SIMD Packet Techniques for Photon Mapping

Shawn Singh* Petros Faloutsos†

UCLA Computer Science Department

Figure 1: Sponza Atrium scene (model by Marko Dabrovic) at 900×300 and 100 samples per pixel, showing only indirect lighting, rendered with our framework in 91 seconds using 2 threads, averaging approximately 25 million shader operations per second.

ABSTRACT

We present a novel photon mapping framework that uses Single Instruction, Multiple Data (SIMD) parallelism to accelerate the final gathering phase of photon mapping. By using SIMD instructions, four coherent tasks can be computed in parallel using almost the same memory traffic as it would cost to process one task alone. This approach has been very successful for real-time ray tracing, but until now it has been unclear how to effectively apply the same approach to final gathering. Our solution is to use sample-point density estimation instead of *k*-nearest neighbor density estimation, a technique drawn from reverse photon mapping. Sample-point estimation removes the overheads that make SIMD instructions impractical, while retaining the same benefits and image quality as traditional photon mapping.

Additionally, an important question arises whether it is better to use forward or reverse photon mapping. In an interactive context, classical asymptotic algorithmic analysis is not enough to compare the two algorithms. We provide a novel asymptotic bandwidth analysis, which addresses more issues found in practice. The analysis motivates the use of forward photon mapping when using SIMD parallelism as well as partial reordering for improved scalability. The resulting framework can achieve interactive rates for photon mapping at low resolutions, including the time it takes to trace photons and build the photon map.

Index Terms: I.3.6 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing; I.3.1 [Computer Graphics]: Hardware Architecture—Parallel processing

1 INTRODUCTION

Photon mapping is generally accepted as an efficient, versatile, and high-quality global illumination algorithm. While photon mapping has many advantages in its algorithmic simplicity and good scalability, it is typically regarded as an offline rendering algorithm, and no existing implementation is truly real-time. This limits the scope of where photon mapping could be used, even though there are numerous applications where interactive photorealism would be ideal. In this paper we consider the goal of interactive photon mapping using SIMD data parallelism, towards the long-term goal of achieving real-time photorealism.

There is a striking similarity between the computations used in real-time ray tracing and those used in photon mapping. In ray tracing, a ray traverses through a KD-tree (or similar data structure) of geometric primitives, to find ray-intersection tests to compute visibility. In photon mapping, a point sample traverses through a KD-tree (or similar data structure), finding $\langle$*photon, camera-point*$\rangle$ pairs to compute global illumination. In other words, both algorithms involve a core operation – ray intersection tests and $\langle$*photon, camera-point*$\rangle$ *shader operations* – and both algorithms use a spatial database, often a KD-tree, to efficiently cull the number of core operations that must be performed.

Because of this strong similarity between ray tracing and photon mapping, the techniques used in real-time ray tracing could also apply to photon mapping. Specifically we are referring to the use of SIMD packets, which has been a key factor in achieving real-time ray tracing [21]. The observation of applying packet techniques to photon mapping has been suggested before [22], but to our knowledge no real implementation has demonstrated the use of packets in this way.

This problem can be attributed to the type of search being performed by photon mapping. Traditional photon mapping performs a *k*-nearest neighbor search using the KD-tree, which has additional overheads. For necessary algorithmic efficiency, the lists of *k* items are organized as heaps or priority queues [6], which incurs more processing overhead. More importantly, each heap or priority queue has different branching logic while being updated, and this branching cannot be handled in parallel with SIMD instructions. Because of this, it has remained unclear how to effectively use SIMD parallelism for photon mapping.

Contributions. We propose a way to solve this problem by replacing the *k*-nearest neighbor query with a different density estimation technique known as *sample-point estimation* [19]. This results in a simpler search that does not need to keep track of *k* neighbors at once, while retaining approximately the same quality

*e-mail: shawnsin@cs.ucla.edu

†e-mail: pfal@cs.ucla.edu

of estimation. Sample-point estimation has been used previously in reverse photon mapping [4] to gain algorithmic benefits over photon mapping, but it has not been used previously in the context of traditional (forward) photon mapping. By applying this new estimation technique to forward photon mapping, each ⟨*photon, camera-point*⟩ pair can be computed independently. This is what makes it possible to effectively use SIMD instructions for photon mapping.

In addition to the main contribution described above, we also ask whether it is better to use forward or reverse photon mapping for interactive performance. As explained below, an algorithmic analysis does not give enough information to answer this question. We provide a novel asymptotic bandwidth analysis that gives a more complete comparison, concluding that forward and reverse photon mapping have the same scalability except for higher potential coherence in forward photon mapping. Also, an important aspect of fast photon mapping is how to efficiently order the queries to a photon map for better coherence. We describe a method of implementing partial Hilbert reordering, which introduces a way to tradeoff the cost of reordering with the cost of queries to the photon map. Finally, we demonstrate these contributions with a software framework that uses real-time ray tracing, a fast KD-tree build, our SIMD accelerated photon-map queries, and partial reordering.

The rest of this paper is organized as follows. Section 2 discusses related work in performance for photon mapping. Section 3 describes the sample-point estimator. Section 4 gives a novel asymptotic bandwidth analysis which motivates the decisions made for our implementation. Sections 5 describes the SIMD framework, Section 6 describes our approach to ray coherence and photon coherence including Hilbert reordering, and Section 7 discusses performance results. Finally, Section 8 concludes.

2 Related Work

Our implementation of real-time ray tracing is based heavily on the work by Wald [20]. The reader is referred to Wald's work and recent surveys [20, 24, 25] for more information on highly optimized ray-triangle intersections, KD-tree traversal, SIMD ray tracing techniques, and global illumination using ray tracing.

The first works on photon mapping are by Jensen [5]. The key advantage of photon mapping is that the representation of illumination in the scene, photons traced from light sources, is organized separately from geometry. This allows lighting complexity to scale independently of geometry. Photon mapping is widely agreed to be a scalable and high-quality global illumination algorithm.

