Jia et al. [12].

The following subsections outline the details and advantages of OpenVDB, introduce the GPU-focused library NanoVDB and the difficulties faced with interfacing such data structures with a GPU, and finally detail the approach used by NanoMap to leverage the GPU to accelerate occupancy mapping.

2.1. OpenVDB and NanoVDB

OpenVDB [13] is a data structure developed for the efficient representation and manipulation of sparse volumetric grids. Originally developed by Dreamworks Animation to enable improved performance during simulation and renders of particle systems, the library is useful for storing any spatially organised information thanks to its impressive speed and flexibility.

The VDB grids supplied by the OpenVDB project are freely configurable tree structures that share a number of similarities with B+ Trees. Some have referred to VDB grids as "Volumetric Dynamic B+ tree grids" as a result, but the simple truth is that VDB itself is just a name. A key advantage of the VDB data structure that sets it apart from Octrees is that a VDB grid has a steep branching factor and user-programmable depth. The steep branching nature of VDBs means that lookup and insertion operations on the grid have a small maximum number of steps because there are a small fixed number of parents for any given voxel.

Furthermore, VDB grids implement caching during look-ups and insertion, meaning that for a group of look-ups or insertions that are spatially coherent, only the initial lookup has to traverse the entire tree and each subsequent lookup references the cache, oftentimes finding the target location in memory for the next operation immediately. In the case of an initial failure, the tree is traversed in reverse order from the cached location, meaning that time savings can still occur even when the data points are not strictly adjacent. These optimisations mean spatially coherent look-ups and insertions are extremely efficient when done in bulk compared to traditional data structures like octrees. Furthermore, as

demonstrated by M. Besselmann et al. [10] these features make VDB grids ideal for use as an occupancy map data structure.

Unfortunately, while impressively quick, there are still aspects of the VDB data structure and the point cloud insertion problem that are limited by the poor parallel capabilities of a central processing unit (CPU). While threading is an option, CPUs still lack the number of threads to meaningfully accelerate the processing of dense point clouds with tens to hundreds of thousands of points. This is where NanoVDB [14] and Nvidia's Compute Unified Device Architecture (CUDA) [15] become relevant to the problem.

Given the original rendering and simulation focus of the OpenVDB project, it was only a matter of time until key components of the project were developed to use the Nvidia CUDA GPU API, leveraging the impressive parallel operations provided by Graphics Processors. This development came in the form of the NanoVDB module.

NanoVDB is a module developed primarily to enable improved rendering of OpenVDB data structures. Due to the nature of interfacing with GPUs, the sparse nature of the OpenVDB data structure can be loaded into and read from the GPU to enhance render speeds, but only as a read-only object. This is because dynamically manipulating memory structures on the GPU to insert new entries into the grid is not possible. The exception to this case is when using a dense grid representation of an OpenVDB grid because all voxel locations in the grid are already allocated, but doing so completely removes the advantages of the OpenVDB data structure and would have extreme memory requirements. As a result, the explicit NanoVDB grid structure is only used by the NanoMap library during simulation. However, the NanoVDB module does still provide useful objects and tools that can be used on the GPU and are essential for operation of the NanoMap library. The only other time when NanoVDB grids might be directly utilised on the GPU and CPU at the same time is by systems that have unified system memory, such as the Nvidia Jetson Devices, and this is an approach to be explored in future works.

2.2. GPU-Accelerated Mapping

Given that sparse NanoVDB grids objects are read-only on traditional systems with discrete GPUs, how does NanoMap accelerate the mapping process?

Memory management between the CPU and the GPU was the core problem with implementing NanoMap. Firstly, the input point cloud needed to be copied to the GPU, which is a relatively trivial operation using the CUDA application programming interface (API), and few optimizations could be made because the point cloud would be a fixed size and would need to be recopied every loop. Thankfully, copying even a million points between pinned memory on the CPU and GPU does not take long at all on modern systems. For extremely large sensor clouds, the cloud could be copied over in a reduced format, using only distance readings that map to a particular ray in the sensor model. This would shrink the size of copy operations, but so far this has not been necessary.

The key issue with memory management was maintaining a reasonable memory footprint and copy time for the data structures that contain the ray-casting results. The solution provided by this work leverages the tree structure of the VDB grid to initially ray cast at the leaf node level of the grid to calculate which leaf nodes have rays pass through them. Algorithm 1 provides an overview of the algorithm used by the NanoMap library with the steps described further in this section.

In the case of an OpenVDB grid, the leaf nodes of the grid are the nodes in the tree that contain voxels. Using the default configuration for grids, leaf nodes in NanoMap contain 8 voxels on each side, meaning each leaf node contains 512 voxels. A leaf node that has a ray pass through it is considered active. Once the active leaf nodes have been determined, they are assigned an index ranging from 0 through to the number of currently active leaf nodes. The library then populates a buffer using the assigned index for each node, and the origin of that node relative to the agent. This maps each active node to 3D space. This means that even with a larger sensor model, only the nodes that contain new information are copied to the CPU memory. Figure 1 shows how this initial process would work for a 2D case.

Algorithm 1 Processing a point cloud using Nanomap

```
if GPU Enabled then
    Allocate Memory on GPU and CPU at initialisation
    while there exists unprocessed point cloud do
        Copy Cloud to GPU Memory
        Calculate sensor offset from current occupied leaf node
        Calculate the offset of the aforementioned leaf node from the origin of the grid
        Ray-cast cloud in parallel, making note of each node a ray passes through
        Index each active node and store its relative position to the origin
        if size of voxel arrays not sufficient for active nodes then
            Re-allocate voxel level arrays
            Zero voxel arrays
        else if size of voxel arrays sufficient for active nodes then
            Zero voxel arrays
        end if
        Ray-cast at voxel level, populating voxel array according to sensor properties
        Reduce the voxel array from a float (4 Byte) array to a single byte array
        Copy the voxel byte array and the leaf index array to CPU memory
        Using the leaf index array and voxel byte array, update the occupancy grid
    end while
else if CPU Enabled then
    while there exists unprocessed point cloud do
        Ray-cast the cloud in serial
        Populating temporary occupancy grid during ray-casting operation
        Iterate over temporary grid and update primary occupancy grid as required
    end while
end if
```

