

Efficient Global Occupancy Mapping for Mobile Robots using OpenVDB

1st Raphael Hagmanns

*Karlsruhe Institute for Technology (KIT) and
Fraunhofer Institute of Optronics,
System Technologies and Image Exploitation
Karlsruhe, Germany
raphael.hagmanns@kit.edu*

3rd Marvin Grosse Besselmann

*FZI Forschungszentrum Informatik
Karlsruhe, Germany
grossebesselmann@fzi.de*

2nd Thomas Emter

*Fraunhofer Institute of Optronics,
System Technologies and Image Exploitation
Karlsruhe, Germany
thomas.emter@iosb.fraunhofer.de*

4th Jürgen Beyerer

*Karlsruhe Institute for Technology (KIT) and
Fraunhofer Institute of Optronics,
System Technologies and Image Exploitation
Karlsruhe, Germany*

Abstract—In this work we present a fast occupancy map building approach based on the VDB datastructure. Existing log-odds based occupancy mapping systems are often not able to keep up with the high point densities and framerates of modern sensors. Therefore, we suggest a highly optimized approach based on a modern datastructure coming from a computer graphic background. A multithreaded insertion scheme allows occupancy map building at unprecedented speed. Multiple optimizations allow for a customizable tradeoff between runtime and map quality. We first demonstrate the effectiveness of the approach quantitatively on a set of ablation studies and typical benchmark sets, before we practically demonstrate the system using a legged robot and a UAV.

Index Terms—occupancy mapping, map representation, UAV, OpenVDB

I. INTRODUCTION

A detailed understanding of a potentially unknown environment plays a fundamental role in mobile robotic applications. Different robots and environments come along with varying requirements for the map building process in terms of accuracy, efficiency and usability. Common SLAM methods, which attempt to map an environment while simultaneously localizing the robot in it, usually have to find a balance between these properties. Due to the complexity of the task, it is still very challenging to perform SLAM while maintaining a dense map representation. Compared to commonly used sparse map representation, *occupancy grids* have many advantages as they integrate all available information into a single representation which is easy to understand for an operator and also allows for efficient queries.

2D projections of such dense representations have been used extensively for mobile robot navigation tasks. As robots

This work has been conducted within the competence center ROBDEKON – Robotic Systems for Decontamination in Hazardous Environments, which is funded by the Federal Ministry of Education and Research (BMBF) within the scope of the German Federal Government’s Research for Civil Security program under grant no. 13N14674.

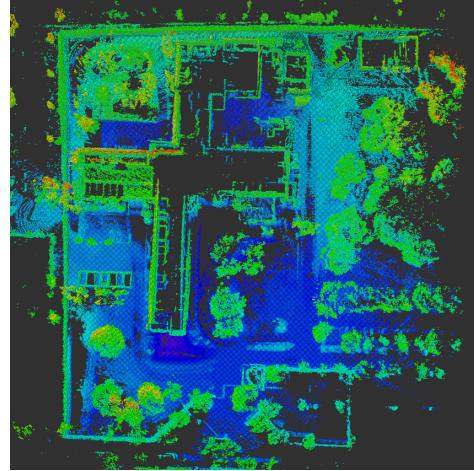


Fig. 1: Map of the Fraunhofer IOSB site using the mapping approach. Preprocessing using a custom factor graph approach described in [2].

become more agile, scenes more complex and sensors more capable, it is also desireable to adopt these structures for the third dimension. For ground vehicles, so called 2.5D elevation maps as suggested by Herbert et. al [1] may be sufficient. However, for agile robots with complex kinematics such as legged robots or UAVs, a full 3D representation of the environment is essential.

To store the 3D map memory efficient and with certain complexity guarantees for random access operations, different datastructures have been suggested. The most prominent example for such a 3D map representaion is the OctoMap [3] framework by Hornung et al., working on octrees as hierarchical tree structure. It allows for a memory effiecient storing through efficient pruning and propagating leaf states to higher levels of the tree. OctoMap has been considered state-of-the-art for a long time but as sensors are achieving higher rates with millions of points per second, the *insert* operation of

OctoMap is not able to achieve real-time performance. As the octree has a fixed layout, it is also difficult to later increase the volume without performance overhead. One of the most popular frameworks for global consistent mapping is VoxBlox by Oleynikova et al. [4]. They incrementally build a *Truncated Signed Distance Map* (TSDF) [5] instead of an occupancy voxel grid and reach almost real-time performance on a single core implementation using a hashmap representation instead of a tree as fundamental datastructure.

