Cache-Oblivious Ray Reordering

Bochang Moon Yongyoung Byun Tae-Joon Kim
Pio Claudio Sung-Eui Yoon

CS/TR-2009-314

May, 2009

KAIST
Department of Computer Science

Cache-Oblivious Ray Reordering

Bochang Moon Yongyoung Byun Tae-Joon Kim Pio Claudio Sung-Eui Yoon
KAIST (Korea Advanced Institute of Science and Technology)

Dept. of CS, KAIST, Technical Report CS-TR-2009-314, May., 2009
URL: http://sglab.kaist.ac.kr/CORR/

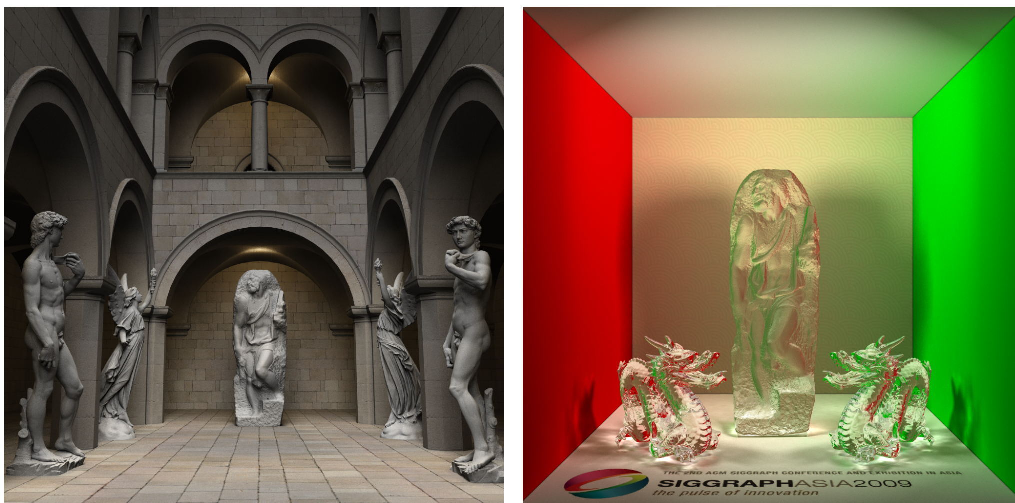

Figure 1: *The left and right images show the results of our method applied to path tracing and photon mapping respectively. The left image shows a St. Matthew model, two Lucy, and two David models in a Sponza model. This scene consists of 104 million triangles, requiring 12.8 GB for the original meshes and their acceleration hierarchies. The right image shows a transparent St. Matthew model in the Cornell box with two transparent dragon models. This scene has 128 M triangles and takes 15.7 GB. These two global illumination methods generate many incoherent rays to render these images. By reordering such rays, we achieve more than one order of magnitude performance improvement in a machine with 4 GB main memory, compared to without reordering rays. This performance improvement is caused by the improved ray coherence, which results in reducing cache misses for the L1/L2 caches, main memory, and disk during the ray tracing.*

Abstract

We present a novel, cache-oblivious ray reordering method for ray tracing. Many global illumination methods such as path tracing and photon mapping use ray tracing and generate lots of rays to simulate various realistic visual effects. However, these rays tend to be very incoherent and show lower cache utilizations during the ray tracing of models. In order to address this problem and improve the ray coherence, we propose a novel *hit point heuristic* (HPH) to compute a coherent ordering of rays. The HPH uses the hit points between rays and the scene as a ray reordering measure. We reorder rays by using a space filling curve based on their hit points. Since a hit point of a ray is available only after performing the ray intersection test with the scene, we compute an approximate hit point for the ray by performing an intersection test between the ray and simplified representations of the original models. Our method is a highly modular approach, since our reordering method is decoupled from other components of common ray tracing systems. We apply our method to photon mapping and path tracing and achieve more than an order of magnitude performance improvement for massive models that cannot fit into main memory. Also, our method shows a performance improvement even for ray tracing small models that can fit into main memory. This performance improvement for small and massive models is caused by reducing cache misses occurring in the L1/L2 caches, main memory and disk. This result demonstrates the cache-oblivious nature of our method, which works for various kinds of cache parameters. Because of the cache-obliviousness and the high modularity, our method can be widely applied to many existing ray tracing systems and show performance improvements with various models and machines that have different caches.

1 Introduction

Ray tracing has been widely used as the main rendering engine of various global illumination methods (e.g., path tracing and photon mapping). Typically, ray tracing generates lots of primary, secondary, and shadow rays, in order to simulate realistic rendering effects (e.g., soft shadows, reflections, caustics, motion blur, etc.). However, ray tracing has been still known to be slow to provide these realistic visual effects.

In order to improve the performance of ray tracing, a lot of studies have been done on designing efficient intersection tests, constructing efficient acceleration hierarchies, and exploiting data-level parallelism using the SIMD functionality and GPUs [Shirley and Morley 2003; Pharr and Humphreys 2004; Wald et al. 2007]. Most research has focused on improving the performance of ray tracing with primary rays. However, the focus has been recently shifted towards efficiently handling secondary rays that can provide realistic visual effects.

It has been widely known that secondary rays generated for simulating realistic visual effects show a low ray coherence and thus low cache utilizations during the processing of these rays with meshes and their acceleration hierarchies. One of the main challenges to efficiently handle secondary rays, therefore, is to achieve a high ray coherence and cache utilizations during the processing of rays. This problem of achieving high cache utilizations is becoming more important, since there is the widening gap between the data access speed and the data processing speed [Hennessy et al. 2007].

