OctoCache: Caching Voxels for Accelerating 3D Occupancy Mapping in Autonomous Systems

Peiqing Chen
University of Maryland
College Park, MD, USA
pqchen99@umd.edu

Minghao Li
Harvard University
Cambridge, MA, USA
minghaoli@g.harvard.edu

Zishen Wan
Georgia Institute of Technology
Atlanta, GA, USA
zishenwan@gatech.edu

Yu-Shun Hsiao
Harvard University
Cambridge, MA, USA
yushun_hsiao@g.harvard.edu

Minlan Yu
Harvard University
Cambridge, MA, USA
minlanyu@g.harvard.edu

Vijay Janapa Reddi
Harvard University
Cambridge, MA, USA
vj@eecs.harvard.edu

Zaoxing Liu
University of Maryland
College Park, MD, USA
zaoxing@umd.edu

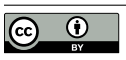
Abstract

3D mapping systems are crucial for creating digital representations of physical environments, widely used in autonomous robot navigation, 3D visualization, and AR/VR. This paper focuses on OctoMap, a leading 3D mapping framework using an octree-based structure for spatial efficiency. However, OctoMap's performance is limited by slow updates due to costly memory accesses. We introduce OctoCache, a software system that accelerates OctoMap through (1) optimized cache memory access, (2) refined voxel ordering, and (3) workflow parallelization. OctoCache achieves speedups of 45.63%~88.01% in 3D environment construction tasks compared to standard OctoMap. Deployed in UAV navigation scenarios, OctoCache demonstrates up to 3.02× speedup and reduces mission completion time by up to 28%. These results highlight OctoCache's potential to enhance 3D mapping efficiency in autonomous navigation, advancing robotics and environmental modeling.

CCS Concepts: • **Computer systems organization** → **Real-time systems**.

Keywords: Caching, Mapping Systems, IoT, UAV Autonomy, OctoMap

ACM Reference Format:
Peiqing Chen, Minghao Li, Zishen Wan, Yu-Shun Hsiao, Minlan Yu, Vijay Janapa Reddi, and Zaoxing Liu. 2025. OctoCache: Caching Voxels for Accelerating 3D Occupancy Mapping in Autonomous Systems. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3676641.3716263



This work is licensed under a Creative Commons Attribution International 4.0 License.

ASPLOS '25, March 30-April 3, 2025, Rotterdam, Netherlands
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1079-7/2025/03.
https://doi.org/10.1145/3676641.3716263

1 Introduction

3D mapping systems serve as digital representations of 3D space, capturing real-world settings or objects to facilitate the creation of intricate models and immersive experiences. They are fundamental to various fields, including robotics, computer graphics, and 3D vision applications [7, 13, 21, 23, 37, 38]. For example, in robotic applications, 3D mapping systems typically gather data from sensors like LiDAR and depth cameras that continuously scan the surroundings. This data is periodically collected, processed, and integrated into a 3D representation model (i.e., maps).

Currently, OctoMap [26] and its derivatives [12, 16, 40, 43] have emerged as a widely used mapping system in autonomous navigation [5, 41, 57, 58], search and rescue operations [50, 51], and scanning and mapping tasks [9, 52]. By leveraging OctoMap's clear delineation of occupied, free, and unknown spaces, robot/UAV applications can achieve safe, collision-free navigation. Furthermore, thanks to its memory-efficient internal representation of extensive 3D environments, OctoMap is suitable for various edge devices with limited memory capacity.

Although there has been notable progress in deploying OctoMap across various hardware and software platforms [28, 60], the OctoMap software continues to be a significant performance bottleneck in the entire autonomous navigation pipeline. Our experiments with MavBench [8] show that OctoMap can consume up to 72% of the total runtime in a realistic UAV autonomous navigation workflow. The primary bottleneck is due to its distinctive octree structure. To represent spatial occupancy, OctoMap seamlessly

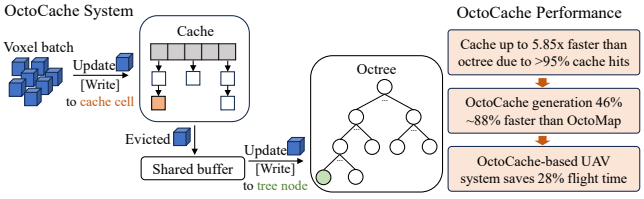


Figure 1. OctoCache Overview. faster voxel update by generating high (e.g., >95%) cache hits and fewer (e.g., 0.125X) memory visits in the cache compared with octree.

divides the entire space into uniformly sized, fine-grained voxels[1]. During a voxel update, OctoMap executes a root-to-leaf round-trip traversal in its octree to find the corresponding leaf node and update its occupancy information. This process involves multiple memory accesses, resulting in considerable latency, particularly when using high-resolution or long-range sensors for mapping and navigation. Moreover, robot navigation requires multiple queries to the voxel occupancy information, which cannot proceed until OctoMap finishes the voxel updates. As queries occur after the updates are completed, a slow update can result in Octomap missing the deadline (e.g., UAVs want to detect an oncoming obstacle and alter the path to avoid collisions). When a slow update to OctoMap occurs, UAVs may reduce their flight speed, leading to task inefficiencies and battery drains [32, 46].

We tackle the software optimization problem in this paper to ensure mapping compatibility with various edge devices [47–49]. Ideally, an OctoMap software platform should achieve: (1) **low mapping system update latency**, enabling faster navigation queries to improve robot agility in obstacle detection; (2) **query consistency** that guarantees same query result and APIs as the vanilla OctoMap; and 3) **low extra resource overhead** (i.e., CPU and memory footprints) for running on resource-constrained edge devices such as NVIDIA Jetson TX2 [47].

In this paper, we introduce OctoCache, an innovative caching layer built on top of OctoMap to enhance the runtime efficiency of OctoMap. Driven by the significant duplication of input voxels, the primary concept of OctoCache to reduce voxel update latency is by **caching recently accessed voxels and their occupancy values before reaching the octree on the critical path.** As illustrated in Figure 1, we initially construct a flattened, table-based cache to handle all voxel updates, requiring fewer memory accesses compared to the costly octree. Also, we arrange the evicted voxels from the cache using the Morton Code ordering[2] [54] of their 3D coordinates and then update them to the octree. This is because different orderings of the same voxel set result in varying levels of data locality and lead to different total update times to the octree. Furthermore, we demonstrate that the Morton Code order can optimally leverage data locality and minimize the overall update time. Lastly, we postpone the octree update process to another CPU thread and execute it concurrently with other functions. This enables queries to experience lower latency as they do not need to wait for the octree update of voxels evicted from the cache.

To guarantee query consistency, the cache always stores voxel coordinates with the accumulated occupancy value (as a mini "octree"), instead of the most recent occupancy values. Thus, the cache can provide correct results upon cache hits, and any query upon a cache miss is redirected to the back-end octree. Also, we use one mutex to ensure that octree reads and octree writes are mutually exclusive in time, which eliminates the possibility of data racing. To guarantee resource efficiency, we evict records from the cache bucket size beyond a threshold. Such eviction of outdated voxel records from the cache ensures that the cache memory overhead remains below a threshold.

We evaluate the performance of OctoCache on a UAV simulation platform [8] using different UAV types, environments, mapping resolutions, and sensing ranges to test its capabilities comprehensively. Our evaluation ensures robust results applicable to various real-world scenarios. We conduct experiments on a Jetson TX2 [47], an edge computing platform widely used for UAV applications due to its balance of power efficiency and computational capability. Our results show that OctoCache improves OctoMap runtime by 45.63% to 88.01%, highlighting OctoCache's ability in handling varying complexities and scales of mapping tasks. For example, dealing with different mapping resolutions, sensing ranges, and obstacle densities. With the OctoCache-powered mapping system in the pipeline, the task completion time is reduced by up to 28%, allowing the UAV to achieve greater agility through faster computation and improve efficiency in time-sensitive missions such as search and rescue or surveillance.

In summary, we present three main contributions:

- Introduction of a general software cache method to accelerate OctoMap updates while ensuring query consistency and efficient resource utilization.
- Theoretical demonstration that the sequential order based on Morton Code optimally supports voxel insertion into the octree and leverages the cache to reorganize the eviction voxel sequences.
- Integration of OctoCache into the complete autonomous navigation process of UAVs.

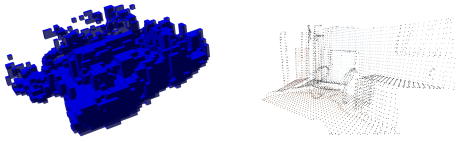
We have open-sourced our code at [1].

2 Background and Related Work

We start with an overview of 3D mapping systems. We then discuss the details of OctoMap and the related work.

[1] A voxel is a 3D cube in space.

[2] Morton Code transforms 3D integer coordinates into a single-dimensional integer. A Morton Code order means arranging voxels in ascending order based on their individual Morton Code.



(a) Voxel grid map (b) Point cloud

Figure 2. Visualization of two dense map examples.

2.1 3D Mapping Systems

Dense map and sparse map: Mapping systems opt for either dense or sparse maps based on their ability to represent the complete 3D environment. Examples of dense maps include voxel grid maps (Figure 2(a)) [15, 35], point clouds (Figure 2(b)) [20, 53], etc. Dense maps facilitate accurate collision detection and motion planning by providing comprehensive spatial information.

Conversely, sparse maps such as SLAM [4, 21, 36, 44, 55], 3D meshes [11, 27] and elevation maps [19, 33, 34] lack information about the entire 3D space, which facilitates faster surface feature extraction and design processes [7, 10, 23, 25, 37]. As a type of voxel grid map, OctoMap (and its variants) are preferred in robot autonomy applications due to their ability to accurately represent occupied, free, and unknown spaces simultaneously.

Mapping systems in autonomous robot navigation workflow (Figure 3): Autonomous robot workflows consist of three stages: *perception*, *planning*, and *control*. In the perception stage, sensor data (e.g., LiDAR, depth cameras) is converted into point clouds and voxel information, which updates the mapping system. The planning stage generates collision-free paths by querying the map, checking voxels along potential trajectories for obstacles. The control stage adjusts rotors to follow the planned trajectory. This cyclic process ensures continuous environmental updates as the robot/UAV moves. Compute latency in this workflow is critical for task efficiency [22, 24, 30, 56]. The mapping system significantly impacts overall performance. If a mapping system update is too slow, the queries in the planning stage will have to be delayed, and thus a robot/UAV will take a longer time to detect an obstacle. To ensure collision avoidance, UAVs typically will reduce their flight speed to accommodate the slow update of the mapping system, resulting in increased mission completion time and higher battery/energy usage.

