Improving SIMD Efficiency for Parallel Monte Carlo Light Transport on the GPU

Dietger van Antwerpen
Delft University of Technology, Netherlands

Abstract

Monte Carlo Light Transport algorithms such as Path Tracing (PT), Bi-Directional Path Tracing (BDPT) and Metropolis Light Transport (MLT) make use of random walks to sample light transport paths. When parallelizing these algorithms on the GPU the stochastic termination of random walks results in an uneven workload between samples, which reduces SIMD efficiency. In this paper we propose to combine stream compaction and sample regeneration to keep SIMD efficiency high during random walk construction, in spite of stochastic termination. Furthermore, for BDPT and MLT, we propose to evaluate all bidirectional connections of a sample in parallel in order to balance the workload between GPU threads and improve SIMD efficiency during sample evaluation. We present efficient parallel GPU-only implementations for PT, BDPT, and MLT in CUDA. We show that our GPU implementations outperform similar CPU implementations by an order of magnitude.

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

Keywords: Monte Carlo Light Transport, Path Tracing, GPU

1 Introduction

Since the introduction of GPGPU programming, there has been a great deal of research on Path Tracing on the GPU. Purcell was the first to publish a GPU path tracer [Purcell et al. 2002]. Since then, many researchers have attempted to improve GPU path tracing performance. Most of this work has been focused on efficient ray traversal on the GPU [Popov et al. 2007; Aila and Laine 2009]. However, since the introduction of the path tracing algorithm, the field of physically based rendering has made significant advances. Considerable improvements are Bi-Directional Path Tracing and the Metropolis Light Transport algorithm [Veach and Guibas 1995; Veach and Guibas 1997]. These Monte Carlo Light Transport algorithms are often viewed as being embarrassingly parallel [Marques and Santos 2010]. Extracting parallelism seems as trivial as constructing many independent samples in parallel. However, when mapping PT or BDPT to the GPU, such a trivial implementation does not take the GPU's streaming architecture into account. On GPUs with CUDA architecture, threads are processed simultaneously in groups of 32, called warps, all executing in SIMD [NVIDIA Corporation 2009]. To achieve maximum fine-grained parallelism, a GPU requires coherence in both code execution and memory access between threads in a warp. Because of this, efficiently executing these Monte Carlo algorithms in parallel on the GPU proves to be challenging.

Novák pointed out that the main complication lies in the stochastic nature of the algorithms, resulting in differences in path length for random walks [Novák et al. 2010]. This leads to an uneven workload and incoherent execution between samples. After each bounce during eye/light path construction, a fraction of all paths is stochastically terminated and the corresponding threads become inactive. Because the number of active threads in a warp drops as the paths are randomly terminated, SIMD efficiency is reduced. For BDPT this problem reappears when evaluating the connections between light and eye paths. The number of connections per sample is linear in the lengths of both its eye and light path. Hence, some samples will have to evaluate many more connections than others. As of now only little effort has been made to tackle these problems and implement advanced algorithms such as BDPT and MLT on the GPU [Novák et al. 2010; Pajot et al. 2011].

With this work, our contributions are the following:

- We propose to combine stream compaction and sample regeneration to keep SIMD efficiency high in the face of stochastic random walk termination.
- For BDPT and MLT we propose to evaluate all bidirectional connections for a sample in parallel in order to balance the workload between GPU threads and improve SIMD efficiency.
- We present efficient CUDA implementations for PT, BDPT and MLT, all running entirely on the GPU.

The rest of the paper is structured as follows: In Section 2 we start with a short description of Monte Carlo light transport, followed by a discussion of relevant related work in Section 3. In Section 4 we discuss the PT implementation. The BDPT and MLT implementations are discussed in Sections 5 and 6. Finally, we discuss the results and conclusions in Section 7.