OpenVDB is a modern framework developed in a computer graphic context. Similar to OctoMap it is based on a hieracical structure but it comes with certain accelerators to support almost constant insertion and read procedures using a B+ tree like structure. The main reason for the improved performance is an advanced indexing and caching system. We therefore leverage OpenVDB as underlying datastructure for our map building approach and present its capabilities in further detail in the upcoming Section II. OpenVDB as flexible backbone allows not only fast insertions but also supports efficient raycasting step samplers and a virtually infinite map size.

Only few works utilize the VDB datastructure as occupancy representation so far. Our work was originally based on [6] by Besselmann et al. and can be considered a successor with a revised update scheme and optimized insertion procedure. They bring up the idea to integrate data into a temporary grid first to cope with discretization ambiguities which arise when raycasting new data. In [7], Zhu et al. present a full framework for occupancy and distance mapping, which also uses a raycast based insertion scheme in order to create the occupancy map. They put the focus on the *Euclidean Distance Transform* step and disregard the map integration itself. Macenski et al. [8] built a spatio-temporal voxel layer on top of OpenVDB. They focus on local dynamic maps and therefore use a sensor frustum based visibility check instead of raycasting as integration scheme.

In our work we further push the limits of the underlying OpenVDB structure by supporting a flexible multithreaded raycasting insertion scheme into the map supported by additional ray-level hashing to avoid unnecessary operations. Fast merging operations of single *bit-grids* allow for a minimal lock time for modifying the global occupancy grid. Different subsample strategies can be selected to allow for a dynamically adjustable tradeoff between map accuracy and efficiency.

The main contribution of the work can be summarized as follows:

- We present a real-time capable and multithreaded dense mapping approach for efficiently creating occupancy maps based on the VDB data structure.
- We introduce several optimizations in the integration scheme allowing for a user-definable tradeoff between map accuracy and efficiency.
- We conduct benchmarks on different operations and test the whole pipeline in simulation and real environments.
- We open-source the codebase and a corresponding wrapper for the Robot Operating System II (ROS II [9]) which enables fast prototyping for mobile robotic applications.

The remainder of this work is structured as follows. Section II formalizes the problem of mapping and introduces OpenVDB as underlying datastructure. We also give a detailed overview on the insertion scheme and introduce various optimizations leading to improved performance. In Section III we carry out different experiments to verify the effect of the introduced optimizations. We summarize and conclude the work and discuss potential future improvements in Section IV.

II. MAPPING PIPELINE

In this section, we first formalize the problem before discussing the proposed insertion scheme.

A. Problem Statement

The main goal of occupancy mapping is to create a map \mathcal{M}_{occ} which stores the occupancy probability $p(x_i|s_{1:t}, z_{1:t})$ for each cell $x_i \in \mathcal{M}_{\text{occ}}$ given some sensor measurements $s_{1:t}$ and the corresponding robot poses $z_{1:t}$, where $1:t$ denotes the sequence from the start up to time t . We consider a cell to be an obstacle if $p(x_i|s_{1:t}, z_{1:t})$ exceeds a certain threshold ϕ_{occ} and free if it falls below the threshold ϕ_{free} . Note that being marked as *free* is different from not being observed yet.

B. OpenVDB

The OpenVDB framework has been introduced by Museth et al. in 2013 [10]. Originally it was designed for computer graphic applications such as rendering animations of complex mesh structures and time-varying sparse volumes such as clouds. Since then, it has been widely adopted to different applications as it allows for flexible modifications to its core structure. At its heart it leverages a B+ tree [11] variant as main datastructure. This structure is supported by hieracically organized caches to faciliate fast access to inner tree nodes. Such a datastructure is ideal to store sparse voxelized environment representations. The discretization of the space can be adjusted by choosing the size of the leaf nodes accordingly. Other adjustable parameters are the tree depth and branching factors which can further improve the memory footprint depending on the sparsity of the environment. Typical branching factors

