Reordering for Cache Conscious Photon Mapping

Joshua Steinhurst Greg Coombe Anselmo Lastra

Department of Computer Science
University of North Carolina at Chapel Hill
{jsteinhu, coombe, lastra}@cs.unc.edu

Abstract

Photon mapping is a global illumination algorithm for generating and visualizing a sparse representation of the incident radiance on surfaces. Photon mapping places an enormous burden on the memory hierarchy. A 512×512 image using the standard *kd*-tree data structure requires more than 196GB of raw bandwidth to access the photon map. This bandwidth is a major obstacle to our long term goal of designing hardware capable of real time photon mapping.

This paper investigates two approaches for reducing the required bandwidth: 1) reordering the *k*NN searches; and 2) cache conscious data structures. Using a Hilbert curve reordering, we demonstrate an approximate lower bound of 15MB of bandwidth. This improvement of four orders of magnitude requires a prohibitive amount of intermediate storage. We then demonstrate two more cost-effective algorithms that reduce the bandwidth by one order of magnitude to 24GB with 1MB of storage. We explain why the choice of data structure can not, by itself, achieve this reduction. Irradiance caching, a popular technique that reduces the number of required *k*NN searches, receives the same proportional benefit as the higher quality photon gathers.

Key words: Global illumination, Graphics hardware

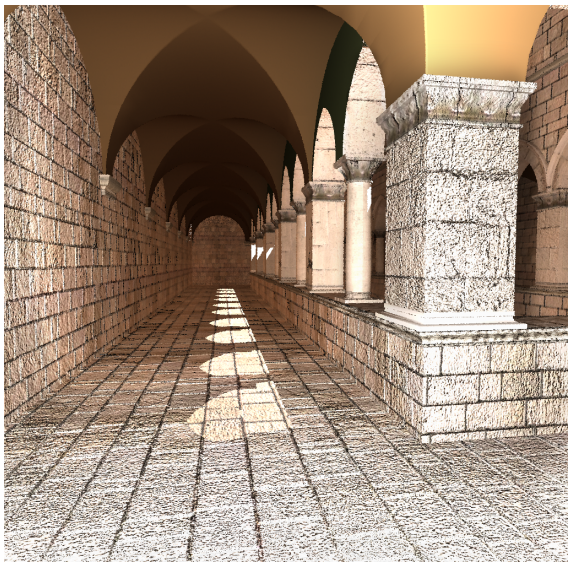


Figure 1: The bandwidth required to compute this image, which features indirect illumination and reflection, can be reduced to 4GB using 16×16 tiles and the Hilbert curve *k*NN search reordering (10GB without irradiance caching). *Model by Marko Dabrovic, RNA Studios.*

1 Introduction

Our long-term goal is to devise efficient hardware architectures for global illumination. The two major challenges of hardware design are the computation and the memory-bandwidth requirements. Whereas the number of transistors on a chip has increased dramatically, the off-chip memory-bandwidth has not. We expect to develop an architecture with many parallel processing elements sharing a standard memory interface. Therefore, this paper investigates approaches to reduce the bandwidth of photon mapping to levels that we believe can be attained in 3-5 years on a single PC expansion card. These approaches may also prove useful for software systems.

Photon mapping is a strong contender because it is a popular global illumination algorithm that can produce a wide range of realistic visual effects including indirect illumination, color bleeding, and caustics on complex diffuse, glossy and specular surfaces represented using arbitrary geometric primitives [13]. After a view independent preprocess, the cost of photon mapping is mostly due to two core operations: ray casting (single intersection of a ray with a scene) and *k*-Nearest-Neighbor (*k*NN) photon searches. The computation and bandwidth requirements of ray casting have received significant attention. Efficient ray casting algorithms have been devised for general purpose CPUs and custom hardware [25]. Pharr [19] used a space-filling curve on the screen to generate the eye rays in an order that increased the effectiveness of a geometry cache. Several researchers [5, 21, 23, 25] have described different ways of scheduling rays after they are generated to maximize geometry cache use. In this paper,

we assume that a memory efficient algorithm is used for ray casting, and focus on the second core operation, the *k*NN photon searches.

Each *k*NN search estimates the incident radiance on a non-specular surface by gathering the energy of the k photons closest to a query point and dividing by their approximate surface area. A typical image can require at least 100 *k*NN searches per pixel to obtain desired quality [13], which would consume approximately 196GB of bandwidth per 512×512 frame with a cached memory.

This paper studies the cache behavior of photon mapping and explores techniques to improve performance along two axes: 1) the order in which the *k*NN searches are performed; and 2) the data structure used to store the photon map.

