The International Journal of Robotics Research
31(2) 187–200

DOI: 10.1177/0278364911429335
ijr.sagepub.com

GPU-based parallel collision detection for fast 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 data-parallelism and multi-threaded capabilities. In order to take advantage of the 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 per second with our benchmarks, which is 10 times faster than prior GPU-based techniques. Moreover, we can compute collision-free paths for rigid and articulated models in less than 100 ms for many benchmarks, almost 50–100 times faster than current CPU-based PRM planners.

Keywords

path planning for manipulators, collision detection, real-time planning, simulation, virtual reality

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 degree-of-freedom (DOF) robots are based on randomized sampling. These include planning algorithms based on probabilistic roadmaps probabilistic roadmaps (PRMs) (Kavraki et al. 1996) and rapidly-exploring random trees rapidly-exploring random trees (RRTs) (Kuffner and LaValle 2000). These methods tend to approximate 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 (e.g. Willow Garage's PR2) tend to use live sensor data to generate a reasonably accurate model of 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, that is, the planning result of one subtask is used to construct the formulation of the following subtasks (Talamadupula et al. 2009). A fast and almost real-time planning algorithm is important for these applications.

It is known that a significant fraction (e.g. 90% or more) of randomized sampling algorithms is spent in collision checking. This includes checking whether 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 exploit the computational power and massive parallelism of commodity graphics processing units (GPUs) for almost real-time computation (Pan et al. 2010a,b). 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 compared to CPUs. As

Department of Computer Science, University of North Carolina at Chapel Hill, USA

Corresponding author:
Jia Pan, Department of Computer Science, University of North Carolina at Chapel Hill, Campus Box 3175, Sitterson Hall, Chapel Hill, NC 27599-3175, USA.
Email: panj@cs.unc.edu

a result, we need to design appropriate parallel collision and planning algorithms that can map well to GPUs.

1.1. 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 use 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 bounding volume overlap tests 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 10 times faster than prior GPU-based collision checking algorithms. We also use our collision-checking algorithm for GPU-based motion planners of high-DOF rigid and articulated robots. The resulting planner can compute collision-free paths in less than 100 ms for various benchmarks and appears to be 50–100 times faster than CPU-based PRM planners.

The rest of the paper is organized as follows. We survey related work on real-time motion planning and collision-detection algorithms in Section 2. Section 3 gives an overview of our approach and we present our parallel algorithm for collision queries in Section 4. We highlight the performance of our algorithm against different benchmarks in Section 5. A preliminary version of this work was presented in Pan and Manocha (2011).

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 LaValle (2006). Many parallel algorithms have also been proposed for motion planning by utilizing the properties of configuration spaces (Lozano-Perez and O'Donnell 1991). The distributed representation (Barraquand and Latombe 1991) can be easily parallelized. In order to deal with high dimensional or difficult planning problems, distributed sampling-based techniques have been proposed (Plaku et al. 2007).

The computational power of many-core GPUs has been used for many geometric and scientific computations (Owens et al. 2007). The rasterization capabilities of a GPU can be used for real-time motion planning of low DOF robots (Hoff et al. 2000; Sud et al. 2007) or improve sample generation in narrow passages (Pisula et al. 2000; Foskey et al. 2001). Recently, GPU-based parallel motion planning algorithms have been proposed for rigid models (Pan et al. 2010a,b).

2.2. Parallel collision queries

Some of the widely used algorithms for collision checking are based on *bounding volume hierarchies* (BVH), such as k-DOP trees, OBB trees, AABB trees, etc. (Lin and Manocha 2004). Recent developments include parallel hierarchical computations on multi-core CPUs (Kim et al. 2009; Tang et al. 2010) and GPUs (Lauterbach et al. 2010). 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 (Lauterbach et al. 2010) use work queues to parallelize the computation on multiple cores. All of these approaches are primarily designed to parallelize a single collision query.

The capability to efficiently perform high numbers of collision queries is essential in motion-planning algorithms, for example, multiple 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 (Amato and Dale 1999; Akinc et al. 2005; Pan et al. 2010a,b). Since 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 new 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 a theoretical peak performance of several teraFLOP/s for single-precision computations and hundreds of gigaFLOP/s for double-precision computations. This peak performance is significantly higher compared to current multi-core CPUs,

thus outpacing CPU architectures (Lindholm et al. 2008) for a relatively modest cost of $400 to $500. However, GPUs have different architectural characteristics and memory hierarchy, which 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 a total of 480 processors while the GTX 280 has 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 in GPUs is quite different from that of CPUs and the cache sizes in the GPUs are considerably smaller. Moreover, each GPU core can handle several separate tasks in parallel and switch between different tasks in the hardware when one of them is waiting for a memory operation to complete. This hardware multi-threading approach is thus designed to hide memory access latency. Thirdly, all GPU threads are logically grouped in blocks with a per-block high-speed 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, multiple GPU threads are physically managed and scheduled in the *single-instruction, multiple-thread* (SIMT) way, that is, threads are grouped into chunks and each chunk executes one common instruction at a time. In contrast to *single-instruction multiple-data* (SIMD) schemes, the SIMT scheme allows each thread to have its own instruction address counter and register state, and therefore, freedom to branch and execute independently. However, 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 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 (Gunther et al. 2007). 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 an appropriate size for the 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 literature. The size of a chunk (*chunk size* or *warp size*) is 32 on current NVIDIA GPUs (e.g. the GTX 280 and 480).

block The logical collection of GPU threads that can 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. The ith BVH node is denoted as $\text{BVH}_a[i]$ and its child nodes are $\text{BVH}_a[2i]$ and $\text{BVH}_a[2i+1]$ with $1 \le i \le 2^{L-1}-1$. The nodes at the lth level of a BVH tree are represented as $\text{BVH}_a[k], 2^l \le k \le 2^{l+1}-1$ with $0 \le l < L$. 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.

