2024 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)
October 14-18, 2024. Abu Dhabi, UAE

FRAGG-Map: Frustum Accelerated GPU-Based Grid Map

Michele Grimaldi[1,2], Narcis Palomeras[1], Ignacio Carlucho[2], Yvan R. Petillot[2] and Pere Ridao Rodriguez[1]

***Abstract*— In robotics, occupancy grids serve as required repositories of information about the environment in numerous applications. One such critical application is Simultaneous Localization and Mapping (SLAM), where robots dynamically scan and explore their surroundings while in motion. In the context of extended-duration missions, it becomes imperative to confront the complexities linked to the expansion of occupancy grids as well as handling loop closure detection. These challenges primarily revolve around two key aspects: enabling the seamless expansion of the map on multiple occasions, thus avoiding the need to map smaller regions in numerous separate missions, and ensuring real-time updates to the map to sustain the robot's knowledge base and enhance its responsiveness. To address these challenges, we introduce an innovative map called Frustum Accelerated GPU-Based Grid Map (FRAGG-Map). This map adopts a highly parallelizable 3D grid structure and leverages the power of CUDA kernels to facilitate efficient insertion of point-clouds and enables real-time updates of the map. FRAGG-Map identifies the portions of the map that require updates and utilises the GPU to update them, significantly enhancing computational performance. Our results show that FRAGG-Map can run 31 times faster than OctoMap, significantly outperforming state-of-the-art methods.**

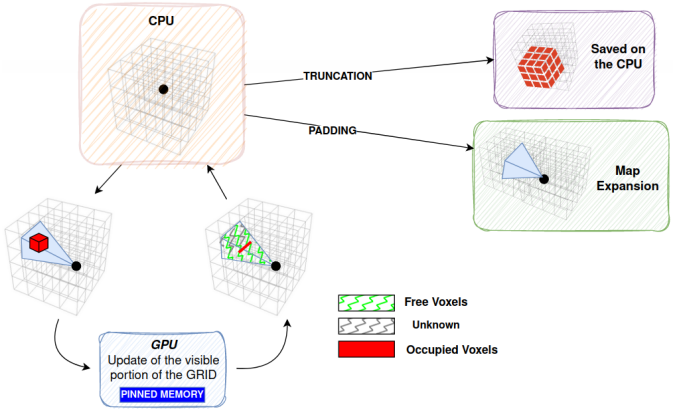


Fig. 1: FRAGG-Map overview. An initial grid is instantiated in the CPU. Visible portions of the map are uploaded to the GPU and updated in real-time. The generated grid can be expanded, if the robot is close to the boundaries, or it can be truncated and fused at any point, allowing for maximum flexibility.

I. Introduction

An important feature of robotics is the construction of maps faithful to reality from the observations of the robot and the estimation of its position. These maps can be represented using an occupancy grid [1]. This grid divides the space into voxels of a given resolution and a probability of occupancy is assigned to each voxel. The occupancy grids allow a robot to determine which areas are free and which are occupied and therefore plan safe trajectories. To be able to plan in real-time, it is necessary to work on a single global map which encompasses everything that has been explored. Currently, many of the real applications of SLAM with occupancy maps use particle filters to estimate the robot's position, where each particle builds its own map [2] [3]. This algorithm implies having many copies of the map, as many useful particles are computed by the filter. For maps of considerable sizes, this implies a significant computational cost.

Other SLAM solutions are based on the relationships between different positions of the robot in which useful measures of the environment have been obtained, such as point-clouds of objects obtained from a laser scanner [4]. By adding information on the positions and measurements, a single global map can be built, eliminating the cost of copying entire maps. Finally, the SLAM solutions need a way to correct the global map data when the position of a previous measurement is updated (for example after loop closure) in a fast way.

In this work, we introduce FRAGG-Map, an accelerated grid map that takes advantage of the field of view of the sensor used to perceive the environment to work only on the portion of the visible portion of the map. The main motivation of this work is the development of a lightweight algorithm able to provide large maps that can be updated in real-time. To achieve this goal we make use of GPU architectures, which provide fast and parallel computing capabilities, that are ideally suited to modifying multiple elements of a map concurrently. Figure 1 provides an overview of the system. By using a dual GPU and CPU architecture and the field of view of the mapping sensor, FRAGG-Map can insert and remove huge point-clouds allowing fast map updates. Furthermore, due to the grid nature of the map, part of the grid can be easily truncated and stored to allow the robot to work with the part of the map in which the robot is sensed to work and load the stored parts if needed (i.e. long-term deployment scenarios). We conducted extensive experimental evaluations using the Freiburg Campus dataset [5] as well as the data collected using an underwater laser scanner [4]. Our results show that FRAGG-Map can insert point-clouds 31 times faster than OctoMap while maintaining a similar quality and

