A Programmable Architecture for Robot Motion Planning Acceleration

Sean Murray
Duke University
s.murray@duke.edu

Will Floyd-Jones
Duke University
william.floyd.jones@duke.edu

George Konidaris
Brown University
gdk@cs.brown.edu

Daniel J. Sorin
Duke University
sorin@ee.duke.edu

***Abstract*—We have designed a programmable architecture to accelerate collision detection and graph search, two of the principal components of robotic motion planning. The programmability enables the architecture to be applied to a wide range of different robots and motion planning applications. We present the architecture of our accelerator and describe and evaluate its microarchitecture implementation.**

I. Introduction

Motion planning is the process of finding a collision-free path from a robot's starting position to a goal state. Our contribution in this work is a programmable architecture that accelerates several aspects of motion planning in an efficient manner. More specifically:

- We develop the microarchitecture of a novel collision detection accelerator that is fully retargetable.
- We present the microarchitecture of a novel accelerator for path search.
- For both the collision detection and path search accelerators, we implement the microarchitecture in Verilog.

Our motion planning accelerator microarchitecture can perform collision detection and calculate a path in less than 3 microseconds, with a modest power consumption of 35 watts. This is several orders of magnitude faster than current state-of-the-art approaches [7], [18].

II. Background

A. Components of Motion Planning

There are four main components involved in creating a motion plan.

Perception is the use of a combination of sensors and processing to produce a model of the environment. We assume sensors that produce an occupancy grid. An occupancy grid is a data structure representing which regions of space contain obstacles in a discretized view of the environment. Each discretized region of space is termed a "voxel", a 3D (volumetric) pixel.

Roadmap construction is the creation of a graph of poses and motions in a robot's configuration space. Each vertex in this graph completely defines the state of the robot in a specific pose, and each edge defines a motion between poses. Common algorithms create the roadmap by randomly sampling poses from configuration space.

Collision detection determines if a motion of a robot is in collision with itself or the environment. Triangle meshes are commonly used as the models for the robot and environment, so collision detection involves checking if any triangles of the robot model intersect with those of the environment model. This process is quite computationally expensive, and is the bottleneck in conventional planning algorithms [2].

Path search traverses the roadmap to check if a path from the starting position to the goal exists and to identify optimal paths. It is often done with variants of A* or Dijkstra's algorithm [3].

B. Related Work

Prior work has investigated ways to speed up motion planning, but none of it meets our goals.

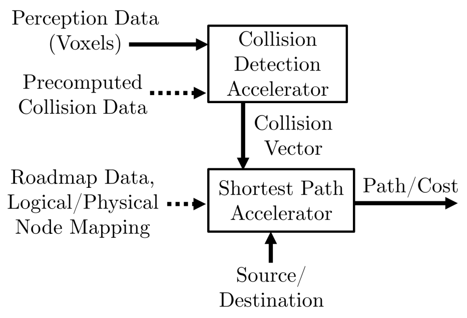
Roadmap Construction. Roadmap construction can be accelerated by using algorithms that improve convergence behavior [10], reduce the workload by keeping the roadmap to a reasonable size [23], or improve nearest neighbor search [25]. Our approach completely removes the runtime latency of roadmap construction by performing it once at design time.

Collision Detection. Atay and Bayazit [1] developed hardware to directly accelerate the triangle-triangle intersection tests of commonly-used collision detection subroutines. The huge resource demands of the design constrained it to impractically small problems. Murray et al. created a custom hardware microarchitecture [14], [15] that performs exhaustive collision detection ahead of time and uses the data to create a specialized circuit for each motion of the roadmap. While fast, this design is not retargetable to different robots and scenarios. Dadu-P [12] takes a similar approach to Murray et al., but stores edge data in memory rather than in circuits, enabling reconfigurability. The reliance on external memory transfer causes a 25X latency increase in collision detection compared to Murray et al., and this effort also did not accelerate path search.

Path Search. Bondhugula [5] developed hardware to accelerate a block-variant of the Floyd-Warshall algorithm. Sridharan [19] designed an accelerator for a parallelized version of Dijkstra's algorithm, and Takei [21] extended this for large-scale graph processing. These accelerators achieve insuffcient performance for our goals, because they rely on slow memory accesses, and they do not exploit properties of the path search problem that are specific to robot motion planning.

