Photon Mapping On The GPU

Scott Grauer-Gray
University of Delaware
sgrauerg@gmail.com

Abstract

This paper discusses photon mapping and explores various issues of implementing the algorithm using the parallel processing power of the GPU via CUDA. The first implementation is a straight-forward, naive implementation of photon mapping using a simple scene consisting of only Lambertian surfaces. Then, the implementation is modified to address various shortcomings of the naive implementation, to allow the surfaces to exhibit the Phong reflection model, and to decrease noise. The results are promising; it is clear that there is potential for photon mapping on the GPU.

1. Photon Mapping

Photon mapping is a popular technique in computer graphics that is used to simulate the illumination objects in a scene. The technique was developed by Henrik Wann Jensen in 1996 and marked an improvement from the then-popular ray tracing/radiosity techniques in many situations[2]. The key feature of photon mapping is the separation of the photon map from the geometric representation of the scene, making it possible to simulate illumination in complex scenes.

Specifically, photon mapping can be viewed as a two-pass algorithm. In the first pass, photons are emitted from random points in each light source and sent into the scene in random directions. When a photon intersects with a surface in the scene, the photon is either absorbed by the surface or reflected and continues to potentially hit more surfaces.

The fate of the photon at the photon-surface intersection is determined using Russian Roulette, a Monte Carlo technique that uses random sampling. First, a random decimal value between 0 and 1 is generated. Then, if the decimal value is below a certain reflectance REFL for the surface, the photon is reflected, and the location of the photon-surface intersection is stored in the photon map to use in the second pass. Otherwise, the photon is absorbed.

In the second pass of the algorithm, each point on each surface of the scene is rendered using the photon map. The more photons on the surface that are within a radius RAD of the point, the greater the illumination of the point.

It must be noted that there are many possible variations to the general algorithm; the algorithm as described in this paper is the version that is implemented on the GPU using CUDA. In addition, the illumination of a surface point is often divided into four parts: direct illumination, specular illumination, caustics on diffuse/slightly glossy surfaces, and soft indirect illumination. Photon mapping is a popular technique for caustics and soft indirect illumination, but ray tracing is often the choice method for direct illumination and specular illumination since 'too many' photons may be required for sufficient illumination accuracy. However, in the implementation described in this paper, photon mapping is the illumination method used for every source of illumination.

2. Photon Mapping on the GPU using CUDA

2.1. Previous Work

There is no scholarly literature describing a general CUDA implementation of photon mapping. In [4], Purcell describes a GPU implementation of photon mapping using shaders and the graphics API. Then in [8], Zhou presents photon mapping as an application of his CUDA real-time KD-tree construction implementation, but he limits photon mapping to caustics and not to other types of illumination. Finally, Singh and Steinhurst both discuss photon mapping on simulated GPU-like architecture in [5] and [6], respectively.

2.2. Random Values on the GPU

One issue regarding photon mapping on the GPU with CUDA is the parallel generation of random values. Multiple steps of the photon mapping algorithm require the generation of random values, and there is no built-in implementation of random value generation on the GPU. One option is to sequentially generate all random values on the CPU and pass them to the GPU. Fortunately, this is not necessary; implementations of random number generation using CUDA on the GPU are available. One implementation uses the Mersenne twister algorithm [3] and another uses the rand48 algorithm [7] to generate random values in parallel. The rand48 implementation is used in the GPU implementation described in this paper. It is worth noting that some initialization steps, including the generation of the initial set of random values and of the rand48 algorithm parameters, must be performed on the CPU in order to begin GPU random number generation.

2.3. Naive Implementation

Naively, photon mapping easily maps to the SIMD model favored by CUDA and the GPU[1]. First, photons are emitted from random locations in random directions from the light source(s) in the scene. Then, the photon-surface intersections are calculated and stored for all photons in parallel. In both steps, each photon is mapped to a different thread; calculations for all photons can be performed in parallel. Then, in the rendering pass, and the number of photons within RAD of each point on the current surface is retrieved in order to determine the illumination of each point in a 64 X 64 mesh on each surface, and the illuminated scene is displayed by taking advantage of the interoperability between CUDA and OpenGL. Here, the process works in parallel by mapping each surface point to a single thread when determining surface illumination. The operations in each pass require no data sharing across threads in this model; the algorithms can be viewed as 'embarrassingly parallel'.