This work was supported by the European Union's Horizon 2020 Research and Innovation Program through the ATLANTIS "The Atlantic Testing Platform for Maritime Robotics: New Frontiers for Inspection and Maintenance of Offshore Energy Infrastructures" Project under Grant 871571

[1] Computer Vision and Robotics Institute, University of Girona, Girona, Spain

[2] School of Engineering & Physical Sciences, Heriot-Watt University, Edinburgh, UK m.grimaldi@hw.ac.uk

memory footprint. A qualitative comparison of the methods can be seen in Figure 2. The main contributions of the work are as follows:

- We introduce a GPU-accelerated map coupled with the frustum view to update only the visible voxels.
- We present the functionalities of the FRAGG-Map library and we will release it as open source.[1]
- We provide extensive evaluations that show that our method can significantly outperform state-of-the-art methods such as Octomap [6].

II. Related Work

Occupancy grid mapping (OGM) encompasses several frameworks that aim to address the challenges of efficient mapping and navigation in robotic systems. OctoMap [6] is a widely adopted OGM framework that employs an OcTree data structure to represent occupancy information. It offers a flexible resolution representation, allowing for adaptive map resolution based on the complexity of the environment. OctoMap provides efficient updates, traversability analysis, and collision-checking capabilities. However, its CPU-based implementation may struggle to handle the high data rates generated by modern 3D lidar sensors and maintain rich voxel representations in real-time. Another notable framework in the field of OGM is UFOMap [7] which is an extension of the OctoMap. UFOMap stands out by utilizing an explicit representation of all three states on the map: occupied, free, and unknown. By explicitly representing each state, UFOMap achieves efficient memory utilization while preserving the necessary information about the environment. UFOMap provides methods that enable significantly faster insertions into the octree data structure, facilitating real-time coloured volumetric mapping at high resolutions.

Occupancy Homogenous Mapping (OHM) [8] is an open-source GPU-based OGM framework that addresses the computational limitations of CPU-based implementations. The regions are stored in a hash map where each region is a dense 3D grid of cubic, fixed-size voxels allocated as a contiguous memory block. It supports modern OGM algorithms such as Normal Distributions Transform-Occupancy Maps (NDT-OM), Normal Distributions Transform-Traversability Maps (NDT-TM), decay-rate, and Truncated Sign Distance Function (TSDF). In [9] the authors present a mapping framework for robot navigation which features a multi-level querying system capable of obtaining rapid representations as diverse as a 3D voxel grid, a 2.5D height map and a 2D occupancy grid. These are inherently embedded into a memory and time-efficient core data structure organized as a Tree of SkipLists. Compared to the OctoMap approach it exhibits a better time efficiency, thanks to its simple and highly parallelizable computational structure, and has a similar memory footprint when mapping large workspaces. GPU-Voxels [10], another notable framework, harnesses GPU acceleration. Its target is fast collision avoidance which is handled utilizing GPU parallel computation and a fixed size map. The mapping framework Voxblox [11] as well as the GPU optimized version Nvblox[2] uses a signed distance field [12] voxel grid, with voxel hashing for dynamic growth, as representation. It was mainly developed for planning or trajectory optimizations in the context of micro aerial vehicles (MAVs). The signed distance field representation makes trajectory optimizations faster by storing the distance to the closest obstacle in each voxel. In the recent OctoMapRT [13] the authors propose to use a hybrid of off-the-shelf ray-tracing GPUs and CPUs to substantially improve OctoMap CPU-version. OctoMap-RT employs massively parallel ray-shooting using GPUs to generate occupied and free voxel grids and to update their occupancy states in parallel, and it exploits CPUs to restructure the OGM using the updated voxels. Unfortunately, the implementation is not publicly accessible, preventing direct comparison.

The aforementioned OGM algorithms face a persistent challenge: as the map size increases, the time needed for updates escalates accordingly. This issue is particularly pronounced with larger maps and accentuated for the Tree-based solutions. Consequently, there comes a point where rebuilding the map becomes necessary to accommodate new data, potentially leading to the loss of the previous map. Furthermore, there is another limitation to consider, as some of these maps are only expandable up to a certain size. For instance, Octomap and SkiMap can cover areas up to $2^{16} \times 0.001$ m, totalling 65.536 square meters each, while UFOMap is limited to $2^{21} \times 0.001$ m, amounting to 2097.152 square meters in each dimension. These expansions are constrained by a 1 mm voxel size and maximum octree depths of 16 and 21, respectively.

