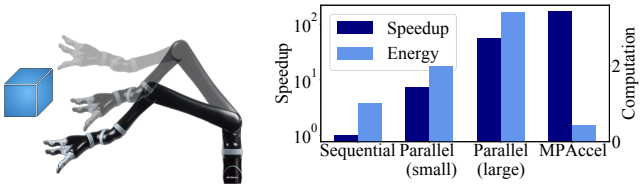
Energy-Efficient Realtime Motion Planning

Deval Shah, Ningfeng Yang, and Tor M. Aamodt
Department of Electrical and Computer Engineering, University of British Columbia
Vancouver, BC, Canada
devalshah@ece.ubc.ca,nxyang@ece.ubc.ca,aamodt@ece.ubc.ca

ABSTRACT

Motion planning is a fundamental problem in autonomous robotics with real-time and low-energy requirements for safe navigation through a dynamic environment. More than 90% of computation time in motion planning is spent on collision detection between the robot and the environment. Several motion planning approaches, such as deep learning-based motion planning, have shown significant improvements in motion planning quality and runtime with ample parallelism available in collision detection. However, naive parallelization of collision detection queries significantly increases computation compared to sequential execution. In this work, we investigate the sources of redundant computations in coarse-grained (inter-collision detection) and fine-grained (intra-collision detection) parallelism. We find that the physical spatial locality of obstacles results in redundant computation in coarse-grained parallelism. We further show that the primary sources of redundant computation in fine-grained parallelism are *easy cases* where objects are far apart or significantly overlapping. Based on these insights, we propose MPAccel to improve the energy efficiency of parallelization in motion planning. MPAccel consists of SAS, a Spatially Aware Scheduler for coarse-grained parallelism, and CECDUs, Cascaded Early-exit Collision Detection Units for fine-grained parallelism. SAS results in 7× speedup using 8× parallelization with 6% increase in the computation compared to 3.7× speedup with 83% increase in computation for naive parallelization. CECDU can perform collision detection in $46-154$ cycles for a robot with 6 degrees of freedom. We evaluate MPAccel to execute a state-of-the-art learning-based motion planning algorithm. Our simulations suggest MPAccel can achieve real-time motion planning for a robot with 7 degrees of freedom in 0.014ms-0.49ms with an average latency of 0.099ms compared to 1.42ms on a CPU-GPU system.



(a) (b)

Figure 1: (a) Motion planning for a Kinova Jaco2 robotic arm to reach the goal position while avoiding collision with surroundings. (b) Comparison of the speedup and energy efficiency for different execution modes on ASIC hardware. Small and large represent the scale of parallelization. Experimental methodology is described in Section 6.

CCS CONCEPTS

• **Computing methodologies** → **Robotic planning**; **Parallel algorithms**; • **Hardware** → **Application-specific VLSI designs**.

KEYWORDS

Robotics, Hardware acceleration, Motion planning, Collision detection

ACM Reference Format:
Deval Shah, Ningfeng Yang, and Tor M. Aamodt. 2023. Energy-Efficient Realtime Motion Planning. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23), June 17–21, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3579371.3589092

1 INTRODUCTION

Motion planning is a crucial task for autonomous robots. The goal of this task is finding a path through a robot's physical environment to some end goal while avoiding collisions with obstacles along the way (Figure 1a). Robotics tasks, including object manipulation, footstep planning, and full-body movement, require motion planning. Motion planning is computationally intensive with demands that increase rapidly with the number of degrees of freedom (DOF) of the robot and environment complexity [46]. As robots need to react to moving objects in their environment motion planning must be accomplished within strict real-time constraints. The energy efficiency of motion planning is important in increasing the operation time of mobile robots. Motion planning accelerators suitable for real-time motion planning can contribute to 15%-50% of its total power consumption [24, 31, 33]. Thus, improving the performance and efficiency of motion planning is important to enabling the deployment of robotics in challenging environments and tasks. Robots with higher

ISCA '23, June 17–21, 2023, Orlando, FL, USA.

ACM ISBN 979-8-4007-0095-8/23/06…$15.00
https://doi.org/10.1145/3579371.3589092

degrees of freedom (e.g., the 7-DOF Baxter robotic manipulator [45]) can perform more than one task or complex tasks in a cluttered environment. While the cost of high-DOF robots is decreasing, the latency of motion planning for high-DOF autonomous robots is currently a major impediment to its deployment [51, 54, 56].

Several acceleration approaches have been proposed for collision detection and traditional motion planning algorithms to meet the real-time computation requirements, including on GPUs [4, 15], FPGAs [2, 33, 48], and ASICs [3, 22, 29, 32, 52, 58]. Bakhshalipour et al. [3] proposed a voxelized robot-environment collision detection approach. Jia et al. [22] proposed a mapping accelerator that supports collision detection between a voxel and environment. Bakhshalipour et al. [3] introduced RACOD and proposed a speculative parallelism-based accelerator for path planning of robots with 2 or 3 DOFs (e.g., autonomous cars or drones). However, the proposed approach does not apply to motion planning algorithms for robots with higher DOFs [3]. Other works have proposed hardware accelerators for sampling-based motion planning algorithms suitable for high-DOF robots [29, 32, 33, 52, 58]. However, these accelerators are not scalable for complex robots and environments. The reason is that the underlying motion planning algorithm (i.e., probabilistic roadmaps) used in these works requires significantly more computation as the complexity of robotic tasks and environment increases. For example, a probabilistic roadmaps-based motion planning accelerator suitable for dynamic environment and challenging tasks requires more than 40MB on-chip memory or 40GBPS off-chip memory bandwidth [29, 32].

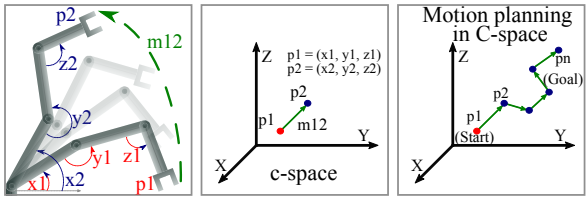
There have been improvements in the field of motion planning algorithms. More recently, informed sampling-based motion planning algorithms have exhibited significant improvement in the computation requirement, path quality, and success rate of motion planning compared to conventional algorithms [21, 27, 43, 44, 57]. These approaches use different heuristics to improve the sampling efficiency. For example, learning-based motion planning approaches use neural networks for sampling [21, 27, 43, 44]. MPNet [43], a state-of-the-art motion planning approach, has shown 15× speedup on CPU and 40% improvement in the path quality compared to the traditional sampling-based motion planning algorithms. Though such more efficient algorithms have been explored and studied at the software level, their architectural implications have not been studied.

For sampling-based motion planning algorithms, collision detection between the robot and the environment consumes $\sim$ 90% of execution time [4, 33]. Sampling-based motion planning provides an approximate trajectory for the robot by finding a set of intermediate positions that the robot can take to reach the end goal. For example, a neural network provides this approximate trajectory in learning-based motion planning. The motion between two intermediate positions is generated by a local planner. Typically, the linear interpolation between two positions is used as a local planning approach [26, 42]. Collision detection is used to find which intermediate positions provide collision-free motion and optimize this trajectory. Here, short motions between intermediate positions can be checked for collision in parallel. Similarly, a short motion is discretized into several robot positions, and collision detection for each position is performed, providing inter-collision detection parallelism. Further, for each collision detection query, different parts of the robot and environment can be checked in parallel for a collision, and the parallelism available in the collision detection algorithm can be used, providing fine-grained parallelism.

There is ample coarse- and fine-grained parallelism in sampling-based motion planning. However, we find that naive parallelization of collision detection is work-inefficient, which significantly increases computation and energy consumption. A parallel algorithm is said to be work-efficient if the amount of work done by it is asymptotically equal to the work performed by the fastest sequential algorithm for the same problem and several works have focused on this problem for different algorithms [5, 39, 49]. Collision detection queries' serial evaluation terminates once the desired outcome is found (e.g., a colliding position is found) and discards subsequent queries. However, a parallel configuration executes multiple queries simultaneously, resulting in redundant operations compared to serial evaluation. Figure 1b compares the speedup and computation for sequential and parallel evaluation on specialized hardware. Parallel evaluation results in 50× speedup with 3.4× computation compared to sequential evaluation.

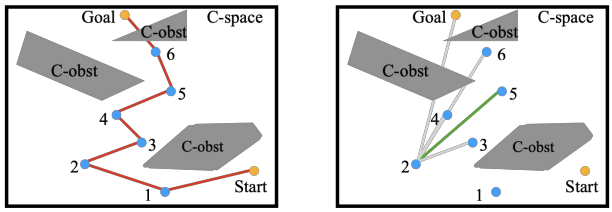
In this work, we analyze the sources of redundant computation in coarse- and fine-grained parallelization and propose an algorithm-hardware optimization approach *MPAccel* to improve the energy efficiency and execution time of motion planning. MPAccel consists of a novel scheduler and collision detection units. We call the former Spatially Aware Scheduler (SAS) and the latter Cascaded Early-exit Collision Detection Units (CECDUs). SAS exploits coarse-grained parallelism to improve work efficiency. CECDUs exploits fine-grained parallelism. The primary source of redundant computation in coarse-grained parallelization is the physical locality of objects in the environment. Collision detection outcomes for spatially nearby robot positions are likely to be similar due to the physical locality of obstacles. Thus, scheduling distant positions in a batch to cover more space is crucial to reduce redundant computation. SAS groups spatially distant computation in a batch to improve the work efficiency of parallel execution. We further show that easy collision detection cases with significantly far or overlapping objects contribute the most to redundant computations in fine-grained parallelism. We propose a cascaded early-exit unit, CECDU, that filters such easy collision-free and colliding cases by performing low-compute collision tests using simple geometric primitives (e.g., spheres) bounding and inscribing an object (e.g., robot's link). CECDU performs a precise collision detection only if required.

SAS results in 7× speedup using 8 collision detection cores (e.g., CECDU) compared to sequential execution, with 6% increase in energy. CECDU can perform collision detection in $46 - 154$ cycles for a 6-DOF robot. MPAccel enables real-time motion planning for 7-DOF robot using a learning-based



(a) (b) (c)

Figure 2: (a) Spatial poses and motion of a 2D robot with three DOFs (x, y, z), (b) represents these poses and motion in the robot's C-space, and (c) shows a path made of multiple intermediate poses in the C-space.



