GPU-based Parallel Collision Detection for Real-Time Motion Planning

Jia Pan and Dinesh Manocha

Abstract We present parallel algorithms to accelerate collision queries for sample-based motion planning. Our approach is designed for current many-core GPUs and exploits the data-parallelism and multi-threaded capabilities. In order to take advantage of high number of cores, we present a clustering scheme and collision-packet traversal to perform efficient collision queries on multiple configurations simultaneously. Furthermore, we present a hierarchical traversal scheme that performs workload balancing for high parallel efficiency. We have implemented our algorithms on commodity NVIDIA GPUs using CUDA and can perform $500,000$ collision queries/second on our benchmarks, which is 10X faster than prior GPU-based techniques. Moreover, we can compute collision-free paths for rigid and articulated models in less than 100 milliseconds for many benchmarks, almost 50-100X faster than current CPU-based planners.

1 Introduction

Motion planning is one of the fundamental problems in algorithmic robotics. The goal is to compute collision-free paths for robots in complex environments. Some of the widely used algorithms for high-DOF (degree-of-freedom) robots are based on randomized sampling. These include algorithms based on PRMs [12] and RRTs [14]. These methods tend to capture the topology of the free configuration space of the robot by generating a high number of random configurations and connecting nearby collision-free configurations (i.e. milestones) using local planning methods. The resulting algorithms are probabilistically complete and have been successfully used to solve many challenging motion planning problems.

In this paper, we address the problem of designing fast and almost real-time planning algorithms for rigid and articulated models. The need for such algorithms arises not only from virtual prototyping and character animation, but also task planning for physical robots. Current robots (such as Willow Garage's PR2) tend to use live sensor data to generate a reasonably accurate model of the objects in the physical world. Some tasks, such as robot navigation or grasping, need to compute a collision-free path for the manipulator in real-time to handle dynamic environments. Moreover, many high-level task planning algorithms perform motion planning and subtask execution in an interleaved manner, i.e. the planning result of one subtask is used to construct the formulation of the following subtask [27]. A fast and almost real-time planning algorithm is important for these applications.

It is well known that a significant fraction (e.g. 90% or more) of randomized sampling algorithms is spent in collision checking. This includes checking whether

University of North Carolina, Chapel Hill, NC, USA
{*panj,dm*}*@cs.unc.edu*
http://gamma.cs.unc.edu/gplanner

a given configuration is in free-space or not as well as connecting two free-space configurations using a local planning algorithm. While there is extensive literature on fast intersection detection algorithms, some of the recent planning algorithms are exploiting the computational power and massive parallelism of commodity GPUs (graphics processing units) for almost real-time computation [23, 22]. Current GPUs are high-throughput many-core processors, which offer high data-parallelism and can simultaneously execute a high number of threads. However, they have a different programming model and memory hierarchy as compared to CPUs. As a result, we need to design appropriate parallel collision and planning algorithms that can map well to GPUs.

Main Results: We present a novel, parallel algorithm to perform collision queries for sample-based motion planning. Our approach exploits parallelism at two levels: it checks multiple configurations simultaneously whether they are in free space or not and performs parallel hierarchy traversal for each collision query. Similar techniques are also used for local planning queries. We present clustering techniques to appropriately allocate the collision queries to different cores, Furthermore, we introduce the notion of *collision-packet traversal*, which ensures that all the configurations allocated to a specific core result in similar hierarchical traversal patterns. The resulting approach also exploits fine-grained parallelism corresponding to each bounding volume overlap test to balance the workload.

The resulting algorithms have been implemented on commodity NVIDIA GPUs. In practice, we are able to process about $500,000$ collision queries per second on a \$400 NVIDIA GeForce 480 desktop GPU, which is almost 10X faster than prior GPU-based collision checking algorithms. We also apply our collision checking algorithm for GPU-based motion planners to high DOF rigid and articulated robots. The resulting planner can compute collision-free paths in less than 100 milliseconds for various benchmarks and appears to be 50-100X faster than CPU-based planners.

The rest of the paper is organized as follows. We survey related work on real-time motion planning and parallel collision-checking algorithms in Section 2. Section 3 gives an overview of our approach and we present our new parallel algorithm for parallel collision queries in Section 4. We highlight the performance of our algorithm on different benchmarks in Section 5.

2 Previous Work

In this section, we give a brief overview of prior work in real-time motion planning and parallel algorithms for collision detection.

2.1 Real-time Motion Planning

An excellent survey of various motion planning algorithms is given in [17]. Many parallel algorithms have also been proposed for motion planning by utilizing the properties of configuration spaces [20]. The distributed representation [5] can be easily parallelized. In order to deal with very high dimensional or difficult planning problems, distributed sampling-based techniques have been proposed [25].

The computational power of many-core GPUs has been used for many geometric and scientific computations [21]. The rasterization capabilities of a GPU can be used for real-time motion planning of low DOF robots [10, 26] or improve the sample generation in narrow passages [24, 7]. Recently, GPU-based parallel motion planning algorithms have been proposed for rigid models [23, 22].

2.2 Parallel Collision Queries

Some of the widely used algorithms for collision query are based on *bounding volume hierarchies* (BVH), such as *k*-DOP trees, OBB trees, AABB trees, etc [18]. Recent developments include parallel hierarchical computations on multi-core CPUs [13, 28] and GPUs [16]. CPU-based approaches tend to rely on fine-grained communication between processors, which is not suited for current GPU-like architectures. On the other hand, GPU-based algorithms [16] use work queues to parallelize the computation on the multiple cores. All these approaches are primarily designed to parallelize a single collision query for sample-based motion planning.

The capability to perform a high number of collision queries efficiently is essential in motion planning algorithms, e.g. for parallel collision queries in milestone computation and local planning. Some of the prior algorithms perform parallel queries in a simple manner: each thread handles a single collision query in an independent manner [23, 22, 3, 2]. As current multi-core CPUs have the capability to perform *multiple-instruction multiple-data* (MIMD) computations, these simple strategies can work well on CPUs. On the other hand, current GPUs offer high data parallelism and the ability to execute a high number of threads in parallel to overcome the high memory latency. As a result, we need different parallel collision detection algorithms to fully exploit their capabilities.

