Mining Effective Parallelism from Hidden Coherence for GPU Based Path Tracing

Wang Tong*
Department of Microelectronics and Nanoelectronics
Tsinghua University

Yangdong Deng
Department of Microelepctronics and Nanoelectronics
Tsinghua University

Abstract

As one of the essential global illumination algorithms, Monte Carlo path tracing has long been considered as a typical irregular problem that is less friendly for graphics hardware. To improve the efficiency of Monte Carlo path tracing, such techniques as ray reordering, ray compaction and wavefront formulation have been proposed to exploit the inherent coherence in processing different paths and materials for better SIMD efficiency on GPUs. In this paper, we develop a novel technique to extract extra parallelism in Monte Carlo path tracing applications by identifying hidden coherence. The basic idea is to perform a partial traversal in the fast on-chip memory of GPU and then identify coherent paths by analyzing the traversal results as well as other features of rays. Such coherence enables a higher level of parallelism that not only compensates the overhead of traversal, but also leads to improved performance. Experiments prove that our techniques deliver a traversal throughput higher than leading-edge results by up to 15%.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing;

Keywords: path tracing, global illumination, hashing

1 Introduction

Monte Carlo path tracing is an essential method of global illumination to render physically realistic graphics effects. At the top level, a Monte Carlo path tracer is friendly for massively multi-threaded processors exemplified by Graphics Processing Units (GPUs) due to its nature embarrassingly parallel [Wald et al. 2001]. Basically, it is straightforward to split the workload of path tracing into millions of rays and then assign each independent ray to a thread. Meanwhile, spatial acceleration structures such as KD-tree [Zhou et al. 2008] and bounding volume hierarchy [Kopta et al. 2012] which are vital for fast intersection detection of rays and primitives, can also be efficiently built and traversed on GPUs.

Despite the advantages of GPUs for path tracing, the irregularity among the workloads for different threads remains to be a major stumbling block for a higher level of performance in path tracing [Aila and Laine 2009]. As rays are randomly bouncing in the scene at each round, their origins and directions can vary considerably. On the other hand, modern GPUs such as NVIDIAs Kepler always simultaneously process a given number of threads as a batch (which is called a warp in NVIDIAs terminology) on a given core. Given two sizably different rays, the two threads for processing them will have greatly diverged execution paths. If the two threads belong to the same warp, the divergence in both control and memory access leads to performance overhead due to the SIMD execution mechanism. Furthermore, it will be much tougher to hide global memory latency because of unpredicted and un-coalesced memory requests from divergent threads [Nvidia 2012].

To solve the divergence problem, it is a critical caveat to group rays that traverse similar nodes in the space acceleration structure to minimize the chance of serialization. We call rays with similar traversal processes as *coherent rays*. However, it is a tough problem to identify such rays. In fact, the perfect information of coherence is only available after the traversal processes are completed. Researchers proposed various heuristics to find potentially coherent rays [Wald 2011; Garanzha and Loop 2010; Hoberock et al. 2009; Novák et al. 2010], but there is still large room for improvement as evidenced by this work.

This work aims to identify coherent rays with a concept of coherence code. This code is defined by considering both static characteristics of a ray as well as a dynamic prediction of the ray with primitives in a scene. The coherence code of each ray consists of 3 components: ray origin hashing value, ray direction hashing value, and partial traversal value. The first and the second values are generated based on the spatial features of the ray and the distributions of scene primitives. The third part faithfully records ray traversal steps in the top 10 levels of the acceleration tree structure. In other words, we use the partial traversal behavior to predict the coherence in the rest process of traversal.

The partial traversal is performed in the GPU on-chip shared memory is used to reduce overhead. In NVIDIA GPUs memory hierarchy, on-chip shared memory is faster than un-cached global memory by two orders of magnitude. We design the spatial data structure so that the top 10 levels of the search tree can fit into 48KB of on-chip shared memory and serve as a pre-split tree for ray classification. Rays are then sorted by their coherence codes so that logically adjacent rays are likely to be assigned to the same warp and efficiently executed on SIMD hardware.

2 Previous Work

Monte Carlo path tracing has attracted significant research effort. A complete treatment of this topic is out of the scope of this paper. Please refer to [Jensen and Christensen 2007] for details. Many studies have been performed to improve the SIMD efficiency by identifying more coherent rays. Such works can be classified into 4 groups.

Hit point prediction: Our work is inspired by [Moon et al. 2010], which proposed a hit point heuristic to order rays into coherent groups. The idea is to find an approximated intersection point for each ray on a simplified mesh of the original scene. However, it can be hard to find a reduced mesh for certain classes of scenes. In addition, the work was CPU oriented, while the reduced mesh can still be too large to fit fast on-chip memory of GPUs.