III. Roadmap Construction

Our roadmap construction approach is similar to Leven and Hutchinson [11], and it differs from conventional sampling-based planners such as PRM or RRT. Conventional planning



Perception Data (Voxels)
Precomputed Collision Data
Collision Detection Accelerator
Collision Vector
Roadmap Data, Logical/Physical Node Mapping
Shortest Path Accelerator
Path/Cost
Source/ Destination

Fig. 1. The overall dataflow of our design. Dotted arrows indicate communication that happens during the programming phase, and solid arrows indicate runtime communication.

algorithms incrementally build up a roadmap at runtime to navigate around the obstacles present at that time.

Our approach involves precomputing a more general and much larger roadmap than a conventional algorithm. Any *a priori* knowledge about the scenario can be leveraged by including fixed objects such as walls or tables. (This is an optional optimization and the strategy does not depend on it.) The roadmap is made large and redundant enough to be robust to obstacles. This allows successive queries to be done rapidly in dynamic environments without reprogramming the accelerator. We find good results from beginning with an extremely large roadmap (hundreds of thousands of edges) and pruning using heuristics [14], [15].

IV. Collision Detection

Our collision detection workflow involves two stages. First, before runtime, we precompute collision detection results for the roadmap in a discretized view of the environment. Second, at runtime, the collision detection accelerator streams in the obstacle voxels present and flags edges that are in collision.

A. Precomputation of Collision Data

To minimize work at runtime, we perform a large amount of precomputation. We discretize the environment into voxels. We then exhaustively precompute all possible collisions between the swept volume of every motion in the roadmap and the voxels in the discretized space. This is time-consuming, but does not take place at runtime and is thus not on the critical path. After this process, each motion has a corresponding set of voxels with which it collides. During the accelerator's programming phase, this precomputed data is sent to the accelerator to be used at runtime.

B. Runtime Collision Detection

Given the precomputed data, the runtime task is to determine which roadmap motions collide with the current environment, provided by the perception system as an occupancy grid. Any motion that collides causes that motion's edge in the roadmap to be temporarily removed until the environment changes.

Previous work [14], [15] built hardware that performed collision detection by creating circuits of combinatorial logic that directly correspond to the swept volume of each motion in the roadmap. This circuit-based approach is high performance, but also highly inflexible. To achieve programmability, we implement a sea of compute-elements containing registers that are filled with swept-volume data for the robot and roadmap of interest. After configuration, at runtime the voxels from the sensed occupancy grid are sent to the accelerator to be checked against the swept volumes. The results of collision detection are then to the path search architecture. The overall dataflow for the accelerator can be seen in Figure 1.

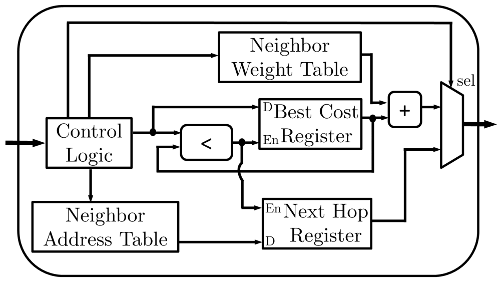


Fig. 2. Each Bellman Ford Compute Circuit (BFCC) has a table of the physical addresses of its logical neighbors, and a table of their edge weights.

V. Path Search

The main challenge in designing a programmable microarchitecture for accelerating graph processing is to be able to handle any expected graph topology yet have reasonable resource requirements. Our key insight is that since our strategy involves a roadmap statically constructed ahead of time, we can guarantee certain properties such as its maximum degree, maximum edge weight, and maximum path cost. Bounding these quantities enables us to design much more compact and efficient storage structures and datapaths than if we allowed for arbitrary graphs.

The approach we take is a dataflow microarchitecture designed to perform the Bellman-Ford algorithm. The microarchitecture consists of a sea of circuits we refer to as Bellman-Ford Compute Circuits (BFCCs) connected via a low-cost interconnection network used to handle different topologies.

