Efficient data management for incoherent ray tracing

Xin Yang\textsuperscript{a,b,*}, Duan-qing Xu\textsuperscript{b}, Lei Zhao\textsuperscript{b}

\textsuperscript{a} Dalian University of Technology, China
\textsuperscript{b} Zhejiang University, China

\textbf{A B S T R A C T}

To obtain good performance on the GPU hardware, it is necessary to design algorithms to manage data, access memory under GPU memory hierarchy, and schedule more efficient threads. In this paper, we propose an efficient data management and task management designed for GPU based ray tracing. Due to the dynamic and uncertainty in ray tracing, we design data-management layer and task-management layer combined with fuzzy spatial analysis, use the two-level ray sorting and a ray bucket structure to reorganize ray data, then a warp’s threads can be scheduled to access coherent geometry and nodes data, reduce memory bandwidth, and dispatch the data locally. We schedule tasks in data-driven execution according to coherent data, propose an adaptive ray compaction to eliminate inactive threads, maintain task efficiency of threads in a warp, and design two heuristics to decrease the compaction cost. On the basis of it, we also introduce a memory-optimized dynamic traversal management to reduce incoherent memory access, and avoid frequent sorting computation and compaction operations. Our experiments demonstrate all of these work combined can achieve good performance.

Crown Copyright © 2012 Published by Elsevier B.V. All rights reserved.

1. Introduction

Since the introduction of ray tracing in its classic form [1] among the two decades, exponential growth in the available compute power and a variety of algorithmic developments have been used to realize real-time ray tracing on commodity processors. Current research on real-time ray tracing focuses on optimizing primary and shadow rays. The results are encouraging, approaching similar frame rates compared with rasterization-based techniques. However, the attractive power of ray tracing is the ability to easily produce photorealistic effects by secondary rays, such as soft shadows, glossy reflections, motion blur, depth of field, and diffuse global illumination. Thus, how to generate these fuzzy effects should be a valuable topic.

Different from primary rays, which have a common origin (the camera/eye) and may traverse the same nodes and intersect the same primitives, secondary rays here mean that those rays have multiple bounces of reflections and refractions, multiple shadow rays per light source or compute multiple bounces of reflections or refractions, the first level of rays does not account for a large percentage of the rendering time. On the other hand, despite the secondary rays are not completely random, it is quite obvious that they do not have the same manner as coherent primary rays: two secondary rays generated from near points may intersect with objects that are far away in the scene. Therefore, efficient computation of secondary rays is a harder problem than for primary rays.

Coupled with current architecture trends, it is necessary that we may apply highly parallel algorithms to leverage throughput-oriented architectures such as GPUs and to scale well as processors become increasingly parallel. For a highly parallel algorithm such as ray tracing, efficiently exploiting many cores is very important. GPU-based ray tracers have recently achieved a performance equal and even better than that of CPU implementations, various techniques have been proposed to use GPU to accelerate ray tracing. Though some optimizations have been applied to obtain benefits, the limitations of the underlying current architecture programming model have hampered the efficiency and performance of these algorithms. We focus specifically on designing algorithms for CUDA [2] that take advantage of the massively multi-threaded design of modern NVIDIA GF110 GPUs [3]. NVIDIA GF110 architecture contains up to 512 process units, and is commonly viewed as a massively multi-threaded scalar architecture.

Because of some hardware limitations of current GPU architecture, if to obtain good performance, it is necessary to design some special algorithms to manage data, access memory based on the
hierarchy of GPU memory, and schedule more efficient threads execution. To exploit the high utilization in maximum, we should follow two principles according to the features of current parallel hardware architecture, that is not very important to the algorithm implemented on CPU: First, high performance can only be obtained when all of a warp's threads mostly execute the same instruction; Secondly, the warp's threads should access local memory as much as possible, here "local" means to a partial continuous area relative to the whole memory area. However, complex lighting algorithms for photo-realistic images may contain many incoherent memory accesses and divergent execution, and the implementation of the dynamic and uncertainty in ray tracing violates these two principles, finally causes low efficiency on GPU.

