Real-Time Photon Mapping on GPU

Sanket Gupte*
University of Maryland Baltimore County

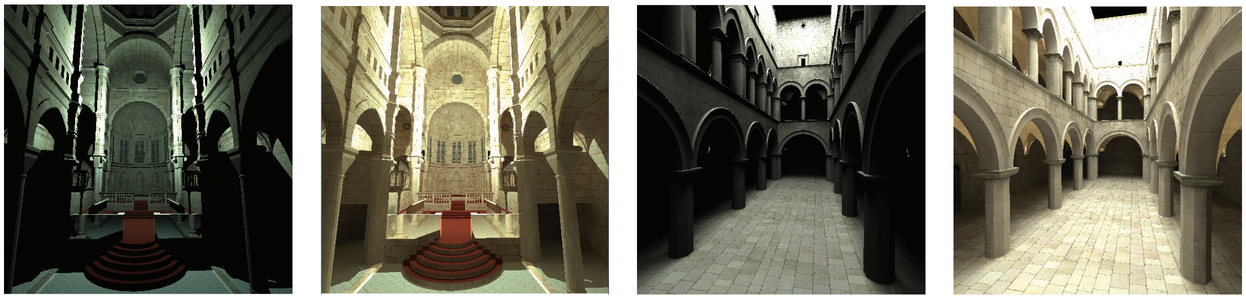


Figure 1: *Sample images rendered without and with photon-mapping*

Abstract

This paper presents a hybrid photon-mapping approach for global illumination . It represents a significant improvement over a previously described approach, both with respect to speed and accuracy.

Using OptiX for ray tracing provides a considerable improvement in the speed of ray tracing and would keep synchronization to a minimum by using texture memory to cache access to the photon information. [Parker et al. 2010]. Recent developments in processor design towards multi-core systems do not favor the tree based photon map approach. So, instead we use the Spatial Hashing Method [Fleisz 2009] to store and retrieve a photon map. Also, the density of photon maps is reduced by storing photons selectively on a local required density criterion, while preserving the correct illumination. At some selected locations with high photon density, we pre-compute the irradiance to speed-up the final gather stage.

We use the method to simulate some global illumination scenes and objects. Comparision with existing photon mapping techniques indicates that our method gives significant improvement in speed with the same or better accuracy of the scene.

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

Keywords: Photon mapping, global illumination, OptiX

Links: DL PDF

*e-mail: sgupte1@umbc.edu

1 Introduction

Fast and high quality global illumination has been a goal of computer graphics since a very long time. A huge variety of algorithms have been developed since the past twenty-five years and the techniques for global illumination have evolved a lot, from bidirectional path tracing, radiosity to photon mapping [Kajiya 1986; Lafortune and Willems 1993; Keller 1997; Henrik Wann Jensen 2000; Jensen 2009; Jensen 1996] The photon map method is an extension of ray tracing that makes it able to efficiently compute caustics and soft indirect illumination on surfaces and in participating media. As [Christensen 2000] describes, photon mapping has the following desirable properties for computation of Global Illumination

1. It can handle all combinations of specular, glossy, and diffuse reflection and transmission (including caustics).
2. Since the photon map is independent of surface representation, the method can handle very complex scenes, instanced geometry, and implicit and procedural geometry.
3. It is relatively fast.
4. It is simple to parallelize.

Photon mapping method in the simplest way can be divided into three steps: photon tracing, photon map sorting, and rendering. In the photon tracing step, photons are emitted from the light sources and traced through the scene and stored at the diffuse surfaces they intersect on their path. In the next step, the stored photons are sorted in such a way that they can be quickly accessed. The final step consists of rendering, where specular reflections and incident illumination on diffuse surface is computed. To compute the incident illumination a final gather is performed by shooting many rays to sample the hemisphere above the point. Each final gather is quite time-consuming since numerous rays are shot.

With recent developments in processor design, parallel computing has become the latest way of creating high performance applications. GPUs, with their enormous computation power are now easily available in a lot of consumer machines these days. [Fleisz 2009] To make use of the real potential of this processing power, all the algorithms need to be parallelized. [nVidia Corporation 2009]

Parallelizing the task of photon mapping can be a real challenge due to various reasons. A simple Photon mapping places an enormous burden on the memory hierarchy. Rendering a normal image of a simple scene can require hundreds of Gigabytes of raw bandwidth to the photon map data structure. This bandwidth is a major obstacle to real-time photon mapping. Also, Jensen discusses parallelization of photon search in [Davis et al. 2000a], but it is a quite straightforward way to do it and not too fast either. A data structure, for storing the photon map, is usually a balanced kd-tree [Bentley 1975] and each radiance lookup needs to find the k nearest neighbors in the kd-tree, which can be more costly than shooting several rays. Therefore, the nearest-neighbor queries often dominate the rendering time of a photon map based renderer. Moreover, all threads have to access the same data structure, so it is difficult to parallelize the tree construction.