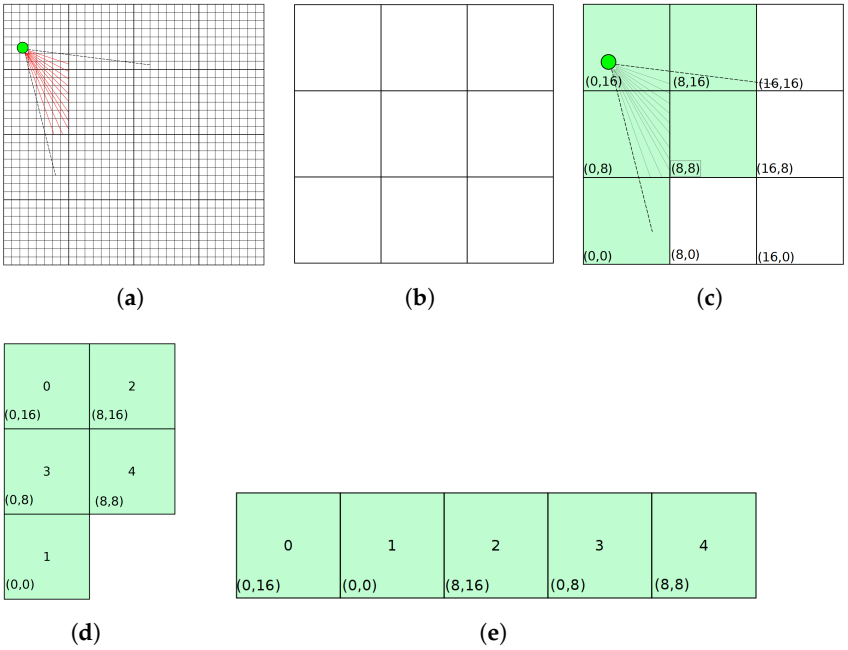


Figure 1. Node level ray-casting. Used to determine the nodes within the potential view-space of the sensor that are traversed by the rays defined by the input point cloud. (**a**) Agent receives point cloud data. (**b**) Potential area of new information according to known sensor properties. (**c**) Active nodes detected via ray-casting. (**d**) Each active node assigned a unique index. (**e**) Leaf Index array populated with active leaf node information.

Once each node is assigned an index, a float array is allocated by the GPU with a size equal to the active leaf node count multiplied by the volume of a leaf node in voxels (512 by default). If an array of appropriate size already exists, it is zeroed rather than reallocated. The GPU then ray-casts a second time, iterating at the voxel level. Whenever a new node is traversed by the ray, the index of that node is fetched, and then all voxels that are traversed while inside that node are added to the buffer at the location corresponding to the node index with an offset determined by the XYZ location of that voxel within the node. When a ray passes through a voxel but does not terminate within that voxel, the sensor's log odds miss value is added to the location; the log odds hit value is added for voxels that contain the end point of the ray. The atomic add function provided by CUDA is used so that the additions to the buffer remain accurate even when performed in parallel. Figure 2 shows how the process from Figure 1 is continued at the voxel level.

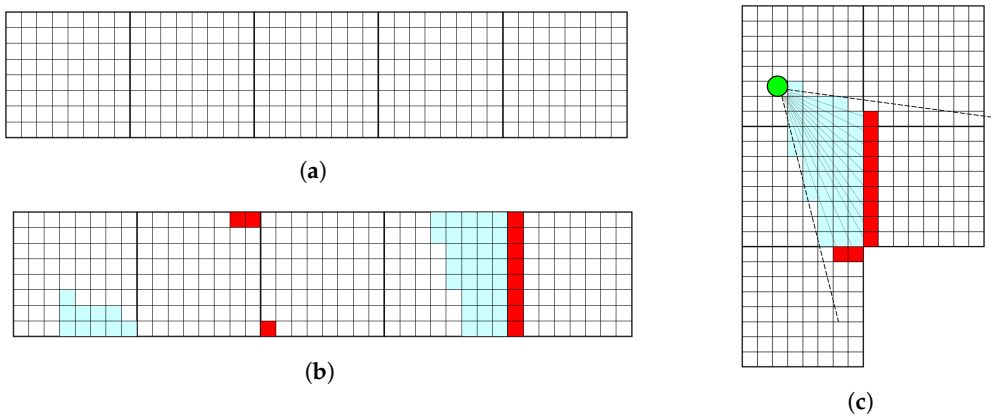


Figure 2. Voxel level ray-casting. (**a**) Voxel-level array allocated or zeroed. (**b**) Voxel array populated with ray-casting results. (**c**) Voxel-level ray-casting performed.

The values in the voxel array are then copied by the GPU from their float array to an equivalent single byte integer array. This is carried out to further shrink the size of the array to improve the copy speed from the GPU to the CPU. Prior to copying the float values to the buffer, they are multiplied by one hundred (100) to preserve their first two decimal values and then capped at -128 and 127, meaning that the range for cell updates is limited to between -1.28 and 1.27 during a single update. Because NanoMap uses a log-odds style update similar to OctoMap and other probabilistic mapping techniques, the slight loss in precision is acceptable. Values in the occupancy grid are capped between -0.87 and 1.5, allowing relatively large updates to still be made to each voxel at each time step.

At this point, the ray casting data are on the CPU, and the primary GPU functionality of NanoMap ceases until the next point cloud is received.

The CPU then begins to process the data retrieved. There are two main approaches currently implemented, one that works directly at the voxel level, and another that works at the node level. Depending on the characteristics of the problem and input point clouds, either approach can be applicable. The voxel approach is simple; NanoMap iterates over the input buffer and updates the occupancy grid when encountering any non-zero values. The node level approach leverages the fact that the data received from the GPU arrive as a buffer of nodes each containing 512 voxels; sometimes it makes sense to avoid voxel-level interactions with the existing map when dealing with extremely dense information. This approach involves probing the existing map at a given leaf-node location. If no leaf node exists, the leaf node provided by the GPU is processed by itself and inserted as a new node into the Map.

If a node does exist, the values of the existing node are considered when processing the new data, and the whole node is replaced with the new processed node. In doing this, the number of VDB grid queries and insertions is significantly reduced. One benefit to this process is that the leaf nodes can be handled in parallel using Intel's Threaded Building Blocks to perform the population of the leaf nodes in parallel.

In addition to the regular raycasting operation defined above, Nanomap provides a GPU-accelerated voxelisation filter. This filter maps all points in the point cloud to their corresponding terminal voxel before ray-casting. Then, instead of ray-casting each point, ray-casting is only performed once for each active voxel. There are a number of modes available for this discrete voxelisation filter, each with its own trade-off. The key difference between filtering modes is the type of data stored, and how those data are stored. Table 1 outlines the four available filter types, the minimum CUDA architectures they are compatible with, and their memory consumption as a percentage of the memory used by Filter Mode 0, whether the filter is "precise" or "simple", and the data-type used by the filter. The precise filter modes track the average offset for all rays that terminate within a voxel to use as an endpoint for raycasting, while the simple filter modes simply count the number of rays that terminate within a voxel and use the centre of the voxel as the termination point, trading accuracy for memory consumption.

Table 1. NanoMap filter modes.

Filter Mode	CUDA Arch Support	Memory Usage	Filter Type	Data Type
Mode 0	>2.x	100%	Precise	float
Mode 1	>6.x	50%	Precise	half2
Mode 2	>2.x	25%	Simple	int
Mode 3	>2.x	6.25%	Simple	uint8_t

The number of points and the average of their locations are maintained within each voxel to maintain the correct information for the raycasting and probabilistic calculations. This optimisation significantly increases processing speed for cases where the grid resolution or point cloud density results in multiple points occupying a single voxel. The downside to the filter is the increased memory usage needed. This is a result of needing to pre-allocate an array that encompasses the entirety of the information space of a given sensor. This makes the voxelisation filter potentially unusable for long-range LIDAR sensors. Additionally, because of the sparse nature of the information provided by LIDARs, they do not benefit from the ray reduction provided by the voxelisation filter.

