2026 IEEE International Conference on Robotics and Automation (ICRA 2026)
June 1-5, 2026. Vienna, Austria

VCC: Efficient Voxel-Based Collision Checking Framework for Real-Time Robotic Motion Planning

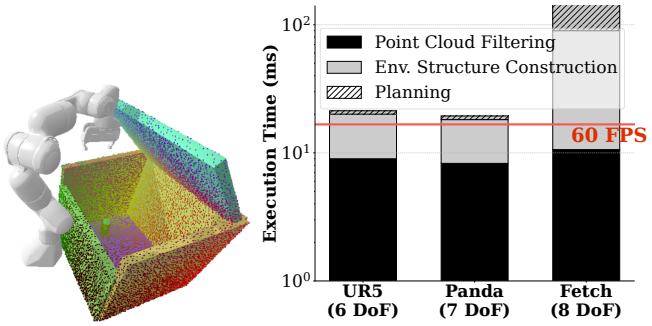
Ching Chen[1] and Tsung-Tai Yeh[1]

***Abstract*— To navigate environments with dynamic obstacles, a robot must continuously scan for them and find collision-free paths to reach a goal position. This process starts with receiving obstacle information in the form of a point cloud, followed by a pre-planning stage that involves preprocessing to remove unnecessary points and constructing an environment data structure. However, the pre-planning stage can consume more than 16× the runtime of the planning stage, slowing the robot's reaction speed. Thus, in this work, we propose *VCC*, an efficient collision checking framework that primarily targets the pre-planning bottleneck. VCC first cleans the point cloud using Center-selective Voxel Filtering. It then divides the environment into voxels using Adaptive Workspace Voxelization and organizes them in a Multilevel Voxel Table (MVT). In addition, VCC manages the MVT in two memory pools to ensure high data locality and SIMD-aligned data layout. During motion planning, the planner can perform low-latency SIMD-accelerated collision checking using the MVT. Compared with the state-of-the-art method, the experimental results show a 3.63× speedup in filtering. In terms of environment data structure, MVT achieves a 220.48× speedup during construction and reduces memory usage by 97.73%. Additionally, VCC accelerates sampling-based planning by 1.94×. Altogether, VCC achieves an end-to-end speedup of 7.71× on the desktop CPU platform and 4.23× on the embedded computer platform, making real-time motion planning practical for resource-constrained edge devices.**

I. Introduction

Motion planners [1], [2] identify collision-free paths in real time to ensure that robots operate safely in the presence of dynamic obstacles. Such real-time responsiveness is particularly critical for collaborative robots, which must avoid human collaborators as they approach. To achieve this, the robot needs to frequently capture environmental data, typically in the form of point clouds from depth cameras. As illustrated in Fig. 1a, the Panda robotic manipulator perceives its surroundings as a dense collection of points. These points are then filtered to remove redundant points and organized into an environment data structure. The motion planner queries this structure to verify whether a robot pose is safe during the planning of a collision-free path. A more efficient processing pipeline enables faster reactions to dynamic obstacles.

However, our analysis reveals that the filtering and data structure construction stages significantly slow down the entire process. We investigated these performance issues by benchmarking Collision-Affording Point Trees (CAPT) [1], a state-of-the-art collision checking framework that leverages Single Instruction, Multiple Data (SIMD) parallelization for real-time motion planning. We evaluated 700 planning problems on an Orange Pi 5B (OPi) [3] single-board embedded computer using the dynamic-domain balanced RRT-Connect (RRT-C) motion planning method across three robotic manipulators with 6 to 8 degrees of freedom (DOF). Fig. 1b illustrates the average execution-time breakdown for point cloud filtering, environment structure construction, and motion planning, plotted on a logarithmic scale. Our findings indicate that for the UR5 and Panda robots, the primary bottleneck is not the motion planning itself, but the filtering and construction stages, which we collectively term the pre-planning stage. For the Fetch robot, the pre-planning stage is so computationally intensive that it alone nearly prevents the system from meeting a 10 frames per second (FPS) refresh rate. The bottleneck directly limits the robot's responsiveness, thereby increasing the risk of collisions.



(a) (b)

Fig. 1: (a) Robotic manipulator and point cloud obstacles in a simulated environment. (b) Execution time breakdown of Collision-Affording Point Tree (CAPT) [1] on Orange Pi 5B Embedded device. [3]

This paper introduces VCC, an efficient voxel-based collision checking framework that accelerates every stage of the motion planning pipeline. VCC begins with our proposed **Center-selective Voxel Filtering** to remove redundant point cloud data, followed by **Adaptive Workspace Voxelization** to divide the environment into voxels. Subsequently, these voxels are organized into a novel **Multilevel Voxel Table** structure, providing a high-performance foundation for motion planning. Experimental results demonstrate that VCC outperforms the state-of-the-art method, achieving a 3.63× speedup in filtering efficiency and a 220.48× speedup in environment structure construction. Furthermore, VCC reduces the memory usage for the environment structure by 97.73%. Altogether, VCC achieves an end-to-end speedup of 7.71× on a desktop platform and 4.23× on the OPi,

[1]Department of Computer Science, National Yang Ming Chiao Tung University, Taiwan. Emails: cchen.cs13@nycu.edu.tw, ttyeh@cs.nycu.edu.tw

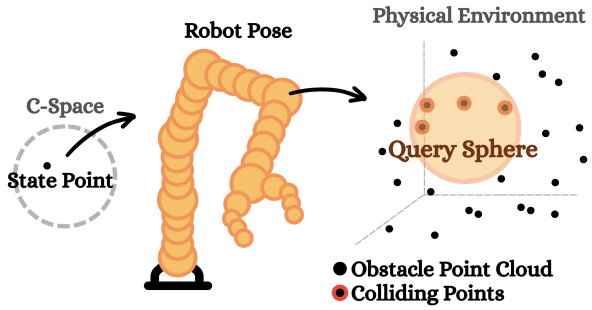


Fig. 2: The mapping from a state point in $\mathcal{C}$-space to its physical pose. The robot's volume is approximated using query spheres to facilitate collision checking against the surrounding obstacle point cloud.

effectively enabling real-time motion planning on resource-constrained embedded computers.

