g-Planner: Real-Time Motion Planning and Global Navigation Using GPUs

Jia Pan and **Christian Lauterbach** and **Dinesh Manocha**
Department of Computer Science, University of North Carolina at Chapel Hill
{panj, cl, dm}@cs.unc.edu
http://gamma.cs.unc.edu/gplanner/

Abstract

We present novel randomized algorithms for solving global motion planning problems that exploit the computational capabilities of many-core GPUs. Our approach uses thread and data parallelism to achieve high performance for all components of sample-based algorithms, including random sampling, nearest neighbor computation, local planning, collision queries and graph search. This approach can efficiently solve both the multi-query and single-query versions of the problem and obtain considerable speedups over prior CPU-based algorithms. We demonstrate the efficiency of our algorithms by applying them to a number of 6DOF planning benchmarks in 3D environments. Overall, this is the first algorithm that can perform real-time motion planning and global navigation using commodity hardware.

Introduction

Motion planning is one of the fundamental problems in algorithmic robotics. The classical formulation of the problem is: given an arbitrary robot, R, and an environment composed of obstacles, compute a continuous collision-free path for R from an initial configuration to the final configuration. It is also known as the *navigation problem* or *piano mover's problem*. Besides robotics, motion planning algorithms are also used in CAD/CAM, computer animation, computer gaming, computational drug-design, manufacturing, medical simulations etc.

There is extensive literature on motion planning and global navigation. At a broad level, they can be classified into local and global approaches. The local approaches, such as those based on artificial potential field methods (Khatib 1986), are quite fast but not guaranteed to find a path. On the other hand, global methods based on criticality analysis or roadmap computation (Schwartz and Sharir 1983; Canny 1988) are guaranteed to find a path. However, the complexity of these exact or complete algorithms increases as an exponential function of the number of degrees-of-freedom (DOF) of the robot and their implementations have been restricted to only low DOF.

Practical methods for global motion planning for high-DOF robots are based on randomized sampling (Kavraki et al. 1996; LaValle and Kuffner 2000). These methods attempt to capture the topology of the free space of the robot by generating random configurations and connect nearby configurations using local planning methods. The resulting algorithms are probabilistically complete and have been successfully used to solve many high-DOF motion planning and navigation problems in different applications. However, they are too slow for interactive applications or dynamic environments.

Main Results: We present a novel parallel algorithm for real-time motion planning of high DOF robots that exploits the computational capability of a \$400 commodity graphics processing unit (GPU). Current GPUs are programmable many-core processors that can support thousands of concurrent threads and we use them for real-time computation of a probabilistic roadmap (PRM) and a lazy planner. We describe efficient parallel strategies for the construction phase that include sample generation, collision detection, connecting nearby samples and local planning. The query phase is also performed in parallel based on graph search. In order to design an efficient single query planner, we use a lazy strategy that defers collision checking and local planning. In order to accelerate the overall performance, we also describe new hierarchy-based collision detection algorithms.

The performance of the algorithm is governed by the topology of the underlying free space as well as the methods used for sample generation and nearest neighbor computation. In practice, our algorithm can generate thousands of samples for robots with 3 or 6 DOFs and compute the roadmap for these samples at close to interactive rates including construction of all hierarchies. It performs no precomputation and is applicable to dynamic scenes, articulated models or non-rigid robots. We highlight its performance on multiple benchmarks on a commodity PC with a NVIDIA GTX 285 GPU and observe a 10-80 times performance improvement over CPU-based implementations.

The rest of the paper is organized as follows. We survey related work on motion planning and GPU-based algorithms in Section 2. Section 3 gives an overview of our approach and we present parallel algorithms for the construction and query phase in Section 4. We highlight our performance on different motion planning benchmarks in Section 5 and compare with prior methods.

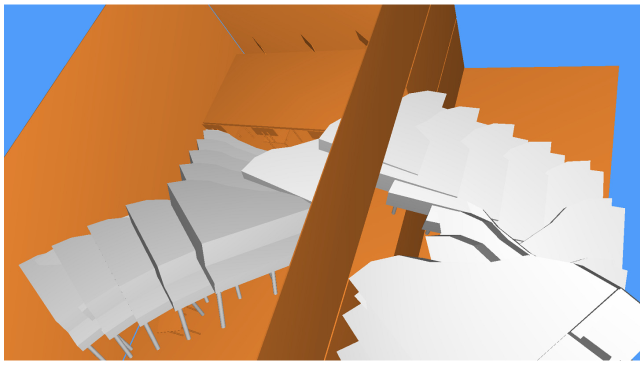

Figure 1: Our GPU-based algorithm, g-Planner, can compute the collision free path for this piano in 111 ms on a PC. This is the first planner to solve these complex problems in real-time.

