The Visual Computer manuscript No.
(will be inserted by the editor)

The Photon Pipeline Revisited

A Hardware Architecture to Accelerate Photon Mapping

Shawn Singh, Petros Faloutsos

Computer Science Department, University of California, Los Angeles

Received: date / Revised version: date

Abstract With the development of real-time ray tracing in recent years, it is now very interesting to ask if real-time performance can be achieved for high-quality rendering algorithms based on ray tracing. In this paper, we propose a pipelined architecture to implement reverse photon mapping. Our architecture can use real-time ray tracing to generate photon points and camera points, so the main challenge is how to implement the gathering phase that computes the final image. Traditionally, the gathering phase of photon mapping has only allowed coarse-grain parallelism, and this situation has been a source of inefficiency, cache thrashing, and limited throughput. To avail fine-grain pipelining and data parallelism, we arrange computations so that photons can be processed independently, similar to the way that triangles are efficiently processed in traditional real-time graphics hardware. We employ several techniques to improve cache behavior and to reduce communication overhead. Simulations show that the bandwidth requirements of this architecture are within the capacity of current and future hardware, and this suggests that photon mapping may be a good choice for real-time performance in the future.

1 Introduction

One of the main goals in computer graphics is to achieve real-time, photorealistic rendering. The state of the art is close to this goal – a decent subset of global illumination effects is possible in real-time with large-scale parallelism. However, for real-time global illumination that includes physically accurate lighting effects, performance still needs to improve by at least an order of magnitude.

These days, tracing a ray through a scene can be done very fast [36], and it is straightforward to model full global illumination with ray tracing. The current state of the art can trace 4-8 million rays per second on a single processor, which is enough to render 15 frames per second at 512×512 resolution, with basic Phong shading. With specialized ray tracing hardware, there will be significantly more parallelism and efficiency, and it is projected that hundreds of millions of rays per second can be achieved within the next 5-10 years [32]. This suggests that within 5-10 years a variety of high-quality algorithms that use ray tracing can also work in real-time – for example, bidirectional path tracing, instant global illumination, or photon mapping.

We feel that photon mapping is promising for real-time performance. It is widely agreed to be an efficient, versatile method for realistic image synthesis. Compared to classic path tracing and bidirectional path tracing, photon mapping requires fewer number of rays for a given image quality. At the same time, it can handle glossy reflections and complex light paths better than current real-time techniques such as instant global illumination. The first phase of photon mapping can already benefit from real-time ray tracing: photons are generated by tracing rays from light sources, and camera points are generated by tracing rays from the camera view.

The main issue discussed in this paper is how to effectively implement the second phase of photon mapping, which we call *gathering*. In this phase, each camera point uses the photons to compute a *radiance estimate* that is displayed in the final image. Mathematically, this is done by estimating the density of photons. Traditional photon mapping uses k *nearest neighbor* (kNN) density estimation, where each camera point searches for k nearest photons. In the past this process has been slow, because each kNN search was performed immediately after each ray tracing operation. This order of operations not only thrashes cache, but also prevents the use of fine-grain pipelining and data parallelism, two key properties of existing real-time graphics hardware.

Our key observation is that *sample-point estimation*, another kernel density estimation technique, exposes a finer granularity of parallelism. Instead of waiting for k nearest photons to be found for each camera point, we process every $\langle$*photon, camera point*$\rangle$ pair indepen-

dently. Each pair contributes a *partial radiance estimate* directly to the pixels of the image. Sample-point estimation and kNN estimation are both adaptive kernel estimation techniques, and thus using sample-point estimation does not degrade the quality of density estimation [33]. Exposing this fine-grain data parallelism is the key to our approach, because it allows us to implement a pipeline that can achieve large-scale parallelism on a single piece of hardware.

Our approach builds on a variation of photon mapping known as *reverse photon mapping* [12]. In their work, they also use sample-point estimation, but it is used only to gain algorithmic benefits. Instead of camera points searching for photons, photons search for camera points. As a result, the logarithmic dependency is placed on the larger set of points (camera points), and the linear dependency is placed on the smaller set of points (photons). In software, reverse photon mapping works efficiently, but not real-time.

Contributions. In this paper, we present an efficient, pipelined architecture that implements reverse photon mapping. This work has several main contributions. First, we modify the reverse photon mapping algorithm so that each photon contributes directly to pixels on the image. Second, we propose the architecture itself, which is designed for fine-grain pipelining, data parallelism, and good cache behavior to achieve high throughput. Finally, we quantitatively demonstrate that the bandwidth requirements for photon mapping are indeed feasible, even on today's hardware technology.

The rest of this paper is organized as follows. In Section 2 we describe background and related work. Section 3 discusses sample-point estimation and reverse photon mapping. Our architecture is detailed in Section 4. Section 4.2 describes the KD-tree build process, Section 4.3 describes the tree traversal process, and Section 4.4 describes the shader stage. An analysis of cache behavior and bandwidth is given in Section 5. In Section 6 we discuss limitations and future work. Finally, we conclude in Section 7.

2 Background and Related Work

Ray tracing. There are two tasks to trace a ray, and both can be done extremely fast. The first task is to compute the point of intersection between a ray and a geometric primitive, such as a triangle. The state of the art can efficiently compute a ray-triangle intersection test in less than 25 floating-point operations [34]. Recent work suggests that intersections can also be computed in real-time for complex surfaces such as NURBS [2].

The second task is to decide what intersection tests must be performed. Naively, a ray must be tested against every object or triangle in the scene. A better approach is to organize geometry so that it can be searched easily, for example in a *KD-tree* – a binary tree where each node divides a k-dimensional space into two regions. Havran [11] presents a detailed survey of the data structures that can be used to accelerate the tracing of rays and shows that KD-trees are generally the best choice. Most state of the art approaches build a KD-tree that optimizes the cost of tracing a ray through each node instead of building a balanced tree [22].

Real-time ray tracing has a substantial list of advantages over the traditional polygon pipeline [36]. A fully programmable ray tracing architecture has been prototyped [37], demonstrating a variety of practical scenes rendered in real-time with simple shading. Simulations show that a real hardware implementation will soon be able to trace hundreds of millions of rays per second [32]. With this level of performance it will be possible to implement high-quality rendering algorithms based on ray tracing that have real-time performance.

Global illumination using ray tracing. Generally speaking, these high-quality rendering algorithms use ray tracing to determine light paths between the camera view and light sources. For example, basic Monte Carlo path tracing [24], [8] traces light paths starting from the camera view. When the ray intersects a surface, a new ray is generated based on the reflection properties of the surface, and this ray is also traced. At each point along the path, rays are traced to light sources to compute the illumination that contributes to the path. Repeatedly tracing sample paths eventually converges to an accurate estimate of the lighting for a given pixel.