(a) Feasibility checking (b) Path optimization

Figure 3: Example of different collision detection phases in sampling-based motion planning. Here the planning is shown in the C-space. C-obst represents environmental obstacles projected in the C-space.

motion planning algorithm in 0.014ms-0.49ms with 0.099ms on average. In summary, we make the following contributions:

- We study the sources of redundant computation in coarse-grained and fine-grained parallelism in motion planning.
- We propose MPAccel; it consists of a Spatially Aware Scheduler (SAS) to handle coarse-grained parallelism, and Cascaded Early-exit Collision Detection Units (CECDUs) to handle fine-grained parallelism.
- We evaluate a learning-based motion planning algorithm, MPNet, on the proposed MPAccel.

2 BACKGROUND AND MOTIVATION

This section briefly summarizes sampling-based motion planning and collision detection.

2.1 Sampling-based Motion Planning

Motion planning aims to find a collision-free path between the start and end pose of the robot. Motion planning is typically performed in a robot's configuration space (C-space). The C-space of a robot has the same dimensions as its degrees of freedom (DOFs), where each dimension represents the range of values for a DOF (e.g., the angle of a rotational joint). Figure 2a represents a 3-DOF robot, and Figure 2b represents its C-space. A point in the C-space represents a pose/position of the robot, represented by angles of three joints (x, y, z). The straight line between $p1$ and $p2$ in the C-space (e.g., $m12$) corresponds to a short motion between corresponding poses in the physical space. This motion can be a sequence of poses corresponding to a linearly interpolated line in the C-space [26, 42] (Figure 6a).

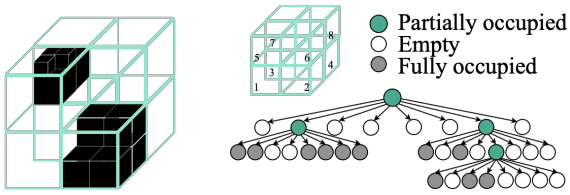
Motion planning complexity increases exponentially with the DOFs. Hence, sampling-based motion planning algorithms are widely used for motion planning, in which the C-space is sampled coarsely to find intermediate poses between the start and end poses, as shown in Figure 2c. Two adjacent poses p_i and p_{i+1} are connected by a short motion. The motions between adjacent poses in $\{p_1, p_2, ..., p_N\}$ must be collision-free for this path to be collision-free.

The C-space can be sampled in an informed manner to improve motion planning efficiency. More recently, deep learning-based informed sampling has shown significant improvement in the runtime and path quality [21, 27, 43]. These approaches use neural networks for sampling intermediate poses between the start and end pose. We profile a state-of-the-art learning-based motion planning algorithm, MPNet, on a CPU-GPU system. Our profiling results show that neural network inference (GPU) and collision detection (CPU) consume 2% and 95% of total time, respectively. Figure 3 represents different phases of a sampling-based motion planning algorithm. In the "Feasibility checking" phase, all motions in the final path are checked for collision. In the "Path optimization" phase, a shortcutting algorithm is used to smoothen the trajectory by removing redundant intermediate poses/-points [14, 16, 19, 43, 47, 60]. For example, in a greedy shortcutting algorithm [43], linear motions between p_2 and $\{p_3, p_4, ..., p_N\}$ are checked for collision. If a motion between from p_2 to p_i is collision-free, poses $p_3, p_4, ..., p_{i-1}$ are considered redundant as p_2 and p_i are connected. Removal of such redundant intermediate poses results in a smoother path.

2.2 Collision Detection

Collision detection between a robot and its environment is a crucial part of motion planning. Collision detection finds if the robot collides with objects in the environment for its given pose. Collision detection for a motion can be performed by discretizing the motion into multiple discrete poses, and these poses are checked for collision. A key design consideration for the collision detection algorithm is the geometric representation. The geometric representation decides the data structures and primitives used to store the space occupied by the robot or obstacles. In the simplest form, an object can be represented as a set of primitives such as spheres, cubes, boxes, or oriented boxes. For example, the environment can be discretized into fixed-size cubes (also known as voxels). Partially or fully occupied voxels are set to 1, and the rest are set to 0. The shape and size of primitives determine the representation accuracy and the storage requirement.

Bounding volume hierarchies have been proposed to reduce collision detection time and storage requirements [25]. In this approach, a tree-type structure represents the space occupied by objects. Figure 4b shows an example of an octree representation of the occupied voxels from Figure 4a. In octree representation, each node divides the space into octants and stores the occupancy information (empty/fully occupied/-partially occupied) of all octants. Only partially occupied



(a) Spatial division of the occupied space **(b) Octree structure**

Figure 4: Octree representation (figure adapted from [29]).

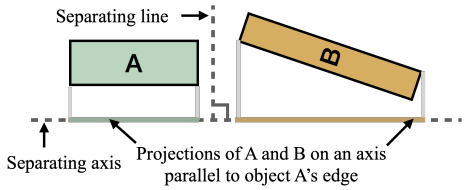


Figure 5: Separating axis test to find if two convex objects overlap. Here, a separating axis is found as projections of objects A and B on this axis do not overlap.

octants are further divided into smaller octants. Collision detection between an object and the space occupied by this octree is performed by traversing the tree. For each visited node, collision detection between the object and bounding boxes corresponding to occupied octants is performed. The node corresponding to an octant is further traversed if a collision is found with the octant.

Intersection Test: Intersection test between two primitive shapes is fundamental to collision detection. The separating-axis test can be used to perform an intersection test between any two convex objects [13]. Two objects do not overlap if there exists a line or plane that separates two 2D or 3D objects. The separating axis is any line perpendicular/orthogonal to the separating line/plane. Figure 5 gives an example of a separating-axis test between an axis-aligned bounding box A (AABB) and an oriented bounding box B (OBB). Here, projections of A and B on the axis parallel to object A's edge do not overlap, which shows that a line perpendicular to this axis separates the two objects. A separating axis is found in this case. For 3D objects, a plane perpendicular to the separating axis separates two objects. Depending on the shape of objects, candidates for possible separating axes can be determined. For example, there are 15 separating axis candidates to perform an intersection test between two 3D OBBs [17], as there are 15 possible separating planes. A plane parallel to a face of either of the OBBs is a candidate for separating planes, which gives six separating axes (three unique face orientations per OBB). Further, a plane spanning the axes of two edges, one from each OBB, is also a candidate for separating planes. The separating axis corresponding to this plane can be found using the cross-product of the axes of these two edges. Since each box has three unique edge directions, there are 3×3 candidates for separating axes. Two objects are determined to be colliding if none of these 15 axes is a separating axis.

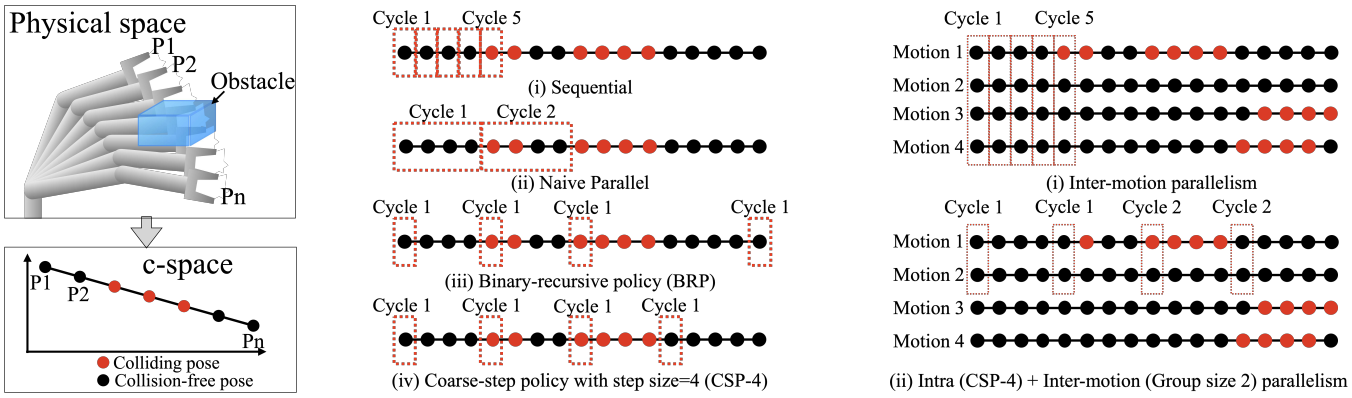
3 SPATIALLY AWARE SCHEDULER

This section summarizes our coarse-grained parallelism (i.e., inter-collision detection query parallelism) analysis. Further, the proposed approach to exploit coarse-grained parallelism, Spatially Aware Scheduler (SAS), is explained in detail.

Each phase of motion planning provides coarse-grained inter-motion and intra-motion parallelism between collision detection queries (Section 2.1). We first perform a limit study to analyze the impact of the number of collision detection units (CDUs) (i.e., degree of parallelization) on the number of collision detection tests (i.e., a measure of work efficiency) and collision detection runtime. Benchmarks used for this study are described in Section 6. The limit study assumes zero cycle latency for the scheduler and latency of one cycle for a collision detection query. We observe that the number of collision tests increases by 2.4× with 12.4× reduction in the runtime for 16× naive parallelization. As the degree of parallelization increases, the number of collision tests and energy increase with speedup.

The key reason behind this mismatch in the number of collision detection runs between sequential and parallel evaluations is that once a collision is detected for any robot pose along a motion, there is no need to perform collision detection for the following poses from this motion. Figure 6a represents a robot's pose in the physical space and C-space. Black/red dots represent the discrete poses checked for collision detection of this motion. Figure 6b.i represents the sequential evaluation of collision detection for a motion. A collision is detected and completed for this motion at cycle 5. However, parallel evaluation using 4 CDUs, as shown in Figure 6b.ii, results in more collision detection queries compared to sequential evaluation. The effect becomes more pronounced as the scale of parallelization increases, as shown in Figure 7 (NP). Another approach is to use inter-motion parallelism. However, different motions are not necessarily independent tasks in motion planning. For example, in shortcutting for motion planning, the goal is to find the first collision-free motion from a pool of motions [16, 19, 47, 60]. In Figure 6c.ii, collision-detection for motion 3 and 4 is redundant as motion-2 is collision-free. Inter-motion parallelism reduces redundant computation compared to naive parallelism. However, its effectiveness reduces as the scale of parallelization increases (See MNP in Figure 7). Thus a combination of intra- and inter-motion parallelization is required.

The increase in redundant computation for intra-motion parallelism is due to the physical spatial locality of the robot's poses and obstacles. There is considerable overlap between the physical space covered by the poses of the robot corresponding to adjacent points, as shown in Figure 6a. Hence, in most cases, collision detection results for nearby poses (i.e., nearby points in Figure 6) are likely to have the same output. Naively grouping adjacent poses in a batch for parallelization degrades work efficiency, as spatially similar poses are checked together.