Related Work

In this section, we briefly survey related work on motion planning and global navigation as well as GPU-based algorithms.

Motion Planning An excellent survey of various motion planning algorithms is given in (LaValle 2006). Many early works are combinatorial approaches based on computational geometry and potential field based approaches. Most of the recent work in terms of global motion planning of high DOF robots has been based on randomized methods such as PRMs (Kavraki et al. 1996) and RRTs (Kuffner and LaValle 2000).

Many applications need real-time motion planning algorithm to interact with dynamic environments. Some approaches design problem-specific heuristics to accelerate the planners, such as decomposition strategy (Brock and Kavraki 2001), corridor maps (Garaerts and Overmars 2007) and replanning methods (Bruce and Veloso 2002). However, these methods may not work well for a general planning scenario. Many parallel algorithms have also been proposed for motion planning. Some of the earlier algorithms utilize properties of the configuration space of the robot (Lozano-Perez and O'Donnell 1991) or use parallel VLSI architectures (Deo, Cavallaro, and Walker 1991). The distributed representation (Barraquand and Latombe 1991) can be easily parallelized (Challou et al. 2003). It is relatively simple to parallelize PRM on shared memory systems (Amato and Dale 1999). An approximate version of the RRT algorithm can also be parallelized (Carpin and Pagello 2001). (Gini 1996) also describe a parallel version for potential field methods. In order to deal with very high dimensional or difficult planning problems, a sampling-based roadmap of trees can be used for parallel computation (Plaku and Kavraki 2005; Plaku et al. 2005). Above approaches are not suitable for mobile applications due to the large volume of the multi-processor system.

GPU-based Algorithms The computational power of many-core GPUs has been used for many geometric and scientific computations (Owens et al. 2007). The rasterization capabilities of a GPU can be used for real-time motion planning of low DOF robots (Lengyel et al. 1990; Hoff et al. 2000; Sud et al. 2006; 2007) or improve the sample generation in narrow passages (Pisula et al. 2000; Foskey et al. 2001). However, rasterization based planning algorithms are complete only to the resolution of image-space. GPUs can also be used to accelerate the roadmap search algorithms (Kider et al. 2010), nearest neighbor search computation (Garcia, Debreuve, and Barlaud 2008) and collision queries (Lauterbach, Mo, and Manocha 2010; Govindaraju et al. 2003; Govindaraju, Lin, and Manocha 2005).

Overview

In this section, we highlight some issues in developing parallel motion planning algorithms for current GPU architectures. One of our goals is use sample-based planners that are relatively easy to parallelize and can be used for single-query and multiple-queries .

Current programmable GPUs are massively parallel architectures with very high theoretical computational power, which makes them attractive for a wide range of complex problems. In general, they have a high number of independent processing cores (around 30 for current GPUs) that are all connected to high bandwidth but also high latency memory. In addition, each GPU core also is a vector processor that can execute an instruction on several (typically 8-16) elements at the same time. Finally, each core also supports hardware multi-threading, i.e. it can quickly switch between several execution threads as needed, e.g. to avoid idle waiting for a memory access. This is particularly important as there are no or just very small caches to speed up memory requests.

As a result, algorithms that run on GPUs need to be designed such that they can exploit this parallelism. In practice, this means that we need to provide hundreds or thousands of parallel tasks for the processor to work on in order to fully utilize the hardware. Communication during computation is relatively slow on GPUs, so it is also important that the algorithm require as little synchronization during the algorithm as necessary.

We choose the PRM algorithm as the underlying method, because it is most suitable to exploit the multiple cores and data parallelism on GPUs. The PRM algorithm is composed of several steps and each step performs similar operations on the input samples or the links joining those samples. Many other efficient CPU-based algorithms have also been developed, including RRT (Kuffner and LaValle 2000) and SBL (Sanchez and Latombe 2001). However, these methods may not be able to exploit the GPU parallelism due for two reasons. Firstly, these methods proceed in an incremental manner in terms of adding new samples to the underlying tree structure. Secondly, the order of adding new samples is critical in terms of how the tree expands.