When it comes to laser or LIDAR functionality, Nanomap currently only provides CPU-based processing. Due to the sparseness and often lower resoluton of LIDAR sensors, they do not currently benefit from GPU accelerations during ray-casting, as any performance they gain from parallel processing is lost due to overhead. Investigating potential GPU-based optimisations for LIDAR sensors within NanoMap is an area for future work. The optimisations provided by NanoMap in its current form rapidly accelerate the processing of a dense cloud of information where rays produced by the point cloud often traverse the same space. This makes NanoMap particularly good at accelerating sensors with a Frustum view shape; RGB-D and Stereo cameras in particular benefit significantly from the use of the GPU.

2.3. Simulation and Robotic Navigation

In addition to the mapping component of NanoMap, the simulation component benefits from the acceleration of the GPU and the NanoVDB Grid data structure, making it quite useful for simulating agent sensing and mapping within an existing OpenVDB Grid. A method is provided for generating randomised voxelised caves using 3D Cellular Automata to use in simulations and deep reinforcement learning. The only difference between the operation of the mapping in the simulation is that the point cloud is generated on the GPU from a sensor model and a provided grid of the simulation environment, rather than copied from an input. The simulation component of NanoMap was developed to enable the validation of planning and control methodologies that rely on building

occupancy maps, and to enable the rapid training of deep reinforcement learning agents by reducing the step time taken to generate sensor and mapping information.

Additionally, packages have been written that interface the Nanomap library with the Robotic Operating System (ROS1) and the Robotic Operating System 2 (ROS2); the library has been tested on Melodic, Noetic, and Galactic. Simulation and server functionality is provided. Simulation functionality takes some external pose input and sensor information and generates a point cloud against a user-provided environment before processing that point cloud into a published occupancy grid. Server functionality provides an ROS node that listens for a pose input and a point cloud and publishes an occupancy grid. It still requires the sensor to be defined in a configuration file.

A benchmarking package is also available for ROS1 and ROS2 with functionality that compares the performance between an optimised CPU-only version of the approach presented by M. Besselmann et al. [10], GPU Nanomap with a variety of operation types, and Octomap with and without the discrete optimisation. This benchmarking package was used for the evaluation contained in Section 3.

Finally, an ROS2 visualisation (RVIZ2) plugin compatible with ROS2 Galactic has been written to enable streaming and rendering of OpenVDB grids to avoid first having to convert them into point clouds.

Figure 3 shows the map created by a simulated agent operating within a randomly generated "cave" environment.

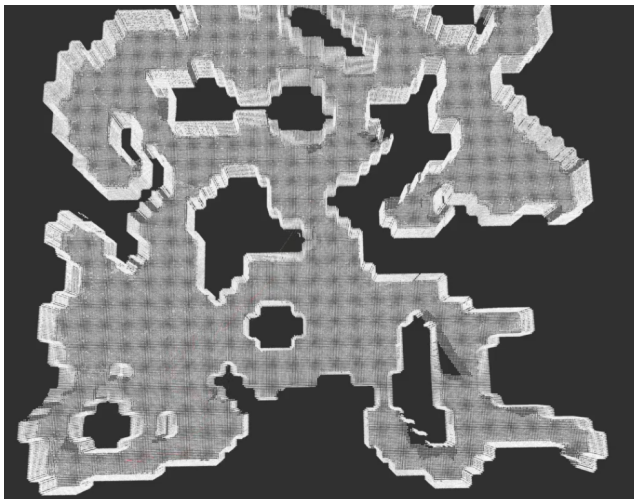


Figure 3. NanoMap mapping simulation result.

A video (https://youtu.be/UBrlLRqY_E4) is provided that shows simulation runs using a Frustum Sensor, a Laser Sensor, and a simulation run using a combination of the Frustum and Laser Sensors. These simulations were carried out using using the Nanomap ROS2 Galactic package and the Nanomap RVIZ2 Render plugin.

3. Evaluation

Evaluation was performed on both a laptop containing a Ryzen 4900HS and Nvidia RTX 2060 MaxQ GPU and a Jetson Nano Single Board Computer. Small GPU-enabled computers are the target platform for the NanoMap solution, but even laptops and desktops can benefit from the enhancements during simulation.

3.1. Frustum Based Testing

For testing the improvement provided to Frustum-based point cloud processing, the Freiburg Long Office Household Dataset from the TUM Computer Vision Lab was used as the input data. The down-sampled rosbag with ground truth was used as the basis for the benchmark input data.

ROS Melodic was used for benchmarking on the Jetson Nano, while Galactic was used on the laptop. The rosbag was first loaded into memory before being processed frame by frame. The time per frame was recorded, and with the average time calculated at the conclusion of each benchmark.

Table 2 outlines the methods tested and their key features. The benchmarks were each run 10 times with a target grid resolution of 0.1 m, 0.05 m, 0.025 m, and 0.01 m on the laptop, and 0.1 m, 0.05 m and 0.025 m on the Jetson Nano device. Figure 4 and Table 3 show the differences in GPU memory consumption for each of the GPU-based methods for each mapping resolution.

Table 2. Key features of evaluated methods.

Methods Tested	CPU Ray Cast	GPU Ray Cast	Discrete Voxel Filter
NM Filter 0	No	Yes	Yes
NM Filter 1	No	Yes	Yes
NM Filter 2	No	Yes	Yes
NM Filter 3	No	Yes	Yes
NM No Filter	No	Yes	No
NM CPU Only *	Yes	No	No
Octomap **	Yes	No	No
Discretized Octomap ***	Yes	No	Yes

* An improved implementation of the method presented by [10]. ** Octomap InsertPointCloud method with lazy eval = false, and discretize = false. *** Octomap InsertPointCloud method with lazy eval = false, and discretize = true.

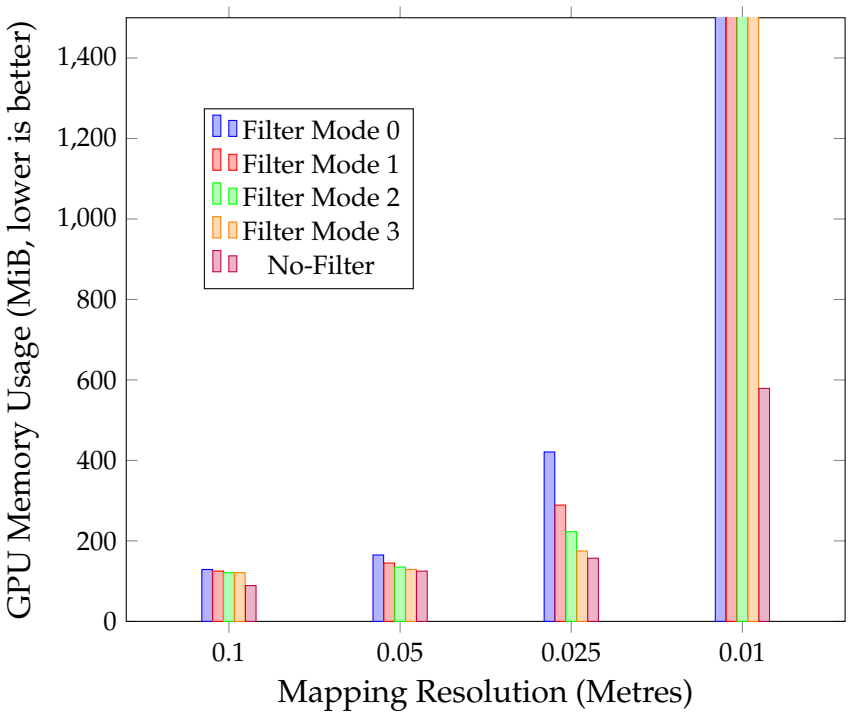


Figure 4. NanoMap GPU memory consumption.