III. Method description

Addressing the need for real-time map data correction and long-term exploration in autonomous robots with limited computing capability, we propose the development of an algorithm for fast and efficient single-map construction and maintenance. Utilizing the parallel processing capabilities of GPUs, known for their efficiency in parallel computation, we aim to optimize mapping processes for rapid deployment and resource conservation.

A. Grid Structure

We undertook the development of a 3D grid utilizing CUDA programming. Given the grid's nature, similar to a linear array, most functions can be parallelized, making them easier to convert into CUDA kernels. In practice, the entire grid resides in the CPU, while a portion of the grid is loaded into the GPU. The structure of the grid includes the resolution, an initial number of cells N, and a list of cells that are loaded in the GPU. The grid cells contain two values:

- n_{points}: a short integer for the number of points added to the cell (which can be useful for some statistics)
- $occupancy$: a $\_\_half$, which is a single 16-bit floating point quantity/type specific to the CUDA library, for the occupancy probability.

[1] https://github.com/Michele1996/FRAGG-Map

[2] https://github.com/nvidia-isaac/nvblox

(a) (b) (c) (d) (e)

Fig. 2: a) Map generated with the Dead Reckoning b) Map generated with the SLAM and not corrected c) Map generated with Octomap d) Map generated with our method f) The pipe structure.

Thus each cell has a size of 4 bytes. The grid has an initial size and fixed resolution and the robot is placed at its centre.

B. Dynamic Map - Frustum

Frustum culling is a technique used in computer graphics to determine whether an object (or a portion of it) is visible within the viewing frustum of a camera [14]. The viewing frustum is the space region visible in the sensor view. If an object is outside the frustum, it can be culled (discarded) from rendering. Using viewing frustum to limit spatial perception range has been widely used in Computer Vision and SLAM, with or without GPU acceleration [15] [16] [17] [18]. We use the sensor frustum to load into the GPU the portion of the grid visible from the equipped sensor. This allows us to rapidly update the map via the GPU. In the case of the grid, we use the sensor Field of View (FOV) to determine the visible cells and we load them in the GPU to update them using the acquired point-cloud. In the presence of obstacles, the frustum encompasses the 3D space behind the obstacles but the point-clouds will not contain any points within that area. One advantageous aspect of the frustum is that it can be utilized to determine if a space is free and safe for navigation, even if no points are detected. By knowing the visible cells within the frustum and the sensor's range, the system can update the occupancy probability similarly to a point-could insertion. Rejection planes, forming the boundaries of the frustum, play the role of filter in this process, defining the limits of what is considered visible. These planes are computed based on sensor parameters, including their position, orientation, and field of view. A similar approach has also been explored in [19] for path planning. The number of voxels contained in a pyramidal frustum and consequently, the relative memory occupied in the GPU can be expressed as follows:

$$N_{voxels} = \frac{V_{frustum}}{R^3} \tag{1}$$

$$V_{frustum} = \frac{D}{3}\left(S1 + S2 + \sqrt{S1 \times S2}\right) \tag{2}$$

$$S1 = \frac{\pi}{4}\left(\tan\left(\frac{FOV}{2}\right) \times x\right)^2 \tag{3}$$

$$S2 = \frac{\pi}{4}\left(\tan\left(\frac{FOV}{2}\right) \times (x + D)\right)^2 \tag{4}$$

where N_{voxels} is the number of voxels in the frustum, $V_{frustum}$ is the frustum's volume (pyramid shape), R is the resolution of the map, $S1$ is the area of the near base, $S2$ is the area of the far base, D is the maximum depth of the sensor, FOV is the field of view, x is the distance from the sensor to the near end of the frustum, and $x + D$ is the distance to the far end. In our case the near base coincides with the sensor origin, thus $x = 0$.

C. Optimizations

The primary function of the view frustum is to determine which portion of data should be processed on the GPU. Effective achievement of this task hinges on maintaining coherence across both spatial and temporal dimensions to prevent unnecessary data transfer, particularly in scenarios where frustums overlap. We employ several optimization methods to refine the portion of data processed on the GPU.

1) Spatial Coherence - Plane Rejection Memory: We use the optimization principles introduced by [20], exploiting the observation that when a cell is rejected by a specific plane—especially during slow camera movements—the same plane is likely to reject objects in subsequent frames. This spatial-temporal coherence is complemented by the observation that the field of view (FOV) transitions smoothly from one point cloud to another. To further enhance efficiency, priority is given to testing against the plane responsible for rejection in the previous frame, with each cell retaining information about the last plane that resulted in its exclusion.