The PRM algorithm has two phases: roadmap construction and query phase. The roadmap construction phase includes four main steps: 1) generate samples in the configuration space; 2) compute milestones that correspond to the samples in the free space by performing discrete collision queries; 3) for each milestone, find other milestones that are nearest to it; 4) connect nearby milestones using local planning and form a roadmap. The query phase includes two parts: 1) connect initial and goal configurations of query to the roadmap; 2) execute a graph search algorithm on the roadmap and find collision free paths.

Parts of the PRM algorithm such as the collision queries are embarrassingly parallel (Amato and Dale 1999). However, we can use a many-core GPU to significantly enhance the performance of the other components as well. The framework of our PRM algorithm on the GPU is shown in Fig 2. We parallelize each of the 6 steps of PRM algorithm efficiently: First, each thread of a multi-core GPU generates a random configuration of robot and some samples will collide with obstacles. All of the collision-free samples are the milestones and become vertices of the roadmap graph. Then each GPU thread computes the k-nearest neighbors of one milestone and we collect all the neighborhood pairs. Next, each thread checks whether it is possible to connect these adjacent pairs by performing local planning. If there is a collision-free path between that neighborhood pair of milestones, we add the edge to the roadmap. Once the roadmap is built, queries are connected to the roadmap in parallel and we use a parallel graph search algorithm to find paths.

The resulting GPU-based framework is very efficient for a multi-query version of the planning problem. The most expensive step in this computation is the local planning algorithm, therefore we use new collision detection algorithms to improve its performance. In order to accelerate the single query algorithm, we also introduce a solution that uses a lazy strategy and defers collision checking for local planning. In other words, the algorithm connects all the edges corresponding to the nearest neighbors and searches for paths between the initial and final configurations. Finally it performs local planning on the edges that constitute these paths.

Parallelized PRM Motion Planning Algorithm

In this section, we give details of our algorithm and describe how each step is parallelized.

Hierarchy Computation

We construct a bounding volume hierarchy (BVH) for the robot and another one for all the obstacles in the environment to accelerate the collision queries. We use the GPU-based construction algorithm introduced in (Lauterbach et al. 2009), which can construct the hierarchy of *axis-aligned bounding boxes* (AABB) or *oriented bounding boxes* (OBB) for a given triangle data in parallel on the GPU. For collision detection, we use the OBB hierarchy as it provides higher culling efficiency and improved performance on GPU-like architectures. These hierarchies are stored in the GPU memory and we apply appropriate transformations for different configurations.

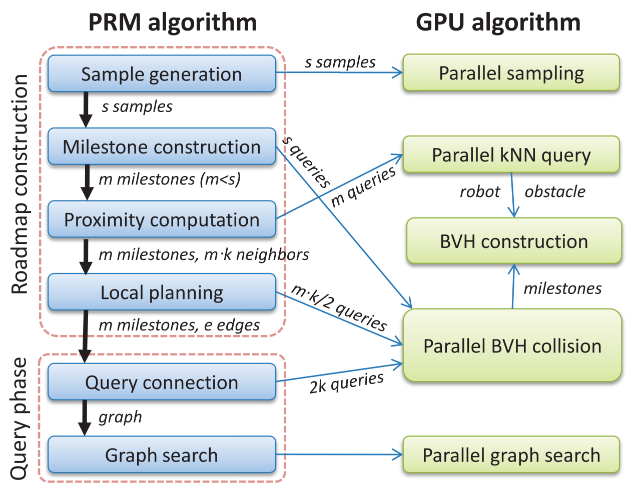

Figure 2: PRM overview and parallel components in our algorithm

Roadmap Construction

The roadmap construction phase tries to capture the connectivity of free configuration space, which is the main computationally intensive part of the PRM algorithm.

Sample Generation We first need to generate random samples within the configuration space. Since samples are independent, we schedule enough parallel threads to utilize the GPU and use the MD5 cryptographic hash function based method (Tzeng and Wei 2008) which in practice provide good randomness without a shared seed.

Milestone Computation For each configuration generated in the previous step, we need to check whether it is a milestone, i.e. lies in the free space and does not collide with the obstacles. We use a hierarchical collision detection approach using bounding volume hierarchies (BVHs) to test for overlap between the obstacles and the robot in the configuration defined by the sample. The collision detection is performed in each thread by using a traversal algorithm in the two BVHs. The traversal algorithm starts with the two BVH root nodes and tests the OBB nodes for overlap in a recursive manner. If two nodes overlap, then all possible pairings of their children should be recursively tested for intersection.