In order to achieve a high cache coherence for ray tracing, two orthogonal and complementary approaches, *layout reordering* and *ray reordering*, have been studied. Layout reordering methods [Sagan 1994; Yoon et al. 2008] aim to compute cache-coherent layouts of meshes and hierarchies such that data elements (e.g., vertices, triangles, and nodes) that are close in meshes and hierarchies are also closely stored in their one dimensional data layouts in main memory and external drive.

Although meshes and hierarchies are stored coherently in their layouts, the data access pattern on these layouts should be coherent as well, in order to design cache coherent ray tracing. A few ray reordering techniques [Pharr et al. 1997; Navratil et al. 2007; Budge et al. 2009] have been proposed. The seminal ray reordering method proposed by Pharr et al. [1997] does not process each ray as it is gen-

erated. Instead, the method queues rays into ray buffers associated with regions of the mesh and processes these regions in a coherent manner to reduce the number of expensive disk I/O accesses. Most other ray reordering methods are based on variations of this ray reordering framework. The original method proposed by Pharr et al. uses a scheduling grid and sorts rays into each grid cell during the scene traversal. Other techniques have extended this method to use an acceleration hierarchy and sort rays into nodes of the hierarchy during the hierarchy traversal, while considering available cache information.

These methods essentially exploit the information about whether a data is cached or not given a cache and sort rays depending on the data access pattern during the scene or hierarchy traversal. Although this kind of approaches can achieve high cache utilizations during the ray tracing of models, it complicates the ray tracing system by coupling the traversal and the ray reordering algorithm. Furthermore, all of these prior methods focused only on either reducing L1/L2 caches for small models or reducing the disk I/O cache misses for massive models that cannot fit into main memory, because of the cache-aware nature of these methods.

Main contributions: In this paper, we present a novel, cache-oblivious ray reordering method to achieve high cache utilizations during the ray tracing of models for global illumination methods. Our approach decouples the ray reordering method from the hierarchy traversal to achieve a high modularity. In order to reorder rays, we propose a novel *hit point heuristic*, which uses hit points between rays and the scene as a ray reordering measure (Sec. 4). Since the hit point of a ray is only available once the ray is processed by traversing the hierarchy, we approximate the hit point by using a simplified model of the original model. We then use a space-filling curve to reorder rays based on their hit points. This enables our method to work with different cache parameters and to achieve high cache utilizations for various memory levels. We apply our ray reordering method to path tracing and photon mapping (Sec. 5). By reordering rays, we achieve more than an order of magnitude performance improvement compared to rendering without reordering rays for massive models that cannot fit into main memory. Moreover, our method shows a performance improvement for small models that fit into main memory, because of reduced L1/L2 cache misses. These results demonstrate the benefits of the cache-oblivious nature of our ray reordering method. Because of the high modularity and cache-obliviousness, our method can be widely applied to many existing ray tracing systems and can improve the performance for various models on different machines that have different caches. We conclude in Sec. 6 with future work.

2 Related Work

Ray tracing and global illumination methods have been well studied. Also, good surveys and books are available [Shirley and Morley 2003; Pharr and Humphreys 2004; Wald et al. 2007]. In this section, we review prior work related directly to our problem.

2.1 Computation Reordering

Computation reordering strives to achieve a cache-coherent order of runtime operations in order to improve program locality and reduce the number of cache misses. Computation reordering methods can be classified into either *cache-aware* or *cache-oblivious*. Cache-aware algorithms utilize the knowledge of cache parameters, such as cache block size [Vitter 2001]. On the other hand, cache-oblivious algorithms do not assume any knowledge of cache parameters [Frigo et al. 1999]. There is a considerable amount of literature on developing cache-efficient computation reordering algorithms for specific problems and applications [Arge et al. 2004; Vitter 2001]. In computer graphics, out-of-core algorithms [Silva et al. 2002], which are cache-aware methods, have been designed to handle massive models.

2.2 Cache-coherent Ray Tracing

There has been extensive research on exploiting the coherence in ray tracing. These can be classified into packet methods, layout reordering, and ray reordering methods.

Packet ray tracing: Neighboring rays can exhibit spatial coherence and utilizing this coherence can improve the performance of ray tracing. Earlier attempts include beam tracing [Heckbert and Hanrahan 1984]. Wald et al. [2001] exploited the coherence of primary and shadow rays by grouping rays into packets and utilizing the SIMD functionality of modern processors. Reshetov *et al.* [2005] proposed an algorithm to integrate beam tracing with the kd-tree spatial structure and were able to further exploit coherence of primary and shadow rays. There have been a few ray reordering methods that can utilize the SIMD functionality for secondary rays [Boulos et al. 2008; Gribble and Ramani 2008]. These ray reordering methods for the SIMD utilization can be performed on rays reordered by our method.

Layout reordering: The order of data stored in memory or external drives can affect the performance of ray tracing given the widely used block-fetching caching scheme [Yoon et al. 2008]. In this caching scheme, blocking related nodes in a cluster can reduce the number of cache misses. The *van Emde Boas* layouts of trees [van Emde Boas 1977] are constructed by performing a recursive blocking to nodes. Havran analyzes various layouts of hierarchies in the context of ray tracing and improves the performance by using a compact layout representation of hierarchies [Havran 1997]. Yoon and Manocha [2006] developed cache-efficient layouts of hierarchies for ray tracing. Also, there are a few cache-coherent mesh layouts [Yoon and Lindstrom 2006; Sagan 1994].

