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

Efficient Nearest-Neighbor Computation for GPU-based Motion Planning

Jia Pan and Christian Lauterbach and Dinesh Manocha
Department of Computer Science, UNC Chapel Hill
http://gamma.cs.unc.edu/gplanner

Abstract— **We present a novel k-nearest neighbor search algorithm (KNNS) for proximity computation in motion planning algorithm that exploits the computational capabilities of many-core GPUs. Our approach uses locality sensitive hashing and cuckoo hashing to construct an efficient KNNS algorithm that has linear space and time complexity and exploits the multiple cores and data parallelism effectively. In practice, we see magnitude improvement in speed and scalability over prior GPU-based KNNS algorithm. On some benchmarks, our KNNS algorithm improves the performance of overall planner by $20-40$ times for CPU-based planner and up to 2 times for GPU-based planner.**

I. Introduction

Sampling-based motion planning algorithms are widely used in robotics and related areas to compute collision-free motion for the robots. These methods use randomized technique to generate samples in the configuration space and connect nearby samples using local planning methods. A key component in these algorithms is the computation of the k-nearest neighbors (KNN) of each sample, which are defined based on a given distance metric.

The problem of nearest neighbor computation has been well studied in computational geometry, databases and image-processing besides robotics. Some of the well known algorithms are based on Kd-trees, bounding volume hierarchies (BVHs), R-tree, X-trees, M-trees, VP-trees, GNAT, iDistance, etc [18].

In this paper, we address the problem of k-nearest neighbor search (KNNS) for realtime sampling-based planning. Our work is based on recent developments in motion planning that use the computational capabilities of commodity many-core graphics processing units (GPUs) for real-time motion planning [15]. The resulting planner performs all the steps of sampling-based planning including sample generation, collision checking, nearest neighbor computation, local planning and graph search using parallel algorithms. In practice, the overall planning can be one or two orders of magnitude faster than current CPU-based planners.

We present a novel algorithm for GPU-based nearest neighbors computation. Our formulation is based on *locality sensitive hashing* (LSH) and *cuckoo hashing* techniques, which can compute approximate k-nearest neighbors in higher dimensions. We present parallel algorithms for LSH computation as well KNN computation that exploit the high number of cores and data parallelism of the GPUs. Our approach can perform the computation using Euclidean as well as non-Euclidean distance metrics. Furthermore, it is memory efficient and can handle millions of sample points within 1GB memory of a commodity GPU. Our preliminary results indicate that the novel KNNS algorithm can be faster by one or two orders of magnitude than prior GPU-based KNNS algorithms. This also improves the performance of the GPU-based planner by $25-100\%$ as compared to gPlanner [15]. We highlight its performance on different benchmarks.

The rest of the paper is organized as follows. We survey related work in Section II. Section III gives an overview of our motion planning framework. We describe the novel GPU-based parallel algorithm for KNN computation in Section IV. We highlight its performance on motion planning benchmarks in Section V.

II. Related Work

In this section, we briefly survey the related work on motion planning, nearest neighbor computation, locality sensitive hashing as well as GPU-based algorithms.

A. Motion Planning

An excellent survey of various motion planning algorithms is given in [13]. Most of the recent work is based on randomized or sampling-based methods such as PRMs and RRTs.

Many task planning applications need a real-time motion planning algorithm for dynamic environments

Jia Pan, Christian Lauterbach and Dinesh Manocha are with the Department of Computer Science, University of North Carolina at Chapel Hill, NC 27599 USA {panj, cl,dm}@cs.unc.edu. This work 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.

with moving objects. Some approaches use application-specific heuristics to accelerate the planners. Many parallel algorithms have also been proposed for real-time motion planning that can be parallelized on multi-CPU systems, such as [8], [3] etc.