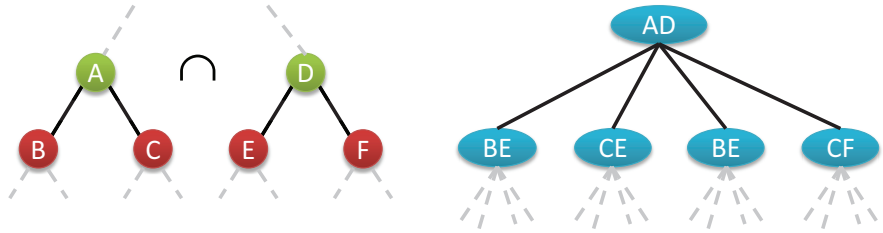
BVTT$_{a,b}$ The *bounding volume test tree* (BVTT) represents a recursive collision query traversal between two objects a, b. It is a 4-ary tree, whose nodes are ordered in the breadth-first order starting from the root node. The ith BVTT node is denoted as $\text{BVTT}_{a,b}[i] \equiv (\text{BVH}_a[m],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

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

The relationship between BVH trees and BVTT is also shown in Figure 1. Notice that given the BVHs of two geometric models, the BVTT is completely determined using those BVHs and is independent of the actual configuration of each model. The model configurations only affect the actual traversal path of the BVTT.

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 (Larsen et al. 2000). The simplest parallel algorithms used to perform



(a) Two BVH trees (b) BVTT tree

Fig. 1. BVH and BVTT: (a) shows two BVH trees and (b) shows the BVTT tree for the collision checking between the two BVH trees.

Algorithm 1 Simple parallel collision checking; such approaches are widely used on multi-core CPUs

```
Input: N random configurations {q_i}_{i=1}^N, BVH_a for the robot and BVH_b for the obstacles
Output: return whether one configuration is in free space or not
t_id ← thread id of the current thread
q ← q_{t_id}
◁ traversal stack S[] is initialized with root nodes
shared/global S[] ≡ local traversal stack
S[] ←BVTT[1] ≡(BVH_a(q)[1],BVH_b[1])
◁ traverse BVTT for BVH_a(q) and BVH_b
loop
  (x, y) ← pop(S).
  if overlap(BVH_a(q)[x],BVH_b[y]) then
    if !isLeaf(x) && !isLeaf(y) then
      S[] ←(2x, 2y), (2x, 2y+1), (2x+1, 2y), (2x+1, 2y+1)
    end if
    if isLeaf(x) && !isLeaf(y) then
      S[] ←(2x, 2y), (2x, 2y+1)
    end if
    if !isLeaf(x) && isLeaf(y) then
      S[] ←(2x, 2y), (2x+1, 2y)
    end if
    if isLeaf(x) && isLeaf(y) && exactIntersect(BVH_a(q)[x],BVH_b[y]) then
      return collision
    end if
  end if
end loop
return collision-free
```

multiple collision queries are based on each thread traversing the BVTT for one configuration and checking whether the given configuration is in free space or not. Such a simple parallel algorithm is shown 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 traversal stack for the BVTT. The stack size should be at least $3(\log_4(N_a)+\log_4(N_b))$ to avoid stack overflow, where N_a and N_b are the numbers of primitive triangles of BVH_a and BVH_b, respectively. The stack can be implemented using global memory or shared memory. Global memory access on GPUs tends to be slow, which affects BVTT traversal. Shared memory access is much faster but it may be too small to hold the large stack for complex geometric models composed of thousands of polygons. Moreover, increasing shared memory usage will limit the extent of parallelism. Second, different threads may traverse the BVTT tree with incoherent patterns: there are many branching decisions performed during the traversal (e.g. **loop**, **if**, and **return** in the pseudocode) 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 other threads may have finished the traversal early and are idle because there is no BV overlap or a primitive collision has already been detected. 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 the length of these paths can be large for different configurations of a robot. As a result, differences in the workloads of different threads 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 incoherent traversal. Similar issues also arise during local planning when each thread determines whether two milestones can be joined by a collision-free path by checking for 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 for local planning computations. The first algorithm uses clustering techniques 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 a 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 additional computations [e.g. retraction for handling narrow passages (Zhang and Manocha 2008)]. The second method can provide 5–10 times 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 (Gunther et al. 2007; Aila and Laine 2009). 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 some 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), the rays in a packet should be similar to each another, that is, have similar traversal paths with few differing branches. For ray tracing, the packet construction is simple: as shown in Figure 2, rays passing through the same pixel on the image space make a natural packet. We extend this idea to parallel collision checking and refer to our algorithm as the *multiple configuration-packet* method.

The first challenge is to cluster similar collision queries or configurations into groups because, unlike ray tracing, there are no natural packet construction rules for collision queries. In some cases, the sampling scheme (e.g. 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 $\mathcal{X}$

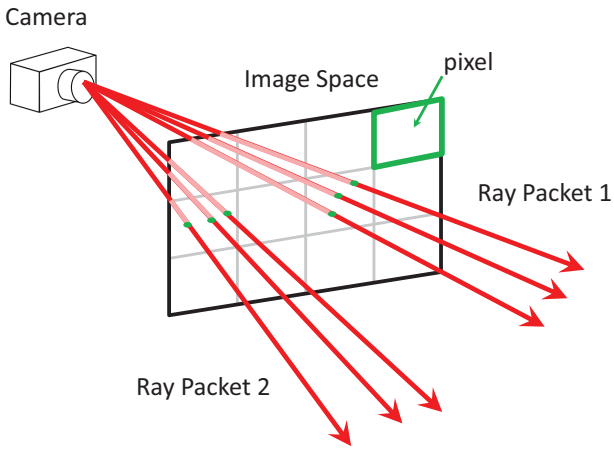


Fig. 2. Ray packets for faster ray tracing. Neighboring rays constitute a ray packet and this spatial coherence is exploited for fast intersection tests.

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: $|C_k| = \text{chunk size}, 1 \leq k \leq K$. That is, each cluster should fit in one chunk on a GPU, except for the last cluster, and $K = \lceil \frac{N}{\text{chunk size}} \rceil$. Using the formulation of k-means, the clustering problem can be formally described as:
Compute $K = \lceil \frac{N}{\text{chunk size}} \rceil$ 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 $|C_k| = \text{chunk size}, 1 \leq k \leq K$. In our knowledge, there are no clustering algorithms designed for this specific problem. One possible solution is to use *clustering with balancing constraints* (Banerjee and Ghosh 2006), which has additional constraints $|C_k| \geq m, 1 \leq k \leq K$, where $m \leq \frac{N}{K}$.