Unfortunately, there are a couple of shortcomings of this algorithm:

1. As the first pass progresses, more and more photons as absorbed by surfaces in the scene and the threads that these photons map to have 'nothing' to do while the few threads mapped to photons still reflecting off surface(s) still have work to do. It is not possible to 'only' run the CUDA kernel on the threads mapped to 'still-active' photons in this implementation, so the kernel must continue to call every thread. As a result, cycles are wasted on threads with nothing to do.

2. To prevent write conflicts, the photon-surface intersection is stored at index (numBounce * TOTAL_NUM_PHOTONS + photonNum) in the implementation, requiring the allocation of (NUM_POSSIBLE_BOUNCES * TOTAL_NUM_PHOTONS) slots in the photon map. However, most of the photons will be absorbed by a surface before NUM_POSSIBLE_BOUNCES bounces, wasting space in GPU memory since storage must be allocated to store photon-surface intersections that ultimately do not happen.

In particular, the second shortcoming limits the number of possible bounces of each photon in this implementation due to the limited memory on many GPUs. As a result, the 'naive' algorithm's capability to simulate soft indirect illumination is greatly weakened since indirect illumination comes as a result of multiple photon bounces.

3. GPU Photon Mapping Adjustments

Fortunately, the GPU photon mapping implementation can be adjusted to overcome these shortcomings. In particular, an implementation that takes advantage of the atomic operations available on CUDA removes both the wasted cycles and the wasted space present in the naive implementation.

In the adjusted implementation, the incoming photons and outgoing photons are in separate arrays. In the first step, the currentNumPhotonsInPhotonMap is initialized to 0 and photons are emitted from the light source(s) as in the naive implementation. Then, the process continues in a loop from 0 to NUM_POSSIBLE_BOUNCES with for every currently active photon in parallel:

1. Set currentNumberOutputPhotons to 0.

2. Calculate the photon-surface intersection for each photon in the inputPhotons array.

3. Determine whether the photon reflects off the surface using Russian Roulette.

4. If the photon reflects:

   (a) Store the photon in the outputPhotons array at index currentNumberOutputPhotons and then increment the global value currentNumberOutputPhotons using the atomic operation.

(b) Store the location of the photon-surface intersection at index currentNumPhotonsInPhotonMap in the photonMap array and increment the global value currentNumPhotonsInPhotonMap using the atomic operation.

5. Loop back to 1, swapping the inputPhotons and outputPhotons arrays since the output photons of the current bounce are the input photons of the next bounce.

The first difference between this implementation and the previous one is the presence of an output photon array in the kernel. For each 'bounce', the array contains currentNumberOutputPhotons in which every slot in the array from 0 to currentNumberOutputPhotons is filled with a currently active photon; the photons absorbed in the current bounce are not there. It is no longer necessary to allocate a thread for every photon for NUM_POSSIBLE_BOUNCES bounces; the programmer only needs to allocate threads for the photons that have not yet been absorbed.

The second implementation adjustment is that each photon-surface intersection is now stored at index currentNumPhotonsInPhotonMap; currentNumPhotonsInPhotonMap is a global value that is initialized at 0 and then incremented using the atomic operation with each photon-surface reflection. This algorithm modification greatly reduces the space necessary to allocate for the photonMap array, particularly when NUM_POSSIBLE_BOUNCES is set to a high value.

Unfortunately, the use of the atomic operations does take away some of the 'parallelism' of the implementation, and atomic operations are only available in nVidia GPUs with compute capability 1.1 or higher[1].

4. Results

4.1. Scene

A simple scene consisting of an 'open box' with two lights is constructed. The coordinates of the box extend from -.25 to .25 in the x, y, and z directions. The box is open in one direction and is shown from multiple angles in figure 1. Circular lights are present in the center of two inner 'surfaces/walls' of the box. The illumination from the lights is uniform from all locations and in all directions. Each surface/wall in the scene reflects 75% of the photons that hit it, and each wall uses the Lambertian reflectance model where the angle of the viewer and the photon reflectance is insignificant.