As discussed above, it is necessary to reorganize the data, so that a warp's threads can access coherent data, reduce memory bandwidth usage, and dispatch the data locally. Then, we can schedule task executions according to the coherent data. In this paper, we propose an efficient data management especially designed for GPU based ray tracing according to the two optimization principles mentioned above. Our approach is especially efficient for GPU memory features, and exploits the hardware parallel computation heavily.

We provide a way to cluster and manage geometry and ray data, then threads can execute almost same instructions and access local memory in maximum. We design two-level ray resorting, first group rays by direction vectors to make them into eight buckets, then within each bucket, further to divide subsets based ray origins and acceleration structure information. Also, the tasks are distributed among CPU and GPU based on features of computation capabilities of these two devices. Specially, we propose an adaptive ray compaction to eliminate inactive threads, and maintain task efficiency of threads in a warp, and design two heuristics to decrease the compaction cost. On the basis of it, considering the dynamic and unpredictable feature of secondary rays during traversal, we introduce a memory-optimized dynamic traversal management to reduce incoherent memory access, and avoid frequent sorting computation and compaction operation. Intersection information is stored in hierarchy among ray generations. Acceleration structure and geometry data are stored on texture units to improve cache utilization. Our experiments demonstrate all of these work combined can achieve good performance.

Section 2 presents previous work in this area, while Section 3 introduces more formal definitions of our approach. Section 4 discusses our algorithm implementation, analyzes the performance of our implementation and compares it with current work. Finally, in Section 5, we offer some thoughts on future work.

2. Previous work

In this section, we give a brief overview of prior work about ray tracing.

2.1. Coherent ray tracing

Wald et al. [4] first present packet tracing using SSE vector instructions. For coherent primary rays and shadow rays from a point light, they obtained good performance, but shading and reflection rays were handled in single ray code. Wald et al. [5] added a packet algorithm to BVH, which resulted in high performance for dynamic scenes. Reshetov [6] proposed a method to travel KD-tree as a BVH style to keep packets together, even when their directional signs disagreed. However, incoherent ray distributions led to extremely low SIMD utilization. Reshetov [7] also showed that a fast primitive culling test under shallower kdtrees with larger numbers of primitives per leaf, while maintaining similar rendering performance. The approach was not extended for incoherent rays, however, the utility of the culling test clearly relies on ray coherence. Pharr et al. [8] introduced memory coherent ray tracing of complex scenes by grouping rays and geometry into a spatial scheduling voxel grid. They processed the voxels one at a time, by tracing the contained rays against the contained geometry, and first compute the voxels with geometry currently on cache. By carefully designing ray, geometry and texture caches, they argued that rendering times can be substantially improved. However, since this method specifically works for already coherent sets of rays, it is hard that they could be applied with much benefit in handling the incoherent rays of a globally illuminated scene.

2.2. Secondary rays

Different from primary rays, secondary rays might not always look highly coherent, and they may have either "hidden" coherence or no coherence at all. Several recent works [9] have focused this problem of incoherence in secondary rays. Wald et al. [10] handled streams of rays on a hardware architecture supporting scatter and gather operations with a wide SIMD. This method first traces a large group of rays in breadth-first order through BVH, then sweeps out inactive rays from the stream at each traversal step. Partition traversal was introduced by Overbeck [11], they used ray reordering to a modified packet traversal, and introduced a simple partitioning scheme to update a list of active rays. Boulou et al. [12] described packet assembly techniques for distribution ray tracing as for standard recursive ray tracing, which achieved similar performance. In addition, Mansson et al. [13] introduced several coherence measures for ray reordering to achieve interactive performance with secondary rays. Although the fact that sorting may increase coherence, the sorting operation is still expensive, and resulted in too many additional operations. In the other hand, Reshetov [6] has shown that even for narrow SIMD units, perfectly specular reflection rays with multiple bounces quickly still may incur almost completely incoherent ray packets and 1/k SIMD efficiency.