Instead of solving Equation (1) exactly, we use a simpler clustering scheme to compute an approximate solution. First, we use the k-means algorithm to cluster the N queries into C clusters, which can be implemented efficiently on GPUs (Che et al. 2008). Next, for the kth cluster of size S_k, we divide it into $\lceil \frac{S_k}{\text{chunk size}} \rceil$ subclusters, each of which corresponds to a *configuration packet*. This simple method has some disadvantages. For example, the number of clusters is

$$\sum_{k=1}^{C} \left\lceil \frac{S_k}{\text{chunk size}} \right\rceil \geq K = \left\lceil \frac{N}{\text{chunk size}} \right\rceil$$

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.

The configuration clustering method is illustrated in Figure 3. The green points are random configuration

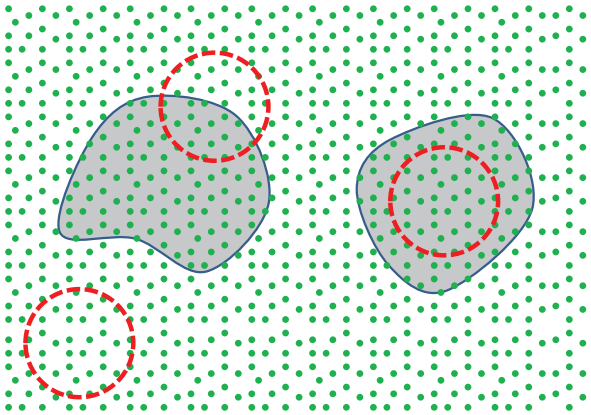


Fig. 3. Multiple configuration packets for parallel collision detection.

samples in $\mathcal{C}$-space. The gray areas are $\mathcal{C}$-obstacles. Configurations adjacent in $\mathcal{C}$-space are clustered into configuration packets (red circles). Some packets are completely in free space; some packets are completely within $\mathcal{C}$-obstacles; some packets are near boundaries of $\mathcal{C}$-obstacles. Configurations in the same packet have similar BVTT traversal paths and are mapped to the same warp on a GPU.

Next we map each configuration packet to a single chunk. Threads within one packet will traverse the BVTT synchronously, that is, 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 child nodes and tests the BVs for overlap to determine the remaining traversal order, that is, 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 most threads in the packet. We also push other children into the packet's traversal stack. If 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 the chunk size and results in higher parallel efficiency.

The traversal order decision rule is shown in Figure 4. The four trees in the first row are the BVTT trees for configurations in the same chunk. For convenience, we represent the BVTT as a binary tree instead of a 4-ary tree. The 1 or 0 at each node represents whether the BV-overlap or exact intersection test executed at that node is in collision or collision free. The red edges are the edges visited by the BVTT traversal algorithm and the indices on these edges represent the traversal order. In this case, the four different configurations have traversal paths of length 5, 5, 5, and 6. The leaf nodes with a red 1 are locations where collisions are detected and the traversal stops. The tree in the second row shows the synchronous BVTT traversal order determined by our heuristic rule, which needs to visit 10 edges to detect the collisions of all four configurations.

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 a nearly identical order, this implies that other threads in the same packet might not traverse the subtree either. We define the probability that the subtree rooted at (x, y) will be traversed by one thread as

$$p_{x,y} = \frac{\text{number of overlap threads}}{\text{packet size}}.$$

For any traversal pattern P for a BVTT, the probability that it is carried on by a BVTT traversal will be

$$p_P = \prod_{(x,y)\in P} p_{x,y}.$$

As a result, our new traversal strategy guarantees that traversal patterns with higher traversal probabilities will have shorter traversal lengths, and will therefore minimize the overall path for the packet.

The decision about which child node is a candidate for the next traversal step is computed using sum reduction (Harris 2009), which can compute the sum of n items in parallel with $\mathcal{O}(\log(n))$ complexity. Each thread writes a 1 in its own location in the shared memory if it detects an overlap in a child and 0 otherwise. The sum of the memory locations is computed in five 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.

4.2. *Parallel collision query with workload balancing*

Both Algorithm 1 and Algorithm 2 use the per-thread per-query strategy, which is relatively easy to implement. However, when idle threads wait for busy threads or when the execution path of threads diverges, the parallel efficiency of the GPUs reduces. Algorithm 2 can alleviate 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 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.

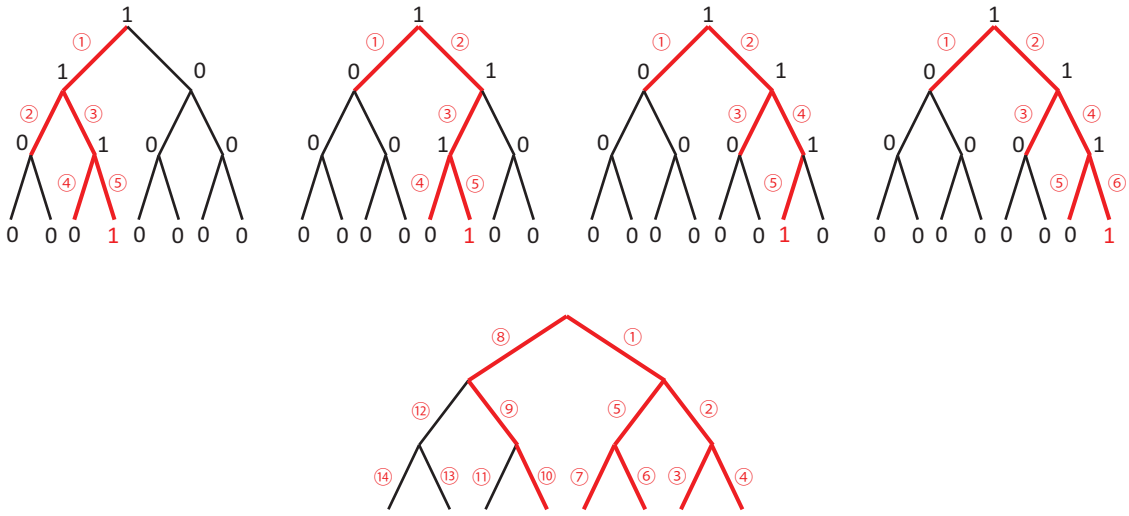


Fig. 4. Synchronous BVTT traversal for packet configurations.