We conducted experiments with four reordering techniques, presented in Section 3. Of these, the Hilbert curve [6, 17] reduces off-chip bandwidth by up to four orders of magnitude (to 15MB), but requires 1GB of intermediate storage. We present alternative techniques that achieve one order of magnitude improvement using only 1MB of intermediate storage. Irradiance caching [28], a popular technique that reduces the number of photon gathers, is shown to receive the same proportional reduction in bandwidth. Since data structures also impact the bandwidth, in Section 4 we analyze the performance of three data structures, *kd*-tree [2], Block Hashing [15] and *kdB*-tree [24].

2 Photon Mapping

Photon mapping is a two-step algorithm. The first step, photon tracing, shoots photons outward from the light sources into the environment. For a typical scene with indirect illumination, this requires shooting hundreds of thousands of photons [13]. These photons are probabilistically reflected, refracted and finally absorbed at diffuse or glossy surfaces. When a photon is absorbed, its location, power, and incident direction are stored in a view-independent photon map. For static scenes the photon map can be generated as a preprocess and reused for multiple viewpoints.

The second step of the photon mapping algorithm uses the photon map to estimate incident radiance on the surfaces. The viewpoint is fixed and an eye ray, $\vec{\omega}$, is cast into the scene for each pixel (u, v) of the final image. Multiple samples per pixel may be cast for antialiasing. At the point x where the eye ray intersects the scene geometry the direct illumination is computed and additional rays are cast to sample any specular reflection. The indirect diffuse illumination is computed from the data in the photon map.

Jensen describes two methods for visualizing the photon map: 1) direct visualization; and 2) final gather visualization. For the direct visualization method, illustrated in Figure 2, a *k*-Nearest-Neighbor (*k*NN) search is conducted in a small neighborhood around the point x using the photon map. Reasonable values for k are 100 or more [13]. Every photon in the photon map has a power $\triangle\Phi_p$ and direction $\vec{\omega_p}$. If a photon p is selected by the *k*NN search, these values are used with the surface reflectance properties (BRDF) f_r to compute the contribution to the reflected radiance. This is referred to as a *photon gather*. Unfortunately, direct visualization exhibits strong visual artifacts unless a very large number of photons are used [13].

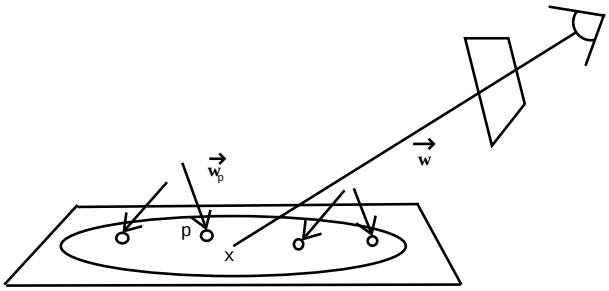


Figure 2: In direct visualization of the photon map a single photon gather is performed for each eye ray.

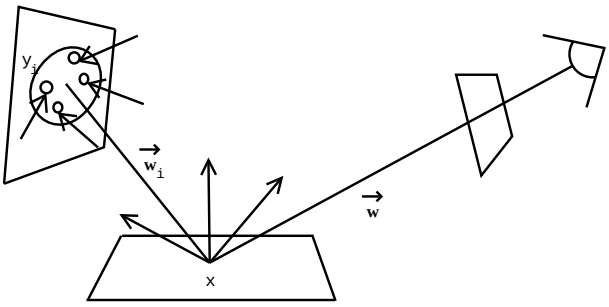


Figure 3: A higher quality estimate of incident radiance is obtained by using Monte Carlo integration to perform a final gather. A photon gather is performed at the end of each secondary ray, $\vec{\omega_i}$.

The final gather visualization, shown in Figure 3, estimates the rendering equation using a Monte Carlo integration at point x. The hemisphere surrounding x is sampled and N rays $\vec{\omega_i}$ are cast out into the scene. At each intersection point y_i a photon gather is performed. These results are then weighted by the BRDF of the surface at x and summed to get the indirect diffuse illumination. Our work uses the higher-quality final-gather visualization.

2.1 Irradiance Caching

A common bandwidth reduction technique is Irradiance Caching [28]. It avoids performing a final gather at each point x by interpolating previously computed irradiance estimates at nearby points. This works only when certain assumptions are met: 1) the surfaces must be purely diffuse; 2) the surface geometry and normals must be smooth. Even when these assumptions are valid, it can still produce noticeable visual artifacts. We show in Section 3 that Irradiance Caching reduces the bandwidth and computation requirements of a relatively simple scene dramatically. We also find that the bandwidth reduction techniques we develop in this paper remain effective with irradiance caching.

Christensen has described two alternative methodologies for irradiance caching. The first computes the irradiance at a subset of the photons during a preprocess and uses the photon map as the irradiance cache [3]. Recently, he described a software system using a hierarchy of irradiance voxels [4].

