Appears in the 49th Annual IEEE/ACM International Symposium on Microarchitecture, 2016

The Microarchitecture of a Real-Time Robot Motion Planning Accelerator

Sean Murray, William Floyd-Jones, Ying Qi, George Konidaris, and Daniel J. Sorin
Departments of Computer Science and Electrical & Computer Engineering
Duke University

***Abstract*—We have developed a hardware accelerator for motion planning, a critical operation in robotics. In this paper, we present the microarchitecture of our accelerator and describe a prototype implementation on an FPGA. We experimentally show that the accelerator improves performance by three orders of magnitude and improves power consumption by more than one order of magnitude. These gains are achieved through careful hardware/software co-design. We modify conventional motion planning algorithms to aggressively precompute collision data, as well as implement a microarchitecture that leverages the parallelism present in the problem.**

I. Introduction

For important applications, it is well known that specialized hardware accelerators can provide better performance and power-efficiency than running software on general purpose processors. The use of specialized hardware is even more attractive now that power is a primary constraint in chip design. Recent work has developed accelerators for applications such as web search [1], neuromorphic computing [2], radix sort [3], and molecular dynamics [4]. The specialized hardware can be a stand-alone processor (e.g., like a GPU), a co-processor, or even a special functional unit.

In this paper, we present and evaluate the microarchitecture of a specialized processor for accelerating an application that is critical to robotics: motion planning. Motion planning, described in more detail in Section II, is the process of computing a path that a robot can take to reach a goal without colliding with any obstacles in its environment. This path is often in a high-dimensional space, as it must specify trajectories for all degrees of freedom present on the robot. Motion planning is vitally important for robots, yet current implementations are too slow to be used in real-time applications. Moreover, while speed is obviously critical for real-time motion planning, power is also a major concern for untethered robots or environments with many robots.

At first glance, real-time motion planning may seem like a simple problem. Many people are familiar with robots that perform real-time motion planning in 2D (e.g., warehouse robots moving pallets around) and even 3D (airplane autopilots), but these low-dimension problems are vastly simpler than the motion planning required for high degree-of-freedom robotic arms. Many interesting robots have upwards of 6-10 degrees of freedom in a single arm, and finding paths in these high-dimensional spaces is far more difficult. The current state of the art uses high-performance GPUs with highly tuned software to achieve motion plans on the order of hundreds of milliseconds [5][6] at a power cost of hundreds of watts. This performance is insufficient for real-time planning, and the power consumption is both infeasible for untethered robots and overly expensive for applications with many robots (e.g., facilities with tens of thousands of robots).

The primary challenge in motion planning is collision detection. Given a description of the robot's environment (e.g., obstacles near the robot), collision detection is the process of determining whether a robot motion collides with obstacles. Because a robotic arm moves in a high-dimensional space, collision detection involves a vast amount of computation when done using traditional approaches from computational geometry. Although GPUs can accelerate these calculations, they still cannot provide the needed speed or power-efficiency. Currently, the collision detection bottleneck often comprises 99% of the time required for motion planning [7].

We introduced the concept of a motion planning processor and showed its effectiveness for planning in prior work [8]. Our contribution in this paper is the microarchitecture of the processor and a description of how it performs real-time, low-power collision detection. Section II gives a brief background of motion planning and an overview of a probabilistic algorithm that is widely used to create plans. In Section III we go through the process of designing an accelerator for the problem, evaluating several distinct strategies. Although we initially considered a design that straightforwardly accelerates current algorithms from computational geometry, we found that we could achieve far better results by designing the accelerator from scratch and co-designing the algorithm to be more amenable to hardware acceleration. As described in Section IV, we have implemented our design on an FPGA and interfaced the processor to a Jaco2 robotic arm in our laboratory (seen in Figure 5). We demonstrate real-time motion planning in which the robotic arm performs pick-and-place tasks using motion plans computed by our processor. Section V gives performance and power results, as well as a comparison to other solutions. Results on the robot in our lab show that we can perform collision detection in less than 20 microseconds drawing less than 15 watts. At this speed, which is three orders of magnitude faster than the state of the art, motion planning is truly real-time. At this power, which is more than an order of magnitude less than the state of the art, a robot could

978-1-5090-3508-3/16/$31.00

be untethered for longer periods of time, and energy savings would be significant when working with a large number of robots.

II. Background

A. Motion Planning

Motion planning is a fundamental problem in robotics; it is how the controller of a robot finds a safe (collision-free) path from its current position to a goal position. An analogy can be made to how a human decides how to best reach under a desk to unplug something. This is a challenging problem, because all joints must be coordinated, and care must be taken that no extremities (arms, legs, head) collide with the desk (or with other extremities). Planning is done in "configuration space," which has as many dimensions as the robot has degrees of freedom [9]. The most popular family of motion planning algorithms constructs a graph-based discretization of configuration space. The typical graph theory abstractions are used to describe navigation in this space. A node in configuration space completely defines a specific pose of the robot, and an edge in configuration space represents a movement between two poses. A graph of robot poses and movements is termed a "roadmap." Motion planning in this paradigm thus involves constructing and finding a path through a roadmap that does not collide with any obstacles. Even in the absence of obstacles, care must be taken so that a path does not result in the robot colliding with itself. An example roadmap is shown in Figure 1.

The problem becomes quite difficult when working with robots with many degrees of freedom (DOF), as it suffers from the same state space explosion problem present in many other fields. Indeed, we have reached a point where robots are capable of extremely complex, precise, and dexterous movements, but we lack the algorithms to efficiently utilize them in real-time applications. This disconnect between the mechanical capabilities of robots and our ability to use them is a major barrier to expanding the influence of robotics to new spaces such as manufacturing in unstructured environments, disaster response, or personal assistance (or any application requiring humans and robots to be co-located). The only deployed systems doing real-time planning are low-DOF machines; currently, virtually all industrial high-DOF robots work in tightly controlled environments that depend on objects being in exactly the same spot every time, eliminating the need to plan motions at runtime.

B. Probabilistic Roadmaps

The problem of creating plans for robots with many degrees of freedom has been extensively studied. Most modern solutions rely on probabilistic techniques to make the problem more tractable. In their 1996 work, Overmars and Kavraki [10] detail a process for constructing probabilistic roadmaps (PRMs) from which queries could produce motion plans. The use of random sampling enables this method to find solutions in high-dimensional spaces with far fewer samples than would have been required with uniform coverage. This paper is a

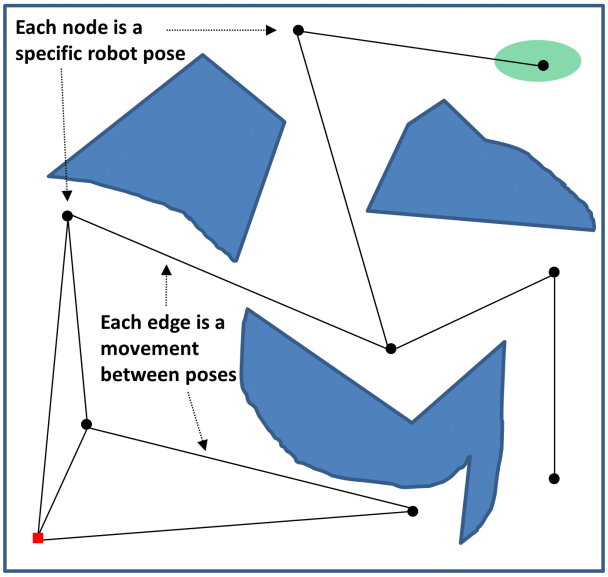


Fig. 1: Roadmap showing how a path could be found from a starting configuration (red square node) to the goal region (green oval) by sampling in configuration space and avoiding obstacles (amorphous blue regions). This example illustrates planning in 2D, while planning for most robotic arms takes place in a higher dimensional space.

seminal work in motion planning; many developments in the field over the past two decades draw on their basic strategy. We use this paradigm as a baseline for our study.