4.2. Experiments

The experiments described were all performed on a laptop with a Intel Core 2 Duo CPU running at 2.00 GHz and a nVidia 8600M GT GPU with two multiprocessors.

One issue present in GPU programming is limited memory, especially in a mid-level GPU such as the 8600M. Due to memory limitations, it is not possible to run the photon mapping implementations with much more than 5000 photons.

First, we ran the naive implementation described in 2.3 using 5000 total photons. 2500 photons are emitted from each light in the scene, and each photon is only allowed two bounces in the scene due to shortcoming 2. The illumination of each surface is retrieved via the use of a 64 X 64 mesh of points on the surface, where a photon-surface intersection contributes to the point illumination if it is within distance RAD = .05 of the point. A photon power magnitude that is equivalent to 90 or more photons within RAD of the point represents full illumination, and the point illumination drops with a linear pattern proportional to the drop in photon power magnitude until the photon power hits 0 magnitude and there is no illumination at the point. The results are shown in figure 1. The lights are present in the 'bottom' and 'left' walls in the figure. The results appear to be accurate in the sense that the lights illuminate the scene in the expected manner. In particular, the area of the walls that are directly lit are much brighter than the areas lit indirectly, but the entire scene is lit to some extent due to the presence of indirect illumination. However, a lot of noise is present, likely due to the limited number of photons.

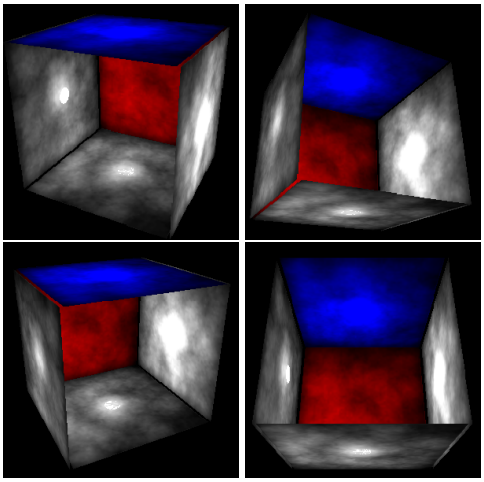

Figure 1. Photon mapping results.

Next, the modified implementation described in 2.3 is run. The modifications to the algorithm overcome shortcoming 2 of the naive implementation (see 2.3), so each photon is now allowed up to fifteen total bounces. All the other parameters remain the same as in 4.2. The results are similar to the previous results, but the areas more dependent on indirect illumination are better lit since each photon is allowed more bounces to illuminate the scene. A comparison of the results of the previous results allowing 2 bounces per photon and the current results allowing 15 photon bounces is shown in figure 2; the previous results are on the right and the current results are on the left.

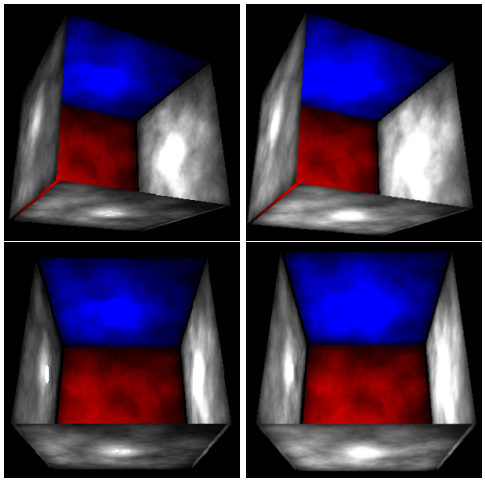

Figure 2. Photon mapping comparison between naive implementation allowing 2 bounces/photons (left) and modified implementation allowing 15 bounces/photon (right).

4.3. Running Times

In order to test whether the use of the GPU could potentially improve the efficiency of photon mapping, both the naive and modified GPU implementations of photon mapping are compared to a CPU implementation. The structure of the algorithm used in the CPU implementation is the same as the 'modified' GPU algorithm described in 2.3. The implementations are benchmarked on a laptop with a Intel Core 2 Duo CPU running at 2.00 GHz and a mid-level nVidia 8600M GT GPU with two multiprocessors. The first pass, the creation of the photon map, is timed separate from the rendering pass. Only two bounces per photon are allowed during this benchmarking in order to have a fair comparison across all implementations.