2.2 Bandwidth Requirements

If final gather visualization is used, it is common to perform at least 100 gathers per eye ray [13]. If each photon requires 20 bytes of storage [13], the minimum raw bandwidth for $k = 100$ and a 512×512 image is $512 \times 512 \times 100 \times 100 \times 20B = 50$GB. This figure is obtainable only with an oracle search function that examines exactly the photons that are needed. In practice, a *k*NN search in a *kd*-tree photon map data structure must examine more photons then are needed. Using the simulation framework described later in this section, we find that 166GB of raw memory traffic is generated in the absence of a cache. We explore the impact of alternative data structures in Section 4.

2.3 Block Memory Model

Like Ma and McCool [15], we assume that memory is relatively slow, is off-chip, and is accessed in blocks. Standard DRAM designs achieve peak bandwidth when the accesses are coherent, long, and aligned to page boundaries. This means that reading individual photons from scattered locations in memory is highly inefficient. The naive bandwidth estimates of 50GB and 166GB are generated with 20B reads of single photons scattered across memory. Assuming 20 bytes per photon and standard 128 byte cache lines [11], this means that about six photons fit in a cache line (with some extra room for data structure bookkeeping). Efficient algorithms should make use of several, if not all, of the six photons stored in a cache line. Several of the data structures that are discussed in Section 4 are explicitly organized in blocks that are the size of the cache lines in order to maximize the utility of each read.

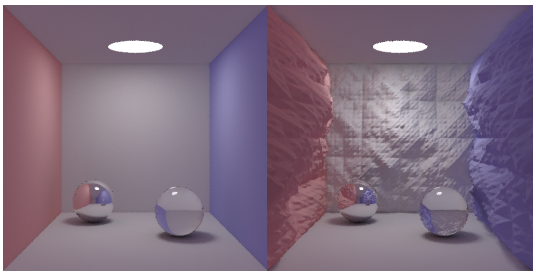


Figure 4: Our experiments use a variation of the Cornell box scene (left). When the geometric complexity was increased while maintaining the same illumination (right), the required bandwidth did not change significantly.

2.4 Experimental Setup

We modified two software packages for our experiments, an open source photon mapper [12] and `pbrt` [20]. We added a software cache simulator that analyzes the stream of memory references. The cache simulator models a fully-associative cache with Least Recently Used (LRU) eviction policy. Fully associative caches are costly to implement in hardware, so they are usually implemented as less optimal set-associative. However, it has been shown that, in the absence of a highly regular access pattern, reducing the associativity raises the number of block evictions and fetches by a constant factor of approximately 39% for an 8-way set-associative cache [9].

Following Jensen [13], our experiments were conducted using a variation of the Cornell box scene shown on the left of Figure 4. The box has two colored walls, a reflective sphere on the left, and a refractive sphere. Since a large proportion of the *k*NN search locations will likely be located on the flat walls, we performed additional experiments using the fractal box scene shown on the right side of Figure 4. The geometric complexity of the fractal scene leads to a more complex distribution of search locations while retaining the same overall illumination. Our experiments showed little difference in the bandwidth requirements between the two scenes.

3 Query Reordering

Caches reduce memory traffic by exploiting locality in the memory request stream. Using any reasonable data structure, each *k*NN search examines only a small portion of the photon map data structure. If a search is in the same region of the photon map as the previous search then a cache will be effective.

The key insight in this paper is that there exists a large amount of coherence amongst all of the *k*NN searches that is ignored by the standard final gather algorithm, re-

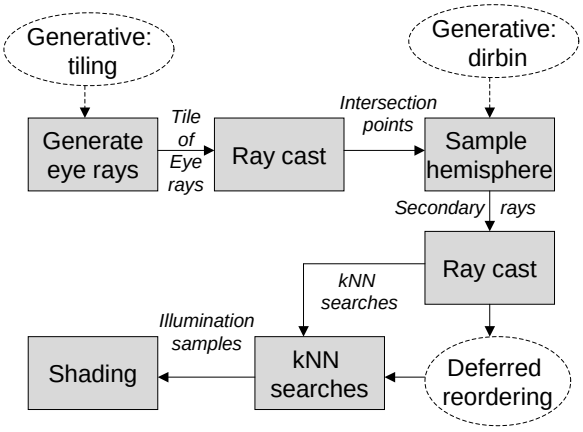


Figure 5: After the eye rays for each tile are generated, they are cast into the scene. At each intersection point the hemisphere of directions is sampled, possibly using generative reordering. The secondary rays are then intersected with the scene, determining where the *k*NN queries will take place. This list of queries may be reordered again before processing. The final result for each query is a contribution to an individual pixel. The results are accumulated in the framebuffer after shading.

sulting in poor cache behavior. By *reordering* the *k*NN searches, this coherence is exposed and memory bandwidth is dramatically reduced without changing the final image.