B. k-Nearest Neighbor Search

k-nearest neighbor search (KNNS), also known as proximity computation, is an optimization problem in the metric space: given a set S of points in a d-dimensional space $\mathcal{M}$ with metric $\|\cdot\|_p$ and a query point $\mathbf{q} \in \mathcal{M}$, find the closest k points in S to $\mathbf{q}$.

Various solutions to KNNS computation have been proposed. Some of them are based on spatial data structures, e.g. Kd-tree, and compute the exact k-nearest neighbors efficiently when the dimension d is low (e.g., up to 10). However, these methods suffer from either space or query time complexity that tends to grow as an exponential function of d. For large enough d, these methods may only provide little improvement over the brute-force search algorithm that compares a query to each point from S. Such phenomenon is called the *curse of dimensionality*. Some other approaches tend to overcome the difficulty by using approximate methods for KNNS computation [4], [14], [1]. In these formulations, the algorithm is allowed to return a point whose distance from the query point is at most $1+\epsilon$ times the distance from the query to its nearest points; $\epsilon > 1$ is called the approximation factor. The appeal of these approaches is that in most cases an approximate nearest neighbor can be almost as good as the exact one.

C. Locality Sensitive Hashing

Locality-sensitive hashing (LSH) is another popular algorithm for approximate KNNS in high-dimensional spaces [10]. The key idea is to hash the points in $\mathcal{M}$ using several hashing functions to ensure that for each function the probability of collision is much higher for points that are close to each other than for those that are far apart. One formal description is:

$$\mathbb{P}_{h\in\mathcal{F}}[h(\mathbf{x}) = h(\mathbf{y})] \propto sim(\mathbf{x},\mathbf{y}), \tag{1}$$

where $h(\cdot)$ is the hash function and belongs to the so called *locality-sensitive hash* (LSH) function family $\mathcal{F}$; $\mathbf{x},\mathbf{y}$ are two points in d-dimensional space; $sim(\cdot)$ is a function to measure the similarity between $\mathbf{x}$ and $\mathbf{y}$: for KNNS large $sim(\mathbf{x},\mathbf{y})$ means $\|\mathbf{x}-\mathbf{y}\|_p$, the distance between $\mathbf{x}$ and $\mathbf{y}$, is small. Different function families $\mathcal{F}$ can be used for different metrics. Datar et al. [5] proposed LSH families for l_2 metric based on p-stable distributions, where each hash function is defined as:

$$\mathbf{g}(\mathbf{x}) = \langle h_1(\mathbf{x}), h_2(\mathbf{x}), ..., h_M(\mathbf{x})\rangle \tag{2}$$

$$h_i(\mathbf{x}) = \lfloor \frac{\mathbf{a}_i \cdot \mathbf{x} + b_i}{W} \rfloor \quad i = 1,2,...,M. \tag{3}$$

In this case, $\mathbf{a}_i$ is a d-dimensional random vector with entries chosen independently from a normal distribution $\mathcal{N}(0,1)$ and b_i is drawn uniformly from the range $[0,W]$. M and W are parameters used to control the locality sensitivity of the hash function.

The KNNS search structure is a hash table: points with the same hash values are pushed into the same bucket of hash table. The query process is to scan within the buckets which the query point is hashed to. In order to improve the quality of KNNS, generally L different hash functions $\mathbf{g}_j(\cdot), j = 1,...,L$ are chosen randomly from function family $\mathcal{F}$. As a result, query algorithm must perform repeatedly on all L hash tables. LSH-based KNNS can perform one query in nearly constant time, which means the nearest neighbor in motion planning algorithm can be completed within linear time complexity to the number of samples.

D. GPU-based Algorithms

Many-core GPUs have been used for many geometric and scientific computations. The rasterization capabilities of a GPU can be used for motion planning of multiple low DOF robots [19], motion planning in dynamic scenes [9] or improve the sample generation in narrow passages [6], [16]. However, rasterization based planning algorithms are accurate only to the resolution of image-space.