We also compute the actual BVH structure both for robot and obstacles on the GPU by using a parallel hierarchy construction algorithm (Lauterbach et al. 2009). Since the robot's geometric objects move depending on the configuration, its BVH is only valid for the initial configuration. In order to avoid recomputing a BVH for each configuration, we instead transform each node of the robot's BVH with the current configuration sample before performing overlap tests. Thus, only nodes that are actually needed during collision testing must be transformed.

Previous work (Lauterbach, Mo, and Manocha 2010) has used BVH collision on GPUs to parallelize the tests within one query. The approach here is different because we instead parallelize a high number of collision queries, so each thread performs a fully independent collision. In addition, we do not need to find the actual intersection, just whether one exists or not. Therefore, we can also abort the traversal operation as soon as any collision is found and do not have to exhaustively search the hierarchy. In our implementation, each thread performs traversal in a stack-based DFS algorithm. We can store the DFS stack in the shared memory of the GPU which has a higher access speed than global memory on the GPU. Moreover, the DFS also supports early exit from the traversal when the first collision between leaf nodes is computed.

Proximity Computation For each milestone computed, we need to find its k-nearest neighbors (KNN). In general, there are two types of KNN algorithms: exact KNN and approximated KNN which is faster by allowing a small relaxation. Unlike previously proposed GPU solutions using brute force (Garcia, Debreuve, and Barlaud 2008), our proximity algorithm is based on a range query that uses a BVH structure of the points in configuration space. We describe the method for 3-DOF robots and then present its extension to high-DOF robots.

For 3-DOF Euclidean space, we first construct the BVH structure for all the milestones using a parallel algorithm (Lauterbach et al. 2009). For each configuration $\mathbf{q}$, we enclose it within an axis-aligned ϵ box: a box with $\mathbf{q}$ as center and with 2ϵ as edge length. Next, we traverse the BVH tree to find all leaf nodes (i.e. configurations) that within the ϵ box. This reduces to a range-query for $\mathbf{q}$. For non-Euclidean DOFs, we duplicate samples to transform it into a Euclidean space locally. For example, suppose one DOF is rotation angle $\alpha \in [0, 2\pi]$. We add another sample $\alpha^* \in [-\pi, 3\pi]$ with a distance 2π to α. If all 3-DOF are rotations, we need to add another 7 samples for each milestone. Once the range query finishes, we choose the k-nearest one from all the query results. Overall, this gives us the exact nearest neighbors.

For high dimensional spaces we use a decomposition strategy to compute the approximate nearest neighbors. We use 6-DOF as an example. We first decompose each configuration $\mathbf{q}$ into 3-DOF projections $\mathbf{q_1}$, $\mathbf{q_2}$ and obtain two 3-DOF groups. For each of them we build separate BVHs and perform range queries. Suppose we find k_1 neighbors within $\mathbf{q_1}$'s ϵ_1 box and k_2 neighbors within $\mathbf{q_2}$'s ϵ_2 box. We then compute the distances of these candidates to q in 6-DOF space and choose k of them that are nearest to q. For configuration spaces with higher dimensions, we just repeat above process until all dimensions are considered. The final result is the approximated KNNs whose distance to $\mathbf{q}$ is at most $\sqrt{3}\sum_i \epsilon_i$.

To further improve the performance of proximity computation in high dimensional space, we have developed a new k-nearest neighbor algorithm, which uses locality sensitive hashing (LSH) and cuckoo hashing to efficiently compute approximate k-nearest neighbors in parallel on the GPU. For more details, please refer to our recent work (Pan, Lauterbach, and Manocha 2010).

Local Planning Local planning checks whether there is a local path between two milestones, which corresponds to an edge on the roadmap. Many methods are available for local planning. The most common way is to discretize the path between the two milestones into n_i steps and we claim the local path exists when all the intermediate samples are collision-free by performing discrete collision queries (DCD) at those steps. We can also perform local planning by continuous collision detection (CCD), a local RRT algorithm or computing distance bounds (Schwarzer, Saha, and Latombe 2005).

Local planning is the most expensive part of the PRM algorithm. Suppose we have n_m milestones, each milestone has at most n_k nearest neighbors. Then the algorithm performs local planning at most $n_m \cdot n_k$ times. If we use DCD, then we need to perform at most $n_l = n_m \cdot n_k \cdot n_i$ collision queries, which can be very high for a complex benchmark. For multi-query problems, this cost can be amortized over multiple queries as the roadmap is constructed only once. For a single-query problem, computing the whole roadmap is too expensive.