Since then, some research has suggested that photon mapping would be interesting for real-time. Larsen and Christensen [8] simulate the photon map using a hemi-cube render-to-texture approximation on the GPU. Purcell et al. [12] suggest and demonstrate that a GPU implementation of photon mapping could eventually benefit from the power of the streaming paradigm. Both these works used a previous generation of GPU technology, and it would be interesting to revisit GPU implementation of photon mapping in future work now that GPUs have become more general-purpose. Günther et al. [3] integrated the photon map with a real-time ray tracer, but only used the photon map for caustics.

An important aspect of improving performance of photon mapping is reordering computations and generally reducing memory traffic. Steinhurst et al. [16] show that re-ordering the nearest-neighbor queries can drastically reduce the amount of required memory traffic to render a single frame. Specifically, they used *Hilbert reordering*, which sorts query points along a space-filling Hilbert curve for a very high coherence. They show that it offers up to four orders of magnitude reduction of the required memory traffic, and that the resulting order of operations was in some sense optimal, because photons were loaded to memory on average only 2-3 times. Several other works [4, 11, 20] also address the topic of exploiting coherent operations, usually by reordering computations in some way that results in significantly improved cache behavior.

Another equally important aspect of photon mapping performance is how to improve the cost of each photon map query. Since this operation occurs millions of times, it is critical to increase its performance as much as possible. For example, a 512×512 resolution image with 100 secondary rays per pixel would require more than 25 million queries, each of which would have roughly 15-100 shader operations. Massive parallelism (e.g., [7]) is an essential way to improving the throughput of queries. Wald et al. [22] show how the photon map can be organized to optimize the cost of performing each k nearest neighbor query. Ma and McCool [9] propose an approximate nearest neighbor technique that has lower overhead than the traditional k-nearest neighbor search. Havran et al. [4] describe *reverse photon mapping*, which uses sample-point density estimation to achieve algorithmic benefits compared to traditional photon mapping. In reverse photon mapping, instead of each camera point searching through a photon map, each photon searches through a database of camera points.

To our knowledge, our work is the first implementation to demonstrate the possibility of SIMD packets for photon mapping queries, and no prior work has explicitly described the asymptotic bandwidth scalability of photon mapping. With this bandwidth analysis we reach a different conclusion than the analysis found in Havran et al. [4], because their work focuses on fast production-quality rendering instead of interactive performance. In an interactive setting, preprocessing such as reordering and building a photon map must also be included in the analysis. Unlike Günther et al. [3], we use photon mapping for general global illumination instead of caustics. Finally, unlike some other works that approximate the algorithm or sacrifice scalability for better performance, our work retains the mathematical soundness (Section 3) and scalability (Section 4) of traditional photon mapping.

3 Density Estimators for Photon Mapping

Each query to the photon map computes the radiance along a given ray by using the location and color of nearby photons. This is done by estimating the density of photons, and this estimate can be computed using techniques drawn from density estimation literature in statistics. Here we briefly recall the estimation technique used in traditional photon mapping, as well as the technique used in reverse photon mapping and our work.

Traditional photon mapping [6] uses a k-nearest neighbor estimator [13, 19]. Photons are organized into a KD-tree to make them efficient to search. Using these photons and the radius r that encloses these photons, the radiance along each camera ray can be computed. This k-nearest neighbor estimator can be expressed as:

$$\hat{L}(y,\omega_o) = \frac{1}{\pi r^2}\sum_{i=1}^{k} f(y,\omega_i,\omega_o)\Delta\Phi_i K\left(\frac{|\,y-x_i\,|}{r}\right), \qquad (1)$$

where, L is the radiance we want to estimate, K is a kernel function centered around each photon. The kernel is scaled by the photon power $\Delta\Phi_i$ and the BRDF reflectance f, and r is the radius of a sphere that encloses all k photons. As mentioned above, in order to efficiently perform a k-nearest neighbor search, a heap or priority queue data structure is needed to keep track of the closest photons. Also, the radiance estimate must wait for all k photons to be collected before r can be computed.

We use a *sample-point estimator* [19], which can be written as:

$$\hat{L}_r(y,\omega_o) = \sum_{i=1}^{n} \frac{1}{h(x_i)^2} f(y,\omega_i,\omega_o)\Delta\Phi_i K\left(\frac{|\,y-x_i\,|}{h(x_i)}\right), \qquad (2)$$

where $h(x_i)$ is the kernel width, derived from an initial coarse estimate of photon density. As before, a kernel function centered around each photon is applied to the estimate each camera point.

However, instead of searching for a fixed number of photons, this time all n photons whose kernel width $h(x_i)$ overlaps the query point are used for shading.

Each photon can have a potentially different kernel width, but for this work we used the same kernel width for all photons. Previous works [4, 14] demonstrate variable kernel widths, and our initial experiments suggest that it can be done with relatively little overhead compared to fixed kernel widths. For the purposes of this paper, fixed-width kernels result in the same subjective image quality. Variable kernel widths are more accurate in dark regions of a scene where photons are sparser, and it will be appropriate to address in future work.

In the sample-point estimator in Equation 2, there are no parameters outside the summation. This means each portion of the sum can be computed independently and directly added to the final image. A photon can be used for shading immediately when it is found, without having to wait for k other photons. Even if an implementation still maintains a queue of photons to shade, the order no longer matters because it is not a k-nearest neighbor search, and thus a heap or priority queue is not needed. This is the key change that allows SIMD parallelism to be used effectively.

4 Asymptotic Bandwidth Behavior

Because we are using sample-point estimation, originally used in reverse photon mapping, an interesting question arises whether it is better to use forward or reverse photon mapping for interactive performance. In this section we address this question based on the trend of multi-core parallelism where bandwidth will become a limited resource for each core. In the context of SIMD packets, we conclude that forward photon mapping is more appropriate over reverse photon mapping. We also use this analysis to motivate the use of partial reordering, which reduces the implicit constant cost of reordering.