The computational capabilities of many-core GPUs have been exploited to improve KNNS. Garcia et al. [7] describe an implementation of brute-force KNNS on GPU. Recently, some efficient GPU-based algorithms have been proposed to construct Kd-trees [20]. Pan et al. [15] propose a new approach to compute KNNS in low-dimensional space, which is based on the collision operations between *bounding volume hierarchies* (BVH) on GPU [12], [11].

III. Overview

In this section, we give an overview of the GPU-based motion planning framework, which is relatively easy to parallelize and can be used for single-query and multiple-query problems.

Our work is built on GPU-based sample-based planning algorithm called gPlanner [15]. We choose PRM as the underlying motion planning algorithm, because it is more 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.

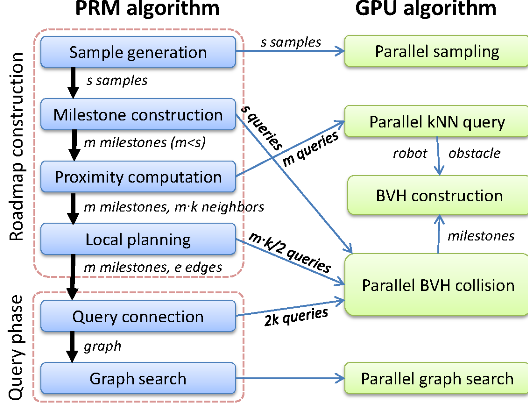

Fig. 1. Overview of the GPU-based real-time planner [15].

The PRM algorithm has two phases: roadmap construction and query phase, whose basic flowchart is shown in the left part of Fig 1. We use a many-core GPU to improve the performance of each component significantly and the framework for the overall GPU-based planner is shown in the right side of Fig 1.

We first use MD5 cryptographic hash function to generate random samples for each thread independently. For each sample generated, we need to check whether it is a milestone, i.e. does not collide with the obstacles using BVH trees [12] and exploit GPU parallelism [11].

For each milestone, we perform KNN query to find its nearest neighbors. Pan et al. [15] used parallel KNNS based on BVH collision, which is effective but has three drawbacks: 1) it is difficult to extend to high-dimensional cases; 2) it provides no theory guarantee for the timing performance; 3) it is difficult to control the approximation level of the nearest neighbors it computed. Instead, we use the LSH-based KNNS to overcome these difficulties.

Once the roadmap is constructed by local planning, we connect initial-goal configurations in one query to the roadmap, using similar algorithms as the KNN query. Finally we perform a parallel graph search on roadmap to obtain a collision-free path.

For more details about motion planning framework on GPU, please refer to [15].

IV. LSH-based KNNS on GPUs

In this section, we present our GPU-based algorithm for LSH based KNNS computation. We compute the nearest neighbors for n milestones $\{\mathbf{q}_i\}_{i=1}^n$ in the d-dimensional configuration space $\mathcal{M}$. The algorithm includes two steps: *parallel LSH computation* and *parallel KNNS computation*. We first compute the LSH value for each milestone and then search for the nearest neighbor in the milestones with the same LSH value.

A. Parallel LSH Computation

In this step, each GPU thread computes the LSH value for one milestone $\mathbf{q}$. We assume that $\mathcal{M}$ is an Euclidean space with weighted l_2 metric: $\|\mathbf{q}\|_\lambda = (\sum_{i=1}^d q_i^2 \lambda_i^2)^{1/2}$, where λ is the weight vector. We will discuss how to handle non-Euclidian case in Section IV-D.