Better GPU utilization: Ray compaction [Wald 2011; Hoberock et al. 2009] and ray regeneration [Novák et al. 2010] techniques are dedicated to improving SIMD efficiency by better utilizes of GPU computing resource. As more rays are compacted together, threads in a warp tend to become more divergent. Wald [2011] showed a 5X

*e-mail:tong-wang11@mails.tsinghua.edu.cn

SIGGRAPH Asia 2013, November 19 – 22, 2013, Hong Kong.

slow-down in tracing incoherent heavy warps than light warps. Van Antwerpen [2011] combined both stream compaction and ray regeneration techniques to get a more coherent ray set, but the quality is hard to maintain due to the lack of ray classification and sorting processing. In this work, we implemented whole-frame adaptive path compaction and path regeneration techniques for comparison.

Spatial Hash: Garanzha and Loop [2010] introduced a hashing function to sort rays for better coherence in an extra stage in the ray tracing pipeline. Since only simple spatial features of rays are used, our experiments showed similar hash values may not suggest good coherence behavior.

Scene Based Methods: Rays can also be sorted into groups with better coherence by considering the characteristics of materials and/or shading functions [Hoberock et al. 2009; Laine et al. 2013]. Such works are orthogonal to ours.

3 Partial Traversal Heuristic

To better identify coherent rays, we propose a heuristic to predict the coherence of different rays. The heuristic is combined with the spatial features of rays to allocate rays to warps for processing.

We adopted the stack-based ray traversal techniques proposed by Aila and Laine [2009] in this work. KD-tree is selected as the spatial acceleration structure. Given a KD-tree storing the primitives in a scene, each ray traverses the KD-tree with a specific trajectory that can be encoded by a sequence of stack operation. Such information is referred as traversal record in this paper. Obviously, two rays with similar traversal records are likely to incur similar control and memory behaviors, so we can expect less warp divergence and higher SIMD efficiency by assigning them into a single warp. Of course, it would not make sense to perform a whole traversal to identify a complete traversal record. A logic hypothesis is that a partial traversal to the upper level of the underlying KD-tree can serve as a good predictor for a complete traversal. Especially, the partial traversal can be made extremely fast if it is feasible to store the upper level KD-tree in the fast on-chip memory, i.e., shared memory in NVIDIAs terminology, of GPUs. Such a heuristic of traversing a partial KD-tree and making prediction of each rays coherence feature is designated as *partial traversal (step) heuristic (PTH or PTSH)* .

In this work, we only traverse the top 10 level of the acceleration structure according to the hardware setup of current GPUs. When a ray visits a node in the KD-tree, two decisions have to be made. First, we need to determine if the ray should visit the left or the right child node of the current node. Second, we also decide whether a stack push operation is necessary. The two decisions can be encoded in 2 bits. Therefore, the partial traversal record can be encoded with at most 20 bits. Figure 1 illustrates how to derive the partial traversal record.

Our experiments show that the origin and direction of rays also provide useful complementary information for coherence. Rays with similar origins are likely to go through nodes starting from the origin, and thus share similar control flow before hitting an object. In addition, grouping rays with similar directions in a tight frustum has the potential to further reduce the divergence of memory access. We use a more accurate version of spacial hashing on the basis of [Garanzha and Loop 2010]. The 3-D coordinates of the ray origin are adaptively discretized to into a uniform grid, while the ray direction represented as a 3-D vector is also translated into a grid defined by a sphere coordinate system. Each ray can thus generate its hash value based on the 6-D grid information.

The PTH bits and hash values of both origin and direction value of a ray is finally packed into a single coherence code. The hash value of the origin coordinates is chosen as the most significant bits because it helps quickly filter out rays that share similar directions but incur divergent traversal trajectories. Experimental results proved that it leads to better performance to use the PTH bits of the first 10 moves of each ray as the second component and the hash value of the direction as the third.