Ray reordering: To reorder primary rays, space-filling curves like Z-curves [Sagan 1994] have been used. Mansson et al. [2007] showed coherence among secondary rays based on their proposed ray coherence measures. However, it was not demonstrated to achieve a higher runtime performance based on their proposed ray reordering heuristics. Pharr et al. [1997] proposed a ray reordering method for ray tracing massive models that cannot fit into main memory. Their method uses a scheduling grid for queueing rays and processes rays in a coherent manner, while considering the available cache information. Steinhurst et al. [2005] reorder kNN searches of photon mapping to reduce the memory bandwidth. Navratil et al. [2007] presented a ray scheduling approach that improves a cache utilization and reduces DRAM-to-cache bandwidth usage. Budge et al. [2009] employed a ray reordering method to utilize hybrid resources such as multiple CPUs and GPUs. These techniques are based on Pharr et al.'s ray reordering method, which couples the ray reordering and the scene traversal. By doing so, these methods can easily know which parts of meshes and hierarchies are accessed and cached during the processing of rays. As a downside of coupling the ray reordering and scene traversal, the modularity of these methods is lowered.

2.3 Ray Tracing Massive Models

Ray tracing massive models has been studied well. In-core techniques exist to perform the ray tracing of massive datasets [DeMarle et al. 2004; Stephens et al. 2006] by using large, shared memory systems. There are also out-of-core techniques including latency hiding [Wald et al. 2004]. There are different approaches aiming at designing compact representations, by applying the quantization on acceleration hierarchies [Cline et al. 2006], reducing costs of representing meshes and hierarchies [Lauterbach et al. 2008], or efficient culling techniques [Reshetov 2007]. These methods can be combined with our proposed method to further improve the performance of ray tracing massive models.

3 Overview

In this section, we discuss the ray coherence of different rays and briefly explain the overall approach of our method that increases the

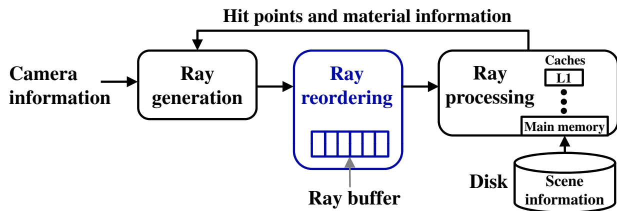

Figure 2: *This figure shows different modules of our ray reordering framework. Our main contribution is the hit point heuristic (HPH) based ray reordering method employed in the ray reordering module.*

ray coherence and thus cache utilizations during the ray tracing.

3.1 Ray Coherence

Ray tracing generates a lot of rays to simulate various visual effects. These rays can be classified into primary, shadow, and secondary rays. Primary rays are known to show a high coherence during the hierarchy traversal and mesh accesses. Space-filling curves such as Z-curves have been used to reorder primary rays [Pharr et al. 1997], based on positions of primary rays in the image plane. Once a primary ray has intersected with an object, shadow rays to lights and secondary rays (e.g., reflection rays), depending on the material property of the intersected object, are generated. Since light positions can be arbitrary and the intersected triangle can have an arbitrary normal, shadow and secondary rays generally have a lower coherence than primary rays. If rays are incoherent, then the data access pattern on the acceleration hierarchies and meshes can be incoherent. This incoherence may result in a high number of cache misses in various memory levels (e.g., L1/L2 caches, main memory, disk) and lower the runtime performance. Therefore, processing rays in a cache-coherent manner is critical to design cache-coherent ray tracers.

3.2 Ray Reordering Framework

In order to reorder rays, we use a ray reordering framework (see Fig. 2) extended from typical ray tracing systems. This framework consists of ray generation, ray reordering, and ray processing modules. The ray generation module constructs rays including primary, secondary, and shadow rays. The ray processing module takes each ray and finds a hit point between the ray and the scene by accessing acceleration hierarchies and the meshes of the scene. Also, the ray processing module performs shading based on the hit point and its corresponding material information. If we have to generate shadow and secondary rays, the ray processing module sends the hit points and material information to the ray generation module. Typical ray tracing systems consist of only these two modules and process rays as they are generated without reordering rays.

In addition to these modules, we also use the ray reordering module. The ray reordering module maintains a *ray buffer* that can hold a user defined number of rays. Once the ray generation module constructs rays, these rays are stored in the ray buffer and then reordered in a way such that meshes and hierarchies are accessed in a cache-coherent manner during the processing of reordered rays in the ray processing module. Note that our ray reordering framework is similar to previous ray reordering methods [Pharr et al. 1997; Navratil et al. 2007; Budge et al. 2009]. A main difference of our framework over these prior methods is that we decouple the ray reordering module from other modules. Therefore, our method achieves a high modularity and is easily applicable to existing ray tracing systems.

Given this ray reordering framework, the key component that governs the performance improvement is the ray reordering method. To maximize the benefits of the reordering method, the overhead of reordering should be kept small. We propose a simple cache-oblivious reordering method that has a low reordering overhead, increases the cache coherence, and improves the performance of ray tracing models that have different model complexities.