2 Background and Previous work

To create a 2D image of a 3D world, the Ray Tracing technique is used. The first ray tracing algorithm was introduced by Arthur Apple [Appel 1968] and with some modifications is still used in current ray tracers. The problem with Ray Tracing and Radiosity is that they are not able to model all possible lighting effects in a scene. Ray tracing only simulates indirect illumination by adding constant ambient term in the lighting calculation, whereas Radiosity only simulates diffuse reflections, completely ignoring mirrored surfaces. Therefore, approaches were made combining both the methods, so that each technique would complement the other. However, such approaches are still not sufficient as they both fail to model focused light effects, like caustics. Solutions to this problem were presented by [Wallace et al. 1987; Rushmeir and Torrance 1990; Sillion et al. 1991] but they introduced other problems as [Jensen and Christensen 1995] explains.

Jensen [Jensen 1996] introduced photon mapping, which has since been expanded by several methods. All of these simulate light transport forward along rays from emitters through multiple bounces (i.e., scattering events) against the scene, and represent the resulting incident radiance samples as a photon map of stored photons at the bounce locations. At each bounce scattering is performed by Russian roulette sampling against the surfaces bidirectional scattering distribution function (BSDF). To produce a visible image, a renderer then simulates transport backwards from the camera and estimates incident radiance at each visible point from nearby samples in the photon map. Traditional photon mapping algorithms, [Jensen 1996; Jensen 2009]; simulate transport by geometric ray tracing and implement the radiance estimate by gathering nearest neighbors from a k-d tree photon map. This approach represents a significant improvement of a previously described approach for global illuminations both with respect to speed, accuracy and versatility. In the first pass two photon maps are created by emitting packets of energy (photons) from the light sources and storing these as they hit surfaces within the scene. One high resolution caustics photon map is used to render caustics that are visualized directly and one low resolution photon map that is used during the rendering step. The scene is rendered using a distribution ray tracing algorithm optimized by using the information in the photon maps. Shadow photons are used to render shadows more efficiently and the directional information in the photon map is used to generate optimized sampling directions and to limit the recursion in the distribution ray tracer by providing an estimate of the radiance on all surfaces with the exception of specular and highly glossy surfaces. The results presented demonstrate global illumination in scenes containing procedural objects and surfaces with diffuse and glossy reflection models.

Since the process of final gathering was quite time consuming, in [Christensen 2000] it was speeded up by pre-computing the irradiance at selected locations. The irradiance is pre-computed at photon positions, since the photons are inherently dense in areas with large illumination variation. Storing the computed irradiance and surface normal consumes about 28

Two approaches were investigated in [Steinhurst et al. 2008] for reducing the required bandwidth for photon mapping : 1) reordering the kNN searches; and 2) cache conscious data structures. An approximate lower bound of 15MB of bandwidth, reduced from 196GB was demonstrated using a Hilbert curve reordering. This improvement of four orders of magnitude needs a prohibitive amount of intermediate storage. Two more cost effective algorithms that reduce the bandwidth by one order of magnitude to 24GB with 1MB of storage were demonstrated. It was explained why the choice of data structure cannot, by itself, achieve this reduction. Irradiance caching, a popular technique that reduces the number of required kNN searches, receives the same proportional benefit as the higher quality photon gathers. The techniques used were more amenable to hardware implementation and capture a large portion of the possible gains. With the fast GPU memory bandwidth rates, the kNN search reordering will enable the development of interactive photon mapping hardware.

The two pass approach to Global illumination by Jensen is extended to a 3 pass solution by [Peter et al. 1998]. In the first pass, particle tracing of importance is performed to create a global data structure, called importance map. Based on this data structure importance driven photon tracing is used in the second pass to constrict a photon map containing information about the global illumination in the scene. In the last pass, the image is rendered by distributed ray tracing using the photon map. The photon tracing process, improved by the use of importance information, creates photon-maps with an up to 8-times higher photon density in important regions of the scene. This allows a better use of memory and computation time resulting in better image quality. The importance of a global illumination algorithm is emphasized. It also shows that importance driven construction of photon maps leads to better results. The photon maps thus constructed possess a high density at the points where they are used for global illumination calculation.

A modified photon mapping algorithm that is capable of running entirely on GPUs is presented in [Purcell et al. 2003]. The implementation uses breadth-first photon tracing to distribute photons using the GPU. Methods to construct a grid-based photon map and how to perform a search for at least k-nearest neighbors using the grid, entirely on the GPU are demonstrated. The photons are stored in a grid-based photon map that is constructed directly on the graphics hardware using one of two methods: the first method is a multipass technique that uses fragment programs to directly sort the photons into a compact grid. The second method uses a single rendering pass combining a vertex program and the stencil buffer to route photons to their respective grid cells, producing an approximate photon map. They also present an efficient method for locating the nearest photons in the grid, which makes it possible to compute an estimate of the radiance at any surface location in the scene. Finally, they describe a breadth-first stochastic ray tracer that uses the photon map to simulate full global illumination directly on the graphics hardware. All of the algorithms are compute bound, meaning that photon mapping performance will continue to improve as next-generation GPUs increase their floating point performance. Also several refinements for extending future graphics hardware to support these algorithms more efficiently is proposed.