In this section we examine two different approaches for changing the order in which we process the *k*NN searches: 1) *generative reordering*, modifying the order in which searches are generated; and 2) *deferred reordering*, generating a list of *k*NN searches, Y, and reordering the list before performing the searches. These approaches are illustrated in Figure 5.

For the purpose of comparing the reordering algorithms, we fix the cache to be 128KB in size with 128B cache lines, use the *kd*-tree data structure and generate a 512×512 image of the Cornell box scene. The results of the following techniques can be compared in Figure 6 while Figure 7 formed the basis for our choice of cache size.

3.1 Generative reordering

The first set of techniques for improving the locality of *k*NN searches consists of generating the search locations in a coherent order.

Naive ordering. In the naive algorithm each pixel in the image is processed in scanline order. Each *k*NN search is processed as soon as it is generated and the result is stored directly at the destination pixel.

This algorithm requires 196GB of traffic from the cache to main memory. Each 128B cache line holds approximately six photons from the *kd*-tree; however, only one is usually accessed. Algorithms that do not account for the block nature of memory are inefficient because they cannot exploit spatial coherence.

Tiled reordering. In most scenes the eye rays from neighboring pixels will intersect the scene at points x in close proximity to each other [27]. There will therefore be a high degree of coherence among the origins of the rays cast during the Monte Carlo integration. This coherence can be exploited by breaking the screen into tiles and creating a list of associated query sites, $Y^{<a,b>}$. Each individual tile is processed in scanline order. A similar technique is commonly used in graphics rasterization hardware to improve texture memory locality [16]. The drawback to the tiled approach is that while the origins x of the rays used by the Monte Carlo integration are similar, the directions $\vec{\omega_i}$ remain spread across the hemisphere, as shown in Figure 3. In our experiments, which consisted of a relatively open room, the resulting search locations y_i remain too scattered throughout the scene to improve cache efficiency.

On highly specular surfaces the photon mapping algorithm spawns both reflective and transmissive rays. If a tiled reordering is used to reduce bandwidth, care must be taken to avoid polluting the cache. For each tile we generate a list of specular rays and process them after the *k*NN gathers for all the eye rays of this tile.

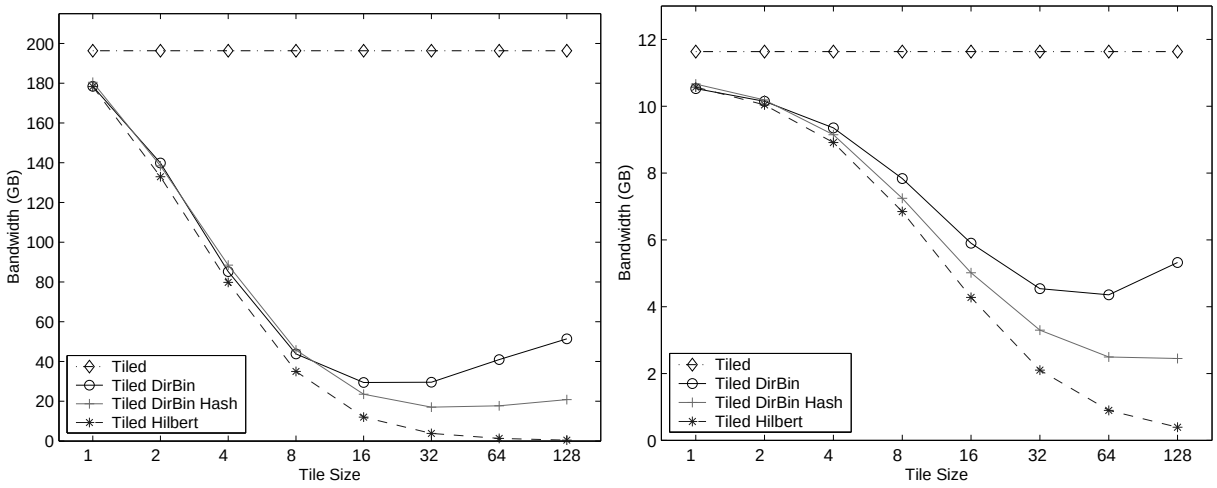
Tiled direction-binning reordering. The tiled algorithm can be improved by explicitly grouping the secondary rays by direction, $\vec{\omega_i}$, in addition to the implicit grouping by origin, x. The resulting groups of rays will be coherent and intersect the scene near to each other [27]. This generative ordering is implemented by performing multiple passes over the tile. Each pass only generates those rays, $\vec{\omega_i}$, that fall within a specified portion of the hemisphere.