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, thereby balancing the load among them, than to distribute the collision query tasks.

All the tasks are stored in 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 (Lauterbach et al. 2010), the tasks are in the form of BVTT nodes (x, y). Each thread will fetch some tasks from a 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 is a sufficient number of tasks in all the work queues. However, during the 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 5). 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, *manage kernel* is called, which copies the local queues back onto global work queues. If no work queue has too many or too few tasks, *task kernel* restarts. Otherwise, *balance kernel* is called to balance the tasks among all the queues. If there are insufficient tasks in the queues, more BVTT root nodes will be 'pumped' in by the *pump kernel*.

Suppose the number of tasks in each work queue is

$$n_i, \quad 1 \leq i \leq Q.$$

Whenever there exists i such that $n_i < T_l$ or $n_i > T_u$, we execute our balancing algorithm among all the queues and the number of tasks in each queue becomes

$$n_i^* = \frac{\sum_{k=1}^{Q} n_k}{Q}, \quad 1 \leq i \leq Q,$$

where T_l and T_u are two thresholds (we use the chunk size for T_l and W – chunk size for T_u, where W is the maximum size of a work queue).

In order to handle N collision queries simultaneously, we use several strategies, which are highlighted and compared in Figure 6. First, we can repeat the single query algorithm (Lauterbach et al. 2010) introduced above 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 $\gg 10,000$ for sample-based motion planning). Secondly, for each query, the 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 of the curve in Figure 6(a). Similarly, at the end of each query, most tasks have finished and some of the GPU cores are idle, which corresponds to the slow descending part of the curve in Figure 6(a).

As a result, we use the strategy shown in Figure 6(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

Algorithm 2 Multiple configuration-packet traversal

```
1: Input: N random configurations {q_i}_{i=1}^N, BVH_a for the robot and BVH_b for the obstacles
2: t_id ← thread id of current thread
3: q ← q_{t_id}
4: shared CN[]≡ shared memory for children node
5: shared TS[]≡ local traversal stack
6: shared SM[]≡ memory for sum reduction
7: if overlap(BVH_a(q)[1], BVH_b[1]) is false for all threads in chunk then
8:    return
9: end if
10: (x, y) =(1, 1)
11: loop
12:    if isLeaf(x) && isLeaf(y) then
13:       if exactIntersect(BVH_a(q)[x],BVH_b[y]) then
14:          update collision status of q
15:       end if
16:       if TS is empty then
17:          break
18:       end if
19:       (x, y) ← pop(TS)
20:    else
21:       ◁ decide the next node to be traversed
22:       CN[] ←(x, y)'s children nodes
23:       for all (x_c, y_c) ∈ CN do
24:          ◁ compute the number of threads that detect overlap at node (x_c, y_c)
25:          write overlap(BVH_a(q)[x_c],BVH_b[y_c]) (0 or 1) into SM[t_id] accordingly
26:          compute local summation s_c in parallel by all threads in chunk
27:       end for
28:       if max_c s_c > 0 then
29:          ◁ select the node that is overlapped in the most threads
30:          (x, y) ← CN[argmax_c s_c] and push others into TS
31:       else
32:          ◁ select the node from the top of stack
33:          if TS is empty then
34:             break
35:          end if
36:          (x, y) ← pop(TS)
37:       end if
38:    end if
39: end loop
```

Algorithm 3 Traversal with workload balancing: task kernel

```
1: Input: abort signal signal, N random configurations {q_i}_{i=1}^N, BVH_a for the robot and BVH_b for the obstacles
2: shared WQ[] ≡ local work queue
3: initialize WQ by tasks in global work queues
4: ◁ traverse on work queues instead of BVTTs
5: loop
6:    (x, y, i) ← pop(WQ)
7:    if overlap(BVH_a(q_i)[x],BVH_b[y]) then
8:       if isLeaf(x) && isLeaf(y) then
9:          if exactIntersect(BVH_a(q_i)[x],BVH_b[y]) then
10:            update collision status of ith query
11:         end if
12:      else
13:         WQ[] ←(x, y, i)'s children
14:      end if
15:   end if
16:   if WQ is full or empty then
17:      atomically increment signal, break
18:   end if
19: end loop
20: return if signal > 50%Q
```

because we need to use $t \cdot M$ GPU global memory to store the transform information for the queries, where constant

$$t \leq \frac{\text{size of global memory}}{M}$$

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 for a collision query. The details for this strategy are shown in Algorithm 3.

We can further improve the efficiency by using the pump operation, as shown in Algorithm 4 and Figure 5. That is, instead of initializing a work queue when it is completely empty, we add M BVTT root nodes of unresolved collision queries into a work queue when the number of tasks in it decreases to a threshold (we use $10 \times$ chunk size). As a result, the few ascending and descending parts in Figure 6(b) can be further flattened as shown in Figure 6(c). The 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.

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 for analyzing the complexity of parallel algorithms (JáJá 1992). Of course, current GPU architectures have many properties that cannot be described by the PRAM model, such as SIMT, shared memory, etc.

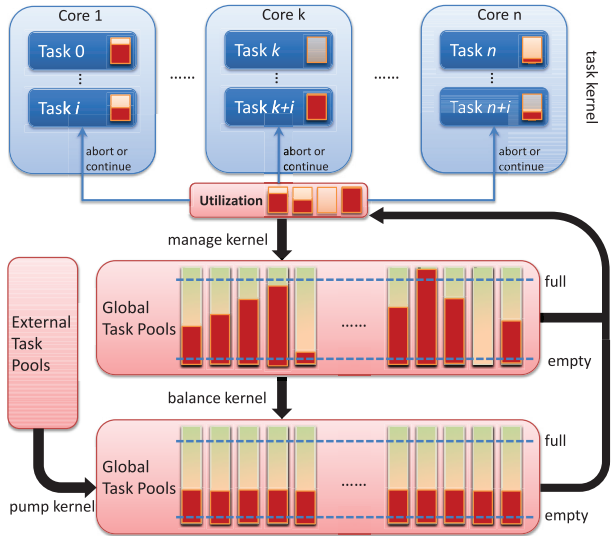


Fig. 5. Load balancing strategy for our parallel collision query algorithm.

