The 2010 IEEE/RSJ International Conference on
Intelligent Robots and Systems
October 18-22, 2010, Taipei, Taiwan

Parallel Compact Roadmap Construction of 3D Virtual Environments on the GPU

Avi Bleiweiss

Abstract — **The representation of probabilistic graphical model often encodes a network whose size is unboundedly large. Such networks pose particular challenges to inference algorithms, specifically making the task of robot path queries highly inefficient due to poor locality of memory references. Whereas a more predictable, resolution complete method yields a highly compact graph structure that captures much of the signal in distributing the configuration free space. In this paper we demonstrate an efficient data parallel algorithm for mapping the computationally intensive, Reachability Roadmap method on the GPU. For our implementation on the recently introduced NVIDIA's Fermi architecture, we show roadmap construction time under twenty seconds for a closure resolution of 55x55x55 cells. Moving forward, our system is well positioned to address smooth navigation of robots in a dynamically changing 3D virtual environment.**

I. Introduction

In the broader sense, motion planning concerns itself with enabling multi mobile robots to safely achieve their goals as they maneuver in temporal changing environments. The active research of the past two decades contributed algorithms to effectively address collision-free navigation in 3D configuration free spaces, C_{free}. Besides robotics, the successes of these methods attracted applications to solve challenging problems in a variety of fields, including CAD systems, synthetic environments, video games, traffic control and protein folding.

The roadmap approach to path planning captures C_{free} and creates a graph structure of collision-free configurations [1], [2]. Path planning is then reduced to connecting the robot's start and goal configurations to the roadmap R, and searching R for a minimal, transition cost path. Roadmap methods have usually been designed for one of a single or multiple query planning systems. Many of them use randomization with the accepted tradeoff that they are incomplete but nonetheless converge to a solution with any probability given sufficient running time. Yet, to subscribe efficiently to a parallel GPU motion planner, construction running time and compact memory area of R are essential properties of the algorithm.

The Rapidly-exploring Random Tree (RRT) [3] was introduced as an efficient data structure to quickly search high dimensional spaces with the key idea to bias the exploration towards unexplored portions of the configuration space. RRT construction is simple and fast but is however computed in line with every query. We found RRT architecturally limited for deployment in our system that runs multiple queries concurrently. In contrast, construction and query are disjoint tasks for a Probabilistic Roadmap (PRM) [4], [5], [6]. PRM constructs a graph by sampling collision-free configurations uniformly at random and uses a local planner to connect each node with its k nearest neighbors. One PRM shortfall is its inherent oversampling of C_{free}, especially when narrow passages are present, that often contributes to large footprint graph structures and consequently leads to suboptimal query performance. Moreover, trading off PRM construction running time with incomplete validation, may result in a higher rate of unsuccessful robot path queries. Nonetheless, PRM is highly parallelizable and scalable [7], exploiting decomposition of configuration space [8] and a parallel search formulation [9] to a distributed representation [10]. On the GPU, the reader is referred to the nice overview of previous, motion planning assisted algorithms [11].

Being probabilistically complete, both RRT and PRM lack a quantifiable criterion for when the construction algorithm terminates. This severely impedes our parallel GPU implementation as there is no upper bound guaranteed, and by mainly relying on running time it becomes impractical to consistently predict performance [12]. On the other hand, the Reachability Roadmap (RRM) method [13] ensures a systematic corollary between a discretized C_{free} and the roadmap in defining coverage and maximal connectivity. RRM is thereby space resolution complete in arriving at its solution, but suited at most to two and three dimensional environments. If there exists a valid path in C_{free}, then coverage warrants that the query endpoint configurations can be directly connected to the roadmap, and captured connectivity ensures a matching path to be found in the roadmap. Furthermore, RRM produces considerably tighter memory space for graph data structures compared to PRM, and is hence vital in our design for efficient access of global memory resources, shared amongst thousands of dispatched GPU threads [14].

Our main contribution of this work is a parallel, GPU friendly design of the computationally intractable RRM construction algorithm. We describe a family of extensions that introduce various forms into the workflow to optimize the running time for building a roadmap from non trivial 3D virtual environments. We demonstrate considerable performance gains in running on NVIDIA's recently introduced Fermi GPU with construction time well under

Avi Bleiweiss is a member of the architecture group at NVIDIA Corporation, Santa Clara, CA 95050 USA. Email: ableiweiss@nvidia.com

twenty seconds for closure resolution of 55x55x55 cells. Our navigation software is layered on top of NVIDIA's CUDA architecture [15] that has gained universal acceptance in a wide range of research communities. Next, we highlight the mathematical model of RRM, emphasizing methods that mostly affected the refitting of the algorithm in our GPU implementation.