2.2 OctoMap

OctoMap [26] is a popular dense map that uses voxels with occupancy values to represent spatial occupancy. Each voxel has a central coordinate $< x, y, z >$ and side length d. The mapping boundary is a 3D cubic space centered at $< 0, 0, 0 >$ with side length D. The entire cubic space is considered as one large voxel. The voxel is recursively decomposed into 8 smaller voxels with side lengths halved. For example, this large voxel is decomposed into 8 smaller voxels

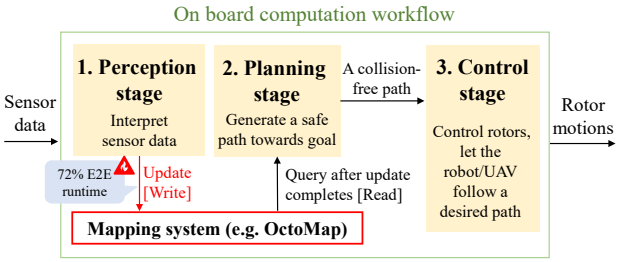


Figure 3. A typical robot autonomous navigation pipeline. The mapping system can take up to 72% of the end-to-end runtime [8].

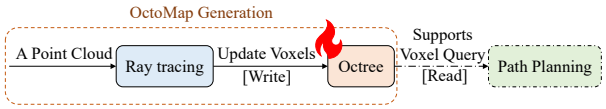


Figure 4. Workflow of OctoMap. OctoMap can support voxel queries only after the current generation process finishes.

with side length $\frac{D}{2}$ and centered at $< \pm\frac{D}{4}, \pm\frac{D}{4}, \pm\frac{D}{4} >$ respectively. The decomposition process applies to each voxel until its side length reaches a predefined mapping resolution (e.g., 0.05m). In OctoMap, each voxel v is stored along with a float-type occupancy value $occupancy(v)$ (i.e., *logodds* as specified in OctoMap [26]) denoting the probability that this space is occupied by an obstacle. OctoMap sets $occupancy(v)$ bounded by min_{occ} and max_{occ} so that it can deal with dynamic environments [26]. There is a pre-defined threshold t where $occupancy(v) \geq t$ denotes an occupied voxel, while $occupancy(v) < t$ denotes a free voxel. The initial occupancy value is set to t.

The **OctoMap workflow** has two main components: *ray tracing* and *octree*, as shown in Figure 4. When the sensor delivers a new point cloud[3], OctoMap performs ray tracing and then updates voxels to the octree. The ray tracing function[4] converts the point cloud into a voxel batch of occupied and free voxels based on the obstacle surface information, as detailed in Section 3.1. In robot/UAV applications, following the OctoMap process, there are functions such as path planning functions that involve querying specific voxel occupancy. OctoMap supports queries from the octree data structure. Notably, if there is an ongoing OctoMap generation process, the query must wait until it finishes. This guarantees the query is always performed on the latest mapping information. Meanwhile, it also renders a high latency for queries.

Octree [39, 59] is a memory-efficient data structure to store and represent multiresolution voxel information. There are three types of nodes in octree (Figure 5): occupied, free, and null. An occupied node represents a voxel v with $occupancy(v) \geq t$, while a free node represents a voxel v with $occupancy(v) < t$. The null node represents a voxel entirely in the unknown space and thus has no occupancy

[3] A point cloud is a large set of 3D coordinates. Each coordinate represents a sampled point on the obstacle surface scanned by the sensor.
[4] Ray tracing is performed towards each point in the point cloud, usually from a top-left to bottom-down order.

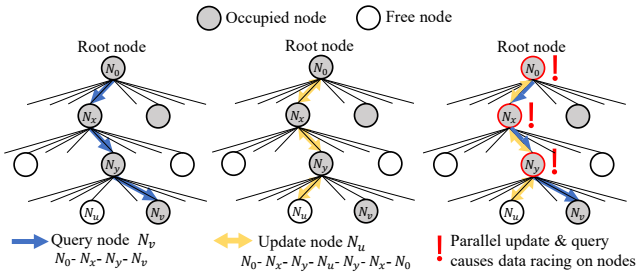
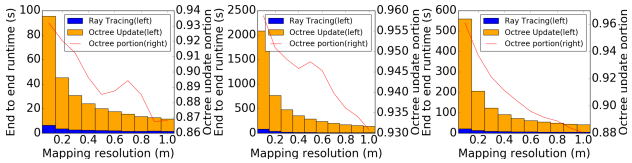


Figure 5. The node (memory) visit pattern of querying and updating in the octree. Both demand a root-to-leaf traversal and cannot be run in parallel due to data racing.



(a) FR-079 corridor (b) Freiburg Campus (c) New College

Figure 6. OctoMap generation workflow decomposition on 3 public datasets [2]. The octree update is the major bottleneck, especially at high resolutions.

Table 1. Existing work and their features.

Related Work	Hardware/ Software	Addressing Octree bottleneck	Resource efficiency	Flexibility
OMU [28]	Hardware	✓	✓	×
OctoMap-RT [43] NanoMap [18] GPU OctoMap [42]	Hardware	×	✓	✓
Naive software parallelization	Software	×	✓	✓
VoxelCache [29]	Software	×	✓	✓
Skimap [12]	Software	✓	×	✓
OctoCache (Ours)	**Software**	✓	✓	✓

value. Each node corresponds to a voxel in the 3D space. The root node N_0 represents the entire cubic mapping space, which is the largest voxel. Each node apart from the leaf nodes has 8 children, denoting 8 smaller voxels split from itself. The occupancy value of each node equals the maximum among its 8 children. To save memory, a node will have all its children pruned if they have the same occupancy value.

Octree updates and queries both require root-to-leaf node traversal in the octree in the worst case (Figure 5). To query voxel v, OctoMap computes the location of node N_v based on v's 3D coordinate, then traverses from the root node to N_v and returns a boolean value based on whether $occupancy(v) < t$. This query path visits nodes N_0, N_x, N_y, and N_v. To update voxel u, OctoMap first searches for node N_u. The occupancy value is updated as $max(occupancy(v) - \delta_{free}, min_{occ})$ if u is free, or $min(occupancy(v) + \delta_{occupied}, max_{occ})$ if u is occupied. Here, δ_{free} and $\delta_{occupied}$ are predefined heuristics reflecting the map's sensitivity to free or occupied voxel updates. After updating N_u, a trace-back from N_u to the root node N_0 updates all ancestor nodes, with each ancestor's occupancy value based on the maximum among its children. This update path sequentially visits nodes N_0, N_x, N_y, N_u, N_y, N_x, and N_0. The root-to-leaf traversal constraint prevents parallelizing updates or queries, as simultaneous operations on the same node could cause data races. For example, querying v while updating u may race on nodes N_0, N_x, and N_y.

2.3 Existing Efforts in Accelerating OctoMap

Table 1 summarizes existing efforts to speed up OctoMap. Our work uniquely advances prior art by simultaneously addressing the octree bottleneck in terms of resource efficiency, flexibility, and the construction of a dense map.

Hardware acceleration: Several hardware accelerators solutions have been proposed to accelerate OctoMap. For instance, **OMU** [28] is a hardware accelerator designed to enhance parallel ray tracing and octree updates. Nevertheless, these solutions require dedicated special hardware, limiting the scalability needed to accommodate the diverse computation platforms employed in different robots/UAVs. Other hardware efforts focus on accelerating ray tracing with GPUs, including **NanoMap [18], OctoMap-RT [43], and GPU-accelerated OctoMap** [42]. However, we identify that the primary bottleneck of the mapping system lies in voxel updates (as discussed in Section 3), and accelerating ray tracing has limited performance improvements.

Software optimizations: The need for flexibility drives improvement in the software design and implementation of OctoMap. The effort closest in spirit to our work is **VoxelCache** [29]. VoxelCache introduces an additional indexing data structure to enhance octree query and update speeds by quickly locating voxels from the index. However, VoxelCache does not address the key octree performance bottleneck, as queries still need to wait for all voxel updates in the octree to complete. For each voxel update, an expensive octree traversal is still required. In comparison, our objective is to quickly cache the recent voxel and allow queries to be served immediately thereafter, before these updates are completed in the octree. Moreover, our caching creates an opportunity to optimize the order of voxel updates to the octree for reduced total update time. Alternative software solutions strive to substitute the octree with more computationally efficient data structures [6, 12, 45]. However, these improvements are undesired (e.g., Skimap [12] utilized a tree+linked list data structure which demands a much higher memory overhead.).

Existing software solutions fall short in addressing the performance bottlenecks of OctoMap. Our goal is to revisit the software optimizations to address the key bottleneck in OctoMap while remaining flexible and resource-efficient.

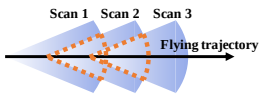


Figure 7. High overlap (Orange dotted area) between each 2 continuous scans along the flight trajectory.

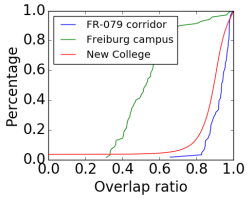


Figure 8. Overlap ratio between 3 update batches.

3 OctoMap Bottleneck Analysis

We analyze the bottlenecks of OctoMap by running experiments on an NVIDIA Jetson TX2 device using three public datasets [2]. As depicted in Figure 6, we demonstrate the runtime breakdown of OctoMap: octree update is a main bottleneck, where the entire update time accounts for more than 86% of the total OctoMap runtime. In particular, when the mapping resolution increases (e.g., 0.1m or higher resolution is generally used for indoor environments), this bottleneck can reach as high as 93%~96%. Moreover, when hardware accelerators that improve ray tracing speed are in place, the octree update accounts for 99% of the total runtime [28]. The slow update of the mapping system impedes fast autonomous navigation for robots/UAVs. To understand why the octree update is slow, we have two key observations based on our profiling results.