The PRM workflow has two phases. The computationally expensive *learning* phase involves the creation of a roadmap consisting of several possibly unconnected graph components. The fast and inexpensive *query* phase is where a path is (hopefully) found through the map to a specified goal configuration. As long as the environment is unchanged, several of these lightweight query calls can be made on the same roadmap. One of the reasons PRM and similar algorithms have been adopted so widely is that they provide asymptotic completeness. As the number of random samples drawn in the learning phase approaches infinity, the likelihood of not finding a safe path, if one exists, approaches zero.

1) Learning Phase: The learning phase follows an iterative process. In each iteration, a random configuration $\mathbf{C_{new}}$ is chosen by sampling values for all the independent degrees of freedom in the robot's configuration space. The first test done is to check whether the obtained configuration is itself collision-free; if so, the node is added to the graph, otherwise it is discarded and the next iteration begins. A list of potential neighbor nodes is assembled by choosing some distance function $\mathbf{D(a,\ b)}$ and associated threshold $\mathbf{T}$; all nodes $\mathbf{n}$ with $\mathbf{D}(\mathbf{n}, \mathbf{C_{new}}) < \mathbf{T}$ are added to the list. Working from the closest node in the list to the furthest, each node $\mathbf{n}$ is tested with a "local planner" and collision-checker to see if the path from $\mathbf{C_{new}}$ to $\mathbf{n}$ is collision-free. The properties of the local planner are simply that it must deterministically define a motion between two nodes, because only this path

will be verified to be collision-free. Depending on the desired connectivity of the graph, one can add edges from $\mathbf{C_{new}}$ to a variable number of nodes from the potential neighbor list that are determined to have collision-free connections. Ending conditions for the construction step can also be tailored to fit specific application needs and could be a desired number of configurations within a goal region or something as simple as a maximum number of total iterations.

The collision detection involved in the learning phase is the most computationally expensive part of the PRM process. Each edge in the roadmap represents a movement between two configurations. This movement creates a "swept volume" in 3D space that must be checked to ensure it does not collide with any obstacles. The swept volume (i.e., the region of space with which a motion intersects) is typically approximated as a mesh of polygons; triangles are normally used to take advantage of their (relatively) simple properties. The obstacles in the environment can also be represented as a mesh of triangles. Collision detection is then simply checking for intersections between the triangles in the swept volume and those in the obstacle representations. Each representation consists of hundreds of triangles, so many thousands or even millions of triangle intersection tests may be necessary. Collision detection has been found to consume up to 99% of the compute time in motion planning [7].

2) Query Phase: The query phase is much simpler. It simply involves finding paths between given start and end configurations in the graph. Any graph search or shortest path algorithm suffices. The same local planner is used to create the paths between configurations as was used during the learning phase. It can be run much faster this time, however, because collision checking is not required during the query phase; paths generated by the local planner are already guaranteed to be collision-free since they were verified in the learning phase.

C. Remaining Challenges

It may seem like PRM is quite sufficient as a solution. Computation can be done up front in a slow learning phase, and then lightweight calls can be made at runtime in the query phase. However, the roadmap is guaranteed to be safe only as long as the environment remains unchanged. Any change requires the connectivity (safeness) of the roadmap to be recomputed. There are variants of the PRM algorithm that try to minimize this cost, but recomputation is still quite expensive. Since collision detection consumes 99% of the time of building the roadmap, re-verifying safeness requires almost as much computation as building the roadmap from scratch. This limitation is acceptable in tightly controlled car assembly lines, but is not reasonable for robots that must quickly plan in dynamic environments. This challenge is what our work hopes to tackle; we aim to design a solution that will enable planning to occur in real time for dynamic environments.

III. Designing an Accelerator

The first step in designing an accelerator is to determine its architecture. In other words, the accelerator needs a definition of what task will be done, and the designer must consider what interface will be presented to the rest of the system. We present a sequence of designs in Sections III.A and III.B as we first explore ways to accelerate the PRM algorithm directly, but then transition to a modified algorithm more amenable to acceleration. Section III.C will discuss general architectural issues in accelerator design.

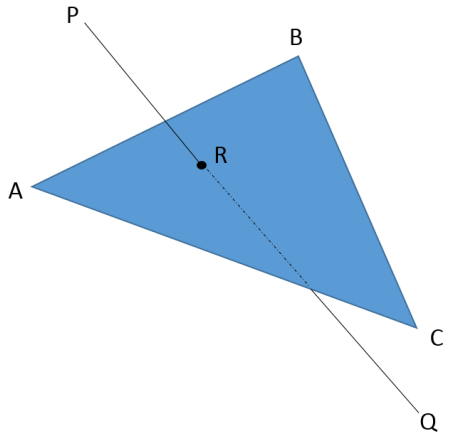


Fig. 2: A line segment vs triangle test involves determining whether the point **R** at which line **PQ** intersects the plane defined by **ABC** lies within the triangle **ABC**.

A. Direct Acceleration of Existing Algorithm

Our first strategy was to design a triangle-triangle intersection test accelerator. This was the logical place to start; collision detection had been proven to be the bottleneck in motion planning, collision detection involves performing possibly millions of triangle-triangle intersection tests, and there is a huge amount of parallelism in these tests to exploit [11].

Although there are several clever ways to reduce the amount of computation involved, the simplest method of determining if two triangles **ABC** and **DEF** intersect in 3D space is to test if any of the line segments (**AB**, **AC**, or **BC**) intersect with the triangle **DEF** and the same for the segments of **DEF** against triangle **ABC**. Thus, a single triangle-triangle intersection test can be decomposed into the logical disjunction of six line segment-triangle intersection tests (which can conveniently be performed in parallel) [12].

A line segment-triangle test is performed by taking the line segment and checking at what point the *line* that extends the segment intersects with the plane of the triangle. For the example shown in Figure 2, the equation to check at what point the line extending **PQ** intersects with the plane specified by **ABC** is:

$$\mathbf{A + x(B-A) + y(C-A) = P + t(Q-P).}$$

The variables **t**, **x**, and **y** can be calculated through the following equations (terms in bold typeface can be precomputed, as will be explained later):

$$t = \frac{(P-A) \bullet \mathbf{[(B-A) \times (C-A)]}}{(P-Q) \bullet \mathbf{[(B-A) \times (C-A)]}},$$

$$x = \frac{(\boldsymbol{C-A}) \bullet [(P-Q) \times (P-A)]}{[(\boldsymbol{B-A}) \times (\boldsymbol{C-A})] \bullet (P-Q)},$$

$$y = \frac{(\boldsymbol{A-B}) \bullet [(P-Q) \times (P-A)]}{[(\boldsymbol{B-A}) \times (\boldsymbol{C-A})] \bullet (P-Q)}.$$

If **t** is found to be less than zero or greater than one (the division can be avoided by simply comparing the size of the numerator to the size of the denominator), then the line segment **PQ** does not even intersect the plane of the triangle. If **t** does fall between 0 and 1 and the following inequalities hold:

$$0 \leq x, y \leq 1,$$

$$x + y \leq 1,$$

then the line segment-triangle test returns **true**. A potential architecture to accelerate this process is given in Figure 3. The interface accepts a stream of triangles representing swept volumes and a stream of triangles representing obstacles. A single bit for each swept volume is output, representing whether or not that swept volume is in collision. Internally, the accelerator could contain many triangle-triangle functional units to conduct pairwise checks in parallel, with each functional unit performing the six required line segment versus triangle tests in parallel.

