Proceedings of the Fifth IIEEJ International Workshop
on Image Electronics and Visual Computing 2017
Da Nang, Vietnam, February 28- March 3, 2017

EFFICIENT ADAPTIVE GPU PATH TRACING

Namo Podee†
†Hokkaido University

Kei Iwasaki††
††Wakayama University

Yoshinori Dobashi†
†Hokkaido University

Tsuyoshi Yamamoto†
†Hokkaido University

ABSTRACT

We present an adaptive technique for efficient path tracing on a GPU. The technique improves the efficiency of the current state of the art parallel path tracing methods. Our method uses a stream compaction algorithm to generate, in parallel, a list of pixels to be traced, also called a sample stream, which may contain multiple samples for each pixel. To accelerate the convergence, we choose pixels to be traced by predicting the square error reduction rate, which is computed by comparing the past path tracing result and its filtered version with a bilateral filter. Then, we use traditional stream compaction path tracing for the generated sample stream and accumulate the result iteratively, in parallel. We show that our method is up to 2.6 times faster compared to previous parallel path tracing techniques for equal-quality rendering. We also analyze how much improvement has been achieved in different scenes and discuss the limitations of our method.

1. INTRODUCTION

Light transport problems used to be practically impossible to solve interactively because of their complexity. However, the advances made in general purpose graphics processing units (GPGPU), now means these problems can be solved interactively by using a massive amount of threads on a GPU.

While a naive GPU implementation can speed up the path tracing algorithm, this does not take full advantage of the computing power of the GPU due to the uneven workload between threads. To solve this, several methods [2] [7] [12] [11] that make better use of the parallel computing capability have been implemented. However, little research has been done on efficient adaptive path tracing on a GPU.

In this paper, we propose a solution to this problem; we propose adaptive path tracing on a GPU with minimal synchronization. Our method spends the computing power on the erroneous pixels, instead of the whole image, and achieves improvements in the overall path tracing performance. From our experiments, we conclude that our method can increase the computation speed by up to a factor of 2.6 compared to previous parallel path tracing methods with equal-quality rendering.

Our main contributions are:

- We propose a path tracing method that can adaptively sample pixels on a GPU and accumulate the results into an image buffer without any conflicts in memory access.
- We demonstrate that our method has higher performance than the previous GPU path tracing techniques.

2. RELATED WORK

Our work is split into two main parts: one part is concerned with the path tracing algorithm and the other with error prediction by filtering the result. Therefore, we split our discussion on the related work into two parts.

Firstly, there has been little work reported on efficient thread management for GPU path tracing, which we use and improve in our research. The Russian roulette technique is usually used in path tracing [4][5] to probabilistically terminate the path tracing process. However, if we use this algorithm on multiple threads, it will cause an unbalanced workload due to the unpredictable termination of the algorithm on each thread. Novák et al. [7] proposed a path regeneration method to solve this problem. This method regenerates paths from the originating pixel for each terminated path. Since then, Wald [12] has presented an active thread compaction method to speed up the path tracing process on a GPU. He used stream compaction [1] to remove terminated paths from a path list and let the threads efficiently process these paths. Antwerpen [11] proposed an even more efficient method by combining active thread compaction and path regeneration techniques, which improves the SIMD efficiency of path tracing.

Secondly, Monte Carlo path tracing suffers from noise due to variance. One of the solutions to reduce noise is denoising by filtering, which is an active research field [13]. Rousselle et al. presented a denoising method that applies multiple filters to a noisy result then chooses one of the filters for each pixel to minimize the mean square error [8] .

They made further improvements by using non-local mean filtering [9] and using feature information, such as normal direction, texture and depth, to filter the input to produce multiple candidates [10]. Gastal et al. [3] presented a real-time high dimensional filtering technique by computing multiple manifolds of the result and denoising the input. Mehta et al. [6] also proposed an image-space filtering method for interreflections interactively by analyzing signals in the Fourier domain and using an axis-aligned filter. While our method builds on these methods, we focus on implementation of an efficient adaptive sampling technique on a GPU.

3. EFFICIENT ADAPTIVE PATHTRACING

A naive path tracer shoots rays over a whole image uniformly. However, in many cases, some pixels need more samples than others in order to converge faster to the correct result. For example, a pixel through which a tracing path hits a glossy reflective surface requires more paths than pixels corresponding to a highly reflective surface, which reflects an incident ray into a narrower path. Thus, there much research has been done on adaptive path tracing in which pixels to be sampled are chosen according to the estimated errors [13]. However, there has not been much work done in the area of thread management.