2 Monte Carlo Light Transport

In the context of Computer Graphics, Monte Carlo light transport algorithms render an image by simulating physical light transport through the scene. To make an image, the algorithm samples many random paths transporting light from the light source to the eye. A light transport path $\overline{x} = x_0 \cdots x_k$ is a sequence of points with x_0 on the lens, x_k on the light source and $x_1 \cdots x_{k-1}$ on the scene surface or inside a participating medium [Lafortune and Willems 1996]. The total radiance I reaching the eye is found by estimating the integral of the measurement contribution function $f(\overline{x})$ over the space Ω of all light transport paths:

$$I = \int_{\Omega} f(\overline{x})\, d\Omega(\overline{x})$$

This multi-dimensional integral is estimated using the Monte-Carlo method. For efficiency, algorithms usually sample collections of correlated light transport paths. Using multiple importance sampling, these paths are combined into a single sample X from corresponding sample space P and with measurement contribution function $f(X)$ [Veach and Guibas 1995]. The algorithm generates N

Copyright © 2011 by the Association for Computing Machinery, Inc.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions Dept, ACM Inc., fax +1 (212) 869-0481 or e-mail permissions@acm.org.
HPG 2011, Vancouver, British Columbia, Canada, August 5 – 7, 2011.
© 2011 ACM 978-1-4503-0896-0/11/0008 $10.00

delivered by
EUROGRAPHICS DIGITAL LIBRARY
www.eg.org diglib.eg.org

such samples $X_1 \cdots X_N$ according to some probability distribution $p(X)$. These samples are combined into a single Monte Carlo estimate as

$$I \approx \frac{1}{N} \sum_{i=1}^{N} \frac{f(X_i)}{p(X_i)} \quad (1)$$

The variance in this estimate shows as noise in the rendered image. The various light transport algorithms differ in the way random light transport paths are sampled, resulting in different variances for different lighting effects.

Path tracing generates samples by simulating a random walk through the scene (see Figure 1). A path starts at the eye and is traced backwards into the scene. The random walk is randomly terminated at each bounce using Russian roulette. Light transport paths are found by explicitly connecting the path vertices to light sources. When the random walk happens to hit a light source directly this also produces a light transport path, called an implicit path. Note how a single path tracing sample may contain multiple light transport paths.

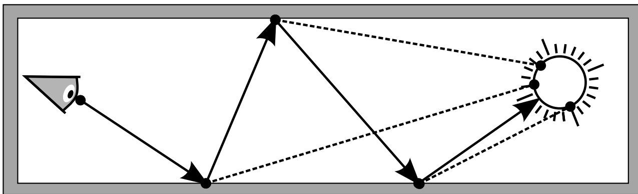

Figure 1: *Path Tracing sample.*

Although PT is fairly efficient for rendering outdoor scenes, it often suffers from high variance for indoor scenes with significant amounts of indirect light and caustics. Bi-Directional Path Tracing usually performs much better for such scenes [Lafortune and Willems 1993; Veach and Guibas 1995]. It samples an eye path $Y = y_0 \cdots y_s$ and a light path $Z = z_0 \cdots z_t$ using random walks and connects these to form complete light transport paths (see Figure 2). The eye path starts at the eye and is traced backwards into the scene. The light path starts at a light source and is traced forward into the scene. The light transport paths are constructed by explicitly connecting all vertices on the eye path to all vertices on the light path.

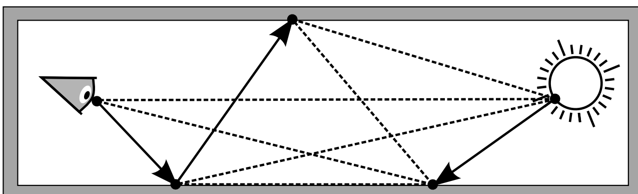

Figure 2: *Bi-Directional Path Tracing sample.*

Importance sampling is a technique to reduce the variance in a Monte Carlo estimate by sampling approximately proportional to the measurement contribution function $f(X)$; the more $p(X)$ resembles $f(X)$, the lower the variance. Ideally the sampling probability $p(X)$ is proportional to $f(X)$. The Metropolis Light Transport algorithm generates such a sequence of samples using a Markov chain in which the next sample X_{i+1} is generated from X_i through mutations. The stationary distribution of the produced sequence $X_1 \cdots X_N$ is proportional to $f(X)$. Note that the initial sample X_1 is usually not sampled proportional to $f(X)$, but according to some convenient distribution $p'(X)$. Because of this, the algorithm suffers from startup bias [Szirmay-Kalos et al. 1999]. This bias is larger when $f(X)$ and $p'(X)$ are less alike. MLT often performs much better than PT or BDPT for scenes with pathologically difficult lighting effects.