3 Overview

In this section, we first provide some background on current GPU architectures. Next, we address some issues in designing efficient parallel algorithms to perform collision queries.

3.1 GPU Architectures

In recent years, the focus in processor architectures has shifted from increasing clock rate to increasing parallelism. Commodity GPUs such as NVIDIA Fermi[1] have theoretical peak performance of Tera-FLOP/s for single precision computation and hundreds of Giga-FLOP/s for double precision computations. This peak performance is significantly higher as compared to current multi-core CPUs, thus outpacing CPU architectures [19] at relatively modest cost of \$300 to \$400. However, GPUs have different architectural characteristics and memory hierarchy, that impose some constraints in terms of designing appropriate algorithms. First, GPUs usually have a high number of independent cores (e.g. the newest generation GTX 480 has 15 cores and each core has 32 streaming processors resulting in total of 480 processors while GTX 280 has only 240 processors). Each of the individual cores is a vector processor capable of performing the same operation on several elements simultaneously (e.g. 32 elements for current GPUs). Secondly, the memory hierarchy on GPUs is quite different from that of CPUs and cache sizes on the GPUs are considerably smaller. Moreover, each GPU core can handle several separate tasks in parallel and switch between them in the hardware when one of them is waiting for a memory operation to complete. This hardware multithreading ap-

[1] `http://www.nvidia.com/object/fermi_architecture.html`

proach is thus designed to hide the memory access latency. Thirdly, all GPU threads are logically grouped in blocks with a per-block shared memory, which provides a weak synchronization capability between the GPU cores. Overall, shared memory is a limited resource on GPUs: increasing the shared memory distributed for each thread can limit the extent of parallelism. Finally, the threads are physically processed in chunks in SIMT (single-instruction, multiple-thread). This is different from SIMD (single-instruction multiple-data) and each thread can execute independent instructions. The GPU's performance can reduce significantly when threads in the same chunk diverge considerably, because these diverging portions are executed in a serial manner for all the branches. As a result, threads with coherent branching decisions (e.g. threads traversing the same paths in the BVH) are preferred on GPUs in order to obtain higher performance [8]. All of these characteristics imply that – unlike CPUs – achieving high performance in current GPUs depends on several factors: (1) generating a sufficient number of parallel tasks so that all the cores are highly utilized; (2) developing parallel algorithms such that the total number of threads is even higher than the number of tasks, so that each core has enough work to perform while waiting for data from relatively slow memory accesses; (3) assigning appropriate size for shared memory to accelerate memory accesses and not reduce the level of parallelism; (4) performing coherent or similar branching decisions for each parallel thread within a given chunk. These requirements impose constraints in terms of designing appropriate collision query algorithms.

3.2 Notation and Terminology

We define some terms and highlight the symbols used in the rest of the paper.

chunk The minimum number of threads that GPUs manage, schedule and execute in parallel, which is also called *warp* in the GPU computing literatures. The size of chunk (*chunk-size* or *warp-size*) is 32 on current NVIDIA GPUs (e.g. GTX 280 and 480).

block The collection of GPU threads that will be executed on the same GPU core. These threads synchronize by using barriers and communicate via a small high-speed low-latency *shared memory*.

BVH_a The *bounding volume hierarchy* (BVH) tree for model a. It is a binary tree with L levels, whose nodes are ordered in the breadth-first order starting from the root node. Each node is denoted as $\mathrm{BVH}_a[i]$ and its children nodes are $\mathrm{BVH}_a[2i]$ and $\mathrm{BVH}_a[2i+1]$ with $1 \leq i \leq 2^{L-1}-1$. The nodes at the l-th level of a BVH tree are represented as $\mathrm{BVH}_a[k], 2^l \leq k \leq 2^{l+1}-1$ with $0 \leq l < L$. The inner nodes are also called *bounding volumes* (BV) and the leaf nodes also have a link to the primitive triangles that are used to represent the model.

$\mathrm{BVTT}_{a,b}$ The *bounding volume test tree* (BVTT) represents recursive collision query traversal between two objects a, b. It is a 4-ary tree, whose nodes are ordered in the breadth-first order starting from the root node. Each node is denoted as $\mathrm{BVTT}_{a,b}[i] \equiv (\mathrm{BVH}_a[m], \mathrm{BVH}_b[n])$ or simply (m,n), which checks the BV or primitive overlap between nodes $\mathrm{BVH}_a[m]$ and $\mathrm{BVH}_b[n]$, while $m = \lfloor i - \frac{4^M+2}{3} \rfloor + 2^M$, $n = \{i - \frac{4^M+2}{3}\} + 2^M$ and $M = \lfloor log_4(3i-2) \rfloor$. BVTT node (m,n)'s children are $(2m,2n)$, $(2m,2n+1)$, $(2m+1,2n)$, $(2m+1,2n+1)$.

q A configuration of the robot, which is randomly sampled within the configuration space $\mathscr{C}$-Space. **q** is associated with the transformation $\mathbf{T_q}$. The BVH of a model a after applying such a transformation is given as $\text{BVH}_a(\mathbf{q})$.

3.3 Collision Queries: Hierarchical Traversal

Collision queries between the geometric models are usually accelerated with hierarchical techniques based on BVHs, which correspond to traversing the BVTT related with the BVHs of the models [15]. The simplest parallel algorithms to perform multiple collision queries are based on each thread traversing the BVTT and checking whether a given configuration is in free space or not. Such a simple parallel algorithm is highlighted in Algorithm 1. This strategy is easy to implement and has been used in previous parallel planning algorithms based on multi-core or multiple CPUs. But it may not result in high parallel efficiency on current GPUs due to the following reasons. First, each thread needs a local traverse stack on the shared memory which may not be effective for complex models with thousands of polygons. Second, different threads may traverse the BVTT tree with incoherent patterns: there are many branching decisions performed during the traversal (e.g. **loop**, **if**, **return** in the pseudo-code) and the traversal flow of the hierarchy in different threads diverges quickly. Finally, different threads can have varying workloads; some may be busy with the traversal while the others may have finished the traversal early due to no overlap and are idle. These factors can affect the performance of the parallel algorithm.