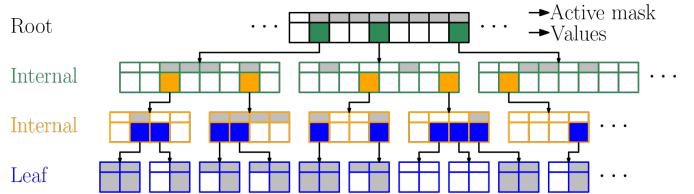


Fig. 2: Illustration of the underlying VDB datastructure. Image is adopted from the original publication [10]. The height of the tree is typically 4 with one root node (gray) and two internal layers (green, orange) as well as leaf nodes, which store the actual tile values (blue). All nodes in lower levels have a branching factor equal to a power of two. Root and internal nodes store pointers to their respective child nodes. An *active* bitmask for each nodes encodes if subsequent tiles are active or not (gray values).

for the VDB datastructures are very large compared to the octree branching factors, which are typically two in each spatial dimension and thus lead to less shallow trees. The schematics of the underlying tree structure is depicted in Figure 2. The fixed height of the tree makes it possible to implement *insert* and *read* operations in constant time on average [10]. It also allows the framework to efficiently utilize typical cache architectures on modern CPUs to further speed up read operations for tiles with spatial proximity. Access to the tree’s tile values is implemented via a virtually infinite *index space*, which can be accessed by signed 32-bit integer coordinates, allowing the map to grow in each direction without additional overhead. As noted in Figure 2, tree nodes additionally store bitsets indicating if subsequent nodes are active or not. This allows for a fast traversal of a sparse volume without the need to visit tile values explicitly. The features of the tree structure are described in further detail in the original work [10]. Moreover, VDB allows for efficient raycasting operations using optimized index space iterators. Therefore, we utilize a raycast based sensor insertion scheme which is described in further detail in the following section.

C. Map Updates

The proposed structure is not tied to a specific sensor. Typical data comes from a LiDAR sensor, mimicing the raycast operation which is also performed to create the obstacle map. Cells store their occupancy probability in order to reflect not only occupied but also free space. To represent time-dependent updates we faciliate the very commonly used log-odds based update scheme initially formulated by Moravec and Elfes in [12]. Essentially we calculate

$$\mathcal{M}_{\text{occ}}(x_i|s_{1:t}) = \mathcal{M}_{\text{occ}}(x_i|s_{1:t-1}) + \log \left[\frac{p(x_i|s_{1:t})}{1 - p(x_i|s_{1:t})} \right] \quad (1)$$

in each update step and for each $x_i \in \mathcal{M}_{\text{occ}}$.

Algorithm 1 gives an abstracted overview on the insertion process. Essentially we do parallel raycasting in coaligned temporary grids, which are merged together in a later step. We first create a coaligned map \mathcal{M}_{agg} (line 2), which we later use to aggregate the temporary maps. We then divide the incoming points into different chunks which can be processed in parallel (line 5). Points of each chunk are inserted by calculating their respective end position in world coordinates (line 9) and marching along the ray using OpenVDBs digital differential analyzer (DDA) implementation (lines 19-22) and marking all visited voxels as *active*. The only place where we need to lock the threads is during the merge operation in lines 23-24. This can be done efficiently as we simply XOR the boolean grids together. As OpenVDB stores the active state of nodes in a fast accessible bitmask, we can now efficiently iterate over all *active* values in our aggregated map \mathcal{M}_{agg} (line 25). We increase or decrease the occupancy value following Equation 1 (lines 27 to 29). If a voxel exceeds or falls below a certain threshold it will be marked as occupied or unoccupied.

We suggest and implement two runtime optimizations, which are roughly based on similar ideas used in Voxblox [4],

namely a *subsampling* and a *bundling* optimiztation. *Subsampling* (cf. Figure 3c) uses an additional map \mathcal{M}_{sub} which increases the resolution \mathcal{M}_{occ} by a *subsampling-factor* δ_{sub} . Typically, δ_{sub} is restricted to powers of 2, we use 4 in most setups. In addition we use a Hashmap \mathcal{H}_{sub} (line 3) storing for each cell in \mathcal{M}_{sub} , if it has already been visited in the current integration step. If this is the case, all subsequent integrations are skipped (lines 12 and 13). This optimization comes with the cost of inaccurate details but can save a lot of integration steps especially in dense environments with large voxel sizes, where a single voxel is hit multiple times.