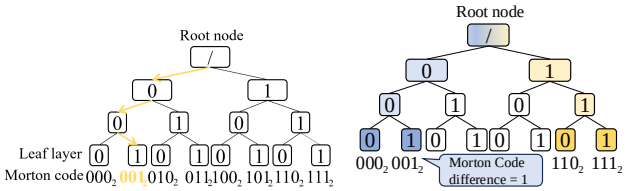
- **Numerous duplicated voxel updates:** The octree update process involves repeatedly updating the same voxels multiple times, leading to inefficient utilization of computational resources.
- **Slow updates for each voxel:** The update operation for individual voxels within the octree is computationally expensive, further exacerbating the overall slowdown of the octree update.

These observations provide insight into potential avenues for optimizing the OctoMap algorithm and alleviating the performance bottleneck. In the remainder of the section, we analyze these observations and discuss the design opportunities.

3.1 High Duplication in Voxel Updates

The voxel duplication exists both intra each update batch and inter consecutive update batches.

High voxel duplication intra each update batch stems from the creation of voxels via ray tracing. In ray tracing, a straight ray shoots from the sensor $< x_0, y_0, z_0 >$ to each coordinate $< x', y', z' >$ in the point cloud. Each voxel on this straight line is free, while the last voxel containing $< x', y', z' >$ is occupied as this coordinate denotes an obstacle surface. Since these rays form a conical shape and intersect some common voxels near the source, they result in numerous duplicate free voxels close to $< x_0, y_0, z_0 >$. However, since the point-cloud density is significantly higher than the voxel resolution, multiple points fall within the same voxel. During ray tracing, each of these points is converted into an occupied voxel as they represent samples on the obstacle surface. Consequently, these points lead to multiple duplicate occupied voxels. The duplication rate within each update batch varies from 2.78 to 31.32×.



(a) Morton Code (MC) based on root-to-leaf path.

(b) Smaller MC differences (same colored leaf nodes) indicate more common ancestors.

Figure 9. Morton code [54] example for a binary tree. The root node is empty and each child node is marked as 0 or 1 depending on whether it's a left or a right child node. The Morton code is generated by combining tree node codes from a root-to-leaf order in binary. Tree nodes with small Morton code differences share more common ancestors.

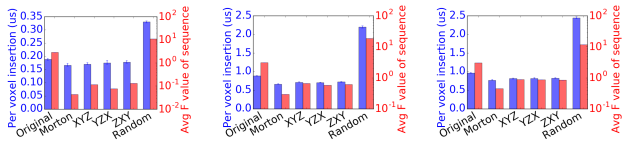
High voxel overlap inter consecutive update batches originates from the sensor's scanning pattern. Assuming a UAV is using OctoMap, Figure 7 shows a general scanning pattern when the UAV flies along a given trajectory. To ensure that the UAV can detect newly emerged or incoming obstacles along the trajectory, the UAV carries out constant scanning during flight time, instead of just scanning anew when reaching the current mapping boundary. This usually results in a high overlap between the voxels generated by continuous scans. Figure 8 shows the cumulative distribution of the overlap ratio of non-duplicate voxels between three consecutive updates. For two of the three datasets, over 80% of voxels in each update are duplicated from the previous three updates. For the *Freiburg campus* dataset, this number drops to 40%, but it remains reasonably high. When considering duplicate voxels, this overlap ratio will be even higher.

Design opportunity 1: The aforementioned observations present a chance to design **a fast cache** as an additional layer preceding the octree, aimed at producing cache hits for repeated voxel updates. Searching and updating a voxel within the cache should require significantly fewer memory accesses compared to the deep octree. Furthermore, by allocating a cache of appropriate size and retaining the duplicated voxels within it, numerous cache hits can be generated, substantially reducing the need for searches in the octree.

3.2 Slow Per-Voxel Update

Apart from the high voxel volume, another bottleneck is the significant delay per voxel update, which is caused by frequent memory accesses and an inefficient order of voxel updates in the octree.

Numerous memory accesses for each voxel update: Updating a voxel requires a root node to leaf node round-trip visit in the octree, involving up to 32 memory accesses for a standard



(a) FR-079 corridor (b) Freiburg Campus (c) New College

Figure 10. Per-voxel octree update time overhead based on different voxel orders. The per-voxel insertion speed has a positive correlation with the $\mathcal{F}$ value in Section 4.3. Ordering voxels by Morton Code gives the fastest insertion speed to the octree.

16-level octree. This situation deteriorates further when the map resolution increases and the octree depth expands.

Voxel sequence order affects octree update speed: Since each voxel update requires a traversal from the root to a leaf node, an optimal voxel ordering can leverage the data locality of tree nodes more effectively when updating the octree. Figure 9 illustrates a binary tree, which serves as a simplified model for the octree. Leaf nodes with smaller Morton Code bit-value difference (e.g., the 2 blue nodes in Figure 9(b)) share more common ancestor nodes, and can result in more CPU cache hits if they are updated together to the octree. In contrast, updating a blue node and a yellow node together does not give as many cache hits. Consequently, grouping voxels with a shorter "shortest-path distance in the tree" can better exploit data locality. In Section 4.3, we prove that inserting the voxels into the octree by their Morton Code [54] order provides optimal data locality and thus optimizes the tree update speed.

In a NVIDIA Jetson TX2 testbed, we evaluate tree update performances with various voxel update orders, as shown in Figure 10. We have evaluated with orders of 1) random shuffle, 2) sorting of voxels by their X, Y, and Z coordinates, respectively[5], 3) Morton code order of their 3D coordinates, and 4) the original order in OctoMap generated from ray tracing. We insert 5M voxels into an empty octree and compute the mean value with the [10%,90%] confidence interval over 100 runs on 3 public datasets. All results indicate a positive correlation between the data locality (i.e., $\mathcal{F}$ defined in Section 4.3) of the voxel sequence and per-voxel insertion speed. Across all 3 datasets, voxel sequences ordered by Morton Code consistently show the best insertion speeds, ranging from 1.38×~1.34× faster than the original order in OctoMap and 1.97×~3.32× faster than the worst case where all voxels are randomly ordered.

Design opportunity 2: Our cache design enables an opportunity to **arrange specific eviction orders of these voxels before updating into the octree**. Our result show leveraging the cache to reorder the voxel sequence by Morton Code can accelerate the slow octree update procedure.

[5]e.g., XYZ order sorts all voxels first by X coordinates. Voxels with the same X are further sorted by Y coordinates, and then Z.

4 OctoCache

This section details the design of OctoCache. First, we outline the primary design objectives for the system. Then, we discuss a strawman serial hash-based OctoCache design to demonstrate the basic cache workflow and associated data structures. Next, we present an improved caching strategy using the Morton code as voxel ordering. Lastly, we describe a multithread OctoCache to enhance performance by paralleling the cache and the tree in different threads.

4.1 Design Goals and Key Ideas

Our objective is to develop a cache to reduce the number of duplicated voxels and optimize the voxel update order in the octree. Specifically, we aim to address three design challenges.

- **Low map update (write) latency:** The low latency requirement comes in two aspects. Firstly, there should be a small wait time until the queries on the current voxel batch can be served. This enables quicker detection of obstacles in the current scan data. Secondly, the entire mapping system generation process should also be faster. As the mapping system keeps taking in point clouds from the sensor, a faster mapping system pipeline can support a higher rate of sensor input.

  Key ideas: Our approaches are three-fold: (1) We build a cache to store newly arrived voxels and the occupancy values. Once the cache update finishes, the cache supports voxel queries from the path planner without the need to wait for the completion of octree updates. (2) In cache eviction, we use Morton Code to organize voxel orders for faster updates to the octree. We index the voxels by the Morton Code of their 3D coordinates during cache insertion and evict outdated voxels sequentially from the cache. Thus these evicted voxels can form a Morton Code order which best speeds up octree update (Section 4.3). (3) We run the octree updates with ray tracing and cache eviction in parallel. This can give an even shorter mapping system generation time by overlapping the workflow of consecutive update batches (Section 4.4).
- **Query (Read) consistency:** The voxel query result and query interface from OctoCache should match that from OctoMap for streamline integration. Our new design shall return the same occupancy value when querying any voxel, and the query API should be consistent so that other functions that demand the voxel query from the storage can remain unchanged.

  Key ideas: The cache maintains the accumulated occupancy values of the voxels in the same way as the octree does. This ensures the cache can serve queries independently without octree upon cache hits, different from other cache-based index services [29]. Meanwhile, any voxel evicted from the cache will overwrite its occupancy

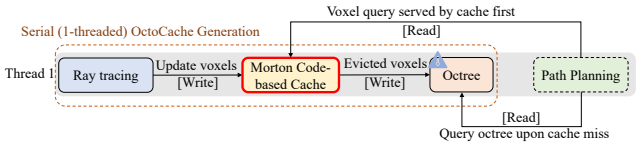


Figure 11. Workflow of serial OctoCache.

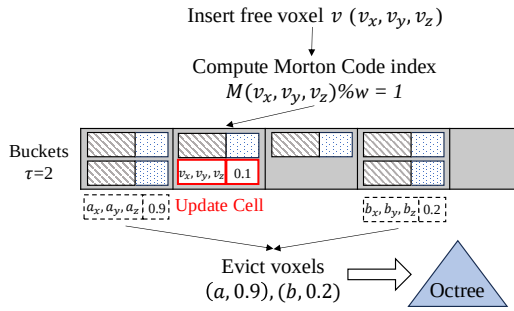


Figure 12. Cache insertion (voxel v) and eviction (voxel a, b) example of a cache with $w = 5$ buckets and eviction threshold $\tau = 2$.

value to the octree to ensure that octree can serve consistent query upon cache miss (Section 4.2). We use a mutex to schedule the reads and writes of octree. The mutex is locked when thread 2 starts executing octree update. The lock is not released until thread 2 finishes. Thread 1 locks the mutex and starts executing cache insertion, and releases it when the query finishes. This eliminates the possibility of data races.

- **Low extra resource overhead:** The mapping systems usually run on edge devices on robots and UAVs with limited memory and computation resources. For example, an NVIDIA Jetson TX2 [47] has 8GB RAM and 6 CPU cores. The new design should incur reasonable extra memory and CPU resources.

  Key ideas: To control the memory overhead, we set a predefined threshold to limit the maximum voxel records in the cache after eviction. This allows us to control the memory overhead throughout runtime (Section 4.2). For the CPU overhead, OctoCache requires only one extra CPU core, which is available under most robot/UAV applications (Section 4.4).