Hardware space limitations quickly became apparent when pursuing this strategy, so we made several assumptions to reduce the complexity of the specialized functional units. The first was to use fixed point arithmetic to avoid expensive floating point operations, as well as to use 9-bit numbers for the coordinates instead of 32. Second, we decided that in order to reduce the amount of online computation needed, we would create a roadmap ahead of time, which would allow the precomputation of many of the terms for the robot triangles.

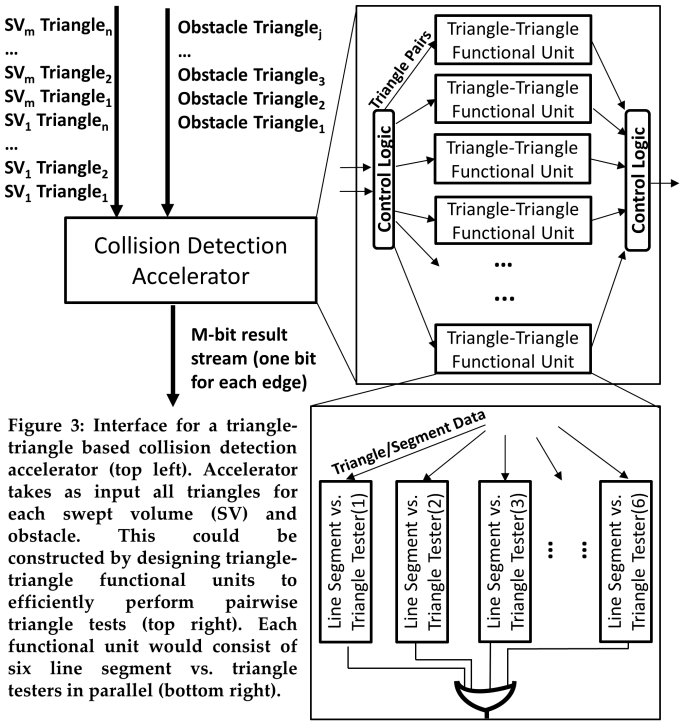


Figure 3: Interface for a triangle-triangle based collision detection accelerator (top left). Accelerator takes as input all triangles for each swept volume (SV) and obstacle. This could be constructed by designing triangle-triangle functional units to efficiently perform pairwise triangle tests (top right). Each functional unit would consist of six line segment vs. triangle testers in parallel (bottom right).

In the equations for **t**, **x**, and **y** above, the clauses in bold could be calculated ahead of time, assuming **ABC** represents a robot triangle and **PQ** is a line segment from an obstacle.

Even with both these simplifications, the complete parallelization of the triangle-triangle test requires 24 dot products and 7 cross products (the odd number of cross products arises from a corner case that must be checked). This is equivalent to 114 multiplications, 48 additions, and 21 subtractions. The high hardware cost to parallelize a single test was concerning. Indeed, we implemented our design and found that only a few tens (at most) of triangle-triangle functional units would fit on our prototype FPGA board, which was unacceptable for the throughput we desired.

B. Acceleration of Hardware-Friendly Algorithm

From the first design, we learned that what was needed was a co-designed algorithm to go with custom hardware. Existing algorithms simply did not match well to hardware acceleration. The next route we considered was to aggressively precompute not just some amount of robot geometry, but a whole suite of collision data, and to memoize this data on hardware for fast later access. We accomplish this by taking advantage of the way robotic perception data is structured. Perception sensors typically create an "occupancy grid" at some resolution, indicating the presence or absence of an obstacle at a given location in 3D space. Instead of creating triangle meshes from this data at runtime, we leverage the fact that a given resolution implies a finite number of possible obstacles. Expensive collision detection can be done for all edges *in advance* to calculate which regions in space each movement collides with. The precomputed data can be used to create a data structure for each edge that can be queried for membership of a given obstacle point (from here on called a "voxel"). This strategy represents a fundamental change to the PRM algorithm, which typically builds a roadmap for the specific environment at hand.

Our algorithm is similar to an approach by Leven and Hutchinson [13], except they went the opposite direction, creating data structures for each voxel that represented the edges that should be invalidated if present. In effect we are trading a much larger amount of up-front computation for a smaller amount of runtime work. Instead of having to build/reconnect a roadmap each time the environment changes, we build a roadmap in an obstacle-free environment and perform exhaustive collision detection once at design time. Then at runtime we simply access the precomputed data to see how the obstacles in a given environment affect the edges in the roadmap.

Because our goal is to achieve the highest degree of parallelism possible, we avoid storing and accessing the precomputed data in memory elements in software. Instead, we encode a binary representation for each discretized voxel and create logical circuits representing the collision data for each edge. Having a logical circuit representation made for an intuitive mapping to hardware descriptive languages. The

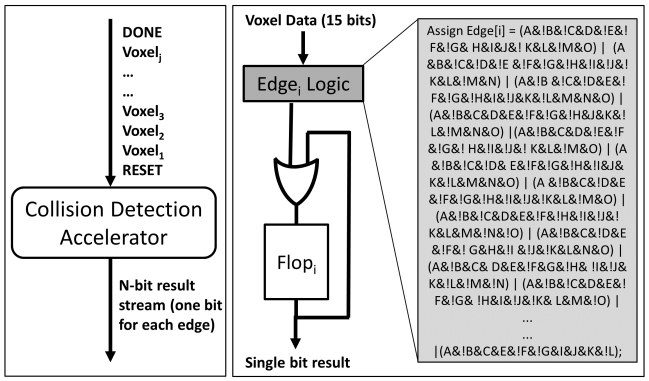


Fig. 4: The interface of a collision detection accelerator using precomputed data (left). Each collision detection circuit (CDC) contains the logic for an edge, an OR gate, and a flop (right). The variables A through O in the expanded Edge Logic block represent binary voxel ID representations, as will be explained in Section IV.

binary representation for any voxel is streamed over this logic, and if that point is in collision, the circuit outputs **true**.

We augment the collision logic for each edge with additional circuitry to maintain a limited amount of state regarding the task at hand. A given environment contains many obstacle voxels. If even a single voxel is in collision with a swept volume, the edge represented by that swept volume must be removed from consideration for use in a path. To achieve this, the output of the logic function can be stored in a flip-flop after being OR'ed with the flop's currently stored value, thus allowing many voxels to be streamed through, saving any positive result since the last RESET. The basic structure of this collision detection circuit (CDC) can be seen in Figure 4 along with the interface presented to the system. To interface with the accelerator, the host processor can send a RESET signal to instruct the accelerator to clear all flops, followed by a stream of obstacle voxel data (encoded in a fixed-length format), and finally a DONE signal, which initiates transfer of flop data back to the host.

Within our strategy of precomputing collision data there are several additional orthogonal design choices. For example, this strategy is agnostic to the configuration sampling method. Precomputed roadmaps by definition sacrifice asymptotic completeness for the sake of speed, so there is no need for sampling to be probabilistic. Indeed, any *a priori* knowledge about probable obstacle or goal regions can (and should) be leveraged to select more useful edges. In addition, the method of discretizing space can be adjusted to the case at hand to create the most useful representation in its most compact form possible.