II. Background

A. Sampling-based and Neural-based Motion Planning

Motion planning aims to find a collision-free path to a goal position in the physical environment. Motion planners typically determine manipulator movement within the configuration space ($\mathcal{C}$-space), where each state point corresponds to a joint configuration that maps to a unique physical pose, as illustrated in Fig. 2. To check whether a state is collision-free, in practice, forward kinematics maps the state to the physical environment, where a set of query spheres approximates the robot poses. A state is valid only if all associated spheres remain clear of the obstacle point cloud.

In this study, we focus on two prominent planning paradigms: sampling-based and neural-based approaches. Sampling-based Motion Planning (SBMP), such as Rapidly-exploring Random Trees (RRT) [4], RRT-Connect (RRT-C) [5], and Probabilistic Roadmaps (PRM) [6], explores C-spaces by randomly sampling points and connecting them to form a path. The performance of SBMPs is heavily dependent on the efficiency of collision checking. In contrast, Neural-based Motion Planning (NBMP) [7], [8], [9] utilizes deep learning models to infer state points in $\mathcal{C}$-space, "guessing" efficient paths to avoid the random search inherent in SBMPs. However, the diversity of the training data limits NBMP generalization. Furthermore, on resource-constrained platforms, high inference latency often hinders real-time performance. For both SBMP and NBMP, efficient environment representation for low-latency collision checking is essential. VCC provides a high-performance, voxel-based framework compatible with both paradigms.

B. Efficient Environment Data Structure

Efficient data structures representing the physical point cloud environment are fundamental to accelerating collision checking. A common representation is the occupancy map; for example, OctoMap [10] is a widely used implementation that employs an octree structure to maintain a probabilistic occupancy map. It is integrated with collision-checking libraries, such as the Flexible Collision Library (FCL), and planning frameworks such as MoveIt. Unlike occupancy maps, Signed Distance Fields (SDF) store the Euclidean distance to the nearest obstacle boundary at each voxel. VoxBlox [11] and VoxGraph [12] are popular CPU-based approaches for constructing these fields from sensor data. While SDFs provide critical gradient information for trajectory optimization, they typically require the initial construction of a Truncated Signed Distance Field (TSDF) to fuse sensor measurements, which increases computational overhead. For direct point cloud representation, k-d trees are a standard choice for nearest-neighbor searches. Well-known implementations include FLANN [13], NanoFLANN [14] and Nigh [15]. Recently, the Collision-Affording Point Tree (CAPT) [1] has emerged as a state-of-the-art implementation for collision checking. By utilizing a branch-free SIMD-parallel traversal and an optimized memory layout, CAPT significantly outperforms traditional structures, including OctoMap [10], FLANN [13], NanoFLANN [14], and Nigh [15], achieving an average query time of less than 10 nanoseconds on high-performance desktop CPUs.

III. Memory-Efficient Voxel Encoding

This section introduces a highly optimized pre-planning stage for motion planning. Specifically, we propose Center-selective Voxel Filtering for efficient point cloud downsampling, Adaptive Workspace Voxelization for optimal spatial partitioning, and the Multilevel Voxel Table for real-time construction and rapid collision checking.

Preliminaries and Assumptions

The robot's geometry is approximated by a set of spheres with radii ranging from $r_{\min}$ to $r_{\max}$. We define the robot's reachable region as a cubic axis-aligned bounding box (AABB), denoted by $\mathbf{W} = [\mathbf{w}_{\min}, \mathbf{w}_{\max}]$ with side length W_{width}. The AABB defines the robot's **workspace**. We derive the workspace for fixed-base manipulators from the base position and the manipulator's maximum extension. The workspace $\mathbf{W}$ is divided into a uniform 3D voxel grid, with the number of voxels per dimension defined as G_{width}.

We employ this pipeline to generate collision-free trajectories from point cloud frames:

1) **Point Cloud Filtering:** Removes redundant data from raw point clouds.
2) **Environment Data Structure Construction:** Builds an environment data structure to represent the filtered point cloud, aiming to accelerate collision checking (CC) queries.
3) **Motion Planning:** Iteratively generates candidate states and issues frequent CC queries for safety evaluation.
4) **Simplification:** Refines paths for robot execution.

High-frequency execution of this pipeline enables real-time dynamic obstacle avoidance through continuous trajectory updates.

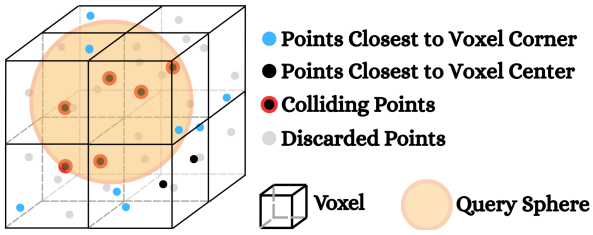


Fig. 3: Rationale for Center-selective Voxel Filtering

A. Center-selective Voxel Filtering

Robots capture environmental information as point clouds via depth cameras. However, these point clouds often contain redundant data, which increases computational overhead without improving CC quality. To address this, Center-selective Voxel Filtering (CenterVox) aims to reduce data redundancy while preserving the geometric coverage required for safe navigation.

CenterVox excludes points outside the workspace and identifies the single representative point closest to the center of each voxel. The voxel side length L_{filter} can be adjusted for different point densities. Fig. 3 illustrates the rationale: blue points are nearest voxel corners, while black points are closest to the center. Selecting only the eight blue points can lead to misleading gaps in the point cloud. When query spheres overlap with these gaps, they may miss collisions, leading to false negatives. By selecting the eight black points nearest the centers, we effectively minimize these misleading gaps and mitigate the risk of false negatives.