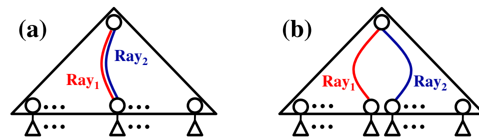

Figure 3: *These two figures show data access patterns on the hierarchy during the processing of two different rays, whose hit points are close to each other. The difference between the left and right figures is that two rays' directions are similar in the left, but different in the right.*

Cache-coherent layouts of meshes and hierarchies: Our ray reordering method works on the assumption that geometrically close mesh data (e.g., vertices or triangles) and topologically close hierarchy data (e.g., nodes) are also stored closely in their corresponding mesh and hierarchy layouts respectively. There are many layouts satisfying such a property for meshes [Sagan 1994; Diaz-Gutierrez et al. 2005; Yoon and Lindstrom 2006] and for hierarchies [van Emde Boas 1977; Havran 1997; Yoon and Manocha 2006]. In our implementation, we use cache-oblivious layouts of meshes and hierarchies [Yoon and Lindstrom 2006; Yoon and Manocha 2006]

4 Cache-Oblivious Ray Reordering

In this section, we introduce our cache-oblivious ray reordering method.

4.1 Hit Point Heuristic

To reorder rays, we propose a novel *hit point heuristic* (HPH). A hit point of a ray is defined as the first intersection point computed between the ray and the scene, starting from the ray's origin. The main idea of the HPH method is to reorder rays based on their hit points using a space-filling curve (e.g., Z-curve). The rationale why we use the hit point of a ray as a reordering measure is twofold. First, if the hit points of rays are geometrically close to each other, then the mesh regions accessed during the processing of these rays are likely to be close too. Second, suppose that a hierarchy is decomposed into lower and upper regions. Lower regions of the hierarchy are closer to leaf nodes and upper regions of the hierarchy are closer to the root node of the hierarchy. Then, the lower regions of the hierarchy accessed during the processing of rays whose hit points are close are likely to be close too because of the same reason that were for meshes (see Fig. 3). Although hit points of rays are close to each other, these rays' directions may be very different. In this case, their access pattern on upper regions of the hierarchy may be very different (see Fig. 3-(b)). However, the size of these upper regions of the hierarchy is relatively small compared to those of lower regions of the hierarchy. Also, the upper regions of the hierarchy are accessed by almost all the rays and thus are unlikely to be unloaded from the cache. Therefore, we may not get additional cache misses during the processing of rays with the upper regions of the hierarchy. As a result, we conclude that hit points between rays and the scene are more important features to our problem than ray directions and ray origins, which have been widely considered as reordering measures in most prior works.

An issue of the HPH method is that it requires hit points between rays and the scene to reorder rays. However, computing these hit points requires processing rays by traversing the hierarchy and accessing the mesh, which may cause a high number of cache misses that we attempted to avoid by reordering. To address this problem, we compute approximate hit points efficiently by performing the intersection tests between rays and simplified representations of the original models.

4.2 Approximate Hit Points

We compute a simplified mesh of the original model using an out-of-core mesh simplification method [Yoon et al. 2008]. This simplification method decomposes an input model into a set of clusters, each of which can be stored in main memory. Then, we simplify each cluster one by one. In order to compute approximate hit points

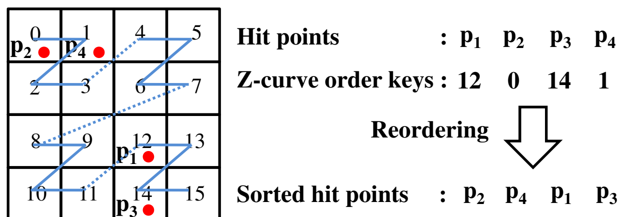

Figure 4: *This figure shows an ordering of hit points with the Z-curve ordering of cells in the uniform grid.*

that are close to the exact hit points, the simplified model should be geometrically similar to the original model. We use quadrics and choose edge collapses in an increasing order of simplification errors for each cluster by using a heap [Garland and Heckbert 1997] within each cluster. While simplifying each cluster, we also allow simplifying edges that span multiple clusters. For a simplified mesh, we set the bounding box of the simplified model to be the bounding box of the original model. Therefore, if a ray does not intersect with the bounding box, it is guaranteed that the ray does not intersect with the original model.

For each simplified mesh, we build a hierarchy in the same manner as building the hierarchy for the original model. In order to reduce the overhead of computing hit points with the simplified meshes at runtime, we drastically simplify the models. In our tests, we use simplified models consisting of 2% of the complexity of the original models. We found that this strikes a good balance between the overhead of our method and the approximation quality and thus achieves the best performance improvement of using our ray reordering method (see Sec. 5.1).

To compute approximate hit points of rays, we perform intersection tests between the rays and the simplified models of the scene. If a ray intersects with one primitive of the simplified models, we use the hit point for the ray reordering by using a space-filling curve. If the ray does not intersect with any primitives of simplified models, but one of the bounding boxes of the original models, we use the intersection point between the ray and the bounding boxes as a *virtual hit point* and use it for the ray reordering. For other rays that do not intersect with any of the bounding boxes, we terminate the processing of these rays, since it is guaranteed that they do not intersect with the original models of the scene. We use the computed approximate hit points only for reordering, not for other computations (e.g., shading).