2) Rotation Coherence: If a cell is rejected by the left plane, and the camera slightly rotates to the right, the cell will once again be outside the frustum. However, it is essential to exercise caution when using this method. Continuous rotation of the camera may eventually bring the cell back inside the view frustum. To address this, we use a timestamp and retain the record of the plane's rejection only for a short period.

3) Translation Coherence: When a cell is rejected by the near plane (i.e., it is positioned behind the view frustum), and the camera translates forward, the cell will undoubtedly remain outside the view frustum. It is worth noting that even if in the case of a 360° LiDAR we cannot optimize the frustum using these techniques, the frustum is already efficient since at time t most of the visible cells will be already in the GPU at time $t + 1$.

D. Map and Occupancy Updated

At the beginning of the mapping, the robot is at the centre of the 3D grid and the cells have an occupancy probability equal to 0.5. We distinguish unknown, free, and occupied

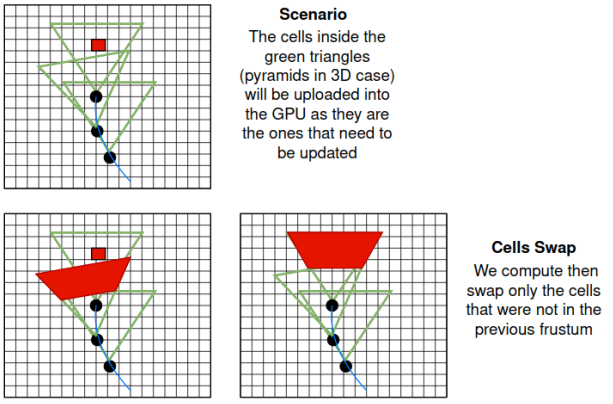


Fig. 3: View of the grid's portion uploaded into the GPU.

cells:

$$Cell_i = \begin{cases} Unknown & \text{if } 0.45 \leq p_i \leq 0.55 \\ Occupied & \text{if } p_i > 0.55 \\ Free & \text{otherwise} \end{cases} \quad (5)$$

To update cell probability p_i we employ a specialized CUDA kernel that implements Bresenham's algorithm [21] for ray-tracing. The CUDA kernel utilizes the visible cells within the frustum, along with the sensor's position, orientation, and sensor probability. By using atomic operations, the kernel updates the occupancy probability of cells ensuring consistency along the ray traversal. The probability update is computed using the standard log odds:

$$logOdds_{factor} = log(s_{prob}/(1 - s_{prob})) \quad (6)$$

$$logOdds_{next} = logOdds_{prior} + logOdds_{factor} \quad (7)$$

where, s_{prob} represents the sensor's measurement probability.

As the robot begins mapping, it continuously uses the sensor's position to compute frustums representing the field of view of the sensors. These frustums are projected onto the 3D grid, and only the cells intersecting with the frustum are considered for further processing. When the robot ventures closer to the grid's boundaries, the algorithm employs a dynamic grid resizing strategy. To accommodate new regions of interest, the grid is padded in one or more directions based on the robot's mapping trajectory. This padding mechanism ensures that the robot can seamlessly explore beyond the original boundaries without compromising the mapping process. In this way, the system can overcome the limits of OctoMap, SkiMap and UFOMap in terms of covered space. At each insertion, the probability of occupancy probabilities is then clamped using the clamping update policy proposed in [22], which is also employed in OctoMap. The system pseudo code algorithm is presented in Algorithm 1.

1) Vast Map Handling - Loop Closure: Compared to other methods using the OcTree structure, our system benefits from an easier expansion mechanism. We also have the possibility of removing the point-clouds and reinserting them. For this, in the CPU we store a map containing each point-cloud along with the robot's pose from which it was inserted. During the deletion phase, the point-clouds are removed using the inverse approach to insertion. The process involves computing the frustum based on the robot's pose rp_i at the time of insertion, loading the cells visible during that time for the specific point-cloud p_i, and applying ray-tracing with reversed log-odds function. Subsequently, the same point-cloud p_i is reinserted using another robot's pose rp_j. This is particularly interesting when handling loop closures since in the case of huge maps, for OcTree-based maps, it is often required to rebuild the map instead of updating the existing one due to the update time.