(a) (b) (c)

Figure 6: (a) represents a 2-DOF robot's motion in the physical and C-space. Here, a dot in the C-space represents the robot's pose in discretized motion. Collision detection for all discrete poses is performed for the motion's collision detection. (b)-(c) represents different scheduling approaches for sequential and parallel evaluation (4 CDUs) of collision detection queries. (b.ii)-(b.iv) are examples of intra-motion parallelism, (c.i) represents only inter-motion parallelism (referred to as multi-motion), and (c.ii) represents an example of intra+inter motion parallelism.

A remedy to this inefficiency is to schedule physically distant poses in a batch for parallel evaluation to cover more space. Based on this insight, we propose a Spatially Aware Scheduler (SAS), which schedules physically distant poses in a batch. We explore two scheduling policies for SAS.

We first explore a binary-recursive traversal-based scheduling policy. The difference between the indices of poses gives a measure of the physical distance between the poses in Figure 6a. Binary recursive scheduling policy (BRP) selects the order of poses using the binary-recursive algorithm, thus sampling the motion from coarse to fine. Figure 6b.iii represents scheduling using the binary-recursive algorithm, where poses with identifiers 0, N, $N/2$, and $N/4$ are selected in the first cycle. However, BRP requires maintaining a queue. We also explore a simpler scheduling strategy based on coarse steps. This coarse-step scheduling policy (CSP) uses a value greater than one as the step size to select poses in a batch. For example, for a step size of four, points 1 to N are scheduled in the order of $\mathbf{0}, 4, 8, ..., \mathbf{1}, 5, 9, ..., \mathbf{2}, 6, 10, ..., \mathbf{3}, 7, 11, ..., N$. Figure 6b.iv represents CSP for a step size of 4. CSP schedules discrete poses in a motion in a coarse-to-fine manner to avoid *physical-locality-induced redundant collision checks.*

We combine inter-motion and intra-motion parallelism to reduce redundant computation in the parallel execution of collision detection queries. We propose *Multi-motion Coarse-step Scheduling Policy* (MCSP) for SAS that combines CSP with inter-motion parallelism to take advantage of both kinds of parallelism. In MCSP, a group of motion, determined by the group size for inter-motion parallelism, is considered for scheduling. Within a motion, the order of poses is selected based on CSP. Figure 6c.ii represents the MCSP approach for a group size of two and a step size of four.

Figure 7 represents a limit study on the number of collision detection queries and runtime for different scheduling approaches. We also compare with a random selection of points within a motion. We compare different combinations of

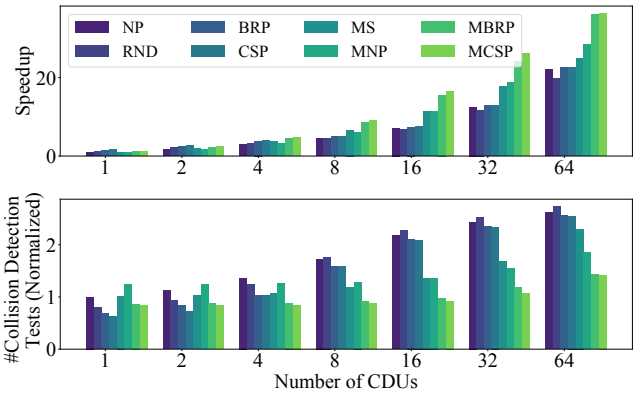


Figure 7: Limit study on the effect of scheduling policies on the number of collision detection cycles and runs for different numbers of CDUs. NP: Naive parallel, RND: Random scheduling; BRP: Binary recursive policy, CSP: Coarse-step policy; prefix M represents inter-motion parallelism. MS represents only inter-motion parallelism.

{Seq (S), Naive Parallel (NP), Random (RND), Coarse-step Policy (CSP), Binary Recursive Policy (BRP)} and {With inter-motion parallelization (M), Without inter-motion parallelization (M omitted)}. As shown in the figure, naive or naive+inter-motion parallelization is not sufficient for energy efficiency and speedup for a higher number of CDUs. Furthermore, CSP results in faster collision detection than the ordered selection of poses for sequential evaluation (i.e., #CDU=1) because of the efficient exploration of the space covered by a motion. We also see that CSP performs very similarly to the BRP and translates to a simpler hardware/-software implementation as the binary recursive approach requires maintaining one or more hardware/software queues. In contrast, coarse-step-based scheduling can be implemented using registers and adders. The figure shows that MCSP can

achieve up to 13.5× speedup using 16 CDUs with only a 10.5% increase in the number of collision detection tests.

4 CASCADED EARLY-EXIT COLLISION DETECTION UNIT

Collision detection is a widely studied problem with applications in various fields. In motion planning, collision detection is used to find if the robot collides with its surroundings for a given pose. This section summarizes the intra-collision detection query parallelism analysis. The proposed Cascaded Early-exit Collision Detection Unit, CECDU, is explained in detail.

We consider mainly three factors for selecting the geometric representation and collision detection algorithm. The first factor we consider is the calculation of the robot's occupied space for a pose. Prior works precompute the robot's occupied space for different poses and store it in memory [29, 32, 34, 58]. However, such precomputation does not allow collision detection for arbitrary poses explored by the motion planning algorithms. Furthermore, the storage requirement increases with the complexity of the robot and its tasks [29, 32]. We consider on-chip calculation of the robot's occupied space for a given pose. Thus we rule out the use of bounding volume hierarchy (BVH) for the robot's geometric representation, as BVH tree generation is compute-intensive [13]. Based on this insight, we use a set of oriented bounding boxes (OBB) to represent the robot. As the robot changes its pose, each link goes through a rigid transformation, i.e., its orientation and translation change. The size of the bounding box for each link of the robot can be precomputed. At runtime, the robot's pose (e.g., angle of its joint) is used to find the orientation and center of these OBBs using trigonometric functions and matrix multiplication [12].

The second factor is the collision detection computation requirement for colliding and collision-free cases. We find that more than 95% of the collision detection tests are collision-free in motion planning benchmarks used in this work (Section 6). BVH tree-based representation reduces computation for collision-free cases, as collision detection can terminate if no collision is found at a node. As mentioned earlier, generating a BVH tree for a robot is expensive as more than 1000 poses are tested for collision for each motion planning query. However, the environment is updated only once for a motion planning query. Based on this observation, we use an octree representation of the environment (Section 2.2). Prior works have focused on mapping sensor data (e.g., point cloud, 2D images) to octree [20, 22, 55]. Jia et al. [22] proposed a mapping accelerator to build octree from point cloud data. Such mapping accelerators can be used to provide the environment's octree representation.

The third factor we consider is the scalability of the intersection test for higher precision or a larger environment. Collision detection between a robot (represented by a set of OBBs) and environment (represented by an octree) consists of multiple intersection tests between OBBs and AABBs (from the octree nodes). One simple approach includes rasterization

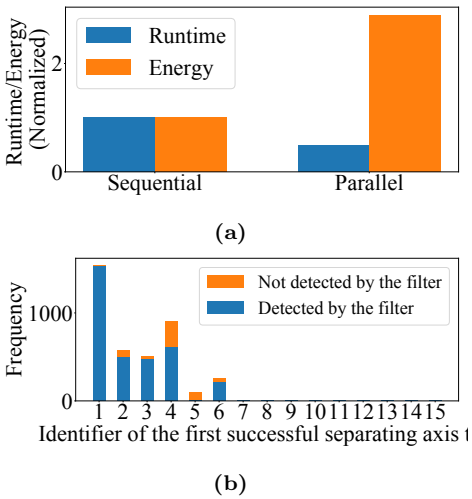
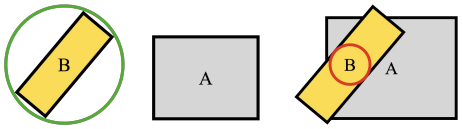


Figure 8: (a) Comparison of runtime (#cycles) and energy for sequential and parallel execution of the separating axis tests. (b) Distribution of the successful separating axis test identifiers for OBB-octree collision detection.

of OBB to a set of voxels. Collision detection is performed between voxels and environment octree. However, the number of voxels increases significantly with the resolution of rasterization. For example, we find that the number of intersection tests increases by $\sim 5\times$ when the discretization step size is decreased by half for OBBs of the Jaco2 robotic arm [24]. Moreover, this requires checking all voxels for collision-free cases. Considering this, we select the separating axis test for OBB-AABB intersection test. The separating axis test consists of multiplications and additions. The separating axis test allows an accurate intersection test between the OBB and the environment, reducing false positives (i.e., a collision-free pose is flagged as colliding).

For the OBB-AABB intersection test, 15 potential separating axes can be checked as explained in Section 2.2. Two objects overlap if none of the axes is a separating axis. All 15 tests can be performed in parallel to accelerate the intersection test. Figure 8a shows the number of multiplications performed (i.e., approximated energy) for sequential and parallel execution of separating axis tests for collision-free cases. Parallel execution results in approximately 3× increase in the energy. We find that the primary sources of this increase are collision-free cases, where a separating axis is found after testing the n^{th} separating axis candidate, and executing all 15 tests is redundant. To understand the inefficiency of parallel evaluation, we profile the distribution of identifiers of the separating axis that returns true (i.e., is a separating axis) in Figure 8b. We use collision detection tests between OBBs for random poses of Jaco2 robot [24] and octree for random environmental scenarios. In most cases, a separating axis is found in the first six axes. Based on this, we propose a three-stage execution mode, in which the 15 tests are divided as $6-5-4$ among three stages. A later stage is only executed if the previous stage returns false, i.e., a separating axis is not



(a) Bounding sphere (b) Inscribed sphere

Figure 9: Use of spheres to filter easy cases and reduce computation, where objects are far apart (a) or significantly overlapping (b). Objects are represented in 2D for clarity.

found. This modification decreases multiplication operations by 1.5× compared to fully parallel execution.

Furthermore, as shown in Figure 8b, for most collision-free cases, the first axis returns true for the separating axis test. We find that in most cases where the objects are far apart, a separating axis is found in the first few separating axis candidates. Prior works have proposed to use a computationally simple intersection test between bounding spheres of OBBs before performing a detailed intersection test [9]. Figure 9a gives an example of the bounding sphere for an OBB. The intersection test between a sphere and an AABB requires three multiplications compared to 81 for checking all 15 separating axes for the OBB-AABB intersection test. The blue bars in Figure 8b show the fraction of cases filtered by a sphere-AABB test. The majority of the intersection tests that find a separating axis in the first test and hurt the energy efficiency of parallel execution can be filtered by the bounding sphere-AABB test.