A. Bellman-Ford Compute Circuits

Every (logical) vertex in the graph is statically assigned to a physical BFCC on the chip. The physical addresses of each of the vertex's logical neighbors are stored in a table. The edge weights to each of these neighbors are stored in another table.

Each BFCC has a register to hold its current best-cost to the destination. These registers are all initialized to a maximum value which represents infinity. To start the search, the BFCC to which the destination node was mapped is updated to a cost of zero. The destination vertex then iterates over its neighbor table, and sends to each neighbor a message with its cost (zero) added to that neighbor's edge weight. When the neighbor receives this message, it compares this new cost with its current cost. If the new cost is less than its current cost, then several things happen. First, the vertex updates its best-cost register as well as its next-hop pointer. Second, it begins iterating over its own neighbor table to find the physical addresses of its neighbors, and sends each of them a message with its cost added to that neighbor's edge weight. Figure 2 shows the basic microarchitectural layout of the Bellman Ford Compute Circuit.

Aside from cost update messages, the BFCCs handle two other types of messages. If the BFCC receives a next-hop query, it responds with the address of the neighbor from which it received its best-cost. This allows the path itself to be extracted. The BFCC can also receive a best-cost query, to which it responds with the current value of its best-cost register.

We designed the BFCC to be small enough that the microarchitecture can scale to large graph sizes. Because we precompute the roadmap, we can guarantee that each vertex will have at most four neighbors without affecting the quality of the roadmap. This limitation can be overcome if necessary by logically splitting a vertex with too many neighbors into multiple vertices connected with an edge weight of zero. Similar decisions must be made with maximum path and edge cost in order to fix the size of the registers storing these data. Graph edge costs can be scaled to respect these constraints. If an edge is truly needed with a cost greater than the register size allows, it can be represented as two (or more) edges in serial with costs such that the sum is the desired value.

B. Interconnection Network

To execute the Bellman-Ford algorithm, the vertices in each BFCC need to communicate with their logical neighbors. However, because the microarchitecture must be programmable, this communication must happen over a network so that the sea of physical BFCCs can emulate the behavior of the desired graph topology. The network enables the vertices on each BFCC to abstract away communication issues and behave as if they are actually connected to their logical neighbors, even though they may not be physically adjacent.

We based our network on the low-cost router microarchitecture by Kim [9]. We took this design and applied several optimizations enabled by our application characteristics.

We must smartly map roadmap topologies to physical BFCCs. We use a simulated annealing approach to obtain an acceptable solution to this mapping problem during a preprocessing phase. Annealing found quality mappings from roadmaps to the physical interconnection network in a matter of seconds to minutes, depending on the parameters used.

VI. Results

We evaluate using the pick-and-place task since it is ubiquitous in robotics. We generate roadmaps of various sizes for the six degree-of-freedom Jaco II robotic arm from Kinova. We run experiments on sampled environments consisting of randomly placed and sized obstacles and different source/destination pairs. We consider roadmaps ranging from 4k to 256k edges, but our area and timing numbers focus on a 128 x 128 network implementation solving problems for a 16k-vertex, 32k-edge roadmap. Previous work has shown this size suffices to solve challenging robotics problems [3].

Our results are not unique to the specific application or robot used. Our design can be used in any roadmap-based planning task, including autonomous driving [17], automated inspection [22], and automated machine-tending [4]. We have tested different iterations of our accelerator with four different robots, a range of end-of-arm-tooling, and in a variety of scenarios with consistent results.

Component	**Area (mm^2)**	**Transistor Estimate (M)**
Collision Detection	397	1990
Network Routers	24	122
BFCCs	19	97
Control Nodes	10	48
Total	**450**	**2260**

TABLE I
COMPONENT SIZES FOR 128X128 IMPLEMENTATION.

A. Performance/Area/Power

We used the Synopsys toolchain and the NanGate 15 nm Open Cell Library [13] to synthesize our design and obtain performance, area, and power estimates. The following numbers are for an implementation with 16,384 vertices (128 x 128).