4.3 Space-Filling Curve based Reordering

Once we compute approximate hit points for rays stored in the ray buffer, we reorder these rays by using a Z-curve, a simple space-filling curve. Since a Z-curve is defined in a uniform structure, we place hit points in a variation of a grid structure and compute ordering keys for these hit points and their corresponding rays by using a Z-curve ordering of cells in the grid structure. More specifically, we propose to use a two-level nested grid structure in order to represent more points uniquely for reordering with low time and space overheads.

The two-level grid structure consists of a high-level grid, where each cell of the high-level grid contains a low-level grid. Each high-level and low-level grid has 2^{3c} grid cells, where 2^c is the number of cells in each x, y, and z dimension. Since we use the two-level grid structure, our two-level grid has 2^{6c} cells. Extents of the high-level grid are set to be the bounding volume of the scene. We initialize each cell of the high-level grid to have a unique ordering key value computed from a Z-curve ordering of cells. An example of the Z-curve ordering of uniform grid cells is shown in Fig. 4. Given the three dimensional coordinates of a hit point, we can get an ordering key by referring the grid cell containing the point. We can easily compute the grid cell containing the point by using a simple hash function that computes the grid index from the coordinates of the point. Let us call the ordering key computed from the high-level grid *high-level ordering key* (K_h).

Scene	Rendering time (hour)		Num. of Disk I/O Accesses (M)	
	W/O Re.	W/ Re.	W/O Re.	W/ Re.
Path tracing	48.72	3.22	24.04	2.05
Photon mapping	165.62	12.61	62.54	8.13

Table 1: *This table shows the overall rendering time and the number of the disk I/O accesses that occurred during the generation of a rendering image for each benchmark.* **W/O Re.** *and* **W/ Re.** *represent without reordering and with reordering rays respectively.*

We also compute a *low-level ordering key* (K_l) of a point by referring to a low-level grid whose extents are set to the cell of the high-level grid that contained the point. Note that we can use the same grid structure and pre-computed ordering key values for both the high-level and low-level grids. Only the hash functions that compute grid indices from coordinates of points are different between the high-level and low-level grids. Once we compute the high-level and low-level ordering keys, our final ordering key value, K_f, of a ray is defined as $K_f = K_h * 2^{3c} + K_l$, where 2^{3c} indicates the number of low-level cells contained in each cell of the high-level grid.

In our current implementation, we choose c to be 6. In this case, our two-level grid structure decomposes the bounding volume of the scene into more than 68 billion uniform-sized cells. Therefore, most final ordering keys computed from rays are likely to be unique with models that we can have in practice. Also, our two-level grid structure has very little overhead of computing the ordering key values and requires only 2 MB to store the pre-computed Z-curve ordering key values for the high-level and low-level grids. We also tried Hilbert-curves, but found that using Z-curves has a lower memory requirement given our grid structure, while having only minor performance degradation (e.g., 2%) to Hilbert-curves.

Once we compute the ordering key values for rays, we sort rays based on the ordering key values. We use the 2-way merge sort due to its simplicity. After sorting rays using their associated approximate hit points, sorted rays are processed in the ray processing module.

5 Results and Discussions

We have implemented and integrated our ray reordering module in a CPU-based out-of-core ray tracing system that uses bounding volume hierarchies (BVHs) with axis-aligned bounding volumes for models. We use 512 by 512 image resolutions and perform various tests with a machine consisting of a 3.0 GHz processor, a disk that supports a sequential reading performance of 101 MB per second, and 4 GB memory with the 32 bit Windows unless mentioned otherwise. Although the machine has 4 GB main memory, all the programs in the 32 bit Windows can use only up to 3.25 GB. Also, the Windows OS in our test machine uses about 0.2 GB. Therefore, our ray tracer can use up to about 3.05 GB.

Our ray buffer consists of in-core and out-of-core parts. We allocate 100 MB of the main memory space to an *in-core* ray buffer. Once the in-core buffer is full, we push these rays into an *out-of-core* ray buffer on the disk and then store the next rays in the in-core ray buffer. If there are no more rays that we can generate, we sort the rays stored in the in-core and out-of-core ray buffers. We test our method with two global illumination methods: path tracing and photon mapping, both of which generate many incoherent rays to produce realistic visual effects. We generate primary rays in Z-curves for all the tests.

Path tracing: The left image of Fig. 1 shows an unbiased rendering image of the St. Matthew, two Lucy, and two David models in the Sponza scene using a path tracing method [Shirley and Morley 2003; Pharr and Humphreys 2004]. This scene consists of 104 M triangles; we do not use any instancing for duplicate models. BVHs and meshes of models in the scene take 12.8 GB. Since our ray tracer with the 32 bit test machine can use only 3.05 GB, the machine can cache 23.8% of all the data for our out-of-core ray tracer. To illuminate the scene, we use 8 area lights. We generate 100 primary rays (i.e., paths) per pixel and use a simple importance sampling by generating shadow rays to the lights. We use the Russian roulette method to determine the path length. In this scene and machine