The LSH function for weighted Euclidean space is similar to Equ 2 and 3, except that we need to consider the weight function as LSH only handles l_2 metric:

$$\mathbf{g}^\lambda(\mathbf{x}) = \langle h_1^\lambda(\mathbf{x}), h_2^\lambda(\mathbf{x}), ..., h_M^\lambda(\mathbf{x}) \rangle \tag{4}$$

$$h_i^\lambda(\mathbf{x}) = \lfloor \frac{\mathbf{a}_i \cdot \widehat{\mathbf{x}} + b_i}{W} \rfloor, \quad i = 1, 2, ..., M \tag{5}$$

where $\widehat{\mathbf{x}} = [x_1\lambda_1, ..., x_d\lambda_d]$. $\mathbf{q}$ will be stored in an indexing table $\mathbf{T}$ with the computed $\mathbf{g}^\lambda(\mathbf{q})$ as indexing key. For notational convenience, thereafter we denote $\mathbf{g}^\lambda(\cdot), h^\lambda(\cdot)$ simply by $\mathbf{g}(\cdot)$ and $h(\cdot)$.

B. Parallel KNNS

In this step, each GPU thread computes the k-nearest neighbor of one milestone $\mathbf{q}$. First, the LSH value $\mathbf{g}_1(\mathbf{q})$ is calculated and we use it as the key to find the bucket in indexing table $\mathbf{T}_1$ that $\mathbf{q}$ is located in. Then we search within the bucket to find the k nearest neighbors of $\mathbf{q}$. Usually the size of each bucket is quite small and the overall process is fast. We repeat the above process L times to handle all the L tables $\mathbf{T}_1$, $\mathbf{T}_2$, ..., $\mathbf{T}_L$ and the final result is the approximate k-nearest neighbor for $\mathbf{q}$.

C. Bucket Hashing

The indexing table $\mathbf{T}$ can be a M-dimensional grid which covers the domain of LSH function $\mathbf{g}(\cdot)$. However, $\mathbf{T}$ can be quite sparse and the range of $\mathbf{g}(\cdot)$ can be too large to store all the possible buckets explicitly. Therefore we need to use algorithms to store these LSH values in a compressed form. To this end, we use a secondary *bucket hashing* which includes two levels of hashing operations: a universal hashing (line 8 in Algo 1) and a cuckoo hashing (line 9-16 in Algo 1).

First, we hash the LSH values once more with a universal hash function:

$$h_u(\mathbf{g}(\cdot)) = \sum_{i=1}^M r_i \cdot h_i(\cdot), \tag{6}$$

where r_i is an integer chosen uniformly in $[0, 32768)$ and $h_i(\cdot)$ is the function defined in Equ 5. For all the n milestones, we can compute $\{h_u(\mathbf{g}(\mathbf{q_i}))\}_{i=1}^n$ in parallel. The function h_u reduces the M-dimensional indexing key into a one-dimension entity. We denote the hash value $h_u(\mathbf{g}(\mathbf{q}_i))$ as k_i.

Algorithm 1: Parallel KNNS algorithm on GPU

Input : milestones $\{\mathbf{q}\}$, neighbor size k
Output: k-nearest neighbor of each milestone

```
1  begin
2    foreach q ∈ milestones in parallel do
3      allocate memory size of k and attached
       with a max-heap heap[q]
4      heap[q] as q's neighbor cache
5    for i = 1 to L do
6      foreach q ∈ milestones in parallel do
7        compute LSH keys:
         g(q) = ⟨h_1(q), ..., h_M(q)⟩
8        compute bucket hashing keys:
         h_u(q) ≡ h_u(g(q))
9      KM = {(h_u(q), q)}, the array of
       key-milestone pairs
10     perform parallel radix sort on KM
       according to key
11     perform parallel difference on KM
12     perform parallel prefix sum on KM to
       determine N unique keys T_a
13     for i-th unique key T_a[i] do
14       compute the start index in KM: start[i]
15       compute the number of the key in KM:
         count[i]
16     T_i ≡ T_b =
       cuckoo-hashing({(T_a[i], i)}_{i=1}^N)
17     foreach (key, q) ∈ KM do
18       id = hash-lookup(T_b)
19       S = [start[id], start[id] + count[id])
20       for x ∈ S do
21         add KM[x].q to heap[q]
22   The k elements in heap[q] is the KNN for q
23 end
```