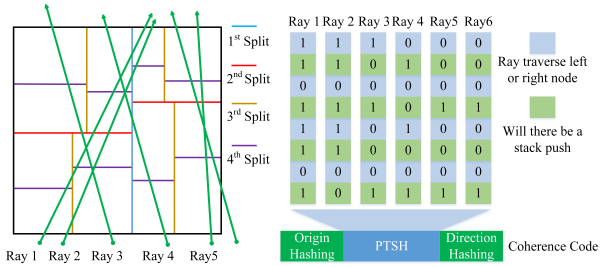


Figure 1: *Illustration of the Partial Traversal Heuristic. At each traversal step, PTH records the child node that a ray will process next and the stack operation that it takes. For instance, ray 1 intersects with both left and right children of the root node, so it will push the right node to stack, and first goes to the left node. Accordingly, we set the first bit to 1 to designate the left node, and the second bit to 1 because it requires a stack push. It then moves on to the next level and writes the next two bits.*

4 Fast Traversal of Upper Levels of KD-Tree

One major reason that justifies the adoption of PTH is that modern GPUs are equipped with fast on-chip scratchpad memory (i.e., shared memory in NVIDIA taxonomy) with sufficient capacity to contain the top a few levels of the underlying KD-Tree hierarchy. It is able to access one 4-byte data word in one cycle from shared memory, while it takes around 400 cycles to get a cache line of data from the global memory. The low latency to access shared memory promises efficient traversal operations. In this section, we explain how to perform fast partial traversal in the shared memory.

4.1 Generation of a Top Sub-Tree

A typical GPU like NVIDIA Kepler and Fermi consists of multiple multi-threaded SIMD multiprocessors (i.e., streaming multiprocessor) with each equipped with 48KB of shared memory. In our implementation derived from [Zhou et al. 2008], one KD-tree node containing the indices and positions of child nodes as well as the directions and positions of split axis takes 16 bytes of storage. Hence, it is feasible to put three 1K-node sub-trees into the shared memory so as to launch 3 concurrent thread blocks on each multiprocessor for occupancy consideration. Our KD-tree construction method ensures that no leaf nodes will appear in the top 1K nodes for a scene with approximately one million primitives. We follow [Zhou et al. 2008] to build a pre-order array and use a typical GPU based BFS traversal method [Harish and Narayanan 2007] to pick up the top 1K node for partial traversal. Note that this process only needs to do once during the KD-tree construction process and the overhead to perform BFS on 1K nodes is negligible with 7% of the entire KD-tree construction time.

4.2 Fast Traversal in Shared Memory

The algorithmic flow of traversing a KD-tree in the shared memory is simpler than its equivalent in the global memory due to less usage of memory resources.

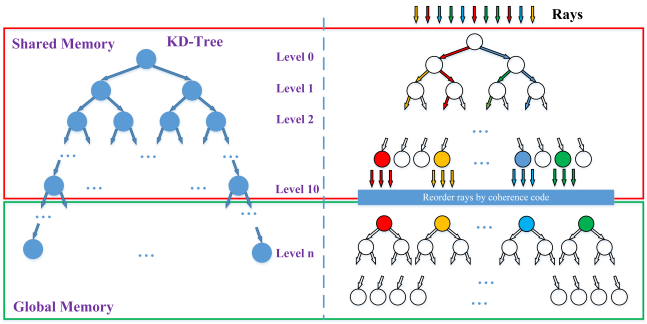


Figure 2: *Traversing top sub-tree in shared memory.*

Figure 2 demonstrates the idea of storing a top level sub-tree of the original KD-tree in the shared memory and performing partial traversal for rays. The PTH value is gradually formed during traversal and written to the coherence code. As a ray leave the sub-tree in the shared memory, its traversal states are pushed to a global stack. Rays are then sorted with the coherence code as keys and coherent ray sets can be identified according to the sorting results. Upon execution, rays in a coherent group are assigned to a warp and resume their traversal of the underlying KD-tree by restoring their states from the global stack. Now coherent rays are logically adjacent and more likely to be executed in the same warp. In addition, our technique does not involve redundant traversal operations.

An important issue is how bank conflicts affect traversal process because such conflicts may considerably drag down the overall performance. NVIDIA GPUs have their shared memory organized in 32 banks, while 32 threads in a warp are concurrently executed. If two threads visit addresses in the same bank, then a two-way conflict happens and the memory effective throughput is reduced by half.

Assume there are n parallel threads access n banks. Also it is reasonable to presume the memory addresses required by ray traversals are totally randomly distributed. The probability that bank i is not accessed by a given thread will be $1 - \frac{1}{n}$. So the probability that a bank is not accessed by any thread is $(1 - \frac{1}{n})^n$. Here we define a random variable X_j:

$$X_j = \begin{cases} 1 & \text{if bank j is not accessed by any thread;} \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

Thus the expectation of X_j can be derived as $E(X_j) = (1 - \frac{1}{n})^n$. Thus we can get that $E(X_j) = (1 - \frac{1}{n})^n$. The number of banks that did not accessed by any thread is $X = X_1 + X_2 + \cdots + X_n$ by definition. Hence the expected number is:

$$E(X) = \sum_{i=1}^{n} E(X_i) = E(X_1)+E(X_2)+\cdots+E(X_n) = n(1-\frac{1}{n})^n \quad (2)$$