CenterVox distinguishes itself through an efficient structure that optimizes both memory footprint and data locality. It leverages the Multilevel Voxel Table (MVT), detailed in Sec. III-D, but diverges in point-to-voxel insertion logic. While the MVT accommodates variable-length point arrays in each voxel, CenterVox greedily constrains each voxel to retain only the point closest to its center. Notably, the algorithm requires only a single pass over the point cloud and achieves a linear time complexity of $\mathcal{O}(n_{\text{raw}})$, where n_{raw} is the number of raw points.

B. Adaptive Workspace Voxelization

Following point cloud filtering, VCC constructs a voxel-based data structure for rapid retrieval of potential colliding points during CC queries. To determine an optimal voxel side length L, we analyzed its impact on both memory usage and CC time. Decreasing L improves grid resolution but increases the metadata footprint, as illustrated by the solid line in Fig. 4. L governs CC time by balancing the number of voxel lookups against the point count within each voxel. The MVT's rapid lookup mechanism, detailed in Sec. III-D, ensures that voxel lookup overhead remains negligible. Thus, the point count within each voxel dominates the performance. As shown by the dotted line in Fig. 4, a larger L increases points per voxel, causing more redundant checks on non-colliding points. Conversely, a smaller L narrows the candidate point set, improving speed.

In light of this analysis, we select $L = r_{\max}$ as the design parameter. This configuration strikes a favorable balance, maintaining a compact memory footprint while delivering competitive CC performance.

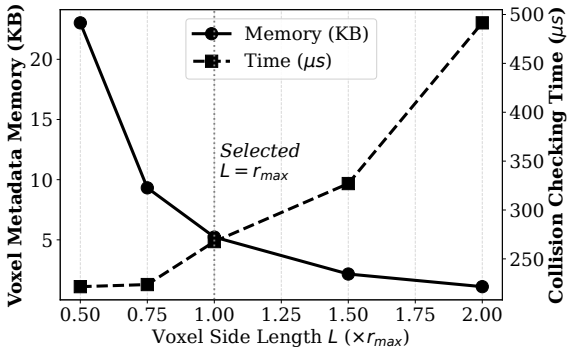


Fig. 4: Impact of voxel side length L on performance for a Fetch robot navigating a *box* scene using RRT-C. The plot shows the average collision checking time across 100 planning problems, along with memory usage for a representative problem.

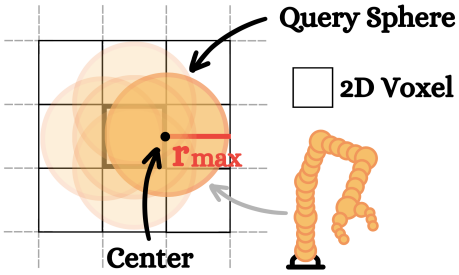


Fig. 5: Adaptive Workspace Voxelization sets the voxel side length $L = r_{\max}$ and partitions the point cloud into voxels accordingly. In this 2D example, the bold central voxel contains the query sphere's center. Even if the center lies on a boundary line, the sphere's surface reaches at most its immediate neighboring voxels. Thus, checking the 3×3 grid is sufficient for accurate collision checking. In 3D space, this logic ensures that checking for collisions within $3 \times 3 \times 3$ block (27 voxels) is sufficient.

C. Analysis of Selected Voxel Side Length

Defining $L = r_{\max}$ establishes a strictly bounded search space for collision checking. If a query sphere center ($r \leq r_{\max}$) lies within a voxel, its volume is guaranteed to stay within that voxel and its immediate neighbors. In 3D space, this restricts the relevant voxels to a fixed $3 \times 3 \times 3$ block—comprising the central voxel and its 26 neighbors—regardless of the sphere's position. Fig. 5 illustrates this property in a 2D example.

Note that in our implementation, we expand L by a predefined point radius r_{point} to account for the point cloud's physical volume, ensuring that any point capable of touching the query sphere is strictly captured within the $3 \times 3 \times 3$ block. Derived from Eq. 1, the G_{width} values for the UR5, Panda, and Fetch manipulators evaluated in our experiments are 29, 28, and 12, respectively.

$$G_{\text{width}} = \lfloor \frac{W_{\text{width}}}{r_{\max} + r_{\text{point}}} \rfloor \quad (1)$$

D. Multilevel Voxel Table

1) Sparse Hierarchical Design: Leveraging the adaptive workspace voxelization, VCC encodes point cloud data in a hierarchical spatial structure called the Multilevel Voxel Table (MVT). Given the inherent sparsity of point clouds, our analysis reveals that the percentage of empty voxels in the *table_pick* scene for the UR5, Panda, and Fetch manipulators is 98.08%, 98.64%, and 95.83%, respectively. To avoid the memory waste of a dense grid, the MVT employs a sparse three-level indexing scheme. As illustrated in Fig. 6, empty entries, represented by gray boxes in X, Y, and Z-level tables, store invalid indicators, bypassing unnecessary memory allocation in subsequent levels.