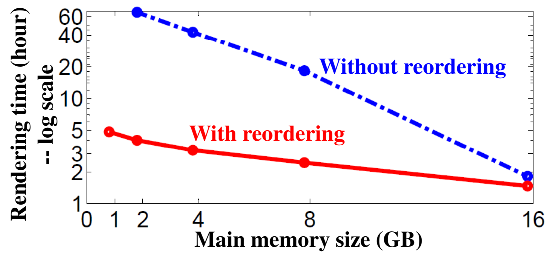

Figure 5: *This graph shows the rendering time of path tracing in the Sponza scene with different physical main memory sizes. Our ray tracer can utilize the memory space except for the space of the Windows OS, which is about 0.2 GB.*

configurations, our method achieves a 15.1 times performance improvement over rendering without reordering rays. We measure the number of the disk I/O accesses occurring during the accessing of meshes and BVHs (see Table 1), by using the Windows built-in performance monitor tool, *perfmon*. By reordering rays, we reduce the number of the disk I/O accesses that occurred without reordering rays by 91.4%. We also measure the average disk I/O access performance (MB/sec.) per disk I/O access. We found that reordering rays improves the disk I/O access performance by 183%. This is because the disk I/O accesses become more coherent and the disk can process these I/O accesses with a higher reading performance during the random accesses on BVHs and meshes. Because of these two factors, the reduction of disk I/O accesses and the improvement of disk I/O performance, we achieve more than an order of magnitude performance improvement by caching only 23.8% of all the data in main memory.

Photon mapping: The right image of Fig. 1 shows a rendering of the transparent St. Matthew and two transparent dragon models in the Cornell box scene using the photon mapping method [Jensen 2005]. This scene consists of 128 M triangles and takes 15.7 GB for its meshes and BVHs; therefore, the machine can cache only 19% of the total model size. We use 4 lights, generate 25 primary rays per pixel and 10 final gathering rays, and use 100 samples for the irradiance estimation for various performance tests. In this configuration, our method achieves a 13.1 times improvement compared to rendering without reordering rays. By reordering rays, we reduce 87% of the disk I/O accesses and improve the disk I/O performance by 195%.

5.1 Analysis

We discuss various factors that affect the performance of our method with the path tracing benchmark of the Sponza scene in this section, unless mentioned otherwise.

Performance vs. cache size: We measure the overall rendering time of path tracing the Sponza scene, as a function of the available memory size with and without using our ray reordering method (see Fig 5). For this test, we use a 64 bit machine; note that the OS uses 0.2 GB space from the physical main memory. When we use 16 GB main memory, the whole data of the scene can be uploaded into main memory. Even in this case, our method improves the performance by 24% over not using our reordering method, because our method improves the cache utilizations of L1/L2 caches. As we decrease the memory size, the performance of ray tracing also decreases. Nonetheless, the performance with our ray reordering method decreases more gracefully. By caching 1.8 GB, 14.1% of the whole data, in main memory, our method shows a 16 times improvement. Even when the available memory size is 0.8 GB, 6.2% of the whole data, our method can render the Sponza scene without I/O thrashing.

Cache-oblivious nature of our method: Our method uses Z-curves for reordering rays and has the cache-oblivious property caused by using the space-filling curve [Yoon and Lindstrom 2006] that works with different cache parameters. Therefore, it can reduce cache misses for various caches including L1/L2 caches, main memory, and disk. To demonstrate the cache-oblivious property of our method, we test our method with photon mapping of the Armadillo model consisting of 346 K triangles in the Cornell box (see the accompanying report for its image). The whole data of this small scene takes 43.5 MB, which fits into main memory. In this case, we reorder rays when our in-core ray buffer is full, instead of dumping rays stored in the ray buffer to out-of-core ray buffer. In this scene, our method shows 23% performance improvement by reordering rays. We also measure the L2 cache miss ratios by simulating the 6MB wide 24-way set-associative L2 cache of our test machine. We observe more than two times cache miss reduction by reordering rays compared to without reordering rays. Also, even when we perform path tracing of the 12.8 GB Sponza scene with 16 GB main memory, we achieve 24% performance improvement because of the reductions of the L1/L2 cache misses.

Multi-core architectures: We also test our method in the 32 bit machine with a quad-core CPU. Our reordering method can be easily parallelized since computing ordering keys of hit points is done by simply accessing a grid cell referred by a hash function and reading the ordering key stored in the cell. Also, the 2-way merge sort method that we used for sorting rays can be easily parallelized. We measure the performance improvement by reordering rays when we use four threads for ray tracing and our reordering method. By reordering rays, we achieve 10.5 times improvement over without reordering rays when we use four threads in the quad-core CPU machine.

Performance vs. complexity of simplified models: The complexity of simplified models can affect the performance improvement of our ray reordering method. We measure the performance improvement caused by our reordering method with different complexities of simplified models. We achieve the highest performance when we use simplified models whose model complexities are 2% of original models. Moreover, we also found that the performance of our method does not decrease much as we use drastically simplified models (e.g., 0.0125% of the original models for the simplified models). We attribute this result to the high quality of our simplification method based on quadrics and edge collapses. We also measure the total overhead of our method, which consists of computing approximate hit points and sorting rays stored in the ray buffer. We found that the overhead of our method is 6% of the total rendering time when we use 2% of the original model complexity for the simplified models.

Limitations: Our method has certain limitations. Our ray reordering method like other ray reordering methods may not work with shaders that do not allow deferred shading, though most general shaders work with the deferred shading. Also, there is no guarantee that our method will improve the performance of ray tracing because of the overhead of our method. Nonetheless, we achieved performance improvements with all the tests with our benchmarks.

5.2 Comparisons

There have been a few ray reordering methods that attempt to achieve a higher ray coherence and the performance improvement for ray tracing [Pharr et al. 1997; Navratil et al. 2007; Budge et al. 2009]. It is very hard to directly compare these methods with our method. However, our method has two main advantages over the prior works: the high modularity and the cache-obliviousness.