II. Workflow and Methods

To satisfy coverage and maximal connectivity, the core RRM algorithm computes a small number of guards that view the complete free configuration space and then connects those by placing connectors in overlapping reachability regions, for each pair of guards [16]. Medial axis samples of C_{free} are considered first choice guard candidates because the medial axis is a complete representation for motion planning purposes. In particular, it has the appealing property of large clearance from obstacles [17] that leads to enhanced coverage. Moreover, this sampling framework encompasses exact retraction of a given configuration on to the medial axis for 3D free spaces [18]; and hence facilitates a graceful extension of the reachability region in the event C_{free} is only partially covered after all original medial axis samples have been assigned to guards. Fig. 1 illustrates the process workflow for constructing the RRM graph.

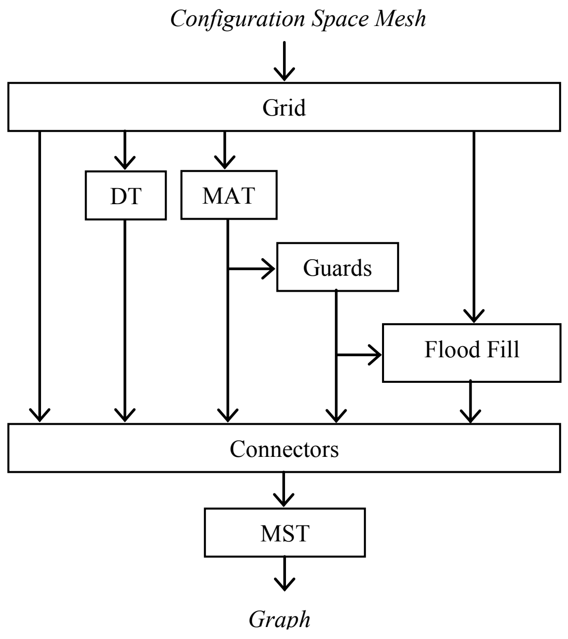


Fig. 1. RRM graph construction workflow: first the configuration space is discretized in to a 3D grid, followed by computing the medial axis transform (MAT) and the distance transform (DT) of C_{free}. Then, guards are applied to a visibility based flood fill algorithm and finally, connectors are placed and further optimized by running a minimal spanning tree (MST) algorithm.

First, the 3D configuration space is discretized and a 3D binary grid is generated, marking a cell as false once its configuration overlaps an obstacle. The serial algorithm for grid creation is of $O(n)$ complexity with n the number of triangular faces of the 3D mesh representation. Then, we compute the medial axis transform (MAT) of C_{free}, leveraging the interchangeability of MAT and the chessboard distance transform (CDT) [19]. The governing equation of the 3D MAT operator follows:

$$MAT(i,j,k) = \min\{ \max(|i-x|, |j-y|, |k-z|) \},$$

where $i \leq x \leq N,\ j \leq y \leq N,\ k \leq z \leq N$, and N being the closure dimension for each axis. A single threaded MAT algorithm for a grid resolution of n is of $O(n^3)$ running time, concerning each of its steps and includes CDT computation of the lower-right-front sub volume of each cell, evaluating the qualifier grid $T(i,j,k)$, and the final resolution of valid medial axis samples.

To further classify guard selection by their clearance to obstacles we compute the distance transform (DT) of C_{free}. We use the highly efficient framework for generalizing DT to arbitrary sampled functions [20], which reduces 3D DT to the composition of one dimensional transforms for each of the grid axes, applied to an oriented scan line of the grid slices, columns and rows. The transform overarching equation under the squared Euclidian distance is given by:

$$DT_f(p) = \min\left((p-q)^2 + f(q)\right),$$

where q is a cell of the grid. Inherently, separated and order independent, multiple 1D transforms simplify the DT implementation considerably. Note the sequential version of DT runs at $O(n^3)$ time with n the grid dimension. Obstacle clearance is then determined by correlating cells of the MAT and DT resulting grids, with medial axis samples additionally sorted in a decreasing distance order. By granting priority to guards of a larger obstacle distance, the following coverage algorithm is markedly optimized.

```
S = stack of cells
g = guard cell
c = current cell
C = 3D coverage data structure
S ← g
while S not empty do
    c ← pop S
    if g not visible from c continue
    C[c] ← C[c] ∪ g
    // extend to six adjacent cells
    foreach adjacent neighbor $n_i$ of c do
        if $n_i \in C_{free}$ && n not covered do
            S ← n
```

Fig. 2. Pseudo code of the non recursive, single cell, 3D Flood Fill algorithm flow, using a private stack.