4.1 Asymptotic algorithm complexity

Let us first briefly recall the algorithmic analysis of forward and reverse photon mapping. In forward photon mapping, p photons are organized into a KD-tree, and for all m camera points, a k-nearest-neighbor query is performed. In reverse photon mapping, m camera points are organized into a KD-tree, and for all p photons, a fixed-radius search is performed, resulting in an average of k photons contributing to each camera point. Thus, the algorithmic complexity is

$$\mathscr{A}_{forward} = O(p\log p + m\log p + km), \quad (3)$$

$$\mathscr{A}_{reverse} = O(m\log m + p\log m + km). \quad (4)$$

Here, the first term is the cost of building the tree for search, the second term is the cost of performing all searches, and the third term is the cost of actually shading k photons for each camera point. Usually the number of camera points is far greater than the number of photons, that is, $m \gg p$. Therefore, the $O(m\log m)$ and $O(km)$ terms dominate reverse photon mapping, while the $O(m\log p)$ and $O(km)$, dominate in forward photon mapping. The value of k typically varies from 35 to 150, while $\log m$ can vary from 25-50 – the important point being that in practice $O(m\log m)$ and $O(km)$ have roughly the same scalability. This shows that algorithmic analysis does not give us enough information to compare forward and reverse photon mapping.

4.2 Asymptotic bandwidth complexity

An asymptotic *bandwidth* analysis is equally important, if not more important, to compare forward and reverse photon mapping. With the growing trend of multi-core parallelism, computational throughput is likely to continue increasing exponentially, while bandwidth will scale much more slowly. This means that any algorithm that is embarrassingly parallel, such as photon mapping, will be able to scale until it becomes bandwidth limited. To understand the true scalability of photon mapping, we must understand asymptotically how much memory traffic the photon mapping algorithm requires.

We define bandwidth complexity to be an expression that is proportional to the number of $O(1)$ memory operations needed to compute an algorithm, where each memory operation transfers a constant number of bits between processor cache and main memory. By this definition, bandwidth complexity cannot be larger than algorithmic complexity, because any larger cost would have to be part of the algorithmic complexity in the first place. The bandwidth complexity can be less than the algorithmic complexity, typically influenced by good cache behavior.

It turns out that the bandwidth required for the search portion of photon mapping is very low, because of the compact, efficient representations for a KD-tree node. Most of these nodes may already be cached, and each node is only 8 bytes. In [14], the bandwidth required for the shading portion is consistently less than 5% of the total raw bandwidth cost. Even asymptotically, looking at Equations 3 and 4, the complexity of the search, $O(m\log p)$ and $O(p\log m)$, will not grow faster than the shading complexity, $O(km)$. For these reasons it is appropriate to eliminate search complexity from the bandwidth analysis entirely.

On the other hand, the bandwidth required to construct a KD-tree is very high. To build one node of the KD-tree, all the points contained inside that node must be traversed in order to bin it to the right or left child – this applies to any type of KD-tree. Thus the bandwidth complexity is the same as the algorithmic complexity – $O(p\log p)$ for forward photon mapping and $O(m\log m)$ for reverse photon mapping. Near the bottom layers of the tree, all the points contained in a given node can fit into cache. Simulations from previous work [14] show that caching a depth-first KD-tree build can reduce the bandwidth by about half. Asymptotically, however, the required bandwidth will outgrow the benefits of cache, and so the cost remains $O(n\log n)$ for n points.

The bandwidth cost of shader computations can be characterized in two different ways, depending on the ordering of operations. If consecutive operations are mostly incoherent, we can assume that the bandwidth cost will be proportional to the algorithmic cost, $O(km)$. In this case, the total bandwidth complexity is:

$$\mathscr{B}_{forward} = O(p\log p + km), \quad (5)$$

$$\mathscr{B}_{reverse} = O(m\log m + km), \quad (6)$$

where the logarithmic term is the cost of building the KD-tree, and km is the cost of shading computations. Because p is usually several orders of magnitude smaller than m, forward photon mapping requires less bandwidth than reverse photon mapping.

If consecutive operations are coherent, it is unclear exactly how the km term is reduced. We can estimate the required bandwidth as $O(km/q)$, where q represents the coherency between consecutive operations. The higher the coherence, the lower the bandwidth cost will be.

One major approach to increasing coherence is to explicitly reorder computations [4, 11, 16]. To our knowledge, the best reordering techniques require $O(n\log n)$ algorithmic and bandwidth cost to reorder n operations. For example, Hilbert reordering, shown to be nearly optimal [16], can be implemented as building an octree in $O(n\log n)$ [1]. It is still an open problem to determine if similar optimal coherence can be achieved with less costly preprocessing, but intuitively it seems that $O(n\log n)$ is the best that can be done, since "optimal ordering" implies that all n operations have been properly sorted or organized in a tree structure. Finally, note that reordering improves only the bandwidth cost of shading, but it adds an additional term to both algorithmic and bandwidth analysis. Accounting for the benefit and additional cost of optimal reordering,

we can approximate the bandwidth complexity to be:

$$\mathscr{B}'_{forward} = O(m\log m + p\log p + km/q), \quad (7)$$

$$\mathscr{B}'_{reverse} = O(p\log p + m\log m + km/q). \quad (8)$$

Interestingly, in this case forward and reverse photon mapping have the same bandwidth cost. However, now both methods have the extremely costly $O(m\log m)$ term.

4.3 Our approach

One detail that distinguishes Equations 7 and 8 is the use of SIMD parallelism. SIMD parallelism is only beneficial when the four simultaneous queries are very coherent. If they are not coherent, then the average number of active queries per SIMD operation decreases, resulting in very little gain compared to a non-SIMD implementation. Photons are generally sparser and more evenly distributed than camera points, and thus, even with reordering, a SIMD packet of photons would be less efficient than a packet of camera points.