On the other extreme, photon splatting [20] traces paths starting from light sources. *Photons* are abstract points emitted by light sources and scattered throughout the scene by simulating how light reflects off each object. While photons are traced through the scene, they are projected, or "splatted," onto the camera plane. Most photons will not be visible in the final image, so many photons are required to ensure that enough of them contribute to the final image. Photon splatting has been proposed for real-time, however in this form it is dependent on rasterization and loses many of the advantages of ray tracing.

In both Monte Carlo path tracing and photon splatting, the number of rays required to produce a realistic image is unreasonably high for real-time performance. A typical image rendered with Monte Carlo ray tracing may require thousands of paths per pixel, and each path would require at least 6-10 rays, including shadow rays [24]. Bidirectional path tracing [18] reduces this requirement by tracing from both the camera view and light sources, completing paths somewhere in the middle. This approach requires significantly fewer total rays for a given image quality, but still may require thousands of paths per pixel.

Two methods that require even fewer rays for full global illumination are instant global illumination [35] and photon mapping [14], both of which use a bidirectional approach. Instant global illumination is based on instant radiosity [17], which traces a minimal number of

particles from light sources, and uses these particles as *virtual point lights* (VPLs). The original instant radiosity method uses multi-pass rendering on the traditional polygon pipeline, so in practice it is limited to diffuse surfaces. Instant global illumination uses only ray tracing, which allows it to handle non-diffuse reflections, such as caustics or glossy surfaces. Instant global illumination can be understood as bidirectional path tracing with significant re-use of the paths traced from light sources, and it has been demonstrated in real-time with large-scale parallelism.

Photon mapping. Photon mapping goes one step further. Instead of treating particles as virtual light sources, it uses density estimation to compute how each particle (photon) contributes to a given camera point. While the computational overhead is slightly more than computing with virtual point lights, photon mapping does not need to trace a shadow ray for each virtual point light, and so the number of photons can scale much higher than the number of VPLs without affecting performance. For high image quality, instant global illumination and photon mapping may also require hundreds or thousands of samples per pixel, but unlike all previously mentioned approaches, photon mapping would require only one or two rays per sample.

Heckbert [13] observed that the two-pass approach (simulating light transport and computing visibility) is *density estimation*, a well studied field of statistics [28]. The goal of density estimation is to estimate the underlying density of a function at any point, using samples at various point locations. In our context, rays traced from the camera view generate points where we want to estimate the density of photons. Initially this notion was applied only to caustics [5], [7]. Soon after, it was applied to overcome the setbacks of radiosity [16], [27]. Since then, density estimation has become widely used as way to visualize particle tracing illumination information [8]. The use of density estimation in photon mapping is particularly effective, because the representation of illumination, photons, is separated from geometry.

Prior work has tried to address performance bottlenecks in photon mapping, aiming to eventually accelerate it towards real-time. P. H. Christensen [6] precomputes irradiance at a subset of photon locations to speed up radiance estimates. Larsen and N.J. Christensen [19] simulate the photon map using a hemi-cube render-to-texture approximation on the GPU. Purcell et al. [25] have mapped the photon mapping algorithm to the streaming paradigm, demonstrating it very effectively on the GPU. However, because the GPU does not support recursion or unrestricted access to memory, these approaches require many rendering passes and simpler data structures, resulting in limited scalability.

As a growing number of processing cores share the same memory bandwidth on a single piece of hardware, bandwidth becomes the bottleneck in photon mapping, and some work tried to address this. Ma and McCool [21] use hashing to implement an approximate nearest neighbor search. Hashing avoids the need to traverse a tree structure and also reduces memory overhead. Steinhurst et al. [31] show that reordering the kNN queries can improve cache behavior theoretically by several orders of magnitude. In practice they were able to reduce the bandwidth requirement of photon mapping down to tens of gigabytes per frame, an order of magnitude improvement. Reverse photon mapping [12], the algorithm we use in this paper, also shows good cache behavior by reordering computations.

Comparison to our work. The main contribution of our work is to propose one of the first hardware architectures to accelerate photon mapping. To our knowledge, only one other hardware architecture is being proposed for photon mapping [30], concurrent to our work. While most previous work on photon mapping tries to approximate or speed up the k nearest neighbor search, we avoid this overhead by using a fixed-radius search, similar to reverse photon mapping [12]. However, our implementation is different from the original reverse photon mapping in two major ways. First, we modify reverse photon mapping so that each photon contributes directly to each pixel. Our modification exposes fine-grain parallelism and further reduces the overhead of the fixed-radius search. Second, we use a breadth-first traversal instead of the usual depth-first traversal. This results in more coherent ordering of operations that improves cache behavior.

Unlike other approaches to real-time global illumination, our architecture is designed to allow for pipelining, drawing from the early success of graphics hardware [4]. Most implementations of global illumination use coarse-grain parallelism on clusters of CPUs [26], or the streaming paradigm on GPUs, like many of the works described above. However, the bandwidth available on a network to connect CPUs is not enough for a high-performance pipeline, and today's GPU programming paradigm does not explicitly allow for pipelining of different stages of an algorithm. We feel that pipelining is an important technique in addition to multithreading and data parallelism. Instead of requiring each processing element to satisfy multiple functions, each stage in a pipeline can be fully optimized to its task, resulting in a simpler, more compact implementation – allowing for large-scale parallelism on a single piece of hardware.

Terminology. We use the term *camera rays* to denote any rays that originate from the camera view, including secondary rays. We use the term *camera points* to denote the points that were generated by tracing camera rays, while *photons* are the points generated by tracing rays from light sources. Recall that *radiance* is power per unit area per unit solid angle. A *radiance estimate* is an approximation of radiance exiting a surface point along a specific direction (ray). Given a ray, photon mapping computes a radiance estimate based on the density and lighting information of nearby photons.

3 Exposing Fine-grain Parallelisim of Reverse Photon Mapping

In this section, we briefly describe photon mapping and reverse photon mapping, and we show how fine-grain parallelism can be exposed in reverse photon mapping. Reverse photon mapping was first presented by Havran et al. [12]. As the name suggests, it is essentially photon mapping, except the algorithm is inverted. Reverse photon mapping is physically based and just as accurate as photon mapping, shown in Figure 1. It retains the benefits of traditional photon mapping, specifically that it is logarithmically scalable and can handle arbitrary BRDFs using importance sampling. Havran et al. showed that under certain reasonable assumptions, reverse photon mapping is algorithmically faster than traditional photon mapping. They also described a coarse-grain parallel implementation that is good for software and reordering for good cache behavior.