The problems of low parallel efficiency in Algorithm 1 become more severe in complex or articulated models. For such models, there are longer traversal paths in the hierarchy and the difference between these paths can be large for different configurations. As a result, differences in the workloads between different threads

Algorithm 1 Simple parallel collision checking; Such approaches are frequently used on multi-core CPUs

```
1: Input: $N$ random configurations $\{\mathbf{q}_i\}_{i=1}^N$, $\text{BVH}_a$ for the robot and $\text{BVH}_b$ for the obstacles
2: Output: return whether one configuration is in free space or not
3: $t_{id} \leftarrow$ thread id of current thread
4: $\mathbf{q} \leftarrow \mathbf{q}_{t_{id}}$
5: $\lhd$ traverse stack $S[]$ is initialized with root nodes
6: shared $S[] \equiv$ local traversal stack
7: $S[] \leftarrow \text{BVTT}[1] \equiv (\text{BVH}_a(\mathbf{q})[1], \text{BVH}_b[1])$
8: $\lhd$ traverse BVTT for $\text{BVH}_a(\mathbf{q})$ and $\text{BVH}_b$
9: loop
10:     $(x,y) \leftarrow pop(S)$.
11:     if $overlap(\text{BVH}_a(\mathbf{q})[x], \text{BVH}_b[y])$ then
12:         $S[] \leftarrow (2x,2y),(2x,2y+1),(2x+1,2y),(2x+1,2y+1)$ if $!isLeaf(x)$ &&$!isLeaf(y)$
13:         $S[] \leftarrow (2x,2y),(2x,2y+1)$ if $isLeaf(x)$ && $!isLeaf(y)$
14:         $S[] \leftarrow (2x,2y),(2x+1,2y)$ if $!isLeaf(x)$ && $isLeaf(y)$
15:         return $collision$ if $isLeaf(x)$ && $isLeaf(y)$ && $exactIntersect(\text{BVH}_a(\mathbf{q})[x], \text{BVH}_b[y])$
16:     end if
17: end loop
18: return $collision\text{-}free$
```

can be high. For articulated models, each thread checks the collision status of all the links and stops when a collision is detected for any link. Therefore, more branching decisions are performed within each thread and this can lead to more incoherence. Similar issues also arise during local planning when each thread determines whether two milestones can be joined by a collision-free path by checking the collisions along the trajectory connecting them.

4 Parallel Collision Detection on GPUs

In this section, we present two novel algorithms for efficient parallel collision checking on GPUs between rigid or articulated models. Our methods can be used to check whether a configuration lies in the free space or to perform local planning computations. The first algorithm uses clustering and fine-grained packet-traversal to improve the coherence of BVTT traversal for different threads. The second algorithm uses queue-based techniques and lightweight workload balancing to achieve higher parallel performance on the GPUs. In practice, the first method can provide 30%-50% speed up. Moreover, it preserves the per-thread per-query structure of the naive parallel strategy. Therefore, it is easy to implement and is suitable for cases where we need to perform some additional computations (e.g. retraction for handling narrow passages [29]). The second method can provide 5-10X speed up, but is relatively more complex to implement.

4.1 Parallel Collision-Packet Traversal

Our goal is to ensure that all the threads in a block performing BVTT-based collision checking have similar workloads and coherent branching patterns. This approach is motivated by recent developments related to interactive ray-tracing on GPUs for visual rendering. Each collision query traverses the BVTT and performs node-node or primitive-primitive intersection tests. In contrast, ray-tracing algorithms traverse the BVH tree and perform ray-node or ray-primitive intersections. Therefore, parallel ray-tracing algorithms on GPUs also need to avoid incoherent branches and varying workloads to achieve higher performance.

In real-time ray tracing, one approach to handle the varying workloads and incoherent branches is the use of ray-packets [8, 1]. In ray-tracing terminology, packet traversal implies that a group of rays follows exactly the same traversal path in the hierarchy. This is achieved by sharing the traversal stack (similar to the BVTT traversal stack in Algorithm 1) among the rays in the same warp-sized packet (i.e. threads that fit in one chunk on the GPU), instead of each thread using an independent stack for a single ray. This implies that the same additional nodes in the hierarchy may be visited during ray intersection tests, even though there are no intersections between the rays and those nodes. But the resulting traversal is coherent for different rays, because each node is fetched only once per packet. In order to reduce the number of computations (i.e. unnecessary node intersection tests), all the rays in one packet should be similar to one another, i.e. have similar traversal paths with few differing branches. We extend this idea to parallel collision checking and refer to our algorithm as *multiple configuration-packet* method.

The first challenge is to cluster similar collision queries or the configurations into groups. In some cases, the sampling scheme (e.g. the adaptive sampling for lazy

PRM) can provide natural group partitions. However, in most cases we need suitable algorithms to compute these clusters. Clustering algorithms are natural choices for such a task, which aims at partitioning a set $\mathscr{X}$ of N data items $\{\mathbf{x}_i\}_{i=1}^N$ into K groups $\{C_k\}_{k=1}^K$ such that the data items belonging to the same group are more "similar" than the data items in different groups. The clustering algorithm used to group the configurations needs to satisfy some additional constraints: $|C_k| = \textit{chunk-size}, 1 \le k \le K$, i.e. each cluster should fit in one chunk on GPUs, except for the last cluster and $K = \lceil \frac{N}{\textit{chunk-size}} \rceil$. Using the formulation of k-means, the clustering problem can be formally described as: compute $K = \lceil \frac{N}{\textit{chunk-size}} \rceil$ items $\{\mathbf{c}_k\}_{k=1}^K$ that minimizes

$$\sum_{i=1}^{N} \sum_{k=1}^{K} \mathbf{1}_{\mathbf{x}_i \in C_k} \|\mathbf{x}_i - \mathbf{c}_k\|, \tag{1}$$

with constraints $|C_k| = \textit{chunk-size}, 1 \le k \le K$. To our knowledge, there are no clustering algorithms designed for this specific problem. One possible solution is *clustering with balancing constraints* [4], which has additional constraints $|C_k| \ge m, 1 \le k \le K$, where $m \le \frac{N}{K}$.

