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@ecee.ubc.ca, nxyang@ecee.ubc.ca, aamodt@ecee.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, Casscaded 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.

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:

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
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 $15 \times$ 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 $50 \times$ speedup with $3.4 \times$ 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 $7 \times$ 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
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 pi and pi+1 are connected by a short motion. The motions between adjacent poses in {p1, p2, ..., pN} 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.

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...
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.
Energy-Efficient Realtime Motion Planning

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 0, 4, 8, ..., 1, 5, 9, ..., 2, 6, 10, ..., 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 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 EARL-y-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 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 ~5× 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., 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
Energy-Efficient Realtime Motion Planning

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.

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 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 1. 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 2. The controller receives these intermediate poses and generates a set of motions for collision detection based on the motion planning algorithm 3. 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 4. The scheduler also collects results from the CECDUs 5 and sends back the aggregated result to the controller once the execution finishes 6.

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
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, 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. Collision detection (CD) Query Generator generates a discrete pose to be checked for collision and sends it to free CECDUs. 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. 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. Depending upon the function mode, the result processing unit signals other units to stop operation and sends the result to the controller.

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 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 and OBB-octree Collision Detector (OOCD).
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 $3 \times 3$ 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 and stores the root node’s address (i.e., 0) to the Address Register and sets its valid bit 6. The SRAM stores the environment octree. The Memory Request Generator sends a memory request when the Address Register has a valid entry. The received data is then added to the Node Queue. The Node Queue can store 8 entries with 24 bits per entry. The Node Processing Unit receives this node information 6. 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 7. 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 ($6 \times 16$ 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) 8. 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
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 $7 \times$ speedup with $6\%$ increase in the energy compared to $3.7 \times$ speedup with $83\%$ increase in energy for NP. Similarly, for 16 CDUs, MCSP results in $11.03 \times$ speedup with $22\%$ increase in the energy compared to $6.2 \times$ 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 $2.52 \times$ and $1.77 \times$ 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 $\sim 4.1 \times$ 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.

<table>
<thead>
<tr>
<th></th>
<th>Single Intersection Unit</th>
<th>Four Intersection Units</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Multi-cycle</td>
<td>Pipelined</td>
</tr>
<tr>
<td>Latency (Cycles)</td>
<td>154.4</td>
<td>137.5</td>
</tr>
<tr>
<td>Area (mm²)</td>
<td>0.21</td>
<td>0.32</td>
</tr>
<tr>
<td>Power (mW)</td>
<td>92.6</td>
<td>100.8</td>
</tr>
</tbody>
</table>

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).

Table 2: Area and power breakdown for hardware units.

<table>
<thead>
<tr>
<th>Module</th>
<th>Area (mm²)</th>
<th>Power (mW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scheduler</td>
<td>0.110</td>
<td>60.7</td>
</tr>
<tr>
<td>CECDU (with four multi-cycle OOCD)</td>
<td>0.694</td>
<td>215.7</td>
</tr>
<tr>
<td>OBB Transformation Unit</td>
<td>0.054</td>
<td>51.6</td>
</tr>
<tr>
<td>Octree Traversal Unit</td>
<td>0.029</td>
<td>16.7</td>
</tr>
<tr>
<td>Intersection Unit (Multi-cycle)</td>
<td>0.143</td>
<td>24.34</td>
</tr>
<tr>
<td>Intersection Unit (Pipelined)</td>
<td>0.251</td>
<td>32.57</td>
</tr>
<tr>
<td>MPAccel (Scheduler + 16 CECDUs)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Config 1: 4 multi-cycle OOCDs/CECDU</td>
<td>11.21</td>
<td>3.51W</td>
</tr>
<tr>
<td>Config 2: 4 pipelined OOCDs/CECDU</td>
<td>18.12</td>
<td>4.03W</td>
</tr>
</tbody>
</table>

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 (~ 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
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 thread. We perform collision detection between a leaf node and OBB. Table 3 summarizes the CPU and GPU runtime for 2\textsuperscript{nd} OBB-octree collision detection queries. For comparison, 16 CECDUs with four multi-cycle OOCDs per CECDU (11.1 mm\textsuperscript{2}, 3.4 W) take 0.91 ms to execute the same number of OBB-octree collision detection queries. Similarly, 16 CECDUs with four pipelined OOCDs per CECDU (18.0 mm\textsuperscript{2}, 4.0 W) take 0.53 ms 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.

**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 compare 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. [58] 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 7x speedup using 8 collision detection cores compared to sequential execution, with 6% increase in energy. The proposed CECCU can perform collision detection in 46 – 154 cycles for a robotic arm with 6 degrees of freedom. SAS and CECCU 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


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 mpaaccel python==3.7.0
- conda activate mpaaccel
- 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

## Generate collision detection runtime for SAS + 16 CECDUs
## Each CECDU consists of four multicycle OOCDS
## Above command stores traces in ../traces/mpnet_logfile/bench*/16cdu_result_16_16_4_mcsp.txt
## Above command stores traces in ../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/1fh6JHsgryuAHE8J4PheXZ9tLo-sP57aQn/view?usp=share_link
   # and
   # https://drive.google.com/file/d/1GwDy2i9tkyvX7eJyOx9tH9O_zr4gEU/view?usp=share_link
   bash download.sh
   bash E4_run.sh