In [Suykens and Willems 2000] they introduce a method to control the density of photon maps by storing photons selectively based on a local required density criterion and while ensuring a correct illumination representation in the map. This reduces memory usage significantly since less photons are stored in unimportant or overdense regions. Benefits of the original method like, the ability to handle complex geometry and materials are preserved. Results for caustic photon maps and global photon maps representing full illumination show a decrease in number of photons of a factor of 2 to 5. The required density states how accurate the photon map should be at a certain location and determines how many photons are needed in total. They use only a fixed number of nearest photons in the reconstruction.

Two data layout techniques for dynamically allocated data structures are mentioned in [Truong et al. 1998]: field reorganization, and instance interleaving. The application of these techniques may be guided by program profiling. This allows significant cache behavior improvements on some applications. To support instance interleaving a specific memory allocation library called ialloc was created. An ialloc-like library may be of great help in a toolbox for performance tuning of general-purpose applications. Field reorganization consists in regrouping together the most frequently used fields of a structure to fit them in a single cache line. This may reduce cache line space wasted. However there are generally only a few fields frequently used in a structure, not enough to fill a cache line. Therefore instance interleaving would be helpful here. It consists in grouping the most frequently used fields of several instances to fit them into the same cache line. Instance interleaving can be an efficient way to improve the memory behavior and the overall performance of the application.

In [Fabianowski and Dingliana 2009b], they first present a compact representation for BVH and then demonstrate and analyze its application in ray tracing and photon mapping. They give a compact representation for binary hierarchies of Axis aligned bounding boxes. By eliminating only redundant information, full bounding tightness is maintained while reducing the BVHs memory footprint by 43-50Image space photon mapping described in [McGuire and Luebke 2009] achieves real-time performance by computing the first and last scattering using a GPU rasterizer.

In [Fabianowski and Dingliana 2009a] they present a highly parallel photon mapping algorithm that utilizes CUDA architecture and computes diffuse and specular indirect lighting at interactive frame rates, but the geometry is static. By handling diffuse reflections using photon differentials, footprints are generated at all photon hit-points. This enables illumination reconstruction by density estimation with variable kernel bandwidths without having to locate k-nearest photons hit first. The BVH builds termination criterion is automatically tuned to the scene and illumination conditions using a special heuristic.

In [Havran et al. 2005] they accelerated the search for final gathering by reorganizing the computation in reverse order. Using two trees, the position of photons as well as final gather rays is organized spatially. By doing this the computation time for normal photon mapping is reduced by more than an order of magnitude, and the algorithmic speedup comes from the logarithmic factor of searching using trees and highly coherent access pattern to the data. It requires more memory but only small portion of data needs to be in main memory. This idea was again utilized in [Singh 2006] where they proposed a pipelined architecture for fast global illumination, which exploited fine-grain pipelining, parallelism, efficiency and good cache behavior. This paper was then revised with small changes in [Shawn Singh 2007]

[Grauer-Gray 2008] took an initiative in describing Photon-mapping entirely on a GPU using CUDA. It goes gradually from nave implementation to several modifications for improved rendering and effects. [Fleisz 2009] describes a spatial hashing approach instead of the usual tree based approaches and compared it with the benchmarks to display that the spatial hashing approach was much faster with the same image quality.

In order to reduce the cache misses and have higher cache utilization [Moon et al. 2010] propose an approach to compute a coherent ordering of rays, using hit points between rays and scene as ray ordering measure. They show that the approach is quite modular and produces a magnitude of improvement in performance not only in photon mapping but various other models and machines with different cache parameters too.

3 Implementation

Like any other rendering technique, the only goals of Photon-mapping are: to be as accurate as possible while achieving frame-rates that could render in real-time. In our implementation of photon mapping, we perform the entire set of computations on the GPU, and to achieve the best performance we utilize OptiX Ray tracing engine and the OptiX API. Due to the various limitations for OptiX we implement a set of approaches for different phases of photon mapping and compare them to the traditional or currently popular approaches, concluding our combination gives the best frame rates for photon mapping.

OpenGL and Direct3D were built on the basis that programmable rasterizations can be implemented using a small set of operations. The conception of OptiX [Parker et al. 2010] was based on a similar analogy applied to implement fast ray-tracing. With the development in ray tracing it has been possible to perform real-time ray tracing, but when it becomes a part of a more complex rendering, the overall rendering time is far from real-time. NVIDIA released OptiX, a ray tracing engine in Nov09. [Parker et al. 2010]. The OptiX engine focuses exclusively on the fundamental computations required for ray tracing only and avoids embedding rendering specific constructs, and demonstrates that most ray tracing algorithms can be implemented using a small set of light-weight programmable operations. It provides a very simple and abstract ray tracing execution model with the most accustomed execution mechanisms and a domain specific compiler. Moreover, OptiX uses a mega-kernel approach [Aila and Laine 2009] which minimizes kernel launch overhead. Using OptiX for photon mapping we can keep synchronization to a minimum and use texture memory to cache access to the photon information.