Also, the bandwidth cost of reordering can be reduced via partial reordering. With effective reordering, photon mapping can become compute-limited instead of bandwidth limited. Therefore, there comes a point where reordering computations further no longer improves performance significantly. For this reason, we employ a partial Hilbert reordering scheme that reorders queries to the photon map just enough that the best performance is achieved, but without wasting computation on further reordering. Since $m \gg p$, partial reordering benefits forward photon mapping more than reverse photon mapping. Our specific implementation of reordering is described in Section 6.2.

In summary, our bandwidth analysis reveals several points that a traditional algorithmic analysis does not show. First, the implicit constant cost associated with search is very small, and thus the cost of searching for shader operations is negligible. Second, we see that reordering and building a KD-tree are the bottlenecks towards better scalability, because of the high bandwidth requirement. Third, our bandwidth analysis shows that forward and reverse photon mapping have effectively the same bandwidth scalability, and so our approach is to favor forward photon mapping because it can benefit greater from SIMD and partial reordering techniques.

5 SIMD Photon Mapping Framework

In Section 3 we described the sample-point estimator that allows SIMD parallelism, and then in the previous section we justified the use of forward photon mapping. In this section we describe our SIMD framework.

5.1 Photon and ray tracer

The foundation of our framework is a real-time ray tracer based on work by Wald [20], with minor differences in data layout. We use up to four threads, which allows efficient execution on multi-core processors. On a 2.66 GHz Core 2 Duo, this foundation can trace up to 20 million rays per second on trivial scenes, 10 million rays per second on moderately complex scenes including texture mapping, BRDF evaluation, and direct lighting, and 2-8 million rays per second with secondary rays that gather indirect lighting. While it is possible to improve performance even further on state of the art processors, such work is beyond the scope of this paper, and this performance gives ample time for photon map queries to be computed.

Our specific implementation of final gathering is described as follows. Primary rays are traced from the camera into the scene, and many secondary rays are spawned from the intersection point of each primary ray. Throughout the paper, the metric "samples per pixel" is equal to the number of secondary rays per primary ray, because we use only one primary ray per pixel. The intersection point found for each secondary ray becomes a photon map query, where we estimate radiance along the secondary ray.

We use only single rays to trace photons in order to avoid altering the distribution of photons. The number of photons is small enough that tracing single rays is still very fast. Also, in practice, we found that the $O(p\log p)$ cost of building a KD-tree photon map is fast enough for interactive performance when p, the number of photons, is approximately 500,000 or less. This is enough photons to capture indirect lighting effectively for most scenes. Note that this is why reverse photon mapping, though very efficient for offline rendering, cannot be used interactively: every frame would require building a KD-tree of millions of camera-points. Finally, our photon tracing and KD-tree build do not exploit multiple threads, but parallelizing these two phases of computation is straightforward and should give the expected speedups.

5.2 Data layout

Our data structures and layout are as follows. Photons are stored in two separate lists, one for the "hot" data that is accessed frequently, and one for the "cold" data that is only accessed when the photon contributes its information to a camera point:

```
// 16 bytes
struct PointSample {
  Point p; // x, y, z, location
  void * coldData;
};

// 24 bytes
struct PhotonData {
  Vector incomingDirection;
  Color power; // red, green, blue
};
```

The photon map is a KD-tree, stored as an array of `KDNode` structures. The layout is similar to the structure described by Pharr and Humphreys [10].

```
// 8 bytes
struct KDNode {
  union {
    float splitPlane;
    unsigned int axis; // 2 bits
  };
  int childrenIndex;
};
KDNode photonKDTree[NUM_PHOTONS];
```

In this tree, one photon is placed with each KD-tree node, and the split plane of the node is chosen specifically so that the photon lies on the split plane. Because of this, we must use an index instead of a pointer for the children, and in turn we must place the 2-bit axis either in the upper bits of the index or in the lower bits of the floating-point mantissa of `splitPlane`. Putting this flag in the upper bits of a number requires using conditional branching or costly shift operations, therefore for this implementation we placed the bits in the floating-point mantissa. The loss of accuracy because of this is negligible in our experience.

Originally we tried to adapt the sliding-midpoint KD-tree that is effective for reverse photon mapping [4]. However, this data structure stores up to 30 points per leaf node, and no points in inner nodes. This approach works well for reverse photon mapping, where a photon may contribute to several hundred camera points, and thirty camera points in one leaf are all likely to be used. For forward photon mapping, however, where each camera point typically uses fewer than 100 photons, this data structure resulted in too many wasteful shader operations where the photon was too far from the camera point, resulting in no contribution. We also tried reducing the number of points per leaf node for the sliding-midpoint KD-tree, but this resulted in a prohibitively large tree. By using the tree

where photons lay on the split planes, we were able to avoid most of the unnecessary shader operations and place known bounds on the size of the tree – at most the KD-tree will have the same number of nodes as there are photons.

To implement SIMD photon gathering, we use Intel SSE instructions. The first piece of data is four points that were found from tracing a packet of rays, and the second piece of data is the incident directions.

```
struct SSEPoint {
  __m128 x;
  __m128 y;
  __m127 z;
} queryPoint;

struct SSEVector {
  __m128 dx;
  __m128 dy;
  __m128 dz;
} incomingDirection;
```

Here, `__m128` holds four floating-point values that are processed in parallel with SIMD instructions. These two data structures already occur in the packet ray tracer as the origin and direction of a ray. For photon mapping, `queryPoint` represents four camera points where photon illumination needs to be computed, and `incomingDirection` represents the incident direction on each point, used for shading.

5.3 SIMD traversal

After a packet of secondary rays has been traced, the resulting points of intersection are query points where the illumination information of photons is computed using the sample-point estimator. This can be done by using a fixed-radius search through the photon map. Two pieces of information are passed on from the packet of rays to the SIMD photon gathering code. The first piece of data is a bit mask indicating which rays of a packet did intersect anything, and which rays of the packet should be omitted from the photon query. The second piece of data is the intersection points of the packet of rays.