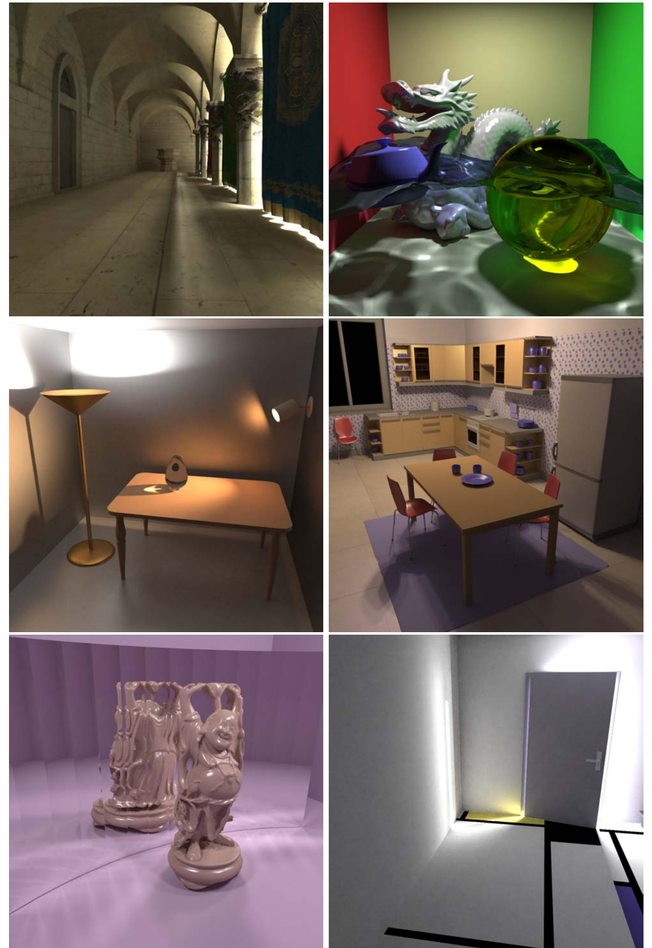

Figure 3: *Test scenes (from top left to bottom right): Crysis Sponza, Cornell Box, Glass Egg, Kitchen, Buddha and Invisible Date.*

3 Related Work

Particularly relevant to our discussion is the work of Novák, who studied the decrease in SIMD efficiency due to the stochastic termination of paths during PT. He proposed to keep the average SIMD efficiency high by immediately regenerating terminated paths [Novák et al. 2010]. Horn studied the use of parallel stream compaction for deferred shading in GPU ray tracing. By sorting the samples based on shader type he improved the SIMD efficiency during shading [Horn et al. 2007].

Recently, some attempts have been made at mapping the BDPT algorithm to the GPU. Novák presented the first BDPT implementation running entirely on the GPU [Novák et al. 2010]. First, a large number of independent light paths are generated and stored in GPU memory. Then, during eye path tracing, each eye vertex is connected to all vertices on one of these light paths. This method uses a suboptimal importance sampling scheme where termination decisions are shared by all threads in a warp to keep GPU efficiency

high. Pajot presented a hybrid BDPT implementation, balancing work between CPU and GPU [Pajot et al. 2011]. Large batches of independent light and eye paths are generated on the CPU. Then, all path vertices of all samples are copied to the GPU where all light path vertices are connected to all eye path vertices of all samples. This results in high GPU efficiency at the cost of some extra correlation between the samples. A disadvantage of this method is that implicit paths and caustic paths (light paths directly connected to the eye) are only sampled on the CPU. As a consequence, light effects such as caustics and reflected caustics, which are mainly sampled through implicit and caustic paths, benefit less from the GPU and remain relatively noisy. Furthermore, because both the method of Pajot and Novák connect each eye path to several different light paths, these methods are not well suited as an underlying sampler for MLT.