Algorithm 1: FRAGG-MAP Insertion Cycle

```
Data: 3DGrid, sensorPose, sensorFOV, logOddsNext
Function FrustumKernel(sensorPose, sensorFOV)
    visibleCellsIndex ← Empty List;
    for each cellPosition in 3DGrid do
        if cellPosition is in Frustum(sensorPose, sensorFOV) then
            add cellPosition to visibleCellsIndex;
    return visibleCellsIndex;
Function Swap(gpuNodes, currentIndexes, nextIndexes)
    copyValuesFromGPU(currentIndexes);
    copyValuesToGPU(gpuNodes[nextIndexes]);
    currentIndexes ← nextIndexes;
Function InsertPointCloudKernel(pointCloud, sensorPose, logOddsNext)
    for each point in pointCloud do
        if point is valid then
            apply Bresenham(point, sensorPose, logOddsNext);
Function Update()
    nextIndexes ← FrustumKernel(sensorPose, sensorFOV);
    Swap(gpuNodes, currentIndexes, nextIndexes);
    InsertPointCloudKernel(pointCloud, sensorPose, logOddsNext);
```

E. Memory management

Using the GPU to execute algorithms improves their speed but at the cost of being slower when exchanging data between the CPU and the GPU. Since the peak bandwidth between the device memory and the GPU is much higher than the peak bandwidth between host memory and device memory, the implementation of data transfers between the host and GPU devices can make a real difference in terms of performance.

1) Pinned memory: One way to speed up the data transfer is to use the Pinned Memory. Host (CPU) data allocations are pageable by default. The GPU cannot access data directly from pageable host memory, so when a data transfer from pageable host memory to device memory is invoked, the CUDA driver must first allocate a temporary page-locked, or "pinned", host array, copy the host data to the pinned array, and then transfer the data from the pinned array to the device memory. We can avoid the cost of the transfer between pageable and pinned host arrays by directly allocating our host arrays to pinned memory.

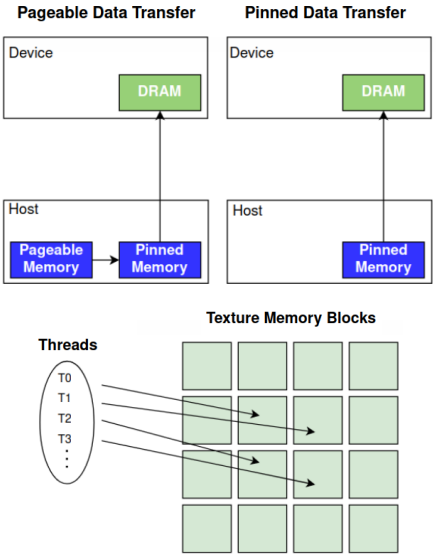


Fig. 4: Top: Memory transfer using Pinned memory. Bottom: Texture memory functioning.

2) Texture memory: Texture memory, initially designed for graphics applications, serves as a read-only memory variant enhancing performance and reducing memory traffic with specific access patterns. While traditionally used in graphics, it proves effective in certain GPU computing applications. Cached on-chip, it boosts effective bandwidth by minimizing off-chip DRAM requests. Ideal for spatially localized memory access patterns, it is employed in our system for point-clouds. In this read-only space, a texture fetch costs one device memory read on a cache miss, otherwise, just a read from the texture cache.

IV. Framework Functionalities

In this section, we present several key features that contribute to the system's efficiency and seamless integration.

*a) **Frontiers and Viewpoints**:* In addition to mapping, the system provides frontier detection and viewpoint computation. The frontiers are boundary zones between known and unknown areas, signifying potential areas of interest for further exploration. We achieve frontier detection by asynchronously analyzing smaller grids within the environment and identifying transition zones between known and unknown spaces. This process is optimized due to the grid structure since it is easy to identify the 26 adjacent cells surrounding the target cell at position (i, j, k). After identifying frontiers, the system calculates information gain through entropy reduction, signifying the decrease in uncertainty in information distribution. Each frontier is assessed based on its potential to reduce entropy, with prioritization given to those expected to yield a greater reduction. Frontiers with higher information gain are deemed strategically important for exploration.

*b) **Collision check**:* Instead of relying on external libraries, we have implemented an internally optimized GPU-based approach for collision checking. This involves converting waypoints into 3D grid coordinates and utilizing the GPU to efficiently iterate through them, checking for collisions by examining cells within a specified region or contained within the bounding box matching the robot's size. To enhance collision accuracy, we have developed a second version that utilizes the robot's model which in our case is an OBJ file. The algorithm iterates over each triangle in the robot's model and it checks for collision with the 3D grid for each waypoint. It uses bounding box checks (each triangle in the robot's model has a bounding box represented by triMin (minimum coordinates) and triMax (maximum coordinates)) to quickly eliminate triangles that are not intersecting with the grid at a high level. If the bounding boxes overlap, we apply the Möller–Trumbore algorithm [24] for a more detailed intersection check between each triangle and the grid.

*c) **Expansion and Truncation**:* For the 3D grid, we implemented the expansion as padding based on the displacement of the robot and its proximity to the boundaries of the grid. This adaptive approach ensures that the grid accommodates the movement of the robot while maintaining its spatial constraints. We also implemented a truncation mechanism that generates multiple sub-grids for different regions of interest which can be efficiently stored and manipulated, allowing the system to merge them back into the main grid as we retain knowledge of where each sub-grid was truncated.