The coverage of C_{free} is resolved iteratively as each guard is added to the list of graph nodes. The underlying method used is an obstacle aware, 3D Flood Fill algorithm [21] with its pseudo code for evaluating a single cell and using a private stack, shown in Fig. 2. We start by pushing the guard cell g on to the stack S. While the stack is not empty, the stack is popped and if the line between the current cell and the guard is unobstructed, we further examine the six adjacent neighbors n_i of the current cell. Neighbors are pushed on to the stack only if they meet both conditions namely, being in C_{free} and uncovered by the guard. As the algorithm progresses we fill a 3D coverage data structure, a grid that stores in each of its cells a set of guard indices. The implicit RRM optimization for adding a guard to the node list, unless it is visible by any of the previously added guards, renders the coverage algorithm as inherently serial and poses a challenge to a GPU parallel implementation we discuss later.

```
1  $G$ = list of guard indices
2  $C$ = 3D coverage data structure
3  $N$ = list of connectors
4  foreach pair $(g_i, g_j) \in G : i < j$ do
5      foreach cell $c \in C$ do
6          if $g_i \nexists c_{set}$ or $g_j \nexists c_{set}$ continue
7          if $(g_i, g_j) \in N$ do
8              resolve connector
9          else do
10             insert connector
```

Fig. 3. Connector insertion pseudo code. Line 8 grants priority to the connector that is either on the medial axis or to the one that has the largest distance from an obstacle.

The next computational step of RRM is connector insertion (Fig. 3) in grid cells covered by multiple guards. A connector is a data structure composed of an edge linking a pair of guards (g_i, g_j), and a 3D grid cell coordinate, c. If a connector edge already exists in the connector list then priority is given to the one on the medial axis or to the one with the greatest distance from an obstacle. The non parallel algorithm runs at $O(n^2)$ with n the number of guard nodes.

Finally, for each connector in the list we add a node v_c and a pair of edges (g_i, v_c) and (v_c, g_j) to the graph data structure. The graph is then pruned by using Kruskal's minimum spanning tree (MST) algorithm [22]; removing edges of weights with little effect on maximal connectivity.

III. Implementation

In this section we describe our RRM implementation that uses NVIDIA's CUDA programming environment and targets the recently announced NVIDIA's Fermi architecture [23]. Fermi's true cache hierarchy is essential to efficient memory access of the fairly large intermediate, 3D data structures generated throughout the RRM work flow. Moreover, with 512 CUDA cores the computation power increases almost fourfold over prior GPU generation and makes Fermi well suited to match the compute intensive RRM challenge. However, not all RRM stages translate naturally to the highly parallel environment of the GPU and some require substantial algorithm refitting, primarily to avoid costly, global synchronization. We launch one kernel for every RRM step, each of a different level of parallelism, to achieve high occupancy. On the other hand, RRM imposes dependent global memory allocations and incurs a barrier wait overhead, per stage. Upon completion, kernel output data structures are persistently retained in global memory throughout the construction process, providing implicit low overhead, inter kernel communication paths.

We now look more closely at GPU design tradeoffs to specifically address RRM concurrency challenges and to achieve linear performance scalability as a function of increased configuration space, closure resolution.

A. *MAT and DT*

To the best of our knowledge we are the first to introduce a parallel solution to 3D MAT and DT algorithms, leveraging GPU computing. Much like the parallel block based approach that performs on a relatively expensive system [24], our goal for the 3D MAT algorithm [19] is $O(1)$ time with a small constant factor; barring, of course, not to exceed in flight, Fermi's 24576 thread limit that amounts to running a closure of up to 29x29x29 cells, in a single launch. We achieve this goal by dispatching n^3 GPU threads for a 3D grid of dimension n, each computing the maximum sized cube adjacent to a cell. These threads operate independently by reading in the 3D binary grid and writing into neighboring locations of the medial axis data structure in global memory. Computing the qualifier 3D grid T, commences after all CDT threads terminate, followed synchronously by the final transform resolution step.

For 3D DT [20] that is less compute resourceful compared to MAT, we only launch n^2 GPU threads and reach an $O(n)$ time. Every thread runs the algorithm in three passes, first computing DT of a slice directed scan line in the binary grid, followed by column and row casts of the DT grid. Passes are though dependent and require thread synchronization for each, before moving on to the next pass.