The tiled direction-binned reordering algorithm requires less than a third of the naive algorithm's bandwidth for all tile sizes larger than 4×4. In our experiments this algorithm requires only 24GB of bandwidth with a tile size of 16×16. Larger tiles tend to cover multiple surfaces, reducing the coherence of the secondary rays.

We hypothesize that tiled direction-binning will work even better when importance sampling is used in the Monte Carlo integration. This is because the search locations y_i will be clustered in smaller areas of the hemisphere corresponding to the directions of strongest incident radiance.

3.2 Deferred reordering

The second set of techniques for improving the locality of the photon gathers consists of generating a list Y of the



(a) Without Irradiance Caching

(b) With Irradiance Caching

Figure 6: The tiled Hilbert curve reordering results in the lowest bandwidth for each tile size. Only tiles of moderate size are practical due to internal storage constraints. 16×16 and 32×32 are both feasible and perform well for the cost effective reorderings of tiled direction binned, both with and without hash reordering. When irradiance caching is used, (b), *k*NN search reordering retains the same proportional benefit.

search locations with one of the generative techniques. The elements $y_i \in Y$ are then reordered to form Y' before processing. Some techniques require the full list Y to be built before reordering, while others operate in a streaming fashion.

Decoupling the search location generation from the processing introduces computation and storage overhead. For each photon gather the following must be stored: the search location, the direction from which this point is viewed, a pointer to the local material properties, the destination pixel, and an RGB pixel contribution weight. Our system used 44 bytes for each deferred search. If 16×16 tiles are used with 100 gathers per pixel, then 1MB of intermediate storage is required. If all the searches for a 512×512 image are deferred then 1GB is required.

Hashed reordering. Several authors [10, 7] have explored hashing algorithms for *k*NN searches. We implemented a hashed reordering algorithm that has low computation costs and manageable memory use. This algorithm is similar to the ray-queue hashing of the GI-Cube [5]. This algorithm reorders a stream of *k*NN searches within a fixed-size window by hashing them into a set of buckets based on their 3D position. The expectation is that search locations that hash to the same bucket will be coherent.

The searches are removed sequentially from a single queue and passed to the *k*NN search processor. When that bucket is emptied the next largest bucket is processed until it is emptied. We have found that a very large number of queues is required to get even a modest reduction in bandwidth.

Tiled direction-binning Hashed reordering. The more coherent the initial list of search locations, the better the hash reordering will perform. A deferred reordering step can easily be performed on the result of a generative ordering. By combining tiled direction-binning with hashed reordering, $Y' = \text{Hash}(< Y^{<1,1>} \cdots Y^{<M,N>} >)$, we are able to reduce the bandwidth requirement to 24GB for a practical tile size of 16×16.

There are two parameters that we had to adjust, the number of hash buckets and the size of these buckets. We generally use 37 hash buckets of 128 elements, although we have experimented with up to 109 buckets with more elements. Increasing the number of buckets or the size of the buckets reduces the number of collisions marginally.

Hilbert curve reordering. The Hilbert space-filling curve can be used to produce a linear mapping of a multidimensional space [6]. A desirable property of Hilbert curves is that the locality between objects in the multidimensional space is preserved in the linear ordering [17]. We sort the entire set of *k*NN searches along a three-dimensional Hilbert curve, $Y' = \text{Hilbert}(Y)$ using an efficient algorithm [14].

The Hilbert reordering algorithm, applied over all of the searches in Y, reduces the bandwidth from 196GB to 15MB, an improvement of four orders of magnitude. This is nearly optimal result, the average photon is transferred

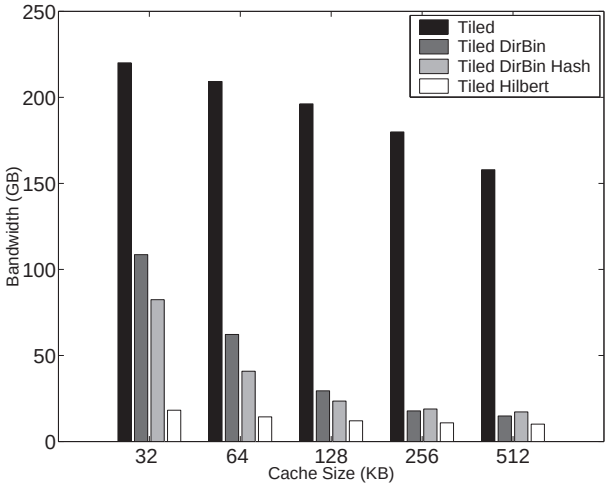


Figure 7: The practical reordering algorithms experience diminishing returns on caches larger than 128KB. The simple tiled algorithm will continue to benefit as the cache size is increased. There is little difference for the tiled Hilbert algorithm, as its working set is less then 32KB. (*kd*-tree, 128B cache lines and 16×16 pixel tiles.)

from memory less then twice, shows that reordering has the potential to dramatically reduce the cost of photon mapping. However, this does requires that we store the entire list Y before it is sorted which is not feasible because it would require 1GB of storage.