With all these advantages we opted to use OptiX for our implementation but using it entirely, for all the phases of the photon mapping is not feasible due to the following limitations. The OptiX buffer mechanism currently does not have extensions for append, reduction and sorting value operations. The OptiX acceleration structures only support triangle data. It would have been better if the acceleration structure was generalized, allowing additional programmable operations or provide a mechanism for the user to implement their own acceleration structures. These reasons, create a major impact over the accuracy and speedup of photon mapping, hence we fuse other techniques with it to produce close to accurate global illumination in real-time.

OptiX in Photon Mapping

An OptiX implementation essentially requires two things: a ray generation program and a material program, which gets called when a ray intersects geometry. Exactly as the words sound, the ray generation program generates rays and is called for each pixel of the programs dimensions. The ray cast by this program traverses the scene for intersections, and once a ray intersects geometry it calls the material program. In a classic ray tracer, the material program is responsible for shading. In our case, it is responsible for the computations and creation of the photon map. For all the phases of photon mapping various such programs are required such as program to read scene geometry, ray generation, ray-hit program, ray-miss program, photon pass program, photon gather program etc.

Different acceleration structures have their own advantages and drawbacks. Furthermore, different scenes require different kinds of acceleration structures for optimal performance. The most common tradeoff is construction speed vs. ray tracing performance. OptiX has a few built in acceleration structures, which could be used in the program, but we build our own a spatial hashing technique for storing and retrieving data in a photon map. The Spatial Hashing method provides faster recording and retrieval in cases when there is large amount of data involved. Hence we use it in Photon Map creation and access. At other places where acceleration structures are required, such as for scene objects and bounds storing, the built in kd-tree acceleration structures are used, since they are more suitable and faster.

The Photon Mapping Implementation

Given the time limitations and other restrictions, the concepts have been implemented as modifications in an already existing open source project MNRT http://www.maneumann.com . The comparison of our approach to the existing approach in the source code has been described in the last section.

The implementation can be divided into distinct components as shown in the figure below. Photons are sent out from direct light sources and traced thorough the scene. The objects of the scene are stored in the inbuilt OptiX Kd-tree acceleration structure. For each photon-surface interaction the position, incoming direction and flux of the photon is recorded in a photon map, which would technically be a spatial hash table. These photons can then be used to estimate the indirect illumination using density estimation. The detailed spatial hashing implementation is described in a separate section below.

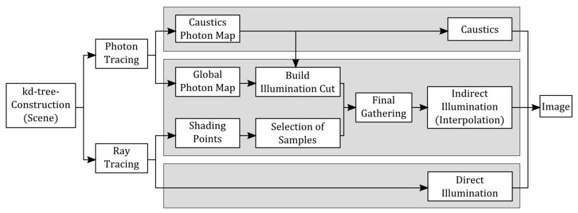


Figure 2: *Stages of Photon mapping*

To handle direct lighting, ray tracing techniques are used. Secondary rays are traced in a similar manner. At every point of intersection, the indirect illumination has to be calculated, which is done using the photons of the photon maps to estimate the incoming radiance from indirect light sources. To improve the quality of this process a final gathering is applied, where gather rays are sent out from each sample point into random directions over the sample point. The incoming radiance due to indirect illumination from the rays direction is estimated for each gather ray, and is used to estimate the incoming indirect illumination using Monte Carlo integration. To have a proper image and avoiding noise it is important to use enough number of gather rays. Creation and processing of rays is handled by the OptiX API.

Using modern hardware and a framework like OptiX, the efficiency of Ray tracing has increased by bounds. Having to include dynamic geometry requires a rebuild of the data structure, that stores the ray-object intersection, for each frame. Hence, we use a Spatial Hashing method to store and retrieve these intersections in a Hash table, which in this context would mean a photon map. [Wang et al. 2009] propose two photon maps: The caustics photon map, which stores all photon interactions that were reflected in a specular way before hitting the surface the interaction represents. Caustics can be captured by performing the density estimation directly without final gathering. All other photon interactions are stored in the global photon map, which is queried only by the use of final gathering. Local photon density is required for precise density estimation. A fixed query radius for the range search can blur out sharp caustics patters. [Jensen 2009] proposed kNN search to look for the closest k photons only. The use of spatial hashing data structure for storing photon data enables fast retrieval of data.

The final gathering stage collects all the sample points for rendering. This process can be speeded up by reducing the total number of sample points where we have to execute the final gathering algorithm. citeWardRub88 proposed the idea to sparsely sample the indirect illumination, observing that indirect illumination in a diffuse environment changes in a smooth way. A sample selection scheme can be used to choose all interpolation samples within a single pass, where the first step would select initial samples using an adaptive approach. The shading points (points of intersection) are classified according to the pixel they represent based on a geometric variation metric. It captures illumination changes by considering the geometric properties of the scene, which would mean more samples are assigned to regions of high geometric variation. The selection of samples can be improved by using a clustering algorithm like k-means iteratively.

To improve temporal coherence for sequences of images, it is advisable to retain sample points from the previous image for generating the next image. It improves performance and avoids possible flickering. The new shading points can be classified to the old shading points and only those that have a large error difference would be considered for gathering new samples.