Next we use radix sorting to sort the array of n key-point pairs $(k_i, \mathbf{q}_i)$ according to key value. After that, we perform difference operation on the sorted result and use *parallel prefix-sum scan* on GPU to find the unique items within the n keys. Suppose there are N unique keys and we store them in an array $\mathbf{T}_a$. For the i-th unique key $\mathbf{T}_a[i]$, we can compute the start index $start[i]$ and the size $count[i]$ for the segment in sorted key-point sequence that corresponds to it.

N can still be quite large ($> 10,000$) which makes it time consuming to find the index for a given key in the array $\mathbf{T}_a$. We further store the key-index pair (u_i, i) in a hash table to accelerate the index lookup for a given key. We use the parallel cuckoo hashing [2] to build the second level indexing structure $\mathbf{T}_c$. Cuckoo hashing places at most one item at each location in the hash table by allowing items to be moved after their initial placement. It stores the key-index pairs in f hash sub-tables ($f \geq 3$).

KNNS process based on the two-level indexing structure $\mathbf{T}_a$ and $\mathbf{T}_c$ is very efficient. Given a query $\mathbf{q}$, we compute its bucket hashing value k according to Equ 6. Then using k as hash key, we can find the index id for k in at most f (i.e. the number of sub-tables) checks. Then the segment $[start[id], start[id] + count[id])$ in the resorted key-point sequence is the bucket for the nearest neighbors of $\mathbf{q}$. (It is possible that the bucket contains points with different LSH values with $\mathbf{q}$ but has the same bucket hashing value. However, we can simply filter these items by computing the hamming distance between the LSH value of $\mathbf{q}$ and theirs.)

Algo 1 shows the detail for parallel LSH computation, parallel KNNS and bucket hashing.

D. Non-Euclidean Metrics

Generally, LSH is difficult to extend to non-Euclidean metric. However, we can handle rotational DOFs using two level LSH-hashing.

First, we use the algorithm in Section IV-A and Section IV-B to handle translational DOFs, but we use a large W so as for each configuration the number of returned nearest neighbors is much larger than K.

Next for one spherical joint, we can compute the direction vector based on the 3 rotational DOFs (Euler anglers). For all the n samples, we can have n dim-3 vectors $\mathbf{x}_1, ..., \mathbf{x}_n$, where $\|\mathbf{x}_i\|_2 = 1$. It is known that for vectors on unit sphere, the following relationship holds:

$$\mathbb{P}(\mathrm{sign}(\mathbf{x} \cdot \mathbf{r}) = \mathrm{sign}(\mathbf{y} \cdot \mathbf{r})) = 1 - \frac{1}{\pi} \cos^{-1}(\mathbf{x} \cdot \mathbf{y}), \quad (7)$$

where $\mathbf{r}$ is a uniform generated vector. According to its similarity with Equ 1, we can use following function as the locality sensitive function:

$$h(\mathbf{x}) = \begin{cases} 1, & \text{if } \mathbf{r} \cdot \mathbf{x} \geq 0 \\ 0, & \text{otherwise} \end{cases} \quad (8)$$

For each joint, we can choose M different $\mathbf{r}$ and thus M different functions $h_i(\cdot), i = 1, 2, ..., M$. Similar to Equ 2, we compose them in to LSH function $\mathbf{g}(\cdot)$. The LSH function for all m spherical joints are the composition of m different $\mathbf{g}(\cdot)$. Then we can perform the KNNS similar to Section IV-A and Section IV-B. Of course, this method uses angle-based metric, which is different from the local l_2 metric used in the first method. As a result, the KNNS results of the two methods may be different.

V. 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 Core2 CPU at 3.2GHz CPU and 6GB memory. We implemented our algorithms using CUDA on a NVIDIA GTX 285 GPU with 1GB of video memory.