Another class methods for incoherent ray traversal design some special data structure, according to the feature of the hardware and incoherent ray packets, such as QBVH [9] and multi-BVH (MBVH) structure [13]. These algorithms trace single rays at a time through a BVH, which has a higher branching factor, usually equal to the SIMD width, and ignore coherency consideration. SIMD units are always occupied at high efficiency regardless of coherency by these algorithms, and allowing scenes to be built with shallower trees, thus generate fewer traversal steps. The disadvantage of this method is that hidden coherency is not exploited like packet tracing. However, coherent executions are important in saving memory bandwidth, as coherent rays can revisit the same nodes along a common path. Therefore, this shows that it is indeed a challenging research about the incoherent rays.

2.3. Memory access

Pharr et al. [8] reordered and cached rays and geometry to guarantee that rays are traced only against geometry in memory. The scheme preprocessed data according to the character of rays, so chunks of geometry with spatial locality keep close together in memory. Navratil et al. [14] improved the scheduling algorithm and used a simulator to prove that they can significantly reduce the amount of geometry transferred into the caches. However, their methods only work for data managing between disk and main memory (or between main memory and processor cache), and is not for the memory hierarchy structure on the newest GPU architecture and the access model of threads. More importantly, secondary
rays are unpredictable, thus the data may be accessed randomly, which is inefficient on GPU architecture.

3. Data management

3.1. Hardware architecture

The main computational unit on CUDA is the thread. As opposed to other GPU architectures, threads on CUDA can read and write freely GPU memory and can synchronize and communicate with each other. To enable communication and synchronization, the threads on CUDA are logically grouped in blocks. Threads in a block synchronize by setting barriers and communicating through a small high-speed low-latency on-chip memory (a.k.a. shared memory). Threads are physically processed in chunks of size 32 in SIMD. CUDA consists of several cores working independently on a disjoint set of blocks. Each core can execute one chunk at any point of time, and switch among the active blocks (hardware multi-threading). CUDA can hide various types of latencies under this mechanism, introduced for example by memory accesses or instruction dependencies.

The CUDA memory consists of a rather large on-board part (global memory), used for storing data, texture units and small on-chip parts, used for caching and communication purposes. Accessing global memory is expensive in terms of the introduced latency. Each core has its shared memory and accessing them is as fast as a register. Current available consumer high-end GF10 GPUs (GeForce GTX 580) have 48 cores, an on-board memory of 1.5GB and 48KB of shared memory per core.

The number of active threads (chunks) on a core is determined by three factors: the number of registers each thread uses, the size of shared memory partition of a block and the number of threads in a block. Using more registers or larger shared memory partitions limits the total number of active threads that a GPU can run, which in turn impacts the performance, since multi-threading is the primary mechanism for latency hiding on GPU. The number of threads per core is limited to 768 and to 12K for the entire GPU. Full utilization of all cores can be achieved if each thread uses no more than 10 scalar registers and 5 words of shared memory.

3.2. Data-management layer

While GPU provides us with strong computation ability, as mentioned above, we have to follow two important principles, which is not important to the algorithm executed on CPU: First, high performance can only be obtained if all of a warp’s threads mostly execute the same instruction; secondly, the threads in a warp should access local memory as much as possible, here “local” means to a partial continuous area relative to the whole memory area.

However, the dynamic and uncertainty of ray tracing violates these two principles, causing low efficiency on GPU. Thus, it is necessary to reorganize the data to make a warp’s threads access coherent data, reduce memory bandwidth, and dispatch the data locally. Different from primary rays, secondary rays cannot guarantee high coherence between rays, but these rays may have the hidden coherence. In this paper, we use the resorting to exploit this coherence.

3.2.1. Two-level ray sorting

For ray tracing, we manage two types of data: rays and geometry. For ray data, we manage them based on a principle: rays can be traced and processed independently (non-recursively). Next, we process ray data first. As mentioned above, coherent rays here mean rays with similar directions and origins.

For primary rays, sorting rays into coherent ones is straightforward, because all rays almost have a common origin (the camera/eye) and similar directions, thus we can easily create coherent data as a packet by grouping the rays of nearby pixels. However, for secondary rays, we have neither a common origin, nor coherent directions. But, we can use this two ray attributes as sorting measure for secondary rays. Further, for secondary rays, we organize them into two levels of memory hierarchy as below.