Instead of solving Equation (1) exactly, we use a simpler clustering scheme to compute an approximate solution. First, we use k-means algorithm to cluster the N queries into C clusters, which can be implemented efficiently on GPUs [6]. Next, for k-th cluster of size S_k, we divide it into $\lceil \frac{S_k}{\textit{chunk-size}} \rceil$ sub-clusters, each of which corresponds to a *configuration-packet*. This simple method has some disadvantages. For example, the number of clusters is $\sum_{k=1}^{C} \lceil \frac{S_k}{\textit{chunk-size}} \rceil \ge K = \lceil \frac{N}{\textit{chunk-size}} \rceil$ and therefore Equation (1) may not result in an optimal solution. However, as shown later, even this simple method can improve the performance of parallel collision queries.

Next we map each configuration-packet to a single chunk. Threads within one packet will traverse the BVTT synchronously, i.e. the algorithm works on one BVTT node (x,y) at a time and processes the whole packet against the node. If (x,y) is a leaf node, an exact intersection test is performed for each thread. Otherwise, the algorithm loads its children nodes and tests the BVs for overlap to determine the remaining traversal order, i.e. to select one child (x_m,y_m) as the next BVTT node to be traversed for the entire packet. We select (x_m,y_m) in a greedy manner: it corresponds to the child node that is classified as overlapping by the most threads in the packet. We also push other children into the packet's traversal stack. In case no BV overlap is detected in all the threads or (x,y) is a leaf node, (x_m,y_m) would be the top element in the packet's traversal stack. The traversal step is repeated recursively, until the stack is empty. Compared to Algorithm 1, all the threads in one chunk share one traversal stack in shared memory, instead of using one stack for each thread. Therefore, the size of shared memory used is reduced by *chunk-size* times and results in higher parallel efficiency.

The traversal order described above is a greedy heuristic that tries to minimize the traversal path of the entire packet. For one BVTT node (x,y), if the overlap is not detected in any of the threads, it implies that these threads will not traverse the subtree rooted at (x,y). Since all the threads in the packet are similar and traverse the BVTT in nearly identical order, this implies that other threads in the same packet might not traverse the sub-tree either. We define the probability that the sub-tree rooted at (x,y) will be traversed by one thread as $p_{x,y} = \frac{\#overlap\ threads}{packet\text{-}size}$. For any traverse pattern P for BVTT, the probability that it is carried on by BVTT traversal will be $p_P = \prod_{(x,y)\in P} p_{x,y}$. As a result, our new traversal strategy guarantees that the traversal pattern with higher traverse probability will have a shorter traversal length, and therefore minimizes the overall path for the packet.

The decision about which child node is the candidate for next traversal step is computed using sum reduction [9], which can compute the sum of n items in parallel with $O(\log(n))$ complexity. Each thread writes a 1 in its own location in the shared memory if it detects overlap in one child and 0 otherwise. The sum of the memory locations is computed in 5 steps for a size 32 chunk. The packet chooses the child node with the maximum sum. The complete algorithm for configuration-packet computation is described in Algorithm 2.

Algorithm 2 Multiple Configuration-Packet Traversal

```
1: Input: $N$ random configurations $\{\mathbf{q}_i\}_{i=1}^N$, BVH$_a$ for the robot and BVH$_b$ for the obstacles
2: $t_{id} \leftarrow$ thread id of current thread
3: $\mathbf{q} \leftarrow \mathbf{q}_{t_{id}}$
4: **shared** $CN[]\equiv$ shared memory for children node
5: **shared** $TS[]\equiv$ local traversal stack
6: **shared** $SM[]\equiv$ memory for sum reduction
7: **return if** $overlap(\text{BVH}_a(\mathbf{q})[1], \text{BVH}_b[1])$ is **false** for all threads in chunk
8: $(x,y) = (1,1)$
9: **loop**
10:    **if** $isLeaf(x)$ && $isLeaf(y)$ **then**
11:        update collision status of **q** **if** $exactIntersect(\text{BVH}_a(\mathbf{q})[x],\text{BVH}_b[y])$
12:        **break**, **if** $TS$ is empty
13:        $(x,y) \leftarrow pop(TS)$
14:    **else**
15:        $\lhd$ decide the next node to be traversed
16:        $CN[] \leftarrow (x,y)$'s children nodes
17:        **for all** $(x_c,y_c) \in CN$ **do**
18:            $\lhd$ compute the number of threads that detect overlap at node $(x_c,y_c)$
19:            write $overlap(\text{BVH}_a(\mathbf{q})[x_c],\text{BVH}_b[y_c])$ (0 or 1) into $SM[t_{id}]$ accordingly
20:            compute local summation $s_c$ in parallel by all threads in chunk
21:        **end for**
22:        **if** $\max_c s_c > 0$ **then**
23:            $\lhd$ select the node that is overlapped in the most threads
24:            $(x,y) \leftarrow CN[\text{argmax}_c s_c]$ and push others into $TS$
25:        **else**
26:            $\lhd$ select the node from the top of stack
27:            **break**, **if** $TS$ is empty
28:            $(x,y) \leftarrow pop(TS)$
29:        **end if**
30:    **end if**
31: **end loop**
```

4.2 Parallel Collision Query with Workload Balancing

Both Algorithm 1 and Algorithm 2 use the per-thread per-query strategy, which is easy to implement. However, when the idle threads wait for busy threads or when the execution path of threads diverges, the parallel efficiency on the GPUs is low. Algorithm 2 can reduce this problem in some cases, but it still distributes the tasks among the separate GPU cores and cannot make full use of the GPU's computational power.

In this section, we present the parallel collision query algorithm based on workload balancing which further improves the performance. In this algorithm, the task of each thread is no longer one complete collision query or continuous collision query (for local planning). Instead, each thread only performs BV overlap tests. In other words, the unit task for each thread is distributed in a more fine-grained manner. Basically, we formulate the problem of performing multiple collision queries as a pool of BV overlap tests which can be performed in parallel. It is easier to distribute these fine-grained tasks in a uniform manner onto all the GPU cores, and thereby balancing the load among them, than to distribute the collision query tasks.