We implement a motion planning algorithm using our LSH-based KNNS as the proximity computation component and compare it with previous motion planning algorithms implemented on CPU (OOPSMP library [17], PRM and RRT, use GNAT for KNNS) and GPU ([15], PRM and lazy PRM, use BVH for KNNS). The result is shown in Table I and the collision free paths for two benchmarks computed by our algorithm are shown in Fig 2. The comparison shows several things: 1) Our method improves KNNS's performance on GPU; 2) LSH-based KNNS can capture the connectivity of the configuration space; 3) LSH-based KNNS can improve the overall performance of motion planning algorithm, because it can provide high-quality nearest neighbor which can reduce the computational complexity in CCD.

We further compare the scalability of LSH and BVH based KNNS and the result is shown in Fig 5. When the number of points is small, the performances of LSH- and BVH-based KNNS are similar. However, when the number of points increases, the performance of BVH-based KNNS reduces much faster than LSH-based KNNS: BVH-based KNNS has superlinear complexity while LSH-based KNNS is linear complexity. Moreover, in Fig 5 there is no timing result for BVH-based KNNS when the number of samples is larger than $40,000$, because our GPU can not allocate enough memory for BVH-tree for so many samples. Therefore, it also shows that LSH-based KNNS has smaller space complexity than BVH-based KNNS.

Finally, we analyze the accuracy of our algorithm. Fig 3 compares the KNNS quality between LSH- and BVH-based methods. We use two criteria to evaluate any approximate KNNS's quality: $\frac{r_{max}}{r^0_{max}}$ and $\frac{r_{min}}{r^0_{max}}$, where r_{min} and r_{max} are the minimum and maximum distance between the query point and the k-nearest neighbor points returned by the algorithm and r^0_{max} is the distance of the exact k-th nearest point to the query point. From this comparison, we can see that LSH-based KNNS provides much better result in terms of approximate KNNS computation. We also show how LSH's L parameter influences the quality of KNNS in Fig 4. For each query point, we compute the intersection part of its neighborhoods computed by LSH-based KNNS and exact KNNS. Fig 4 shows that when L grows, more query points have a large intersection set, which means the quality of KNNS is increasing.

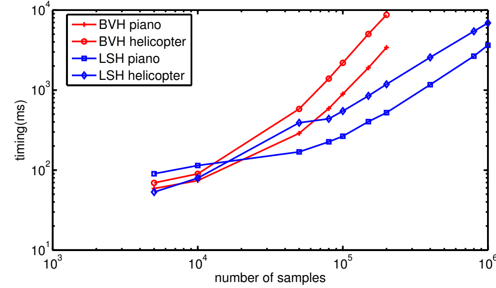

Fig. 5. The scalability of BVH-and LSH-based KNNS. BVH-based KNNS does not have timing when sample number is larger than $40,000$ because GPU can not allocate enough memory for BVH-tree.

VI. Conclusions

In this paper, we have introduced an efficient k-nearest neighbor search algorithm on GPU. Based on local sensitive hashing and cuckoo hashing techniques, our novel algorithm can provide nearest neighbors with faster speed, high accuracy and better scalability, when comparing with previous BVH-based algorithms. It can also be used as the component of GPU-based motion planning algorithm and can improve the overall performance of planner significantly.

References

[1] N. Ailon and B. Chazelle, "Approximate nearest neighbors and the fast johnson-lindenstrauss transform," in *ACM symposium on Theory of computing (STOC)*, 2006, pp. 557–563.

[2] D. A. Alcantara, A. Sharf, F. Abbasinejad, S. Sengupta, M. Mitzenmacher, J. D. Owens, and N. Amenta, "Real-time parallel hashing on the gpu," in *SIGGRAPH Asia*, 2009, pp. 1–9.

[3] N. Amato and L. Dale, "Probabilistic roadmap methods are embarrassingly parallel," *Proceedings of ICRA*, 1999.