B. *Flood Fill*

The intuitive parallelism for finding coverage is at the level of sorted medial axis guards, G. However, we have to give up guard pruning optimization of the original RRM method that serializes the algorithm and requires considerable synchronization cost. Instead, we check at runtime whether a guard $g_i \in G$ is visible from any other guards every time we enter the top of the flood fill loop (line 6 of Fig. 2). If it does, the guard terminates instantly, sets its

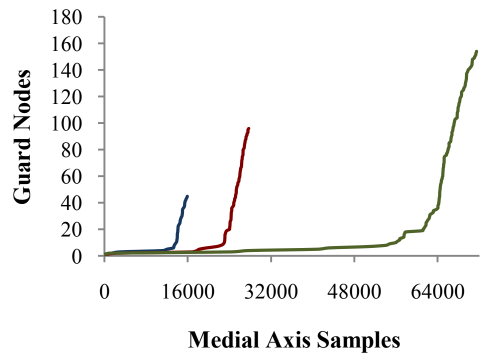


Fig. 4. Guard node selection as processing of sorted medial axis samples commences, shown for three experimental configuration spaces. Guards closer to obstacles appear to contribute more to final guard nodes.

skipped status to true and is removed from further graph node assignment. This implies that on the GPU we assign every non pruned guard g_i to a hardware thread t_i, and all guards execute the 3D flood fill kernel simultaneously and independently. When all threads complete, a single thread process traverses the sets of the resulting 3D coverage data and correlates cells with the 3D binary grid to determine coverage percentage of C_{free}.

Our experiments explore tens of thousands of medial axis samples that by the end of flood fill yield a compact few tens to a little over a hundred valid guard nodes (Fig. 4). The GPU approach for resolving configuration space coverage achieves a high occupancy rate and runs at $O(1)$ time, assuming initial guards are within the hardware thread concurrency range. However, this comes at the cost of an increased global memory footprint. The most expensive resource in the coverage process is the private, thread local stack. To this end we have evaluated both a single cell and a scan line (along the y axis) based flood fill with the latter requiring much less stack space. But the scan line method speedup is unfortunately not commensurate and we therefore use the single cell algorithm with number of stack entries topping several thousands. To avoid overrunning memory space we allow for a maximum of 4096 stack entries and gracefully break down the flood fill process into a succession of smaller guard count, 1024, CUDA launches. Note that in spite the thread aligned layout of the stack we maintain, reads and writes are fairly uncoalesced.

C. Connectors

Like flood fill, connector insertion uses a parallel model that factorizes well. The governing connectivity threads are the surviving guard nodes G from the previous coverage process. More precisely, we iterate every possible pair of guard nodes (g_i, g_j) with $i < j$ and having the kernel traverse for each the entire 3D coverage data structure to resolve a single private connector, per thread. The test for finding a guard index in a set of size k of a coverage cell is $O(\log k)$. In total, we run $n(n-1)/2$ threads for commencing connectivity, with n the number of guard nodes. While launching n^2 threads and let the left triangle of the thread grid return immediately is most intuitive, it turned out to be wasteful. Rather, we launch the exact subscribed threads and have the GPU encode a linear thread id into a pair of guard indices (i, j) in $O(\log n)$ time. As shown in Fig. 2, all previously generated RRM 3D data structures, including the binary, MAT, DT and coverage grids, are read-only by the connector insertion kernel and exhibit good cell locality. For the output, every thread contributes a single, fully resolved connector. Connector data structure reads are perfectly coalesced, but conditional writes do not.

As a final step, all connectors are read back to the CPU and initial, non optimized graph nodes and edges are produced. We then perform MST in a single thread process after which we emit our final graph data structure that is passed onto a GPU search engine [25], [26].

IV. Results

For RRM construction, our goal is to validate anticipated Fermi's architecture performance gains compared to previous generation GPUs. Ideally, once available, we could benefit from an OOPSMP [27] RRM module, and compare our results to an optimized, reference CPU implementation. As it stands, we ended up developing our own multi threaded CPU version, written in C++. Similarly, while some proposed standard motion planning benchmarks [28] are more realistic, they seem limited to only a couple thousands of obstacle triangles. Consequently, to properly validate performance of high occupancy RRM on the GPU, we chose three state-of-the-art video game configuration spaces that explore many tens of thousands of faces. Times are reported using CUDA 3.0 under Windows 32 bit Vista for both NVIDIA's Fermi's [23] family member GTX480 [29] and for the previous GPU series member GTX285 [30]. Timing figures include allocation, processing and copy from host-to-device and device-to-host.