We now summarize the steps in our workflow:

Preprocessing Steps (done once, at design time):

1. Build the roadmap. For this stage, no collision checking is done except for self-collisions and collisions with permanent features in the environment (e.g., the floor, ceiling, or a table that is always present). The goal of this step is to create a roadmap with sufficient coverage for solving the expected motion planning problems at an acceptable rate.

2. Discretize the working space of interest and collision check all edges of the roadmap, creating sets of the voxels that each edge collides with.

3. Encode the voxels in a binary representation and formulate logic functions for each edge. Use these logic functions to create RTL-synthesizable descriptions of the circuits in Verilog/VHDL.

Online Query Steps (done each time a plan is needed):

1. Use data from perception sensors to populate an occupancy grid at the same level of discretization at which the roadmap was collision checked.

2. Send all the voxels present in the occupancy grid to the accelerator and collect the results (a bitmask representing which edges are in collision). Use the results to modify the roadmap, setting the edges in collision to a cost of infinity.

3. Perform a graph search through the modified roadmap. A shortest path algorithm such as Dijkstra's can be used to find the shortest path through the map.

4. If a path is found, use the same local planner as in Design Step Two to guide the robot along a safe path to the goal.

C. Architectural Interface

As with any accelerator, the primary architectural issue is how the accelerator interfaces to the CPU cores. At the most extreme degree of integration, an accelerator is a glorified functional unit; a floating point unit (FPU) falls into this category. At the least extreme degree of integration, an accelerator is an I/O device like a GPU that communicates over a bus protocol like PCIe. We chose this latter option for our current implementation, largely due to the simplicity of prototyping the accelerator in this fashion. We have implemented the accelerator on an FPGA prototyping board with a PCIe interface for communication with a host computer.

In the future, we could imagine an architecture that is between these extremes of integration. Specifically, we are intrigued by the recent integration of FPGA logic with CPU cores, such as IBM's Coherent Accelerator Processor Interface (CAPI) [14] and Intel's integration of FPGA logic on the same die as a Xeon. The tighter integration of the FPGA with the CPU cores would lead to faster communication. The cache-coherent shared memory could also enable accelerator microarchitectures to elide some communication that is currently necessary.

IV. Implementation

We have implemented our accelerator to solve problems for a high-DOF robotic arm. We use an Altera Stratix V FPGA on Terasic's DE5 prototyping board. The FPGA interfaces over PCIe to an Intel Xeon 3.5 GHz 4-core processor with 16 GB of RAM.

The arm we use is the Kinova Jaco2, chosen for its many degrees of freedom. The Jaco arm has one shoulder, two

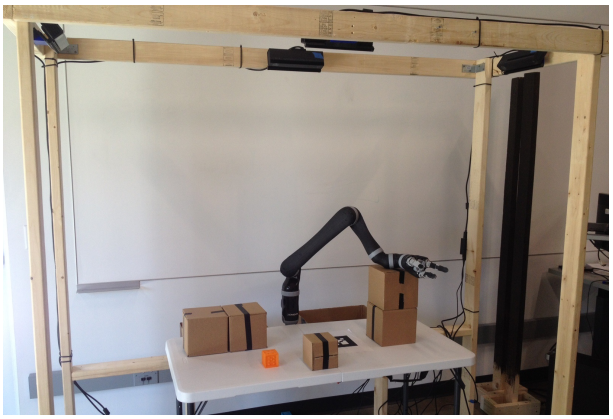


Fig. 5: Physical setup of the pick-and-place experiment. The work table, four Microsoft Kinects, and the base of the arm are all attached to the wooden frame.

elbows, three wrists, and three fingers. The shoulder and wrists have an infinite range of rotation, while the first and second elbows have ranges of 275 and 325 degrees, respectively. The numerous wrists give the arm great dexterity.

We demonstrate on the pick-and-place use case since this problem is ubiquitous in robotics applications. We placed the robotic arm in front of a table, and each scenario challenged the robot to reach out to grab a toy block while avoiding obstacles and return it to one of two bins on either side of its base, seen in Figure 5. Cardboard boxes, ranging from 5-30 cm in each dimension, acted as obstacles.

In the remainder of this section, we discuss the implementation of each stage of the workflow and the implications for the microarchitecture.

A. Design Time

Design Step One: Building the Roadmap

As we knew hardware constraints would be a limitation, great care had to be taken to create useful roadmaps of small size. To accomplish this, we made extensive use of the Klampt robot modeling package [15]. We wanted a high rate of success for working environments, so we followed a heuristic approach combined with probabilistic sampling. First, we created a very large (100,000 edge) roadmap using the PRM algorithm. Planning was done in an environment with only permanent obstacles present.

Because the goals (toy blocks) would always be sitting on the table, we needed thorough coverage of the space 4-8 inches above the table. To achieve this, we augmented the resultant graph with a set of configurations just above the surface of the table. We also added shortcut edges from the starting configurations to various spots over the table. These shortcut edges both add useful cycles to the roadmap and also often provide very direct paths in the absence of certain obstacles [16].

We then profiled this very large and augmented PRM over 10,000 scenarios. These test cases were generated randomly in Klampt by parameterizing details about the world, such as size/number/placement of obstacles and the placement of the goal. For each environment, the PRM was challenged to find a collision-free path to the goal, and if it could, the edges used in the path were saved in a data structure. After testing on the 10,000 scenarios we then had an array showing which edges were being used in paths frequently, which were being used some of the time, and which edges were never used. This enabled us to prune the roadmap by deleting the edges that were never used or used infrequently. The safeness of the pruning was tested by then profiling the reduced set of edges against a *new* set of random scenarios and verifying the success rate had not been significantly compromised. This process can be done iteratively, profiling and pruning until the desired size is reached. We found that as few as 1024 edges is sufficient to maintain a very high success rate (>98%) for this pick-and-place task.

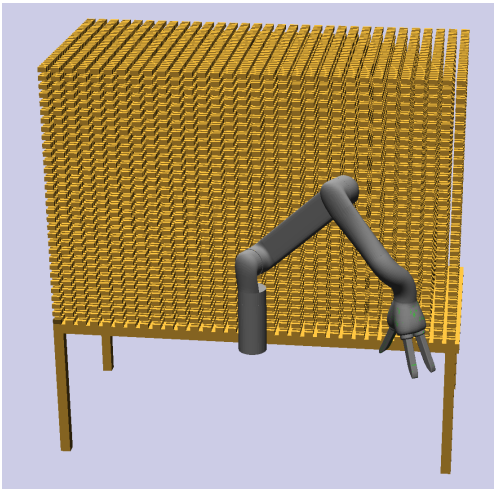
Design Step Two: Discretizing the Workspace and Collision Checking

For the pick-and-place scenarios, the workspace was defined to be the area directly over the table, extending 80 cm high. Once the area of interest is established, the next decision is how exactly to discretize. For simplicity, we employed a simple uniform grid. We examined a few different resolutions ranging from 4 bits to 10 bits in each dimension. The two competing concerns are the desire to have enough resolution such that the occupancy grid is an accurate representation of the actual environment, and the fact that as resolution increases, the logic function for each edge consumes more hardware. The logic element usage for several levels of resolution is shown in Table I. We chose five bits for each dimension as a good middle ground that provided sufficient precision and also took up a reasonable amount of FPGA resources. The table used in our experiments is 121 centimeters long and 60 centimeters deep, so with a 5 bit resolution/dimension, each block in the occupancy grid is $3.75 \times 1.875 \times 2.5$ cm. Figure 6a shows the discretized workspace. Figure 6b shows a non-uniform, more sophisticated way that one could discretize. By using knowledge about expected hard or easy/less-important regions, one can selectively increase or decrease resolution to maintain performance while saving space in the logic functions. For the same roadmap, the discretization strategy in Figure 6b takes up 27% less space on the FPGA, at the cost of slightly higher design effort.