Table 3. NanoMap GPU memory consumption (MiB).

Methods Tested	Mapping Resolution (Metres)			
	0.1	**0.05**	**0.025**	**0.01**
NM Filter 0	129	165	421	4371
NM Filter 1	125	145	289	2475
NM Filter 2	121	135	223	1527
NM Filter 3	121	129	175	817
NM No Filter	89	125	157	579

Table 4 and Figures 5 and 6 show the average frame times for each method (in milliseconds) at the benchmarked resolutions for the laptop platform. Table 5 and Figures 7 and 8 show the results for the same testing on the Jetson Nano platform. Due to limitations of the architecture, not all methods could be tested on the Jetson Nano. The age of the GPU architecture of the Jetson Nano means some CUDA features, data structures, and operations are not readily available and so not all NanoMap methods are usable on the platform. Additionally, the limited memory of the platform means running Filter Mode 0 of NanoMap at 0.01 m grid resolution would crash the system. Figures 9–12 show the difference in the occupancy maps produced by the methods tested at each resolution tested. There was not a significant difference between the two octomap methods, Filter Modes 0 and 1, and Filter Modes 2 and 3. The OpenVDB grids produced by Nanomap were converted to the Octomap.bt format for easier visual comparison. However, part of this process resulted in the heightmap values or scale differing between the octomap and nanomap values when rendered using octovis, causing the colour differences. What is important in comparing the results is the occupancy value of voxels and subsequent shape of the map, and not the respective colours of each voxel.

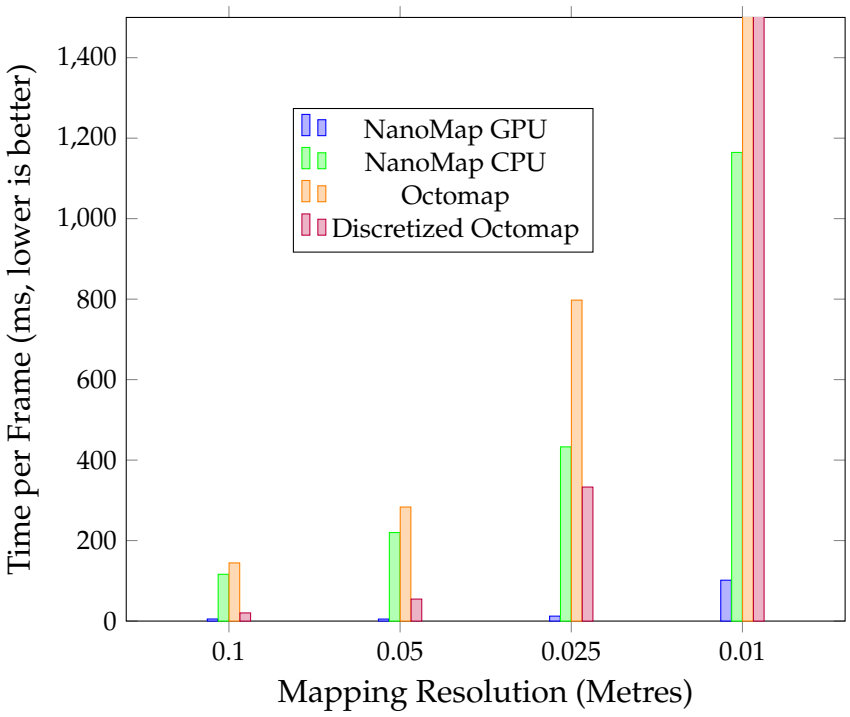


Figure 5. Asus Zephyrus G14 NanoMap GPU vs. CPU performance (see Figure 6).

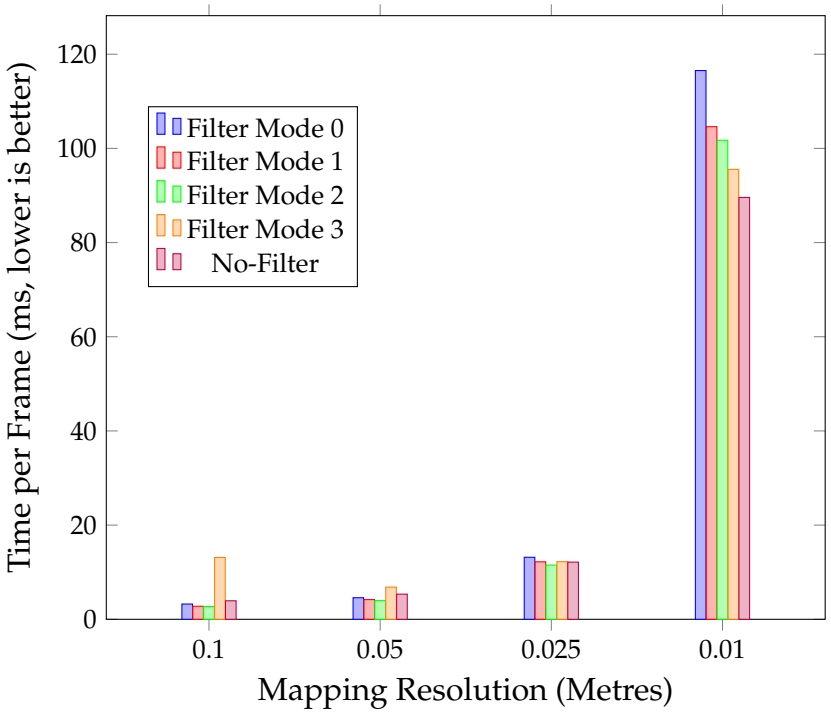


Figure 6. Asus Zephyrus G14 NanoMap GPU performance.