2) Memory-efficient Two-phase Construction: Before motion planning, VCC constructs the MVT to record voxel information. The system utilizes two pre-allocated, contiguous memory pools—*Table Pool* and *Point Coordinate Pool*—to eliminate frequent heap allocation overhead and enhance data locality. To ensure a compact memory footprint, it adopts a two-phase strategy:

a) Phase 1: Hierarchy Construction and Voxel Initialization: The system iterates through all points to perform two simultaneous tasks:

- **Table Construction:** VCC maps Points $\mathbf{p}$ to grid coordinates $\mathbf{v}$ via Eq. 2 and allocates X, Y, and Z-level tables from the *Table Pool* upon first access. The table relationships are represented by **Table Offset** stored in the table array, with address resolved by Eq. 3. As shown in Fig. 6, blue table arrays are allocated from the *Table Pool*. Blue cells indicate the position of the next-level table or voxel structure indices.
- **Voxel Structure Initialization:** The system initializes voxel structure for non-empty voxels and records the per-voxel point count. To avoid redundant point-to-grid coordinate transformations in Phase 2, the voxel structure index of each point is cached in a temporary mapping array.

$$\mathbf{v} = (\mathbf{p} - \mathbf{w}_{\min}) \cdot \frac{G_{width}}{W_{width}} \quad (2)$$

$$\text{Table Addr.} = \text{Table Pool Base Addr.} + \text{Table Offset} \quad (3)$$

b) Phase 2: Point Coordinate Data Distribution: To maximize SIMD efficiency, VCC first rounds up the point coordinate capacity of each voxel to the nearest SIMD lane width: for instance, 16 bytes (4 floats) for ARM NEON, or 32 bytes (8 floats) for x86 AVX2. Using these aligned capacities, VCC calculates the exact total memory requirement and initializes the *Point Coordinate Pool*.

Subsequently, VCC allocates point coordinate arrays for each voxel from the *Point Coordinate Pool* using the Structure of Arrays (SoA) format, ensuring the start address of each point coordinate array is SIMD-aligned, as depicted by the brown arrays in Fig. 6. Finally, the system populates the SoA arrays with points and pads any unfilled slots with ∞.

Overall, MVT construction achieves $\mathcal{O}(n)$ time complexity for n points in the filtered point cloud.

3) MVT Design for Low-latency Collision Checking: The MVT is optimized to accelerate collision checking through four key features:

- $\mathcal{O}(1)$ **Voxel Access Time:** Accessing any voxel only requires three table lookups. As shown in Fig. 6, to retrieve the point data for a voxel at grid coordinates $(v_x, v_y, v_z) = (3, 3, 2)$, VCC accesses the 3rd element (zero-indexed) of the X-level table X to find the Table Offset of the Y-level table Y_3. Resolving by Eq. 3, it then accesses the 3rd element of Y_3 to find the offset of Z-level table $Z_{3,3}$. Finally, the 2nd element of $Z_{3,3}$ gives the voxel index.
- **SIMD-aligned SoA Point Layout:** Point coordinates are stored in SoA format to enable direct vectorized operations. Combined with SIMD-width memory alignment, the system seamlessly streams coordinate data into SIMD registers, thereby maximizing collision checking throughput.
- **Two-tier AABBs:** To increase early-exit opportunities, MVT records two-tier AABBs. A global AABB is maintained to bound the point cloud within the robot workspace, allowing VCC to skip query spheres that lie outside it instantly. Furthermore, each voxel structure stores a local AABB that bounds its internal points to enable bypassing redundant point distance evaluations when a sphere intersects a voxel but not the actual points.
- **Cache-friendly Memory Layout:** Packing multilevel tables and SoA coordinates into two contiguous memory pools confines memory access to bounded regions. While traversing the multilevel tables involves some memory jumps, keeping these accesses within the *Table Pool* eliminates memory fragmentation and reduces cache misses. Furthermore, the SoA in the *Point Coordinate Pool* is stored contiguously, which ensures sequential data access and facilitates hardware prefetching.

With $\mathcal{O}(n)$ construction, $\mathcal{O}(1)$ voxel access, SIMD-specific optimizations, and cache-friendly data layout, MVT serves as a high-performance foundation for real-time motion planning in dynamic environments.

IV. SIMD-Accelerated Collision Checking

This section describes the collision-checking procedure. Leveraging the constant-time voxel access, SIMD-aligned cache-friendly memory layout, and two-tier AABBs of the MVT introduced in Sec. III-D, VCC efficiently evaluates robot states in SIMD-packed batches. The procedure processes N spheres simultaneously, corresponding to N distinct robot postures. As shown in Algorithm 1, given a batch of N sphere centers $\mathbf{C}$ and radii $\mathbf{R}$, along with environment information, the algorithm returns whether any sphere collides with the point cloud environment.

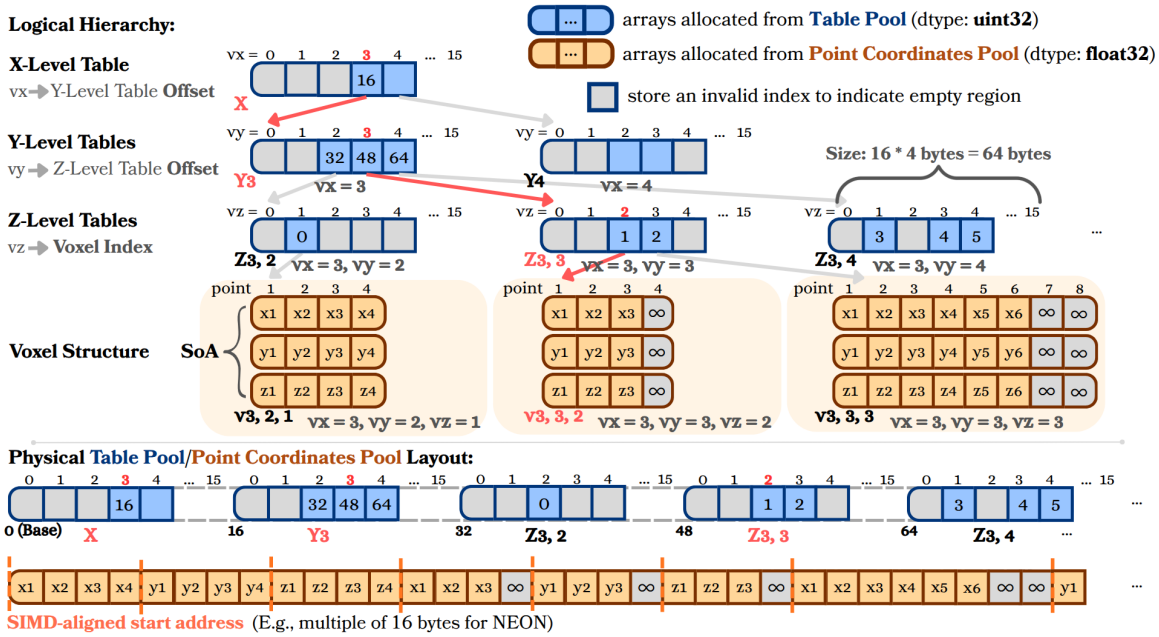


Fig. 6: Structure of the multilevel voxel table. This example assumes a grid width $G_{\text{width}} = 16$ and a SIMD lane width $w_{\text{SIMD}} = 4$.