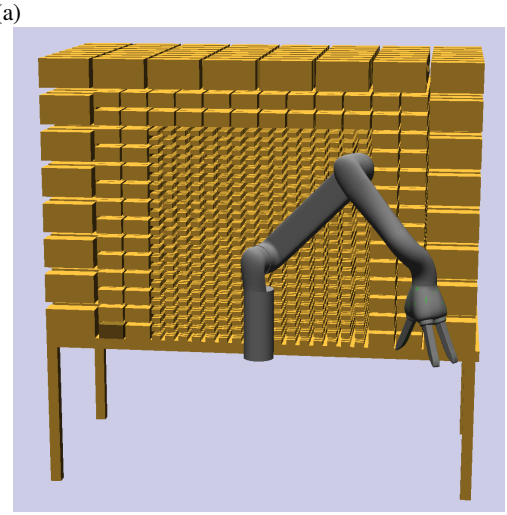
Each edge in the roadmap constructed in Design Step One is then collision checked in Klampt in the environment containing the full occupancy grid of discretized space. For

Bits Per Dimension	**4**	**5**	**6**
Logic Elements/Edge	45	160	700
Voxels Colliding/Edge	75	278	1100

TABLE I: Effect of changing resolution on logic element usage. The logic element usage correlates roughly to the number of voxels with which each edge collides.



(a)



(b)

Fig. 6: a) The workspace shown as uniformly discretized. b) A more sophisticated approach can achieve space savings by selectively choosing critical regions to have high resolution and allowing less important regions to be more coarse.

each edge we create a set of the voxels in collision with that movement. If an edge intersects any part of a voxel, that voxel is included in the set.

Design Step Three: Implementing Logic Functions on FPGA

At the end of Design Step Two, there is a set for each edge containing all the voxels with which that edge collides. A binary representation for each voxel is easily derived since we used a uniform grid in the discretization. We used these binary representations to create a logic function for each edge. For example, if edge 147 collides with the voxels (1,3,5) and (1,3,6), then the voxels are represented by:

00001 00011 00101
00001 00011 00110

The logic functions are then created by simply assigning a variable to each voxel bit (A through O) and combining the two in disjunctive normal form. For this example, the result would be:

$$Edge147 = (!A\&!B\&!C\&!D\&E\&!F\&!G\&!H\&I\& J\&!K\&!L\&M\&!N\&O)||(!A\&!B\&!C\&!D\&E\&!F \&!G\&!H\&I\&J\&!K\&!L\&M\&N\&!O).$$

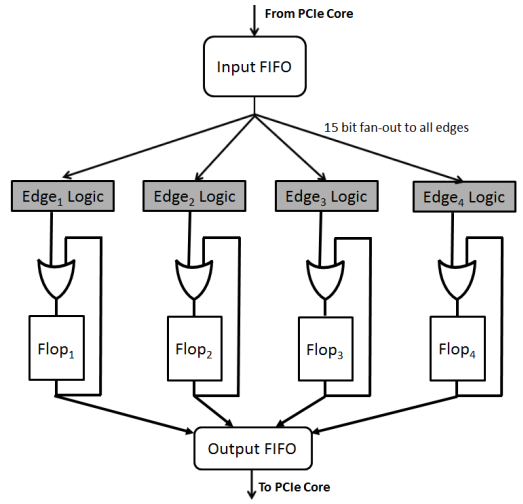
In this simplistic example, the edge collides with only two obstacle voxels; each edge in the actual roadmap collides with 200-500 voxels, so the logic functions can become quite large and take up the bulk of the area of the hardware design. Luckily, there is a significant amount of redundancy in these equations, which Leven and Hutchinson refer to as "spatial coherence" [13]. For example, the above equation can be simplified to:

$$Edge147 = (!A\&!B\&!C\&!D\&E\&!F\&!G\&!H\&I\& J\&!K\&!L\&M)\&[(!N\&O)||(N\&!O)].$$

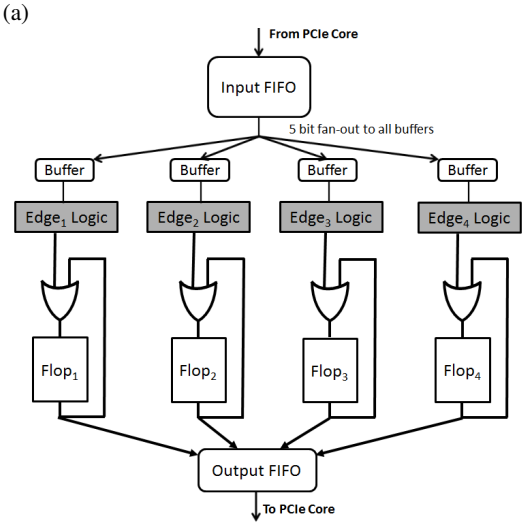
We took several actions to reduce the amount of hardware each edge consumes. Logic minimization is a well-studied problem due to its usage in EDA tools. We used a version of the popular *espresso* heuristic logic minimizer [17][18]. *Espresso* can accept as input a set of truth tables, so the sets created in the previous step were converted to truth tables by simply using the binary encoding of each voxel. We ran these truth tables through *espresso* in groups of 16 to allow the tool to minimize the logic across edges as well. We then converted the output to equivalent Verilog expressions; using *espresso* before converting to Verilog enabled greater than 25% savings in logic utilization on the FPGA, even though all CAD tools (Quartus in this case) do logic minimization of their own. It is likely even more benefit could be realized by "smartly" grouping together edges that shared more in common with each other. Selecting the best edges to group together is a challenging problem ("**N** choose 16" is quite large when **N** is in the thousands) that will be the subject of future work.

Another important design issue is how to distribute the voxels to the CDCs. The board communicates with the host computer over PCIe; as voxels are streamed over the bus to the FPGA they are put into a dual-clocked FIFO, filling up at the bus frequency and draining at the logic frequency. The initial design is in Figure 7a. The fifteen bits of each voxel (five bits for each dimension, as discussed above) fan out from the FIFO to the logic for each edge. Each edge's logic function has an associated flip-flop. The input to the flop is the OR of the edge logic output with the current value (seen in Figure 4). After the input FIFO is drained, the results are fed into an output FIFO which transmits the collision results back to the host computer.