In three-dimensional coordinate system, the direction of every ray is an accumulation of three direction components along x, y, z axis. And for every axis direction, we further to divide it into positive direction and negative direction. We can use $x+y+z$, $x+y-z$, $x-y+z$, $x-y-z$, $y-x+z$, $y-x-z$, $-x+y+z$, $-x+y-z$ to present these direction attributes, and group the rays into eight buckets: $(x+y+z), (x+y-z), (x+y-z), (x-y-z), (x-y-z), (x-y-z), (x-y-z), (x-y-z)$. Other words, at the first level, we group all the rays into a bucket if they have the same direction vectors, and we need eight buckets like this in all to organize all the rays.

On the other hand, considering two rays in the same bucket, if they are relatively far apart at the origins, they also might not have similar travel routes. Thus, in order to get more efficient and coherent ray data, we continue to divide the rays in the same bucket above according to their origins. In order to decrease the complexity and time, while increasing the validity, we use the hierarchy information of acceleration structure to help divide ray data. As for choosing which acceleration structure, our algorithm is general, not designed for the special structure. In the implementation, we use BVH as acceleration structure, and choose eight node bounding boxes at the fourth level. These eight nodes of this level make the whole scene divided into eight space areas. We use the space information as subdivision measure next. According to space information from the eight node bounding boxes and fuzzy spatial analysis [15–17], we further to divide the ray data of a bucket into eight subsets based on the affiliation between ray origins and node bounding boxes. In the implementation, we use bucket-based sorting to sort these ray data on GPU. Moreover, we further to sort ray data in a bucket using Morton.

At data-management layer, we use ray direction and ray origin as dividing measure to organize two level ray data subdivision as mentioned. After finishing ray data subdivision, we need to drive secondary rays to travel along similar routes based on these data next. We will give more details in the next section.

3.2.2. Data generation management

Because primary rays are generally coherent, we process them in a different way. We use the programmable pixel shader to handle primary rays using rasterization, Z-buffer and per-pixel lighting. The same shader outputs the first set of secondary rays in a separate render target as rasterization based techniques, with the starting point and direction. Then, we sort these secondary rays into their corresponding bucket as mentioned above.

Because sorting procession needs heavy logic and light computation, which is not suitable for stream-data process on GPU, we switch this work to CPU side, then transfer back the sorting results to GPU side. Though the transfer operation between CPU and GPU may increase the burden of bandwidth, current PCI express can allow fast transfer between CPU and GPU, and it only accounts for a small fraction of the bandwidth between CPU and GPU, that guarantees the transfer procedure is fast. Moreover, remaining work and data are entirely processed on GPU, which also can save a lot of bandwidth. Thus, even if we need the transfer operation every frame, it will not significantly degrade the performance. Because all sorting work are processed on CPU with plentiful logic function, cache and new/free etc. C++ support, they can be executed in a more efficient manner, compared to sorting operation on GPU.

Because the number of secondary rays is not predictable, we process rays in generation to organize rays and control the number. Different from other rays, shadow rays stop traveling once finding
any intersection. Thus we process shadow rays in current generation first, then any newly spawned non-shadow rays are processed. This process manner limits the amount of active rays in current system while still providing coherent access to scene geometry. In addition, this process may control thread execution behavior to some extent, and weaken the feature of dynamic in ray tracing.

To generate shadow rays and possible secondary rays, we keep the intersection information of current generation of rays, and either do shading at the intersection point immediately (similar with ray casting) or save it for deferred generation of secondary rays. Each ray maintains a pixel id, and the proper pixel can be shaded. When doing super-sampling, samples can be blended and computed in the frame-buffer as they are all available. Once all rays of current generation have been processed, any intersection points are used to generate the next generation of shadow and secondary rays. Shadow rays inherit both the pixel id and the shading information from their spawning ray. Thus, when light visibility has been determined, shading contribution, if any, can be added to the appropriate pixel. This process continues until no new secondary rays are generated. Then, we return these rays to CPU to sort them. Though scattering these data may cause some latency, we can reduce the bandwidth by coalescing.

3.3. Task-management layer

At data-management layer, we reorganize two types of data: rays and geometry. After this organization, we exploit hidden coherent data in chunks, and make algorithm execution especially suitable for the memory hierarchy on GPU. Next, we will schedule tasks execution according to these sorted coherent data, follow two important principles mentioned above, and increase parallel computation performance provided by GPU.