Spatial Hashing:

Creation of a photon map data structure is one of the most significant tasks in photon mapping. For performance reasons this data structure is usually a kd-tree. [Bentley 1975], which helps in speeding up the photon search. However, the process of constructing the tree is quite difficult to be done in parallel since all the threads have to access the same data-structure. It puts a lot of synchronization overhead and defeats the main purpose of parallel processing. Davis et.al. discuss parallelization of photon search in [Davis et al. 2000b], but it is not one of the best solutions to the overall problem.

Tree construction and traversing has a significant problem which is it highly irregular memory access pattern. To find photons in a kD-Tree, lots of scattered memory reads have to be performed, which is not at all optimal. The interdependent operations that have to be performed during tree traversal entail that the hardware is not able to hide these memory latencies.

In his masters thesis Martin Fleisz describes a new photon mapping technique for GPUs. This approach is based on spatial hashing for organizing the photons in the photon-map. It is claimed that this technique uses the parallel computation power more efficiently than the other techniques. So, the main goal would be to use as many threads as possible for the photon map creation and photon search, while trying to avoid any synchronization and having minimum scattered memory access.

The Spatial hashing approach is based on the CUDA particles paper [Green 2008]. It uses a hash table, that allows us to look up a set of potential neighbor photons in O(1) time and which can be easily created and accessed in parallel. Each entry in the hash table references a spatial cell in the scene (# 1 to 16 in fig 1), containing photons. So, to find the right cell, we have to calculate the hash value for the sample point and locate the right cell using the hash table.

Since the sample point can be located close to an edge of the cell, we also need to process photons from the neighboring cells in all

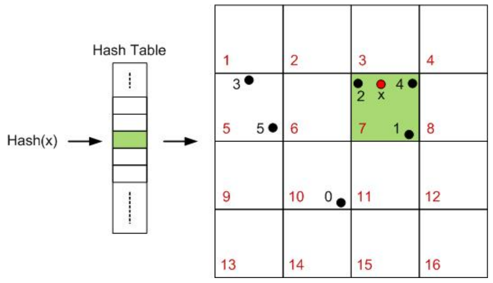
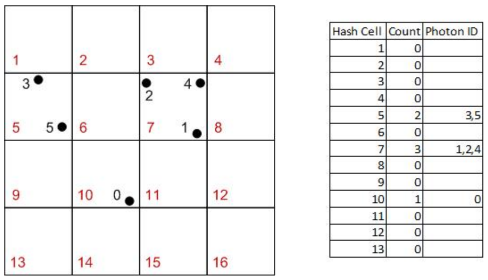


Figure 3: *Immediate lookup of photons in the same cell as the sample point x using hash table*

three dimensions. Finally we collect the closest k photons, using a sorted list that we implement using the fast on-chip shared memory. Photon collection has a time complexity of O(n), where n is the average number of photons in a cell and its neighbors. Photon maps would rarely contain more than a million photons, so there is no problem with the time complexity due to extremely large photon maps. Moreover, the number of photons in a cell grows only at a fraction of the actual photon map size since photons will be distributed over many cells.

The algorithm has several kernels executed after each other. The first kernel would calculate the hash value for each photon in the photon map and store the resulting values in an array along with the associated photon index. The next step is to sort the array based on the hash values, suing radix sort from [Grand 2007]. The final step would be to find start indexes for each entry in the sorted array to create the hash table.



Hash Cell	Count	Photon ID
1	0	
2	0	
3	0	
4	0	
5	2	3,5
6	0	
7	3	1,2,4
8	0	
9	0	
10	1	0
11	0	
12	0	
13	0	

Figure 4: *Photon hashes are calculated from their photons(left), final hash table(right)*

Since the sorted list contains only a reference ID to the actual photon data in the photon table, it would be an overhead. To get rid of it, we reorder the photons data according to their position in the sorted array. Now, simply by looking up its index in the hash table, we can access the data of the first photon in a cell directly and the other photons can be easily iterated in sequence. The array of the reordered photon data is finally bound to texture memory. Texture lookups are cached and the sorted sequential order will improve the coherence when accessing the photons during the photon search stage. Since all hashing functions require a grid position as input, the grid position p for a photon with position p can be calculated from:

$$p' = \lfloor p \cdot \vec{s} \rfloor \tag{1}$$

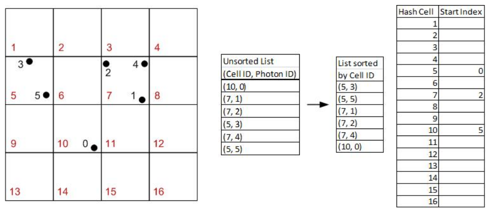
Where the scaling vector $\vec{s}$ specifies the scaling factors for a cell in each dimension. p can now be used to calculate value for our new photon position. Green suggests two different hash functions in his paper. The first one calculates the linear cell id for the given grid position p using the following equation:

$$f_{Hash}(p) = pz \cdot gridSizeY \cdot gridSizeX + py \cdot gridSizeX + px \tag{2}$$

The $gridSize$ factors in the formula specify the number of grid cells along x, y and z axes. Alternatively, Green suggests using hash function, based on the Z-order curve [Morton 1966] to improve coherence of memory access. We can use the sampling points coordinate to calculate its grid position and hash value, in order to find the cell it is located in. All we have to do is to collect the closest k photons from all neighboring cells and we are able to calculate our radiance estimation.