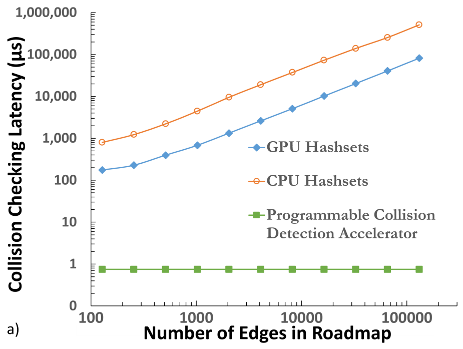
Because the collision detection microarchitecture is completely parallel with respect to the edges in the roadmap, its latency depends solely on the number of obstacle voxels it must process. For the random pick-and-place environments we sampled, there was an average of 750 obstacle voxels, which means collision checking takes an average of 750 cycles, since each voxel requires only a single cycle to process.

For the 16k-vertex graph, the mean time to completion is 360 cycles. Adding the time for the programmable network to detect completion yields a mean of 630 cycles. However, as is common in accelerator design, moving data around takes just as much time as the computation. There is additional overhead of 950 cycles to communicate collisions to the BFCCs and actually extract the path. Including the time to perform collision detection, the total average latency is 2,330 cycles from the time the obstacle voxels arrive to the time a path is ready for output. Synthesis in Synopsys indicates the accelerator could easily be clocked at 1 GHz, so this translates to a 2.3 microsecond latency. This latency is roughly five orders of magnitude faster than conventional sampling-based planners, and two orders of magnitude faster than previous proposed accelerators for the same use case [14], [15].

The breakdown of area on the chip is in Table I. In total, a 16k-vertex design is 450 mm^2 and requires around 2.3 billion transistors. Synopsys estimates the power consumption of the accelerator to be 35 watts.

B. Performance Scaling

Figure 3a shows that because there is dedicated hardware for each edge, the time to perform collision detection is independent of the number of edges in the roadmap. Our 16k-vertex accelerator can handle up to 32,768 edges, but the microarchitecture easily scales to larger roadmaps. This figure also helps deconstruct how much of the benefit derives from the aggressive precomputation compared to the dedicated hardware, by exploiting the same precomputation implemented on a CPU and GPU. The CPU runs a highly-optimized C program running on a 4-core Haswell i7 with 8 GB of RAM, and is instrumented with OpenMP directives to use all hardware



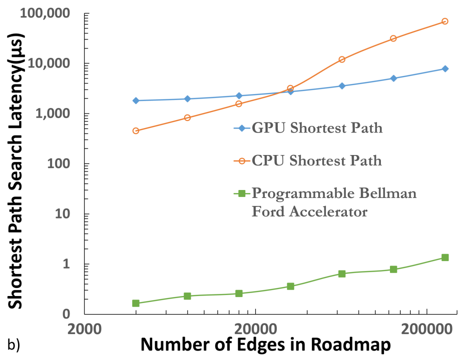


Fig. 3. Performance scaling.

threads. The GPU runs a well-tuned CUDA kernel running on a Tesla K80. Both strategies achieve a roughly 10x speedup compared to a conventional sampling-based planner at runtime, but are still much slower than our accelerator.

Figure 3b shows the scaling of our Bellman-Ford accelerator. Dedicated hardware for each node enables performance to scale linearly with the average number of hops through the graph. The CPU is the same 4-core Haswell i7 running the shortest path implementation in Klampt [8], a well-optimized robotics software package. The GPU uses the nvGraph graph analytics API [16] on a Tesla K80. Because our microarchitecture involves tightly coupling the shortest path with collision detection, whereas the GPU involves communication over PCI-e, no data movement overhead was included for either to be fair (so this figure shows only compute time).

VII. Conclusion

We have presented a programmable architecture for motion planning that is general enough to be applied to any robotic problem. Being able to motion plan in under 3 microseconds makes possible the use of complex decision making algorithms that must invoke motion planning thousands of times as a subroutine [6], [20], [24].

VIII. Acknowledgments

This research was supported in part by DARPA under agreement number D15AP00104. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The content is solely the responsibility of the authors and does not necessarily represent the official views of DARPA. The authors are the founders of Realtime Robotics, Inc., which develops products related to this research.