In our algorithm, task management is a data-driven execution. We have organized coherent rays to visit the same or similar tree node in maximum using rays bucket structure, next we drive and schedule warps of threads based on these data, guaranteeing that warps of threads can access memory in locality, and decrease frequent memory access caused by dynamic execution during secondary ray tracing. In addition, ray buckets can buffer enough rays to ensure threads maximally occupied. We describe the algorithm architecture in Fig. 1.

As discussed above, coherent rays usually visit similar nodes and intersect same primitives, which can reduce either computation or bandwidth significantly. While ray bucket structure can provide coherent rays in maximum, the model with high accuracy still contains many small primitives, which likely causes the rays in the same chunk visit nodes in different routes.

3.3.1. Adaptive ray compaction

For thread execution, it often happens that some long-running threads keep the whole warp hostage. We modified persistent threads to periodically replace terminated rays with new ones starting from the root. These new rays have the same task at the current time, thus guarantee the threads in a warp can continue to execute the same instructions, and increase efficient hardware utilization. In our algorithm, we use reduction and compaction primitive operation to do adaptive ray reorganization, and post-process the output of threads of current active warp.

Stream compaction is a general technique for reordering a disorganized stream of elements into compact sub-streams of elements. Stream compaction is implemented with the more primitive operations of prefix sum and scatter. Prefix sum [3] enumerates the threads (inside a warp) for which a condition is true and returns a unique index \([0, \ldots, M - 1]\) to those threads. Population count returns the number of threads for which a condition is true, i.e., \(M\) above. After the new location of each element is identified reduction operation is finished, and all elements are moved, or “scattered” to their new locations. Here, the element means ray data. However, these ray data are stored on texture unit, which is only-read memory. We need to read these data into shared memory on-chip, which can be accessed in a cycle without bank conflict. Besides, we set a stack structure on shared memory used by rays.

Acceleration structure is constructed and stored in a breadth-first manner, and we traversal it in the same order, which is a good way to access memory on GPU. For prefix sum, this operation has not been paid off on current CUDA architecture, and we adapt CUDPP’s implementation [19]. However, this operation is still expensive and slow, compared to other computation operations. Thus, we design two heuristics to decrease the compaction cost.

First, we find that, at the beginning of traversal, the warp of threads launched by data-driven model may still execute the same tasks, and access similar nodes. Thus, it is unnecessary to execute compaction operation on results computed by the warp of threads every time. We set a threshold for the level of acceleration structure. As the level of acceleration structure increases, the divergence in a warp of threads gradually begins to appear clearly. When current level at which ray travels exceeds the threshold, we begin to do the reduction operation for the output of warp of threads. In the
implementation, we do the reduction operation for the output of warp when the traversal level of tree exceeds five.

Secondly, compacting rays only when a small portion of rays become inactive is wasteful. Instead of compacting every step, we only compact threads when the utilization of the warp drops below a threshold. We define this utilization to be: \( P/Q \), here \( P \) is the number of active rays in chunk, and \( Q \) is the total number of rays in chunk. We begin to compact and resort these rays until the utilization drops below a threshold. In our implementation, we do the compaction when the utilization drops below 50%. We reorganize the rays on shared memory by adaptive compaction into coherent sub-streams, and maintain memory access locally in maximum.

When visiting the leaf node, rays need to intersect the primitives of the leaf. Some algorithms switch that individual rays intersect with different triangles, using parallel threads in SIMD manner to exploit hardware utilization heavily, rather than using ray packets. We do not adopt this method based on the following reasons: First, this method needs the leaf of acceleration structure contain more primitives, but it probably causes bad hierarchy of acceleration structure; Secondly, this approach cannot give any benefit for shading and ray generation; Thirdly, because of using different parameters, it needs two kernels to execute traversal and intersection in succession for all rays, which may cause early termination of some threads in traversal, then wait for still running threads; fourthly, it may access different primitives in a wide range, which will cause cache invalid. On the other hand, we can make the warp of threads maintain coherent in maximum by compaction, thus it is possible that the warp of threads still intersect the same primitive.