Spatial Hashing implementation

The first step would be to specify the world origin, which would be the minimum coordinate values for the x, y and z dimension, stored in our photon-map, since we support photons with negative coordinate values. For the grid position calculation we add the world coordinates to the input position to transform all photon positions into positive coordinate system. We use the maximum photon query radius for the grid cell size therefore, our scaling vector s-¿ is (1/rmax 1/rmax 1/rmax). Finally, we calculate the number of grid cells along each axis, using the cell size and the bounding box extents of the photon map. To ease the handling of border cells, we add two grid cells along each axis, allowing us to avoid any expensive checks, otherwise required for clamping grid positions. The last step during the initialization phase is the creation of the neighbor offset lookup table. The table is used to calculate the neighbor cells grid positions during the photon search. Fig 3.



Unsorted List (Cell ID, Photon ID)
(10, 0)
(7, 1)
(7, 2)
(5, 3)
(7, 4)
(5, 5)

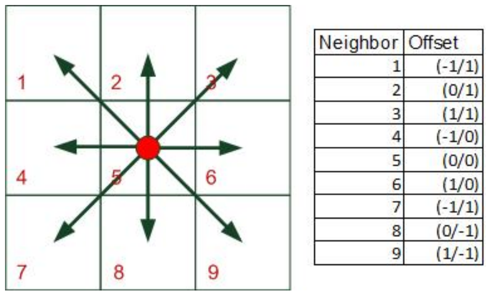
List sorted by Cell ID
(5, 3)
(5, 5)
(7, 1)
(7, 2)
(7, 4)
(10, 0)

Hash Cell	Start Index
1	
2	
3	
4	
5	0
6	
7	2
8	
9	
10	5
11	
12	
13	
14	
15	
16	

Figure 5: *After calculating the hashes the photon list is sorted and the hash table is created*

Then we execute the first kernel which creates the unsorted hash value array, where each entry also contains hashed photons index. After sorting the array based on the photon hashes, using the radix sort algorithm, we then have to determine the start photon index for each grid cell, to create the final hash table. This is done by executing a kernel function for each entry in the sorted array that checks, if the previous photons hash value is different from its own. If it is, we know that the current entry marks the start of a cell and the previous entry marks the end of another cell, which can be efficiently exchanged using the GPUs on chip shared memory. After the resulting indexes have been written to our hash table we reorder the photons to improve memory access coherence and finally we have a hash table that can be indexed using a points hash value and a sorted photon table.

To find the required photons, we execute our parallel search kernel for each point in query set. First, calculating the query points grid position and initializing our photon list to empty and then iterate through the 27 neighbor cells of interest, using the offsets from our pre-computed neighbor offset table. We simply add the values from the table to the current grid position and use that new grid position for the hash calculation. We use a simple sorted list approach for collecting photons. As long as the list is empty, we just keep adding photons until it is full. If we find a closer photon and the list is already full, then we insert that photon to its sorted position in the list



Neighbor	Offset
1	(-1/1)
2	(0/1)
3	(1/1)
4	(-1/0)
5	(0/0)
6	(1/0)
7	(-1/1)
8	(0/-1)
9	(1/-1)

Figure 6: *Calculating neighbor cells' grid positions using the values from the offset lookup table*

and shift all the following photons back, losing the last item. A heap data-structure might give a better performance for such storage. After the kernel finishes the processing of all the cells, the remaining k photons in the shared memory list are written to the result array, stored in global memory. Spatial Hashing Images courtesy: [Fleisz 2009] Photon-Pipeline Image courtesy: [Neumann 2010]

4 Results and Comparisions

In order to validate and compare our technique we use a few measurements as explained below. We use Spatial Hashing only for the construction and retrieval of photon map. At the other places requiring an acceleration structure we use a Kd-Tree.

Comparision: Construction Time Performance

Benchmark Speedup: Kd-Tree Construction for SimpleDragon scene
Warmup Runs: 10
Runs: 200
Total Time: 38561.992 ms
Average Time: 192.810 ms for each run

Benchmark Speedup: Spatial Hashing for SimpleDragon scene
Warmup Runs: 10
Runs: 200
Total Time: 35702.654 ms
Average Time: 178.513 ms for each run

As seen in the comparision above, the spatial hashing technique is much faster than the Kd-Tree approach. It is seen that the Kd-Tree construction cost increases with the amount of photons in the photon map, whereas the hash-table creation takes almost constant time. This is because the hash table creation is fully parallelized, beginning with the hash calculaiton for each photon and ending with the hash table creation and the re-ordering of photons. In contrast the Kd-Tree technique iteratively executes several kernels for scan and split operations at each tree level. Excessive kernel invocation performs implicit synchronization, since a GPU can run only one kernel function at a time.