Algorithm 1: Map Update Scheme

Input : Pointcloud \mathcal{P} , Sensor origin \mathbf{o} , Number of Chunks c , Global Occupancy Map \mathcal{M}_{occ}

```

1  $\mathcal{M}_{\text{sub}} \leftarrow \text{coalign}(\mathcal{M}) // \text{Aligned Subsampling Map}$ 
2  $\mathcal{M}_{\text{agg}} \leftarrow \text{coalign}(\mathcal{M}) // \text{Aligned Aggregation Map}$ 
3 Hashmap<coord  $x$ , bool hit>  $\mathcal{H}_{\text{sub}} \leftarrow []$ 
4 Hashmap<int count, coord  $x$ , bool maxray>  $\mathcal{H}_{\text{bun}} \leftarrow []$ 
5  $\mathcal{P}_i \leftarrow \text{Equal Chunks of } \mathcal{P} \text{ for } i = 1..c$ 
6 foreach  $\mathcal{P}_i \in \mathcal{P}$  in parallel
7    $\mathcal{M}_{\text{temp}} \leftarrow \text{coalign}(\mathcal{M}) // \text{Aligned Temporary Map}$ 
8   foreach  $\mathbf{p} \in \mathcal{P}_i$  do
9      $\mathbf{r}_{\text{end}} = \mathbf{o} + (\mathbf{p} - \mathbf{o}) \text{ in } \mathcal{M}_{\text{agg}}$ 
10     $\mathbf{r}_{\text{end\_sub}} = \mathbf{o} + (\mathbf{p} - \mathbf{o}) \text{ in } \mathcal{M}_{\text{sub}}$ 
11    is_maxray  $\leftarrow$  check for max-length ray
12    if  $\mathcal{H}_{\text{sub}}[\mathbf{r}_{\text{end\_sub}}]$  then
13      continue
14    if  $\mathcal{H}_{\text{bun}}[\mathbf{r}_{\text{end}}].count > \text{thresh}$  then
15      ( $\text{count}, \mathbf{r}_{\text{end}}$ , maxray)  $\leftarrow \mathcal{H}_{\text{bun}}[\mathbf{r}_{\text{end}}]$ 
16    else
17       $\mathcal{H}_{\text{bun}}[\mathbf{r}_{\text{end}}] += (1, \mathbf{r}_{\text{dir}}, \text{is\_maxray})$ 
18      continue
19    do
20       $\mathcal{M}_{\text{temp}}[\mathbf{r}_{\text{dda}}].active = \text{true}$ 
21       $\mathbf{r}_{\text{dda}} ++$ 
22      while  $\mathbf{r}_{\text{dda}} \neq \mathbf{r}_{\text{end}}$ 
23    with  $\text{MapLock}(\mathcal{M}_{\text{temp}})$  do
24       $\mathcal{M}_{\text{agg}} |= \mathcal{M}_{\text{temp}}$ 
25 foreach active value  $x \in \mathcal{M}_{\text{agg}}$  do
26   if  $x$  then
27     increase occupancy on  $\mathcal{M}_{\text{occ}}[x]$  following Eq. 1
28   else
29     reduce occupancy on  $\mathcal{M}_{\text{occ}}[x]$  following Eq. 1