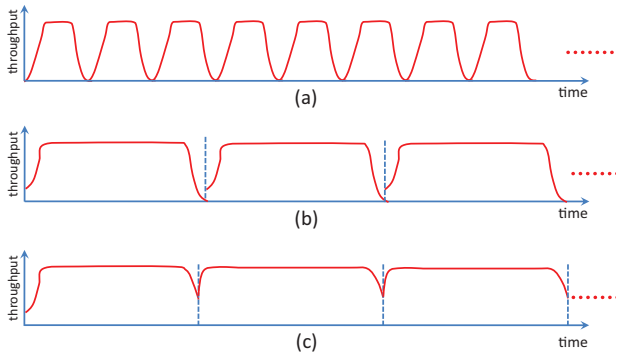


Fig. 6. Different strategies for a parallel collision query using work queues. (a) Naive way: repeat the single collision query algorithm one by one; (b) work queues are initialized by BVTT root nodes and we repeat the process until all queries are performed; (c) is similar to (b) except that new BVTT root nodes are added to the work queues by the pump kernel, when there are insufficient tasks in the queue.

Algorithm 4 Traversal with workload balancing: manage kernel

1: Input: Q global work queues
2: copy local queues on shared memory back to Q global work queues on global memory
3: compute the number of tasks in each work queue $n_i, 1 \le i \le Q$
4: compute the number of tasks in all queues $n = \sum_{k=1}^{Q} n_k$
5: **if** $n < T_{pump}$ **then**
6: call pump kernel: add more tasks in global queue from unresolved collision queries
7: **else if** $\exists i, n_i < T_l || n_i > T_u$ **then**
8: call balance kernel: rearrange the tasks so that each queue has $n_i^* = \frac{\sum_{k=1}^{Q} n_k}{Q}$ tasks
9: **end if**
10: call task kernel again

However, PRAM analysis can still provide some insight into a GPU algorithm's performance.

Suppose we are given n collision queries, which means that we need to traverse n BVTTs of the same tree structure but with different geometry configurations. We denote the complexity of the serial algorithm as $T_S(n)$, the complexity of the naive parallel algorithm (Algorithm 1) as $T_N(n)$, the complexity of the configuration-packet algorithm (Algorithm 2) as $T_P(n)$, and the complexity of the workload-balancing algorithm (Algorithm 4) as $T_B(n)$. Then we have the following result:

Lemma 1. $\Theta(T_S(n)) = T_N(n) \ge T_P(n) \ge T_B(n)$.

Remark. In parallel computing, we say a parallel algorithm is *work efficient*, if its complexity $T(n)$ is bounded both above and below asymptotically by $S(n)$, the complexity of its serial version, that is, $T(n) = \Theta(S(n))$ (JáJá 1992). In other words, Lemma 1 means that all three parallel collision algorithms are work efficient, but that workload balancing is the most efficient and that the configuration-packet algorithm is more efficient than the naive parallel scheme.

Proof. Let the complexity of traversing the ith BVTT be $W(i)$, $1 \le i \le n$. Then the complexity of a sequential CPU algorithm is $T_S(n) = \sum_{i=1}^{n} W(i)$. For GPU-based parallel algorithms, we assume that the GPU has p processors or cores. For convenience, we assume $n = ap, a \in \mathbb{Z}$.

For a naive parallel algorithm (Algorithm 1), each processor executes the BVTT traversal independently and the overall performance is determined by the most time-consuming BVTT traversal. Therefore, its complexity is

$$T_N(n) = \sum_{k=0}^{a-1} \max_{j=1}^{p} W(kp+j).$$

If we sort $\{W(i)\}_{i=1}^{n}$ in ascending order and denote $W^*(i)$ as the ith element in the new order, we have

$$\sum_{k=0}^{a-1} \max_{j=1}^{p} W(kp+j) \ge \sum_{k=1}^{a} W^*(kp). \tag{2}$$

To prove this, we start from $a = 2$. In this case, the summation $\max_{j=1}^{p} W(j) + \max_{j=1}^{p} W(p+j)$ is at a minimum when $\min\{W(p+1), \cdots, W(2p)\} \ge \max\{W(1), \cdots, W(p)\}$. Otherwise, exchanging the minimum value in $\{W(p+1), \cdots, W(2p)\}$ and the maximum value in $\{W(1), \cdots, W(p)\}$ will increase the summation. For $a > 2$, using a similar technique, we can show that the minimum of $\sum_{k=0}^{a-1} \max_{j=1}^{p} W(kp+j)$ happens when $\min_{k=jp+1}^{(j+1)p} \{W(k)\} \ge \max_{(j-1)p+1}^{jp} \{W(k)\}$, $1 \le j \le a-1$. This is satisfied by the ascending sorted result W^* and inequality (2) is proved.

Moreover, it is obvious that $\sum_{i=1}^{n} W(i) \geq T_N(n) \geq \frac{\sum_{i=1}^{n} W(i)}{p}$. Then we obtain

$$T_S(n) \geq T_N(n) \geq \max\left(\frac{T_S(n)}{p}, \sum_{k=1}^{a} W^*(kp)\right),$$

which implies $T_N(n) = \Theta(T_S(n))$.

According to the analysis in Section 4.1, we know that the expected complexity $\hat{W}(i)$ for the ith BVTT traversal in the configuration-packet method (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. In practice, we cannot implement such an ordering exactly because the complexity of a BVTT traversal is not known a priori. Therefore the complexity of Algorithm 2 is

$$T_P(n) \approx \sum_{k=1}^{a} \hat{W}^*(kp),$$

with $\hat{W}^* \leq W^*$. As a result, we have $T_P(n) \leq T_N(n)$.