Mathematically, the difference between traditional and reverse photon mapping is the density estimation technique used to compute radiance along camera rays: the k nearest neighbor estimator is used in traditional photon mapping, while the sample-point estimator is used in reverse photon mapping. Both techniques are *adaptive kernel estimators*, a class of density estimation techniques that can robustly handle arbitrary data without any *a priori* knowledge about the data. These two different techniques lead to different search methods, and in turn different search methods result in different algorithmic properties. In Section 3.1, we describe both density estimation techniques, leading up to the key observation that allows us to achieve fine-grain parallelism. In Section 3.2, we discuss one important issue about how to use sample-point estimation in practice.

3.1 Adaptive Kernel Estimators

K nearest neighbor estimator. Let us first recall traditional photon mapping [15]. First, photons are traced from light sources. These photons are organized into a KD-tree to make them easy to search. Then, camera rays are traced, generating a set of camera points. Each camera point searches for k nearest photons. Using these photons and the radius r that encloses these photons, the radiance along each camera ray can be computed. In density estimation literature, this k nearest neighbor estimate is known as a *balloon estimator* [33]:

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

Here, 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. Note that, 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; this severely limits the potential throughput of the algorithm.

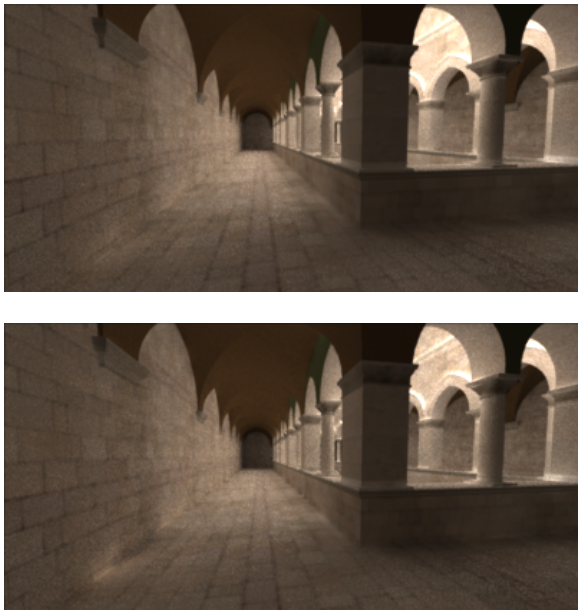

Fig. 1 Sponza scene (model by Marko Dabrovic) using 200k photons and 10.2 million camera rays. The top image was rendered with traditional kNN gathering ($k = 50$). It took 142 seconds, using the equivalent of 500 million shader operations. The bottom image used reverse gathering, taking 64 seconds with 500 million shader operations. This image shows only indirect lighting, so the visual quality is strongly dependent on the quality of the density estimation.

Sample-point estimator. Reverse photon mapping uses a different density estimation technique known as the *sample-point estimator* [33]. The sample-point radiance estimate can be written as:

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

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 each camera point searching for photons, this time all n photons search for nearby camera points. This is a *fixed-radius* search through the set of camera points, updating any rays that are located within the kernel width of the photon. Even though it is a fixed-radius search, the radius used by each photon can vary. Varying the kernel width is discussed in Section 3.2.

The resulting algorithm is naturally inverted. First rays are traced from the camera, rendering direct lighting to the image and building a KD-tree of camera points.

```
For each primary ray {
  Search for k nearest photons
  // wait for all k photons
  Determine radius r
  For i = 1 to k {
    totalScale =  Kernel * BRDF
    estimate += photonPower[i] * totalScale
  }
  estimate = estimate / (PI * r * r)
  Add final estimate to pixel
}
```

Traditional Radiance Estimate

```
For each photon {
  Search for camera points within distance h
  For each camera point {
    totalScale =  Kernel * BRDF / h
    estimate = photonPower * totalScale
    estimate *= partial_contribution
    Accumulate partial estimate to pixel
  }
}
```

Reverse Radiance Estimate

Fig. 2 Comparison of the traditional k nearest neighbor radiance estimate and sample-point radiance estimate. A single kNN estimate uses k photons for one camera ray. Note there is an implicit synchronization barrier that requires waiting for all k photons to be found before they can be processed; this severely limits throughput of any possible implementation. The sample-point estimator processes one photon for one final gather ray, computing a partial sum. The cumulative result is almost the same, but now each partial sum can proceed independently without any synchronization.

Then photons are traced from the light sources. Each photon then searches through the KD-tree to find neighboring camera points. Any camera point that is within the photon's kernel width is updated by computing a portion of the sum from Equation 2, and finally the result is added to the pixel associated with that camera point.

Algorithmically, reverse photon mapping places the logarithmic dependency on the set of camera points, rather than the photons. This is beneficial when the number of camera samples is much greater than the number of photons. In typical scenes, it is common to trace 1-3 orders of magnitude more camera rays than photons. For more algorithmic analysis and a sense of the data structures we use to represent photons and camera points, see the original work by Havran et al. [12].

Our key observation. Note that there are no parameters outside of the summation in Equation 2. Reverse photon mapping does not have to collect k neighbors before processing portions of the sum. Instead, each portion of the sum can be computed independently and applied directly to the appropriate pixels in any order, as shown in Figure 2. With sample-point estimation, there are typically 50-100 $\langle$*photon, camera point*$\rangle$ pairs to compute for every camera point, and therefore there is 50-100 times more parallelism to exploit than is possible with the kNN balloon estimator.

3.2 Varying the Kernel Width

One important concern about using the sample-point estimator is how to determine a good kernel width, denoted as $h(x_i)$ in Equation 2. It has been shown that a constant value for $h(x_i)$ is only good when there is an extremely dense set of samples, otherwise, it is better to vary $h(x_i)$ based on the density itself [33]. This is a chicken-and-egg problem, because the density is what we want to compute in the first place. Thus, it is common to take an initial coarse estimate of density to determine an appropriate kernel width.

Havran et al. [12] calculate $h(x_i)$ by building a second KD-tree over the set of photons. The density can be approximated by knowing the leaf node's bounding box as well as the number of photons contained in the leaf. We use the same technique in this paper, though it has some bandwidth overhead. It is generally agreed in density estimation literature that the quality of the final estimate is relatively unaffected by the quality of the initial coarse estimate [3], [28], so it is possible to find faster ways to compute an initial coarse estimate in future work. Note that building the second KD-tree also sorts the photons coherently, which is a critical issue discussed in Section 5.2 below.