All the tasks are stored in Q large work queues in the GPU's main memory, which has a higher latency compared to the shared memory. When computing a single collision query [16], the tasks are in the form of BVTT nodes (x,y). Each thread will fetch some tasks from one work queue into its local work queue on the shared memory and traverse the corresponding BVTT nodes. The children generated for each node are also pushed into the local queue as new tasks. This process is repeated for all the tasks remaining in the queue, until the number of threads with full or empty local work queues exceeds a given threshold (we use 50% in our implementation) and non-empty local queues are copied back to the work queues on main memory. Since each thread performs simple tasks with few branches, our algorithm can make full use of GPU cores if there are sufficient tasks in all the work queues. However, during BVTT traversal, the tasks are generated dynamically and thus different queues may have varying numbers of tasks and this can lead to an uneven workload among the GPU cores. We use a balancing algorithm that redistributes the tasks among work queues (Figure 2). Suppose the number of tasks in each work queue is $n_i, 1 \leq i \leq Q$. Whenever $\exists i$, $n_i < T_l$ or $n_i > T_u$, we execute our balancing algorithm among all the queues and the number of tasks in each queue becomes $n_i^* = \frac{\sum_{k=1}^{Q} n_k}{Q}, 1 \leq i \leq Q$, where T_l and T_u are two thresholds (we use *chunk-size* for T_l and the $W -$ *chunk-size* for T_u, where W is the maximum size of work queue).

In order to handle N collision queries simultaneously, we use several strategies, which are similar to the ones highlighted in Figure 1. First, we can repeat the single query above algorithm [16] for each query. However, this has two main disadvantages. First, the GPU kernel has to be called N times from the CPU, which is expensive for large N (which can be $\gg 10000$ for motion planning applications). Secondly, for each query, work queues are initialized with only one item (i.e. the root node of the BVTT), therefore the GPU's computational power cannot be fully exploited at the beginning of each query, as shown in the slow ascending part in Figure 1(a). Similarly, at the end of each query, most tasks are finished and some

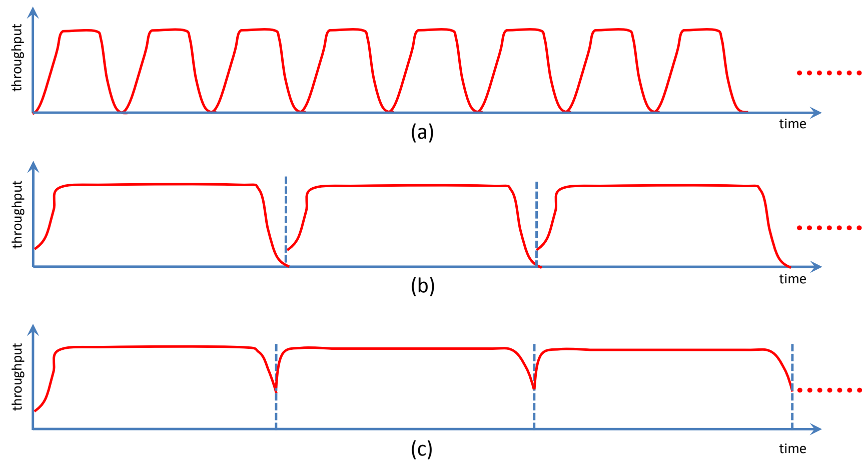

Fig. 1 Different strategies for parallel collision query using work queues. (a) Naive way: repeat the single collision query algorithm in [16] one by one; (b) Work queues are initialized by some BVTT root nodes and we repeat the process until all queries are performed. (c) is similar to (b) except that new BVTT root nodes are added to the work queues by the pump kernel, when there are not a sufficient number of tasks in the queue.

of the GPU cores become idle, which corresponds to the slow descending part in Figure 1(a).

As a result, we use the strategy shown in Figure 1(b): we divide the N queries into $\lceil \frac{N}{M} \rceil$ different sets each of size M with $M \leq N$ and initialize the work queues with M different BVTT roots for each iteration. Usually M cannot be N because we need to use $t \cdot M$ GPU global memory to store the transform information for the queries, where constant $t \leq \frac{\#global\ memory}{M}$ and we usually use $M = 50$. In this case, we only need to invoke the solution kernel $\lceil \frac{N}{M} \rceil$ times. The number of tasks available in the work queues changes more smoothly over time, with fewer ascending and descending parts, which implies higher throughput of the GPUs. Moreover, the work queues are initialized with many more tasks, which results in high performance at the beginning of each iteration. In practice, as nodes from more than one BVTT of different queries co-exist in the same queue, we need to distinguish them by representing each BVTT node by (x, y, i) instead of (x, y), where i is the index of collision query.

We can further improve the efficiency by using the pump operation (Algorithm 3 and Fig 2). That is, instead of initializing the work queues after it is completely empty, we add M BVTT root nodes of unresolved collision queries into the work queues when the number of tasks in it decreases to a threshold (we use $10 \cdot$ *chunk-size*). As a result, the few ascending and descending parts in Figure 1(b) can be further flattened as shown in Figure 1(c). Pump operation can reduce the timing overload of interrupting traversal kernels or copying data between global memory and shared memory, and therefore improve the overall efficiency of collision computation.

Algorithm 3 Traversal with Workload Balancing

```
1: task_kernel()
2: input abort signal $signal$, $N$ random configurations $\{\mathbf{q}_i\}_{i=1}^N$, BVH$_a$ for the robot and BVH$_b$
   for the obstacles
3: shared $WQ[] \equiv$ local work queue
4: initialize $WQ$ by tasks in global work queues
5: $\triangleleft$ traverse on work queues instead of BVTTs
6: loop
7:     $(x,y,i) \leftarrow pop(WQ)$
8:     if $overlap(\mathrm{BVH}_a(\mathbf{q}_i)[x],\mathrm{BVH}_b[y])$ then
9:         if $isLeaf(x)$ && $isLeaf(y)$ then
10:            update collision status of $i$-th query if $exactIntersect(\mathrm{BVH}_a(\mathbf{q}_i)[x],\mathrm{BVH}_b[y])$
11:        else
12:            $WQ[] \leftarrow (x,y,i)$'s children
13:        end if
14:    end if
15:    if $WQ$ is full or empty then
16:        $atomicInc(signal)$, break
17:    end if
18: end loop
19: return if $signal > 50\%Q$

1: balance_process()
2: copy local queue back to global work queue                          $\triangleleft$ manage_kernel
3: compute size of each work queue $n_i, 1 \leq i \leq Q$
4: if $\exists i, n_i < T_l || n_i > T_u$ then
5:     rearrange the tasks so that each queue has $n_i^* = \frac{\sum_{k=1}^{Q} n_k}{Q}$ tasks      $\triangleleft$ balance_kernel
6:     add more tasks in global queue if $\sum_{k=1}^{Q} n_k < T_{pump}$           $\triangleleft$ pump_kernel
7: end if
```

4.3 Analysis

In this section, we analyze the algorithms described above using the *parallel random access machine* (PRAM) model, which is a popular tool to analyze the complexity of parallel algorithms [11]. Of course, current GPU architectures have many properties that can not be described by PRAM model, such as SIMT, shared memory, etc. However, PRAM analysis can still provide some insight into GPU algorithm's performance.