Our SIMD traversal is similar to the SIMD ray tracing traversal shown by Wald [20], but the details are different because we are traversing points instead of rays. The pseudocode is as follows and explained below:

```
float searchRadius
SSEPoint queryPoint; // four values of x, y, z
KDNode currentNode  // initialized to the root node
stack; // search stack

while ( stack.notEmpty())
{
  if(IS_LEAF(currentNode)) {
    ShadeCurrentNode(...);
  }
  else {
    // determine if axis is x, y, or z
    axis = DIMENSION(currentNode.axis);
    dist = queryPoint[axis] - currentNode.splitPlane;
    maskLeft = mask & (dist < -searchRadius);
    maskRight = mask & (dist > searchRadius);
    if (maskLeft)
      stack.push(leftChildNode,mask);
    if (maskRight)
      stack.push(rightChildNode,mask);
    if (maskLeft & maskRight)
      ShadeCurrentNode(...);
  }
  (currentNode,mask) = stack.pop();
}
```

In this pseudocode, `maskLeft` and `maskRight` are 4-bit masks indicating which of four query points should remain active when searching through the left or right children. A query point remains active if its search radius overlaps the child node. If any one of the four simultaneous queries is active in *both* children, that is, if `(maskLeft & maskRight)` is non-zero, then that means the query radius overlaps the splitting plane. In this condition, the photon associated with the current node is used for shading, because the photon lies on the split plane of the current node. Finally, `ShadeCurrentNode` uses the photon associated with the current KD-tree node to compute a portion of the summation described in Equation 2. This function performs a more accurate distance check to make sure the photon actually contributes to the appropriate query points.

6 Coherent ordering

6.1 Ray coherence

It is already given that packets of primary rays, traced from the camera view, will be highly coherent. However, secondary rays are less spatially coherent, and so it is a challenge to handle how memory is accessed throughout the photon mapping algorithm. Given a point where a primary ray intersected, this point becomes the origin for packets of secondary rays. To generate directional vectors for these secondary rays, we use a stratified sampler. A stratified sampler divides the sample space into a regular grid of cells, and chooses a fixed number of samples for each cell. This is a method of uniform sampling, so for importance sampling, this grid of cells is then mapped to the appropriate desired distribution. Stratified sampling has the nice property that as the number of stratified cells increases, the size of each cell decreases, and therefore coherency of samples within a single cell and in nearby cells also improves. We place four random samples in each stratified cell, so that these four samples will create rays with similar directions and the same origin.

We also interleave computations per thread by having threads process every other pixel (or every fourth pixel, in the case of four threads). Simply giving each thread a different tile of the image was clearly incoherent, because four threads began to perform worse than two threads on a dual core processor. By interleaving the computations, even though we do not enforce any synchronization, four threads were able to scale performance slightly better. Our approach to ray coherence works well enough to show performance increase using SIMD packets, but in future work a better ordering technique is necessary for higher performance.

6.2 Photon coherence

For each secondary ray, instead of immediately performing the photon query, we store the points where secondary rays intersect so that all the queries can be reordered and performed after tracing all rays. This approach reduces cache thrashing between the scene geometry data during ray tracing and the photon data during shading. During reordering and shading, the query points can be streamed in and out of cache predictably and therefore with less latency overhead.

To reorder the query points, we use Hilbert reordering, which has been shown to be highly effective for reordering photon map queries [16]. Figure 2 depicts the analogy between a 2-D Hilbert curve and a quadtree. The Hilbert curve traverses all nodes of the quadtree in a specific order. As the Hilbert curve and quadtree are recursively expanded, the coarse ordering determined by previous recursive steps does not change. Infinitely recursing, the Hilbert curve fills the entire space, while the nodes of the quadtree become infinitesimally small. In this way the space filling curve is exactly the same as a quadtree with specific ordering of its nodes. The same analogy holds between a 3-D Hilbert curve and an octree.

Based on this property, query points can be sorted along a Hilbert curve simply by organizing the points into an octree, while enforc-

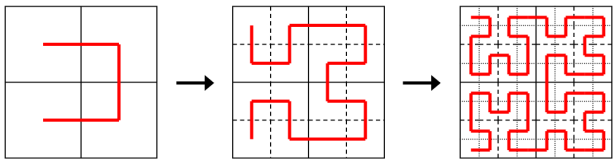

Figure 2: Visualization that shows how a 2-D Hilbert curve is analogous to a quadtree. The same analogy holds between a 3-D Hilbert curve and an octree.

ing a specific ordering of nodes. The ordering of nodes is determined by the path that the Hilbert curve takes, which is given procedurally by a *Lindenmayer system* – a system of rules that define how the curve recursively subdivides. In our case, we store the L-system as a straightforward, compact look-up table that describes the ordering of child nodes given the ordering and orientation of the parent node. This allows us to sort points along a Hilbert curve for essentially the same cost as building an octree, $O(n \log n)$.

Partial reordering. In Section 4.3 we motivated the use of partial reordering. Because a Hilbert curve is defined recursively, we can implement partial reordering by simply stopping recursion before points are fully sorted. Partial reordering allows us to achieve the "sweet spot" where photon mapping becomes compute-limited, where further improving the cache behavior with reordering no longer improves performance. Another "sweet spot" to explore is to optimize the sum of time spent reordering and time spent performing queries. In the case of using SIMD packets, the second sweet spot is more important and requires a finer granularity of reordering to make coherent packets. In this work, we stop recursion when there are 16 or fewer points being recursively sorted. This value was found by manual trial and error, and determining this value automatically is left as future work.

7 Results

General performance can be seen in Table 2, Table 3, and Figure 4. In the rest of this section, we discuss specific aspects of performance in more detail.