[4] A. Chakrabarti and O. Regev, "An optimal randomised cell probe lower bound for approximate nearest neighbour searching," in *IEEE Symposium on Foundations of Computer Science (FOCS)*, 2004, pp. 473–482.

[5] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni, "Locality-sensitive hashing scheme based on p-stable distributions," in *Symposium on Computational Geometry (SCG)*, 2004, pp. 253–262.

[6] M. Foskey, M. Garber, M. Lin, and D. Manocha, "A voronoi-based hybrid planner," *Proc. of IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, 2001.

[7] V. Garcia, E. Debreuve, and M. Barlaud, "Fast k nearest neighbor search using gpu," in *Computer Vision and Pattern Recognition Workshops, CVPRW'08*, 2008, pp. 1–6.

[8] M. Gini, "Parallel search algorithms for robot motion planning," in *IEEE International Conference on Robotics and Automation (ICRA)*, 1996.

[9] K. Hoff, T. Culver, J. Keyser, M. Lin, and D. Manocha, "Interactive motion planning using hardware accelerated computation of generalized voronoi diagrams," *Proceedings of IEEE Conference of Robotics and Automation*, 2000.

		piano (50,000 samples)			helicopter (10,000 samples)		
CPU-PRM		6.53s			8.20s		
CPU-RRT		19.44s			20.94s		
		KNNS	Other	Sum	KNNS	Other	Sum
gPlanner [15]	non-lazy	288.32ms	1397.2ms	1685.5ms	88.68ms	1707.6ms	1796.3ms
	lazy	288.32ms	156.73ms	445.05ms	88.68ms	193.96ms	282.64ms
LSH-based GPU-PRM	non-lazy	168.61ms	602.3ms	770.9ms	79.36ms	677.0ms	756.4 ms
	lazy	168.61ms	201.73ms	370.54ms	79.36ms	198.96ms	278.32ms

TABLE I

TIMING COMPARISON BETWEEN OUR ALGORITHM AND [17] [15]. THE NEAREST NEIGHBOR SIZE IS 15.

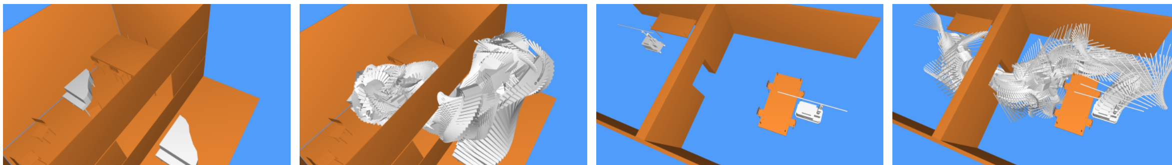

Fig. 2. The first two images show the piano scene and the collision-free path computed by our GPU planner. The other two images show the helicopter scene and the result. It verifies that our LSH-based nearest neighbor algorithm can capture the connectivity of the free space.