The complexity for the workload-balancing method (Algorithm 4) can be given as:

$$T_B(n) = \frac{\sum_{i=1}^{n} W(i)}{p} + B(n),$$

where the first item is the timing complexity for a BVTT traversal and the second item $B(n)$ is the timing complexity for a balancing step. As $B(n) > 0$, the acceleration ratio of a GPU with p processors is less than p. We need to reduce the load of a balancing step to improve the efficiency of Algorithm 4. If a balancing step is implemented efficiently, that is, if $B(n) = o(T_S(n))$, we have $T_N(n) \geq T_P(n) \geq T_B(n)$. □

5. Implementation and results

In this section, we present the details of the implementation and highlight the performance of our algorithm with different benchmarks. All the timings reported here were recorded on a machine using an Intel Core i7 3.2 GHz CPU and 6 GB memory. We implemented our collision and planning algorithms using CUDA on a NVIDIA GTX 480 GPU with 1 GB of video memory.

5.1. GPU-based planner

We use the motion-planning framework called *gPlanner* introduced in Pan et al. (2010a,b), which uses PRM as the underlying planning algorithm as it is more suitable for exploiting the multiple cores and data parallelism of the GPUs. The planner is completely implemented on GPUs to avoid the expensive data transfer between the CPU and a GPU.

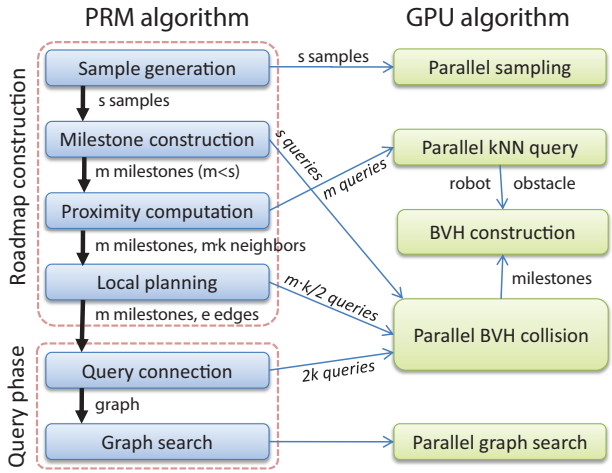


Fig. 7. Overview of the GPU-based real-time planner (Pan et al. 2010a).

The PRM algorithm has two phases: roadmap construction and query phase, whose basic flowchart is shown on the left part of Figure 7. 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 on the right side of Figure 7.

We first use the MD5 cryptographic hash function (Tzeng and Wei 2008) to generate random samples for each thread independently. For each sample generated, we need to check whether it is a milestone, that is, it does not collide with the obstacles using BVH trees (Lauterbach et al. 2009) and that it exploits the GPU parallelism.

For each milestone, we perform a k-nearest neighbor query to compute the nearest neighbors and construct a roadmap for the $\mathcal{C}$-space. In practice, an exact k-nearest neighbor search can be slow in a high-dimensional $\mathcal{C}$-space. As a result, we use an approximate k-nearest neighbor search based on *locality-sensitive hashing* (LSH) and can construct the k-nearest graph in time linear to the number of milestones (Pan et al. 2010a). Finally, we use local planning algorithms to check the validity of the k-nearest graph edges and construct a roadmap in the $\mathcal{C}$-space.

Once the roadmap is constructed, we connect initial-goal configurations to the multiple queries to the roadmap. Finally, we perform a parallel graph search on the roadmap to compute collision-free paths. For single query cases, we use a lazy version of PRM: instead of computing the entire roadmap, we delay the expensive local planning and localize it to only a few edges.

5.2. Implementation

As part of our implementation, we replace the collision-detection module in gPlanner with the new algorithms described above. As observed in Pan et al. (2010b), more

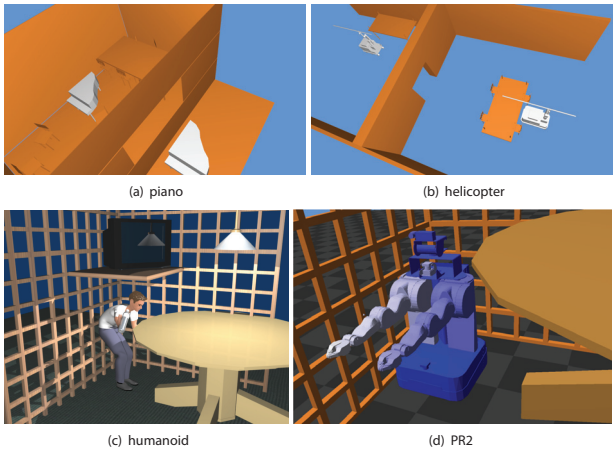


Fig. 8. Benchmarks used in our experiments.

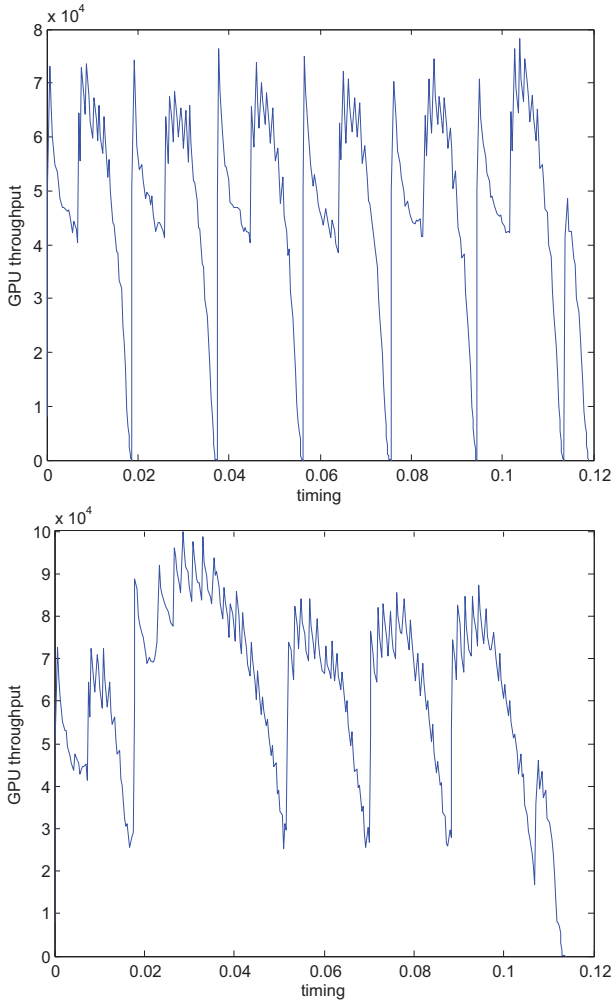


Fig. 9. GPU throughput improvement caused by pump kernel. The left figure shows the throughput without using the pump kernel and the right figure shows the throughput using the pump kernel.

than 90% of the time spent by a motion-planning algorithm is spent in collision queries, that is, milestone computation and local planning.