References

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

[2] 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.

[3] R. Bohlin and L. E. Kavraki, "Path planning using lazy prm," in *IEEE International Conference on Robotics and Automation*, 2000.

[4] R. Bohlin, "Motion planning for industrial robots," *PhD Thesis, Chalmers University of Technology*, 1999.

[5] U. Bondhugula, A. Devulapalli, J. Dinan, J. Fernando, P. Wyckoff, E. Stahlberg, and P. Sadayappan, "Hardware/software integration for fpga-based all-pairs shortest-paths," in *2006 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2006.

[6] C. R. Garrett, T. Lozano-Pérez, and L. P. Kaelbling, "Ffrob: An efficient heuristic for task and motion planning," in *Algorithmic Foundations of Robotics XI*. Springer, 2015, pp. 179–195.

[7] F. Hauer and P. Tsiotras, "Deformable rapidly-exploring random trees," in *Robotics: Science and Systems*, 2017.

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

[9] J. Kim, "Low-cost router microarchitecture for on-chip networks," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009.

[10] J. J. Kuffner and S. M. LaValle, "Rrt-connect: An efficient approach to single-query path planning," in *IEEE International Conference on Robotics and Automation*, 2000.

[11] 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.

[12] S. Lian, Y. Han, X. Chen, Y. Wang, and H. Xiao, "Dadu-p: A scalable accelerator for robot motion planning in a dynamic environment," in *Proc. of the 55th Annual Design Automation Conference*, 2018.

[13] M. Martins, J. M. Matos, R. P. Ribas, A. Reis, G. Schlinker, L. Rech, and J. Michelsen, "Open cell library in 15nm freepdk technology," in *Proc. of the International Symposium on Physical Design*, 2015.

[14] S. Murray, W. Floyd-Jones, Y. Qi, D. Sorin, and G. Konidaris, "The microarchitecture of a real-time robot motion planning accelerator," in *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*, 2016.

[15] ——, "Robot motion planning on a chip," in *Robotics: Science and Systems*, 2016.

[16] Nvidia, *nvGraph API Reference*. http://docs.nvidia.com/cuda/nvgraph/: CUDA Toolkit Documentation, 2017.

[17] B. Paden, M. Cáp, S. Z. Yong, D. S. Yershov, and E. Frazzoli, "A survey of motion planning and control techniques for self-driving urban vehicles," *CoRR*, 2016. [Online]. Available: http://arxiv.org/abs/1604.07446

[18] K. Solovey, O. Salzman, and D. Halperin, "New perspective on sampling-based motion planning via random geometric graphs," in *Robotics: Science and Systems*, 2016.

[19] K. Sridharan, T. K. Priya, and P. R. Kumar, "Hardware architecture for finding shortest paths," in *IEEE Region 10 Conference*, 2009.

[20] S. Srivastava, E. Fang, L. Riano, R. Chitnis, S. Russell, and P. Abbeel, "Combined task and motion planning through an extensible planner-independent interface layer," in *Robotics and Automation (ICRA), 2014 IEEE International Conference on*. IEEE, 2014, pp. 639–646.

[21] Y. Takei, M. Hariyama, and M. Kameyama, "Evaluation of an fpga-based shortest-path-search accelerator," in *The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing*, 2015.

[22] M. Ulrich, G. Lux, L. Jurgensen, and G. Reinhart, "Automated and cycle time optimized path planning for robot-based inspection systems," *6th CIRP Conference on Assembly Technologies and Systems (CATS)*, 2016.

[23] W. Wang, D. Balkcom, and A. Chakrabarti, "A fast online spanner for roadmap construction," *International Journal of Robotics Research*, 2015.

[24] J. Wolfe, B. Marthi, and S. J. Russell, "Combined task and motion planning for mobile manipulation." in *ICAPS*, 2010, pp. 254–258.

[25] A. Yershova and S. M. LaValle, "Improving motion-planning algorithms by efficient nearest-neighbor searching," *IEEE Trans. on Robotics*, 2007.