Most previous methods reorder rays as they traverse their scenes or acceleration hierarchies [Pharr et al. 1997; Navratil et al. 2007; Budge et al. 2009], because the data access patterns of rays are known during the scene or hierarchy traversal. The main benefit of these methods is that since the data access patterns of rays to the hierarchies and meshes are known during the traversal, sorting rays with this information can result in a low number of cache misses. However, this approach will require a tight integration between the ray reordering module and the ray processing module, causing a complication to the overall ray tracing system and a major restructuring of existing systems in order to use these reordering methods. One

Complexity of simplified model	0.0125%	0.05%	2%	8%
Rendering time (sec.)	11,983	11,691	11,591	12,053
Overhead (sec.)	674	688	797	1,236

Table 2: *This table shows the overall rendering time and the total overhead of our method as a function of model complexity of simplified models, represented in the percentage of the original model complexity, in the Sponza benchmark.*

prior method [Mansson et al. 2007] tried to achieve a high ray coherence by reordering rays based on a ray coherence metric instead of coupling the ray reordering and the hierarchy traversal. However, a higher runtime performance was not demonstrated based on their ray coherent metrics compared to not reordering rays. On the other hand, our method shows more than an order of magnitude performance improvement based on our HPH method, while our method decouples the ray reordering module from the ray processing module.

Also, all the prior works focused on improving the performance of ray tracing for either massive models that cannot fit into main memory [Pharr et al. 1997] or small models [Navratil et al. 2007] that fit into main memory, but do not in the L1/L2 caches. Since these methods use cache-aware ray reordering approaches that require the knowledge of parameters of a cache, these methods reduce the number of cache misses only with the cache. On the other hand, our method is cache-oblivious and is not optimized with a particular cache parameter. Instead, it works with various caches, including the L1/L2 caches, main memory, and disk. Since it is very important to reduce the number of cache misses with these different caches, the cache-obliviousness of our method enabled us to achieve the performance improvement for different tested models that span from the Armadillo model consisting of 346 K triangles that fits to main memory to the St. Matthew model consisting of 128M triangles that cannot fit into main memory.

We also compare the performance of our method with the seminal ray reordering work proposed by Pharr et al. [1997] that uses a scheduling grid and processes rays to maximize the cache utilization while considering the cache information. According to the results reported in their paper [Pharr et al. 1997], this method takes about 2.15 times longer path tracing time when the method caches 14.1% of the total data of a lake scene compared to the best result achieved by using memory that can contain the whole data. In our path tracing benchmark using our test machine that can cache 14.1% of all the data, our method takes 1.73 times longer time compared to our best performance, achieved when we use 16 GB main memory. Moreover, if we run the cache-aware method proposed by Pharr et al. [1997] in our test machine with the same cache configuration, it may not show more than 1.73 times performance improvement than our cache-oblivious method, since the cache-aware method cannot reduce the number of cache misses occurring when we have enough memory that can contain all the data. Also, the cache-aware method showed a lower performance compared to without reordering rays, when all the data are stored in main memory because it does not reduce the cache misses of the L1/L2 caches and their reordering overhead may be high. Although we are comparing apples with oranges, we argue that our cache-oblivious method can show comparable performance to the cache aware method [Pharr et al. 1997], unless our method shows a higher performance.

6 Conclusion and Future Work

We have presented a novel, cache-oblivious ray reordering method that achieves the performance improvement for various models. We have proposed a novel hit point heuristic (HPH) as a ray reordering measure and used the Z-curve to reorder rays based on their approximate hit points, which are computed from simplified models of the original models. We have applied our method to path tracing and photon mapping, both of which require lots of incoherent rays to generate realistic visual images with massive models that cannot fit into main memory. By reordering these rays, we have achieved more than an order of magnitude performance improvement by caching less than 20% of all the data. Moreover, our method shows a performance improvement for small models that can fit into main memory. This performance improvement is caused by reducing the cache misses of the L1/L2 caches. In addition to having the cache-oblivious property, our method can be easily applied to many existing ray tracing systems with a minor modification, since our ray reordering module is decoupled from other common ray tracing modules. Because of the high modularity and cache-obliviousness that can improve the performance for a wide set of models, our method can be useful and widely applied to many existing and future ray tracing systems.

There are many exciting future directions lying ahead. Currently, we have tested our ray reordering method only with the CPU architecture. It will be very interesting how our method can be extended to handle incoherent rays and improve GPU cache utilizations in GPU-based global illumination methods. It will be also interesting to apply our method to hybrid ray tracers that run on both CPUs and GPUs. Also, we would like to extend our method to consider ray directions, in addition to hit points of rays. This may improve the ray coherence further. Finally, we presented a cache-oblivious ray reordering method for ray tracing in this paper. This idea can be applied to many different applications whose main bottleneck is in the data access time. Therefore, we would like to extend our current method to different computer graphics applications.

References

ARGE, L., BRODAL, G., AND FAGERBERG, R. 2004. Cache oblivious data structures. *Handbook on Data Structures and Applications*.

BOULOS, S., WALD, I., AND BENTHIN, C. 2008. Adaptive ray packet reordering. In *IEEE Symp. on Interactive Ray Tracing*, 131–138.