Further, we find that after applying the bounding sphere-based filter, ∼ 80% of the operations are used by colliding cases for OBB-AABB intersection tests. We find that in ∼ 85% of colliding cases, the AABBs correspond to the first and second levels of the octree, with 1/2 or 1/4 length of the environment's extent. If such large AABBs intersect with considerably smaller OBBs, it is likely that the OBB significantly or entirely overlaps with the AABB. We find that an intersection test between an AABB and the inscribed sphere of an OBB can efficiently find such colliding cases with fewer operations. An inscribed sphere is the largest sphere inside a shape that touches its edges (Figure 9b).

Figure 10 represents the flowchart for the proposed intersection test for the proposed Cascaded Early-exit Collision Detection Unit (CECDU). The intersection tests are performed in a cascaded manner, exiting early if collision detection output is found. The function returns collision if a separating axis is not found after checking all 15 axes.

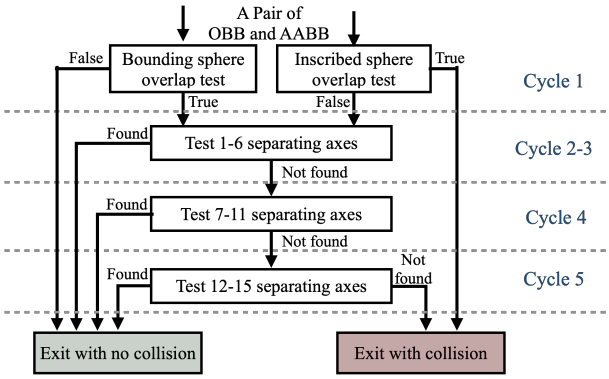


Figure 10: The flowchart for the proposed cascaded early-exit intersection test using bounding and inscribed-sphere filters and separating axis test.

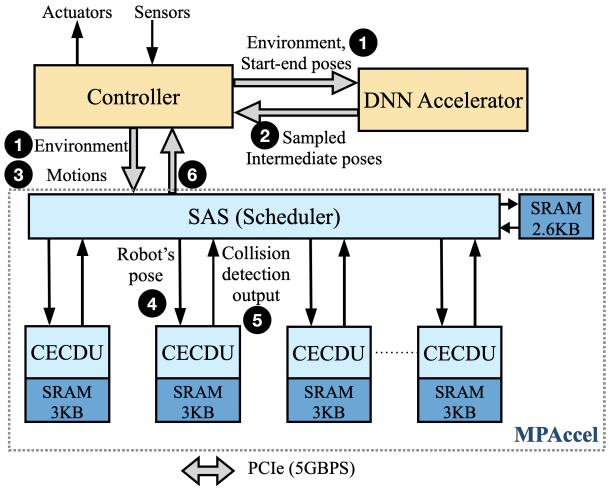


Figure 11: Architecture of MPAccel.

5 MPACCEL

This section describes the microarchitecture of MPAccel, a motion planning hardware accelerator. Figure 11 represents the architecture of the overall system for a learning-based motion planning algorithm (e.g., MPNet). The controller receives the environment's occupancy information from sensors and sends it to the DNN accelerator and SAS ❶. The controller receives a motion planning query consisting of the start and end goals. The controller runs the motion planning algorithm and offloads neural network inferences to the DNN accelerator and collision detection to MPAccel. A simple CPU core can be used as the controller. The DNN accelerator generates intermediate poses for a candidate trajectory between the start and the end goal ❷. The controller receives these intermediate poses and generates a set of motions for collision detection based on the motion planning algorithm ❸. The bandwidth of the bus between the CPU controller and DNN accelerator and SAS is assumed to be 5GBPS, which can be achieved by PCIe [35]. SAS receives the group of motions and function mode from the controller and schedules collision detection queries to the CECDUs ❹. The scheduler also collects results from the CECDUs ❺ and sends back the aggregated result to the controller once the execution finishes ❻.

CECDU calculates the robot's occupied space for different poses on-chip and uses octree representation for the environment, which reduces the storage requirement. We find that on-chip memory of 50KB is sufficient to solve motion planning for high-DOF robots (∼ 7) and complex environments. Hence, we use on-chip SRAM for storage, and MPAccel is

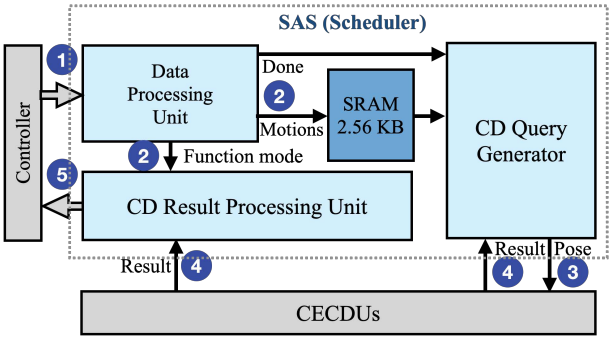


Figure 12: SAS microarchitecture.

not connected to DRAM. Prior motion planning accelerators deployed in real-world applications have proposed to use only on-chip memory to meet the energy and real-time constraints [32, 34, 50, 52].

5.1 Microarchitecture of SAS

Figure 12 represents the microarchitecture of SAS. SAS supports three types of function modes. The "Feasibility test" mode is used to find if all motions are collision-free. In this case, the scheduler stops once a collision for any pose is found. The "Connectivity test" mode is used to find if at least one motion is collision-free. In this case, the scheduler stops when one collision-free motion is found. This functional mode is useful for path optimization (Section 2.1). The "Complete test" mode is to get collision detection results for all motions.

The Data Processing Unit receives the data sent by the controller (1), consisting of metadata and motion data. The metadata includes the number of motions and the function mode. Motion data contains its start pose, the distance between two discrete poses, and the number of discrete poses. The Data Processing Unit processes and sends the received data to other units and SRAM (2). Collision detection (CD) Query Generator generates a discrete pose to be checked for collision and sends it to free CECDUs (3). The CD Query Generator implements the logic to order the poses for collision detection as per the MCSP (Section 3). The step size for MCSP is set to 8. Similarly, group size, i.e., the number of motions considered for inter-motion parallelism in MCSP, is set to 16. The CD Query Generator also receives the collision detection results from the CECDUs (4). It removes a motion from the scheduling list if an intermediate pose for this motion is found to be colliding. This way, it ensures not to schedule redundant work to the CECDUs. The CD Result Processing Unit receives collision detection results from the CECDUs (4). Depending upon the function mode, the result processing unit signals other units to stop operation and sends the result to the controller (5).

5.2 Microarchitecture of CECDU

The CECDU receives the robot's pose from the scheduler and performs collision detection between the robot and the environment. Figure 13 represents the CECDU microarchitecture. The OBB Generation Unit generates a set of OBBs

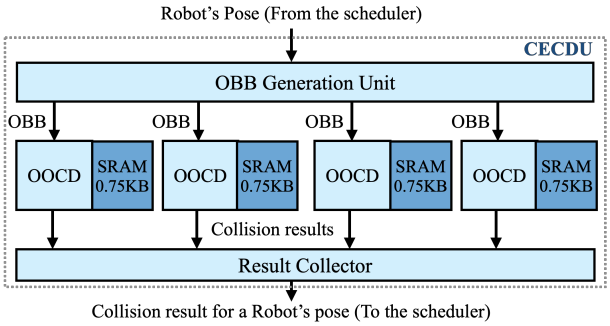
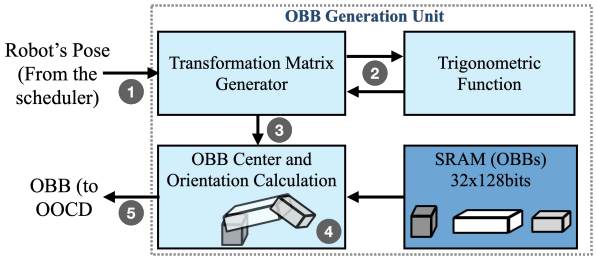
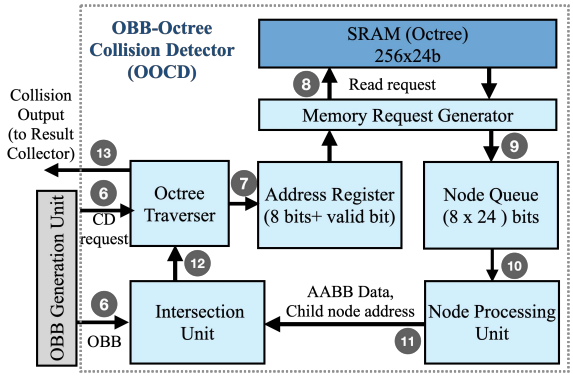


Figure 13: Microarchitecture of the CECDU.



(a) OBB Generation unit.



(b) OBB-octree Collision Detector.

Figure 14: (a) and (b) represents the microarchitectures of OBB Generation Unit unit and OBB-octree Collision Detector (OOCD).

representing the robot's occupied space for the given pose. The generated OBBs are sent to the OBB-octree Collision Detectors (OOCDs). Each OOCD performs collision detection between an OBB and the environment octree. The Result Collector receives results from all OOCDs and sends the final collision detection result (True or False) to the scheduler once collision detection for all OBBs of the robot is done. The Result Collector stops collision detection for a given pose if an OOCD returns true for collision detection between OBB-environment.

Figure 14a represents the microarchitecture of the OBB Generation Unit. For each link, the size of its bounding box, and the radii of upper and lower bounding spheres are stored in the SRAM. At runtime, the OBB Generation Unit receives the robot's pose (1). The transformation matrix generator calculates a transformation matrix (4×4) for each link

for this pose. This matrix is used to find the rotation and translation of a robot link's bounding box [12, 36]. A trigonometric function unit (2) is used for sine/cosine calculation for transformation matrix generation. We use a fifth-order approximation-based trigonometric function unit [11]. The trigonometric function unit is a 5-stage pipelined unit consisting of 8 multipliers, 3 adders/subtractors, and registers. The transformation matrix of each link is then sent to a matrix multiplier and adders (3). These ALUs calculate the center and orientation of the OBB for this link (4). Thus the OBB Generation unit generates a set of OBBs to represent the space occupied by the robot for its given pose and sends the OBBs to OOCDs for collision detection (5). Each OBB is represented by 17 values (16-bit each), 3 for its center. 3 for its size, 9 for its 3×3 orientation, and 2 for radii of the bounding and inscribed spheres.