Comparision: Frame Rate

We compare the overall frame-rates of different scenes achieved for rendering the complete scene by both the techniques

It is clearly seen above that the average frame rate of Spatial Hashing is higher than that achieved by Kd-Tree technique, for the same reasons mentioned above.

Comparision: Ray Tracing rates

Now we compare the time for ray tracing taken by CUDA and OptiX engines.

Table 1: *Comparing frame-rates from different data-structures*

Scene	Data-Structure	Avg Rate	Range
Cardioid	Kd-Tree	1.2 fps	1.1 to 1.4 fps
Cardioid	Spatial Hash	0.7 fps	0.4 to 0.9 fps
Ring	Kd-Tree	0.2 fps	0.2 fps
Ring	Spatial Hash	0.6 fps	0.6 to 0.8 fps
Room	Kd-Tree	0.2 fps	0.2 fps
Room	Spatial Hash	1.2 fps	1.0 tp 1.2 fps
Simple Box	Kd-Tree	0.3 fps	0.3 fps
Simple Box	Spatial Hash	1.4 fps	1.0 to 2.1 fps
Simple Dragon	Kd-Tree	0.1 fps	0.1 fps
Simple Dragon	Spatial Hash	0.6 fps	0.6 to 0.7 fps
Sponza	Kd-Tree	0.2 fps	0.2 fps
Sponza	Spatial Hash	0.7 fps	0.6 to 0.9 fps
Sibenik	Kd-Tree	0.1 fps	0.1 to 0.2 fps
Sibenik	Spatial Hash	0.8 fps	0.5 to 0.9 fps

Table 2: *Comparing time taken for ray tracing*

Scene	Ray Tracing Engine	Trace Time
Cardioid	CUDA	49 ms
Cardioid	OptiX	12 ms
Sponza	CUDA	119 ms
Sponza	OptiX	15 ms
Sibenik	CUDA	132 ms
Sibenik	OptiX	18 ms

The OptiX engine focuses exclusively on the fundamental computations required for ray tracing only and avoids embedding rendering specific constructs, and demonstrates that most ray tracing algorithms can be implemented using a small set of light-weight programmable operations. It provides a very simple and abstract ray tracing execution model with the most accustomed execution mechanisms and a domain specific compiler. This is the reason it gives much higher ray tracing speeds than a normal cuda ray tracer.

5 Conclusion

The goal of the project was to come up with an implementation of Photon mapping which would enable real-time rendering. We began looking at exisitng papers published for photon mapping and how they exploited parallelism. After analyzing the strengths and weaknesses of various techniques, we concluded that using a hybrid approach would avoid the previously discovered shortcomings. We discovered that OptiX is a recent advancement in the field of Ray-tracing, and it would be beneficial to put its potential to our use. Our results concluded that using OptiX, would speed up the process of ray-tracing and hence the photon mapping, by leaps.

Another optimization that we delved into was for the creation and retrieval of photon map data. We observed that if a Spatial Hashing technique was used for creating hash-tables to store photon maps and retrieve them, it gives a considerable amount of speedup. The construction time was observed to be greatly reduced compared to the kD-Tree solution. The Spatial Hashing technique works well with both the global as well as caustic photon maps.

We display in our results, the summary of our observations and proving that if the mentioned approach was used for photon mapping, it is possible to achieve very high frame-rates.

6 Future Work

Due to a limited time frame, it has not been possible to incorporate the entire approach in a single project. Due to the incompatibility of OptiX API with the CUDA API, the desired goal was not achieved. It would be interesting to know the actual frame-rates this hybrid approach can achieve.

There is a lot of scope in Photon Mapping and new implementations, acceleration structures, APIs, ray tracing engines etc. would definitely benefit it.

Future developments in graphics hardware will also have an important impact on Photon mapping techniques.

Acknowledgements

I am grateful to Martin Fleisz and Mathias Newmann for their small but significant inputs on how to improve their source codes. Special thanks to Dr. Mark Olano, for his support, cooperation and help and especially for giving me the access to the lab and its resources. Last but not the least the people in the VANGOOGH lab, who directly and indirectly helped me to reach this stage, which I might not have been able to reach, without their help and guidance.

References

APPEL, A. 1968. Some techniques for shading machine rendering of solids. *AFIPS Joint Computer Conferences, Atlantic City, New Jersey*, 37–45.

BENTLEY, J. L. 1975. Multidimensional binary search trees used for associative searching. *Comm. of the ACM 18*, 509–517.

CHRISTENSEN, P. H. 2000. Faster photon map global illumination. *Journal of Graphics Tools, ACM-1999 4(3)*, 1–10.

DAVIS, T., CHALMERS, A., AND JENSEN, H., 2000. Practical parallel processing for realistic rendering. SIGGRAPH 2000 Course Notes.

DAVIS, T., CHALMERS, A., AND JENSEN, H. W. 2000. Practical parallel processing for realistic rendering. *SIGGRAPH 2000 Course Notes, New Orleans*.

FABIANOWSKI, B., AND DINGLIANA, J. 2009. Interactive global photon mapping. *Comput. Graph. Forum 28*, 4, 1151–1159.