In order to evaluate our RRM construction in practice we have implemented a DirectX [31] geometrical application that converts a mesh OBJ format into our RRM library representation. Table I shows the properties of the three meshes used, their 3D closure resolution, $\sqrt[3]{faces}$, and runtime thread count for MAT (n^3) and DT (n^2).

TABLE I
CONFIGURATION SPACE MESH PROPERTIES

Configuration Space		Closure Resolution	GPU Threads	
Vertices	Faces		MAT	DT
82800	34750	33x33x33	35397	1089
161463	64451	40x40x40	64000	1600
347223	170173	55x55x55	166375	3025

Table II depicts some of our statistics for each of the RRM stages. Including are the number of initial guards that amount to the medial axis samples, coverage percentage after commencing 3D flood fill, number of initial connectors, and the final node and edge count for the generated roadmap. The weight parameter in the right column is the maximum Euclidian edge length in the roadmap. Fig. 8 and 9 show a graphical representation of two of the original meshes, their MAT and DT grids, and the superimposed generated graph data.

TABLE II
RRM STAGE OUTPUT STATISTICS

Guards	Coverage	Connectors	Nodes	Edges	Weight
15956	99.79%	404	114	109	128.72
27747	99.74%	1373	287	286	352.42
71599	99.89%	3154	782	764	406.08

Our coverage percentage shown in Table II is fairly high but still shy from meeting the maximum reachability criterion the original RRM algorithm specifies. We believe this is improvable once we incorporate exact medial axis retraction [18] functionality in our implementation. This is further discussed later.

On the computation side, Fig. 5 shows a typical break down of the running time slices for RRM tasks. As expected, the flood fill is most expensive and dominates the construction process, and second in line is connector insertion. Fig. 6 compares NVIDIA’s GTX480 to GTX285 depicting both running time and speedup plots.

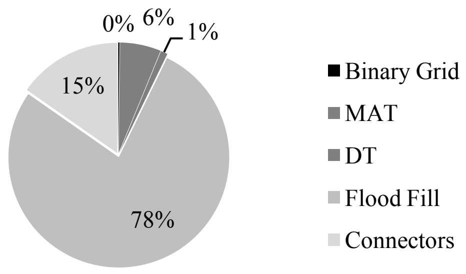


Fig. 5. A typical break down of running time percentage for RRM tasks with flood fill dominating (78%) and connectors comes second (15%).

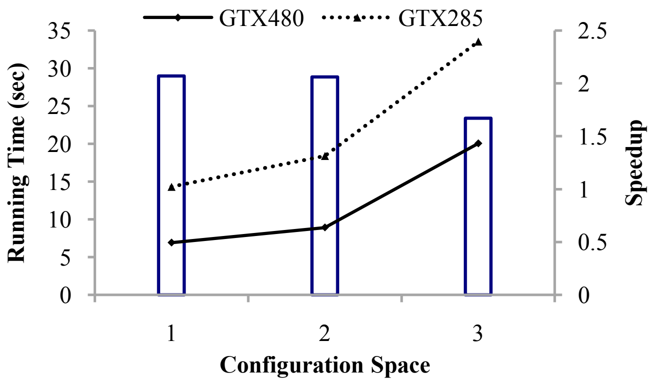


Fig. 6. GPU running time in seconds and speedup (vertical bars) for RRM construction; NVIDIA’s GTX480 vs. GTX285 with stream and memory clock speeds of 1446/1796 MHz and 1476/1242 MHz, respectively.

Finally, a measure of interest for us is Fermi’s relative throughput as a function of increased thread count, shown for the MAT kernel in Fig. 7:

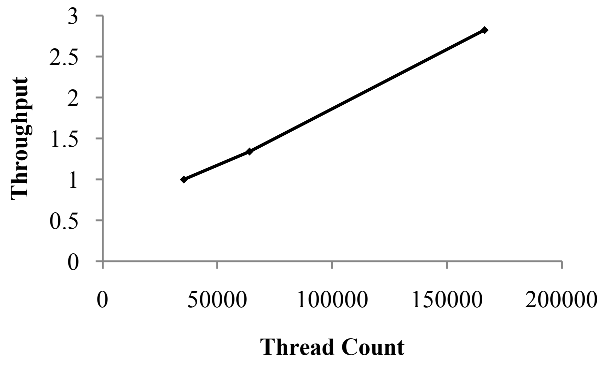


Fig. 7. GTX480 relative, normalized throughput for performing MAT as a function of thread count.