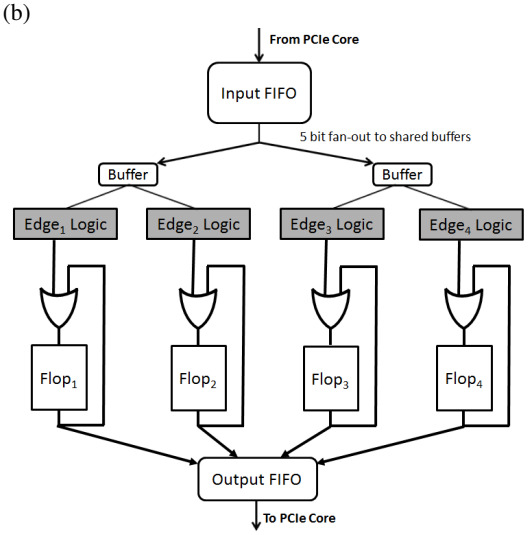
This design is simple, but it is difficult for the FPGA to route in time due to the high fan-out of the input signals. Each input bit must fan out to thousands of CDCs, each of which has several hundred clauses in its logic function. Even clocking the FPGA at 31.25 MHz (1/4 the frequency of the PCIe bus), only 1024 edges could fit on the FPGA. To alleviate this, we



(a)



(b)



(c)

Fig. 7: a) The design unbuffered with high fan-out. b) Individual buffers for each CDC. c) Multiple CDCs sharing a buffer. Note that in b) and c), the flops need an **enable** port to latch only when the buffers contain full values.

investigated a slightly different design. Instead of fanning out the 15 bits of voxel data to all CDCs, only five wires fan out. These five wires are multiplexed over three cycles to send the full 15 bits of voxel data, accumulating the data in a shift register at each CDC. The flops latching the results of the edge logic now need a signal to enable storage only every 3 cycles when input data is valid. This design is in Figure 7b. Routing significantly improved with this design, but at the cost of greatly increased logic utilization (an 83% increase). FPGA CAD tools are somewhat opaque, but we believe the increase comes primarily from fewer opportunities for the CAD tools to optimize/re-use logic clauses now that the CDCs compute on unique sets of inputs, rather than from the additional structures introduced.

We found a middle ground between these design points that balances routing difficulty and logic utilization. Instead of allocating a buffer for each CDC, buffers are shared in a hierarchical fashion between groups of CDCs. Figure 7c shows an example of this design. Table II shows the difference in logic utilization/edge for the original case, the case of each CDC having its own buffer, and of 16, 32, and 64 CDCs sharing a single buffer. These adjusted strategies did not recover all of the logic savings of the unbuffered design, but they were able to be compiled and routed much easier. The 32 edge/buffer design allowed 2500 edges to fit on the FPGA, which is more than sufficient for this prototype.

One potential concern about using this buffering technique is that it now takes three cycles to process a point, instead of a single cycle. However, the decreased routing demands of the strategy in Figure 7c allow it to be clocked at the same clock frequency as the PCIe bus (125 MHz), compared to the unbuffered design at 1/4 the frequency. The total effect of the more complex micro-architecture is thus both larger logic utilization and modestly higher throughput at the same roadmap size.

B. Runtime

Runtime Step One: Create Occupancy Grid

Perception is done in our experiment with several Microsoft Kinects. These supply sufficient accuracy for our purposes, are relatively cheap, and produce data in a convenient format. To get a complete view of the workspace, we used four Kinects, one on each side of a wooden frame built around the table.

Design Choice	Logic Element Usage
Unbuffered	1
Unbuffered/Non-uniform	0.73
Individual Buffers	1.83
Buffers Shared(16)	1.31
Buffers Shared(32)	1.26
Buffers Shared(64)	1.23

TABLE II: Effect of changing hardware design on logic element usage, normalized to the unbuffered design. All designs were uniformly discretized except the one mentioned.

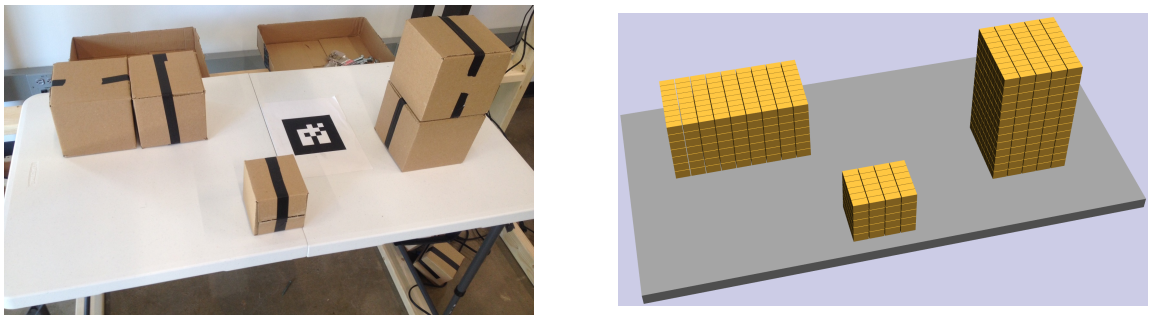


Fig. 8: A real example scenario (left), and the discretized occupancy grid (right).

We secured the Kinects on 3D printed mounts to avoid having to frequently recalibrate the cameras (seen in Figure 5). The output streams from the four Kinects are merged with an iterative closest point (ICP) algorithm to create a unified view of the environment [19][20]. An example of a discretized occupancy grid is shown in Figure 8. The Kinects also identify the location and color of the goal(s) present on the table, to be used in the graph search routine.

Runtime Step Two: Collision Check on FPGA

The occupancy grid is then sent over PCIe to the FPGA and the resulting collision bitmask is collected. The data coming back from the FPGA represents a vector of which edges are in collision. For each edge in collision, the cost in the roadmap is set to infinity to ensure this path will not be taken during a query. Adjusting the cost is faster than actually removing the edge from the data structure.

Runtime Step Three: Graph Search

For each pick-and-place scenario presented to the robot, the objective is to find a path to a spot 10 cm above the goal with the hand pointed down, grasp the object, and return to one of the two bins on either side of the arm depending on the color. The location of the goal is used to select suitable goal configurations in the roadmap. We accelerate the selection process by precomputing forward-kinematic transforms for all n configurations in the roadmap. This data is stored in a k-d tree that can be very quickly accessed to find which configurations (if any) can be used as a suitable destination for this problem scenario. This structure scales well, with searching the tree taking only $log(n)$ time.

A path can now be found using any graph search algorithm on the modified roadmap, using the arm's current configuration as the starting node. We used an unoptimized Dijkstra's shortest path algorithm with a heap implementation; faster techniques certainly exist. Edge costs were calculated back in Design Step One and stored for rapid access. Various metrics could be used for edge cost. Swept volume, distance traveled by the end-effector, or the time required to execute a movement are all metrics that make sense. Swept volume yielded the smoothest paths, so we use an approximation of swept volume for this experiment. The approximation is simply a weighted sum of joint angle differences. If no non-infinite cost path is found to the goal configuration, then the graph has been bisected by obstacles and there is no collision-free path through the precomputed roadmap. In this (unlikely) case, one could fall back onto a conventional software planning routine that has asymptotic completeness. In this way, the common case could be made fast, and the uncommon case would still find a solution (if one exists).

V. Experimental Results

In this section we evaluate the performance, power, and efficacy of our FPGA prototype of the motion planning chip design. In addition, we deconstruct how much of the performance improvements are due to the hardware implementation versus the aggressive precomputation in our algorithm. To determine the source of the speedup, we wrote and evaluated a software version of the same strategy. This implementation used the collision data collected in Design Step Two above to create hashsets of the points in collision with each edge. Collision detection in this implementation simply consists of querying obstacle voxels for membership in all of the hashsets. The results of this test can be used to eliminate edges from the roadmap in exactly the same fashion as described above. The code was instrumented with OpenMP directives to enable the CPU to take advantage of the inherent parallelism in the strategy.

Given the simple and highly parallel nature of this algorithm, it also warranted testing on a GPU. We implemented the same hash set strategy in CUDA and tested on an NVIDIA Tesla K20 (which contains 2496 CUDA cores). Both the CPU and GPU implementations were highly tuned for performance to enable fair comparisons.