FABIANOWSKI, B., AND DINGLIANA, J. 2009. Compact bvh storage for ray tracing and photon mapping. *GV2 Group, Trinity College, Dublin, Ireland*.

FLEISZ, M. 2009. *Photon Mapping on the GPU*. Master's thesis, University of Edinburgh.

GRAND, S. L. 2007. Broad-phase collision detection with cuda. *GPU Gems 3*, 697–721.

GRAUER-GRAY, S. 2008. Photon mapping on the gpu. *University of Delaware*.

GREEN, S. 2008. Particle based fluid simulation. In *Game Developers Conference*, Book.

HAVRAN, V., HERZOG, R., AND SEIDEL, H.-P. 2005. Fast final gathering via reverse photon mapping. *Eurographics 2005 24*, 3, 323–333.

HENRIK WANN JENSEN, N. J. C., 2000. A practical guide to global illumination using photon maps. Siggraph 2000 Course 8, July.

JENSEN, H. W., AND CHRISTENSEN, N. J. 1995. Photon maps in bidirectional monte carlo ray tracing of complex objects. *Computers and Graphics 19*, 215–224.

JENSEN, H. W. 1996. Global illumination using photon maps. In *In Rendering Techniques 96 (Proceedings of the 7th Eurographics Workshop on Rendering*, vol. 0, Eurographics, 21–30.

JENSEN, H. W. 2009. *Realistic Image Synthesis Using Photon Mapping*. A. K. Peters, Ltd., Natick, MA, USA.

KAJIYA, J. T. 1986. The rendering equation. *Computer Graphics . ACM SIGGRAPH 20*, 143–150.

KELLER, A. 1997. Instant radiosity. *Computer Graphics . ACM SIGGRAPH*, 49–56.

LAFORTUNE, E., AND WILLEMS, Y. 1993. Bidirectional path tracing. *3rd International Conference on Computer Graphics and Visualization techniques (Compugraphics)*, 145–153.

MCGUIRE, M., AND LUEBKE, D. 2009. Hardware-accelerated global illumination by image space photon mapping. *ACM SIGGRAPH/EuroGraphics High Performance Graphics 2009*, 77–89.

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

MORTON, G. M. 1966. A computer oriented geodetic database; and a new technique in file sequencing. *IBM Ltd, Ottawa, Technical Report*.

NEUMANN, M. 2010. *GPU-basierte globale Beleuchtung mit CUDA in Echtzeit*. Master's thesis, FernUniversitat in Hagen.

NVIDIA CORPORATION. 2009. Nvidia cuda programming guide version 2.2. Tech. rep., NVIDIA.

PARKER, S. G., BIGLER, J., DIETRICH, A., FRIEDRICH, H., HOBEROCK, J., LUEBKE, D., MCALLISTER, D., MCGUIRE, M., MORLEY, K., ROBISON, A., AND STICH, M. 2010. Optix: A general purpose ray tracing engine. *ACM Transactions on Graphics* (August).

PETER, I., PIETREK, G., AND INFORMATIK, F. 1998. Importance driven construction of photon maps. *Rendering Techniques 98 (Proceedings of the 9th Eurographics Workshop on Rendering)*, 269–280.

PURCELL, T. J., DONNER, C., CAMMARANO, M., JENSEN, H. W., AND HANRAHAN, P. 2003. Photon mapping on programmable graphics hardware. In *In HWWS 03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, vol. 0, Eurographics, 41–50.

RUSHMEIR, H. E., AND TORRANCE, K. E. 1990. Extending the radiosity methods to include specularity reflecting and translucent materials. *ACM Transactions on Graphics 9*, 1–27.

SHAWN SINGH, P. F. 2007. The photon pipeline revisited. *The Visual Computer 23*, 7, 479–492.

SILLION, F. X., ARVO, J. R., WESTIN, S. H., AND GREENBERG, D. P. 1991. A global illumination solution for general reflectance distributions. *Computer Graphics 25*, 187–196.

SINGH, S. 2006. The photon pipeline. In *GRAPHITE*, vol. 0, Book, 333–340.

STEINHURST, J., COOMBE, G., AND LASTRA, A. 2008. Reducing photon-mapping bandwidth by query reordering. *IEEE Trans. Vis. Comput. Graph. 14*, 1, 13–24.

SUYKENS, F., AND WILLEMS, Y. D. 2000. Density control for photon maps. *Rendering Techniques 2000: 11th Eurographics Workshop on Rendering*, 23–34.

TRUONG, D. N., BODIN, F., AND SEZNEC, A. 1998. Improving cache behavior of dynamically allocated data structures. *International Conference on Parallel Architectures and Compilation Techniques*, 322–329.

WALLACE, J. R., COHEN, M. F., AND GREENBERG, D. P. 1987. A two pass solution to the rendering equation: A synthesis of ray tracing and radiosity methods. *Computer Graphics 21*, 311–320.

WANG, R., ZHOU, K., PAN, M., AND BAO, H. 2009. An efficient gpu-based approach for interactive global illumination. *ACM SIGGRAPH*, 1–8.