First, we benchmark the time it takes to emit the photons and generate the photon map. It takes 1.7 ms using the 'naive' GPU implementation described in 2.3, 11 ms using the 'modified' GPU implementation described in 2.3, and 15 ms using the CPU implementation. It is clear that the use of the atomic operations can slow down the generation of the photon map, so there are trade-offs between the different GPU implementations.

Next, the rendering pass is timed. The rendering pass in each implementation is performed at each point in a 64 X 64 mesh on each surface/wall by stepping through every photon-surface intersection stored in the photon map, incrementing the number of photons in the region if the intersection is on the given wall and within radius RAD of the point, and then dividing the total number of photons within RAD by FULL_LIGHT_POWER_NUM_PHOTONS to retrieve the illumination at the point. The process took 3000 ms using the naive GPU implementation, 2000 ms using the modified GPU implementation, and 8000 ms using the CPU. The modified GPU implementation is faster than the naive implementation because the points on the photon map are all 'packed together' from index 0 to the last photon in the photon map with no wasted searches on empty slots present in the naive implementation.

It is clear that the GPU implementation is faster than the CPU implementation in both photon mapping passes, showing the potential of the GPU to speed up the photon mapping process.

4.4. Photon Mapping Using Phong Reflection

Each implementation thus far uses the Lambertian reflectance model for each scene surface/wall; the reflection from incoming light on each surface is diffuse and viewing direction is insignificant to the illumination at each point. If a photon-surface intersection stored in the photon map is within RAD of a given point, the 'illumination power' of the point increases by 1 regardless of the angle of the photon reflection and the viewing direction. However, not all surfaces in the real world work this way; the viewing direction often matters in the illumination of a scene. To investigate photon mapping in these scenarios, the modified GPU implementation in 2.3 is farther adjusted to work for the Phong model of reflection. Using the Phong model, the power of reflection for each photon-surface intersection is equal to the dot product of the normalized viewing direction and the normalized photon reflection direction to the power of the shininess of the surface/wall. This calculation is performed during the generation of the photon map and

then stored; it is not necessary to store the reflectance direction in the photon map.

Results of this implementation are shown in 3; each surface in the left, middle, and right image have shininess parameters of 2.0, .5, and .01, respectively.

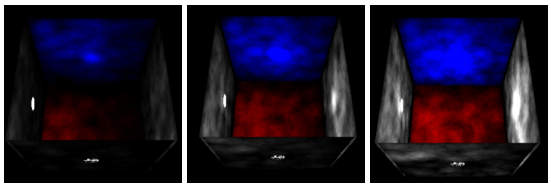

Figure 3. Photon mapping using the Phong model of reflectance with shininess parameters 2.0, .5, and .01 from left to right

4.5. Increasing number of Photons

Unfortunately, the limited storage available on the GPU limits the number of photons that can be processed at one time, since information about every photon must be in memory, and storage must be allocated for each photon-surface intersection. The limited quantity of photons results in an illumination pattern with significant noise, as shown in figures 1, 2, and 3.

To remedy this issue, an array of float values containing the current photon power of each point on the 64 X 64 mesh of each wall is constructed and stored on the GPU. Then, the photon mapping implementation is performed in multiple 'rounds'. At the end of each 'round', the current photon power at each point on each surface/wall is updated using the current set of photons in storage. Now, there is no limitation to the number of photons used to illuminate the scene.

The experiments described in this section are performed with 5000 photons emitted in each 'round' and with each photon allowed 15 bounces. A photon-surface intersection contributes to the illumination of a point on the surface if it is within distance RAD = .025 of the given point, and a point is considered fully illuminated if the total photon power at the point is at least 10 times the number of 'rounds' of photon mapping. The shininess parameter is set to 5.0 for each wall. The results of the implementation are shown in figure 4 for 1, 5, 10, 25, 50, 100 rounds of photon mapping, where the total number of photons in each experiment is equal to 5000 photons/round * number of rounds. It is clear that the amount of noise in the illumination decreases as the number of photons increases. However, the running time does increase with the number of photons, so there is a trade-off. The total running time of the GPU implementation is shown for differing number of photons in table 1.