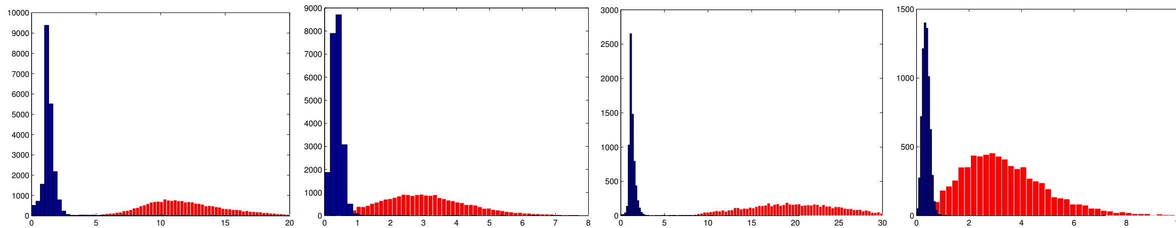

Fig. 3. KNNS accuracy comparison between LSH and BVH-based KNNS. The first two images show the $\frac{r_{max}}{r^0_{max}}$ and $\frac{r_{min}}{r^0_{max}}$ of LSH and BVH-based KNNS on piano benchmark. The blue one is for LSH-based KNNS and the red one is for BVH. The other two show the comparison on the helicopter benchmark. $\frac{r_{max}}{r^0_{max}}$ and $\frac{r_{min}}{r^0_{max}}$ are less for LSH-based KNNS than the BVH-based one on both benchmarks, which means that LSH-based KNNS provides nearest neighbors with higher quality.

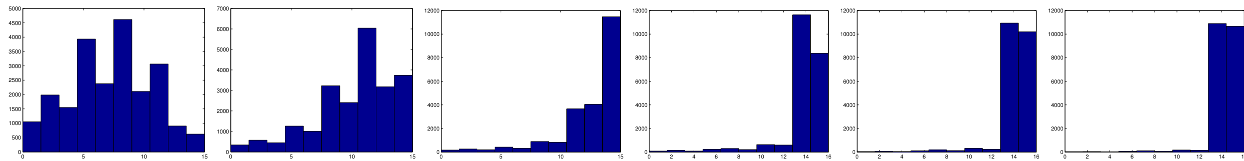

Fig. 4. From left to right, we show the histogram of points according to the number of common items between their k-nearest neighbors ($k = 15$) computed by LSH-based and exact KNNS, when the LSH parameter $L = 5, 10, 20, 40, 80, 160$. When $L = 80, 160$, LSH's result is almost the same as the exact KNNS: 99% points have k or $k-1$ (14 or 15) common parts between results of LSH-based and exact KNNS.

[10] P. Indyk and R. Motwani, "Approximate nearest neighbors: towards removing the curse of dimensionality," in *ACM symposium on Theory of computing (STOC)*, 1998, pp. 604–613.

[11] C. Lauterbach, Q. Mo, and D. Manocha, "gproximity: Hierarchical gpu-based operations for collision and distance queries," *Computer Graphics Forum (Proc. of Eurographics)*, vol. 29, no. 2, pp. 419–428, 2010.

[12] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha, "Fast bvh construction on gpus," *Computer Graphics Forum*, vol. 28, no. 2, pp. 375–384, 2009.

[13] S. M. LaValle, *Planning Algorithms*. Cambridge University Press (also available at http://msl.cs.uiuc.edu/planning/), 2006.

[14] D. Mount and S. Arya, "Ann: A library for approximate nearest neighbor searching," in *CGC 2nd Annual Fall Workshop on Computational Geometry*, 1997.

[15] J. Pan, C. Lauterbach, and D. Manocha, "g-planner: Real-time motion planning and global navigation using GPUs," in *AAAI Conference on Artificial Intelligence (AAAI)*, Jul 2010.

[16] C. Pisula, K. Hoff, M. Lin, and D. Manocha, "Randomized path planning for a rigid body based on hardware accelerated voronoi sampling," in *Proc. of 4th International Workshop on Algorithmic Foundations of Robotics*, 2000.

[17] E. Plaku, K. E. Bekris, and L. E. Kavraki, "Oops for motion planning: An online open-source programming system," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2007, pp. 3711–3716.

[18] H. Samet, *Foundations of MultiDimensional and Metric Data Structures*. Morgan Kaufmann, 2006.

[19] A. Sud, E. Andersen, S. Curtis, M. Lin, and D. Manocha, "Real-time path planning for virtual agents in dynamic environments," *Proc. of IEEE VR*, pp. 91–98, 2007.

[20] K. Zhou, Q. Hou, R. Wang, and B. Guo, "Real-time kd-tree construction on graphics hardware," in *SIGGRAPH Asia 2008*, 2008.