There are two main problems in adaptive path tracing on a GPU: thread-pixel assignment and the accumulation of results. The assignment problem is how to assign a pixel to each thread. It is complicated to efficiently assign pixels to each thread because we have to do everything in parallel, implying that we do not have synchronization between threads, since the parallel adaptive method usually processes one pixel on multiple threads; thus a naive adaptive implementation uses atomic operations to assign samples to threads. Next, the accumulation problem is in regard to the output from adaptive path tracing; it is most likely that we get multiple results for the same pixel because we try to trace erroneous pixels multiple times. With these results for the same pixel, we should not naively write the result to the image buffer simultaneously because a memory access conflict will occur and this slows down the computation.

We have developed a technique that can efficiently generate a list of pixels to be traced, which we call a sample stream. This sample stream is proportional to the estimated error reduction ratio for each pixel and the result can still be efficiently accumulated from a path tracer to the image buffer in parallel without causing conflict.

3.1. METHOD OVERVIEW

Figure 1 illustrates our method. Our method begins with creating an initial image by tracing a set of paths for each pixel. The number of paths is the same for all pixels. We use the light vertex cache bidirectional path tracing method developed by Davidovic et al. [2] because it is the most efficient GPU path tracing method we have tested. The image obtained by this process is then filtered with a bilateral filter. We calculate the difference between the unfiltered and filtered intensities of the pixels then use this to estimate the error reduction, which tells us how much the difference between the final result and the current result will be reduced by after some additional samples. After that, we use the error reduction data as a probability to determine whether additional paths from the viewpoint (eye paths) should be traced for each pixel or not. Next, we generate multiple sets of pixels to be traced iteratively based on the probability, which we will explain in detail in Section 3.3, and combine all these sets together and call this a sample stream. We then trace the eye paths for the pixels in the sample stream and accumulate the contributions from each set in the image buffer.

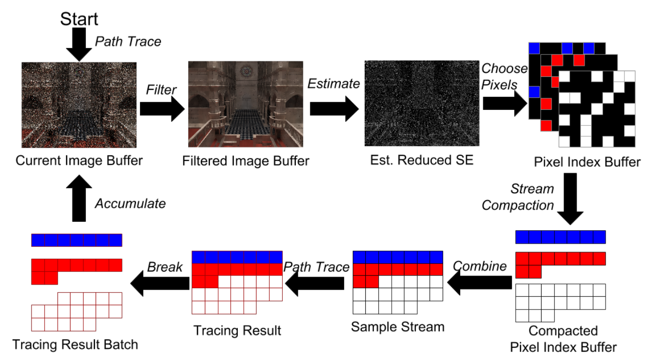

Figure 1: An overview of our method.

3.2. SQUARE ERROR REDUCTION ESTIMATION

We want to create an accurate image as fast as possible. In order to achieve this, we estimate the degree of error reduction that can be achieved by tracing additional paths for each pixel. We compute the square error (or SE) of the path tracing result in a similar way to Rousselle et al. [8] [9]. Initially, we render an image of a scene by tracing a small number of paths for each pixel, which would generally result in a very noisy image. Then we apply a high-dimensional bilateral filter to this noisy image. SE is then obtained by calculating the difference between the filtered and the unfiltered images. Next, we estimate a reduced SE that would be obtained by tracing n additional eye paths. Since SE is inversely proportional to the total number of eye paths traced, SE is reduced by a factor of n / $(n + n_s)$ where n is the number of additional eye paths and n_s is the total number of paths. The reduced SE can be estimated by:

$$E_{re} = E_{cur} \times \frac{n}{n + n_s}, \tag{1}$$

where E_{re} is the amount of error reduced by adding more n samples and E_{cur} is the current SE. We compute E_{re} for each pixel and use it to determine the number of additional eye paths to be traced, as described in the next section.

3.3. SAMPLE STREAM GENERATION

We use the estimated reduced error E_{re} to determine the number of additional eye paths to be traced through each pixel. We generate a list of pixels, which we call the sample stream, to indicate the pixels to be traced. We have developed a technique to efficiently create the sample stream on a GPU.