Effectiveness of SIMD. To measure the effectiveness of SIMD parallelism, we looked at the average number of active shader operations taking place during the function `ShadeCurrentNode`. Since there are at most four parallel operations in SSE instructions and at least one active query if the function is used at all, this metric can range from 1.0 in worst case to 4.0 in best case. Table 1 shows that SIMD can be effectively used for photon map queries, but only when combined with reordering which improves the coherence of photon queries. We also found that more total samples (reordered) and a larger search radius for each query point improve this metric.

Comparison between SIMD and non-SIMD is shown in Table 3. Performance improves significantly, but not ideally. This is because our current implementation only applies SIMD to the search and does not parallelize the shader operations, since each query point in a SIMD packet may potentially have a different BRDF. In future work we expect it will be straightforward to address this by using SIMD parallelism for shader operations that use the same BRDF.

Multi-core scaling. Scalability for our renderer on the Sponza Atrium and Cornell Box scenes can be seen in Table 2. These results were acquired on a 2.66 GHz Intel Core 2 Duo with 4 MB shared cache, except for the last entry running four threads, which was captured on a 2.66 GHz Intel quad-core processor with 8 MB cache. In both scenes, performance scaled linearly with the number of cores. Note that the complexity of photon mapping is mostly independent of scene geometry. Even though the Cornell Box scene has simple geometry, our Cornell Box test uses a bigger average k value, and therefore the queries to the photon map take more time despite the simplicity of the scene. Also note that the results in Table 2 do not use reordering. With reordering, the coherency of query points is independent of the coherence of secondary rays that generated these queries; such results can be seen in Table 3.

	Without reordering	With reordering
Cornell Box		
Direct visualization	3.18	N/A
16 samples per pixel	1.07	2.84
64 samples per pixel	1.16	3.32
100 samples per pixel	1.20	3.41
Sponza		
Direct visualization	3.74	N/A
16 samples per pixel	1.41	3.83
64 samples per pixel	1.72	3.92
100 samples per pixel	1.84	3.93

Table 1: Average number of individual queries active during a SIMD shading operation. This metric ranges from 1.0 (worst case) to 4.0 (best case). These numbers were acquired at 256×256.

	Sponza	Cornell Box
Resolution	308 × 308	308 × 308
Number of triangles	66,454	30
Samples per pixel	576	576
Average k	45	60
SIMD ray tracing only	104 s	65 s
1 core	235 s	276 s
2 cores	118 s	146 s
4 cores	61 s	80 s
Shader-operations per second per core	10 million	11 million

Table 2: Performance for the configuration of Sponza and Cornell Box scenes shown in Figure 3, showing how performance scales with SIMD and multiple cores. k is the average number of shader operations per camera point, and "SIMD ray tracing only" gives baseline performance of the ray tracer, ray tracing all samples, but with no gathering phase.

Interactive performance. Images of an interactive session can be seen in Figure 4, averaging about 10-11 frames per second when visualizing photons directly and 1.14 frames per second when visualizing the photon map indirectly with 16 samples per pixel. These images were captured using 4 threads on the 2.66 GHz Intel quad-core processor. It is possible to interactively move the camera and light sources. Tracing photons, building the photon map KD-tree, tracing primary and secondary rays, reordering, and computing final gathering are all recomputed dynamically in every frame.

Benefits of reordering. Table 1 and Table 3 both show performance with and without reordering. Performance improves significantly with reordering, and it is clearly necessary for effective SIMD parallelism. We observed two distinct benefits of reordering. The first is improved cache behavior, which can be achieved by a coarse partial reordering. The second is improved SIMD performance which benefits more from finer granularity reordering. This distinction may be interesting to exploit for faster, equally effective reordering techniques in future work. Both Table 1 and Table 3 reflect performance using partial reordering, described above, that stops recursion when there are 16 or fewer points in a recursive step.

Tracing photons and building the photon map. The time taken to trace photons using single rays was reasonable, able to trace up to 500,000 photons in less than half a second. It would be straight-

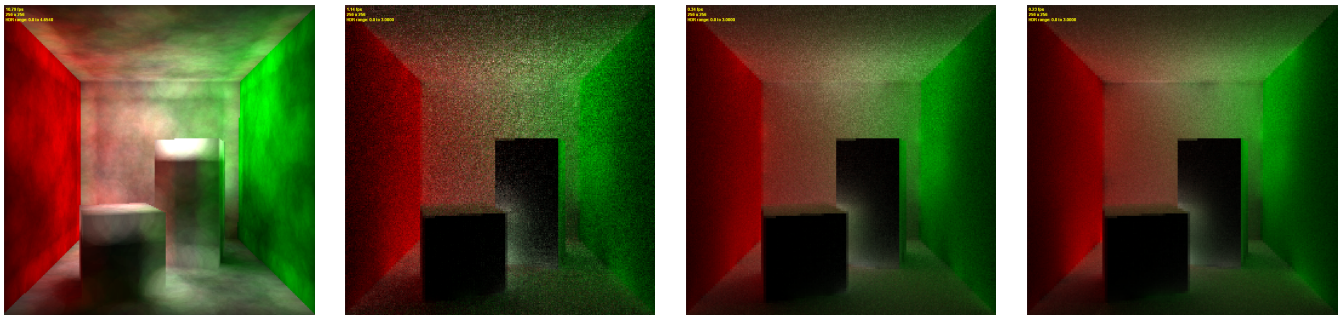

Figure 4: Images that represent the quality of rendering with interactive performance. Photons are traced and organized into a KD-tree dynamically every frame. The 2-D user interface is 900×900, but the rendering software used a resolution of 256×256. The leftmost image shows photons directly visualized, averaging about 10 frames per second (fps). The samples per pixel (spp) and frame rate for the other three images, from left to right, are 16 spp (1.14 fps), 64 spp (0.34 fps), and 100 spp (0.23 fps).