Tiled Hilbert reordering. Generating the Hilbert curve ordering requires a significant amount of processing and storage. To reduce this overhead, the reordering can be done on individual screen tiles generated by the screen tiled algorithm, $Y'^{<a,b>} = \text{Hilbert}(Y^{<a,b>})$. This reduces the computational and storage overhead, but also reduces the effectiveness of the reordering. The bandwidth increases as the number of pixels in the tile decreases, as shown by the bottom curve in Figure 6a. As the tile size increases, the Hilbert reordering can exploit more coherence in the search locations. For reasonable tile sizes of 8×8 to 32×32, the bandwidth ranges from 3GB to 33GB.

4 Data Structures

It is clear that the data structure storing the photon map has a direct impact on cache efficiency [4, 13, 15, 26]. The strong potential spatial locality will favor data structures that store nearby photons close to each other. We investigated four data structures: uniform grid, *kd*-tree, *kdB*-tree, and Block Hashing.

Contrary to our expectations, the choice of data structures did not have as significant an effect on bandwidth as reordering did. Figure 8a shows the bandwidth requirements for the naive ordering as the cache line size increases. The differences between the data structures shrink as more locality is exposed by the reordering. This is shown in Figure 8b for the Hilbert reordering. Table 1 shows the interaction between data structures and reordering algorithms for one cache configuration.

Uniform grid. A simple data structure that has been used effectively in hardware is the uniform grid [22]. The scene is subdivided into grid cells and the photons are distributed into cells. We investigated both fixed-sized cells, which allow for direct addressing, and indirectly-addressed variable-sized cells. Since the photons are located on surfaces, the grid cells that intersect the surfaces contain a large number of photons while the remaining grid cells are empty. Fixed-sized cells are inefficient as all of the cells must have enough storage to prevent photons from being discarded. While the overhead of the indirect method is higher, it can handle the highly variable density slightly better. Overall, however, we have found that this data structure is a poor choice for reducing bandwidth, and we do not report any further results.

kd-tree. Jensen selected the balanced *kd*-tree [2] for his implementation because it enables *k*NN searches at a cost of $O(k + \ln N)$. This analysis relies on the assumption of a constant memory access time, which is not valid in hierarchical memory systems. A memory request that can not be satisfied by the cache will require a memory fetch. This not only incurs a long latency but will also increase the total bandwidth to main memory. This data structure performs well compared to the other data structures at small cache lines, but as cache lines get larger the number of discarded photons increases.

Contemporaneously with our research, Wald analyzed the performance of the *kd*-tree and demonstrated an alternative to balancing that achieves a speedup of 1.2 to 3.4 for *k*NN queries [26]. This improvement is similar in magnitude to what we achieve with the *kdB*-tree de-

	kd-tree	*kdB*-tree	BH
Naive	23,045	11,755	73,523
Tiled Dir-Binned	3,927	4,397	20,316
Tiled Dir-Binned Hash	2,590	2,870	9,912
Tiled Hilbert	1,215	1,134	2,350
Hilbert	1.6	1.8	3.1

Table 1: The average number of memory block fetches for different combinations of data structures and reorderings. This number is directly related to bandwidth by the size of the cache line. Regardless of the data structure, memory efficiency is improved by choosing a better reordering. (128KB cache with 128B cache lines.)