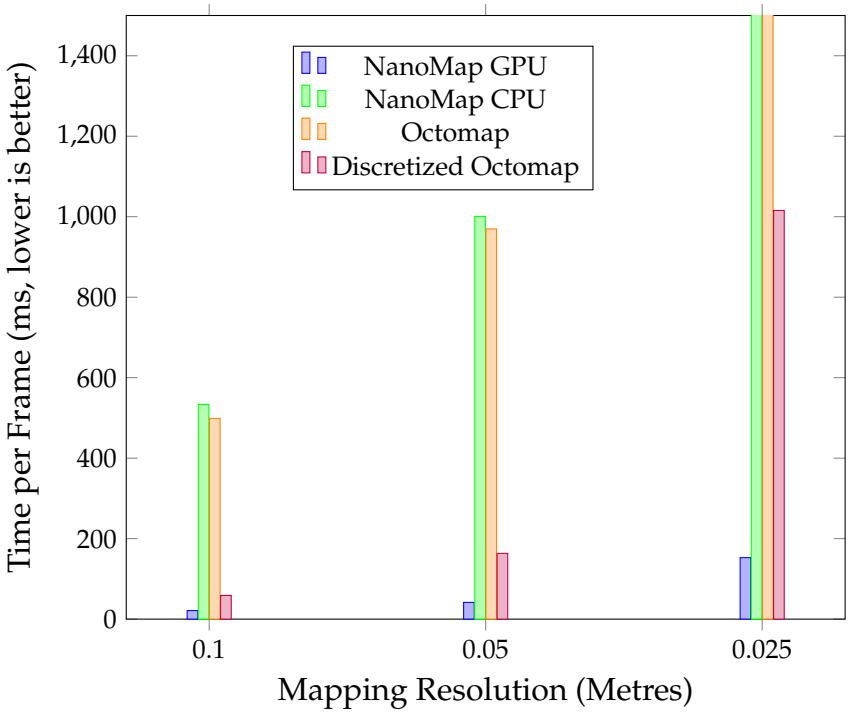


Figure 7. Nvidia Jetson-Nano NanoMap GPU vs. CPU performance (see Figure 8).

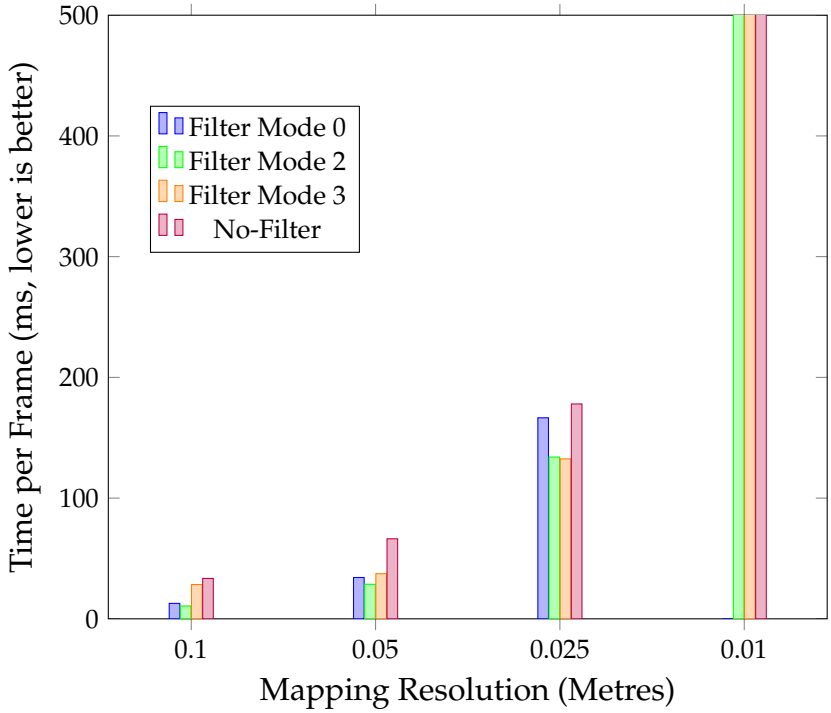


Figure 8. Nvidia Jetson-Nano NanoMap GPU performance.

Table 4. Freiburg dataset average frame time for Asus G14 (ms, lower is better).

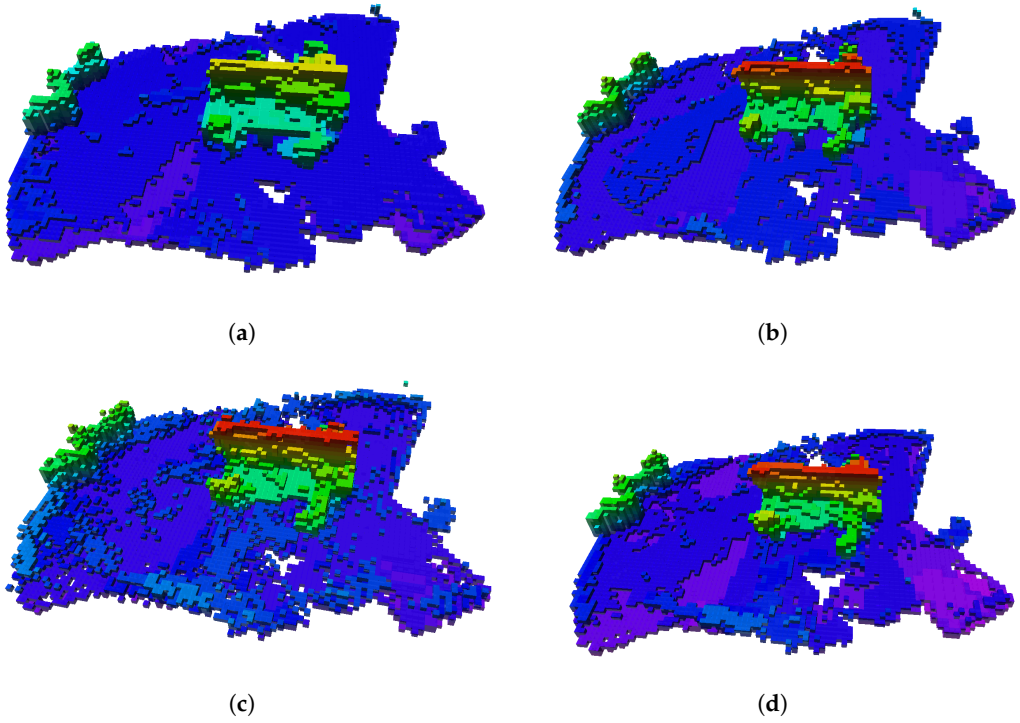
Method	Mapping Resolution (Metres)			
	0.1	**0.05**	**0.025**	**0.01**
NM Filter Mode 0	3.24	4.58	13.17	116.53
NM Filter Mode 1	2.77	4.18	12.21	104.61
NM Filter Mode 2	**2.70**	**3.96**	**11.52**	101.73
NM Filter Mode 3	13.12	6.83	12.24	95.54
NM No Filter	3.93	5.34	12.14	**89.58**
NM CPU Only	116.25	219.93	432.90	1164.55
OctoMap	144.51	283.38	797.57	4983.70
OctoMap Discretized	20.25	54.65	332.99	4559.49

Bold values indicate the lowest average frame time for a given resolution.

Table 5. Freiburg dataset average frame time for Jetson Nano (ms, lower is better).