Suppose we have n collision queries, which means that we need to traverse n BVTT of the same tree structure but with different geometry configurations. We also suppose the GPU has p parallel processors. For convenience, assume $n = ap, a \in \mathbb{Z}$. Let the complexity to traverse the i-th BVTT be $W(i)$, $1 \leq i \leq n$. Then the complexity of a sequential CPU algorithm is $T_S(n) = \sum_{i=1}^{n} W(i)$ and the complexity of Algorithm 1 would be $T_N(n) = \sum_{k=0}^{a-1} \max_{j=1}^{p} W(kp+j)$. If we sort $\{W(i)\}_{i=1}^{n}$ in ascending order and denote $W^*(i)$ as the i-th element in the new order, we can prove $\sum_{k=0}^{a-1} \max_{j=1}^{p} W(kp+j) \geq \sum_{k=1}^{a} W^*(kp)$. Therefore $T_N(n) \geq \sum_{k=1}^{a} W^*(kp)$. Moreover, it is obvious that $\sum_{i=1}^{n} W(i) \geq T_N(n) \geq \frac{\sum_{i=1}^{n} W(i)}{p}$, which means $T_N(n) = \Theta(T_S(n))$, i.e. $T_N(n)$ is work-efficient [11].

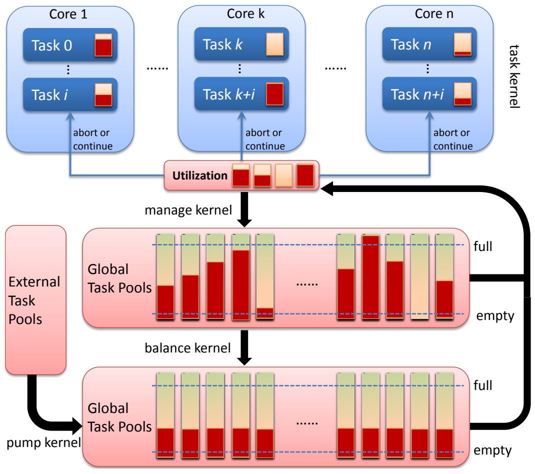

Fig. 2 Load balancing strategy for our parallel collision query algorithm. Each thread keeps its own local work queue in local memory. After processing a task, each thread is either able to run further or has an empty or full work queue and terminates. Once the number of GPU cores terminated exceeds a given threshold, the *manage kernel* is called and copies the local queues back onto global work queues. If no work queue has too many or too few tasks, the *task kernel* restarts. Otherwise, the *balance kernel* is called to balance the tasks among all queues. If there are not sufficient tasks in the queues, more BVTT root nodes will be 'pumped' in by the *pump kernel*.

According to the analysis in Section 4.1, we know that the expected complexity $\hat{W}(i)$ for i-th BVTT traversal in Algorithm 2 should be smaller than $W(i)$ because of the near-optimal traversing order. Moreover, the clustering strategy is similar to ordering different BVTTs, so that the BVTTs with similar traversal paths are arranged closely to each other and thus the probability is higher that they would be distributed on the same GPU core. Of course we can not implement ordering exactly because the BVTT traversal complexity is not known a priori. Therefore the complexity of Algorithm 2 is $T_P(n) \approx \sum_{k=1}^{a} \hat{W}^*(kp)$, with $\hat{W}^* \leq W^*$.

The complexity for Algorithm 3 is simple: $T_B(n) = \frac{\sum_{i=1}^{n} W(i)}{p} + B(n)$, where the first item is the timing complexity for BVTT traversal and the second item $B(n)$ is the timing complexity for balancing step. As $B(n) > 0$, the acceleration ratio of GPU with p-processors is less than p. We need to reduce the overload of balancing step to improve the efficiency of Algorithm 3.

Therefore, all three algorithms are work-efficient. If $B(n) = o(T_S(n))$, then $T_N(n) \geq T_P(n) \geq T_B(n)$ and Algorithm 3 is the most efficient one. If $B(n)$ is $\Theta(T_S(n))$, which means the overhead of balancing kernel is large, then it is possible to have $T_B(n) > T_P(n)$. Moreover, for large models, $W(i)$ would be quite different and the performance difference between three algorithms would be larger.

5 Implementation and Results

In this section, we present some details of the implementation and highlight the performance of our algorithm on different benchmarks. All the timings reported here were recorded on a machine using an Intel Core i7 3.2GHz CPU and 6GB

	piano	large-piano	helicopter	humanoid
#robot-faces	6540	34880	3612	27749
#obstace-faces	648	13824	2840	3495
DOF	6	6	6	38

Table 1 Geometric complexity of our benchmarks. Large-piano is a piano with more vertices and faces by subdividing the piano model.

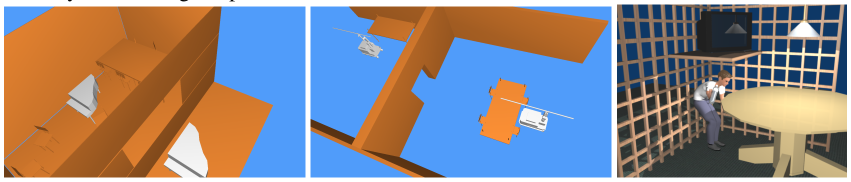
(a) piano (b) helicopter (c) humanoid

Fig. 3 Benchmarks for parallel collision queries.

memory. We implemented our collision and planning algorithms using CUDA on a NVIDIA GTX 480 GPU with 1GB of video memory.