3.3.2. Memory-optimized dynamic traversal management

In the last section, we use adaptive ray compaction to eliminate inactive threads, finally maintain the efficiency of threads in a warp and efficient SIMD ray tracing. However, memory access for secondary rays is irregular, a bad case is that it is hard to control the threads in a warp to access memory in locality, possibly make memory access more randomly and cache invalid, finally the sharp degradation in performance. Further, when the threads in a warp have incoherent memory access, i.e., access a lot of different node data, that will consume large bandwidth usage, and cause memory latency.

For example, recursive ray tracing uses depth-first traversal, though two coherent primary rays may access the same memory area, the access time is not in the same period. In other words, child rays of the first primary ray should finish traveling before the second primary ray starts to travel, then make memory access irregular, result in more frequent data read and write, consume large bandwidth usage.

At data-management layer, we use two-level sorting to obtain coherent rays, and we need to maintain and enhance this coherence during task execution. Considering the dynamic and unpredictable feature of secondary rays during traversal, it is hard to locate an appropriate time point to sort all active rays by data access at the same time. On the other hand, we cannot sort data frequently, which is an expensive computation task, and may make sorting time exceed memory access time. Therefore, we propose a memory-optimized dynamic traversal management to reduce incoherent memory access, and avoid frequent sorting computation and compaction operation.

This technology follows a principle: There are three states for a ray visiting an acceleration structure, i.e., traversal, intersection and exit. A ray should be at one of these three states mentioned above during ray tracing. After finishing tracing, there is no ray at traversal state. Specifically, we create a ray queue for every acceleration structure node. This queue stores the rays to visit the node attached, on the other hand, we also eliminate the rays in queue, which are not to visit the node anymore. This queue mechanism manages rays dynamically, and maintains the coherence between rays. Here, the coherence means coherent memory access, coherent structure and coherent procession.

During traversal, the threads in a warp represent coherent rays with the same traversal routes, and visiting same or similar tree nodes, which is as same as the general ray packet technology. The only difference is that we add a filtration mechanism on tree node. If a ray stops traveling down from a node, then we shift this ray up to the queue, whose attached node the ray just visited. The core idea of the dynamic traversal is to maintain the queue attached to the corresponding tree node contains coherent rays with the same traversal route. Besides, this ray queue also can be manipulated for asynchronous communication, and collect dynamically ray data waiting to visit the node during traversal, further to enhance coherent task execution.

On the other hand, our algorithm can schedule adaptively the traveling node based on the amount of rays in the current queue. In other words, if a ray queue cannot contain enough rays, and cannot exploit parallel procession on GPU, then this ray stops traveling down. When this ray queue recollects enough rays waiting to visit the node, the rays in queue restart to travel down along tree. After all rays finish their traversal tasks, the final state of acceleration
structure should be that the ray queue of every node does not contain any rays.

Another advantage of dynamic ray traversal is that traversal operation does not need to start from root node, instead from the attached node. This manner decreases repeated and invalid traversal operations, reduces data transfer operations, finally save bandwidth usage. Besides, queue mechanism can implement low-cost feedback with indeterminate execution time, then process efficiently a lot of incoherent rays generated by some tree nodes during secondary ray tracing. Thus, this approach can solve efficiently unpredictable and dynamic execution behaviors during secondary ray tracing. Compared with the cost caused by schedule and management, the flexibility introduced by the method produces larger performance improvement.

4. Experiments and results

To evaluate the performance of our method, in this section, we test and verify our algorithm in a variety of experiments. The experiments include comparison with single ray tracing and ray packet methods, and cover frame rate performance, hardware utilization, optimization on block size, data traffic, and GPU utilization. All experiments are done under single light source. The described algorithm has been tested on an Intel Xeon 3.7 GHz CPU with an NVIDIA GTX 580 graphics card. We use OpenGL to display the rendering pictures, and the results do not include the display time and data transfer time between CPU and GPU.

We implemented our approach using NVIDIA’s CUDA framework [20]. Previous GPU programming systems limit the size and complexity of GPU code, due to their underlying graphics API based implementations. CUDA supports kernels with much larger code sizes, a new hardware interface and instruction caching. Also allows for general addressing of memory via a unified processor model, which enables CUDA to perform unrestricted scatter-gather operations.