4.2 Strawman Serial OctoCache

We start with a complete workflow of a strawman OctoMap design as depicted in Figure 11, including insertion, query, and eviction policies. The detailed runtime analysis of this workflow is shown in Figure 13(a). With the help of the cache, we do not need to wait for the slow octree update for querying the current batch. Instead, the latency is the cache insertion time, which is fast. We further show how to utilize a dedicated cache design to maintain query consistency and low memory overhead.

4.2.1 Cache insertion and query: The key idea is to record the accumulated occupancy value together with the voxel ID (3D coordinates) in the cache data structure. During the cache insertion, the incoming voxel will have its occupancy value added to the accumulated value recorded in the cache. Therefore, the cache itself can serve a consistent query result as the vanilla Octomap returns. Additionally, any outdated voxel evicted from the cache will have its occupancy value overwritten into the backend octree. This ensures query consistency upon any cache miss.

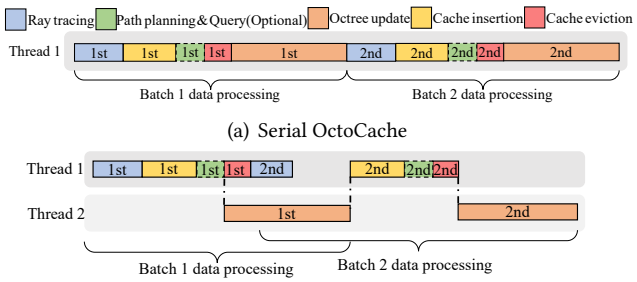
Implementation details (Figure 12): The cache contains an array of w buckets. Each bucket contains a vector of cells. Each cell stores a voxel's 3 coordinates *(uint8_t type)* and *one float-type* accumulated occupancy value. In cache insertion, we map an incoming voxel v to a bucket with a hash function $h(v)\%w$ where w is the width of the bucket array. We set w always as a power of 2 to accelerate the mod operation. After locating the bucket, we search through its cells to determine if we have a cache hit $v, occupancy(v)$. Upon a cache hit, we update the occupancy value as $max(occupancy(v) - \delta_{free}, min_{occ})$ if v is free, or $min(occupancy(v) + \delta_{occupied}, max_{occ})$ if v is occupied. Upon a cache miss, we first search the octree to verify if a record of v exists. If found, we denote the occupancy value obtained as $octree(v)$, which is the accumulated occupancy value of voxel v. We retrieve $(v, octree(v))$ from the octree, write it into a new cell in the bucket, and update it by δ_{free} or $\delta_{occupied}$. Otherwise, if voxel v is not found in the octree, we will simply create a new cell in this bucket and write in $(v, t + \delta_{occupied})$ or $(v, t - \delta_{free})$, depending on v's occupancy status.

4.2.2 Cache eviction: The key idea to control the memory overhead is defining a threshold τ for bucket size upon cache initialization. τ denotes the maximum number of distinct voxels per bucket after processing. In the insertion process, we allow each bucket to hold more than τ records because multiple records can collide into the same bucket. As the size of the cache will not grow beyond an update batch during the cache insertion process, the extra memory overhead is still under control.

Implementation details (Figure 12): Upon cache eviction, we search through each bucket in the cache. If a bucket size is above τ (i.e., holding more than τ voxel recorders), we evict the earliest inserted records until the bucket size goes down to τ. All evicted voxels with their accumulated occupancy values form a batch and will be written into the octree.

4.3 Morton Code-based OctoCache

The previous design ensures a faster cache update time than the octree. However, it does not guarantee a shorter mapping system generation time, which includes updating evicted voxels into the octree. Updating the octree with fewer voxels evicted from the cache may still take longer than the octree update in the basic OctoMap workflow (some (bad) random



(a) Serial OctoCache

(b) Parallel OctoCache

Figure 13. Serial and parallel OctoCache runtime decomposition. Path planning & query (Optional) appears in robot autonomy navigation but not in 3D environment construction. The bar size mimics runtime portions.

voxel ordering can potentially take a longer time, as previously discussed in Section 3.2 and Figure 10). Hence, we **use Morton Code-based cache policy to order evicted voxels based on their Morton Code, which facilitates octree update.** The key idea is to index the voxel based on the Morton Code of its 3D coordinates to locate a bucket in the cache. Meanwhile, in the cache eviction stage, we still apply the sequential eviction policy to evict outdated voxels from the cache. Therefore, we can let the evicted voxels form a Morton Code order which best facilitates the octree update.

Below, we first give a proof sketch of why organizing voxels with their Morton Code of 3D coordinates gives the fastest octree update speed. Then we introduce the implementation details.

Intuitively, we use a formula to count the number of common ancestors shared by each two adjacent voxels in the sequence. This formula can be used to estimate how much data locality a voxel sequence can utilize when inserting them into the octree. Because updating a voxel at the finest resolution demands a round-trip root-to-leaf visit in the octree, a higher number of common ancestors shared always implies a faster update speed and vice versa. We provide a proof sketch and defer the detailed proof (*Lemma A2~A6*) to supplementary materials (open sourced at [1]).

Notations: Denote N leaf nodes in a perfect l-level octree[6]: $a_1, a_2, ..., a_N$. Denote $A(a_1, a_2, ...a_n)$ as the closest common ancestor of leaf node $a_1, a_2, ..., a_n$. Denote $D(a, b)$ as the distance between leaf node a and b, which is twice as the distance between a or b and node $A(a, b)$[7]. S represents an ordering of the N leaf nodes.

Main theorem (Morton code optimality): Ordering the leaf nodes $a_1, a_2, ..., a_n$ by the Morton Code of their 3D coordinates is one of the optimal sequence S^* that minimizes the following formula:

$$\mathcal{F}(S) = D(a_1, a_2) + D(a_2, a_3) + ... + D(a_{N-1}, a_N)$$

Proof sketch:

- We give a proof sketch for a uniform depth octree here. We prove that the theorem also holds true for any non-uniform depth of octree, and the proof is deferred to the supplementary materials [1].
- For any three leaf nodes randomly picked, $\{a, b, c\}$, $A(a, b), A(a, c), A(b, c)$ can be at most two different inner nodes. (*Lemma A2*)
- For any three leaf nodes randomly picked, $\{a, b, c\}$, $D(a, b), D(a, c), D(b, c)$ can be at most two different numbers. (*Lemma A3*)
- For any non-leaf nodes a and b at the same level, the distance between any descendent leaf node of a and b is always the same. This value is always larger than the distance between any two descendent leaf nodes of a. (*Lemma A4*)
- For any non-leaf nodes a and b at the same level, the descendant leaf nodes of a and b can only have at most one pair neighboring in an optimal sequence. (*Lemma A5*)
- For each non-leaf node a in the complete tree, all descendent leaf nodes of a should always be arranged together in an optimal sequence S^*. This holds true for non-leaf nodes at any level in the tree. (*Lemma A6*)
- Voxel ordering that meets *Lemma A6* all give the same $\mathcal{F}$ value. Ordering the voxels by their Morton Code is one of these orderings and thus is one optimal sequence.

Implementation details: To insert voxel v, we replace the location function $H(v)\%w$ with $M(v)\%w$, where the Morton code $M(v)$ is calculated from v_x, v_y, and v_z. Here, we give a detailed example of Morton code calculation. For voxel v with coordinates $(1, 5, 3)$, we first calculate the binary value of v_x, v_y and v_z as $001_2, 101_2$, and 011_2 respectively. Afterward, we concatenate all the bits together from the highest to the lowest in the order of x, y, z, which forms 000_2 for the highest level bits, 110 for the mid-level bits, and 111_2 for the lowest level bits. Putting them together, we have 000110111_2. Eventually, we convert this binary number into a decimal $M(v)$=167 as the final output. The insertion process concludes by updating voxel v and its occupancy value into the 167^{th} bucket.

4.4 Parallel OctoCache: Further Speed Up

Although the cache saves the number of voxels updated to the octree by serving a large number of cache hits, we still observe that the octree update is a major bottleneck on the critical path (see runtime profiling in Section 6.2). As reasoned in Section 2.3, deploying multiple CPU cores to parallelize octree does not help due to data imbalance. As shown in Figure 13(b), we **use another CPU core to run octree update in parallel with ray tracing and cache eviction**, which will not cause data racing. The key idea is to use a mutex to partition operations involving octree reads and writes. Specifically, cache insertions and queries require memory reads from octree. An octree update involves updating the octree nodes as memory writes. Cache eviction and

[6] All leaf nodes are on the same level in a perfect octree.

[7] In a perfect octree, the distance between node a and node $A(a, b)$ equals the distance between node b and node $A(a, b)$

ray tracing operations have no access to octree data structure. This allows executing octree updates alongside cache eviction and ray tracing without a data racing issue. Moreover, the octree update is usually slower than cache eviction and ray tracing combined even after duplication check and a better voxel ordering. Therefore, this parallelization strategy can achieve the best performance gain.

Implementation details (Figure 14): We build a 2-threaded design and move the octree update function off the critical path (thread 1) to another thread 2. To minimize query latency, queries are executed immediately after cache insertion completes. The cache eviction and octree insertion happen after the query of the current batch data finishes. To correctly undertake the evicted voxel data from the cache, we build a shared buffer between the two threads to store voxel coordinates and occupancy values. All voxels evicted from the cache are enqueued into this shared buffer and dequeued on thread 2 for octree updates. We implement the shared buffer using a readerwriterqueue, which supports enqueue from one thread and dequeue from another thread [14].

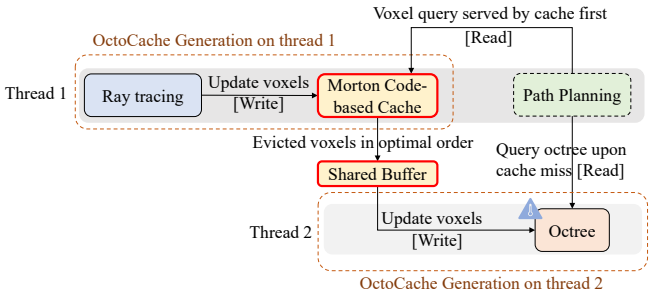


Figure 14. Workflow of parallel OctoCache.