Figure 14b represents the microarchitecture of the OOCD. The Octree Traverser (a finite state machine) receives the collision detection request from the OBB Generation Unit (6) and stores the root node's address (i.e., 0) to the Address Register and sets its valid bit (7). The SRAM stores the environment octree. The Memory Request Generator sends a memory request when the Address Register has a valid entry (8). The received data is then added to the Node Queue (9). The Node Queue can store 8 entries with 24 bits per entry. The Node Processing Unit receives this node information (10). Here, each node represents an AABB in the space and contains the occupancy information of its octants. An octant can be empty (i.e., no obstacle in this space), partially occupied, or fully occupied. The node information (24 bits) consists of occupancy information of all octants and the addresses for children nodes corresponding to partially occupied octants. The Node Processing Unit uses the node information and sends intersection queries for occupied octants (11). Each query consists of the AABB information corresponding to an octant and the address of the child node. An AABB is represented by its center and size (6×16 bits). The Intersection Unit performs an AABB-OBB intersection test using the proposed cascaded early-exit intersection test flow (Figure 10). We explore pipelined and multi-cycle designs for the Intersection Units. The Node Processing Unit sends one query every cycle for pipelined intersection units. For a multi-cycle unit, it sends a query when the Intersection Unit is free. The Intersection Unit consists of fixed-point multipliers and adders. The Octree Traverser receives the intersection test output (0/1), and the child node address (8 bits) (12). If a collision is found for a partially occupied octant, the address for the corresponding child node is stored in the Address Register (7). The Octree Traverser sends back the collision detection result (True or False) to the Result Collector (13) upon traversal completion.

6 METHODOLOGY

We evaluate the proposed hardware accelerator using a detailed microarchitectural simulator. We use 16-bit fixed-point number representation for poses, OBBs, and AABBs. We use Verilog to build the RTL models for the SAS and CECDU blocks. RTL implementations are synthesized using the Synopsys Design Compiler and the OpenRAM Memory Compiler [18] to estimate the area and power at 45nm technology using FreePDK design library [53]. The timing model of the microarchitectural simulator is based on the cycle latency measured from RTL models. We use the microarchitecture simulator for the evaluation. For OOCD, Our proposed method reduces energy by exiting early from the intersection test flow (Figure 10). Thus the proposed method reduces the switching activity. We built an accurate architectural power model to speed up power measurement of OOCD using the methodology described in [7]. We use RTL simulation to find out the leakage and dynamic power of individual blocks (multiplier, adder, mux) and use the microarchitectural simulator to estimate their activity factors.

We use Kinova Jaco2 [24] (6-DOF) and Baxter (7-DOF) robots for our evaluation. Both robotic arms consist of 7 links. We use ten environmental scenarios with 100 pairs of start and end goals per each environmental scenario. Each sample environment contains $5-9$ randomly placed cuboid-shaped obstacles. The size of these obstacles in each dimension is limited to $3\%-12\%$ of the environment's extent. These benchmarks are consistent with other work on motion planning and collision detection [29, 33, 43].

We use MPNet motion planning algorithm [43] to evaluate motion planning runtime for a 7-DOF robotic manipulator Baxter [45] using MPAccel. Note that MPNet is used as an example of a state-of-the-art sampling-based motion planning algorithm. We chose MPNet [43] as it has shown significant improvement in motion planning performance and has code available for evaluation. However, MPAccel can also be used for other sampling-based motion planning algorithms. We also evaluate collision detection and motion planning runtime on GPUs (NVIDIA Titan V and Pascal GPU with 256-CUDA cores) and CPUs (Intel i7-4771 8-core and Cortex A57 4-core) (Section 7.5).

7 EVALUATION

In this section, we present an evaluation of SAS and CECDU and motion planning using MPAccel.

7.1 Performance of the Scheduler

We propose a multi-motion coarse-step-based scheduling policy (MCSP) and corresponding microarchitecture in Section 3 and Section 5. A sampling-based motion planning algorithm consists of multiple phases (Section 2), where a set of motions is sent to the scheduler for collision detection in each phase. We use MPNet algorithm and report the average runtime and energy for an entire set of motions for different schedulers. The proposed CECDUs are used as collision detection units (CDUs) in this evaluation. The group size is set to 16 for inter-motion parallelism based on empirical results.

Figure 15 compares the performance and energy of different scheduling policies. The number of collision detection tests is used as a measure of energy. For given benchmarks, the

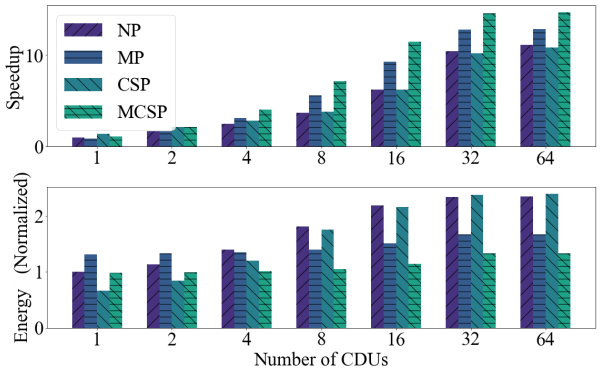


Figure 15: Comparison of different schedulers for coarse-grained parallelism. MCSP: Coarse-step policy + inter-motion parallelism (the proposed approach), NP: naive parallelization, CSP: Coarse-step policy, and MP: Only inter-motion parallelism.

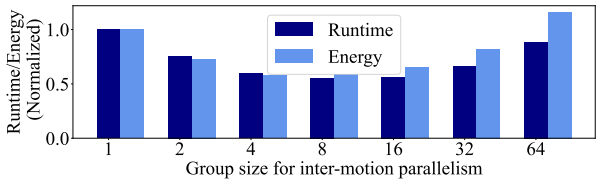


Figure 16: Effect of group size for inter-motion parallelism on runtime for MCSP.

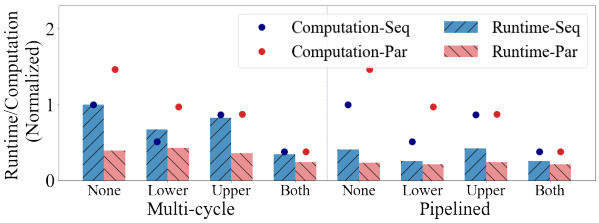


Figure 17: Comparison of the runtime and computation for sequential and parallel collision detection.

on-chip memory of OOCD is sufficient for collision tests, and there is no memory access coalescing across collision tests. Thus, energy increases linearly with the number of collision detection tests. This increase may not be linear if off-chip memory is required (e.g., for high-resolution environments such as those used in games or physics simulations). However, the insights of this evaluation still hold for such cases. As parallelism increases, MCSP and MP outperform NP and group "useful" work to be dispatched to the CDUs. For eight CDUs, MCSP results in 7× speedup with 6% increase in the energy compared to 3.7× speedup with 83% increase in energy for NP. Similarly, for 16 CDUs, MCSP results in 11.03× speedup with 22% increase in the energy compared to 6.2× speedup with 113% increase in energy for NP. The energy consumption for MCSP is slightly higher than predicted by the limit study (Section 3). The limit study assumes zero-latency scheduling and equal latency for collision detection. However, the CDU introduces a delay in receiving results for CD queries. In this delay, the scheduler might schedule more CD queries for a motion even though dispatched queries might return true for collision detection.

SAS can schedule up to one CD query per cycle. If the latency of CDUs is less than the number of CDUs, then increasing the number of CDUs does not help with speedup as the scheduler can not dispatch CD queries fast enough. Hence the speedup saturates as the number of CDUs reaches 32. However, as shown in the limit study with the ideal scheduler and CDU (zero latency) in Section 3, increasing the number of CDUs beyond 64 does not help with speedup and significantly increases energy consumption.

7.1.1 Effect of Inter-motion Parallelism: Group size represents the number of motions used for inter-motion parallelism. We compare the effect of group size on speedup and energy for MCSP in Figure 16 for eight CDUs. Smaller group size does not take advantage of inter-motion parallelism and results in higher runtime and energy. As the group size increases, inter-motion parallelism helps with improving the runtime by reducing redundant computation. The runtime and energy both increase for group sizes greater than 16. Collision detection for a group of motions can be run in different function modes (Section 5.1). For example, in the "Connectivity test" mode, once a collision-free motion is found, the subsequent motions can be discarded without checking for collision. More motions are scheduled together as the group size increases, and some motions that could have been discarded are also scheduled for collision detection. This results in the increased energy consumption for larger group sizes.

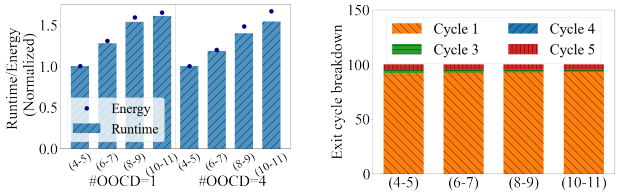
7.2 Performance of the CECDU

7.2.1 Evaluation of OBB-octree collision detection: We propose a cascaded early-exit collision detection unit (CECDU) for OBB-octree collision detection. First, we compare the latency/computation of the OBB-octree collision detection for parallel and sequential execution of the separating axis test without early-exit approach. Since the intersection test computation is dominated by multiplications, we use the number of multiplications as an estimate of computation. We also provide latency/computation for pipelined and multi-cycle versions. For the baseline version (without spheres filters), parallel execution results in 46% higher computation with 2.52× and 1.77× speedup compared to the sequential execution for multi-cycle and pipelined configurations, respectively. The proposed approach based on the bounding sphere closes the gap between the computation of sequential and parallel execution, and parallel execution provides $1.2 - 1.4\times$ speedup with 1.3% more computation. Furthermore, the proposed filter using an inscribed sphere to reduce computation for colliding cases reduces the computation by 33%. Both filters together provide $\sim$ 4.1× speedup compared to sequential execution (without filters) with 61% computation savings.

7.2.2 Collision detection latency for a robotic arm: We evaluate the collision detection latency for a 6-DOF (7 links) robotic arm using CECDU. Each CECDU can have more

Table 1: Collision detection latency for different CECDU configurations for Jaco2 robot with 7 links and 6 degrees of freedom.

	Single Intersection Unit		Four Intersection Units	
	Multi-cycle	Pipelined	Multi-cycle	Pipelined
Latency (Cycles)	154.4	137.5	54.8	46.3
Area (mm^2)	0.21	0.32	0.69	1.12
Power (mW)	92.6	100.8	215.7	248.7



(a) (b)

Figure 18: (a) represents the runtime and energy for single and four intersection units, and (b) represents the breakdown of the exit cycle from the proposed flow (Figure 10) for different environmental complexity (number of obstacles in this example).

than one OOCD, where each OOCD performs collision detection between an OBB (i.e., robot's link) and the environment. We evaluate two configurations of CECDUs. In the first configuration, the CECDU consists of a single OOCD unit that performs collision detection for all OBBs in a robot serially. If a collision is found between an OBB and the environment, subsequent OBBs are discarded. In the second configuration, the CECDU consists of four OOCD units for parallel collision detection. We also provide evaluation for pipelined and multi-cycle Intersection Units.