A. Performance

This section compares the performance of the hardware design to several alternative solutions. The motion planning process consists of several tasks, so we provide a comparison of collision detection in isolation, as well as a comparison of the entire planning process as a whole. To reliably time the speed of the various components of the now-heterogeneous system controlling the robot, we fed 10,000 occupancy grids into the planner and took measurements.

With our processor, the total time between obstacle data being sent, and having a motion plan to execute, is less than 650 microseconds (of which only 16 is collision detection). Previously, collision detection has always been the bottleneck

Custom Hardware	Precomputed Hashsets	
FPGA	CPU	GPU
16	9,550	1,100

TABLE III: The time (in microseconds) required to perform collision detection using the FPGA as a collision detection accelerator compared to using the same aggressive precomputation to create hashsets for fast query on a CPU or GPU. Results here are for roadmaps containing 2,500 edges.

in planning algorithms, but in this approach it is actually one of the fastest components. The vast majority of the 650 microseconds to produce a plan is spent on operating system delays and in the graph search phase. Most occupancy grids for the example scenarios contain <500 points and, at the clock speed of 125 MHz, are processed by the FPGA in less than 16 microseconds. Traversing the k-d tree to find suitable destination configurations takes 10-20 microseconds. Modifying the roadmap with collision data to assign infinite cost to colliding edges requires 50 microseconds. This leaves the latency for communication and graph search. Subtracting the computation time from the round trip latency across the FPGA yields 150 microseconds for communication. Unsophisticated drivers were used to interface with the FPGA over PCIe, and this latency could be reduced. Communicating the same data over PCIe with mature GPU drivers takes around 15 microseconds each way, so the I/O is a feasible target for optimization. The longest step by far is Dijkstra's shortest path algorithm, which requires 425 microseconds in our implementation. It is likely that at a relatively small roadmap size there are faster ways than Dijkstra's to find a short path, but this is not the focus of our work.

The design-time steps are very slow compared to what is executed at runtime. The computationally expensive collision detection required to build the logic functions takes on the order of 45 minutes, and heuristic roadmap pruning can take several hours. Both these steps happen only once at design time, however, and thus are not a concern.

Table III shows data comparing the different methods of accelerating collision detection. When running with 16 threads, the software hashset implementation takes less than 10 milliseconds to produce the collision data on the same roadmap

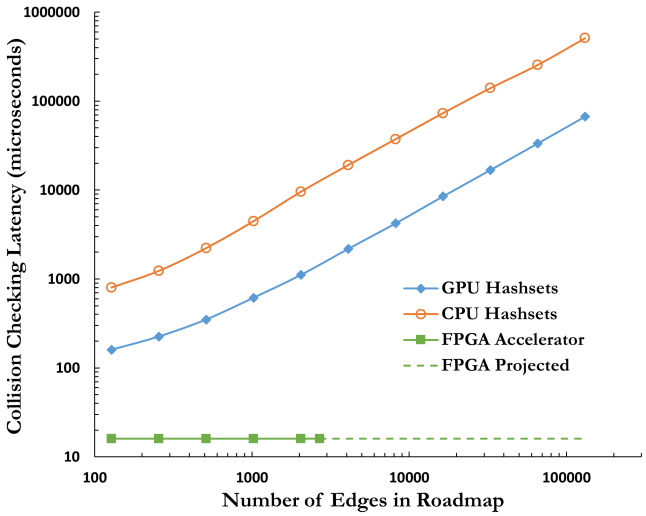


Fig. 9: Both CPU and GPU solutions scale linearly with respect to the number of edges in the roadmap. The FPGA-architecture described in this paper, however, executes completely in parallel, and thus has a latency independent of roadmap size. All results reflect the latency to process occupancy grids of the same size.

used in the experiments on the FPGA. When fully parallelized, the GPU hashset kernel spawns more than 500,000 threads and completes queries in less than 1.1 milliseconds for the same roadmap. NVIDIA's profiling tool *nvprof* was used to evaluate the memory transfer times. For the runtime query transfers (transferring obstacle voxel data to GPU, and result data back to host), communication happened in less than 15 microseconds each way. Transferring the actual hashsets takes much longer, at 8 milliseconds, but this only needs to happen a single time, after which many queries can be made.

These results demonstrate that significant benefit is gained by attacking the problem from both sides. Our speedup comes from both the improved algorithm to reduce problem complexity and a hardware implementation that can exploit more parallelism. Even though the software hashset implementations achieve a large speedup compared to conventional solutions, they do not scale as well as the custom hardware solution. Any given CPU/GPU with a fixed number of hardware threads will experience a linear increase in compute time as the number of edges grows in the roadmap. This effect can be

Custom Hardware	Precomputed Hashsets		Conventional Software Approaches (CPU)	
FPGA	CPU	GPU	PRM	RRT
650	10,000	1,600	815,000	756,000

TABLE IV: The time (in microseconds) required to produce motion plans using the FPGA as a collision detection accelerator compared to using hashsets on either a CPU or GPU, or conventional software approaches (run on a CPU). RRT is a single-query probabilistic algorithm very similar to PRM, used more commonly when speed is a concern rather than data-reuse. These numbers include time to perform a path search and the latency to transmit data to and from the remote devices for the FPGA/GPU solutions. Although the CPU and GPU can achieve impressive speed-ups using hashsets, these are on relatively small graphs (2,500 edges), and these solutions scale linearly with roadmap size.

seen in Figure 9 showing the performance of these solutions on roadmaps of increasing size. By contrast, the hardware design is completely parallel and takes constant time to do the computation regardless of the number of edges. The only obstacle in achieving this parallelism for huge roadmaps is the fixed hardware budget.

Collision detection is only one part of motion planning, so we next consider the timing of the motion planning process as a whole. Table IV shows the time to generate a complete motion plan using the FPGA, precomputed hashsets, and conventional solutions. Running the PRM algorithm on the high performance workstation described above on the same sample environments took 0.8 seconds to produce a solution. Rapidly Exploring Random Trees (RRT) is a single-query probabilistic method that is slightly faster at 0.75 seconds. Our approach produces motion plans three orders of magnitude faster than conventional solutions running on CPUs and two orders of magnitude faster than current GPU methods.

B. Power and Energy

Another advantage enjoyed by specialized accelerators is power efficiency. In order to quantify power consumption, we wired a high-wattage current shunt resistor in series with the DE5 board's power supply to determine the peak power consumption of our design while computing collisions for a 1024-edge roadmap. We used an oscilloscope to measure the voltage drop across the resistor during a collision detection query, which caused the board power to increase from a nominal 12W to a peak 15W. The board has other components contributing to this power consumption, so the FPGA would be some fraction of this. GPU solutions, in contrast, are often much higher power. The Tesla K20 is rated for 225 W, and it takes a longer time to execute than the FPGA, thus consuming far more energy.

C. Motion Planning Efficacy

The FPGA solution we present in this work is robust to a wide range of pick-and-place environment distributions. Simulation allowed profiling the final roadmap over tens of thousands of scenarios, and results were verified by implementing the physical system end-to-end and testing it against real environments. The method is conservative; discretization of space makes obstacles appear larger than they are. No planned motion will ever mistakenly be in collision, which is of the utmost importance for safety reasons. However, using this conservative discretization strategy does mean that sometimes the chip will report failure when a solution exists. This situation could occur either due to obstacles appearing too large, or simply if the roadmap created at design-time does not contain appropriate edges for the given scenario. Testing showed that these situations rarely happen in practice. With the help of heuristic pruning, edges were chosen such that even modestly-sized roadmaps that would fit on an FPGA solved example scenarios with a success rate of greater than 98%. In the rare cases in which a path is not found, the system can fall back on a conventional probabilistic algorithm that maintains asymptotic completeness.