The parallel OctoCache design benefits the runtime on the critical thread (thread 1) by offloading the octree update function to thread 2. However, this parallelism introduces two synchronization overheads. 1) Thread 1 can potentially wait for thread 2 (Figure 13(b)): To eliminate data race, the cache insertion of the incoming batch (thread 1) needs to wait for the octree update (thread 2) to finish. This can cause a waiting time on thread 1 if the octree update takes a longer time than the total runtime of ray tracing and cache eviction. Based on our evaluation in Section 6.2.2, a larger cache with more cache hits can decrease this waiting time. 2) Inter-thread data transmission via the shared buffer has queuing latency (i.e., enqueue and dequeue): The buffer enqueue occurs during cache eviction on thread 1 where outdated voxels and their accumulated occupancy value are written into the buffer. The buffer dequeue happens on thread 2 where voxels with their accumulated occupancy value are read from the buffer and written into the octree. Our experiments show that this queue delay is negligible (Section 6.2.2).

5 Experimental Setup

This section details the experimental setup for UAV autonomous navigation simulation and local 3D environment construction. Both evaluations were conducted on a Jetson TX2 [47] with power fixed at 15 watts. The Jetson TX2, a high-end embedded platform from Nvidia, features a Quad ARM CPU, an embedded GPU, and 8GB of SRAM. We evaluated four mapping systems: OctoMap, OctoCache, OctoMap-RT [43][8], and OctoCache-RT. OctoMap-RT employs a distinct ray-tracing method that eliminates duplicated voxels, while its octree insertion process remains identical to OctoMap. For fair comparison, we compared OctoMap-RT and OctoCache-RT end-to-end, with OctoCache-RT using the same ray-tracing method as OctoMap-RT.

5.1 UAV Autonomous Navigation

We simulate the UAV autonomous navigation task using MAVBench [8], a closed-loop simulator with an end-to-end application benchmark suite for UAV system benchmarking. The operating scenarios, sensors, and UAV kinematics and dynamics are simulated using the Unreal game engine [17] on a Windows laptop. The entire computation workflow, including perception, planning, and control, runs on a Jetson TX2. Both computers are connected to the same router for low-latency communication. We exhaustively test the system using *two types of UAVs, four simulation environments, a range of sensing ranges, and various mapping resolutions.*

Two types of UAVs: 1) AscTec Pelican: 1872g weight, rotor pull power 3600N, and 2) DJI Spark: 350g weight, rotor pull power 588N. Both UAVs have 50Hz FPS sensors.

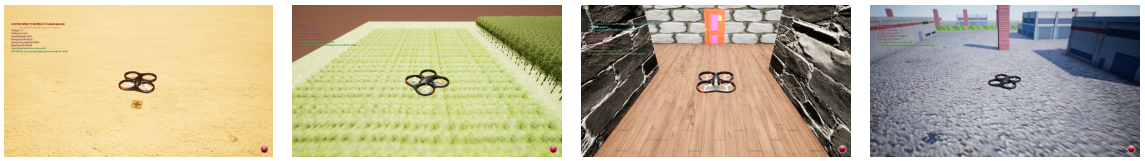
Four simulation environments (Figure 15): *Openland* is a structured outdoor environment with a goal 100m away. *Farm* is an unstructured outdoor environment with a goal 50m away. *Room* is an indoor environment with a goal 12m away. *Factory* is a mixed outdoor and indoor environment with a goal 70m away. We rank the task difficulty as *Room* > *Factory* > *Farm* > *Open land.*

Sensing ranges and mapping resolutions: We use the parameter setting as suggested in [8]. The baseline <sensing range, mapping resolution> for OctoMap vs OctoCache are: Openland <8m,1m>, Farm <4.5m,0.3m>, Room <3m,0.15m>, Factory <6m,0.5m>. As OctoMap-RT and OctoCache-RT have a much shorter runtime and can support even higher resolutions, their baselines are: Openland <8m,0.04m>, Farm <4.5m,0.02m>, Room <3m,0.01m>, Factory <6m,0.03m>. In real-world applications, the sensing range and mapping resolution may vary according to application and robot hardware demands. Therefore, our test runs are based on a range covering the baseline parameter choices.

Cache setup: The cache contains 512K buckets. Each bucket holds at most $\tau = 4$ cells after eviction. Thus, the cache size is constrained to 14MB (each cell contains 3D coordinates with 3 bytes and an occupancy value of 4 bytes). The cache setup is the same for OctoCache and OctoCache-RT.

Evaluation metrics:

[8]Since OctoMap-RT is not open-source, we reimplemented their algorithm on the Jetson TX2 CPU.



(a) Openland (b) Farm (c) Room (d) Factory

Figure 15. The MAVBench testbed includes 4 different environment scenarios, covering both indoor & outdoor, and structured & unstructured environments.

Table 2. Details of OctoMap 3D scan datasets.

Dataset	Point Cloud #	Resolution	Nonduplicate Voxel #	Duplicate Voxel #
FR-079 corridor	66	0.1m	6.26M	196.1M
		0.2m	1.07M	101.1M
		0.4m	1.8M	54.5M
		0.8m	35.1K	29.5M
Freiburg campus	81	0.1m	209.5M	2042M
		0.2m	36.03M	1030.9M
		0.4m	5.8M	525.3M
		0.8m	1.02M	273.1M
New college	92361	0.1m	317.4M	883.6M
		0.2m	95.9M	448.9M
		0.4m	28.2M	231.4M
		0.8m	8.8M	122.8M

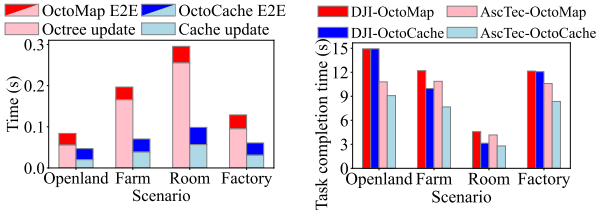
(1) *End-to-end runtime:* The UAV autonomous navigation workflow includes point cloud generation, ray tracing, mapping system update, collision checking, motion planning, etc. During the whole flight time, the end-to-end runtime of each cycle may vary due to different input data volumes. We calculate the average runtime over the entire process.

(2) *UAV flight velocity:* Flight velocity is determined by UAV physics and how fast the UAV can compute. Krishnan et al. give a model [30] to compute the max safe velocity bound using UAV's sensing range, sensor FPS, UAV's weight, rotor pull power, computation platform power, and system end-to-end runtime. Under the same UAV platform, a faster perception stage (i.e., mapping system update) allows the UAVs to react to obstacles in a shorter time and achieve higher safe flight velocities.

(3) *UAV task completion time:* Mission completion time is $\frac{\text{Navigation path distance}}{\text{UAV speed}}$, indicating how soon the UAV can reach the goal safely based on its navigation algorithm. Prior efforts [3, 31] have demonstrated that mission completion time is the critical end-to-end metric that users care about and can reflect the correlation impacts of several factors. It also directly correlates with energy usage because 95% of the UAV energy is consumed by the rotor during the entire flight [30].

5.2 3D Environment Construction

The workflow of 3D environment construction mirrors Figure 4: Multiple point cloud data are read from static files, transformed into voxels, and inserted into the octree/our cache. We compare the performances of OctoMap vs. serial OctoCache vs. parallel OctoCache, and OctoMap-RT vs. serial OctoCache-RT vs. parallel OctoCache-RT.



(a) System end-to-end runtime (b) Task completion time

Figure 16. System end-to-end runtime, max flight velocity, and task completion time comparison between OctoMap / OctoCache based UAV systems.

Datasets: We study three public 3D scan datasets [2] under nine different resolutions: from 0.1m to 0.9m. These static datasets consist of pre-collected point clouds based on a fixed sensing range. Therefore, we can only demonstrate various input data by choosing different mapping resolutions. The details for each dataset are depicted in Table 2.

Cache setup: For fairness of evaluation, we pick an appropriate size of the cache as 3 to 4× of the average number of non-duplicate voxels in each update batch. The bucket size threshold τ=4. The cache set up is the same for OctoCache and OctoCache-RT.

Evaluation metrics: (1) *Runtime:* This includes the total runtime of the mapping system generation (i.e., constructing a 3D map using all 3D scan data), and the decomposition of each function runtime (i.e., ray tracing, octree/cache update, cache eviction). (2) *Cache hit rate:* The cache hit rate is calculated with the average number of cache hits in each update batch.

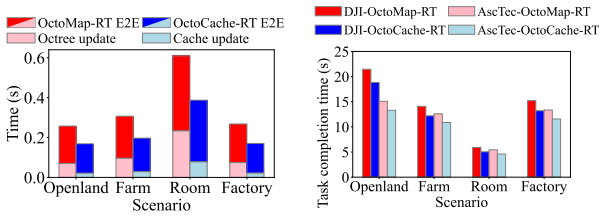
6 Evaluation Results

In general, OctoCache significantly enhances OctoMap performance in both UAV autonomous navigation and 3D environment constructions, especially under high mapping resolution and long sensing range scenarios.

6.1 UAV Autonomous Navigation

We show in a logical order that OctoCache improves the mapping system generation, UAV autonomous system runtime, UAV max safe flight velocity, and task completion time.

6.1.1 End-to-end runtime on UAV autonomous navigation: OctoCache demonstrates universal performance gains over OctoMap across all scenarios, particularly under high-resolution and long-sensing-range



(a) System end-to-end runtime (b) Task completion time

Figure 17. System end-to-end runtime, max flight velocity, and task completion time comparison between OctoMap-RT / OctoCache-RT based UAV systems.

conditions. Figure 16(a) shows that the OctoCache-based system is 1.78×, 3.02×, 2.95×, and 1.98× faster than the OctoMap-based system in end-to-end runtime. Figure 17(a) indicates that the OctoCache-RT-based system is 1.33×, 1.53×, 1.51×, and 1.45× faster than the OctoMap-RT-based system in end-to-end runtime. Figures 18(a) and 18(c) reveal that for high-resolution and long-sensing-range settings, the OctoCache-based system is 2.46× faster than OctoMap with a 4m sensing range and 0.15m resolution, and 3.66× faster with a 3m sensing range and 0.1m resolution. Even under low-resolution and short-sensing-range scenarios, OctoCache outperforms OctoMap, e.g., 1.02× faster with a 2m sensing range and 0.15m resolution, and 0.85× faster with a 3m sensing range and 0.1m resolution. OctoCache-RT consistently outperforms OctoMap-RT, achieving a 37× speedup under a high resolution of 0.01m.