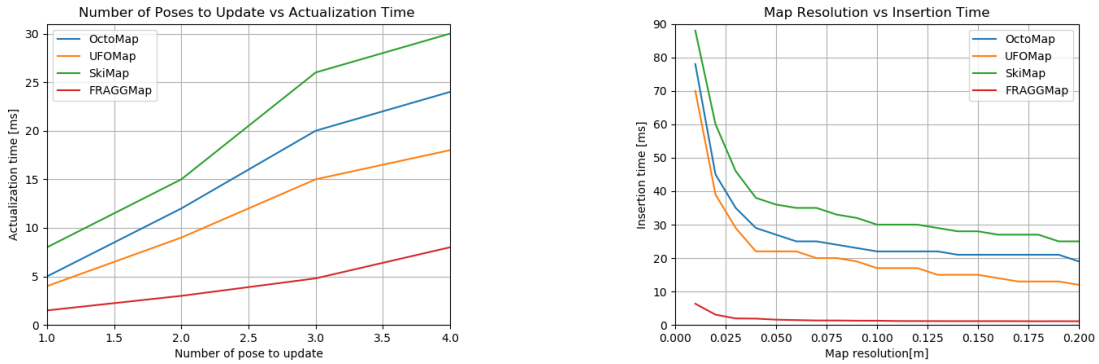
*d) **ROS**:* The system has a dedicated ROS node responsible for map creation and receiving real-time updates on sensor position and point-clouds. Furthermore, the system publishes visualization marker messages with cubic representations upon each point-cloud insertion, enabling convenient incremental visualization of the grid.

V. Evaluation

Our goal is to show that our system can handle multiple operations on the map while maintaining a similar quality compared to the systems in the literature. For that, we divided our tests into two parts. We evaluated the algorithm's insertion time using the Freiburg Campus dataset[3] as well as data collected using an underwater laser scanner [4]. The implementation was tested on Ubuntu 20.04 with GeForce RTX^{TM}3070 Ti 8GBs, Intel Core i7-11379H CPU (8 cores) @ 3.3GHz, 32 GB of RAM.

We compared our algorithm to the mapping frameworks OctoMap [6], UFOMap [7] and SkiMap [9]. For the evaluation, we used the Freiburg Campus dataset with a 5cm resolution. The data presented in Table I indicates that, on average, our system achieved an insertion time of 0.64 milliseconds per scan. Each scan was comprised of approximately 180000 points, with an additional CPU-GPU transfer of visible cells taking an average of 0.03 milliseconds. This cumulative time amounted to 0.67 milliseconds, making our system the fastest among the tested configurations. These results indicate that our method has an insertion time that is 31 times faster than OctoMap, with a comparative memory footprint. It is worth noticing that these results were obtained under the

[3]http://ais.informatik.uni-freiburg.de/projects/datasets/fr360



(a) (b)

Fig. 5: Comparison on the Freiburg Campus dataset a) comparison of the time needed by the algorithms to update the map as a function of the number of poses to update. b) comparison of the time needed by the maps to insert a point-cloud of ≈ 300000 points from Stonefish as a function of the resolution.

TABLE I: Comparison of all the maps in terms of voxel size, the time needed by all the maps to insert a point-cloud of ≈ 180000 points as well the memory footprint for the Freiburg Campus dataset, averaged over 100 insertions.

Algorithm	Voxel Size [bytes]	Time per Insertion [ms]	Mem. 3D grid [MB]
OctoMap	16	21 ± 2.0	155.4
UFO-Map	16	15 ± 1.5	**58.7**
Ski-Map	12	26 ± 3.2	588
FRAGG-Map	**4**	**0.67** ± 0.02	164.4

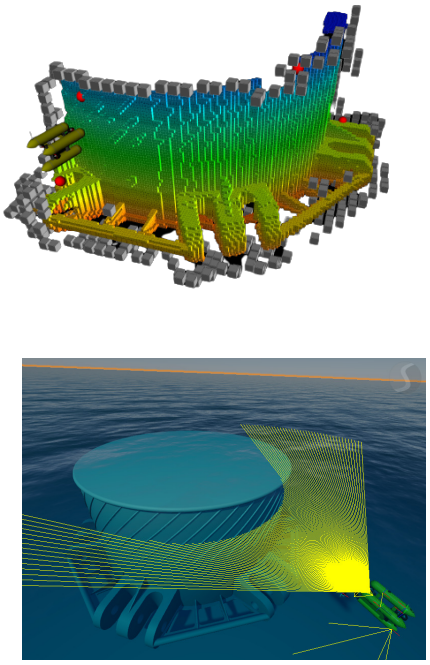


Fig. 6: (Top) Global view of the structure. The grey blocks represent the frontiers while the yellow and red spheres represent the viewpoints. (Bottom) Simulation in Stonefish [23] of the robot while scanning the structure.

assumption that the maps already hold the necessary data without requiring expansion. In the alternate case where the maps need to perform an expansion before being able to insert the new data, the advantage of our system will become even more apparent, as the update time remains constant. This time's stability is attributed to consistently working with the same data volume, and the map's padding simply instantiates new voxels without impacting the update process.