Table 1 compares the collision detection latency, area, and power for different combinations. Note that the use of four OOCD units does not reduce the runtime proportionally for two reasons. First, subsequent OBBs are not checked for collision in serial execution once a collision is detected. Second, the collision detection time for parallel intersection tests is dominated by the highest intersection test time across all units as we use synchronous scheduling. The end-to-end latency of intersection test is same for pipelined and multi-cycle units. However, the pipelined version can process more than 1 intersection tests at the same time in the pipeline. Therefore, the latency of robot-environment collision detection (which consists of multiple intersection tests) is lower for pipelined version. CECDU performs collision detection for the robotic arm in $46 - 154$ cycles.

We further analyze the effect of environmental complexity (e.g., number of obstacles) on CECDU. Figure 18a represents the robot-environment collision detection runtime and energy for environments with increasing number of obstacles. The runtime increases by $\sim$ 50% as the number of obstacles doubles. Figure 18b provides the breakdown of cycles required

Table 2: Area and power breakdown for hardware units.

Module	Area (mm^2)	Power (mW)
Scheduler	0.110	60.7
CECDU (with four multi-cycle OOCD)	0.694	215.7
OBB Transformation Unit	0.054	51.6
Octree Traversal Unit	0.029	16.7
Intersection Unit (Multi-cycle)	0.143	24.34
Intersection Unit (Pipelined)	0.251	32.57
MPAccel (Scheduler + 16 CECDUs)		
Config 1: 4 multi-cycle OOCDs/CECDU	11.21	3.51W
Config 2: 4 pipelined OOCDs/CECDU	18.12	4.03W

for the intersection test (Figure 10) for different environments. As shown in the figure, the proposed method effectively filters easy cases (cycle-1) across different environmental complexity.

Bakhshalipour et al. [3] also proposed a collision detection acceleration unit CODAcc for OBB-voxelized environment collision detection. In their approach, an OBB is converted to occupied voxels, and read requests for the environment's occupancy information corresponding to these voxels are sent to memory. We could not quantitatively compare our proposed OOCD unit with CODAcc as absolute performance is not reported in their work, and we could not get access to their implementation. Below we provide our insights and approximate comparison of both approaches. Voxelization of OBB results in a simpler intersection test; however, the number of voxels to be checked and memory accesses increase significantly with the resolution of voxelization. Our approximate measurement for the Jaco2 robot shows that for voxels of size 2.56cm (environment's extent is 180cm), the voxelized environment requires 32KB storage and $30 - 154$ memory accesses. In contrast, OOCD uses an octree-based compact environment representation and performs collision detection between OBB-environment in < 40 cycles with 0.75KB on-chip SRAM.

7.3 Area and power

Table 2 summarizes the area and power estimation obtained from the synthesis of RTL implementations of all hardware modules and submodules. The intersection unit is a major contributor to the total area and power. Further, the total area and power of the Intersection Unit are dominated by fixed-point multipliers ($\sim$ 85%), which can be reduced significantly by employing custom-designed multiplier cells [41]. The critical-path delay for pipelined/multi-cycle OOCD is 1.48ns/2.24ns, and is dominated by multipliers. This delay can be reduced by using optimized 16-bit multipliers' standard cells [41] and/or pipelining.

7.4 Motion planning runtime for MPNet

We further evaluate the runtime for motion planning queries for MPNet algorithm using MPAccel. We use an estimate of 12TOPS for the DNN accelerator, which can be achieved by existing DNN accelerators [10, 59]. Similarly, we set the

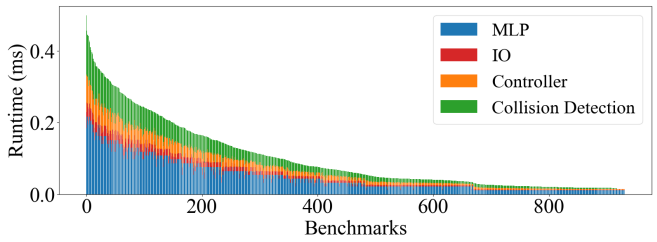


Figure 19: Motion planning runtime using MPAccel for different benchmarks. Baxter robot is used for the evaluation. Here the number of CECDUs is set to eight, and each CECDU has four multi-cycle Intersection Units.

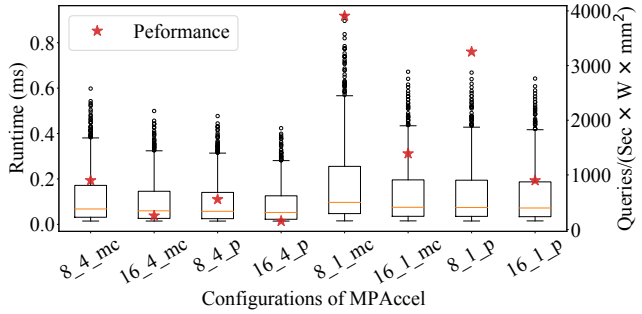


Figure 20: Motion planning runtime and performance for different MPAccel configurations. The performance is measured using number of motion planning queries executed per (Second×Watt×mm^2).

IO bandwidth to 5GBPS. We estimate the latency of the controller using the number of instructions.

Figure 19 provides the motion planning latency for different benchmarks for MCSP-based scheduler and 16 CECDUs with four multi-cycle OOCDs each. The motion planning time varies from 0.014ms to 0.49ms, with an average runtime of 0.099ms. This motion planning runtime meets the requirement of real-time motion planning ($< 1ms$ as the actuators' response rate is typically $\sim$ 1kHz).

We further compare the latency and performance of different configurations of the scheduler and CECDUs. Figure 20 compares the motion planning latency and performance for different configurations of MPAccel. Here X_Y_mc/p configuration represents MPAccel with X CECDUs, Y OOCDs per CECDU, and multi-cycle or pipelined design of the Intersection Unit. The left Y-axis compares the latency distribution for motion planning queries. The right Y-axis represents the performance of configurations (motion planning queries/(Second×Watt×mm^2)).

7.5 Evaluation on GPU and CPU

Collision detection is the most time-consuming kernel in motion planning [4, 33]. To help compare against GPU baselines, we thus wrote our own OBB-octree collision test for a GPU. Here each thread performs OBB-octree collision detection. We form a warp (32 threads) such that all OBBs in a single warp have physical locality, which reduces divergence as all threads are likely to follow a similar traversal order in the octree. We also implement two optimizations specific to GPU. Each thread accesses a FIFO queue during the octree traversal. We interleave queues for all threads in a single warp to reduce memory divergence. The second optimization is based on [30] to reduce warp divergence. We also implement an OBB-octree leaf nodes collision detection. In this approach, each thread performs collision detection between a leaf node and OBB. Table 3 summarizes the CPU and GPU runtime for 2^{20} OBB-octree collision detection queries. For comparison, 16 CECDUs with four multi-cycle OOCDs per CECDU (11.1mm^2, 3.4W) take 0.91ms to execute the same number of OBB-octree collision detection queries. Similarly, 16 CECDUs with four pipelined OOCDs per CECDU (18.0mm^2, 4.0W) take 0.53ms to execute the same number of OBB-octree collision detection queries. We built a simulator for the CPU+DNN Accelerator and GPU+Controller+DNN Accelerator system to evaluate motion planning runtime. Table 3 summarizes the overall motion planning runtime.

Table 3: Collision detection runtime for GPU and CPUs.

	NVIDIA Titan V	NVIDIA Jetson TX2 GPU	i7-4771 (8-core)	Cortex-A57 (4-core)
OBB-octree (ms)	24	5833	153	360
OBB-octree + GPU optimizations (ms)	12	3403	N/A	N/A
OBB-octree leaf nodes (ms)	6	1373	890	3304
Power (W)	156.8	3.5	65	4.2
Average motion planning runtime (ms)	1.42	110.27	4.13	11.62

8 RELATED WORK

Bakhshalipour et al. [3] proposed an acceleration approach for a path planning algorithm for robots with 2-3 DOFs. However, this approach does not apply to motion planning algorithms for robots with higher DOF. They also proposed a collision detection acceleration unit CODAcc based on voxelized OBB-voxelized environment intersection. Here, the number of intersection tests increases with voxelization resolution. We chose the separating axis test as it does not discretize the robot's space and is scalable for fine-resolution intersection tests. Jia et al. [22] proposed a mapping accelerator to map environment point cloud data to octree representation. The proposed accelerator also supports a collision detection test between a voxel and the environment. However, similarly, the number of voxel queries increases with the voxelization resolution used for the robot's occupied space.

Other works have proposed motion planning acceleration for CPUs and GPUs [1, 4, 8, 15, 23, 28, 37, 38, 40]. While GPU-based acceleration approaches provide significant acceleration, state-of-the-art motion planning approaches on GPUs still do not provide the energy efficiency and performance. Hardware acceleration approaches have been proposed for specific motion planning algorithms to meet the computation and real-time requirements, including on FPGA [2, 33, 48] and ASIC [3, 29, 32, 52, 58]. Murray et al. [33] proposed an acceleration approach using FPGAs for probabilistic roadmap-based motion planning (PRM). Further, they expanded the

work to a programmable motion planning chip [32, 52]. They proposed to use a fixed set of motions and precompute the space occupied by these motions. These swept spaces are represented using sets of voxels. At runtime, these precomputed swept spaces are used for collision detection. Lian et al. [29] proposed to use octree representation for swept spaces. Though the pre-computation step reduces the motion planning runtime, to solve challenging motion planning tasks, precomputed swept spaces require more than 40MB on-chip memory or $>$ 40GBPS off-chip memory bandwidth. Yang et al. [58] proposed to use near-memory computing to reduce the memory bandwidth requirement. However, these approaches use a fixed set of motions for motion planning. Such approaches are suitable for a constrained environment with fixed obstacles and tasks but are not scalable for autonomous robots performing diverse tasks in highly dynamic environments.

RoboRun [6] proposes to control the volume and precision of the environment depending upon its speed and the distance from obstacles. The proposed optimization can be applied to MPAccel, as the environment's octree representation supports variable precision using octree node pruning.

9 CONCLUSION

In this work, we analyze sampling-based motion planning algorithms and identify the sources of possible acceleration. We show that speedup through naive parallelization comes at the cost of increased energy due to redundant computation. We identify the sources of these redundant computations in coarse-grained and fine-grained parallelization. Based on this, we propose an algorithm-hardware-based approach to increase the energy efficiency of parallel execution. SAS, the proposed scheduler unit, results in 7× speedup using 8 collision detection cores compared to sequential execution, with 6% increase in energy. The proposed CECDU can perform collision detection in $46 - 154$ cycles for a robotic arm with 6 degrees of freedom. SAS and CECDU enables real-time motion planning for a 7-DOF robot in 0.014ms-0.49ms with an average of 0.099ms when evaluated for a learning-based motion planning algorithm.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers and artifact evaluators for their feedback. We would also like to thank Mabel Wang for helping with MPNet experiment setup. This research has been funded in part by the National Sciences and Engineering Research Council of Canada (NSERC) through the NSERC strategic network on Computing Hardware for Emerging Intelligent Sensory Applications (COHESA) and through an NSERC Strategic Project Grant. Tor M. Aamodt recently served as a consultant for Huawei Technologies Canada Co. Ltd and Intel Corp.