Therefore, in the single-query case, we use a lazy strategy to defer local planning until absolutely needed. Given a query, we compute several different candidate paths in the roadmap graph from initial to final configuration and only check local planning for roadmap edges on the candidate paths. Local planning may conclude that some of these edges are not valid and we delete them from the roadmap. If there exists one candidate path without invalid edges, the algorithm computes a collision-free solution. Otherwise, we compute candidate paths again on the updated roadmap and repeat the above process. This lazy strategy can greatly improves performance for single queries.

Query Phase

The query phase includes two parts: connecting queries to the roadmap and executing graph searches to find paths.

Query Connection Given the initial-goal configurations in one query, we connect them to the roadmap. For both of these configurations, we find the k nearest milestones on the roadmap and add edges between query and milestones that can be connected by local planning. We use the same algorithm from the roadmap construction phase except that k used is $2 - 3$ times larger so as to increase the probability to find a path.

Graph Search The search algorithm tries to find a path on the roadmap connecting initial and goal configurations. Many solutions for this, such as DFS, BFS, A*, R* (Kider et al. 2010). A* and R* can find the shortest path, which is not necessary for the basic motion planning problem. Moreover, A* and R* are not efficient when a lazy strategy is used. As a result, we use DFS or BFS algorithms. For the multi-query case, each GPU thread traverses the roadmap for one query in a DFS way and the final results are collision-free paths. For the single-query case, we exploit all the GPU threads to find the path for one query using a BFS search: for nodes that are of the same steps to the initial node, we can add their unvisited neighbors into the queue in parallel. In other words, different GPU cores traverse different part of graph. The main challenge of this method is that work is generated dynamically as BFS traverse progresses and the computational load on different cores can change significantly. To address the problem of load balancing and work distribution so that the availability of parallelism for all cores is maintained, we use the light-weight load balancing strategy in (Lauterbach, Mo, and Manocha 2010).

When using a lazy strategy, we first run BFS for several iterations and find a set of nodes that are reachable from initial node. Then we run DFS/BFS/A* in each thread with one of these nodes as initial node and find several candidate paths. We use local planning to check whether any one of them are collision-free path. If yes, we return a valid path. Otherwise we remove all invalid edges from the roadmap and repeat the process again.

Implementation and Results

In this section, we present some details of the implementation and highlight the performance of our algorithm on a set of benchmarks. All the timings reported here were taken on a machine using a Intel Core i7 CPU (∼$600) at 3.2GHz CPU and 6GB memory. We implemented our algorithms using CUDA on a NVIDIA GTX 285 GPU (∼$380) with 1GB of video memory.

Our algorithm is designed to work well on any massively processor- and data-parallel architecture by using vector parallelism and low synchronization overhead. In this regard, both NVIDIA and ATI (as well as Intels Larrabee processor) are relatively similar. We used CUDA since it was the most stable development platform at the moment, but look forward to testing in OpenCL and comparing across architectures.

We implement the PRM on the GPU (G-PRM) for multi-query planning problems and its lazy version (GL-PRM) for single-query problems. We compare them with the PRM and RRT algorithms implemented in the OOPSMP library (Plaku, Bekris, and Kavraki 2007) which is a popular library for motion planning algorithms on CPU. The benchmarks used are shown in Fig 3. Our comparisons are designed as follows: For each benchmark, we find a suitable setting where C-PRM finds a solution, then we run G-PRM with comparable number of samples. After that we run GL-PRM with the same setting as G-PRM and C-RRT with the same setting as C-PRM. Of course, the compared PRM algorithms on GPU and CPU are not identical in terms of the final result, e.g. due to underlying random sample generation. Even though these random generators are slightly different, the number of collision-free nodes and collision-free arcs in the computed roadmaps are comparable. Moreover, the final paths for the lazy version are somewhat close. In practice, the total work performed by the non-lazy GPU planner is actually higher than the CPU version.

Table 1 shows the comparison of timings between algorithms. In general G-PRM is about 10 times faster than C-PRM and GL-PRM can provide another 10 times of acceleration for single query problems. C-RRT is usually faster than C-PRM. G-PRM is fast enough for even dynamic scenes. The current C-RRT and C-PRM are both single-core version. However, even a multi-core version of PRM would only improve the timing by 4x at most, because on a 8-core CPU it is hard to scale the hierarchy computations and nearest neighbor computation linearly. Therefor GPU can still provide 1-2 orders of magnitude higher performance than CPU.