In order to compare the performance of different parallel collision-detection algorithms, we use the benchmarks shown in Figure 8. The geometric complexity of these benchmarks is shown 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 an articulated-model benchmark, we generate 100,000 random configurations. For milestone computation, we directly use our collision-detection algorithm. For local planning, we first need to unfold all the interpolated configurations: we denote the BVTT for the jth interpolated query for the ith local path as BVTT(i,j) and its node as (x,y,i,j). In order to avoid unnecessary computation, we first add BVTT root nodes with small j into the work queues, that is $(1,1,i,j) \prec (1,1,i',j')$, if$j < j'$. As a result, once a collision is computed at BVTT(i,j_0), we need not traverse BVTT(i,j) when $j > j_0$.

For Algorithm 1 and Algorithm 2, we further test the performance for different traversal sizes (i.e. 32 and 128). Both algorithms give correct results when using a larger stack size (i.e. 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 Tables 2 and 3. We observe: (1) Algorithm 1 and Algorithm 2 both work better when the local traversal stack is smaller and the pre-clustering technique is used. However, for large models, a traversal stack of size 32 may result in overflows and the collision results can be incorrect, which happens for the large-piano benchmarks, as shown in Tables 2 and 3. Algorithm 1's performance is considerably reduced when the size of the traversal stack increases to 128. This is because Algorithm 2 uses a per-packet stack, which is about 32 times smaller than using a per-thread stack. Moreover, clustering and configuration-packet traversal can result in more than a 50% speed-up. Moreover, the improvement in the performance of Algorithm 2 over Algorithm 1 is greater for complex models (e.g. the large-piano). (2) Algorithm 4 is usually the fastest of the three algorithms. It can result in more than 5–10 times speed-up over the other methods.

As observed in Pan et al. (2010a,b), the performance of the planner in these benchmarks is dominated by milestone computation and local planning. Based on the novel collision-detection algorithm, the performance of PRM and lazy PRM planners can be improved by at least 40%–45%.

In Figure 9, we also show how the pump kernel increases the GPU throughput (i.e. the number of tasks available

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

	piano	large-piano	helicopter	humanoid	PR2
# robot faces	6,540	34,880	3,612	27,749	31,384
# obstacle faces	648	13,824	2,840	3,495	3,495
DOF	6	6	6	38	12 (one arm)

Table 2. Comparison of different algorithms in milestone computation (times in milliseconds). The sizes used for the traversal stack were 32 and 128; C and no-C indicate whether pre-clustering was used or not, respectively; the times for Algorithm 4 include traversal and balancing.

	Algorithm 1				Algorithm 2				Algorithm 4	
	32, no-C	32, C	128, no-C	128, C	32, no-C	32, C	128, no-C	128, C	traversal	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	2,392	2,322	2,379	2,316	2,068	1,877	2,073	1,823	337	106

Table 3. Comparison of different algorithms in local planning (times in milliseconds). The sizes used for the traversal stack were 32 and 128; C and no-C indicate whether pre-clustering was used or not, respectively; the times for Algorithm 4 include traversal and balancing.

	Algorithm 1				Algorithm 2				Algorithm 4	
	32, no-C	32, C	128, no-C	128, C	32, no-C	32, C	128, no-C	128, C	traversal	balancing
piano	1,203	1,148	2,213	2,076	1,018	822	1,520	1,344	1,054	34
large-piano	4,126	3,823	8,288	7,587	5,162	4,017	7,513	6,091	1,139	66
helicopter	4,528	4,388	7,646	7,413	3,941	3,339	5,219	4,645	913	41
humanoid	5,726	5,319	9,273	8,650	4,839	4,788	9,012	8,837	6,082	1,964

in work queues for GPU cores to fetch) in the workload-balancing Algorithm 4. 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 the piano and helicopter models, we can compute a collision-free path from the initial configuration to the goal configuration in 879 ms and 778 ms, respectively, using PRM or 72.79 ms or 72.68 ms, respectively, using lazy PRM.

5.3. Articulated models

Our parallel algorithms can be directly applied to articulated models. In this case, checking for self-collisions among various links of a robot adds to the overall complexity. We use a model of the PR2 robot as an articulated benchmark. The PR2 robot model has 65 links and 75 DOFs. We only allow one arm (i.e. 12 DOFs) to move. A naive approach would involve exhaustive self-collision checking, and reduces to checking $65 \times (65-1)/2 = 2,080$ self-collisions among the links for each collision query. As shown in Table 4, the GPU-based planner takes more than 10 seconds for the PR2 benchmark when performing exhaustive self-collision, though it is still much faster than the CPU-based implementation.

However, exhaustive self-collision checking is usually not necessary for physical robots, because the joint limits can filter out many of the self-collisions. The common method is to manually set some link pairs that need to be checked for self-collisions. This strategy can greatly reduce the number of pairwise checks. As shown in Table 4, we can compute a collision-free path for the PR2 model in less than 1 second, which can be further reduced to 300 ms if the number of samples is reduced to 500. The collision-free path calculated by our planner is shown in Figure 10.

6. Conclusion and future work

In this paper, we introduced two novel parallel collision query algorithms for real-time motion planning on GPUs.