V. DISCUSSION

The quantitative GPU runtime data for RRM construction of variable mesh complexity sets the context for the discussion.

A. *Performance*

Our first goal for creating highly compact roadmap data structures is reaffirmed by the RRM data of Table II. On average, for our experiments, the ratio between the initially seeded, medial axis samples to the final graph nodes is at about 100:1. Consequently, graph data structure tops 1000 nodes, leading to an efficient, multi threaded search on the GPU [25], [26]. Secondly, for configuration space closures of dimensions up to 55x55x55, we demonstrate RRM construction time of less than 20 seconds, when running on Fermi. Fermi’s GTX480 shows a speedup of up to 2X compared to GTX285, with about 90% of performance gains attributed to its new architecture. Moreover, Fermi’s throughput remains almost linear (Fig. 7), while dispatching hundreds of thousands of threads, well beyond the in flight hardware parallel capacity.

B. *Parallel Programming and Debugging*

Porting our RRM, CPU implementation on to CUDA-C++ was a fairly smooth process, coding wise. More importantly, developing up front a modular task manager has proven highly useful, increasing our productivity markedly. The task manager launches thread grids in a hierarchical form (DAG) and dissects a GPU compute scope into smaller workloads due to either limited memory resources or for introducing explicit synchronization. Recently introduced, NVIDIA’s Parallel Nsight [32] appreciably simplifies the debugging of parallel software developed to leverage GPU computing.

C. *Limitations*

While our work establishes a remarkable potential for parallel RRM construction on the GPU, we believe that there is more work to be done in this area.

Stack space Our implementation could benefit from a tighter stack space, allocated per thread in the flood fill stage, to minimize multi kernel launches. For the guard count listed in Table II we perform 16, 28, and 72 flood fill launches, respectively, each evaluating 1024 guards concurrently. But in spite the initial ramping up overhead for a launch, by being asynchronous with no explicit synchronization, we afterwards mitigate construction cost in leveraging Fermi’s out of order thread block activation, on a stream multiprocessor.

Medial axis retraction To reach the goal of complete coverage of C_{free}, as specified by RRM, we need to expand on our current GPU implementation and retract uncovered cells from the DT grid on to the medial axis. This have to include the sorting of n^3 DT cells in a decreasing nearest contact clearance. While suboptimal, our proposed method for concurrent retraction on the GPU would bucket DT cells for a launch followed by synchronization and check for maximum coverage before the next bucket is dispatched. This seems a reasonable tradeoff, since empirically there are fewer contributing DT cells to coverage than MAT cells.

VI. Summary and Future work

We have demonstrated a parallel, GPU friendly construction of compact roadmaps for 3D path planning problems. Our design builds on modern ideas [33] for navigating safely mobile robots. While running time rates, for fairly complex configuration space scenarios, are not yet interactive, we believe NVIDIA's Fermi architecture has the capacity to lead us into smooth motion in highly dynamic 3D virtual environments [34]. One of the more interesting outcomes of our work is the design tradeoffs between RRM construction time, access patterns and storage requirements. Our implementation tries to balance these three metrics, but applications that, for example, are less limited by storage space, may choose to make different design decisions. Some of the areas we consider for future work include:

- Exact, concurrent medial axis retraction so we can reach maximum coverage.
- Adding useful cycles to avoid long detours around obstacles and provisions for shorter path extractions.
- Retraction of graph edges on to the medial axis representation of C_{free} to ensure high clearance paths and less query computation cost.

Acknowledgment

We would like to thank the anonymous reviewers for their constructive and helpful comments.

References

[1] J. C. Latombe, *Robot Motion Planning*. Kluwer, 1991.

[2] S. M. LaValle, *Planning Algorithms*. Cambridge University Press, http://msl.cs.uiuc.edu/planning/, 2005.

[3] J. Kuffner and S. Lavalle, "RRT-Connect: An Efficient Approach to Single-Query Path Planning," *in Proceedings of IEEE International Conference on Robotics and Automation*, pp. 995–1001, Apr. 2000.

[4] T. Simeon, J. P. Laumound, and C. Nissoux, "Visibility Based Probabilistic Roadmaps for Motion Planning," *in International Journal of Advanced Robotics*, vol. 14, no. 6, pp. 477–493, Apr. 2000.

[5] 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.

[6] G. Song, S. L. Thomas, and N. M. Amato, "A General Framework for PRM Motion Planning," *in Proceedings of IEEE International Conference on Robotics and Automation*, pp. 4445–4450, Sep. 2003.