	C-PRM	C-RRT	G-PRM	GL-PRM
piano	6.53s	19.44s	1.71s	111.23ms
helicopter	8.20s	20.94s	2.22s	129.33ms
maze3d1	138s	21.18s	14.78s	71.24ms
maze3d2	69.76s	17.4s	14.47s	408.6ms
maze3d3	8.45s	4.3s	1.40s	96.37ms
alpha1.5	65.73s	2.8s	12.86s	1.446s

Table 1: The left two columns highlight the implementations of PRM and RRT algorithm in the OOPSMP. The right two columns highlight the performance of our GPU-based algorithms.

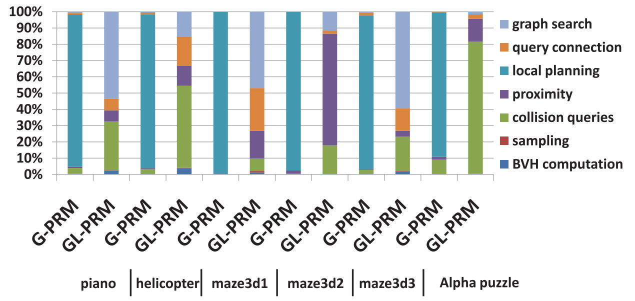

Figure 4: Split-up of timings: the fraction of time spent in parts of algorithm differ between G-PRM and GL-PRM.

Fig 4 shows the timing breakdown between various steps for G-PRM and GL-PRM and the difference between the performance of two algorithms is clear: In G-PRM, local planning is the bottleneck which dominates the timing while in GL-PRM graph search takes longer time because local planning is performed in a lazy or output sensitive manner. In GL-PRM, three components take most timing: milestone construction, proximity computation and graph search, because all of them may perform collision queries heavily. If the environment is cluttered and the model has complex geometry, milestone construction will be slow (*Alpha puzzle* in Fig 4). If the environment is an open space and has many milestones, the proximity computation will be the bottleneck (*mazed3d2* in Fig 4). If the lazy strategy can not guess a correct path, then graph search will be computational intensive due to the large number of collision queries (*maze3d3* in Fig 4). However, in all these environments, GL-RPM outperforms other methods.

We test the scalability of G-PRM and GL-PRM on the *maze3d3* benchmark and the result is shown in Fig 5. It is obvious that GL-PRM is generally faster than G-PRM and both algorithms achieve near-linear scaling on the benchmark. However, notice that the performance of GL-PRM reduces faster than G-PRM. The reason is that when the number of samples increases, proximity computation becomes more and more expensive and dominates the timing when the number of samples is near 1 million.

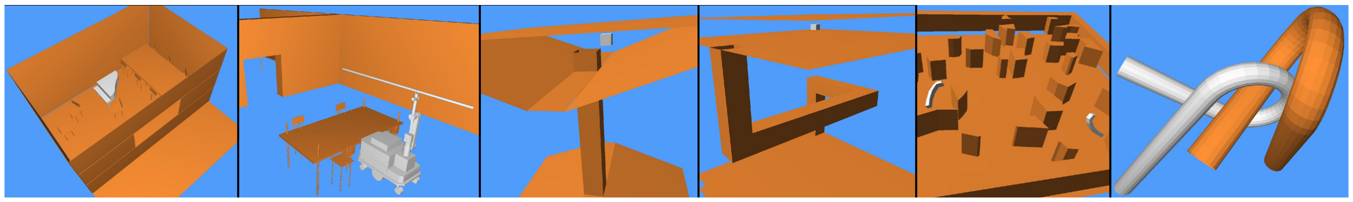

Figure 3: The benchmark scenes used for our algorithms in this order: piano (2484 triangles), helicopter (2484 triangles), maze3d1 (40 triangles), maze3d2 (40 triangles), maze3d3 (970 triangles), alpha puzzle (2016 triangles).

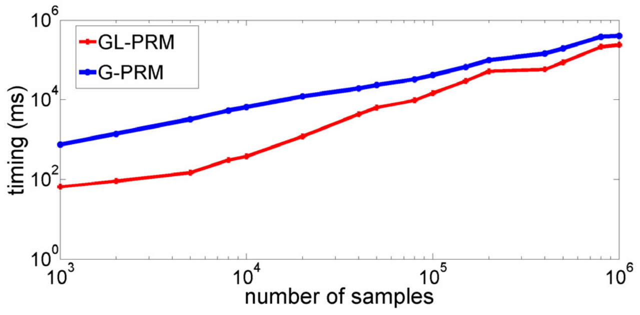

Figure 5: The scalability of G-PRM and GL-PRM algorithms.