VI. Future Work

Both the timing and power measurements would likely improve even further if the design were implemented on an ASIC instead of an FPGA. An ASIC would also likely increase the number of edges that could fit, enabling the use of much larger roadmaps. We used the Synopsys CAD tools to estimate that a 1024-edge design would use fewer than 10 million transistors when mapped to an ASIC. Given that chips with more than 1 billion transistors have been available for more than 10 years, an ASIC could easily fit 100,000-edge roadmaps using an established technology process. The ability to fit roadmaps of this size would allow the chip to solve problems comparable to those faced in industrial applications (in real-time).

VII. Related Work

There have been a variety of approaches by the robotics community to accelerate motion planning. Atay and Bayazit focused on directly accelerating the PRM algorithm on an FPGA [21]. In this work the authors present a design for an accelerator not just for collision detection, but for most of the learning phase of a batch-variant of the PRM algorithm. The authors create functional units to perform the random sampling of new configurations, as well as nearest neighbor search. They hoped to parallelize triangle-triangle testing as well, by implementing multiple intersection test modules (similar to our initial approach described in Section III.A). The idea was that the roadmap would be transmitted back to the host to perform the query phase. However, even on a small low-DOF toy robot example they found they could not fit their design onto an FPGA, resorting to simulation instead.

There has also been work focusing on the use of GPUs to parallelize conventional planning algorithms [5][6][7]. For example, Bialkowski et al. divide the collision detection tasks of the RRT* algorithm into three parallel dimensions to create grids of thread blocks that can simultaneously perform collision computations [7]. The main limitation of approaches utilizing GPUs is that a GPU provides only a constant factor speedup; once the cores of the GPU are fully utilized, the execution time scales linearly with problem size.

The algorithmic side of the problem is also frequently studied. One strategy is to use a "lazy" approach and elide collision checks until an edge is a candidate for use in a shortest path, instead of performing collision detection when building the roadmap [22]. This strategy does reduce the number of collision checks that must be done, but still results in expensive collision checking being done at runtime.

VIII. Conclusions

In this work we have presented a novel microarchitecture for the hardware acceleration of motion planning algorithms. The solution achieves several orders of magnitude speedup over the current state of the art. This sub-millisecond speed is sufficient

to enable previously infeasible robotic applications, such as real-time planning in dynamic environments. We demonstrated an end-to-end implementation on a real, high-DOF robotic arm, and the accelerator was able to solve dynamic pick-and-place scenarios with a high rate of success. The methodology is general enough to be applied to a wide range of robotics applications and scales well with roadmap size. In short, the hardware acceleration of motion planning algorithms is a highly effective approach.

ACKNOWLEDGMENTS

This work was supported in part by the Defense Advanced Research Projects Agency (DARPA) Robotics Fast Track Program. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the view of DARPA OSRF, or the US government. We thank Kris Hauser for his assistance with the Klampt robotics software package [15]. Altera Corporation also provided support for this work. We would additionally like to thank Barret Ames, Saeed Alrahama, Xiangyu Zhang, Martha Barker, and Hayden Bader.

REFERENCES

[1] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, "A reconfigurable fabric for accelerating large-scale datacenter services," in *Proceedings of the 41st Annual International Symposium on Computer Architecuture*, pp. 13–24, 2014.

[2] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "ShiDianNao: Shifting vision processing closer to the sensor," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pp. 92–104, 2015.

[3] X. Liu and Y. Deng, "Fast radix: A scalable hardware accelerator for parallel radix sort," in *12th International Conference on Frontiers of Information Technology*, 2014.

[4] D. E. Shaw, M. M. Deneroff, R. O. Dror, J. S. Kuskin, R. H. Larson, J. K. Salmon, C. Young, B. Batson, K. J. Bowers, J. C. Chao, M. P. Eastwood, J. Gagliardo, J. P. Grossman, C. R. Ho, D. J. Ierardi, I. Kolossváry, J. L. Klepeis, T. Layman, C. McLeavey, M. A. Moraes, R. Mueller, E. C. Priest, Y. Shan, J. Spengler, M. Theobald, B. Towles, and S. C. Wang, "Anton, a special-purpose machine for molecular dynamics simulation," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, pp. 1–12, 2007.

[5] J. Pan and D. Manocha, "GPU-based parallel collision detection for fast motion planning," *International Journal of Robotics Research*, Feb. 2012.

[6] J. Pan, C. Lauterbach, and D. Manocha, "g-Planner: Real-time motion planning and global navigation using GPUs," in *Proceedings of the AAAI Conference on Artificial Intelligence*, 2010.

[7] J. Bialkowski, S. Karaman, and E. Frazzoli, "Massively parallelizing the RRT and the RRT*," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2011.

[8] S. Murray, W. Floyd-Jones, Y. Qi, D. Sorin, and G. Konidaris, "Robot motion planning on a chip," in *Robotics: Science and Systems*, June 2016.

[9] T. Lozano-Perez, "Spatial planning: A configuration space approach," *IEEE Transactions on Computers*, pp. 108–120, 1983.

[10] L. E. Kavraki, P. Svestka, J.-C. Latombe, and M. H. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *IEEE Transactions on Robotics and Automation*, vol. 12, no. 4, pp. 566–580, 1996.

[11] N. M. Amato and L. K. Dale, "Probabilistic roadmap methods are embarrassingly parallel," in *Proceedings of the IEEE International Conference on Robotics and Automation*, pp. 688–694, 1999.

[12] C. Ericson, *Real-Time Collision Detection*. CRC Press, 2004.

[13] P. Leven and S. Hutchinson, "A framework for real-time path planning in changing environments," *The International Journal of Robotics Research*, vol. 21, no. 12, pp. 999–1030, 2002.

[14] J. Stuecheli, B. Blaner, C. Johns, and M. Siegel, "CAPI: A coherent accelerator processor interface," *IBM Journal of Research and Development*, vol. 59, no. 1, pp. 7–1, 2015.

[15] K. Hauser, "Robust contact generation for robot simulation with unstructured meshes," in *Proceedings of the International Symposium on Robotics Research*, 2013.

[16] D. Nieuwenhuisen and M. H. Overmars, "Useful cycles in probabilistic roadmap graphs," in *Proceedings of the IEEE International Conference on Robotics and Automation*, vol. 1, pp. 446–452, 2004.

[17] R. Brayton, G. Hatchel, C. McMullen, and A. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*. Boston, MA: Kluwer Academic Publishers, 1984.

[18] R. Rudell, "Multiple-valued logic minimization for PLA synthesis," Tech. Rep. UCB/ERL M86/65, EECS Department, University of California, Berkeley, 1986.

[19] T. Wiedemeyer, "IAI Kinect2." https://github.com/code-iai/iai_kinect2, 2014 – 2015. Accessed June 12, 2015.

[20] S. Niekum, "ar_track_alvar." http://wiki.ros.org/ar_track_alvar, 2011 – 2015.

[21] N. Atay and B. Bayazit, "A motion planning processor on reconfigurable hardware," in *Proceedings of the IEEE International Conference on Robotics and Automation*, pp. 125–132, 2006.

[22] K. Hauser, "Lazy collision checking in asymptotically-optimal motion planning," in *Proceedings of the IEEE International Conference on Robotics and Automation*, pp. 2951–2957, 2015.