REFERENCES

[1] N.M. Amato and L.K. Dale. 1999. Probabilistic roadmap methods are embarrassingly parallel. In *Proceedings 1999 IEEE International Conference on Robotics and Automation*. 688–694. https://doi.org/10.1109/ROBOT.1999.770055

[2] Nuzhet Atay and Burchan Bayazit. 2006. A motion planning processor on reconfigurable hardware. In *Proceedings of IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 125–132. https://doi.org/10.1109/ROBOT.2006.1641172

[3] Mohammad Bakhshalipour, Seyed Borna Ehsani, Mohamad Qadri, Dominic Guri, Maxim Likhachev, and Phillip B. Gibbons. 2022. RACOD: Algorithm/Hardware Co-Design for Mobile Robot Path Planning. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. Association for Computing Machinery. https://doi.org/10.1145/3470496.3527383

[4] Joshua Bialkowski, Sertac Karaman, and Emilio Frazzoli. 2011. Massively parallelizing the RRT and the RRT. In *International Conference on Intelligent Robots and Systems*. 3513–3518. https://doi.org/10.1109/IROS.2011.6048813

[5] Guy E. Blelloch and Bruce M. Maggs. 1996. Parallel Algorithms. *ACM Comput. Surv.* 28, 1 (1996), 51–54. https://doi.org/10.1145/234313.234339

[6] Behzad Boroujerdian, Radhika Ghosal, Jonathan Cruz, Brian Plancher, and Vijay Janapa Reddi. 2021. RoboRun: A Robot Runtime to Exploit Spatial Heterogeneity. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. 829–834. https://doi.org/10.1109/DAC18074.2021.9586280

[7] D. Brooks, V. Tiwari, and M. Martonosi. 2000. Wattch: a framework for architectural-level power analysis and optimizations. In *Proceedings of 27th International Symposium on Computer Architecture*. 83–94.

[8] D.J. Challou, M. Gini, and V. Kumar. 1993. Parallel search algorithms for robot motion planning. In *Proceedings IEEE International Conference on Robotics and Automation*. 46–51 vol.2. https://doi.org/10.1109/ROBOT.1993.292122

[9] Jung-Woo Chang, Wenping Wang, and Myung-Soo Kim. 2010. Efficient collision detection using a dual OBB-sphere bounding volume hierarchy. *Computer-Aided Design* 42 (2010), 50–57. https://doi.org/10.1016/j.cad.2009.04.010

[10] Coral AI. 2020. AI Google Coral TPU Overview and Products. https://dls.ieiworld.com/IEIWEB/MARKETING_MATERIAL/2021_catalog/0-10_AI_Google_TPU_overview_%26_products.pdf

[11] Florent de Dinechin, Matei Istoan, and Guillaume Sergent. 2014. Fixed-Point Trigonometric Functions on FPGAs. *SIGARCH Comput. Archit. News* 41, 5 (2014), 83–88. https://doi.org/10.1145/2641361.2641375

[12] J. Denavit and R. S. Hartenberg. 2021. A Kinematic Notation for Lower-Pair Mechanisms Based on Matrices. *Journal of Applied Mechanics* 22, 2 (2021), 215–221. https://doi.org/10.1115/1.4011045 arXiv:https://asmedigitalcollection.asme.org/appliedmechanics/article-pdf/22/2/215/6748803/215_1.pdf

[13] Christer Ericson. 2004. *Real-Time Collision Detection*. CRC Press, Inc.

[14] D. Ferguson, N. Kalra, and A. Stentz. 2006. Replanning with RRTs. In *Proceedings 2006 IEEE International Conference on Robotics and Automation*. 1243–1248. https://doi.org/10.1109/ROBOT.2006.1641879

[15] Russell Gayle, Paul Segars, Ming Lin, and Dinesh Manocha. 2005. Path Planning for Deformable Robots in Complex Environments. In *Robotics: Science and Systems*. 225–232. https://doi.org/10.15607/RSS.2005.I.030

[16] Roland Geraerts and Mark Overmars. 2007. Creating High-quality Paths for Motion Planning. *International Journal of Robotics Research* 26 (08 2007). https://doi.org/10.1177/0278364907079280

[17] S. Gottschalk, Ming Lin, and Dinesh Manocha. 1997. OBBTree: A Hierarchical Structure for Rapid Interference Detection. *Computer Graphics* 30 (10 1997). https://doi.org/10.1145/237170.237244

[18] Matthew R. Guthaus, James E. Stine, Samira Ataei, Brian Chen, Bin Wu, and Mehedi Sarwar. 2016. OpenRAM: An open-source memory compiler. In *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–6. https://doi.org/10.1145/2966986.2980098 ISSN: 1558-2434.

[19] Kris Hauser and Victor Ng-Thow-Hing. 2010. Fast smoothing of manipulator trajectories using optimal bounded-acceleration shortcuts. In *2010 IEEE International Conference on Robotics and Automation*. 2493–2498. https://doi.org/10.1109/ROBOT.2010.5509683

[20] Armin Hornung, Kai Wurm, Maren Bennewitz, Cyrill Stachniss, and Wolfram Burgard. 2013. OctoMap: An efficient probabilistic

3D mapping framework based on octrees. *Autonomous Robots* 34 (04 2013). https://doi.org/10.1007/s10514-012-9321-0

[21] Brian Ichter, James Harrison, and Marco Pavone. 2018. Learning Sampling Distributions for Robot Motion Planning. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*. 7087–7094. https://doi.org/10.1109/ICRA.2018.8460730

[22] Tianyu Jia, En-Yu Yang, Yu-Shun Hsiao, Jonathan Cruz, David Brooks, Gu-Yeon Wei, and Vijay Janapa Reddi. 2022. OMU: A Probabilistic 3D Occupancy Mapping Accelerator for Real-Time OctoMap at the Edge. In *Proceedings of the 2022 Conference Exhibition on Design Automation Test in Europe* (Antwerp, Belgium). 909–914.

[23] Joseph T. Kider, Mark Henderson, Maxim Likhachev, and Alla Safonova. 2010. High-dimensional planning on the GPU. In *2010 IEEE International Conference on Robotics and Automation*. 2515–2522. https://doi.org/10.1109/ROBOT.2010.5509470

[24] KINOVA. 2018. KINOVA JACO Assistive robot. https://www.kinovarobotics.com/sites/default/files/KINO-2018-Bro-Assistive-ZH_YUL-06-R-Web.pdf.

[25] J.T. Klosowski, M. Held, J.S.B. Mitchell, H. Sowizral, and K. Zikan. 1998. Efficient collision detection using bounding volume hierarchies of k-DOPs. *IEEE Transactions on Visualization and Computer Graphics* (1998). https://doi.org/10.1109/2945.675649

[26] Steven M. LaValle. 2006. *Planning Algorithms*. http://lavalle.pl/planning/

[27] Lisa Lee, Emilio Parisotto, Devendra Singh Chaplot, Eric Xing, and Ruslan Salakhutdinov. 2018. Gated Path Planning Networks. In *Proceedings of the 35th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 80)*. 2947–2955.

[28] Jed Lengyel, Mark Reichert, Bruce R. Donald, and Donald P. Greenberg. 1990. Real-Time Robot Motion Planning Using Rasterizing Computer Graphics Hardware. *SIGGRAPH Comput. Graph.* 24, 4 (1990), 327–335. https://doi.org/10.1145/97880.97915

[29] Shiqi Lian, Yinhe Han, Xiaoming Chen, Ying Wang, and Hang Xiao. 2018. Dadu-P: A Scalable Accelerator for Robot Motion Planning in a Dynamic Environment. In *Proceedings of the Annual Design Automation Conference* (San Francisco, California) *(DAC)*. Association for Computing Machinery, 6 pages.

[30] Jianqiao Liu, Nikhil Hegde, and Milind Kulkarni. 2016. Hybrid CPU-GPU Scheduling and Execution of Tree Traversals. In *Proceedings of the 2016 International Conference on Supercomputing*. Association for Computing Machinery, Article 2, 12 pages. https://doi.org/10.1145/2925426.2926261

[31] Lynxmotion. 2020. AL5D Robot Arm Specs. http://www.lynxmotion.com/driver.aspx?Topic=specs04.

[32] Sean Murray, Will Floyd-jones, George Konidaris, and Daniel J Sorin. 2019. A Programmable Architecture for Robot Motion Planning Acceleration. In *International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 185–188.

[33] Sean Murray, William Floyd-Jones, Ying Qi, George Konidaris, and Daniel J. Sorin. 2016. The Microarchitecture of a Real-Time Robot Motion Planning Accelerator. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*. IEEE Press. https://doi.org/10.1109/MICRO.2016.7783748

[34] Sean Murray, Will Floyd-Jones, Ying Qi, Daniel J. Sorin, and George Konidaris. 2016. Robot Motion Planning on a Chip. In *Robotics: Science and Systems*. https://doi.org/10.15607/rss.2016.xii.004

[35] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. 2018. Understanding PCIe Performance for End Host Networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18)*. Association for Computing Machinery, 327–341. https://doi.org/10.1145/3230543.3230560

[36] Theodore Pachidis, Christos Sgouros, Vassilis G. Kaburlasos, Eleni Vrochidou, Theofanis Kalampokas, Konstantinos Tziridis, Alexandros Nikolaou, and George A. Papakostas. 2020. Forward Kinematic Analysis of JACO2 Robotic Arm Towards Implementing a Grapes Harvesting Robot. In *2020 International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*. https://doi.org/10.23919/SoftCOM50211.2020.9238297

[37] Jia Pan, Christian Lauterbach, and Dinesh Manocha. 2010. g-Planner: Real-time Motion Planning and Global Navigation using GPUs.. In *Proceedings of the National Conference on Artificial Intelligence*, Vol. 2.

[38] Jia Pan and Dinesh Manocha. 2012. GPU-based parallel collision detection for fast motion planning. *The International Journal of Robotics Research* 31, 2 (2012), 187–200. https://doi.org/10.1177/0278364911429335

[39] Keshav Pingali, Milind Kulkarni, Donald Nguyen, Martin Burtscher, Mario Mendez-Lojo, Dimitrios Prountzos, Xin Sui, and Zifei Zhong. 2009. Amorphous data-parallelism in irregular algorithms. *regular tech report TR-09-05, The University of Texas at Austin* (2009).