TABLE II: Performances of the different algorithms compared to the DeadReckoning: Difference with DeadReckoning (DwDR)

Algorithm	N. Occ. Cells	Diff w/ DR	DwDR %
OctoMap	4497	1042	18.8%
FRAGG-MAP	4572	967	17, 45%
UFO-MAP	4596	**943**	17.02
SKI-MAP	4459	1080	19.49%
DeadReckoning	5539	0	0, 0%

TABLE III: Resulting insertion time with different memory types

Memory Type	Time per Insertion [ms]
Pinned(Cells) & Texture(PCD)	**0.67** ± 0.02
Pageable(Cells) & Texture(PCD)	0.74 ± 0.03
Pinned(Cells)	0.81 ± 0.02
Pageable(Cells)	1.02 ± 0.05

In the second part of the tests, we performed the comparison of the time needed by the algorithms to modify the poses of the point-cloud as it would happen if a loop closure was detected as well as a comparison of the insertion time as a function of the map resolution. In Figure $5a$, the performance of our system in updating the map, relative to the number of positions to change is presented. Notably, as the number

of positions to update increases, the time to update does not increase significantly. Meanwhile, in Figure $5b$, it is evident that our system consistently exhibits the fastest insertion time across various map resolutions. This is most notable at lower resolutions, where our system has an insertion time that is at least 12 times lower than the other methods.

Additionally, we conduct a qualitative comparison of errors among various algorithms utilizing data from [4]. As illustrated in Table II, our system performs similarly to the other tested systems concerning the number of occupied cells. This similarity indicates that our system maintains comparable quality while demanding less computational time. Furthermore, Figure 2 shows the map of the pipe structure using our method compared to the map relying only on the Dead Reckoning, the map generated using the SLAM and not corrected and the map generated using OctoMap.

Finally, we conduct an ablation study on our system, exploring the impact of different memory types for storing point clouds and cells. Table III illustrates the performance metrics for point cloud insertion, specifically with 180,000 points, across four distinct memory configurations:

- Using Pinned Memory for cells alongside Texture Memory for point clouds.
- Employing Pageable Memory, the system's default, with Texture Memory for point clouds.
- Solely using Pinned Memory.
- Solely relying on Pageable Memory.

Our results demonstrate that the configuration combining Pinned Memory for cells and Texture Memory for point clouds exhibit the highest performance. This configuration is optimized for both CPU-GPU transfers and data access, resulting in superior efficiency.

VI. Conclusion

The proposed algorithm represents a significant time gain compared to the state of the art, FRAGG-Map runs 31 times faster than Octomap, 22 time faster than UFOMap, and 38 times faster than SkiMap, guaranteeing map update time between 0.5 and 2ms regardless of the size and resolution of the sub-grid. The update time is linear depending on the number of positions to be updated and together with the update time of 0.5s-2ms we guarantee a total order update time of milliseconds. It should also be noted that using the GPU means a double time gain for the robot, as the CPU can perform other jobs while the GPU is running.

There are several directions for future work. One potential avenue is the integration of semantic mapping, allowing for the representation of not only the physical occupancy but also the semantic meaning of the environment. This extension would enable robots to understand and interact with their surroundings more intelligently. Furthermore, we intend to utilize our system with GTSAM [25], making use of ISAM2 [26] to track graph changes during loop closure.

References

[1] H. Moravec and A. Elfes, "High resolution maps from wide angle sonar," vol. 2, 04 1985, pp. 116 – 121.

[2] G. Vallicrosa and P. Ridao, "H-slam: Rao-blackwellized particle filter slam using hilbert maps," *Sensors*, vol. 18, no. 5, p. 1386, 2018.

[3] A. R. Rormero, P. V. K. Borges, A. Pfrunder, and A. Elfes, "Map-aware particle filter for localization," in *2018 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2018, pp. 2940–2947.

[4] A. Palomer, P. Ridao, and D. Romagós, "Inspection of an underwater structure using point-cloud slam with an auv and a laser scanner," *Journal of Field Robotics*, vol. 36, 09 2019.