We use the motion planning framework called *gPlanner* introduced in [23, 22], which uses PRM as the underlying planning algorithm as it is more suitable to exploit the multiple cores and data parallelism on GPUs. It can either compute a complete roadmap or we use a lazy version to perform a single motion planning query. We replace the collision module in gPlanner with the new algorithms described above. As observed in [23], more than 90% time of the planning algorithm is spent in collision queries, i.e. milestone computation step and local planning step.

In order to compare the performance of different parallel collision detection algorithms, we use the benchmarks highlighted in Figure 3. Their geometric complexities are highlighted in Table 1. For rigid body benchmarks, we generate $50,000$ random configurations and compute a collision-free path by using different variants of our parallel collision detection algorithm. For articulated model benchmark, we generate $100,000$ random configurations. For milestone computation, we directly use the collision detection algorithms. For local planning, we first need to unfold all the interpolated configurations: we denote the BVTT for the j-th interpolated query between the i-th local path as $\mathrm{BVTT}(i,j)$ and its node as (x,y,i,j). In order to avoid unnecessary computations, we first add BVTT root nodes with small j into the work queues, i.e. $(1,1,i,j) \prec (1,1,i',j')$, if $j < j'$. As a result, once a collision is found at $\mathrm{BVTT}(i,j_0)$, we need not to traverse $\mathrm{BVTT}(i,j)$ when $j > j_0$.

For Algorithm 1 and Algorithm 2, we further test the performance for different traversal sizes (32 and 128). Both algorithms give correct results when using a larger stack size (128). For smaller stack sizes, the algorithms will stop once the stack is filled. Algorithm 1 may report a collision when the stack overflows while Algorithm 2 returns a collision-free query. Therefore, Algorithm 1 may suffer from false positive errors while Algorithm 2 may suffer from false negative errors. We also compare the performance of Algorithm 1 and Algorithm 2 when the clustering algorithm described in Section 4.1 is used and when it is not.

The timing results are shown in Table 2 and Table 3. We can observe: (1) Algorithm 1 and Algorithm 2 both work better when local traverse stack is smaller

	Algorithm 1				Algorithm 2				Algorithm 3	
	32, no-C	32, C	128, no-C	128, C	32, no-C	32, C	128, no-C	128, C	traverse	balancing
piano	117	113	239	224	177	131	168	130	68	3.69
large-piano	409	387	738	710	613	535	617	529	155	15.1
helicopter	158	151	286	272	224	166	226	163	56	2.3
humanoid	2392	2322	2379	2316	2068	1877	2073	1823	337	106

Table 2 Comparison of different algorithms in milestone computation (timing in milliseconds). 32 and 128 are the different sizes used for the traversal stack; C and no-C means using pre-clustering and not using pre-clustering, respectively; timing of Algorithm 3 includes two parts: traversal part and balancing part.

	Algorithm 1				Algorithm 2				Algorithm 3	
	32, no-C	32, C	128, no-C	128, C	32, no-C	32, C	128, no-C	128, C	traverse	balancing
piano	1203	1148	2213	2076	1018	822	1520	1344	1054	34
large-piano	4126	3823	8288	7587	5162	4017	7513	6091	1139	66
helicopter	4528	4388	7646	7413	3941	3339	5219	4645	913	41
humanoid	5726	5319	9273	8650	4839	4788	9012	8837	6082	1964

Table 3 Comparison of different algorithms in local planning (timing in milliseconds). 32 and 128 are the different sizes used for the traversal stack; C and no-C means using pre-clustering and not using pre-clustering, respectively; timing of Algorithm 3 includes two parts: traversal part and balancing part.

and when pre-clustering is used. However for large models, traversal stack of size 32 may overflow and the collision results will be incorrect, which happens for the large-piano benchmarks in Table 2 and Table 3. Algorithm 1's performance will be terribly reduced when traverse stack size increases to 128 while Algorithm 2 does not change much. The reason is that Algorithm 2 uses per-packet stack, which is about 32 times less than using per-thread stack. Clustering and packet can result in a more than 50% speed-up. Moreover, the improvement of Algorithm 2 over Algorithm 1 is increased on larger models (large-piano) than on smaller models (piano). (2) Algorithm 3 is usually the fastest one among all the variations of the three algorithms. It can result in more than 5-10x increase in acceleration.

As observed in [23, 22], all these benchmarks are dominated by milestone computation and local planning steps as part of the overall parallel motion planning framework. The two parts take more than 50% running time in both the basic PRM and lazy PRM. Therefore, the overall planning algorithm can be improved by at least 40%-45%.

In Figure 4, we also show how the pump kernel increases the GPU throughput (i.e. the number of tasks available in work queues for GPU cores to fetch) in workload balancing based algorithm Algorithm 3. The maximum throughput (i.e. the maximum number of BV overlap tests performed by GPU kernels) increases from 8×10^4 to nearly 10^5 and the minimum throughput increases from 0 to 2.5×10^4. For piano and helicopter, we can compute a collision-free path from the initial to the goal configuration in in 879ms and 778ms separately using PRM or 72.79ms or 72.68ms using lazy PRM.

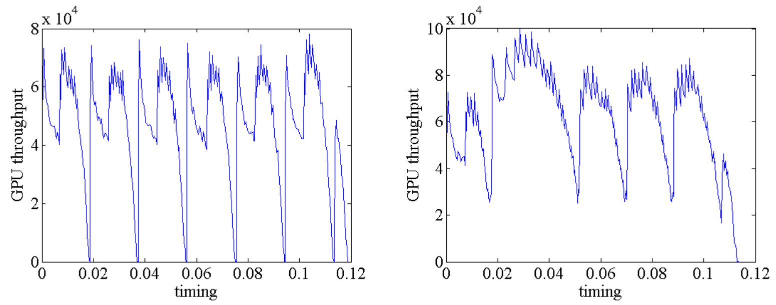

Fig. 4 GPU throughput improvement caused by pump kernel. Left figure shows the throughput without using the pump and right figure shows the throughput using the pump.