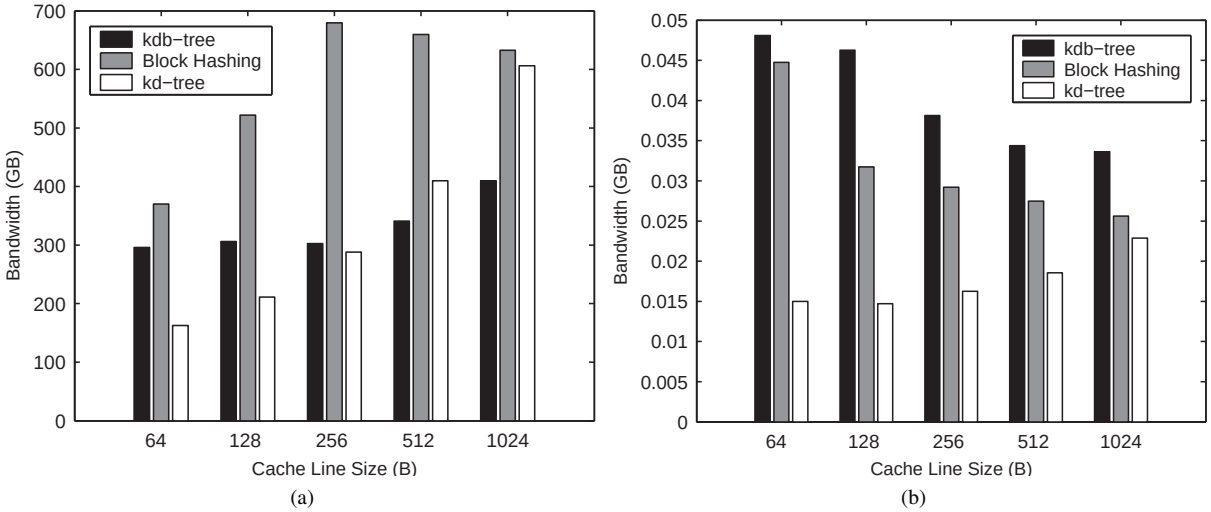


Figure 8: The difference between the data structures is significant for the naive order, (a), but not for the Hilbert reordering, (b). When an effective reordering algorithm is used, the data structure choice has little impact on bandwidth. Large cache lines increase the required bandwidth because unnecessary photons are brought in to the cache. Small cache line sizes penalize block oriented data structures that require the storage of some overhead for each block. (128KB cache with 16×16 tiles)

scribed next.

kdB-tree. The *kdB*-tree [24] blends the advantages of the *kd*-tree with the classical *B*-tree [1] by explicitly organizing the photons into cache-sized blocks. The bounding volume of the photon map is recursively divided into mutually exclusive rectangular regions. Each region contains a list of pointers to either another region or to a leaf node. The leaf blocks contain a list of photons that are contained within a bounding volume. The *kdB*-tree was developed in the database community, which has extensively studied the problem of *k*NN searches [8, 10]. However, as noted by Ma and McCool [15], many of these searches are designed for high-dimensional data.

In practice, the *kdB*-tree data structure can reduce bandwidth below that of the *kd*-tree only when the cache lines are large. Since each block of data imposes some overhead, as the block size shrinks the overhead begins to dominate. In our experiments with the naive ordering, Figure 8a, the crossover point was at 256B cache lines, with the *kd*-tree being a better choice for smaller cache. Although larger cache lines favor the *kdB*-tree, the absolute bandwidth is greater than with small cache lines because of unnecessary photons being read.

Block Hashing. Introduced by Ma and McCool [15], Block Hashing is an approximate-*k*NN data structure that stores spatially coherent photons in cache-line sized blocks. The goal is to reduce memory latency by minimizing the number of dependent memory requests compared to a hierarchical tree traversal. Photons are hashed into blocks based on a Locality-Sensitive Hash, and the blocks are designed to fit within a cache line for fast memory access. There are several parameters that must be adjusted for optimum performance, including hash-bucket size and the number of hash tables. We use the heuristics outlined in the paper to select the parameters.

Block Hashing was designed to reduce the effects of memory latency, not memory bandwidth. We found that it was competitive in terms of bandwidth with the other data structures. Block Hashing was designed with a 256B cache line, so in our cache line experiments we tried to set the accuracy parameter in the same spirit as Ma and McCool [15]. Since Block Hashing is an approximate *k*NN search, we set the accuracy $A = 20$ to better compare with the exact *k*NN searches. As this parameter controls the number of blocks that are examined for each search query, it directly impacts bandwidth. Ma and McCool suggest using $A = 10$ to $A = 16$, but these lower values caused a number of photon blocks to be orphaned at small cache line sizes.

5 Conclusions

We have shown that memory bandwidth can be reduced from 196GB to 15MB in our test scene, an improvement of four orders of magnitude, by reordering the *k*NN searches. This is close to the lower bound, as each block of memory that is touched is brought in less than twice.

It is however impractical as it requires 1GB of storage to enumerate all of the search locations.

We developed and tested techniques that are more amenable to hardware implementation and capture a large portion of the possible gains. These methods give the same proportional benefit whether irradiance caching is used or not. The 16×16 tiled direction-binning hashed reordering algorithm reduces the required bandwidth to 4GB (24GB without irradiance caching). This is an order of magnitude improvement and only twice the bandwidth required by the tiled Hilbert reordering. With current GPU memory bandwidth rates at 35GB/s [18], we believe that *k*NN search reordering will enable the development of interactive photon mapping hardware. The technique may also prove valuable to software systems.