BUDGE, B., BERNARDIN, T., STUART, J. A., SENGUPTA, S., JOY, K. I., AND OWENS, J. D. 2009. Out-of-core data management for path tracing on hybrid resources. *Computer Graphics Forum (Eurographics) 28*, 2, 385–396.

CLINE, D., STEELE, K., AND EGBERT, P. K. 2006. Lightweight bounding volumes for ray tracing. *Journal of Graphics Tools 11*, 4, 61–71.

DEMARLE, D. E., GRIBBLE, C. P., AND PARKER, S. G. 2004. Memory-savvy distributed interactive ray tracing. In *EGPGV*, 93–100.

DIAZ-GUTIERREZ, P., BHUSHAN, A., GOPI, M., AND PAJAROLA, R. 2005. Constrained Strip Generation and Management for Efficient Interactive 3D Rendering. In *Computer Graphics International*, 115–121.

FRIGO, M., LEISERSON, C., PROKOP, H., AND RAMACHANDRAN, S. 1999. Cache-oblivious algorithms. In *Foundations of Computer Science*, 285–297.

GARLAND, M., AND HECKBERT, P. 1997. Surface simplification using quadric error metrics. In *SIGGRAPH 97 Proceedings*, 209–216.

GRIBBLE, C. P., AND RAMANI, K. 2008. Coherent ray tracing via stream filtering. In *IEEE Symposium on Interactive Ray Tracing*, 59–66.

HAVRAN, V. 1997. Cache sensitive representation for the bsp tree. In *Proc. of Compugraphics*.

HECKBERT, P. S., AND HANRAHAN, P. 1984. Beam tracing polygonal objects. In *SIGGRAPH*, ACM Press, New York, NY, USA, 119–127.

HENNESSY, J. L., PATTERSON, D. A., AND GOLDBERG, D. 2007. *Computer Architecture, A Quantitative Approach*. Morgan Kaufmann.

JENSEN, H. W. 2005. *Realistic Image Synthesis Using Photon Mapping*. AK Peters.

LAUTERBACH, C., YOON, S.-E., TANG, M., AND MANOCHA, D. 2008. ReduceM: Interactive and memory efficient ray tracing of large models. *Computer Graphics Forum (Proc. of EG Symp. on Rendering) 27*, 4, 1313–1321.

MANSSON, E., MUNKBERG, J., AND AKENINE-MOLLER, T. 2007. Deep coherent ray tracing. In *IEEE Symp. on Interactive Ray Tracing*.

NAVRATIL, P., FUSSELL, D., LIN, C., AND MARK, W. 2007. Dynamic ray scheduling to improve ray coherence and bandwidth utilization. In *IEEE Symposium on Interactive Ray Tracing*, 95–104.

PHARR, M., AND HUMPHREYS, G. 2004. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

PHARR, M., KOLB, C., GERSHBEIN, R., AND HANRAHAN, P. 1997. Rendering complex scenes with memory-coherent ray tracing. In *ACM SIGGRAPH*, 101–108.

RESHETOV, A., SOUPIKOV, A., AND HURLEY, J. 2005. Multi-level ray tracing algorithm. *ACM Trans. Graph. 24*, 3, 1176–1185.

RESHETOV, A. 2007. Faster ray packets - triangle intersection through vertex culling. In *IEEE Symp. on Interactive Ray Tracing*.

SAGAN, H. 1994. *Space-Filling Curves*. Springer-Verlag.

SHIRLEY, P., AND MORLEY, R. K. 2003. *Realistic Ray Tracing*, second ed. AK Peters.

SILVA, C., CHIANG, Y.-J., CORREA, W., EL-SANA, J., AND LINDSTROM, P. 2002. Out-of-core algorithms for scientific visualization and computer graphics. In *IEEE Visualization Course Notes*.

STEINHURST, J., COOMBE, G., AND LASTRA, A. 2005. Reordering for cache conscious photon mapping. In *Proc. of Graphics Interface*, 97–104.

STEPHENS, A., BOULOS, S., BIGLER, J., WALD, I., AND PARKER, S. 2006. An application of scalable massive model interaction using shared memory systems. *EG Symp. on Parallel Graphics and Visualization*.

VAN EMDE BOAS, P. 1977. Preserving order in a forest in less than logarithmic time and linear space. *Inf. Process. Lett. 6*, 80–82.

VITTER, J. 2001. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 209–271.

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

WALD, I., DIETRICH, A., AND SLUSALLEK, P. 2004. An Interactive Out-of-Core Rendering Framework for Visualizing Massively Complex Models. In *EG Symp. on Rendering*, 82–91.

WALD, I., MARK, W., GUNTHER, J., BOULOS, S., IZE, T., HUNT, W., PARKER, S., AND SHIRLEY, P. 2007. State of the art in ray tracing dynamic scenes. *Eurographics State of the Art Reports*.

YOON, S.-E., AND LINDSTROM, P. 2006. Mesh layouts for block-based caches. *IEEE Trans. on Visualization and Computer Graphics (Proc. Visualization) 12*, 5, 1213–1220.

YOON, S.-E., AND MANOCHA, D. 2006. Cache-efficient layouts of bounding volume hierarchies. *Computer Graphics Forum (Eurographics) 25*, 3, 507–516.

YOON, S.-E., GOBBETTI, E., KASIK, D., AND MANOCHA, D. 2008. *Real-Time Massive Model Rendering*. Morgan & Claypool Publisher.