To account for the physical volume of the points, the algorithm first expands the query radii by the predefined point radius r_{point} (Line 1). The following subsections detail the subsequent process.

A. SIMD-parallel Global Spatial Culling

To minimize redundant computations, the procedure begins with a vectorized intersection test between the N spheres and the global AABB $[\mathbf{p_{min}}, \mathbf{p_{max}}]$ (Line 2). The culling mechanism instantly discards spheres outside this boundary, bypassing all subsequent checks.

B. Voxel Lookup and Local Culling

For spheres within the global AABB, center coordinates $\mathbf{C}$ and radii $\mathbf{R}'$ are simultaneously mapped to grid coordinates using the workspace minimum corner $\mathbf{w}_{\min}$ and the scale factor s derived from Eq. 4 (Lines 5-6). To avoid the computational overhead of calculating exact sphere-voxel intersections, VCC adopts a simpler, conservative strategy: it selects all voxels within the cubic AABB of each query sphere (Lines 8-9) for check. During the MVT lookup (Lines 10-17), VCC skips vast empty regions simply by detecting invalid table offsets or invalid voxel indices. Upon reaching a valid voxel, an intersection test against its local AABB filters out spheres that overlap the voxel but do not intersect its internal points (Line 19), significantly reducing point-wise distance calculations.

$$s = \frac{G_{width}}{W_{width}} \tag{4}$$

C. Vectorized Point-wise Distance Checking

If a query sphere passes the global and local culling stages, VCC performs distance checks against the points within the voxel (Lines 21-22). Leveraging the SoA point coordinate layout, the algorithm computes squared Euclidean distances for multiple points simultaneously. If any SIMD lane identifies a value smaller than or equal to the squared sphere radius $\mathbf{r}_i'^2$, the algorithm reports a collision immediately.

The time complexity per query is $\mathcal{O}(K \cdot \lceil \frac{P}{w_{\text{SIMD}}} \rceil)$, where K denotes the number of relevant voxels, P represents the maximum points per voxel, and w_{SIMD} is the SIMD lane width. Specifically, adaptive workspace voxelization restricts K to at most 27, while CenterVox constrains P by parameter L_{filter}. Since w_{SIMD} remains fixed for a specific hardware architecture, these parameters allow VCC to achieve a constant $\mathcal{O}(1)$ time complexity per query, ensuring its efficiency remains independent of the total point cloud size.

V. EXPERIMENTS

A. Methodology

Problem Set Generation: We utilized 700 motion planning problems across seven scenes generated by MotionBenchMaker [16]: *0: table_pick*, *1: table_under_pick*, *2: box*, *3: bookshelf_small*, *4: bookshelf_tall*, *5: bookshelf_thin*, and *6: cage*. Each scene includes 100 unique problems. To represent environments as point clouds, we uniformly sampled 10,000 points per object primitive. The resulting initial point cloud sizes for the seven scenes are 120,000, 120,000, 70,000, 70,000, 150,000, 210,000, and 80,000 points, respectively.

Baselines: We compared VCC with CAPT [1], the state-of-the-art SIMD-accelerated collision checking method. Similar to CAPT, VCC is implemented within the vector-accelerated motion planning (VAMP) framework [2]. We compared our CenterVox against CAPT's point cloud filtering algorithm, which we refer to as Space-filling Curve Distance-based Filtering (SCDF). Following the authors' experimental configuration, the SCDF radius parameter R_{filter}

Algorithm 1: Collision Checking with MVT

Input: Multi-level Voxel Table $\mathcal{T}$, SIMD batch of N sphere centers $\mathbf{C} = \{\mathbf{c}_1, \ldots, \mathbf{c}_N\}$ and radii $\mathbf{R} = \{r_1, \ldots, r_N\}$, point physical radius r_{point}, workspace minimum corner $\mathbf{w}_{\min}$, global bounding box $[\mathbf{p}_{\min}, \mathbf{p}_{\max}]$, grid scale factor s, grid width G_{width}