[40] E. Plaku and L.E. Kavraki. 2005. Distributed Sampling-Based Roadmap of Trees for Large-Scale Motion Planning. In *Proceedings of the 2005 IEEE International Conference on Robotics and Automation*. 3868–3873. https://doi.org/10.1109/ROBOT.2005.1570711

[41] Liangyu Qian, Chenghua Wang, Weiqiang Liu, Fabrizio Lombardi, and Jie Han. 2016. Design and evaluation of an approximate Wallace-Booth multiplier. In *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*. 1974–1977. https://doi.org/10.1109/ISCAS.2016.7538962

[42] Ahmed Qureshi, Yinglong Miao, Anthony Simeonov, and Michael Yip. 2020. Motion Planning Networks: Bridging the Gap Between Learning-Based and Classical Motion Planners. *IEEE Transactions on Robotics* PP (08 2020), 1–19. https://doi.org/10.1109/TRO.2020.3006716

[43] Ahmed H Qureshi, Anthony Simeonov, Mayur J Bency, and Michael C Yip. 2019. Motion planning networks. In *2019 International Conference on Robotics and Automation (ICRA)*. IEEE, 2118–2124.

[44] Ahmed H Qureshi and Michael C Yip. 2018. Deeply Informed Neural Sampling for Robot Motion Planning. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 6582–6588.

[45] Rethink Robotics. 2013. Baxter. https://robots.ieee.org/robots/baxter/

[46] Oren Salzman. 2019. Sampling-Based Robot Motion Planning. *Commun. ACM* (2019), 54–63. https://doi.org/10.1145/3318164

[47] Ji Hwan Seo, Hyuntae Lee, and Kyoung-Dae Kim. 2021. A Parallelization Algorithm for Real-Time Path Shortening of High-DOFs Manipulator. *IEEE Access* 9 (2021), 123727–123741. https://doi.org/10.1109/ACCESS.2021.3109744

[48] Xuesong Shi, Lu Cao, Dawei Wang, Ling Liu, Ganmei You, Shuang Liu, and Chunjie Wang. 2018. HERO: Accelerating Autonomous Robotic Tasks with FPGA. *International Conference on Intelligent Robots and Systems*, 7766–7772. https://doi.org/10.1109/IROS.2018.8593522

[49] Julian Shun. 2017. *Shared-Memory Parallelism Can Be Simple, Fast, and Scalable*. Vol. 15. Association for Computing Machinery and Morgan.

[50] Daniel Sorin, William Floyd-Jones, Sean Murray, George Konidaris, and William Walker. 2019. Apparatus, method and article to facilitate motion planning of an autonomous vehicle in an environment having dynamic objects. Patent No. US11292456B2, Filed April 1, 2019, Issued April 5, 2022.

[51] Daniel J. Sorin and George Konidaris. 2018. Enabling Faster, More Capable Robots With Real-Time Motion Planning. https://spectrum.ieee.org/enabling-faster-more-capable-robots-with-real-time-motion-planning

[52] Daniel J. Sorin, George Konidaris, William Floyd-Jones, and Sean Murray. 2019. Motion Planning for Aotonomous Vehicles and Reconfigurable Motion Planning Processor. Patent No. US20190163191A1, Filed June 9, 2017, Issued May 30, 2019.

[53] James E. Stine, Ivan D. Castellanos, Michael H. Wood, Jeff Henson, Fred Love, William Rhett Davis, Paul D. Franzon, Michael Bucher, Sunil Basavarajaiah, Julie Oh, and Ravi Jenkal. 2007. FreePDK: An Open-Source Variation-Aware Design Kit. In *IEEE International Conference on Microelectronic Systems Education (MSE)*. 173–174.

[54] Neil Tardella. 2019. Robots with high degrees of freedom face barriers to adoption. https://www.cobottrends.com/robots-with-high-degrees-of-freedom-face-barriers-to-adoption/.

[55] Maxim Tatarchenko, Alexey Dosovitskiy, and Thomas Brox. 2017. Octree Generating Networks: Efficient Convolutional Architectures for High-resolution 3D Outputs. In *2017 IEEE International Conference on Computer Vision (ICCV)*. 2107–2115. https://doi.org/10.1109/ICCV.2017.230

[56] Jonathan Tilley. 2017. Automation, robotics, and the factory of the future. https://www.mckinsey.com/capabilities/operations/

our-insights/automation-robotics-and-the-factory-of-the-future.
[57] Jiankun Wang, Tianyi Zhang, Nachuan Ma, Zhaoting Li, Han Ma, Fei Meng, and Max Q.-H. Meng. 2021. A survey of learning-based robot motion planning. *IET Cyber-Systems and Robotics* (2021). https://ietresearch.onlinelibrary.wiley.com/doi/abs/10.1049/csy2.12020
[58] Yuxin Yang, Xiaoming Chen, and Yinhe Han. 2020. Dadu-CD: Fast and Efficient Processing-in-Memory Accelerator for Collision Detection. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. 1–6.
[59] Amir Yazdanbakhsh, Berkin Akin, and Kiran K Seshadri. 2021. An Evaluation of Edge TPU Accelerators for Convolutional Neural Networks. https://arxiv.org/abs/2102.10423.
[60] Eiichi Yoshida, Kazuhito Yokoi, and Pierre Gergondet. 2010. Online replanning for reactive robot motion: Practical aspects. In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 5927–5933. https://doi.org/10.1109/IROS.2010.5649645

A ARTIFACT APPENDIX

This artifact provides the complete implementation for the MPAccel simulator and the traces used for evaluation reported in Section 7. Section A.1 provides basic information about the artifact. We describe the artifact description and installation procedure in Section A.2. Section A.3 describes the experiment workflow for evaluation using the provided trace files. We also provide the trace generation script example for the Jaco2 robot, which can be used to generate traces for different robots and environmental scenarios (Section A.4).

A.1 Artifact Check-List (Meta-Information)

- **Program:** We provide microarchitecture simulators for CECDU, SAS, and MPAccel.
- **Run-time environment:** All scripts are tested on Linux (Ubuntu 16.04) and macOS (12). It does not require root access.
- Hardware: All evaluation experiments require only CPU (no specific requirement). Trace generation example (traces provided; trace generation not needed for main evaluation) requires GPU for neural network inference.
- **Metrics:** Execution time and energy consumption.
- **Output:** Numerical results and graphs reported in Section 7.
- **Experiments:** README provided with instructions. Provided bash scripts to run all experiments.
- **How much disk space required (approximately)?:** 5GB.
- **How much time is needed to prepare workflow (approximately)?:** Less than 1 hour.
- **How much time is needed to complete experiments (approximately)?:** 10-12 hours for main evaluation using a single machine.
- **Publicly available?:** Yes.
- **Licenses (if publicly available)?:** The simulator code is available under Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.
- **Archived (provide DOI)?:** 10.5281/zenodo.7824123

A.2 Description and Installation

A.2.1 How to access it: MPAccel simulator is available on Zenodo: https://doi.org/10.5281/zenodo.7824123. This repository consists of the microarchitectural simulators and traces used for evaluation.

A.2.2 Hardware dependencies: There are no specific hardware requirements if only executing the MPAccel simulator with provided traces. For MPNet trace generation using provided example, a CPU-GPU system is preferable as it performs neural network inferences.

A.2.3 Software dependencies: Our artifact has been tested on Ubuntu 16.04 and macOS 12. It does not require root access. The execution requires Python 3.7.0 additional Python packages. Conda can be optionally installed to create a virtual environment.

A.2.4 Installation: Create a virtual environment and activate with python==3.7.0 (**optional to use virtual environment).

```
cd MPAccel_simulator
conda create -n mpaccel python==3.7.0
conda activate mpaccel
python -m pip install -r requirements.txt
```

A.3 Evaluation

A.3.1 Cascaded Early-entry Collision Detection Unit (CECDU):. The experiments required for Table 1 and Figure 17 can be executed using the following set of scripts. All scripts should take less than 20 minutes.

```
cd MPAccel_simulator/collision_detection
## Run all experiments for Table 1
bash table_1.sh
## Run all experiments for Figure 17
bash fig17.sh
```

A.3.2 Spatially Aware Scheduler (SAS):. The experiments required for Figure 15 and results reported in Section 7.1 can be run using the following commands for a subset of benchmarks.

```
cd MPAccel_simulator/SAS
bash launch_overall.sh
bash combine.sh
python plot_figure15.py
```

SAS evaluation for all benchmarks (∼60 hour) can be carried out by executing the above commands after making the following changes in launch_overall.sh and combine.sh. Comment lines 5 and 49, and uncomment lines 6 and 50 in launch_overall.sh. Comment lines 6, 12, 18, 24, and uncomment lines 7, 13, 19, 25 in combine.sh. Rerun the above commands to run the SAS simulator for all benchmarks.

A.3.3 MPAccel Evaluation for Motion Planning: These experiments require approximately 2 hours. The following steps will generate the motion planning runtime in Figure 19 and Section 7.4.

```
## Generate collision detection runtime for SAS + 16
    CECDUs
## Each CECDU consists of four multicycle OOCDs
cd MPAccel_simulator/SAS
bash launch_16_cdu.sh
## Above command stores traces in ../traces/
    mpnet_logfile/bench*/16cdu_result_16_16_4_mcsp.txt
cd ../mpaccel_overall
ls ../traces/mpnet_logfile/bench_*/16cdu* >
    result_filenames
python mpaccel_sim.py
```

A.4 Trace Generation

1) OBB information for CECDU:

```
cd MPAccel_simulator/collision_detection
bash E1_run.sh
```

2) Scene information for CECDU:

```
cd MPAccel_simulator/collision_detection
bash E2_run.sh
```

3) Collision latency generation for SAS:

```
cd MPAccel_simulator/collision_detection
bash E3_run.sh
```

4) MPNet trace generation example: We provide the trace generation example script for the MPNet motion planning algorithm. This script runs the motion planning algorithm and stores the traces of motions and motion segments in different phases of the algorithm, which is then used for motion planning runtime analysis of MPAccel.

```
cd MPAccel_simulator/mpnet_tracegen_example
# The following script downloads the trained models for
    MPNet from
# https://drive.google.com/file/d/1
    fh6JMzgruaDNE8J4PhuX29L0-sP57xQn/view?usp=share_link
# and
# https://drive.google.com/file/d/1
    GwDjnxlu9tkyxcX7eJg0Xt9N0_zr4gEU/view?usp=share_link
bash download.sh
# This script generates the traces for MPNet in bench_0_
    * folder
bash E4_run.sh
```