The above equation suggests that we have $n - E(X)$ banks are accessed by at least one thread. Note that the first access to a bank does not have the conflict penalty. As a result, $n - E(X)$ accesses are conflict free, while the remaining $n - (n - E(X))$ accesses incur conflicts. Thus we have:

$$E(conflict) = n - (n - E(X)) = n(1 - \frac{1}{n})^n \quad (3)$$

When $n = 32$, we expect 11.6 bank conflicts in every 32 random accesses issued by a warp to the shared memory. It suggests that one shared memory access in PTH causes on average 11.6 serialized accesses to the shared memory. Although it is approximately a 10 fold drop down from the conflict-free situation, its still an order of magnitude faster than the latency of a global memory access. Experiments under random bank conflicts prove that the PTH computation in the shared memory is 2.0-2.4 times faster than its global memory based implementation. The bottleneck of PTH kernel is actually the global stack operations, which suggest a stack-less implementation for KD-tree traversal will lead to further performance enhancement.

5 Implementation and Results

We implemented a path tracer supporting 11 material parameters and BRDFs [Westin et al. 1992], with traversal algorithms based on [Havran 2000] , and also with the speculative execution technique proposed by Aila et al. [2012]. We implement path re-generation [Novák et al. 2010] and whole frame compaction [Wald 2011] techniques for comparison.

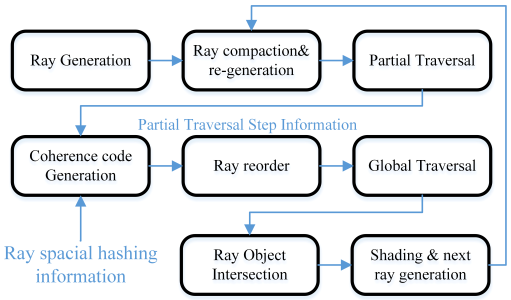
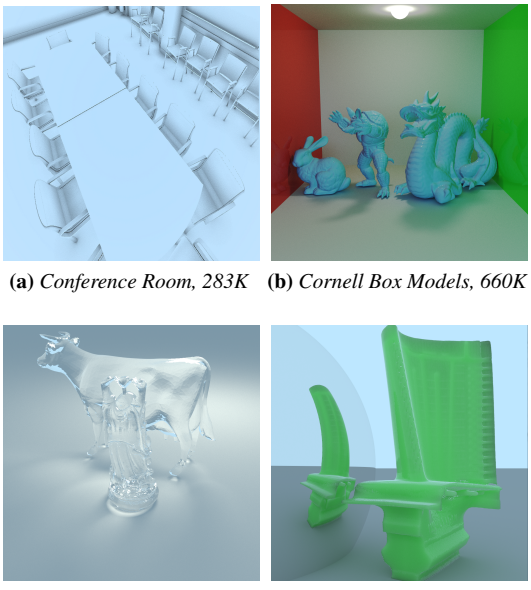


Figure 3: *Modules of our path tracing framework.*



(a) *Conference Room, 283K* **(b)** *Cornell Box Models, 660K*

(c) *Buddha With Cow, 1122K* **(d)** *Blade,1765K*

Figure 4: *Test scenes (from The Stanford 3D Scanning Repository)*

The overall organization of our path tracer is illustrated in Figure

3. The primary rays are generated from camera and then feed to the PTH kernel to derive partial traversal results. The rays are then sorted by the coherence code that combines the PTH value and the spatial hashing values. A reordering process is performed to extract ray sets with better coherence. These ray sets are then assigned to warps for traversal in the global memory. A speculative method is used to postpone leaf node intersection test for less divergence. Upon intersection, new rays are then generated based on BRDFs and other material attributes for the next iteration.

We test the path tracer on a workstation equipped with an NVIDIA Tesla 2075 GPU. The code is compiled with the CUDA 5.0 toolkit. The test scenes contain Conference Room (283K), Cornell Box Models (660K), Buddha With Cow (1122K), and Blade (1765K).