6 Conclusion and Future Work

In this paper, we introduce two novel parallel collision query algorithms for real-time motion planning on GPUs. The first algorithm is based on configuration-packet tracing, is easy to implement and can improve the parallel performance by performing more coherent traversals and reduce the memory consumed by traversal stacks. It can provide more than 50% speed-up as compared to simple parallel methods. The second algorithm is based on workload balancing, and decomposes parallel collision queries into fine-grained tasks of BVTT node operations. The algorithm uses a light-weight task-balancing strategy to guarantee that all GPU cores are fully loaded and achieves close to the peak performance on GPUs. It can provide 5-10X speed-up compared to simple parallel strategy. The overall performance of the GPU-based randomized planner also increases more than 50% when compared to the previous GPU planner.

There are many avenues for future work. We are interested in using more advanced sampling schemes with the planner to further improve its performance and allow us to work on motion planning problems with narrow passages. Furthermore, we would like to adjust the planner to generate smooth paths and integrate our planner with certain robots (e.g. PR2).

Acknowledgements 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.

References

1. Aila, T., Laine, S.: Understanding the efficiency of ray traversal on GPUs. In: High Performance Graphics, pp. 145–149 (2009)
2. Akinc, M., Bekris, K.E., Chen, B.Y., Ladd, A.M., Plaku, E., Kavraki, L.E.: Probabilistic roadmaps of trees for parallel computation of multiple query roadmaps. In: Robotics Research, *Springer Tracts in Advanced Robotics*, vol. 15, pp. 80–89. Springer Berlin / Heidelberg (2005)
3. Amato, N., Dale, L.: Probabilistic roadmap methods are embarrassingly parallel. In: International Conference on Robotics and Automation, pp. 688 – 694 (1999)
4. Banerjee, A., Ghosh, J.: Scalable clustering algorithms with balancing constraints. Data Mining and Knowledge Discovery **13**(3), 365–395 (2006)
5. Barraquand, J., Latombe, J.C.: Robot motion planning: A distributed representation approach. International Journal of Robotics Research **10**(6) (1991)

6. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Skadron, K.: A performance study of general-purpose applications on graphics processors using cuda. Journal of Parallel and Distributed Computing **68**(10), 1370–1380 (2008)
7. Foskey, M., Garber, M., Lin, M., Manocha, D.: A voronoi-based hybrid planner. In: International Conference on Intelligent Robots and Systems, pp. 55 – 60 (2001)
8. Gunther, J., Popov, S., Seidel, H.P., Slusallek, P.: Realtime ray tracing on GPU with BVH-based packet traversal. In: IEEE Symposium on Interactive Ray Tracing, pp. 113–118 (2007)
9. Harris, M.: Optimizing parallel reduction in CUDA. NVIDIA Developer Technology (2009)
10. Hoff, K., Culver, T., Keyser, J., Lin, M., Manocha, D.: Interactive motion planning using hardware accelerated computation of generalized voronoi diagrams. In: International Conference on Robotics and Automation, pp. 2931 – 2937 (2000)
11. JáJá, J.: An introduction to parallel algorithms. Addison Wesley Longman Publishing Co., Inc. (1992)
12. Kavraki, L., Svestka, P., Latombe, J.C., Overmars, M.: Probabilistic roadmaps for path planning in high-dimensional configuration spaces. IEEE Transactions on Robotics and Automation **12**(4), 566–580 (1996)
13. Kim, D., Heo, J.P., Huh, J., Kim, J., Yoon, S.E.: HPCCD: Hybrid parallel continuous collision detection using cpus and gpus. Computer Graphics Forum **28**(7), 1791–1800 (2009)
14. Kuffner, J., LaValle, S.: RRT-connect: An efficient approach to single-query path planning. In: International Conference on Robotics and Automation, pp. 995 – 1001 (2000)
15. Larsen, E., Gottschalk, S., Lin, M., Manocha, D.: Distance queries with rectangular swept sphere volumes. In: International Conference on Robotics and Automation, pp. 3719–3726 (2000)
16. Lauterbach, C., Mo, Q., Manocha, D.: gproximity: Hierarchical gpu-based operations for collision and distance queries. Computer Graphics Forum **29**(2), 419–428 (2010)
17. LaValle, S.M.: Planning Algorithms. Cambridge University Press (2006)
18. Lin, M., Manocha, D.: Collision and proximity queries. In: Handbook of Discrete and Computational Geometry, pp. 787–808. CRC Press, Inc. (2004)
19. Lindholm, E., Nickolls, J., Oberman, S., Montrym, J.: NVIDIA Tesla: A unified graphics and computing architecture. IEEE Micro **28**(2), 39–55 (2008)
20. Lozano-Perez, T., O'Donnell, P.: Parallel robot motion planning. In: International Conference on Robotics and Automation, pp. 1000–1007 (1991)
21. Owens, J.D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A.E., Purcell, T.: A survey of general-purpose computation on graphics hardware. Computer Graphics Forum **26**(1), 80–113 (2007)
22. Pan, J., Lauterbach, C., Manocha, D.: Efficient nearest-neighbor computation for GPU-based motion planning. In: International Conference on Intelligent Robots and Systems (2010). To appear
23. Pan, J., Lauterbach, C., Manocha, D.: g-planner: Real-time motion planning and global navigation using GPUs. In: AAAI Conference on Artificial Intelligence, pp. 1245–1251 (2010)
24. Pisula, C., Hoff, K., Lin, M.C., Manocha, D.: Randomized path planning for a rigid body based on hardware accelerated voronoi sampling. In: International Workshop on Algorithmic Foundation of Robotics, pp. 279–292 (2000)
25. Plaku, E., Bekris, K.E., Kavraki, L.E.: Oops for motion planning: An online open-source programming system. In: International Conference on Robotics and Automation, pp. 3711–3716 (2007)
26. Sud, A., Andersen, E., Curtis, S., Lin, M., Manocha, D.: Real-time path planning for virtual agents in dynamic environments. In: IEEE Virtual Reality, pp. 91–98 (2007)
27. Talamadupula, K., Benton, J., Schermerhorn, P.: Integrating a closed world planner with an open world. In: ICAPS Workshop on Bridging the Gap Between Task and Motion Planning (2009)
28. Tang, M., Manocha, D., Tong, R.: Mccd: Multi-core collision detection between deformable models. Graphical Models **72**(2), 7–23 (2010)
29. Zhang, L., Manocha, D.: A retraction-based RRT planner. In: International Conference on Robotics and Automation, pp. 3743–3750 (2008)