Output: Boolean: **true** if any sphere collides with the point cloud

$\mathbf{R}' \leftarrow \{r_i + r_{\text{point}} \mid r_i \in \mathbf{R}\}$
$\mathcal{S}_{\text{inside}} \leftarrow \{i \mid \text{sphere } (\mathbf{c}_i, r'_i) \text{ intersects } [\mathbf{p}_{\min}, \mathbf{p}_{\max}]\}$
if $\mathcal{S}_{inside} = \emptyset$ **then**
  return false
$\mathbf{V} \leftarrow (\mathbf{C} - \mathbf{w}_{\min}) \cdot s$
$\mathbf{R}'_v \leftarrow \mathbf{R}' \cdot s$
for *each sphere index* $i \in \mathcal{S}_{inside}$ **do**
  $\mathbf{b}_{\min} \leftarrow \text{Clamp}(\lfloor \mathbf{v}_i - r'_{v,i} \rfloor, 0, G_{\text{width}} - 1)$
  $\mathbf{b}_{\max} \leftarrow \text{Clamp}(\lfloor \mathbf{v}_i + r'_{v,i} \rfloor, 0, G_{\text{width}} - 1)$
  for $v_x = b_{\min,x}$ **to** $b_{\max,x}$ **do**
    if $(offset_y \leftarrow \mathcal{T}.X[v_x])$ *is invalid* **then continue**
    $\text{Table}_Y \leftarrow \text{Resolve}(offset_y)$
    for $v_y = b_{\min,y}$ **to** $b_{\max,y}$ **do**
      if $(offset_z \leftarrow \mathit{Table}_Y[v_y])$ *is invalid* **then continue**
      $\text{Table}_Z \leftarrow \text{Resolve}(offset_z)$
      for $v_z = b_{\min,z}$ **to** $b_{\max,z}$ **do**
        if $(idx \leftarrow \mathit{Table}_Z[v_z])$ *is invalid* **then continue**
        $V \leftarrow \text{GetVoxel}(idx)$
        if *sphere* i *does not intersect* V.AABB **then continue**
        for *SIMD batch of points* $\mathbf{p} \in V$ **do**
          if *any*$(\|\mathbf{c}_i - \mathbf{p}\|^2 \leq \mathbf{r'}_i^2)$ **then return true**
return false

was set to 0.02 m, thereby eliminating adjacent points within this threshold. To yield a similar filtered point cloud size, CenterVox's voxel side length L_{filter} was set to 0.031 m. Furthermore, to benchmark the environment data structure, we also evaluated the construction and collision checking latencies of NanoFLANN [14], a highly optimized C++ implementation of the k-d tree. For memory usage analysis, we additionally compared a baseline dense-grid structure.

Robot Platforms: We considered three manipulators: a 6-DOF UR5, a 7-DOF Panda, and an 8-DOF Fetch. Manipulators are represented as a set of approximating spheres. Their radii range from $r_{\min}$ to $r_{\max}$ are (1.5, 8), (1.2, 8), and (1.2, 24.0) cm, comprising 40, 59, and 111 spheres, respectively.

Motion Planners (MPs): We used dynamic-domain [17] balanced [18] RRT-Connect (RRT-C) [5] for our sampling-based planning benchmarks, limited to 1 million iterations. To ensure consistency, all benchmarks used identical sets of randomly sampled configurations. Additionally, we implemented MPNet [8] for a neural-based baseline, trained on the *bookshelf_thin* scene for the Fetch manipulator.

Hardware Platforms: Our primary experimental platform is an Orange Pi 5B (OPi) [3], a single-board computer equipped with an ARM Cortex-A76 CPU. MPNet was accelerated by its built-in Neural Processing Unit (NPU) providing 6 Tera Operations Per Second (TOPS). For a more comprehensive comparison across different edge AI accelerators, we also deployed the experiments on a Raspberry Pi 5 (RPi) [19] equipped with a 26-TOPS Hailo-8 NPU [20]. To evaluate end-to-end latency across various computing tiers, benchmarks were also performed on an Intel Core i7-12700 desktop CPU. All benchmarks were executed on a single thread.

TABLE I: Processing Time of Pre-planning Stage (ms)

Robot	Method	Filtering		Construction	
		Avg.	Std.	Avg.	Std.
UR5	CAPT (SCDF)	8.99	3.25	11.04	4.47
	NanoFLANN	–	–	1.29	0.50
	Ours	**2.54**	**0.92**	**0.14**	**0.04**
Panda	CAPT (SCDF)	8.28	3.56	9.90	5.28
	NanoFLANN	–	–	1.16	0.58
	Ours	**2.36**	**0.85**	**0.12**	**0.05**
Fetch	CAPT (SCDF)	10.61	4.23	79.28	39.32
	NanoFLANN	–	–	1.83	0.78
	Ours	**2.75**	**1.02**	**0.15**	**0.06**

B. Point Cloud Filtering Latency

Table I presents the mean filtering latency and standard deviation across all planning problems. Compared to SCDF, CenterVox achieved an average speedup of 3.63× across all robots with lower standard deviations. Their primary difference lies in how they downsample the points within the workspace. SCDF maps 3D points to a 1D space-filling curve, sorts them, and iterates through the array to eliminate any neighboring points within R_{filter}, resulting in a time complexity of $\mathcal{O}(n \log n)$, where n is the raw point cloud size. SCDF performs the mapping and sorting process six times to adequately capture neighborhood relationships. In contrast, CenterVox requires only a single pass to map points to their corresponding voxels and leverages the MVT's *Table Pool* design. Consequently, CenterVox not only reduces the time complexity to $\mathcal{O}(n)$, but its cache-friendly memory layout further minimizes the execution latency.

C. Environment Data Structure Construction Latency

Table I presents the construction time statistics for CAPT, NanoFLANN, and MVT. We provided NanoFLANN with the same SCDF-filtered point clouds used for CAPT. Across the three manipulators, MVT achieved average speedups of 220.48× and a 10.01× over CAPT and NanoFLANN, respectively, with lower standard deviations. Fundamentally, both CAPT and NanoFLANN leverage k-d trees, which

typically requires $\mathcal{O}(n \log n)$ construction time, where n is the filtered point cloud size. However, during construction, CAPT additionally computes and maintains "affordance sets" to enable branch-free parallel collision queries. This overhead becomes more significant in environments with more clustered points, degrading the construction complexity to $\mathcal{O}(n^2)$. In contrast, beyond its optimized memory layout, MVT construction relies on a direct mapping of points to their respective voxels, ensuring a strictly $\mathcal{O}(n)$ complexity.

D. Collision Checking Latency

In the collision checking evaluation, we identified the most query-intensive planning problem for the Fetch in the *cage* scene and recorded the corresponding queries. We used these queries to benchmark the environment data structures. All structures are constructed using the same SCDF-filtered point cloud. Reported results come from the average of 10 independent trials. Table II summarizes the average latency per collision query. Our serial and SIMD implementations outperform CAPT by 6.69× and 1.35×, respectively, and the serial version surpasses NanoFLANN by 3.10×. Unlike k-d tree-based structures that require $\mathcal{O}(\log n)$ traversals for n points, the MVT provides $\mathcal{O}(1)$ collision checking complexity per query.