[7] N. M. Amato and L. K. Dale, "Probabilistic Roadmap Methods are Embarrassingly Parallel," *in Proceedings of IEEE International Conference on Robotics and Automation*, pp. 688–694, May 1999.

[8] T. Lozano-Perez and P. O'Donnell, "Parallel Robot Motion Planning," *in Proceedings of IEEE International Conference on Robotics and Automation*, pp. 1000–1007, Apr. 1991.

[9] D. Challou, M. Gini, and V. Kumar, "Parallel Search Algorithms for Robot Motion Planning," *in Proceedings of IEEE International Conference on Robotics and Automation*, pp. 46–51, May 1993.

[10] J. Barraquand and J. C. Latomb, "Robot Motion Planning: A Distributed Representation Approach," *International Journal of Robotics Research*, vol. 10, no. 6, pp. 628–649, Dec. 1991.

[11] J. Pan, C. Lauterbach, and D. Manocha, "g-Planner: Real-Time Motion Planning and Global Navigation using GPUs," *AAAI Conference on Artificial Intelligence,* Jul. 2010.

[12] D. Challou, M. Gini, V. Kumar, and G. Karypis, "Predicting the Performance of Randomized Parallel Search: an Application to Motion Planning," *Journal of Intelligent and Robotics Systems*, vol. 38, no. 1, pp. 31–53, Sep. 2003.

[13] R. Geraerts and M. H. Overmars, "Creating Small Roadmaps for Solving Motion Planning Problems," *in Proceedings of IEEE International Conference on Methods and Models in Automation and Robotics*, pp. 531–536, Aug. 2005.

[14] A. Bleiweiss, "Scalable Multi Agent Simulation on the GPU," *in Proceedings of the IASTED 14th Conference on Robotics and Applications*, pp. 143–151, Nov. 2009.

[15] Nvidia, 2007. CUDA Programming Guide. http://www.nvidia.com/object/cuda_home.html

[16] R. Geraerts and M. H. Overmars, "Reachability Analysis of Sampling Based Planners," *in Proceedings of IEEE International Conference on Robotics and Automation*, pp. 406–412, Apr. 2005.

[17] H. Choset and J. Burdick, "Sensor-Based Exploration: The Hierarchical Generalized Voronoi Graph," *International Journal of Robotics Research*, vol. 19, no. 2, pp. 96–125, 2000.

[18] J. Lien, S. L. Thomas, and N. M. Amato, "A General Framework for Sampling on the Medial Axis of the Free Space," *in Proceedings of IEEE International Conference on Robotics and Automation,* pp. 4439–4444, Sep. 2003.

[19] Y. Lee and S. Horng, "The Chessboard Distance Transform and Medial Axis Transform are Interchangeable," *in Proceedings of the 10th International Parallel Processing Symposium*, pp. 424–428, Apr. 1996.

[20] P. F. Felzenszwalb and D. P. Huttenlocher, "Distance Transforms of Sampled Functions," Cornell Computing and Information Science TR2004-1963, 2004.

[21] M. Kalisiak and M. Van-de-Panne, "RRT-Blossom: RRT with a Local Flood-Fill Behavior," *in Proceedings of IEEE International Conference on Robotics and Automation,* pp 1237–1242, May 2006.

[22] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. The MIT Press/ McGraw-Hill Book Company, Second Edition, 2001.

[23] Nvidia, 2009. Fermi Architecture http://www.nvidia.com/object/fermi_architecture.html

[24] Y-R. Wang, "A Novel *O*(1) Time Algorithm for 3D Block-Based Medial Axis Transform by Peeling Corner Shells," *Parallel Computing*, vol. 35, no. 2, pp. 72–82, 2009.

[25] A. Bleiweiss, "GPU Accelerated Pathfinding," *in Proceedings of ACM Siggraph/Eurographics Conference on Graphics Hardware*, pp. 139–147, Jun. 2008.

[26] J. J. Kider, M. Henderson, M. Likhachev, and A. Safonova, "High Dimensional Planning on the GPU," *in Proceedings of IEEE International Conference on Robotics and Automation,* to appear, May 2010.

[27] E. Plaku, K. E. Bekris, and L. E. Kavraky, "OOPS for Motion Planning: An Online, Open Source Programming System," *in Proceedings of IEEE International Conference on Robotics and Automation*, pp. 3711–3716, Apr. 2007.