4 The Photon Pipeline

In Section 3 we observed that sample-point estimation exposes fine-grain data parallelism in reverse photon mapping. In this section, we describe how our architecture implements the gathering phase of reverse photon mapping to benefit from this parallelism. The gathering algorithm is divided into three explicit stages: the KD-tree build, KD-tree traversal, and shader stages. Section 4.1 gives an overview of the architecture, and then the following three sections discuss each of the three stages. Performance is discussed in Section 5.

4.1 Architecture Overview

The conceptual layout of reverse photon mapping can be seen in Figure 3. There are two *ray tracing units*: one to trace camera rays, and one to trace photons. The camera points are sent to the *KD-tree build stage*, where we build the KD-tree over these point samples. Next, photons are traversed through the KD-tree in the *tree traversal stage*. The purpose of this traversal is for each photon to find nearby camera points that it affects. This search generates shader operations that are scheduled

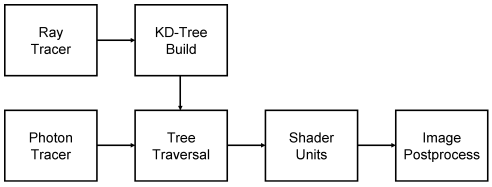

Fig. 3 Conceptual overview.

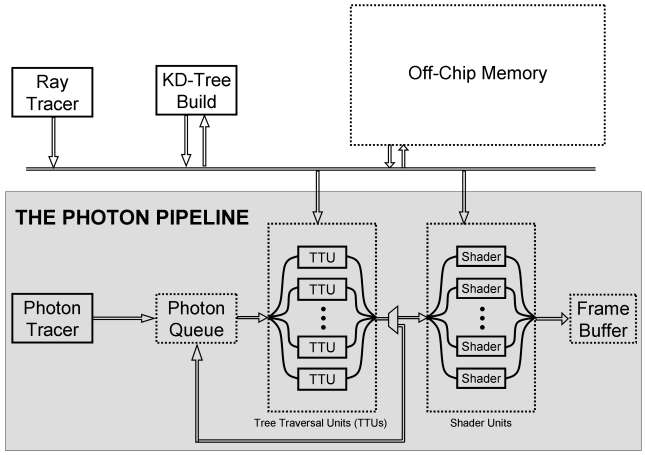

Fig. 4 Overview of the hardware design.

to the *shader stage*. The shader stage essentially implements Equation 2, and applies the result directly to the appropriate pixels. Finally, if desired, an *image postprocess stage* can apply tone-mapping or other image-space effects for display.

The computations within each stage are predictable and feed-forward, and can be easily pipelined. The KD-tree build only needs to calculate a spatial midpoint for each node, and then bin all data items to the left or right of this split plane. The tree traversal unit takes one photon and traverses it through one node of the tree; this requires only one subtraction and one comparison. The shader stage is more involved than the previous stages, but it is entirely feed-forward and can be pipelined. The simplicity of the stages is possible because the conditional branching between functions in software is replaced by conditional scheduling of operations between stages.

Figure 4 depicts a more detailed layout of the architecture. This demonstrates how data flows through the architecture. The ray tracing unit generates camera points, and writes them to memory. The KD-tree build unit reads and writes these camera points numerous times, spatially sorting them as part of the tree build process. Once the tree is built, the camera points reside in off-chip memory until they are used by the shader stage. The tree traversal stage can store the entire KD-tree in cache, so it only needs to manage a queue of photons scheduled to traverse the tree. The tree traversal unit then schedules a shader operation whenever a photon finds a nearby camera-point. For each shader operation, the shader stage tries to read the photon and camera point from cache, faulting to main memory when necessary. Note that caches can be designed to have a predictable latency, and delay pipelines can tolerate this latency of accessing cache without slowing down throughput. Also note that we can load-balance between the tree traversal and shader stages; this allows us to consider the tradeoff between a larger KD-tree that optimizes the number of required shader operations and a simpler KD-tree that results in performing more shader operations.

The data required for each photon and camera point is mostly the same as described in [12]. However, we choose a layout that separates "hot" and "cold" data. Most of the pipeline only needs to deal with the hot data: the 3D point location (12 bytes) of the photons or camera points, along with a 4 byte reference that refers to the remaining data. The remaining cold data is only needed in the shader stage. Cold data for camera points includes: compressed surface normal, compressed incident direction, any relevant BRDF parameters, the pixel location that the camera point belongs to, and the fraction of red, green, and blue contributions that this camera point has to the final image. Similarly, the cold data for photons includes: an incident direction, intensity and color information (we use three floating-point numbers for red, green, and blue), and the coarse estimate of $h(x_i)$ for variable kernel width.

4.2 The KD-tree Build Stage

There are two fundamental aspects to building a node of a KD-tree. First, we must find an appropriate split axis and position for each node of the tree. Second, we must spatially organize the data as we build each node. We use the *sliding-midpoint* rule, as described by Havran et al. [12]. To compute the split-plane for a given node, first we compute the *spatial* midpoint along the longest axis of the node's bounding box. Next, each data item (camera point or photon) is binned to the left or right of this midpoint. Finally, if one side does not have any points, the midpoint "slides" towards the location of the nearest data item.

To build the entire tree, this process repeats for every node starting from the root, until there are few enough data items in a node to define it as a leaf node. Based on the observations in Havran et al. [12], we store at most 30 points per leaf node. However, unlike their tree, we store one data point in each inner node as well. This allows us to schedule shader operations as the tree is being traversed, instead of waiting for all photons to reach the bottom of the tree.

It should be clear from this description that building a KD-tree requires very little computational throughput. Each node simply computes a spatial midpoint, compares each data item to the midpoint, and writes it to

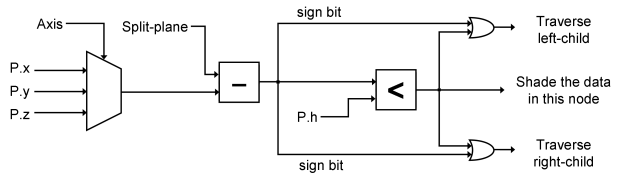

Fig. 5 Conceptual implementation of a single Tree Traversal Unit. One photon is traversed through one node of the KD-tree. The photon is re-scheduled to traverse one or both children nodes. If it is scheduled to both, then it also schedules a shader operation with the camera point in the current node, because this camera point may be within the photon's kernel width.

the left or right child. The major limitation in the performance of KD-tree is bandwidth requirement: for every floating-point comparison, the tree must read and write 16 bytes. In our previous work [29], we used a build method that streams all points from off-chip memory, compares each point to the spatial midpoint, and streams it back to memory. This approach typically requires tens of gigabytes of off-chip communication.