[5] G. G. B. Steder and W. Burgard, "Dataset of 360° 3d scans of the faculty of engineering, university of freiburg, germany," http://ais.informatik.uni-freiburg.de/projects/datasets/fr360.

[6] A. Hornung, K. M. Wurm, M. Bennewitz, C. Stachniss, and W. Burgard, "OctoMap: An efficient probabilistic 3D mapping framework based on octrees," *Autonomous Robots*, 2013.

[7] D. Duberg and P. Jensfelt, "Ufomap: An efficient probabilistic 3d mapping framework that embraces the unknown," *IEEE Robotics and Automation Letters*, vol. PP, pp. 1–1, 08 2020.

[8] K. Stepanas, J. Williams, E. Hernández, F. Ruetz, and T. Hines, "Ohm: Gpu based occupancy map generation," *IEEE Robotics and Automation Letters*, vol. 7, no. 4, pp. 11 078–11 085, 2022.

[9] D. De Gregorio and L. Di Stefano, "Skimap: An efficient mapping framework for robot navigation," in *IEEE International Conference on Robotics and Automation*, 2017.

[10] A. Hermann, F. Drews, J. Bauer, S. Klemm, A. Rönnau, and R. Dillmann, "Unified gpu voxel collision detection for mobile manipulation planning," *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 4154–4160, 2014. [Online]. Available: https://api.semanticscholar.org/CorpusID:6825481

[11] H. Oleynikova, Z. Taylor, M. Fehr, R. Siegwart, and J. Nieto, "Voxblox: Incremental 3d euclidean signed distance fields for on-board mav planning," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2017.

[12] H. Oleynikova, A. Millane, Z. Taylor, E. Galceran, J. Nieto, and R. Siegwart, "Signed distance fields: A natural representation for both mapping and planning," in *RSS 2016 workshop: geometry and beyond-representations, physics, and scene understanding for robotics*, 2016.

[13] H. Min, K. M. Han, and Y. J. Kim, "Octomap-rt: Fast probabilistic volumetric mapping using ray-tracing gpus," *IEEE Robotics and Automation Letters*, vol. 8, no. 9, pp. 5696–5703, 2023.

[14] K. Sung, P. Shirley, and S. Baer, *Essentials of interactive computer graphics: concepts and implementation*. CRC Press, 2008.

[15] M. Nießner, M. Zollhöfer, S. Izadi, and M. Stamminger, "Real-time 3d reconstruction at scale using voxel hashing," *ACM Transactions on Graphics (TOG)*, vol. 32, 11 2013.

[16] M. Klingensmith, I. Dryanovski, S. Srinivasa, and J. Xiao, "Chisel: Real time large scale 3d reconstruction onboard a mobile device using spatially hashed signed distance fields," 07 2015.

[17] L. Han and L. Fang, "Flashfusion: Real-time globally consistent dense 3d reconstruction using cpu computing," 06 2018.

[18] C. Dong, X. Chen, R. Hu, J. Cao, and X. Li, "Mvss-net: Multi-view multi-scale supervised networks for image manipulation detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 45, no. 03, pp. 3539–3553, 2023.

[19] V. Walker, F. Vanegas, and F. Gonzalez, "Nanomap: A gpu-accelerated openvdb-based mapping and simulation package for robotic agents," *Remote Sensing*, vol. 14, no. 21, 2022.

[20] U. Assarsson and T. Möller, "Optimized view frustum culling algorithms," 03 2000.

[21] J. E. Bresenham, *Algorithm for Computer Control of a Digital Plotter*. Association for Computing Machinery, 1998, p. 1–6.

[22] M. Yguel, O. Aycard, and C. Laugier, "Update policy of dense maps: Efficient algorithms and sparse representation," in *Field and Service Robotics: Results of the 6th International Conference*, 2008, pp. 23–33.

[23] P. Cieślak, "Stonefish: An advanced open-source simulation tool designed for marine robotics, with a ros interface," in *OCEANS 2019*, 2019.

[24] T. Möller and B. Trumbore, "Fast, minimum storage ray/triangle intersection," in *ACM SIGGRAPH 2005 Courses*, ser. SIGGRAPH '05. Association for Computing Machinery, 2005, p. 7–es.

[25] M. Kaess, "Gtsam library," 2015.

[26] M. Kaess, H. Johannsson, R. Roberts, V. Ila, J. Leonard, and F. Dellaert, "isam2: Incremental smoothing and mapping with fluid relinearization and incremental variable reordering," in *2011 IEEE International Conference on Robotics and Automation*, 2011, pp. 3281–3288.