All of the tests in this paper use bounding volume hierarchy (BVH) and the barycentric coordinate computation [21] in projected form. BVH was built using the greedy surface-area heuristic with maximum leaf node size 8 for all scenes. The performance of our algorithm is built on optimization principles and focuses less on highly specialized data structures. Register count of the kernels ranged from 21 to 25 to make more threads active. We used a thread block size of 256, which achieved the optimal performance in our experiment. All data was stored as array-of-structures, and nodes were fetched by a 1D texture in order to benefit from the texture cache, while triangles were retrieved directly from global memory. This organization proved to obtain the good performance on GPU, even though global memory access is not cached. Using cache efficiently on GPU is an important step in our algorithm.

We show that by using the data-management layer to efficiently manage ray data, nodes data, and geometry data under the memory hierarchy of CUDA architecture, setting the task-management layer to schedule the threads, and use data sorting instead of data cache. This approach follows well the CUDA design principles, can use efficiently the computational resource provided by CUDA architecture. For secondary ray traversal, this algorithm can improve efficiently cache utilization and GPU hardware utilization, which will become more important for extra lighting and shading data of complex scenes. Our method can both decrease the amount of data transfer and increase the available cache utilization.

For test scenes, we choose the freely available scenes Bunny, Fairy, and Lilo. These scenes span a wide range of complexity, from 69K triangles in Bunny to over 600K triangles in Lilo. Bunny and Fairy are from Stanford 3D scanning repository, and used widely to test ray tracing. Lilo is from Dunhuang Mogao Caves, and used to test algorithm performance as a complex scene. The scenes and the viewpoints for the tests can be seen in Fig. 2. Each test scene was rendered at 1024 × 1024 resolution with a light source and we vary the maximum number of reflection bounces.

We note that Bunny is a scene with low geometry complexity, which contains many big primitives, and this increases potentially the coherence of ray traversal. On the other hand, Lilo scene is not actually closed (rays can escape through the vents), and there are still a small number of rays that hit the ground. Some rays may escape the scene after some tracing, but escaping can be also viewed as ray coherence.

We compare it against single ray tracing on CPU, packet tracing on GPU, and measure the efficiency of these methods in a variety of experiments. (The comparison algorithms are from the open source project Arauna [22].) To generate more incoherent rays, Tables 1 and 2 give performance in FPS for the different methods with two-bounce, and five-bounce perfect reflections. For completeness, two bounces means that we shoot one level of primary rays, two reflection rays, and three sets of shadow rays. As expected, increasing the number of bounces results in a severe drop in performance, depending on the different scenes. Our algorithm is competitive with packet tracing for small numbers of bounces and pulls further away for higher bounces, in which the incoherence between secondary rays appear more intensive, and the

Table 1 Performance in FPS with two-bounce reflections.

<table>
<thead>
<tr>
<th>Scene</th>
<th>Single ray</th>
<th>Ray packet</th>
<th>Our approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bunny</td>
<td>6.3</td>
<td>7.6</td>
<td>10.1</td>
</tr>
<tr>
<td>Fairy</td>
<td>2.7</td>
<td>3.9</td>
<td>5.3</td>
</tr>
<tr>
<td>Lilo</td>
<td>1.9</td>
<td>2.8</td>
<td>3.2</td>
</tr>
</tbody>
</table>

Table 2 Performance in FPS with five-bounce reflections.

<table>
<thead>
<tr>
<th>Scene</th>
<th>Single ray</th>
<th>Ray packet</th>
<th>Our approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bunny</td>
<td>3.9</td>
<td>4.1</td>
<td>6.7</td>
</tr>
<tr>
<td>Fairy</td>
<td>1.9</td>
<td>2.2</td>
<td>3.4</td>
</tr>
<tr>
<td>Lilo</td>
<td>1.5</td>
<td>2.6</td>
<td>2.9</td>
</tr>
</tbody>
</table>
performance of packet tracing drops to the same as single ray tracing, instead our method starts to reflect its advantage. However, for Lifo scene, because most rays escape from the scene after the second reflection, there is no big difference between two-bounce, and five-bounce reflections, the obtained performance improvement mainly comes from the efficient data management.