The basic idea is as follows. We first prepare a pixel index buffer that stores indices of pixels through which additional eye paths should be traced. The size of the pixel index buffer is the same as that of the image. Our method then compares E_{re} for each pixel with a random number. The range we set for this random number is discussed below. If E_{re} is larger than the random number, the pixel is identified as 'to-be-traced' (we call these *trace-pixels*) and its pixel index is stored in the corresponding element of the pixel index buffer. Otherwise, the pixel is identified as 'not-to-be-traced' (we call these *untrace-pixels*) and an invalid pixel number (-1) is stored in the pixel index buffer. This process is performed in parallel for each pixel. Then, using a stream compaction algorithm [1], our method creates the sample stream from the pixel index buffer by removing the invalid pixel numbers.

By repeating the above process, we can spend more computation power on the pixels with high errors since those pixels are more likely to be chosen as trace pixels. However, this method does not make the most of the parallel computing capability of the GPU unless we choose an appropriate range for the random numbers. One simple choice is to set the range from zero to the maximum of E_{re}. However, from our experiments, this does not work well; more than 98% of the pixels were identified as untrace-pixels. This means that the only a small number of eye paths are traced and many GPU threads are unused. To fully exploit the GPU capability, we use a binary search algorithm to determine the range of random numbers, as described below.

We initially set the search range to be from 0 to twice the maximum of E_{re}, and set the range of the random numbers to be from 0 to the middle value of the search range. A sample stream is then created by the method described above. Next, the search range is modified when the number of the trace pixels is not equal to the number of threads. When the number of trace pixels is smaller (or larger) than the number of threads, the maximum (or the minimum) of the search range is replaced by the middle value of the search range and the random number range is updated to be from 0 to the new middle value of the search range. Then, another sample stream is created. These processes are repeated and multiple sample streams are created until the number of trace pixels becomes the number of threads. Note that the pixels with higher errors appear multiple times in different sample streams and thus adaptive sampling of eye paths can be achieved.

3.4. SAMPLE RESULT ACCUMULATION

We now describe an efficient, conflict-free method for accumulating contributions of the eye paths generated for the trace pixels. Let us assume that the number of sample streams computed in the previous section is m. We first create a long sample stream by concatenating m sample streams. We apply path tracing in parallel to all the trace pixels listed in the concatenated sample stream. After tracing the paths, the contributions of the paths need to be accumulated in the image buffer. However, care must be taken in order to avoid a memory write conflict, which reduces the parallel computation performance. We address this by making use of the nature of the sample streams, that is, in each sample stream i, no trace pixels appear multiple times.

We prepare a temporal buffer whose size is the same as that of the concatenated sample stream. The contribution obtained by tracing an eye path for each trace pixel in the concatenated sample stream is stored in the corresponding entry of the temporal buffer. We then accumulate in parallel the contributions for the trace pixels listed in sample stream 1. Next, the contributions for the trace pixels in sample stream 2 are accumulated in parallel. These processes are repeated for $i = 0, 1, \cdots, m-1$. Since no trace pixels appear multiple times in the i-th sample stream, this process can be done in parallel without causing conflict.

4. RESULT

We implemented our method on a Nvidia CUDA 7.5. We initially sampled a scene with the stream compaction path tracing method with 5 samples per pixel, then filtered it with the bilateral filter that uses pixel positions, 3D positions, normal vectors and diffuse texture data. We tested our method on three scenes: The Cornell box with a ball, Sibenik cathedral and a Conference room. We used a machine with an Intel Core i7-4790K 4.00GHz, 32 GB of memory and a single Nvidia GeForce GTX TITAN X. We tested each scene with the output resolution at 800x600; the filter parameters were chosen experimentally for each scene.

Figure 2 shows the root mean square error (RMSE) of the result using our method, compared with the stream compaction method [1] and the adaptive atomic instruc-

Proceedings of the Fifth IIEEJ International Workshop
on Image Electronics and Visual Computing 2017
Da Nang, Vietnam, February 28- March 3, 2017

tion method, plotted against rendering time for three scenes. The adaptive atomic instruction method was implemented by initially sampling a scene, filtering it with the bilateral filter then calculating the square error value of each pixel just as in our method. However, instead of iteratively generating a path sample list, each pixel inserted its own pixel into a global sample stream by using atomic instructions on a GPU. The number of inserted samples of each pixel was determined according to this equation:

$$PixelTraceN(x) = TraceN * \frac{E_{pixel}(x)}{\sum E_{pixel}(x)}, \quad (2)$$

where x is a pixel, $TraceN$ is the thread size or the tracing budget and E_{pixel} is the predicted error for each pixel. After that we traced the global sample stream and simply accumulated the result via atomic instructions on a GPU. As demonstrated by Figure 2, the convergence speed using our method is faster than those using the other methods. For example, when compared to the methods using stream compaction and atomic instructions only, our method was up to 2.6 and 3.0 times faster, respectively. The method using atomic instruction underperformed when the system could not approximate the error of each pixel accurately enough. The method failed because it trusts the estimated error data and manages all of its threads according to this data unlike our method, which is more tolerant to an estimation failure by using probability to choose sample pixels. Due to the similarity of our method and the atomic instruction method, we also compared these and we show their performance breakdowns in Table 1. According to the table, the atomic instruction method is able to perform path tracing more than 2 times faster than our method. The only reason for this is that, in contrast to our method, generated sample streams from the atomic instruction method puts samples from the same pixel next to each other. This enables the GPU to perform caching efficiently and compute path tracing of this stream very quickly. We should be able to improve our method efficiency by adjusting the sample stream layout, but due to the time limitation, we plan to adjust it in future work. A visual comparison of the rendering result using our adaptive method with the results using the other methods is shown in Figure 3.

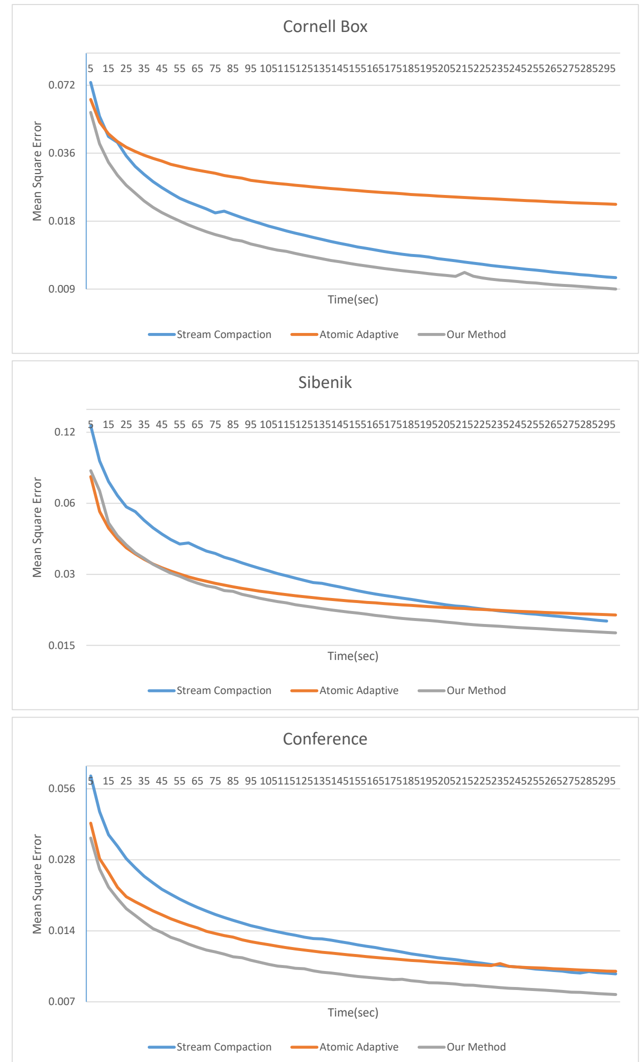

Figure 2: Convergence-time plot of the three scenes in logarithmic scale.

5. CONCLUSION

We have proposed an efficient adaptive path tracing method on a GPU. Our method outperforms the previous state of the art GPU path tracing techniques in that it is up to 2.6 times faster for equal-quality rendering. The outperformance continues as long as the filtered image can faithfully estimate the square errors of the result. However, due to the time limitation we cannot compare our method against other novel adaptive path tracing methods, which we plan to do in future work. Our filter may fail to faithfully estimate the square errors when we use parameters inappropriate for the scene to be rendered. Our filter requires many parameter adjustments, which are not intuitive and may be a tedious task. It is interesting to solve this problem by using a machine learning approach or more advanced filters. Our method can also be improved by adjusting the sample stream layout to be more cache friendly and thereby increase its performance.