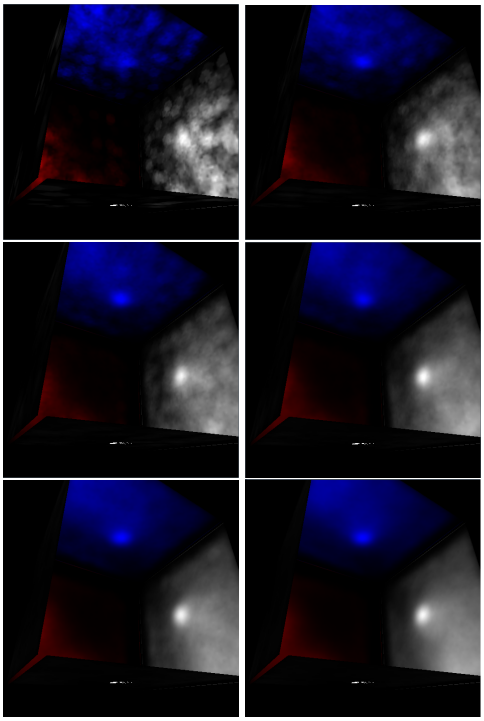

Figure 4. Photon mapping using 1, 5, 10, 25, 50, and 100 rounds of photon mapping with 5000 photons in each round (arranged from top left to bottom right going across then down).

4.6. Other Considerations

The scene used for the experiments described in this paper is relatively simple. In more complex scenes with more points, it may not be practical to store the current illumination of every point on every surface in the GPU or even in the RAM. It might be necessary to store the entire photon map in the GPU or RAM or even in a file in order to properly render the scene. In addition, it may be desirable to structure the photon map as a KD-tree or related structure for improved retrieval of the desired photons for each point. Finally, using photon mapping in combination with another technique such as ray tracing for certain forms of illumination may give improve results with little penalty in processing time.

Number Of Photons	Total GPU time
5000	2130
25000	10670
50000	20850
125000	53280
250000	105640

Table 1. Time in ms of the entire photon mapping implementation on the GPU with differing numbers of photons.

5. Conclusions

The results in this paper are promising and show there is potential to using the GPU and CUDA as a way to decrease the running time of the photon mapping algorithm; both CUDA implementations achieved the same results more quickly than the corresponding CPU implementation. While other considerations should be taken into account when considering photon mapping as the choice technique to retrieve and render scene illumination, a GPU implementation should be considered in spite of possible challenges.

References

[1] *NVIDIA Corporation: NVIDIA CUDA compute unified device architecture programming guide*. NVIDIA Corporation, Jan 2007.

[2] H. W. Jensen. Global illumination using photon maps. pages 21–30. Springer-Verlag, 1996.

[3] V. Podlozhnyuk. Parallel mersenne twister. June.

[4] T. J. Purcell, C. Donner, M. Cammarano, H. W. Jensen, and P. Hanrahan. Photon mapping on programmable graphics hardware. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, page 258, New York, NY, USA, 2005. ACM.

[5] S. Singh and P. Faloutsos. The photon pipeline revisited: A hardware architecture to accelerate photon mapping. *Vis. Comput.*, 23(7):479–492, 2007.

[6] J. Steinhurst, G. Coombe, and A. Lastra. Reordering for cache conscious photon mapping. In *GI '05: Proceedings of Graphics Interface 2005*, pages 97–104, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 2005. Canadian Human-Computer Communications Society.

[7] J. A. van Meel, A. Arnold, D. Frenkel, S. F. P. Zwart, and R. G. Belleman. Harvesting graphics power for md simulations. *Molecular Simulation*, 34:259, 2008.

[8] K. Zhou, H. Q., R. Wang, and G. B. Real-time kd-tree construction on graphics hardware. In *Microsoft Technical Report*, 2008.