Table 4. Collision timing on the PR2 benchmark (times in milliseconds). We use 1,000 samples, 20 nearest neighbors, and discrete local planning with 20 interpolations. A manual self-collision setting can greatly improve the performance of the GPU planner.

	milestone computation	local planning
exhaustive self-collision (CPU)	15,952	643,194
exhaustive self-collision (GPU)	652	13,513
manual self-collision (GPU)	391	392

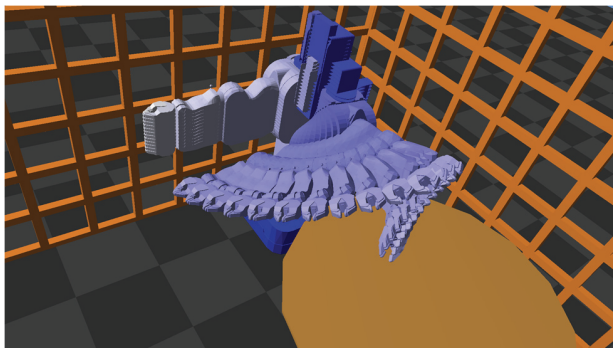


Fig. 10. Our GPU-based motion planner can compute a collision-free path for PR2 in less than 1 second.

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 reducing the memory consumed by traversal stacks. It can provide more than a 50% speed-up compared to simple parallel methods. The second algorithm is based on workload balancing, and decomposes parallel collision queries into fine-grained tasks corresponding to BVTT node operations. The algorithm uses a lightweight task-balancing strategy to guarantee that all GPU cores are fully utilized and achieves close to peak performance on the GPUs. In practice, we observe a speed-up of 5–10 times. The new collision algorithms can improve the performance of GPU-based PRM planners by almost 50%.

There are many avenues for future work. We are interested in using more advanced sampling schemes with the GPU-based planner to further improve its performance and deal with narrow passages. Furthermore, we would like to modify the planner to generate smooth paths and integrate our planner with physical robots (e.g. PR2). We would also like to take into account kinematic and dynamic constraints.

Note

1. http://www.nvidia.com/object/fermi_architecture.html.

Funding

This work was supported by in part by ARO [Contract W911NF-04-1-0088], the NSF [awards 0917040, 0904990, and 1000579], DARPA/RDECOM [Contract WR91CRB-08-C-0137], and Willow Garage.

Conflict of interest statement

None declared.

References

Aila T and Laine S (2009) Understanding the efficiency of ray traversal on GPUs. In *Proceedings of High Performance Graphics*, 145–149.

Akinc M, Bekris KE, Chen BY, Ladd AM, Plaku E and Kavraki LE (2005) Probabilistic roadmaps of trees for parallel computation of multiple query roadmaps. In *Robotics Research*, vol. 15 of *Springer Tracts in Advanced Robotics*. Springer, 80–89.

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

Banerjee A and Ghosh J (2006) Scalable clustering algorithms with balancing constraints. *Data Mining and Knowledge Discovery* 13(3): 365–395.

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

Che S, Boyer M, Meng J, Tarjan D, Sheaffer JW and Skadron K (2008) A performance study of general-purpose applications on graphics processors using CUDA. *Journal of Parallel and Distributed Computing* 68(10): 1370–1380.

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

Gunther J, Popov S, Seidel H-P and Slusallek P (2007) Real-time ray tracing on GPU with BVH-based packet traversal. In *Proceedings of IEEE Symposium on Interactive Ray Tracing*, 113–118.

Harris M (2009) *Optimizing Parallel Reduction in CUDA*. NVIDIA Developer Technology.

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

JáJá J (1992) *An Introduction to Parallel Algorithms*. Addison Wesley Longman Publishing Co., Inc.

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

Kim D, Heo J-P, Huh J, Kim J and Yoon S-E (2009) HPCCD: Hybrid parallel continuous collision detection using CPUs and GPUs. *Computer Graphics Forum* 28(7): 1791–1800.

Kuffner J and LaValle S (2000) RRT-connect: An efficient approach to single-query path planning. In *Proceedings of*

IEEE International Conference on Robotics and Automation, 995–1001.

Larsen E, Gottschalk S, Lin M and Manocha D (2000) Distance queries with rectangular swept sphere volumes. In *Proceedings of IEEE International Conference on Robotics and Automation*, 3719–3726.

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

Lauterbach C, Mo Q and Manocha D (2010) gProximity: Hierarchical GPU-based operations for collision and distance queries. *Computer Graphics Forum* 29(2): 419–428.

LaValle SM (2006) *Planning Algorithms*. Cambridge University Press.

Lin M and Manocha D (2004) Collision and proximity queries. In *Handbook of Discrete and Computational Geometry*. CRC Press, Inc., 787–808.

Lindholm E, Nickolls J, Oberman S and Montrym J (2008) NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro* 28(2): 39–55.

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

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

Pan J and Manocha D (2011) GPU-based parallel collision detection for real-time motion planning. In *Algorithmic Foundations of Robotics IX*, vol. 68 of *Springer Tracts in Advanced Robotics*. Springer, 211–228.

Pan J, Lauterbach C and Manocha D (2010a) Efficient nearest-neighbor computation for GPU-based motion planning. In *Proceedings of IEEE International Conference on Intelligent Robots and Systems*, 2243–2248.

Pan J, Lauterbach C and Manocha D (2010b) g-Planner: Real-time motion planning and global navigation using GPUs. In *Proceedings of AAAI Conference on Artificial Intelligence*, 1245–1251.

Pisula C, Hoff K, Lin MC and Manocha D (2000) Randomized path planning for a rigid body based on hardware accelerated Voronoi sampling. In *Proceedings of International Workshop on Algorithmic Foundation of Robotics*, 279–292.

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

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

Talamadupula K, Benton J and Schermerhorn P (2009) Integrating a closed world planner with an open world. In *Proceedings of ICAPS Workshop on Bridging the Gap Between Task and Motion Planning*.

Tang M, Manocha D and Tong R (2010) MCCD: Multi-core collision detection between deformable models. *Graphical Models* 72(2): 7–23.

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

Zhang L and Manocha D (2008) A retraction-based RRT planner. In *Proceedings of IEEE International Conference on Robotics and Automation*, 3743–3750.