Scene	# of rays (millions)	No H.R. SIMD	With H.R. single	With H.R. SIMD	Time to reorder
Cornell	6.5 M	9.5 s	3.8 s	3.1 s	0.20 s
Cornell	10 M	14.3 s	5.8 s	4.6 s	0.33 s
Sponza	6.5 M	19.9 s	12.32	9.3 s	0.36 s
Sponza	10 M	28.8 s	18.34	13.6 s	0.56 s

Table 3: Comparison showing the cost of partial Hilbert Reordering (H.R.) and how it significantly improves performance. These numbers correspond to the Cornell Box and Sponza scenes rendered at 320×320 with 64 and 100 samples per pixel, using 2 million photons.

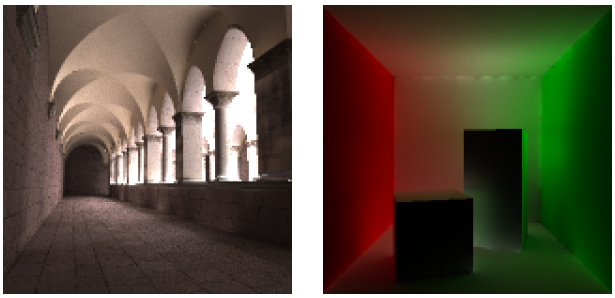

Figure 3: Sponza Atrium scene (model by Marko Dabrovic) and Cornell Box scene using our SIMD photon mapping framework, showing only indirect lighting from 500,000 photons, rendered at 308×308 and 576 samples per pixel. These images correspond to the results in Table 2.

forward to trace photons in parallel with multiple threads, but it is not clear how to trace coherent packets of photons without biasing the distribution of lighting information.

Building the photon map KD-tree can be done very quickly as long as the number of photons is less than about 500,000. In our experience this is true for both the sliding-midpoint KD-tree used in reverse photon mapping as well as the KD-tree we used in our implementation. For 500,000 photons, the KD-tree typically takes about $1/3$ of a second to build. Beyond this, the KD-tree build takes too long for interactive performance. Note that if we had used reverse photon mapping, the KD-tree would have been built over the set of query points, and we would have been forced to rebuild a KD-tree of millions of points every frame, which would be prohibitive for scalable performance.

7.1 Discussion

One useful metric to measure the performance of photon queries is the number of shader operations per second, where each shader operation is the process of computing a $\langle$*photon, camera-point*$\rangle$ pair's contribution to a pixel. The reason we propose this metric instead of the number of queries per second is that the average number of photons per query, k, may vary depending on several factors, and so a direct comparison of queries/sec across different works may not be as informative. The computation required for one shader operation is more consistent and predictable. We typically achieved a rate of 8-12 million shader operations per second per core, as shown in Table 2 and Figure 1 on the first page. While this is enough for interactive performance, we estimate that hundreds of millions of shader operations per second will be necessary for real-time high quality global illumination.

We focused on diffuse surfaces because this represents the worst-case incoherent distribution of secondary rays, especially when using fewer secondary rays per primary ray. Consequently, the distribution of queries to the photon map is also roughly worst-case. A glossy material, on the other hand, will have many more rays bouncing in a similar direction, resulting in better coherence after reordering. Our framework is capable of handling glossy materials by using the appropriate BRDF function – $f()$ in Equation 2.

Concurring with the bandwidth analysis we gave in Equation 7, the performance of this framework is roughly proportional to the number of shader operations, and the bandwidth cost of tree-build and search are usually only 10-30% of the entire time spent rendering. Table 2 shows that nearly half the time for gathering is spent tracing rays, this is because our secondary rays are not ideally coherent. As mentioned before, this ray incoherence is independent of photon coherence when using reordering.

7.2 Future Work

Our implementation has significant room for improvement. The code that performs sampling and material properties is based on the implementation in the pbrt software [10], which has a virtual interface to invoke a BRDF evaluation. This occurs *four times* in the most critical part of the shader computations, because each $\langle$*photon, camera-point*$\rangle$ can potentially have a different surface material to evaluate the contribution. Furthermore, we did not try to aggresively optimize SIMD implementation. Fine-tuning a SIMD implementation is a meticulous process that often requires brute-force trial and error, but can make a significant difference in perfor-

mance. It is very likely that careful use of the prefetch instruction and avoiding needlessly unpacking and repacking SIMD data can hide the latency of several hot spots in our implementation that are currently limiting performance.

The use of sample-point estimation for photon mapping still has many open questions. For many common lighting scenarios, fixed-width kernels are accurate enough, but in future work it will be appropriate to use a variable kernel width. In reverse photon mapping, Havran et al. [4] used the sliding-midpoint KD-tree to estimate kernel widths based on the density of photons in leaf nodes. We tried to apply this same technique with forward photon mapping, but found that the sliding-midpoint KD-tree resulted in too many wasteful shader computations. The sample-point estimator will also be interesting to examine on the GPU, because it has less overhead than the traditional k-nearest neighbor estimation technique and is conducive to massive parallelism.

Finally there are many techniques that could be combined with SIMD photon mapping towards interactive performance in the future work. These include, but are not limited to, frameless rendering [2], temporal coherence [18], density control [17], GPU acceleration [12], custom hardware acceleration [15], and integration with other effective real-time ray tracing techniques for global illumination [23].

8 Conclusion

We have presented a framework that uses SIMD (single instruction, multiple data) parallelism to accelerate photon mapping. Until now, it was unclear how to effectively use SIMD extensions, because the k-nearest neighbor search had data structure overheads and branching logic that is not appropriate for SIMD. By applying the sample-point estimator, previously used in reverse photon mapping, we were able to reduce the overheads that prevented the use of SIMD instructions. At the same time, the question arises whether we should simply apply SIMD instructions to reverse photon mapping, or if forward photon mapping is more applicable to real-time. Since an algorithmic analysis was not enough, we provided a novel bandwidth analysis to show that forward photon mapping and reverse photon mapping have the same bandwidth scalability, but that forward photon mapping has a slight advantage in the context of SIMD parallelism and partial reordering. We demonstrated these contributions with a framework that can compute millions of shader operations per second, which is enough for interactive performance at low resolutions.