```

Output : Updated \mathcal{M}_{occ}

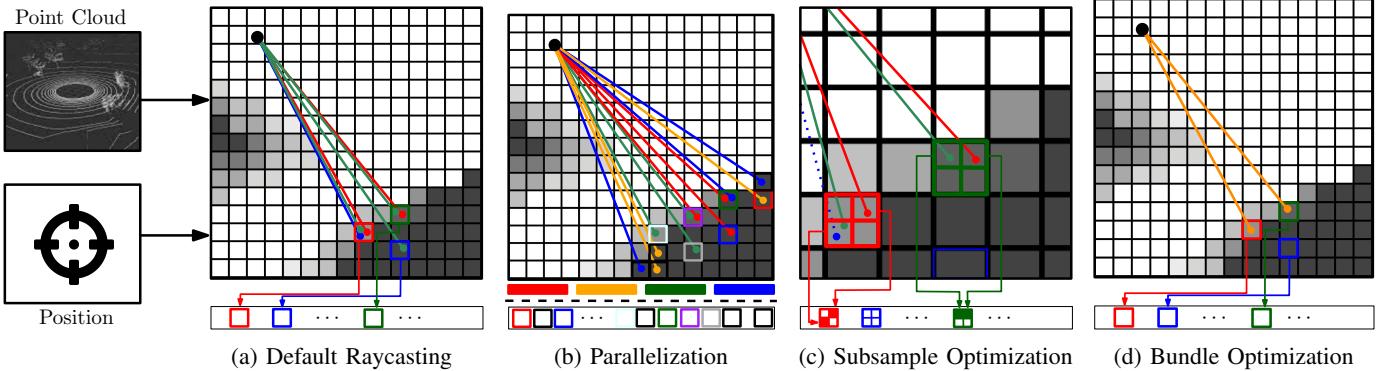


Fig. 3: Scheme of different optimizations on the insertion procedure. In (a), the standard raycasting process is visualized. When enough rays hit a voxel so that ϕ_{occ} is exceeded, this voxel will be marked as occupied. Figure (b) visualizes the process of dividing rays into different chunks which can then be processed in a multithreaded fashion. In (c) the target space is further subsampled with a customizable subsampling map M_{sub} . Rays which hit an already marked subsampled cell (the blue ray) are skipped. In (d) multiple rays are aggregated and averaged if they hit the same target cell.

The *bundling* optimization on the other hand bundles multiple rays together. Again, we use a hashmap H_{bun} where we insert incoming ray end points without actually integrating the rays (lines 17 and 18). If a certain threshold is exceeded, we integrate the whole bundle targeting the end cell r_{end} at once. In the map, we additionally store the original end points and if the ray reached its maximum length. During the integration of the bundle this information is used to average the final end point \bar{r}_{end} . Again, this optimization favors environments with a lot of redundant integrations. Figure 3d indicates that only one orange bundle is inserted into the map, even if multiple rays hit the cell (cf. Figure 3a). Both optimizations can be enabled or disabled individually or together. While enabling the optimizations leads to a deliberately impaired map accuracy, it can be useful as more sensor data can be integrated overall due to the additionally gained performance. It is worth noticing that it is not necessary to deal with discretization ambiguities introduced by sequential raycasting presented in [6] as we use the same two step approach as in [6]: First we activate all visited voxels in a separate aggregation map M_{agg} before we integrate it into the global map M_{occ} .

III. EVALUATION

We will now present the results of experiments which we conducted to measure the performance of our proposed methods under different conditions. We first compare different iterations of our method in a set of ablation studies to measure the effect of different optimizations. In a next step we compare the method on typical benchmark sets before we finally conduct some real world experiments by capturing outdoor and indoor scenes of our lab. All experiments are performed using a machine equipped with a 6-core Intel®Core™i7-10850H and 32 GB of memory. As hardware platforms to carry out our experiments we use a BostonDynamics Spot equipped with an Ouster OS0-64 LiDAR for outdoor experiments as well as a custom UAV platform with a solid-state LiDAR for indoor experiments (see Figure 4 for details).



(a) BostonDynamics Spot with additional sensors (b) Custom UAV platform for indoor usage

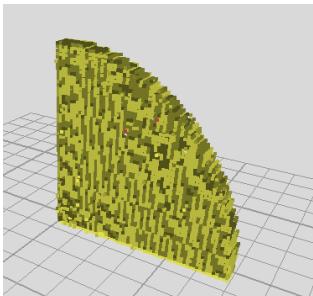
Fig. 4: The hardware setup used for indoor and outdoor experiments. Both robots are equipped with a solid-state LiDAR (Realsense L515) and provide a localization coming from a multisensor-fusion approach.