The performance gain is primarily attributed to the saving in mapping system update (the octree/cache update). The gap between the end-to-end runtime and the mapping system runtime includes point cloud generation, ray tracing, collision checking, motion planning, etc. Figure 18(a), 18(c), 19(a), and 19(c) shows that the mapping system update (i.e., octree update) is a major bottleneck in both OctoMap and OctoMap-RT, accounting for up to 92% and 61% of runtime, respectively, in high-resolution and long-sensing-range scenarios. Replacing the octree with a cache improves critical path runtime and reduces waiting latency for subsequent mapping system queries.

6.1.2 Flight speed & mission completion time: Figures 16(b) and 17(b) show task completion times for 4 scenarios, calculated by dividing distance by the max safe flight velocity achievable within computation speed. The OctoCache-based system reduces task completion times by 13%, 27%, 28%, and 19% compared to the OctoMap-based system on an AscTec UAV. OctoCache-RT improves task completion times by 14%, 12%, 13%, and 15% over OctoMap-RT. No improvement is observed for DJI Spark UAV in Openland and Factory environments, as the bottleneck shifts to UAV rotor power rather than computation. Figures 18(b), 18(d), 19(b), and 19(d) show safe flight velocities and task completion times under varying sensing ranges and resolutions. OctoCache achieves up to 1.65× and 1.72× flight velocities of OctoMap, saving 39% and 42% time, respectively. OctoCache-RT is 25% and 17% faster than OctoMap-RT in two scenarios. Additional scenarios are detailed in the supplementary material.

	Ray tracing	Cache insertion	Cache eviction	Octree update	En-queue	De-queue
FR-079	7.248	16.442	0.289	6.860	0.017	0.050
Freiburg campus	80.126	463.252	14.894	285.061	0.834	1.936
New college	11.094	15.188	1.023	15.308	0.229	0.798

Table 3. Inter-thread data transmission overhead (in seconds).

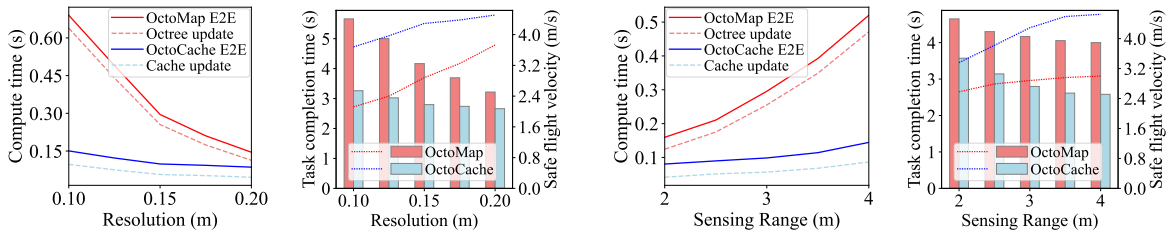
6.2 3D Environment Construction

These evaluations focus on the microbenchmarks: A detailed runtime decomposition allows us to probe into the improvement gains from caching and parallelization.

6.2.1 Total runtime of 3D environment construction: Figure 21 shows the total runtime when using OctoMap, OctoCache, and OctoMap-RT, and OctoCache-RT. Across all 3 datasets and 9 resolutions, we observe a consistent improvement when using OctoCache (Figure 24(a), 24(b), 24(c)). The serial OctoCache has a 1.03× to 2.06× improvement compared with OctoMap under 0.1m resolution. We also see a significant improvement when using parallel OctoCache over serial OctoCache at higher resolutions. When zooming into the resolution range of 0.1 to 0.3m, we see an extra 0.16× to 0.33× gain from the parallel OctoCache over the serial design. The performance gain using OctoCache-RT over OctoMap-RT is also consistent, and is up to 2.51× faster under high resolutions (Figure 21(a), 21(b), 21(c)). The parallel design also worked on OctoCache-RT with an extra 34% gain on a resolution of 0.1m.

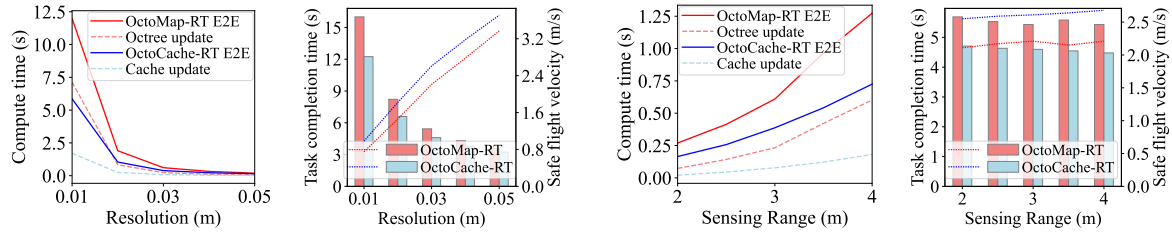
6.2.2 Runtime decomposition: We decompose the runtime into ray tracing, octree update, cache insertion, and cache eviction. Additionally, buffer enqueue (thread 1) and buffer dequeue (thread 2) components handle parallel OctoCache communication. Table 3 shows negligible enqueue and dequeue overhead compared to other components.[9] In OctoCache, thread 1's cache insertion waits for thread 2's octree update to finish, creating a waiting gap (Figure 22(b)). Figure 22 shows cache insertion in OctoCache is 5.85×, 2.57×, and 3.34× faster than octree updates in OctoMap. Thread 2's octree update overhead is only 9.7%, 16.9%, and 23.8% of OctoMap's workflow due to fewer voxel updates handled by the octree. OctoCache-RT's cache is up to 5.12× faster than OctoMap-RT's octree. Thread 2's octree update is faster in OctoCache-RT by using the cache to reduce duplicate octree visits. End-to-end improvement is significant when octree updates dominate in OctoMap-RT. The parallel OctoCache / OctoCache-RT design does not fully benefit from moving the octree bottleneck to thread 2 due to synchronization overhead, specifically the waiting time (i.e., gap) on thread

[9]Parameter settings for Table 3 match Figure 22.



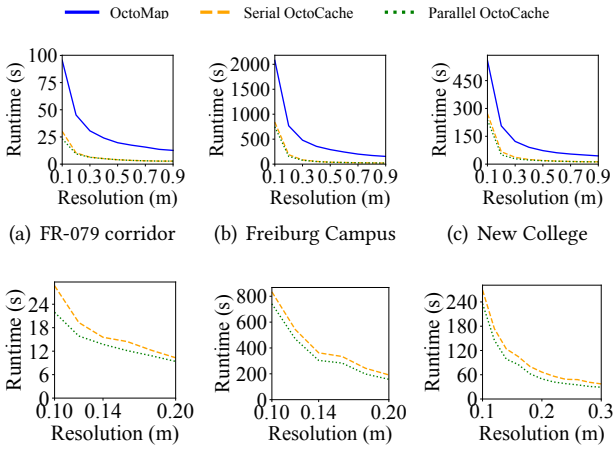
(a) Sensing range=3m, runtime (b) Sensing range=3m, velocity (line) and flight time (block) (c) Resolution=0.15m, runtime (d) Resolution=0.15m, velocity (line) and flight time (block)

Figure 18. OctoMap vs. OctoCache when varying sensing ranges and resolutions using AscTec Pelican UAV. (a),(b) fixed sensing range=3m with resolutions from 0.1 to 0.2m. (c),(d) fixed resolution=0.15m with sensing ranges from 2m to 4m.



(a) Sensing range=3m, runtime (b) Sensing range=3m, velocity (line) and flight time (block) (c) Resolution=0.03m, runtime (d) Resolution=0.03m, velocity (line) and flight time (block)

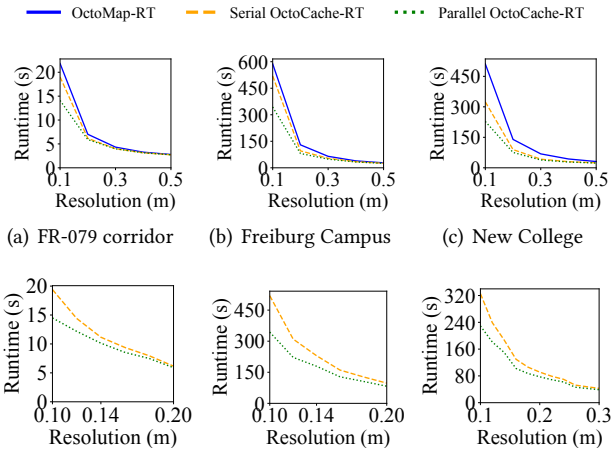
Figure 19. OctoMap-RT vs. OctoCache-RT when varying sensing ranges and resolutions using AscTec Pelican UAV. (a),(b) fixed sensing range=3m, varies resolution from 0.01 to 0.05m. (c)(d) fixed resolution=0.03m, varies sensing range from 2m to 4m.



(a) FR-079 corridor (b) Freiburg Campus (c) New College

(d) FR-079 corridor (0.1-0.2 zoom in) (e) Freiburg Campus (0.1-0.2 zoom in) (f) New College (0.1-0.3 zoom in)

Figure 20. OctoCache vs. OctoMap on 3D environment construction.