We revise this approach to use a conventional, recursive KD-tree build. Because it is recursive, once we reach a sub-tree that is small enough, all data points fit onto cache, and the sub-tree can be built with very little bandwidth cost. Results in Section 5.2 show that the recursive KD-tree build using a 1 MB cache can reduce the required bandwidth by more than half compared to our initial method which required the entire raw bandwidth.

The implementation of the KD-tree build is a straightforward controller that decides which node to build next. The controller maintains a stack to keep track of the recursive build. A typical sliding-midpoint KD-tree is 30-40 layers deep, so conservatively a compact stack of 100 pointers will suffice. As part of building one node, there are two basic states: `bin` and `node-compute`. In the `bin` state, we assume the spatial midpoint is given and traverse through all the data items once, binning them to the left or right of the midpoint. In the `node-compute` state, we apply the sliding midpoint rule, finalize data references for this node, and compute the spatial midpoint for each child node.

4.3 The Tree Traversal Stage

The fundamental operation performed by a single *tree traversal unit* (TTU) is to traverse one photon through one node of the KD-tree. Figure 5 shows that this operation is extremely simple. The TTU compares the photon's position to the split-plane of the node, and the TTU responds to three possible cases: (a) If the distance from the photon to the split-plane is less than the photon's kernel width, then the TTU schedules a ⟨*photon, camera-point*⟩ shader operation, and schedules the photon to traverse both children nodes (two separate traversal operations). (b) If the photon is strictly to the left of the split-plane, the TTU schedules the photon to traverse its left child. (c) If the photon is to the right, the TTU schedules the photon to traverse its right child.

Initially, each photon is scheduled to traverse the root node of the KD-tree, and the queue is processed in-order. Note that this is a "breadth-first" tree traversal – every photon traverses one layer of the KD-tree before going on to the next layer. The major benefit of the breadth-first approach is good data coherence. Because all photons will traverse one layer at a time, each layer of the KD-tree only needs to be loaded once and stored in cache. Even more importantly, similar shader operations are scheduled close together with our breadth-first technique. As a result, the ordering of shader operations is more coherent, thus saving bandwidth. Experimental data for cache behavior is discussed in Section 5.2.

The disadvantage of this scheduling approach is that we must manage a queue of traversal operations. This queue becomes too large to store on-chip, and therefore it will require some off-chip memory resources. Our simulations show that the bandwidth overhead is reasonable, so the benefits of our breadth-first scheduling approach seem to outweigh this problem.

4.4 The Shader Stage

The shader stage takes one photon and one camera point, computes the partial sum inside of the summation in Equation 2, and accumulates this partial sum directly to the image. The conceptual implementation is shown in Figure 6. There are two brief checks to make sure that the photon actually updates the camera ray. First, to ensure that the photon is on the correct side of the surface, this stage computes a dot product between the photon and shading normal. This assumes that the normal was already on the same side as the camera ray's incident direction; otherwise this check requires one more dot product between the camera ray direction and the shading normal. Second, the shader stage computes the squared distance between the photon and camera point, and makes sure it is less than the squared search radius (which is also the kernel width). If these two conditions are met, then the actual shading computation, described by Equation 2, is performed. This involves computing the kernel function $K()$, the BRDF $f()$, and the reciprocal of the kernel width. These factors scale the photon's color, and the entire result is finally scaled by how much the camera point contributes to the final image.

This computation is somewhat more complex than the previous stages, but it is still easy to see how it can be pipelined, requiring typical floating point multipliers, adders, and comparators. One floating-point division is needed to compute the reciprocal of the kernel width, denoted as `P.h`, but it can be computed well ahead of

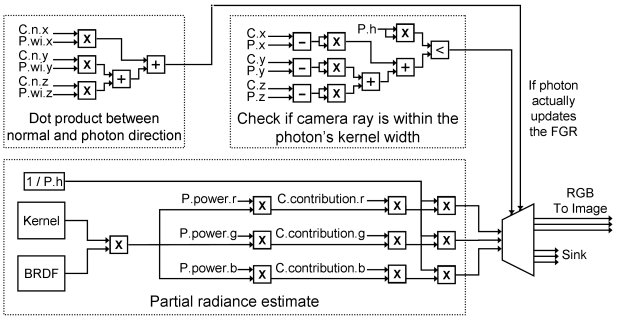

Fig. 6 Conceptual implementation of a single Shader Unit. This implements one portion of the sum described in Equation 2. It also verifies that the photon is on the correct side of the surface (the dot product) and that the distance between the photon and camera point is less than the kernel width before actually updating the appropriate pixel.

time or even cached while the same photon contributes to multiple camera points. Elaborate BRDFs and kernels may require costly square roots, exponentials, and divisions, but simpler versions do not. In the future, arbitrary BRDFs and kernels could be implemented as programmable shaders - either using an instruction set, or by using finer-grain programmable logic such as an FPGA. Also, we can vary the number of shader units for each TTU to load-balance the pipeline. Note that a $\langle$*photon, camera-point*$\rangle$ shading operation is analogous to shading a pixel in the traditional graphics pipeline.

5 Experimental Analysis

We have shown in the previous sections that the computations in each stage are predictable and easy to pipeline. This places the burden of performance on bandwidth. To demonstrate the feasibility of our architecture, we have simulated the cache behavior and bandwidth requirements for each stage. Our results indicate that the required bandwidth is possible even with today's memory speeds.

5.1 Simulation Method

For the KD-tree build unit, we simulate a 1 MB direct-mapped cache with 64-byte cache lines. Each cache line can store four 16-byte points, where each point contains 12 bytes for the 3D location and a 4 byte reference to the rest of the photon or camera point data. Whenever the requested data is not in cache, one cache line is flushed from the cache causing a 64-byte write operation, and the appropriate cache line is read from memory causing a 64-byte read operation. Note that we build *two* KD-trees: one tree to organize camera points for easy search, and the other tree to reorder the photons and to compute each photon's kernel width. The overhead of both is included in our simulations.

For the tree traversal stage, each KD-tree layer can be stored in a large cache, so we assume this does not count towards the bandwidth requirement. We do, however, manage a large queue of tree traversal operations; each operation in the queue consists of the photon's 3D point location and a reference to the tree node that will be traversed. Recall from Section 4.3 that the queue is processed in-order. This means that if one photon is scheduled to traverse many KD-nodes, these operations will all be consecutively queued. Therefore, we simulate a compact queue representation, containing one photon followed by a list of KD-nodes that it should traverse. This approach uses less than half the bandwidth of a straightforward queue.

For the shader stage, we simulate another 1 MB direct-mapped cache with 64-byte cache lines. Whenever there is a cache miss, the cache loads 64 bytes from off-chip memory. Unlike the KD-tree build, this cache does not have to write-back the old cache line, because the shader stage is read-only. Both photon data and camera point data use this cache; 3 photons fit in one cache line, while 2 camera points fit in one cache line.