TABLE II: Average Per-query Collision Checking Time (ns)

	NanoFLANN	CAPT	Ours
Serial	211.77	455.91	**68.14**
SIMD	–	24.17	**17.91**

E. Planning Latency on Sampling-based and Neural-based MP

Table III presents the mean and standard deviation of planning latency for the RRT-C planner across all problems. Using the proposed MVT for collision checking resulted in an average speedup of 1.94× and more consistent performance compared to CAPT. Notably, robots represented by a larger number of spheres, such as the Fetch, required a higher volume of collision queries during planning, highlighting the constant-time collision checking advantages of the MVT.

TABLE III: Planning Latency of Sampling-based MP (ms)

	CAPT		Ours	
	Avg.	Std.	Avg.	Std.
UR5	1.26	3.02	0.96	2.13
Panda	1.22	1.86	0.99	1.39
Fetch	98.60	161.89	29.89	64.73

Table IV compares the planning latency of MPNet and RRT-C for the Fetch in the *bookshelf_thin* scene. The results indicate that while the MVT provided a 1.45× speedup for RRT-C, its impact on MPNet was marginal. This discrepancy arose because MPNet's planning latency is dominated by model inference rather than collision queries. Furthermore, unlike sampling-based planners that explore the search space via random sampling, neural-based planners aim to predict high-quality feasible states, resulting in significantly fewer collision queries. Even with NPU acceleration, MPNet remained slower than the CPU-based RRT-C, suggesting that reducing inference latency on OPi single-board computers remains a challenge for neural-based motion planning.

TABLE IV: Planning Latency of Sampling-based and Neural-based MP (ms)

	RRT-C		MPNet	
	OPi	RPi	OPi	RPi
CAPT	1.67	1.56	42.92	5.20
Ours	**1.13**	**1.10**	42.90	5.19

F. Environment Data Structure Memory Usage

We compare the memory usage of the constructed environment structures for MVT, CAPT, and a dense-grid baseline. The dense-grid baseline employs the same adaptive workspace voxelization as MVT but maintains a 3D array representing every voxel. Each voxel stores three SIMD-aligned `std::vector` instances for point coordinates. On 64-bit systems, this incurs a fixed 72-byte overhead per voxel regardless of point count, as each vector instance requires 24 bytes. MVT's memory footprint consists of multilevel tables, voxel structures, and the point coordinates stored in a SoA with SIMD-aligned padding. CAPT's memory footprint includes k-d tree internal nodes, leaf nodes, and the SIMD-aligned affordance sets. We experimented on a desktop CPU platform with a 32-byte SIMD alignment requirement.

Table V summarizes the average memory usage across the first valid problem from each scene. On average, MVT consumed only 2.27% of the memory required by CAPT and 20.06% of the dense-grid baseline, highlighting that MVT is not only faster in execution but also more memory-efficient than CAPT. Furthermore, this efficiency demonstrates MVT's ability to exploit the inherent sparsity of point clouds, avoiding the massive redundancy of dense grids. Notably, the dense-grid baseline exhibited lower memory usage in the Fetch manipulator scenario because its larger $r_{\max}$ leads to a coarser grid resolution—and thus fewer total voxels—under our voxelization strategy.

TABLE V: Data Structure Memory Usage (KiB)

	CAPT	Dense Grid	Ours
UR5	4,394.07	1,803.30	**144.36**
Panda	4,109.26	1,624.06	**128.54**
Fetch	37,606.83	199.46	**97.17**

G. End-to-end Motion Planning Latency

Fig. 7 illustrates the end-to-end latency, including the entire pipeline—filtering, construction, planning, and shortcut simplification—on both the desktop and the OPi. Note that

the execution time is plotted on a logarithmic scale for better visualization. To evaluate both typical and worst-case performance, the figure shows the mean and 95th percentile. On average, our VCC achieves speedups of 7.71× and 4.23× over CAPT on the desktop CPU and embedded OPi platform, respectively. While the end-to-end latency for complex manipulators, such as the Fetch, may not consistently stay below the 60 FPS threshold, VCC significantly enhances performance across all configurations. These results demonstrate that by leveraging VCC, high-performance motion planning can be extended to more complex robots, larger point clouds, and more resource-constrained hardware.

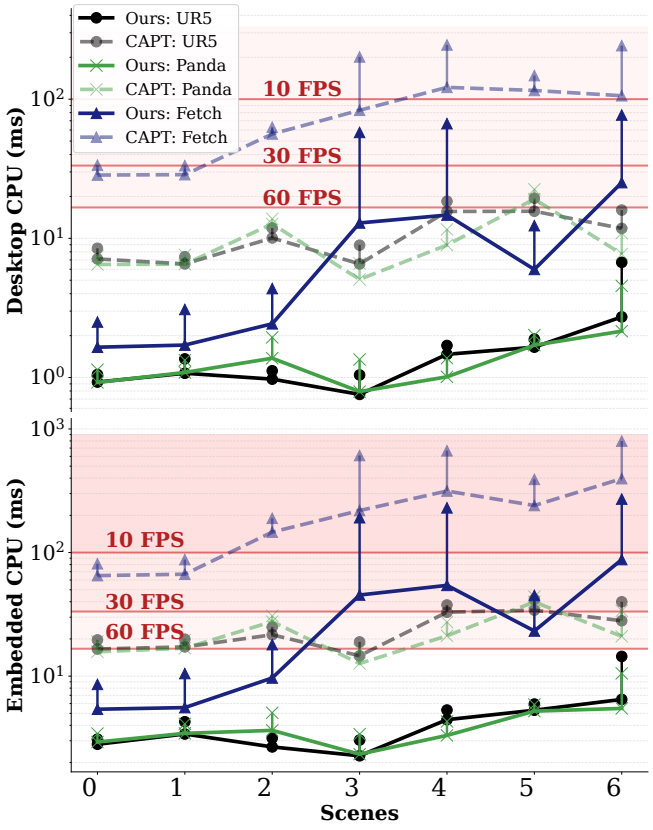


Fig. 7: End-to-end planning latency comparison on different platforms, illustrating both the mean and 95th percentile values.