We give the comparisons of hardware utilization for packet tracing and our algorithm in two bounds and five bounds (Tables 3 and 4). As expected, when the reflection bounce increases, the performance of packet tracing drops to nearly 50%, but our method only drops by 15–25%. In other words, our method can exploit GPU hardware more efficiently, compared to packet tracing, especially for traversal.

Also we test Fairy scene to show the influence of block size on the performance in Table 5. While the grid size corresponds to the size of the frame to be rendered, with one thread per pixel, we select two-dimensional contiguous pixel blocks of different sizes for the kernel invocation. The mapping of block threads to multiprocessors is handled transparently by the CUDA framework, subject to the constraint that there are sufficient resources (registries and local memory) available on the multiprocessor that the block is mapped to, hence there is an upper limit to the number of block threads and hence the block size. The maximum number of 8.3 million rays/s is achieved for a block size of 8 × 32 in Fairy scene.

In Fig. 3, we test Fairy scene to show how efficiently the data-management layer in algorithm can reduce data traffic, compared to packet tracing. Except keeping the advance of packet tracing and reducing the use of bandwidth, our algorithm decreases data transfer by a significant margin of 28% compared to packet tracing, finally reduces access memory, which is important for GPU efficiency. Especially, our method can reduce geometry and nodes data traffic by 41%. While the construction of ray buckets and some sorting work may consume extra bandwidth, it only account for a small fraction of peak bandwidth provided by current GPU architecture. Compared to the final reduction of data transfer, this extra cost can be ignored.

In Fig. 4, we give the comparison of packet tracing and our approach for GPU utilization as the depth of tree increases using Fairy scene. On top of tree, both algorithms keep the high utilization. As the increase in depth, the utilization of packet tracing begins to decline due to the insufficient data and low coherence between rays. However, our method can keep high GPU utilization above 60% during the whole traversal period, not greatly affected by the low coherence between rays.

5. Conclusions and future work

In this paper, we propose an efficient data management and task management designed for GPU based ray tracing, and our algorithm is especially for the features of current parallel hardware architecture. Due to the dynamic and uncertainty in ray tracing, we design the data-management layer and task-management layer combined with fuzzy spatial analysis, use two-level ray sorting and a ray bucket structure to reorganize ray data, then a warp’s threads can be scheduled to access coherent geometry and nodes data, reduce memory bandwidth usage, and dispatch the data locally. We schedule tasks in data-driven execution according to the coherent data, guaranteeing that warps of threads can access memory in locality. Further, we propose an adaptive ray compaction to eliminate inactive threads, and maintain task efficiency of threads in a warp, and design two heuristics to decrease the compaction cost. On the basis of it, considering the dynamic and unpredictable feature of secondary rays during traversal, we also introduce a memory-optimized dynamic traversal management to reduce incoherent memory access, and avoid frequent sorting computation and compaction operation.

Our approach is the newest and least optimized, leaving some problems for improvement. It gives an instructive attempt to design incoherent ray tracing on GPU, and opens a new design space that offers many interesting implementation alternatives. We believe that our work provides a compelling design for future ray-based graphics hardware. Though the existing numbers are promising, a lot still remains to be done. The experiments in this paper explicitly
avoid focusing on shading coherence. We just batch the rays by material in a linear sweep simply, it is obviously inefficient. We also plan to explore our algorithm with cache support expected in the new generations of NVIDIA commodity. We also believe the same techniques we have presented could be useful in adapting some other aspects of general purpose programming on GPU.

References


Xin Yang is a PhD student in Zhejiang University, China and is generally interested in most aspects of computer graphics, and computer vision and image processing techniques that can be applied to graphics problems. He is particularly interested in parallel computing with a focus on real-time rendering and GPU data structure design.

Duan-qing Xu is a professor in Zhejiang University, China. He is interested in broad topics in the field of computer systems, in particular computer graphics.

Lei Zhao is an assistant professor in Zhejiang University, China. His research mainly focuses on computer graphics related, in particular those that use innovative hardware and software working together to solve challenging engineering problems.