We compute a conservative estimate the computational throughput in the following way. Both the KD-tree build and tree traversal stages require very little computation, so we round up to 10 floating-point operations for each build and traversal operation. The shader stage, as seen in Figure 6, requires at least 30 floating-point operations, however, to account for potentially complicated kernel and BRDF reflection functions, we round up to 100 floating-point operations. The purpose of this conservative estimate is to show that the requirements of high-performance photon mapping are within the abilities of existing graphics hardware.

5.2 Simulation Results

Table 1 gives an overview of our simulation results, and Figure 7 shows the scenes used for simulation. The required compute power and required bandwidth are feasible, even with today's hardware technology. For comparison, [9] showed that today's graphics hardware can provide more than 40 GB/s sustained bandwidth and 20 billion floating-point operations per second (GFLOPS) for scientific applications. Furthermore, last year's GPUs claim a peak theoretical computational throughput of 200-250 GFLOPS, while this year's newer architectures claim more than 300 GFLOPS [1]. With this in mind, these results indicate that our architecture would be capable of several frames per second at small resolution such as 300×300.

There is a strong trend of increasing parallelism on a single piece of hardware, and so we are less concerned

Fig. 7 Scenes used for simulating our architecture. These images show only indirect lighting. All images were rendered at 300×300, however, the results presented in Section 5.2 are generally dependent on the number of camera points rather than image resolution. As shown here, the Cornell Box uses 15 million camera points (approx. 160 samples per pixel. The Sponza scene uses 10 million camera points (approx. 115 samples per pixel) and Sibenik scenes both use 23 million camera points (256 samples per pixel).

Scene	# Camera Points (millions)	# Total Shader Ops	Estimated GFLOPS	Required Bandwidth
Cornell Box	3.7 M	69 M	7.8	1.02
	7.5 M	120 M	13.9	2.13
	15 M	220 M	25.6	4.68
Sponza	5.7 M	230 M	25.1	2.33
	11.5 M	430 M	46.5	5.14
	23 M	812 M	88.4	12.03
Sibenik	5.9 M	208 M	22.9	2.57
	11.8 M	376 M	41.4	5.17
	23.6 M	704 M	78.1	11.57

Table 1 Computational and bandwidth requirements for our architecture to render one frame per second. The conservative estimate of floating-point operations per second (FLOPS) is explained in Section 5.1. For comparison, note that current graphics hardware has a peak theoretical throughput of approximately 200 GFLOPS [1] and a sustainable bandwidth of 40 GB/s [9]. Because of the current hardware trend of increasing multi-core parallelism without a proportional increase in bandwidth, we focus on how to reduce bandwidth for future scalability.

with the availability of raw FLOPS speed in future hardware. On the other hand, bandwidth is not likely to scale as much as computational throughput. For this reason, we focus the rest of our analysis on cache behavior and bandwidth requirements to show that our architecture is feasible. The reader is also referred to Section 4 above, which discusses the techniques we use to achieve better cache behavior and reduced bandwidth.

Cache behavior. Table 2 shows cache behavior of the shader stage for various scenes. Our simulations use a simple direct-mapped cache, but in practice it is common to use a set-associative cache which usually performs better than a direct-mapped cache. The first point to notice is that reordering computations drastically improves cache behavior. This confirms previous results demonstrated by [31] and [12]. The method we use to reorder computations is from Havran et al. [12]. They build a second KD-tree over the photons to spatially sort the photons. While the KD-tree itself is not used, the sorted order of photons results in more coherent operations in the shader stage.

We also benefit from our novel breadth-first tree traversal, described in Section 4.3. At first it may seem that this improvement is minor, consistently improving cache behavior by only a few percent. However, once the cache hit rate is already in the 90% range, this few percent becomes significant. For example, in the Sibenik Cathedral scene with 11.8 million camera points, improving the cache hit rate from 95.2% to 97.9% translates to reducing the off-chip bandwidth by more than half.

Another interesting issue is the cache behavior of fixed-width versus adaptive-width kernel estimators. Intuitively it would seem that adaptive kernel estimators can reduce bandwidth by virtue of producing the same image quality with fewer samples. However, the cache behavior shows that this tradeoff is more intricate. For a good basis of comparison, we simulated both fixed-width and adaptive estimators to produce approximately the same number of total shader operations. The resulting cache behavior of the adaptive kernel is notably worse than a fixed kernel, resulting in more bandwidth. It would be interesting to explore this tradeoff more formally when trying to maximize performance.

Bandwidth requirement. A profile of the required bandwidth for the fixed-width kernel estimator is shown in Table 3. This shows that the largest cost of bandwidth comes from having to build a KD-tree in every frame. While this is surprisingly tractable, it is certainly necessary to reduce this portion of bandwidth for better scalability. Some possibilities to reduce this bandwidth are described in Section 6 below.

Scene	# Camera Points (millions)	Fixed-width Unordered D.F.	Fixed-width Unordered B.F.	Fixed-width Reordered D.F.	Fixed-width Reordered B.F.	Adaptive Reordered D.F.	Adaptive Reordered B.F.
Cornell Box	3.7 M	.660	.758	.968	.969	.949	.963
	7.5 M	.677	.754	.964	.971	.937	.962
	15 M	.690	.752	.955	.970	.920	.956
Sponza	5.7 M	.691	.787	.939	.976	.890	.962
	11.5 M	.697	.773	.894	.966	.837	.941
	23 M	.702	.766	.842	.943	.791	.907
Sibenik	5.9 M	.690	.782	.967	.978	.885	.940
	11.8 M	.699	.773	.952	.979	.861	.931
	23.6 M	.705	.769	.923	.974	.829	.915

Table 2 Cache behavior of the shader stage. This table compares depth-first (D.F.) traversal versus breadth-first (B.F) traversal, unordered photons versus reordered photons, and a fixed-width kernel versus an adaptive kernel. Cache behavior is likely to improve even more when an associative cache is used. See Section 5.2 for more discussion.