Method	Mapping Resolution (Metres)			
	0.1	**0.05**	**0.025**	**0.01**
NM Filter Mode 0	12.79	34.23	166.52	NA
NM Filter Mode 1	NA	NA	NA	NA
NM Filter Mode 2	**10.62**	**28.61**	134.00	1575.44
NM Filter Mode 3	28.39	37.42	**132.50**	1438.03
NM No Filter	33.47	66.28	177.98	**1225.86**
NM CPU Only	533.74	1000.86	1955.14	NA
OctoMap	498.77	969.68	2465.05	NA
OctoMap Discretized	58.87	163.42	1015.57	NA

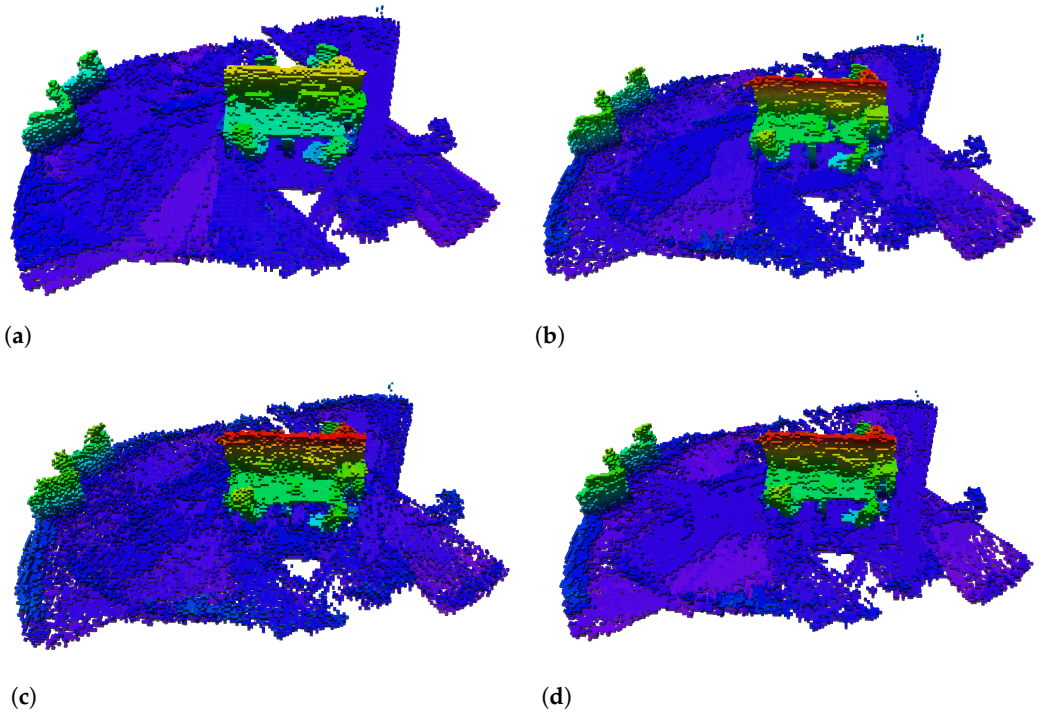
Bold values indicate the lowest average frame time for a given resolution.



(**a**) (**b**)

(**c**) (**d**)

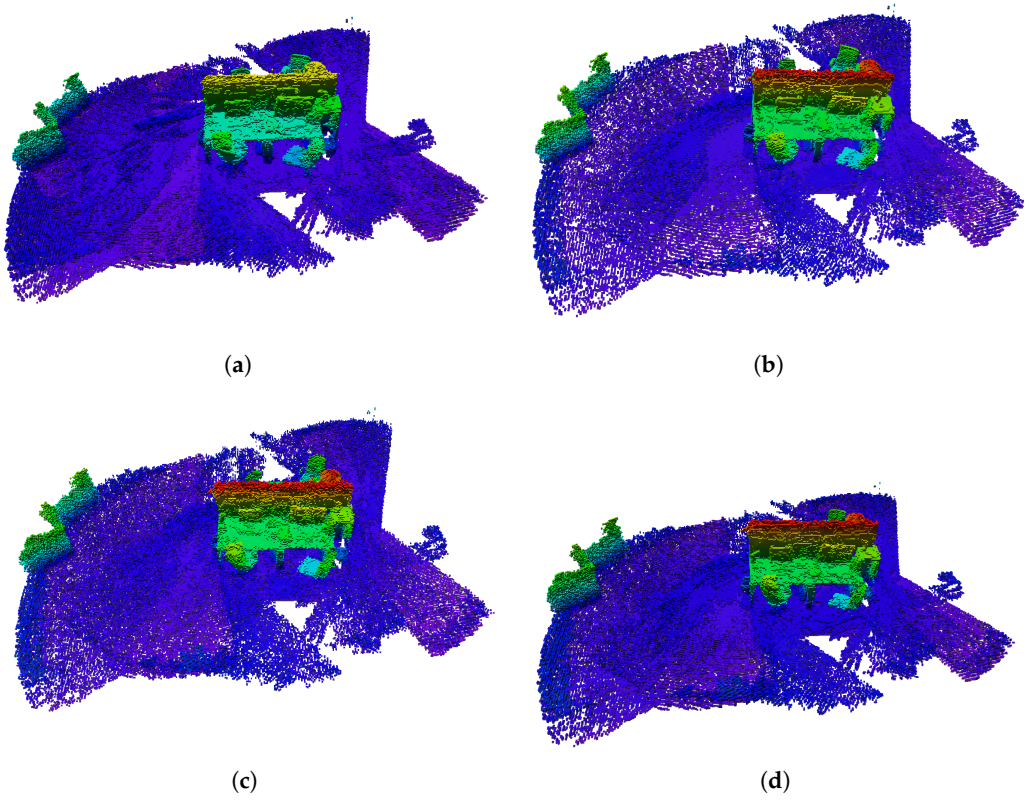
Figure 9. Benchmark outputs at 0.1 m grid resolution. (**a**) OctoMap. (**b**) Nanomap No Filter. (**c**) Nanomap Filter Modes 0 and 1. (**d**) Nanomap Filter Modes 2 and 3.



(**a**) (**b**)

(**c**) (**d**)

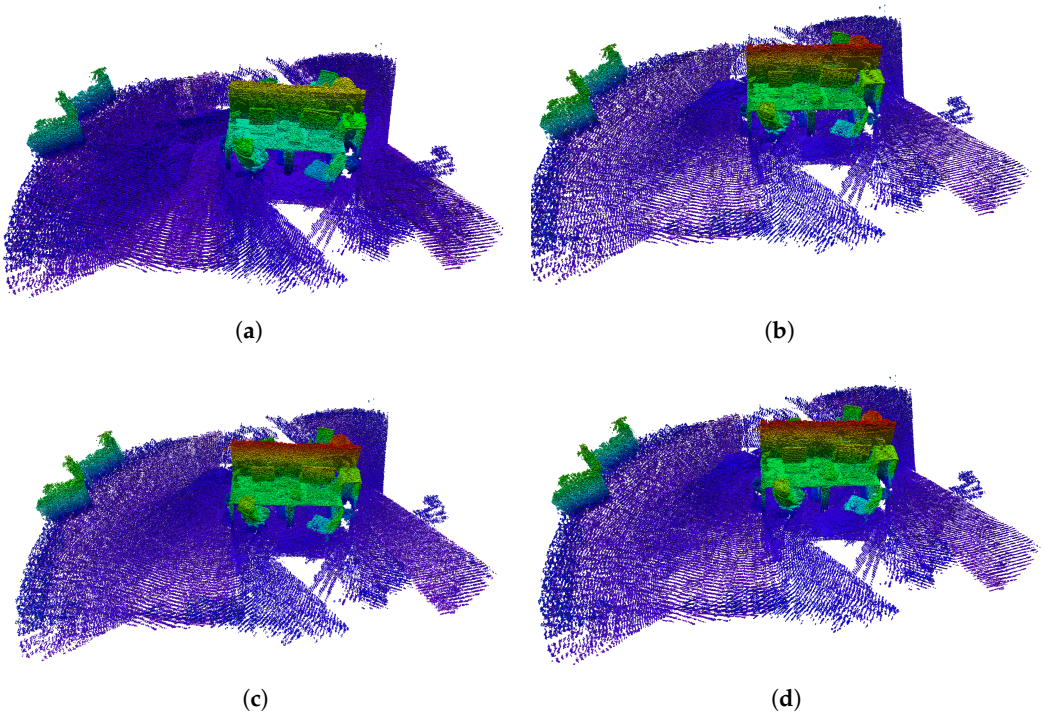
Figure 10. Benchmark outputs at 0.05 m grid resolution. (**a**) OctoMap. (**b**) Nanomap No Filter. (**c**) Nanomap Filter Modes 0 and 1. (**d**) Nanomap Filter Modes 2 and 3.