A. Ablation Studies

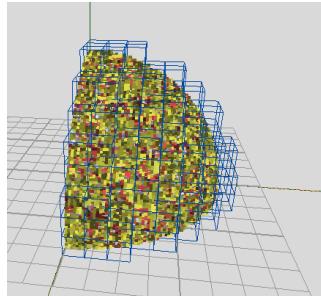
We first compare different versions of our method with each other and against baselines from other works. We use random insertions of pointclouds with different sizes as a baseline setting. All experimental results are averaged over 5 runs. The results are presented in Table I. Even the basic variant using OpenVDB is able to outperform OctoMap by a factor of 2, in multiple settings this advantage grows up to a factor of 30. Interestingly, the parallel integration scheme *VDB-PAR* achieves almost linear speedup for small ray lengths. Figure 6a gives further insight into that evaluation. The reason for the performance drop for higher ray lengths is the increasing amount of time required for the *merge* operations as insertion performance stays constant for increasing submap sizes while the merge workload grows cubic. This insight can be derived from Figure 6b, where the different steps of an integration procedure are measured. Consequently, the best parallelization

TABLE I: Runtime of different variants with 0.1m map resolution and n insertions each. *Random* refers to randomly sampled points in a radius which is $1.2 \cdot$ ray length as depicted in (b). The *structured* environment (a) refers to a sampling where (x, y) coordinates are sampled randomly but z coordinates are restricted to a small width of 1m simulating a wall structure. (*) VDB-BUN and VDB-SUB approaches are only included as rough estimate and do not compare to the other methods, as some rays are not casted, when using bundle or subsampling optimizations. The parallel version VDB-PAR runs on 12 threads and VDB-FMAP is the improved and parallel variant restricted to a single core and with disabled optimizations while VDB-MAP is the VDB based method described in [6].

n	method	runtime [ms]			
		ray length 6m		ray length 60m	
		structure	random	structure	random
50 000	OctoMap [3]	110	747	11676	42 593
	VDB-EDT [7]	102	106	3 005	3 973
	VDB-MAP [6]	55	74	863	1 984
	VDB-FMAP	54	71	906	2 075
	VDB-SUB*	56	75	923	2 149
	VDB-BUN*	31	40	491	1 083
	VDB-PAR	10	20	1308	5 581
500 000	OctoMap [3]	796	2 189	44 072	223 672
	VDB-EDT [7]	957	987	22 010	28 684
	VDB-MAP [6]	547	709	6 042	11 364
	VDB-FMAP	547	671	5 779	10 718
	VDB-SUB*	288	583	5 011	10 998
	VDB-BUN*	261	330	2 202	4 764
	VDB-PAR	54	74	1 979	10 339



(a) Structured Raycasting



(b) Random Raycasting

speedup can be achieved in low-range settings with a lot of points to be integrated. This exactly matches the domain of indoor mapping scenarios using high-resolution solid-state LiDARs as we will show in the next section. A detailed evaluation of different ray lengths is given in Figure 6c. OctoMap and the VDB-EDT approach from [7] are outperformed in low range (below 30m) scenarios by almost a magnitude.

The *bundling* optimization VDB-BUN guarantees to save runtime, as the first ray to a specific voxel is skipped in every case. This approximately halves the runtime over all settings.

TABLE II: Benchmarks on the *cow-and-lady* dataset. #Points denote the amount of processed points over all frames. This varies due to different processing speeds and different temporal alignments between poses and pointcloud. Time measures the total integration time and #Occupied Voxels counts the number of occupied voxels after the integration procedure.

Name	#Points	#Occupied Voxels	Time per frame [ms]
OctoMap [3]	$0.463 \cdot 10^9$	$0.332 \cdot 10^6$	388
VDB-EDT [7]	$0.557 \cdot 10^9$	$0.567 \cdot 10^6$	357
VDB-MAP [6]	$0.518 \cdot 10^9$	$0.536 \cdot 10^6$	263
VDB-FMAP	$0.513 \cdot 10^9$	$0.523 \cdot 10^6$	282
VDB-BUN	$0.522 \cdot 10^9$	$0.316 \cdot 10^6$	94
VDB-SUB	$0.521 \cdot 10^9$	$0.510 \cdot 10^6$	170
VDB-PAR	$0.526 \cdot 10^9$	$0.530 \cdot 10^6$	47