Scene	# Camera Points (millions)	# Photons (thousands)	Raw Bandwidth Profile	Shader Cache Hit Rate	Off-chip Bandwidth Profile	Total Bandwidth
Cornell Box	3.7 M	100k	2.20/0.33/4.12	.969	0.76/0.13/0.13	1.02
	7.5 M	100k	4.59/0.45/7.20	.971	1.75/0.18/0.20	2.13
	15 M	100k	9.64/0.65/13.12	.970	4.03/0.26/0.39	4.68
Sponza	5.7 M	200k	4.17/0.85/13.86	.976	1.65/0.34/0.34	2.33
	11.5 M	200k	8.65/1.27/25.60	.966	3.76/0.51/0.87	5.14
	23 M	200k	18.00/2.04/48.42	.943	8.44/0.82/2.77	12.03
Sibenik	5.9 M	200k	4.57/0.89/12.43	.978	1.94/0.36/0.27	2.57
	11.8 M	200k	9.30/1.26/22.44	.979	4.20/0.50/0.47	5.17
	23.6 M	200k	19.77/1.95/41.95	.974	9.71/0.78/1.08	11.57

Table 3 Fixed-width kernel bandwidth profile. This table shows the bandwidth requirements to render one frame per second using the fixed-width kernel estimator, breadth-first tree traversal, and reordered photons. The three numbers for bandwidth correspond to the Build/Traversal/Shader stages of the architecture, in gigabytes per second. These results are for the scenes shown in Figure 7. At 300×300, 23 million camera points is equivalent to about 256 samples per pixel, and at 500×500, it is equivalent to about 100 samples per pixel. For comparison, recall that current graphics hardware can sustain a bandwidth of 40 GBytes per second [9].

Scene	# Camera Points (millions)	# Photons (thousands)	Raw Bandwidth Profile	Shader Cache Hit Rate	Off-chip Bandwidth Profile	Total Bandwidth
Cornell Box	3.7 M	100k	2.20/0.33/3.91	.963	0.76/0.13/0.15	1.04
	7.5 M	100k	4.59/0.43/6.86	.962	1.75/0.17/0.26	2.18
	15 M	100k	9.64/0.63/12.56	.956	4.03/0.25/0.56	4.84
Sponza	5.7 M	200k	4.17/0.83/13.69	.962	1.65/0.33/0.52	2.50
	11.5 M	200k	8.65/1.25/25.57	.941	3.76/0.50/1.50	5.76
	23 M	200k	18.00/2.03/48.75	.907	8.44/0.81/4.55	13.80
Sibenik	5.9 M	200k	4.57/0.867/12.31	.940	1.94/0.35/0.74	3.03
	11.8 M	200k	9.30/1.24/22.65	.931	4.20/0.50/1.57	6.27
	23.6 M	200k	19.77/1.94/42.94	.915	9.71/0.78/3.63	14.12

Table 4 Adaptive kernel bandwidth profile. Compared to the fixed-width kernel estimator in Table 4, the adaptive kernel estimator requires more bandwidth, particularly in the shader stage. This table shows the bandwidth requirements to render one frame per second using the adaptive kernel estimator, breadth-first tree traversal, and reordered photons. The three numbers for bandwidth correspond to the Build/Traversal/Shader stages of the architecture, in gigabytes per second. These results are for the scenes shown in Figure 7. For comparison, recall that current graphics hardware can sustain a bandwidth of 40 GBytes per second [9].

The bandwidth required for the tree traversal is low, because it is approximately logarithmically dependent on the number of camera points. The performance of the tree traversal and shader stages was somewhat sensitive to the kernel widths used. In our simulations we chose a kernel width that produced approximately the same number of shader operations as a k nearest neighbor search where $k \approx 15$ for the Cornell Box scene, $k \approx 30$ for the Sibenik Cathedral scene, and $k \approx 35$ for the Sponza Atrium scene. Bandwidth is roughly linearly dependent on the number of shader operations, and the number of shader operations is linearly dependent on the number of photons as well as the number of camera points. The number of shader operations is quadratically dependent on the kernel width, but in practice good images could be rendered with slightly smaller kernels, such as the images in Figure 7.

Similarly, a profile of bandwidth for the adaptive kernel estimator is shown in Table 4. Note that the KD-tree build cost is the same, because the adaptive kernel does not affect how the tree is built. As mentioned before, cache behavior is worse, resulting in about a 1-2 GB/s bandwidth increase in worst case. For this extra bandwidth cost, it is not clear whether we can achieve better image quality by increasing the number of fixed-width kernel samples instead. Obviously, the adaptive kernel estimator will be more robust in very complex scenes or darker portions of the image where photons are sparse. For the specific images in Figure 7, there was no subjective difference between the image quality using a fixed-width kernel estimator and the image quality using an adaptive kernel estimator.

6 Discussion and Future Work

We predict that this architecture can fit easily on custom hardware, with room for large caches and plenty of parallelism. In such a case, it is likely our architecture will be capable of hundreds of GFLOPS and high sustainable bandwidth, much like current graphics hardware. As it was simulated, each stage required a 1 MB cache, resulting in a total 3 MB cache. We predict that increasing the cache further will greatly improve bandwidth requirements, because most memory access is already coherent. This argument applies to the recursive KD-tree build stage as well as the shader stage.

In addition to larger caches, in future work we plan to experiment with optimal ordering methods, such as Hilbert reordering [31]. This optimal method has been shown to have orders of magnitude better cache behavior, and we expect that Hilbert reordering could become an additional stage of our architecture.

Interestingly, our architecture has many similarities to a *sort-middle* architecture, which is one approach to implementing the traditional polygon pipeline with load-balanced, massive parallelism [23]. In a sense, the tree traversal is the sorting mechanism that occurs in between two main stages: tracing rays and performing shader operations. It would be interesting to explore whether other types of architectures, such as sort-first, sort-last, or sort-everywhere, could also apply to photon mapping.

Limitations and assumptions. While our breadth-first tree traversal has significantly better cache behavior, it is possible that depth-first traversal requires less bandwidth in the first place, and this is yet to be explored. We predict that an optimal traversal technique may lie somewhere in-between, resulting in an algorithm similar to the *dual-trees* method in density estimation literature [10]. We also assumed that reflection functions only need a surface normal and the directions associated with a camera point and a photon. While this is enough to implement physically realistic lighting, in practice it is much easier and more flexible to store additional reflection parameters in the form of textures. This would require extra bandwidth that we did not include in our simulations.

We assumed that the details of the architecture - controllers, queues, schedulers, and wiring will not be a bottleneck. To verify this, the architecture should be described and simulated at the logic-cell level of abstraction. This level of simulation would give us a better understanding of latency, throughput, power, and area requirements of a real hardware implementation. Furthermore, this architecture is fixed-function, not programmable. This immediately limits its applicability. There are many practical problems that require density estimation, given a large set of samples (photons) and a large set of queries (camera points). Allowing the kernel and BRDF functions to be programmable could be very powerful, combining the programmability analogous to existing GPU fragment shaders with a hardware accelerated tree build and search.

Most ray tracing based renderers are designed to handle static scenes, and they cannot efficiently handle dynamic scenes. It is very likely that addressing both this problem and the bottleneck of our KD-tree build stage are very related. Solving one may provide insight on how to solve the other. Finally, careful and intelligent sampling methods are required to effectively reduce the number of rays to be traced for a given image quality, and it is not clear how much overhead is caused by these high-level decisions.