Conclusions and Future Work

In this paper, we have introduced a whole motion planning algorithm on GPUs. Our algorithm can exploit all the parallelism within the PRM algorithm including the high-level parallelism provided by the PRM framework and the low-level parallelism within different components of the PRM algorithm, such as collision detection and graph search. As a result, our method provides 1-2 orders of magnitude higher performance over previous CPU-based planners. This makes our work the first to perform real-time motion planning and global navigation in general environments. There are many avenues for future work. We are interested in extending the GPU planning algorithms to high-DOF articulated models. We are also interested in using exact algorithms for local planning. Moreover, we hope to apply our real-time algorithms to dynamic scenarios. Finally, we will test our algorithm in OpenCL/DirectX11 and compare across different architectures.

Acknowledgements

We would like to thank Jean-Paul Laumound for providing some of the models. This research was supported in part by ARO Contract W911NF-04-1-0088, NSF awards 0636208, 0917040 and 0904990, DARPA/RDECOM Contract WR91CRB-08- C-0137, and Intel.

References

Amato, N., and Dale, L. 1999. Probabilistic roadmap methods are embarrassingly parallel. In *Proceedings of IEEE International Conference on Robotics and Automation*, 688–694.

Barraquand, J., and Latombe, J.-C. 1991. Robot motion planning: A distributed representation approach. *International Journal of Robotics Research* 10(6):628–649.

Brock, O., and Kavraki, L. E. 2001. Decomposition-based motion planning: A framework for real-time motion planning in high-dimensional configuration spaces. In *Proceedings of IEEE International Conference on Robotics and Automation*, 1469–1474.

Bruce, J., and Veloso, M. 2002. Real-time randomized path planning for robot navigation. In *IEEE/RSJ International Conference On Intelligent Robots and Systems*, 2383–2388.

Canny, J. 1988. *The Complexity of Robot Motion Planning*. ACM Doctoral Dissertation Award. MIT Press.

Carpin, S., and Pagello, E. 2001. A distributed algorithm for multi-robot motion planning. In *Proceedings of Fourth European Workshop on Advanced Mobile Robots*, 207–214.

Challou, D.; Gini, M.; Kumar, V.; and Karypis, G. 2003. Predicting the performance of randomized parallel search: An application to motion planning. *Journal of Intelligent and Robotics Systems*.

Deo, A. S.; Cavallaro, J. R.; and Walker, I. D. 1991. New real-time robot motion algorithms using parallel vlsi architectures. In *Fifth SIAM Conference on Parallel Processing for Scientific Computing*, 369–375.

Foskey, M.; Garber, M.; Lin, M.; and Manocha, D. 2001. A voronoi-based hybrid planner. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems*, 55–60.

Garaerts, R., and Overmars, M. H. 2007. The corridor map method: Real-time high-quality path planning. In *Proceedings of IEEE International Conference on Robotics and Automation*, 1023–1028.

Garcia, V.; Debreuve, E.; and Barlaud, M. 2008. Fast k nearest neighbor search using GPU. In *Proceedings of the CVPR Workshop on Computer Vision on GPU*, 1–6.

Gini, M. 1996. Parallel search algorithms for robot motion planning. In *Proceedings of IEEE International Conference on Robotics and Automation*, 46–51.

Govindaraju, N.; Redon, S.; Lin, M.; and Manocha, D. 2003. CULLIDE: Interactive collision detection between complex models in large environments using graphics hardware. In *Proceedings of ACM SIGGRAPH/EG Workshop on Graphics Hardware*, 25–32.

Govindaraju, N.; Lin, M.; and Manocha, D. 2005. Quick-

CULLIDE: Efficient inter- and intra- object collision culling using GPUs. In *Proceedings of IEEE Virtual Reality*, 59–66.

Hoff, K.; Culver, T.; Keyser, J.; Lin, M.; and Manocha, D. 2000. Interactive motion planning using hardware accelerated computation of generalized voronoi diagrams. In *Proceedings of IEEE International Conference on Robotics and Automation*, 2931–2937.