6. ACKNOWLEDGEMENTS

Cornell Box model courtesy of Cornell University, Conference Room model courtesy of Anat Grynberg and

Table 1: Performance Breakdown

Scene	Method	Stage	Average Time(ms)
Conference	Our Method	Path Generation	15.9242
		Path Tracing	98.6969
		Result Accumulation	3.28532
	Atomic Method	Path Generation	33.9067
		Path Tracing	44.1767
		Result Accumulation	1.05989
Sibenik	Our Method	Path Generation	11.6567
		Path Tracing	175.384
		Result Accumulation	3.26131
	Atomic Method	Path Generation	12.9999
		Path Tracing	76.6641
		Result Accumulation	1.43767
Cornell Box	Our Method	Path Generation	21.0712
		Path Tracing	46.529
		Result Accumulation	3.15267
	Atomic Method	Path Generation	2.08057
		Path Tracing	8.22957
		Result Accumulation	0.736866

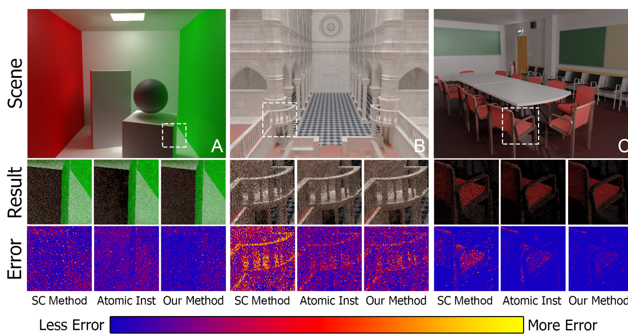

Figure 3: This figure shows a comparison between the results of our method and those of the other methods. The second and third rows show the results and square errors for 5 seconds of execution. Our method shows more accurate and noiseless results. Except for the B scene, the result from the atomic instruction method has less error than the result from our method. However, according to Figure 2, after 10 seconds, our method will outperform the atomic instruction method.

Greg Ward, Sibenik cathedral model courtesy of Marko Dabrovic. This work was supported by JSPS KAKENHI Grant Number JP15H05924.

7. REFERENCES

[1] Markus Billeter, Ola Olsson, and Ulf Assarsson. Efficient stream compaction on wide simd many-core architectures. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pages 159–166, New York, NY, USA, 2009. ACM.

[2] Tomáš Davidovič, Jaroslav Křivánek, Miloš Hašan, and Philipp Slusallek. Progressive light transport simulation on the gpu: Survey and improvements. *ACM Trans. Graph.*, 33(3):29:1–29:19, June 2014.

[3] Eduardo S. L. Gastal and Manuel M. Oliveira. Adaptive manifolds for real-time high-dimensional filtering. *ACM Trans. Graph.*, 31(4):33:1–33:13, July 2012.

[4] James T. Kajiya. The rendering equation. *SIGGRAPH Comput. Graph.*, 20(4):143–150, August 1986.

[5] Eric P. Lafortune and Yves D. Willems. Bi-directional path tracing. In *Proceedings of Compugraphics '93*, pages 145–153, 1993.

[6] Soham Uday Mehta, Brandon Wang, Ravi Ramamoorthi, and Fredo Durand. Axis-aligned filtering for interactive physically-based diffuse indirect lighting. *ACM Trans. Graph.*, 32(4):96:1–96:12, July 2013.

[7] Jan Novák, Vlastimil Havran, and Carsten Dachsbacher. Path Regeneration for Interactive Path Tracing. In H. P. A. Lensch and S. Seipel, editors, *Euro-*

graphics 2010 - Short Papers. The Eurographics Association, 2010.

[8] Fabrice Rousselle, Claude Knaus, and Matthias Zwicker. Adaptive sampling and reconstruction using greedy error minimization. *ACM Trans. Graph.*, 30(6):159:1–159:12, December 2011.

[9] Fabrice Rousselle, Claude Knaus, and Matthias Zwicker. Adaptive rendering with non-local means filtering. *ACM Trans. Graph.*, 31(6):195:1–195:11, November 2012.

[10] Fabrice Rousselle, Marco Manzi, and Matthias Zwicker. Robust denoising using feature and color information. *Computer Graphics Forum*, 32(7):121–130, 2013.

[11] Dietger van Antwerpen. Improving simd efficiency for parallel monte carlo light transport on the gpu. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, HPG '11, pages 41–50, New York, NY, USA, 2011. ACM.

[12] Ingo Wald. Active thread compaction for gpu path tracing. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, HPG '11, pages 51–58, New York, NY, USA, 2011. ACM.

[13] M. Zwicker, W. Jarosz, J. Lehtinen, B. Moon, R. Ramamoorthi, F. Rousselle, P. Sen, C. Soler, and S.-E. Yoon. Recent advances in adaptive sampling and reconstruction for monte carlo rendering. *Comput. Graph. Forum*, 34(2):667–681, May 2015.