Acknowledgements

Partial support provided by the National Science Foundation Graduate Research Fellowship Program, grants 0306478 and 0205425 and the NVIDIA Fellowship Program. We would also like to thank Yuanxin Liu and Jack Snoeyink for the Hilbert curve reordering code.

References

[1] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972.

[2] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[3] Per H. Christensen. Faster photon map global illumination. *J. Graph. Tools*, 4(3):1–10, 1999.

[4] Per H. Christensen and Dana Batali. An irradiance atlas for global illumination in complex production scenes. In *Proc. Eurographics Symposium on Rendering*, pages 133–141, 2004.

[5] Frank Dachille, IX and Arie Kaufman. GI-Cube: an architecture for volumetric global illumination and rendering. In *Proc. Graphics Hardware*, pages 119–128, 2000.

[6] C. Faloutsos and S. Roseman. Fractals for secondary key retrieval. In *Proc. Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of database systems*, pages 247–252, 1989.

[7] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. *Proc. VLDB*, pages 518–529, 1999.

[8] V. Havran. Analysis of cache sensitive representations for binary space partitioning trees. *Informatica*, 23(3):203–210, May 2000.

[9] Mark D. Hill and Alan J. Smith. Evaluating associativity in CPU caches. *IEEE Transactions on Computers*, C-38(12):1612–1630, 1989.

[10] P. Indyk, R. Motwani, P. Raghavan, and S. Vempala. Locality-preserving hashing in multidimensional spaces. In *Proc. of ACM STOC*, pages 618–625, 1997.

[11] Intel. Microburst Architecture. *IA-32 Intel Architecture Software Developers Manual*, 1:37–40, 2003.

[12] Wojciech Jarosz, April 2004. `http://renderedrealities.net/`.

[13] Henrik Wann Jensen. *Realistic Image Synthesis Using Photon Mapping*. AK Peters, 2001.

[14] Yuanxin Liu and Jack Snoeyink. A notation for hilbert curves to support multidimensional spatial indexing. In preparation.

[15] Vincent C. H. Ma and Michael D. McCool. Low latency photon mapping using block hashing. In *Proc. Graphics Hardware*, pages 89–99, 2002.

[16] Joel McCormack, Robert McNamara, Christopher Gianos, Larry Seiler, Norman P. Jouppi, and Ken Correll. Neon: a single-chip 3d workstation graphics accelerator. In *Proc. Graphics Hardware*, pages 123–132, 1998.

[17] Bongki Moon, H. V. Jagadish, Christos Faloutsos, and Joel H. Saltz. Analysis of the clustering properties of the hilbert space-filling curve. *Knowledge and Data Engineering*, 13(1):124–141, 1996.

[18] NVIDIA. Ultra-high-end products, October 2004. `http://nvidia.com/page/qfx_uhe.html`.

[19] Matt Pharr and Pat Hanrahan. Geometry caching for ray-tracing displacement maps. In *Proc. Eurographics Workshop on Rendering Techniques*, pages 31–ff., 1996.

[20] Matt Pharr and Greg Humphreys. *Physically Based Rendering from Theory to Implementation*. Morgan Kaufmann, 2004.

[21] Matt Pharr, Craig Kolb, Reid Gershbein, and Pat Hanrahan. Rendering complex scenes with memory-coherent ray tracing. In *Proc. SIGGRAPH*, pages 101–108, 1997.

[22] Timothy J. Purcell, Craig Donner, Mike Cammarano, Henrik Wann Jensen, and Pat Hanrahan. Photon mapping on programmable graphics hardware. In *Proc. Graphics Hardware*, pages 41–50, July 2003.

[23] E Reinhard and E W. Jansen. Rendering large scenes using parallel ray tracing. In *First Eurographics Workshop on Parallel Graphics and Visualization*, pages 67–80, 1996.

[24] John T. Robinson. The k-d-b-tree: a search structure for large multidimensional dynamic indexes. In *Proc. ACM SIGMOD international conference on Management of data*, pages 10–18, 1981.

[25] Jörg Schmittler, Ingo Wald, and Philipp Slusallek. Saarcor: a hardware architecture for ray tracing. In *Proc. Graphics Hardware*, pages 27–36, 2002.

[26] Ingo Wald, Johannes Guenther, and Philipp Slusallek. Balancing considered harmful – faster photon mapping using the voxel volume heuristic. In *Proc. Eurographics*, 2004.

[27] Ingo Wald, Philipp Slusallek, Carsten Benthin, and Markus Wagner. Interactive rendering with coherent ray tracing. *Computer Graphics Forum*, 20(3):153–164, 2001.

[28] Gregory Ward, Francis Rubinstein, and Robert Clear. A ray tracing solution for diffuse interreflection. In *Proc. SIGGRAPH*, pages 86–92, 1988.