(**a**) (**b**)

(**c**) (**d**)

Figure 11. Benchmark outputs at 0.025 m grid resolution. (**a**) OctoMap. (**b**) Nanomap No Filter. (**c**) Nanomap Filter Modes 0 and 1. (**d**) Nanomap Filter Modes 2 and 3.



(**a**) (**b**)

(**c**) (**d**)

Figure 12. Benchmark outputs at 0.01 m grid resolution. (**a**) OctoMap. (**b**) Nanomap No Filter. (**c**) Nanomap Filter Modes 0 and 1. (**d**) Nanomap Filter Modes 2 and 3.

As can be seen from the average-frame-time tests for both the laptop and Jetson Nano tests, by leveraging the parallel capacity of the available GPUs, Nanomap is able to provide significant speedups compared to non-GPU-based mapping methods. Use of the OpenVDB data structure in the CPU NanoMap case also improves performance over the non-filtered Octomap test on the Asus G14, where single-threaded performance is strong, further supporting the results of [10]. In the case of the Jetson Nano, the CPU NanoMap algorithm only outperforms the non-filtered Octomap solution at higher resolutions. This is because the Octomap algorithm uses multi-threading to improve performance; however, as the resolution increases, the structure of the Octomap makes accessing voxels take considerably longer. With the OpenVDB data structure and its fixed depth, the access times for any given voxel do not increase with resolution; rather, there are just more voxels to access.

Further performance is available by utilising the provided GPU voxel filters for the majority of resolutions. At very fine resolutions, the voxel filters do not shrink the ray cloud enough to offset the overhead of the filter, which can be seen by the non-filtered benchmark outperforming the filtered benchmark at a grid resolution of 0.01 m (Figure 6 and Table 5). Additionally, at such a resolution the Octomap method appears to suffer severe bottle-necking, with the frame time ballooning out to over 4 seconds per frame.

One interesting thing to note is the large performance penalty at the 0.1 m resolution for Filter Mode 3. This is caused by the data structure used to store the voxel filter counts. Rather than storing each count in its own 32-bit integer-like Filter Mode 2, each count is an 8-bit unsigned integer stored within a larger 32 bit integer. This means that when trying to increment the counts of cells that are neighbours, four times as much blocking occurs during atomicAdd() operations; this is a result of atomicAdd() being performed at the 32 bit level for most CUDA architectures. As the resolution becomes higher, the performance deficit decreases as there are fewer conflicts that act as bottlenecks for the GPU. This mode is still useful on lower-powered GPUs by enabling discrete filtering with as small a memory cost as possible (Tables 3 and 5). On GPUs with more power, the filtering is less necessary, and the memory cost can be avoided altogether (Table 4).

Table 5 shows that filtered NanoMap methods are capable of maintaining an average frame time of 30 ms at a grid resolution of 0.05 m per voxel on the low-power single board Jetson Nano computer. This is an exceptional result, showing that even at a 0.05 m resolution, a Jetson Nano using NanoMap would be capable of processing the input from a 30 Hz sensor with time to spare, provided it has similar characteristics to the sensor used in the TUM dataset. This provides a vast improvement over the currently available methods for real-time map creation on mobile robotics platforms.

3.2. Laser Based Testing

For testing the Laser sensor style performance, the CPU-only method provided with NanoMap was compared to the Octomap and Octomap Discrete methods. A simulated laser dataset using the sensor properties of a VLP16 LIDAR was created using the NanoMap simulation capabilities. This dataset was stored as a rosbag for convenient testing. The benchmark was run using three different maximums for the input sensor range. The dataset remained the same, but a maximum range value was used to clip the effective range of the sensor to test performance. Tables 6 and 7 show the results of the benchmarking.

The results validate the results of [10] in their assessment of the performance improvements provided by the OpenVDB data structure when compared to the Octree used by Octomap. The NanoMap CPU-only approach maintains consistent access speeds even as the resolution and range of the sensor are increased on both testing platforms. While the Octomap solution benefits from threading the OpenVDB-based solution offers significant improvements to performance. No doubt as a result of the increased operations performance provided by the fixed search depth and spatially aware operations provided by OpenVDB. Further improvements to the NanoMap CPU algorithm via CPU threading such as those found in Octomap are to be explored in future work.

Table 6. Laser dataset average frame time for Jetson Nano (ms, lower is better).

	Mapping Resolution (Metres)		
	5 m Sensor Range		
Method	**0.10**	**0.05**	**0.025**
NanoMap CPU Only	7.39	16.21	44.84
OctoMap	38.39	184.99	912.20
OctoMap Discretized	37.41	206.68	1162.58
	10 m Sensor Range		
Method	**0.10**	**0.05**	**0.025**
NanoMap CPU Only	16.36	41.57	122.77
OctoMap	79.22	352.65	1667.03
OctoMap Discretized	84.47	409.80	2313.37
	20 m Sensor Range		
Method	**0.10**	**0.05**	**0.025**
NanoMap CPU Only	20.69	55.06	157.59
OctoMap	97.27	421.47	1753.41
OctoMap Discretized	101.11	505.42	2455.97

Table 7. Laser dataset average frame time for Asus G14 (ms, lower is better).