Acknowledgements

This work was partially supported by the NSF grant CCF-0429983. We would like to thank Joshua Steinhurst, Anna Majkowska, and the anonymous reviewers for their comments. We would also like to thank Intel Corp., Microsoft Corp., Ageia Corp., and ATI Corp. for their generous support through equipment and software grants.

References

[1] P. M. Campbell, K. D. Devine, J. E. Flaherty, L. G. Gervasio, and J. D. Teresco. Dynamic octree load balancing using space-filling curves. Technical Report CS-03-01, Williams College Department of Computer Science, 2003.

[2] A. Dayal, C. Woolley, B. Watson, and D. P. Luebke. Adaptive frameless rendering. In O. Deussen, A. Keller, K. Bala, P. Dutré, D. W. Fellner, and S. N. Spencer, editors, *Rendering Techniques*, pages 265–275. Eurographics Association, 2005.

[3] J. Günther, I. Wald, and P. Slusallek. Realtime caustics using distributed photon mapping. In *Rendering Techniques 2004, Proceedings of the Eurographics Symposium on Rendering*, pages 111–121, June 2004.

[4] V. Havran, R. Herzog, and H.-P. Seidel. Fast final gathering via reverse photon mapping. In *Proceedings of Eurographics 2005*, volume 24, pages 323–333, Dublin, Ireland, August 2005. Blackweel.

[5] H. W. Jensen. Global illumination using photon maps. In *Rendering Techniques '96 (Proceedings of the Seventh Eurographics Workshop on Rendering)*, pages 21–30, New York, NY, 1996. Springer-Verlag/Wien.

[6] H. W. Jensen. *Realistic Image Synthesis Using Photon Mapping*. A. K. Peters, Ltd., 2001.

[7] T. Kato and J. Saito. "Kilauea": parallel global illumination renderer. In *EGPGV*, pages 7–16, 2002.

[8] B. D. Larsen and N. J. Christensen. Simulating photon mapping for real-time applications. In A. Keller and H. W. Jensen, editors, *Rendering Techniques*, pages 123–132. Eurographics Association, 2004.

[9] V. C. H. Ma and M. D. McCool. Low latency photon mapping using block hashing. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 89–99, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[10] M. Pharr and G. Humphreys. *Physically Based Rendering: from Theory to Implementation*. Morgan Kaufmann Publishers, 2004.

[11] M. Pharr, C. Kolb, R. Gershbein, and P. Hanrahan. Rendering complex scenes with memory-coherent ray tracing. *Computer Graphics*, 31(Annual Conference Series):101–108, 1997.

[12] T. J. Purcell, C. Donner, M. Cammarano, H. W. Jensen, and P. Hanrahan. Photon mapping on programmable graphics hardware. In *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 41–50, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[13] B. Silverman. *Density Estimation for Statistics and Data Analysis*. Chapman and Hall, 1985.

[14] S. Singh and P. Faloutsos. The photon pipeline revisited. *The Visual Computer*, 23(7):479–492, 2007.

[15] J. Steinhurst. PhD thesis, University of North Carolina, 2007.

[16] J. Steinhurst, G. Coombe, and A. Lastra. Reordering for cache conscious photon mapping. In *GI '05: Proceedings of the 2005 conference on Graphics interface*, pages 97–104. Canadian Human-Computer Communications Society, 2005.

[17] F. Suykens and Y. D. Willems. Density control for photon maps. In B. Peroche and H. Rushmeier, editors, *Rendering Techniques 2000 (Proceedings of the Eleventh Eurographics Workshop on Rendering)*, pages 23–34, New York, NY, 2000. Springer Wien.

[18] T. Tawara, K. Myszkowski, K. Dmitriev, V. Havran, C. Damez, and H.-P. Seidel. Exploiting temporal coherence in global illumination. In *SCCG '04: Proceedings of the 20th spring conference on Computer graphics*, pages 23–33, New York, NY, USA, 2004. ACM Press.

[19] G. R. Terrell and D. W. Scott. Variable kernel density estimation. *The Annals of Statistics*, 20(3):1236–1265, 1992.

[20] I. Wald. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University, 2004. Available at http://www.mpi-sb.mpg.de/~wald/PhD/.

[21] I. Wald, C. Benthin, M. Wagner, and P. Slusallek. Interactive rendering with coherent ray tracing. In A. Chalmers and T.-M. Rhyne, editors, *Computer Graphics Forum (Proceedings of EUROGRAPHICS 2001*, volume 20, pages 153–164. Blackwell Publishers, Oxford, 2001. available at http://graphics.cs.uni-sb.de/ wald/Publications.

[22] I. Wald, J. Günther, and P. Slusallek. Balancing considered harmful - faster photon mapping using the voxel volume heuristic. *Comput. Graph. Forum*, 23(3):595–604, 2004.

[23] I. Wald, T. Kollig, C. Benthin, A. Keller, and P. Slusallek. Interactive global illumination using fast ray tracing. In *Proceedings of the 13th EUROGRAPHICS Workshop on Rendering*. Saarland University, Kaiserslautern University, 2002.

[24] I. Wald, W. R. Mark, J. Günther, S. Boulos, T. Ize, W. Hunt, S. G. Parker, and P. Shirley. State of the art in ray tracing animated scenes. In *STAR Proceedings of Eurographics 2007*. Eurographics Association, Sept. 2007. to appear.

[25] I. Wald, T. J. Purcell, J. Schmittler, C. Benthin, and P. Slusallek. Realtime ray tracing and its use for interactive global illumination. In *Eurographics State of the Art Reports*, 2003.