Kavraki, L.; Svestka, P.; Latombe, J. C.; and Overmars, M. 1996. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics and Automation* 12(4):566–580.

Khatib, O. 1986. Real-time obstable avoidance for manipulators and mobile robots. *International Journal of Robotics Research* 5(1):90–98.

Kider, J. J.; Henderson, M.; Likhachev, M.; and Safonova, A. 2010. High-dimensional planning on the GPU. In *Proceedings of IEEE International Conference on Robotics and Automation*, to appear.

Kuffner, J., and LaValle, S. 2000. RRT-connect: An efficient approach to single-query path planning. In *Proceedings of IEEE International Conference on Robotics and Automation*, 995–1001.

Lauterbach, C.; Garland, M.; Sengupta, S.; Luebke, D.; and Manocha, D. 2009. Fast bvh construction on GPUs. *Computer Graphics Forum (In Proceedings of Eurographics)* 28(2):375–384.

Lauterbach, C.; Mo, Q.; and Manocha, D. 2010. gproximity: Hierarchical GPU-based operations for collision and distance queries. *Computer Graphics Forum (In Proceedings of Eurographics), to appear*.

LaValle, S. M., and Kuffner, J. J. 2000. Rapidly-exploring random trees: Progress and prospects. *Robotics: The Algorithmic Perspective (In Proceedings of the 4th International Workshop on the Algorithmic Foundations of Robotics)* 293–308.

LaValle, S. M. 2006. *Planning Algorithms*. Cambridge University Press.

Lengyel, J.; Reichert, M.; Donald, B. R.; and Greenberg, D. P. 1990. Real-time robot motion planning using rasterizing computer graphics hardware. *Computer Graphics* 24:327–335.

Lozano-Perez, T., and O'Donnell, P. 1991. Parallel robot motion planning. *Proceedings of IEEE International Conference on Robotics and Automation* 1000–1007.

Owens, J. D.; Luebke, D.; Govindaraju, N.; Harris, M.; Krüger, J.; Lefohn, A. E.; and Purcell, T. 2007. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum* 26(1):80–113.

Pan, J.; Lauterbach, C.; and Manocha, D. 2010. Efficient nearest-neighbor computation for GPU-based motion planning. Technical report, Department of Computer Science, University of North Carolina.

Pisula, C.; Hoff, K.; Lin, M. C.; and Manocha, D. 2000. Randomized path planning for a rigid body based on hardware accelerated voronoi sampling. In *Proceedings of International Workshop on Algorithmic Foundation of Robotics*, 55–60.

Plaku, E., and Kavraki, L. 2005. Distributed sampling-based roadmap of trees for large-scale motion planning. In *Proceedings of IEEE International Conference on Robotics and Automation*, 3879–3884.

Plaku, E.; Bekris, K. E.; and Kavraki, L. E. 2007. Oops for motion planning: An online open-source programming system. In *Proceedings of IEEE International Conference on Robotics and Automation*, 3711–3716.

Plaku, E.; Bekris, K.; Chen, B.; Ladd, A.; and Kavraki, L. 2005. Distributed sampling-based roadmap of trees for large-scale motion planning. *IEEE Transactions on Robotics* 21(4):597–608.

Sanchez, G., and Latombe, J. 2001. A single-query bi-directional probabilistic roadmap planner with lazy collision checking. In *Proceedings of International Symposium on Robotics Research*, 403–417.

Schwartz, J. T., and Sharir, M. 1983. On the piano movers problem II, general techniques for computing topological properties of real algebraic manifolds. *Advances in Applied Mathematics* 4:298–351.

Schwarzer, F.; Saha, M.; and Latombe, J. 2005. Adaptive dynamic collision checking for single and multiple articulated robots in complex environments. *IEEE Transactions on Robotics* 21(3):338–353.

Sud, A.; Govindaraju, N.; Gayle, R.; Kabul, I.; and Manocha, D. 2006. Fast proximity computation among deformable models using discrete voronoi diagrams. In *Proceedings of ACM SIGGRAPH*, 1144–1153.

Sud, A.; Andersen, E.; Curtis, S.; Lin, M.; and Manocha, D. 2007. Real-time path planning for virtual agents in dynamic environments. In *Proceedings of IEEE Virtual Reality*, 91–98.

Tzeng, S., and Wei, L.-Y. 2008. Parallel white noise generation on a GPU via cryptographic hash. In *Proceedings of the Symposium on Interactive 3D Graphics and Games*, 79–87.