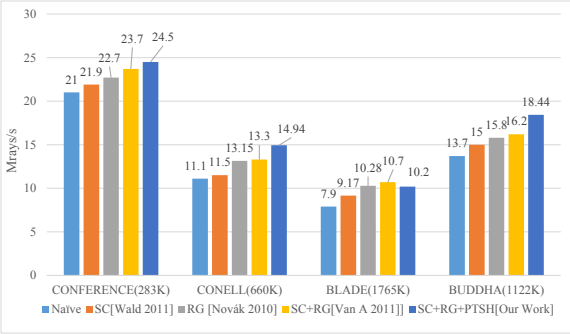


Figure 5: *Performance comparison of a few leading edge path tracers and ours.*

We compared our path tracer with a naive path tracer (NAIVE), a whole frame stream compaction approach (SC) [Wald 2011], path re-generation method (RG) [Novák et al. 2010], and a combination of SC and RG (SC+RG) [Van Antwerpen 2011]. The results are illustrated in Figure 5. Our path tracer delivers a 35% higher throughput in terms of number of rays per second than a straightforward implementation. Compared with leading-edge path tracers mentioned above, our work is up to 15% faster.

6 Conclusion

This work present a novel solution to extract extra parallelism in a typical in a typical set up of path tracing. We propose a partial traversal heuristic to help identify coherent rays for reduced divergence in GPU based traversal. First, each ray first traverses a partial sub-tree of the origin KD-tree. The process is highly efficient because the sub-tree can be stored in the fast on-chip memory of GPUs. The traversal results are then combined with other geometric features to form a coherence code. Next, the rays are sorted by their coherence code. Nearby rays in the ordering are naturally allocated into coherent groups. Note that the partial traversal is not redundant because the later global traversal will resume from the partial results. Our experimental results on scenes with up to 1.765M triangles prove that our techniques attain a ray throughput that is up to 35% higher than a straightforwardly implemented path tracer. The proposed heuristic has a much lower overhead (less than 4% of the overall traversal time) than previous coherence prediction heuristics.

Acknowledgements

Our thanks to the support of National Science Foundation under contract number 20121302065-61272085 and Tsinghua Self-Innovation Foundation under contract number 20121087905.

References

AILA, T., AND LAINE, S. 2009. Understanding the efficiency of ray traversal on gpus. In *Proceedings of the Conference on High Performance Graphics 2009*, ACM, 145–149.

AILA, T., LAINE, S., AND KARRAS, T. 2012. Understanding the efficiency of ray traversal on gpus–kepler and fermi addendum. Tech. rep., NVIDIA Technical Report NVR-2012-02, NVIDIA Corporation, June.

GARANZHA, K., AND LOOP, C. 2010. Fast ray sorting and breadth-first packet traversal for gpu ray tracing. In *Computer Graphics Forum*, vol. 29, Wiley Online Library, 289–298.

HARISH, P., AND NARAYANAN, P. 2007. Accelerating large graph algorithms on the gpu using cuda. In *High performance computing–HiPC 2007*. Springer, 197–208.

HOBEROCK, J., LU, V., JIA, Y., AND HART, J. C. 2009. Stream compaction for deferred shading. In *Proceedings of the Conference on High Performance Graphics 2009*, ACM, 173–180.

JENSEN, H. W., AND CHRISTENSEN, P. 2007. High quality rendering using ray tracing and photon mapping. In *ACM SIGGRAPH 2007 courses*, ACM, 1.

KOPTA, D., IZE, T., SPJUT, J., BRUNVAND, E., DAVIS, A., AND KENSLER, A. 2012. Fast, effective bvh updates for animated scenes. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ACM, 197–204.

LAINE, S., KARRAS, T., AND AILA, T. 2013. Megakernels considered harmful: Wavefront path tracing on gpus. In *Proc. High-Performance Graphics*.

MOON, B., BYUN, Y., KIM, T.-J., CLAUDIO, P., KIM, H.-S., BAN, Y.-J., NAM, S. W., AND YOON, S.-E. 2010. Cache-oblivious ray reordering. *ACM Transactions on Graphics (TOG) 29*, 3, 28.

NOVÁK, J., HAVRAN, V., AND DACHSBACHER, C. 2010. Path regeneration for interactive path tracing. *Proc EUROGRAPHICS Short Papers*.

NVIDIA, C., 2012. Nvidia cuda programming guide.

VAN ANTWERPEN, D. 2011. Improving simd efficiency for parallel monte carlo light transport on the gpu. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, ACM, 41–50.

WALD, I., SLUSALLEK, P., AND BENTHIN, C. 2001. Interactive distributed ray tracing of highly complex models. In *Rendering Techniques 2001*. Springer, 277–288.

WALD, I. 2011. Active thread compaction for gpu path tracing. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, ACM, 51–58.

WESTIN, S. H., ARVO, J. R., AND TORRANCE, K. E. 1992. *Predicting reflectance functions from complex surfaces*, vol. 26. ACM.

ZHOU, K., HOU, Q., WANG, R., AND GUO, B. 2008. Real-time kd-tree construction on graphics hardware. In *ACM Transactions on Graphics (TOG)*, vol. 27, ACM, 126.