(a) FR-079 corridor (b) Freiburg Campus (c) New College

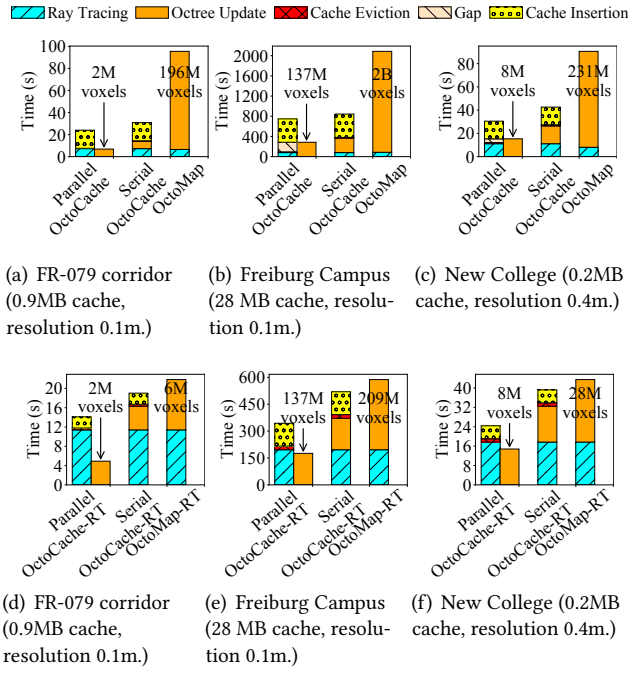
(d) FR-079 corridor (0.1-0.2 zoom in) (e) Freiburg Campus (0.1-0.2 zoom in) (f) New College (0.1-0.3 zoom in)

Figure 21. OctoCache-RT vs. OctoMap-RT on 3D environment construction.

1 to ensure data consistency. The potential gain is limited by $min(T_{raytracing} + T_{cacheeviction}, T_{Octreeupdate})$, with more significant improvements when 1) these runtimes are close, and 2) they constitute a large portion of the serial runtime. Figure 22(c) shows an optimal scenario with over 30% improvement. In cases with a large runtime gap on thread 1 (Figure 22(b)), the additional gain from parallelizing OctoCache is small, indicating a need for a larger cache to increase hits and further reduce octree update overhead on thread 2.

6.2.3 Cache hit ratio and per-voxel insertion speed:

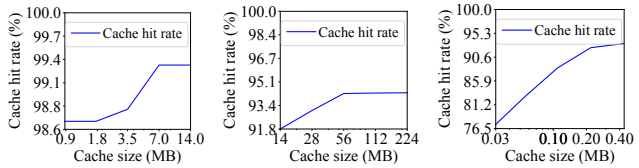
Figure 23 gives a detailed analysis of the cache hit ratio corresponding to the cache size and the octree memory overhead after all voxel updates. On dataset 3, using a cache of only 0.23% of the octree size achieves a more than 93% cache hit rate. We also observe from Figure 23(a) and 23(b) that using a cache size above a threshold stops the hit rate from increasing. This denotes that we have utilized all duplications inter and intra the update batches. Using a larger-sized cache not only generates more cache hits but also reduces the time of searching in each bucket as there will be fewer collisions of voxels in one bucket.



(a) FR-079 corridor (0.9MB cache, resolution 0.1m.) (b) Freiburg Campus (28 MB cache, resolution 0.1m.) (c) New College (0.2MB cache, resolution 0.4m.)

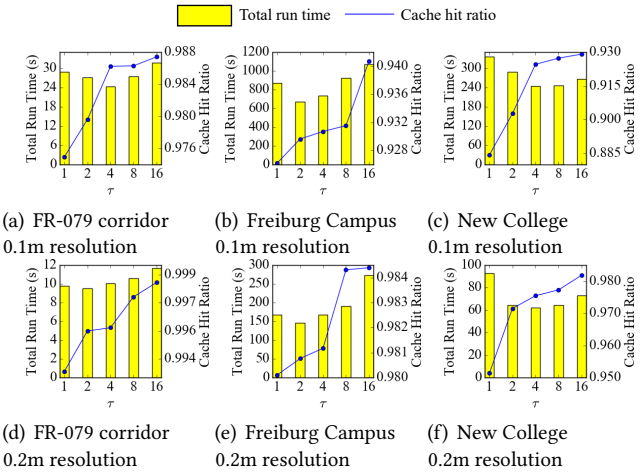
(d) FR-079 corridor (0.9MB cache, resolution 0.1m.) (e) Freiburg Campus (28 MB cache, resolution 0.1m.) (f) New College (0.2MB cache, resolution 0.4m.)

Figure 22. Runtime decomposition. The number of voxels inserted into the octree is listed together with the octree update runtime.



(a) FR-079 corridor (octree 19MB) (b) Freiburg Campus (octree 2.5GB) (c) New College (octree 0.99GB)

Figure 23. Hit ratio raises to a limit as cache size grows.



(a) FR-079 corridor 0.1m resolution (b) Freiburg Campus 0.1m resolution (c) New College 0.1m resolution

(d) FR-079 corridor 0.2m resolution (e) Freiburg Campus 0.2m resolution (f) New College 0.2m resolution

Figure 24. Map construction time and cache hit ratio with τ.

6.2.4 Evaluation on cache shape (τ): We ask *given a fixed cache size limit M, what is the best cache shape?* The shape of the cache depends on 2 parameters, the number of buckets w and maximum number of voxel tuples in a bucket τ. The cache size M is contained as $7 \times w \times \tau$ after cache eviction as each voxel tuple takes 7 bytes. As w is always the power of 2 as explained in Section 4.2, we test τ=1, 2, 4, 8, 16 in the following evaluation. Figure 24 shows that the optimal τ is between 2 and 4 for most datasets. We reason that extreme τ values (too small or too large) hurt runtime: a) *A small τ limits cache capacity*, causing early evictions and more cache misses due to hash collisions. For example, with $\tau = 1$, two voxels mapped to the same bucket will force an eviction, increasing misses. b) *A large τ increases insertion overhead*, as more comparisons are needed per voxel insertion. However, with a w close to the number of non-duplicate voxels, most buckets contain ≤ 4 voxels due to the scanning pattern and Morton code mapping. Thus, we choose $\tau = 4$ as a reasonable bucket size when the cache is 3–4× the number of non-duplicate voxels per batch.

7 Conclusion

The slow update speed in OctoMap is a major bottleneck for robot and UAV applications, caused by large voxel data volume and inefficient per-voxel updates. To address this, we introduced OctoCache, which enhances OctoMap by: 1) creating a flattened cache to reduce memory accesses for duplicated voxels, 2) implementing a Morton Code-based cache policy to optimize octree updates, and 3) parallelizing octree updates with ray tracing and cache eviction. OctoCache maintains query and API consistency while adding minimal memory and CPU overhead, making it suitable for low-powered edge devices. Evaluations on a UAV autonomous navigation platform show that OctoCache reduces end-to-end runtime and improves task completion time by up to 28%. In 3D environment construction, OctoCache achieves up to 3.02× speedup over OctoMap.

8 Acknowledgements

We thank the anonymous ASPLOS reviewers for their thorough comments and feedback. Minlan Yu and Vijay Reddi were partially supported by NSF CNS NeTS 2107078. Zaoxing Liu was supported in part by NSF grants CNS-2431093 and CNS-2415758. This work was also supported in part by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. We also thank Nengneng Yu, Jiachen Lu, and Xinhao Kong for their kind help.

References

[1] Open source:. https://github.com/KaiserV2/OctoCache/tree/cache3.0.

[2] "public point cloud dataset from octomap". http://ais.informatik.uni-freiburg.de/projects/datasets/octomap/.

[3] Mateen Ashraf, Anna Gaydamaka, Bo Tan, Dmitri Moltchanov, and Yevgeni Koucheryavy. Low complexity algorithms for mission completion time minimization in uav-based isac systems. *arXiv preprint arXiv:2310.08311*, 2023.

[4] Thomas Ball and Sriram K Rajamani. The slam toolkit. In *Proceedings of CAV'2001 (13th Conference on Computer Aided Verification)*, volume 2102, pages 260–264, 2000.

[5] BBC. "bbc news, "google plans drone delivery service for 2017."". http://www.bbc.com/news/technology-34704868, 2015.

[6] M Grosse Besselmann, Lennart Puck, Lea Steffen, Arne Roennau, and Rüdiger Dillmann. Vdb-mapping: a high resolution and real-time capable 3d mapping framework for versatile mobile robots. In *2021 IEEE 17th International Conference on Automation Science and Engineering (CASE)*, pages 448–454. IEEE, 2021.

[7] Filip Biljecki, Jantien Stoter, Hugo Ledoux, Sisi Zlatanova, and Arzu Çöltekin. Applications of 3d city models: State of the art review. *ISPRS International Journal of Geo-Information*, 4(4):2842–2889, 2015.

[8] Behzad Boroujerdian, Hasan Genc, Srivatsan Krishnan, Wenzhi Cui, Aleksandra Faust, and Vijay Reddi. Mavbench: Micro aerial vehicle benchmarking. In *2018 51st annual IEEE/ACM international symposium on microarchitecture (MICRO)*, pages 894–907. IEEE, 2018.

[9] Widodo Budiharto, Andry Chowanda, Alexander Agung Santoso Gunawan, Edy Irwansyah, and Jarot Sembodo Suroso. A review and progress of research on autonomous drone in agriculture, delivering items and geographical information systems (gis). In *2019 2nd world symposium on communication engineering (WSCE)*, pages 205–209. IEEE, 2019.

[10] Bin Chen, Fengru Huang, and Yu Fang. Integrating virtual environment and gis for 3d virtual city development and urban planning. In *2011 IEEE international geoscience and remote sensing symposium*, pages 4200–4203. IEEE, 2011.

[11] Paolo Cignoni, Marco Callieri, Massimiliano Corsini, Matteo Dellepiane, Fabio Ganovelli, Guido Ranzuglia, et al. Meshlab: an open-source mesh processing tool. In *Eurographics Italian chapter conference*, volume 2008, pages 129–136. Salerno, Italy, 2008.

[12] Daniele De Gregorio and Luigi Di Stefano. Skimap: An efficient mapping framework for robot navigation. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2569–2576. IEEE, 2017.

[13] EA De Kemp and WD Goodfellow. 3-d geological modelling supporting mineral exploration. *Mineral Deposits of Canada–a synthesis of Major deposit types, District metallogeny, The evolution of geological provinces and Exploration methods., Geol. Assoc. Canada Spec. Publ*, (5):1051–1061, 2007.

[14] Cameron Desrochers. A single-producer, single-consumer lock-free queue for c++. \T1\textquotedblleft https://github.com/cameron314/readerwriterqueue\T1\textquotedblright, 2024. Accessed on 04.30.2024.

[15] Sebastian Dorn, Nicola Wolpert, and Elmar Schömer. Voxel-based general voronoi diagram for complex data with application on motion planning. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 137–143. IEEE, 2020.

[16] Daniel Duberg and Patric Jensfelt. Ufomap: An efficient probabilistic 3d mapping framework that embraces the unknown. *IEEE Robotics and Automation Letters*, 5(4):6411–6418, 2020.