	Mapping Resolution (Metres)		
	5 m Sensor Range		
Method	**0.10**	**0.05**	**0.025**
NanoMap CPU Only	32.39	72.14	190.76
OctoMap	129.37	611.65	2594.81
OctoMap Discretized	129.16	685.85	3112.92
	10 m Sensor Range		
Method	**0.10**	**0.05**	**0.025**
NanoMap CPU Only	72.55	180.72	530.90
OctoMap	263.68	1057.82	4774.17
OctoMap Discretized	276.95	1191.06	6189.28
	20 m Sensor Range		
Method	**0.10**	**0.05**	**0.025**
NanoMap CPU Only	90.27	232.80	681.05
OctoMap	315.14	1165.95	5184.32
OctoMap Discretized	340.61	1328.84	7011.06

4. Conclusions

This work provides the C++ software library NanoMap (https://github.com/ViWalkerDev/NanoMap) and the necessary ROS packages required to create OpenVDB grids using frustum and laser-based sensors on GPU-enabled systems. Evaluation shows that by utilising a GPU, the ray casting of a point cloud for use in occupancy mapping can be rapidly accelerated in comparison to current CPU-only approaches, even on limited single-board GPU-enabled computers. On a Jetson Nano, the approach outperforms Octomap by a factor of 5.5 in the provided frustum-based test case at 0.1 m mapping resolution and a factor of 7.6 at the 0.025 m mapping resolution. Even in the case of CPU-only operation, when handling sparse point cloud inputs such as those provided by LIDAR sensors, NanoMap outperforms both the non-filtered and filtered OctoMap methods.

With access to a more powerful GPU on the Asus G14 test platform, the difference balloons to a factor of 7.5 at 0.1 m mapping resolution and 50.8 times at the 0.01 m mapping resolution.

The library also shows exceptional performance for the simulation and construction of occupancy maps on an Asus G14 laptop, making it useful for reinforcement learning tasks and validation of tasks involving occupancy mapping.

Future work aims to expand the capabilities of the library and provide additional functions utilising the GPU for processing and analysing occupancy maps in real time for use with planning and control tasks and other robotics applications.

Author Contributions: Conceptualization, V.W., F.G. and F.V.; Methodology, V.W., F.G. and F.V.; Software, V.W.; Supervision, F.G. and F.V.; Validation, V.W.; Visualization, V.W.; Writing—original draft, V.W.; Writing—review and editing, F.G. and F.V. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

2D	2-Dimensional
3D	3-Dimensional
ROS1	Robotic Operation System
ROS2	Robotic Operation System 2
RVIZ2	ROS2 Visualisation
CPU	Central Processing Unit
GPU	Graphics Processing Unit
CUDA	Compute Unified Device Architecture
NM	NanoMap
RGB-D	Red–Green–Blue-Depth
LIDAR	Light Detection and Ranging

References

1. Collins, T.; Collins, J.; Ryan, D. Occupancy grid mapping: An empirical evaluation. In Proceedings of the 2007 Mediterranean Conference on Control & Automation, Athens, Greece, 27–29 June 2007; pp. 1–6. [CrossRef]
2. Vanegas, F.; Gonzalez, F. Enabling UAV navigation with sensor and environmental uncertainty in cluttered and GPS-denied environments. *Sensors* **2016**, *16*, 666. [CrossRef] [PubMed]
3. Buerkle, C.; Oboril, F.; Jarquin, J.; Scholl, K.U. Efficient dynamic occupancy grid mapping using non-uniform cell representation. In Proceedings of the 2020 IEEE Intelligent Vehicles Symposium (IV), Las Vegas, NV, USA, 19 October–13 November 2020; IEEE: Piscataway, NJ, USA, 2020; pp. 1629–1634.
4. Sandino, J.; Vanegas, F.; Maire, F.; Caccetta, P.; Sanderson, C.; Gonzalez, F. UAV framework for autonomous onboard navigation and people/object detection in cluttered indoor environments. *Remote Sens.* **2020**, *12*, 3386. [CrossRef]
5. Serna, J.G.; Vanegas, F.; Gonzalez, F.; Flannery, D. A review of current approaches for UAV autonomous mission planning for Mars biosignatures detection. In Proceedings of the 2020 IEEE Aerospace Conference, Big Sky, MT, USA, 7–14 March 2020; IEEE: Piscataway, NJ, USA, 2020; pp. 1–15.
6. Niemand, J.; Mathew, S.J.; Gonzalez, F. Design and testing of recycled 3D printed foldable unmanned aerial vehicle for remote sensing. In Proceedings of the 2020 International Conference on Unmanned Aircraft Systems (ICUAS), Athens, Greece, 1–4 September 2020; IEEE: Piscataway, NJ, USA, 2020; pp. 892–901.
7. Mandel, N.; Sandino, J.; Galvez-Serna, J.; Vanegas, F.; Milford, M.; Gonzalez, F. Resolution-adaptive quadtrees for semantic segmentation mapping in UAV applications. In Proceedings of the 2022 IEEE Aerospace Conference (AERO), Big Sky, MT, USA, 5–12 March 2022; IEEE: Piscataway, NJ, USA, 2022; pp. 1–17.
8. Hornung, A.; Wurm, K.M.; Bennewitz, M.; Stachniss, C.; Burgard, W. OctoMap: An efficient probabilistic 3D mapping framework based on octrees. *Auton. Robot.* **2013**, *34*, 189–206. [CrossRef]
9. De Gregorio, D.; Di Stefano, L. Skimap: An efficient mapping framework for robot navigation. In Proceedings of the 2017 IEEE International Conference on Robotics and Automation (ICRA), Singapore, 29 May–3 June 2017; IEEE: Piscataway, NJ, USA, 2017; pp. 2569–2576.
10. Besselmann, M.G.; Puck, L.; Steffen, L.; Roennau, A.; Dillmann, R. VDB-Mapping: A High Resolution and Real-Time Capable 3D Mapping Framework for Versatile Mobile Robots. In Proceedings of the 2021 IEEE 17th International Conference on Automation Science and Engineering (CASE), Lyon, France, 23–27 August 2021; IEEE: Piscataway, NJ, USA, 2021; pp. 448–454.

11. Macenski, S.; Tsai, D.; Feinberg, M. Spatio-temporal voxel layer: A view on robot perception for the dynamic world. *Int. J. Adv. Robot. Syst.* **2020**, *17*, 1729881420910530. [CrossRef]
12. Jia, T.; Yang, E.Y.; Hsiao, Y.S.; Cruz, J.; Brooks, D.; Wei, G.Y.; Reddi, V.J. OMU: A probabilistic 3D occupancy mapping accelerator for real-time OctoMap at the edge. *arXiv* **2022**, arXiv:2205.03325.
13. Museth, K.; Lait, J.; Johanson, J.; Budsberg, J.; Henderson, R.; Alden, M.; Cucka, P.; Hill, D.; Pearce, A. OpenVDB: An open-source data structure and toolkit for high-resolution volumes. In *Acm Siggraph 2013 Courses, Proceedings of the SIGGRAPH'13: Special Interest Group on Computer Graphics and Interactive Techniques Conference, Anaheim, CA, USA, 21–25 July 2013*; Association for Computing Machinery: New York, NY, USA, 2013; p. 1.
14. Museth, K. NanoVDB: A GPU-friendly and portable VDB data structure for real-time rendering and simulation. In *ACM SIGGRAPH 2021 Talks, Proceedings of the SIGGRAPH'21: Special Interest Group on Computer Graphics and Interactive Techniques Conference, Virtual Event, 9–13 August 2021*; Association for Computing Machinery: New York, NY, USA, 2021; pp. 1–2.
15. Sanders, J.; Kandrot, E. *CUDA by Example: An Introduction to General-Purpose GPU Programming*; Addison-Wesley Professional: Boston, MA, USA, 2010.