[28] Motion Planning Benchmarks, Algorithms & Applications Group, Texas A&M University, http://parasol-www.cs.tamu.edu/dsmft/benchmarks/mp/

[29] Nvidia 2010. Geforce 400 series: http://www.nvidia.com/object/product_geforce_gtx_480_us.html

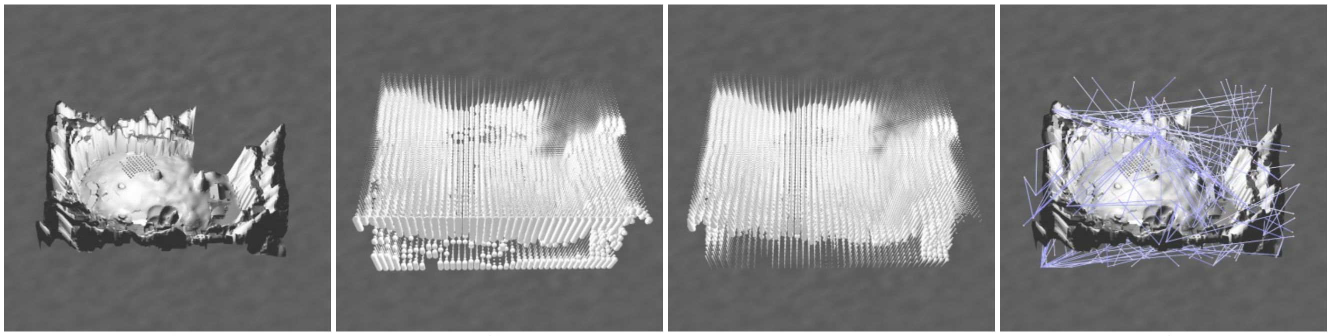
[30] Nvidia, 2008. Geforce 200 series: http://www.nvidia.com/object/geforce_gtx_280.html

[31] Microsoft 2007. DirectX Developer Center. http://msdn.microsoft.com/en-us/directx/default.aspx

[32] Nvidia 2010. Parallel Nsight: http://developer.nvidia.com/object/nsight.html

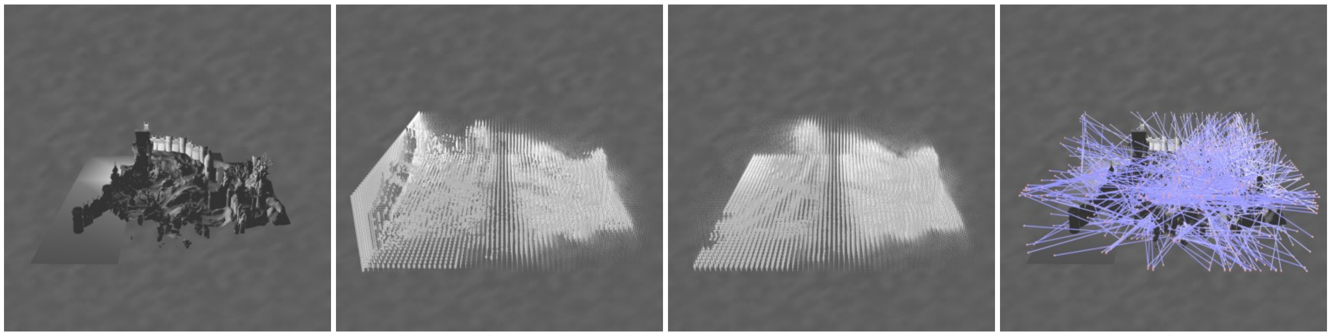
[33] R. Geraerts and M. H. Overmars, "Creating High-quality Roadmaps for Motion Planning in Virtual Environments," *in Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 4355–4361, Oct. 2006.

[34] C. Clark, S. M. Rock, and J. Latombe, "Motion Planning for Multiple Mobile Robot Systems using Dynamic Networks," *in Proceedings of IEEE International Conference on Robotics and Automation*, pp. 4222–4227, Sep. 2003.



(a) (b) (c) (d)

Fig. 8. Configuration space mesh (a) of 161463 vertices and 64451 faces, MAT (b) and DT (c) depicted with spheres centered in grid cells, each of a radius inverse to the adjacent front-lower-right cube height or the obstacle distance, respectively; and superimposed graph nodes and edges (d).



(a) (b) (c) (d)

Fig. 9. Configuration space mesh (a) of 347223 vertices and 170173 faces, MAT (b) and DT (c) depicted with spheres centered in grid cells, each of a radius inverse to the adjacent front-lower-right cube height or the obstacle distance, respectively; and superimposed graph nodes and edges (d).