[17] Inc. Epic Games. Physics simulation — unreal engine, 2024. https://www.unrealengine.com/en-US/unreal-engine-5. Accessed on 04.30.2024.

[18] Peter R. Florence, John Carter, Jake Ware, and Russ Tedrake. Nanomap: Fast, uncertainty-aware proximity queries with lazy search over local 3d data. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 7631–7638, 2018.

[19] Dean Gesch, Gayla Evans, James Mauck, John Hutchinson, and William J Carswell Jr. The national map-elevation. Technical report, US Geological Survey, 2009.

[20] W Nicholas Greene, Kyel Ok, Peter Lommel, and Nicholas Roy. Multi-level mapping: Real-time dense monocular slam. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 833–840. IEEE, 2016.

[21] Giorgio Grisetti, Rainer Kümmerle, Cyrill Stachniss, and Wolfram Burgard. A tutorial on graph-based slam. *IEEE Intelligent Transportation Systems Magazine*, 2(4):31–43, 2010.

[22] Ramyad Hadidi, Bahar Asgari, Sam Jijina, Adriana Amyette, Nima Shoghi, and Hyesoon Kim. Quantifying the design-space tradeoffs in autonomous drones. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 661–673, 2021.

[23] Kumaran Handy. Til ubisoft offered to share their detailed 3d model of notre dame from assassin's creed: Unity, 2023. https://www.artstation.com/blogs/dioeye/1dYG/bridging-the-gap-between-gaming-and-history-how-assassins-creed-unity-is-helping-rebuild-notre-dame. Accessed on 04.30.2024.

[24] Yuhui Hao, Yiming Gan, Bo Yu, Qiang Liu, Yinhe Han, Zishen Wan, and Shaoshan Liu. Orianna: An accelerator generation framework for optimization-based robotic applications. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 813–829, 2024.

[25] Andrew Hogue, Sunbir Gill, and Michael Jenkin. Automated avatar creation for 3d games. In *Proceedings of the 2007 conference on Future Play*, pages 174–180, 2007.

[26] Armin Hornung, Kai M Wurm, Maren Bennewitz, Cyrill Stachniss, and Wolfram Burgard. Octomap: An efficient probabilistic 3d mapping framework based on octrees. *Autonomous robots*, 34(3):189–206, 2013.

[27] Wenqiang Hu, Wenyu Yang, and Youlun Xiong. An adaptive mesh model for 3d reconstruction from unorganized data points. *The International Journal of Advanced Manufacturing Technology*, 26:1362–1369, 2005.

[28] Tianyu Jia, En-Yu Yang, Yu-Shun Hsiao, Jonathan Cruz, David Brooks, Gu-Yeon Wei, and Vijay Janapa Reddi. Omu: a probabilistic 3d occupancy mapping accelerator for real-time octomap at the edge. *arXiv preprint arXiv:2205.03325*, 2022.

[29] Urs Kanus, Gregor Wetekam, and Johannes Hirche. Voxelcache: a cache-based memory architecture for volume graphics. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 76–83, 2003.

[30] Srivatsan Krishnan, Zishen Wan, Kshitij Bhardwaj, Ninad Jadhav, Aleksandra Faust, and Vijay Janapa Reddi. Roofline model for uavs: A bottleneck analysis tool for onboard compute characterization of autonomous unmanned aerial vehicles. In *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 162–174. IEEE, 2022.

[31] Srivatsan Krishnan, Zishen Wan, Kshitij Bhardwaj, Paul Whatmough, Aleksandra Faust, Sabrina Neuman, Gu-Yeon Wei, David Brooks, and Vijay Janapa Reddi. Automatic domain-specific soc design for autonomous unmanned aerial vehicles. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 300–317. IEEE, 2022.

[32] Srivatsan Krishnan, Zishen Wan, Kshitij Bhardwaj, Paul Whatmough, Aleksandra Faust, Gu-Yeon Wei, David Brooks, and Vijay Janapa Reddi. The sky is not the limit: A visual performance model for cyber-physical co-design in autonomous machines. *IEEE Computer Architecture Letters*, 19(1):38–42, 2020.

[33] In-So Kweon, M Hebert, Eric Krotkov, and T Kanade. Terrain mapping for a roving planetary explorer. In *IEEE International Conference on*

Robotics and Automation, pages 997–1002. IEEE, 1989.

[34] In So Kweon and Takeo Kanade. Extracting topographic terrain features from elevation maps. *CVGIP: image understanding*, 59(2):171–182, 1994.

[35] Feng Liu and Xiaoming Liu. Voxel-based 3d detection and reconstruction of multiple objects from a single image. *Advances in Neural Information Processing Systems*, 34:2413–2426, 2021.

[36] Qiang Liu, Zishen Wan, Bo Yu, Weizhuang Liu, Shaoshan Liu, and Arijit Raychowdhury. An energy-efficient and runtime-reconfigurable fpga-based accelerator for robotic localization systems. In *2022 IEEE Custom Integrated Circuits Conference (CICC)*, pages 01–02. IEEE, 2022.

[37] Benjamin Mark, Tudor Berechet, Tobias Mahlmann, and Julian Togelius. Procedural generation of 3d caves for games on the gpu. In *Foundations of Digital Games*, 2015.

[38] Víctor Mayoral-Vilches, Jason Jabbour, Yu-Shun Hsiao, Zishen Wan, Martiño Crespo-Álvarez, Matthew Stewart, Juan Manuel Reina-Muñoz, Prateek Nagras, Gaurav Vikhe, Mohammad Bakhshalipour, et al. Robotperf: An open-source, vendor-agnostic, benchmarking suite for evaluating robotics computing system performance. In *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pages 8288–8297. IEEE, 2024.

[39] Donald Meagher. Geometric modeling using octree encoding. *Computer graphics and image processing*, 19(2):129–147, 1982.

[40] Yu Miao, Alan Hunter, and Ioannis Georgilas. An occupancy mapping method based on k-nearest neighbours. *Sensors*, 22(1):139, 2021.

[41] Arthur Holland Michel. "amazon's drone patents.". https://dronecenter.bard.edu/files/2017/09/CSD-Amazons-Drone-Patents-1.pdf, 2017. Accessed on 04.30.2024.

[42] Heajung Min, Kyung Min Han, and Young J Kim. Accelerating probabilistic volumetric mapping using ray-tracing graphics hardware. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5440–5445. IEEE, 2021.

[43] Heajung Min, Kyung Min Han, and Young J Kim. Octomap-rt: Fast probabilistic volumetric mapping using ray-tracing gpus. *IEEE Robotics and Automation Letters*, 2023.

[44] Raul Mur-Artal, Jose Maria Martinez Montiel, and Juan D Tardos. Orb-slam: a versatile and accurate monocular slam system. *IEEE transactions on robotics*, 31(5):1147–1163, 2015.

[45] Ken Museth. Vdb: High-resolution sparse volumes with dynamic topology. *ACM transactions on graphics (TOG)*, 32(3):1–22, 2013.

[46] Sabrina M Neuman, Brian Plancher, Thomas Bourgeat, Thierry Tambe, Srinivas Devadas, and Vijay Janapa Reddi. Robomorphic computing: a design methodology for domain-specific accelerators parameterized by robot morphology. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 674–686, 2021.

[47] Nvidia. "embedded systems developer kits, modules, and sdks — nvidia histor.". https://www.nvidia.com/en-us/autonomous-machines/embedded-systems-dev-kits-modules/.

[48] Nvidia. "nvidia jetson nano bringing the power of modern ai to millions of devices.". https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-nano/product-development/.

[49] Nvidia. "nvidia jetson xavier a breakthrough in embedded applications..". https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-xavier-series/.

[50] Andri Qiantori, Agung Budi Sutiono, Hadi Hariyanto, Hirohiko Suwa, and Toshizumi Ohta. An emergency medical communications system by low altitude platform at the early stages of a natural disaster in indonesia. *Journal of medical systems*, 36:41–52, 2012.

[51] James Rogers. How drones are helping the nepal earthquake relief effort. *Fox News*, 30, 2015.

[52] Catur Aries Rokhmana. The potential of uav-based remote sensing for supporting precision agriculture in indonesia. *Procedia Environmental Sciences*, 24:245–253, 2015.

[53] Radu Bogdan Rusu and Steve Cousins. 3d is here: Point cloud library (pcl). In *2011 IEEE international conference on robotics and automation*, pages 1–4. IEEE, 2011.

[54] Leo Stocco and Günther Schrack. Integer dilation and contraction for quadtrees and octrees. In *IEEE Pacific Rim Conference on Communications, Computers, and Signal Processing. Proceedings*, pages 426–428. IEEE, 1995.

[55] Takafumi Taketomi, Hideaki Uchiyama, and Sei Ikeda. Visual slam algorithms: A survey from 2010 to 2016. *IPSJ transactions on computer vision and applications*, 9:1–11, 2017.

[56] Zishen Wan, Nandhini Chandramoorthy, Karthik Swaminathan, Pin-Yu Chen, Kshitij Bhardwaj, Vijay Janapa Reddi, and Arijit Raychowdhury. Mulberry: Enabling bit-error robustness for energy-efficient multi-agent autonomous systems. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 746–762, 2024.

[57] Zishen Wan, Bo Yu, Thomas Yuang Li, Jie Tang, Yuhao Zhu, Yu Wang, Arijit Raychowdhury, and Shaoshan Liu. A survey of fpga-based robotic computing. *IEEE Circuits and Systems Magazine*, 21(2):48–74, 2021.

[58] Elizabeth Weise. "amazon delivered its first customer package by drone.". https://www.usatoday.com/story/tech/news/2016/12/14/amazon-delivered-its-first-customer-package-drone/95401366/, 2016. Accessed on 04.30.2024.

[59] Jane Wilhelms and Allen Van Gelder. Octrees for faster isosurface generation. *ACM Transactions on Graphics (TOG)*, 11(3):201–227, 1992.

[60] Chao Yu, Zuxin Liu, Xin-Jun Liu, Fugui Xie, Yi Yang, Qi Wei, and Qiao Fei. Ds-slam: A semantic visual slam towards dynamic environments. In *2018 IEEE/RSJ international conference on intelligent robots and systems (IROS)*, pages 1168–1174. IEEE, 2018.