VI. Conclusion

This work presents VCC, a novel collision checking framework that accelerates every stage of the motion planning pipeline, with a primary focus on the pre-planning stage—point cloud filtering and environment structure construction. To create a real-time, dynamic motion planning system, we first developed CenterVox, which quickly cleans point cloud data and achieves an average 3.63× speedup. Additionally, we designed adaptive workspace voxelization and the cache-friendly Multilevel Voxel Table (MVT), providing a 220.48× speedup in environment structure construction and reducing memory usage by 97.73% over the state-of-the-art framework. Notably, our proposed SIMD-parallel collision checking method accelerates the sampling-based planning time by 1.94×. These accumulated optimizations enable VCC to deliver an end-to-end speedup of up to 7.71× on the desktop CPU platform and 4.23× on the embedded device, empowering a wider range of hardware platforms to handle complex dynamic motion planning problems and massive point clouds in real time.

References

[1] C. W. Ramsey, Z. Kingston, W. Thomason, and L. E. Kavraki, "Collision-affording point trees: SIMD-amenable nearest neighbors for fast collision checking," in *Robotics: Science and Systems*, 2024.

[2] W. Thomason, Z. Kingston, and L. E. Kavraki, "Motions in microseconds via vectorized sampling-based planning," in *IEEE ICRA*, 2024, pp. 8749–8756.

[3] Shenzhen Xunlong Software CO., Limited, *Orange Pi 5B User Manual v1.5.1*, Orange Pi, 2023, accessed: 2025-03-05. [Online]. Available: http://www.orangepi.org

[4] S. LaValle, "Rapidly-exploring random trees: A new tool for path planning," *Research Report 9811*, 1998.

[5] J. J. Kuffner and S. M. LaValle, "Rrt-connect: An efficient approach to single-query path planning," in *IEEE ICRA*, vol. 2, 2000, pp. 995–1001.

[6] L. Kavraki, P. Svestka, J.-C. Latombe, and M. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *IEEE Trans. on Robot. and Autom.*, vol. 12, no. 4, pp. 566–580, 1996.

[7] A. H. Qureshi and M. C. Yip, "Deeply informed neural sampling for robot motion planning," in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2018, pp. 6582–6588.

[8] A. H. Qureshi, A. Simeonov, M. J. Bency, and M. C. Yip, "Motion planning networks," in *2019 ICRA*, 2019, pp. 2118–2124.

[9] A. H. Qureshi, Y. Miao, A. Simeonov, and M. C. Yip, "Motion planning networks: Bridging the gap between learning-based and classical motion planners," *IEEE Transactions on Robotics*, vol. 37, no. 1, pp. 48–66, 2020.

[10] A. e. Hornung, "Octomap: An efficient probabilistic 3d mapping framework based on octrees," *Autonomous robots*, vol. 34, no. 3, pp. 189–206, 2013.

[11] H. Oleynikova, Z. Taylor, M. Fehr, R. Siegwart, and J. Nieto, "Voxblox: Incremental 3d euclidean signed distance fields for on-board mav planning," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2017, pp. 1366–1373.

[12] V. Reijgwart, A. Millane, H. Oleynikova, R. Siegwart, C. Cadena, and J. Nieto, "Voxgraph: Globally consistent, volumetric mapping using signed distance function submaps," *IEEE Robotics and Automation Letters*, vol. 5, no. 1, pp. 227–234, 2019.

[13] M. Muja and D. Lowe, "Flann-fast library for approximate nearest neighbors user manual," *Dept. Computer Science, Univ. British Columbia, Vancouver, BC, Canada*, vol. 5, no. 6, pp. 12–29, 2009.

[14] J. L. Blanco and P. K. Rai, "nanoflann: a C++ header-only fork of FLANN, a library for nearest neighbor (NN) with kd-trees," https://github.com/jlblancoc/nanoflann, 2014.

[15] J. Ichnowski and R. Alterovitz, "Concurrent nearest-neighbor searching for parallel sampling-based motion planning in so (3), se (3), and euclidean spaces," in *International Workshop on the Algorithmic Foundations of Robotics*. Springer, 2018, pp. 69–85.

[16] C. Chamzas, C. Quintero-Peña, Z. Kingston, A. Orthey, D. Rakita, M. Gleicher, M. Toussaint, and L. E. Kavraki, "Motionbenchmaker: A tool to generate and benchmark motion planning datasets," *IEEE Robotics and Automation Letters*, vol. 7, no. 2, p. 882–889, 2022.

[17] L. Jaillet, A. Yershova, S. M. LaValle, and T. Siméon, "Adaptive tuning of the sampling domain for dynamic-domain rrts," in *2005 IEEE/RSJ IROS*. IEEE, 2005, pp. 2851–2856.

[18] J. J. Kuffner and S. M. LaValle, "An efficient approach to path planning using balanced bidirectional rrt search," Robotics Inst., Carnegie Mellon Univ., Pittsburgh, PA, Tech. Rep. CMU-RI-TR-00-05, 2000.

[19] R. Pi, "Raspberry pi 5," 2025, accessed Sept, 9, 2025. [Online]. Available: https://reurl.cc/3MWXr9

[20] HAILO, "Hailo-8 ai accelerator," 2025, accessed Sept, 9, 2025. [Online]. Available: https://hailo.ai/products/ai-accelerators/hailo-8-ai-accelerator/