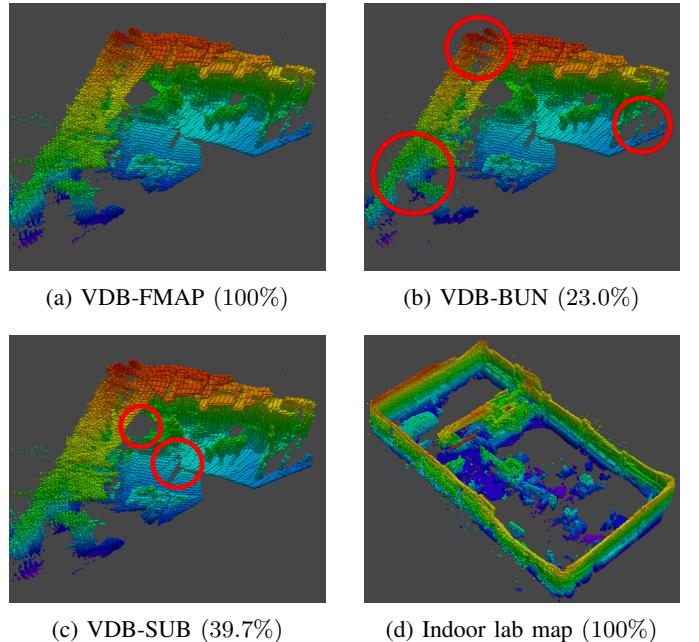


Fig. 5: Qualitative visualization of different mapping optimization effects on the *cow-and-lady* dataset in (a)-(c) as well as the lab environment captured by an indoor UAV (cf. 4b) in (d). The percentage denotes how many of the original points are casted as rays using the respective optimization.

The *subsampling* optimization VDB-SUB on the other hand only applies when many points reside in a small volume. Consequently it saves the most runtime in a setting with many points in a *structured* environment, whereas it is not faster or even slower for different settings.

B. Benchmark and Real Datasets

We evaluate the presented methods on the indoor *cow-and-lady* dataset released as part of VoxBlox [4]. It consists of 2831 depth frames captured by a Microsoft Kinect I as well

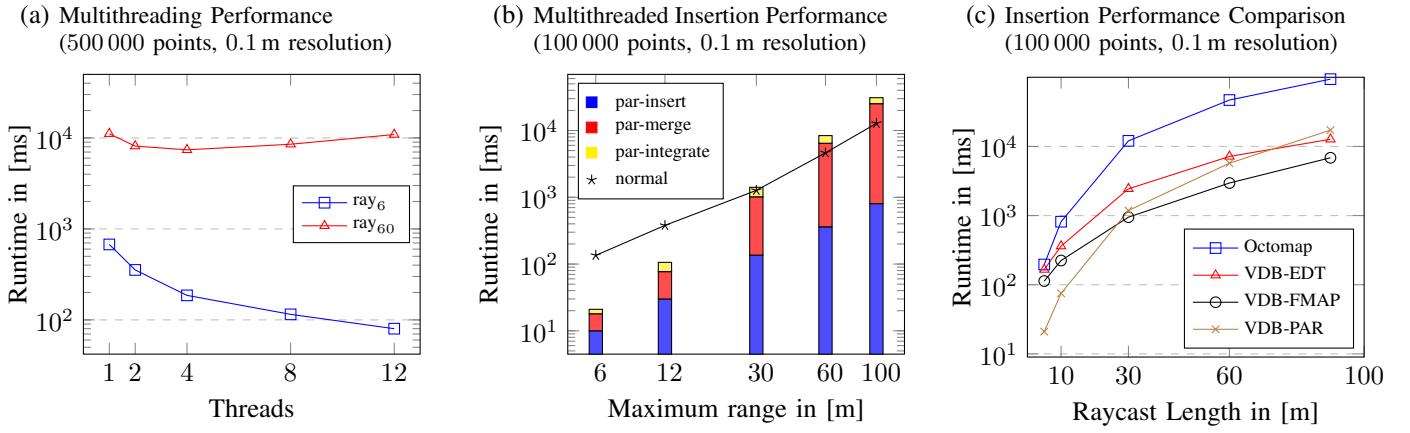


Fig. 6: Evaluation and comparison of map integration procedures. (a) and (b) examines the influence of multithreading on the integration scheme, while (c) compares different ray distance limits with other approaches. Note the log-scale at the runtime.

as corresponding pose data. The results in Table II show that VDB based methods outperform previous methods in terms of simple integration performance. For the *bundling* optimization VDB-BUN, the resulting amount of occupied voxels is only half compared to the other variants. This indicates that the map quality is reduced. On the other hand, the *subsampling* strategy VDB-SUB only comes along with a negligible reduction of occupied voxels. This shows that mostly redundant rays are omitted in the integration procedure while it is almost able to halve the runtime. Again, the parallel version outperforms all other versions without any reduction in accuracy.