Finally, it would be very interesting to try our variation of reverse photon mapping on the GPU. A more generalized stream-processing paradigm has emerged from programmable GPUs [1]. This new unified architecture has overcome some of the prior limitations of GPU programming, making it very interesting to experiment with. Additionally, previous attempts to map photon mapping to the GPU suffered because fine-grain data parallelism, which is ideal for a GPU program, was not exposed. The fine-grain data parallelism exposed in this paper

may map more easily and effectively to the streaming paradigm. It would also be informative to try mapping this algorithm to the programmable ray processing unit (RPU) [37] as well.

7 Conclusion

In this paper we have presented a novel architecture, the photon pipeline, that implements reverse photon mapping. To make our architecture possible, we first exposed fine-grain data parallelism that is available in reverse photon mapping. Data parallelism allows us to consider a compact, pipelined hardware architecture, similar to the key factors in the initial success of traditional graphics hardware. We described how our architecture implements the KD-tree build, KD-tree traversal, and shading operations that are part of reverse photon mapping, and we simulated cache behavior and bandwidth requirements to show that our architecture is feasible. We conclude that real-time photon mapping is certainly possible, and in future work there are many ways to continue progressing towards the holy grail of real-time photorealism.

Acknowledgements

This work was partially supported by the NSF grant CCF-0429983. We would also like to thank Intel Corp., Microsoft Corp. and ATI Corp. for their generous support through equipment and software grants. Lastly we would like to thank Joshua Steinhurst for several insightful discussions on this topic.

References

1. Nvidia cuda complete unified device architecture: Programming guide, version 0.8. 2007.
2. O. Abert, M. Geimer, and S. Müller. Direct and fast ray tracing of nurbs surfaces. In *Proceedings of IEEE Symposium on Interactive Ray Tracing 2006*, pages 161–168, September 2006.
3. I. S. Abramson. On bandwdith variation in kernel estimates — a square root law. *The Annals of Statistics*, 10(4):1217–1223, 1982.
4. T. Akenine-Moller and E. Haines. *Real-time Rendering*. A. K. Peters, Ltd., 2002.
5. S. E. Chen, H. E. Rushmeier, G. Miller, and D. Turner. A progressive multi-pass method for global illumination. In *SIGGRAPH '91: Proceedings of the 18th annual conference on Computer graphics and interactive techniques*, pages 165–174, New York, NY, USA, 1991. ACM Press.
6. P. H. Christensen. Faster photon map global illumination. *J. Graph. Tools*, 4(3):1–10, 1999.
7. S. Collins. Adaptive splatting for specular toDiffuse light transport (also inProceedings of the 5th eurographics workshop on rendering,1994). Technical Report TCD-CS-94-22, 1994.
8. P. Dutre, K. Bala, and P. Bekaert. *Advanced Global Illumination*. A. K. Peters, Ltd., Natick, MA, USA, 2002.
9. N. K. Govindaraju, S. Larsen, J. Gray, and D. Manocha. Memory—a memory model for scientific algorithms on graphics processors. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 89, New York, NY, USA, 2006. ACM Press.
10. A. Gray and A. Moore. Rapid evaluation of multiple density models, 2003.
11. V. Havran. *Heuristic Ray Shooting Algorithms*. Ph.d. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, November 2000.
12. 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.
13. P. S. Heckbert. Adaptive radiosity textures for bidirectional ray tracing. In *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pages 145–154, New York, NY, USA, 1990. ACM Press.
14. 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.
15. H. W. Jensen. *Realistic Image Synthesis Using Photon Mapping*. A. K. Peters, Ltd., 2001.
16. H. W. Jensen and N. J. Christensen. Photon maps in bidirectional Monte Carlo ray tracing of complex objects. *Computers and Graphics*, 19(2):215–224, Mar.–Apr. 1995.
17. A. Keller. Instant radiosity. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 49–56, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.
18. E. P. Lafortune and Y. D. Willems. Bi-directional Path Tracing. In H. P. Santo, editor, *Proceedings of Third International Conference on Computational Graphics and Visualization Techniques (Compugraphics '93)*, pages 145–153, Alvor, Portugal, 1993.
19. 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.
20. F. Lavignotte and M. Paulin. Scalable photon splatting for global illumination. In *GRAPHITE '03: Proceedings of the 1st international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, pages 203–ff, New York, NY, USA, 2003. ACM Press.
21. 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.
22. J. D. MacDonald and K. S. Booth. Heuristics for ray tracing using space subdivision. *The Visual Computer*, 6(3):153–166, 1990.
23. S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs. A sorting classification of parallel rendering. Technical Report TR94-023, 8, 1994.

24. M. Pharr and G. Humphreys. *Physically Based Rendering: from Theory to Implementation*. Morgan Kaufmann Publishers, 2004.
25. 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.
26. E. Reinhard, A. Chalmers, and F. W. Jansen. Overview of parallel photo-realistic graphics. Technical Report CS-EXT-1998-147, 1, 1998.
27. P. Shirley, B. Wade, P. M. Hubbard, D. Zareski, B. Walter, and D. P. Greenberg. Global Illumination via Density Estimation. In P. M. Hanrahan and W. Purgathofer, editors, *Rendering Techniques '95*, pages 219–230, New York, NY, 1995. Springer-Verlag.
28. B. Silverman. *Density Estimation for Statistics and Data Analysis*. Chapman and Hall, 1985.
29. S. Singh. The photon pipeline. In *GRAPHITE '06: Proceedings of the 4th international conference on Computer graphics and interactive techniques in Australasia and Southeast Asia*, pages 333–340, New York, NY, USA, 2006. ACM Press.
30. J. Steinhurst. PhD thesis, University of North Carolina, 2007.
31. 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.
32. E. B. Sven Woop and P. Slusallek. Estimating performance of a ray-tracing asic design. In *Proceedings of IEEE Symposium on Interactive Ray Tracing 2006*, September 2006.
33. G. R. Terrell and D. W. Scott. Variable kernel density estimation. *The Annals of Statistics*, 20(3):1236–1265, 1992.
34. 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/.
35. 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.
36. I. Wald and P. Slusallek. State of the art in interactive ray tracing. In *State of the Art Reports, EUROGRAPHICS 2001*, pages 21–42. EUROGRAPHICS, Manchester, United Kingdom, 2001.
37. S. Woop, J. Schmittler, and P. Slusallek. Rpu: a programmable ray processing unit for realtime ray tracing. *ACM Trans. Graph.*, 24(3):434–444, 2005.