Figure 5 exemplarily shows an excerpt of the *cow-and-lady* dataset after 500 frames for different integration strategies. There are only few details (marked in red) of quality loss for saving more than 60% of the integration steps using the subsampling optimization and almost 80% using the bundled integration. Figures 1 and 5d show that our mapping procedure is capable of producing high quality maps of outdoor and indoor environments using a legged robot or a drone, respectively. Even without preprocessing the poses using a factor graph [13], the resulting map quality is satisfactory, if the pose drift is not too large (cf. 5d).

IV. CONCLUSION AND DISCUSSION

We presented an efficient map building approach for mobile robots, especially suitable for agile robots in dynamic environments. The experimental evaluation demonstrates the effectiveness of the approach particularly in indoor environments with low sensor ranges, where the approach outperforms current solutions. The parallel fusion of different maps allows not only fast but also flexible integration of new sensor data into the map. This leaves room for additional improvements such as an extension with dynamic resolution adaption. Maintaining a real-time distance map is also a valuable extension, which could be implemented efficiently using the VDB datastructure. In order to reduce global inconsistencies coming from drifts in the localization, one could couple the VDB representation with

a factor graph backend. This has been tested in a prototyped fashion as demonstrated in Figure 1 but can potentially be improved by a tighter coupling.

REFERENCES

- [1] M. Herbert, C. Caillas, E. Krotkov, I. Kweon, and T. Kanade, “Terrain Mapping for a Roving Planetary Explorer,” in *Proceedings, 1989 International Conference on Robotics and Automation*, 1989, pp. 997–1002.
- [2] T. Emter and J. Petereit, “3D SLAM With Scan Matching and Factor Graph Optimization,” in *ISR 2018; 50th International Symposium on Robotics*, 2018.
- [3] 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.
- [4] 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 (IROS)*, 2017.
- [5] H. Oleynikova, A. Millane, Z. Taylor, E. Galceran, J. Nieto, and R. Siegwart, “Signed Distance Fields: A Natural Representation for Both Mapping and Planning,” 2016.
- [6] M. Grosse Besselmann, L. Puck, L. Steffen, A. Roennau, and R. Dillmann, “VDB-Mapping: A High Resolution and Real-Time Capable 3D Mapping Framework for Versatile Mobile Robots,” 2021.
- [7] D. Zhu, C. Wang, W. Wang, R. Garg, S. A. Scherer, and M. Q. Meng, “VDB-EDT: An Efficient Euclidean Distance Transform Algorithm Based on VDB Data Structure,” *CoRR*, vol. abs/2105.04419, 2021.
- [8] S. Macenski, D. Tsai, and M. Feinberg, “Spatio-temporal voxel layer: A view on robot perception for the dynamic world,” *International Journal of Advanced Robotic Systems*, vol. 17, no. 2, 2020.
- [9] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, “Robot operating system 2: Design, architecture, and uses in the wild,” *Science Robotics*, vol. 7, no. 66, p. eabm6074, 2022.
- [10] K. Museth, J. Lait, J. Johanson, J. Budberg, R. Henderson, M. Alden, P. Cucka, D. Hill, and A. Pearce, “OpenVDB: An Open-Source Data Structure and Toolkit for High-Resolution Volumes,” in *ACM SIGGRAPH 2013 Courses*, ser. SIGGRAPH ’13. New York, NY, USA: Association for Computing Machinery, 2013.
- [11] R. Bayer and E. McCreight, “Organization and Maintenance of Large Ordered Indices,” in *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, ser. SIGFIDET ’70, 1970, p. 107–141.
- [12] H. Moravec and A. Elfes, “High Resolution Maps from Wide Angle Sonar,” in *Proceedings. 1985 IEEE International Conference on Robotics and Automation*, vol. 2, 1985, pp. 116–121.
- [13] F. Dellaert and M. Kaess, *Factor Graphs for Robot Perception*, 2017.