4 Path Tracing

A GPU path tracer constructs a large stream of samples in parallel. All samples are repeatedly extended with one vertex and connected to a random point on a light source. After each extension, some samples in the stream may have terminated due to Russian roulette. As explained in Section 1, keeping SIMD efficiency high in spite of stochastic sample termination is a major complication when implementing PT on the GPU. Novák proposed to keep SIMD efficiency high by immediately regenerating new samples for terminated samples without waiting for the other samples in the stream to terminate. After each extension, a new sample is generated in-place for each terminated sample in the stream, that is, at the same location in the stream. Consequently, the sample stream has a fixed length, guaranteeing full GPU utilization. Furthermore, because all terminated samples are immediately regenerated, SIMD efficiency remains high during sample extension.

Regenerate SIMD efficiency	
CRYSIS SPONZA	33.4%
CORNELL BOX	25.7%
GLASS EGG	34.2%
KITCHEN	35.1%
BUDDHA	31.1%
INVISIBLE DATE	33.0%

Table 1: *Average fraction of active threads in a GPU warp during in-place sample regeneration.*

A major disadvantage of this method is that terminated samples do not necessarily lie side-by-side but are scattered all over the sample stream. Therefore, because on average only a fraction of all samples in each warp require regeneration, SIMD efficiency during the path regeneration itself is relatively low (see Table 1). Furthermore, this method cannot benefit from primary ray coherence. Primary rays all have similar origin and direction, which is called primary ray coherence [Wald et al. 2001]. Coherent rays will often visit similar parts of the scene. When coherent rays are traced by threads in a single GPU warp, SIMD efficiency increases and GPU caches become more effective. This results in a significant increase in ray traversal performance [Aila and Laine 2009]. However, because regenerated samples are scattered throughout the stream, their primary rays are not traced together and therefore do not benefit from primary ray coherence.

4.1 Overview

We propose to combine path regeneration with stream compaction to keep SIMD efficiency and GPU utilization high without sacrificing primary ray coherence. We will refer to our method as SPT (Streaming Path Tracing). After extending all samples in the sample stream with another path vertex, instead of regenerating all terminated samples in-place, all terminated samples are removed from the stream using stream compaction. This leaves a continuous stream of active samples. Consequently, because the algorithm works on a continuous stream of active samples at all times, stochastic path termination no longer reduces SIMD efficiency during sample extension. Note that because terminated samples are removed from the stream, the stream becomes shorter after each extension. To fully utilize all GPU resources the stream must remain significantly large. Therefore, new samples are regenerated at the end of the stream after each extension. Because these newly generated samplers are placed side-by-side at the end of the stream, SIMD efficiency remains high during sample regeneration. Furthermore, the primary rays of regenerated samples will be traced together, thereby taking advantage of primary ray coherence. Note that, besides primary ray coherence, any secondary ray coherence due to specular reflection is also exploited by this method: When coherent primary rays all hit the same mirror the coherent secondary reflection rays remain side-by-side in the sample stream. Hence, the coherence between samples in the stream is preserved. Therefore, SPT also performs well for regular Whitted style ray tracing [Whitted 1980].

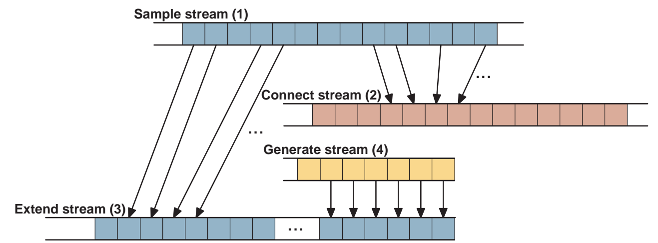

Figure 4: *Stream processing in PT iteration.*

4.2 Details

Figure 4 shows how streams are processed during one iteration of the SPT algorithm. The input stream of an iteration is a long continuous stream of samples (1). Each sample stores only the last vertex on its path. During an iteration, each sample may evaluate one connection to a light source and generate an outgoing extension ray. This results in two output streams: a connection stream (2) and an extension stream (3). The connection stream contains all connections to be evaluated; the extension stream contains all samples that are to be extended. Not all samples generate a connection; if the current path vertex is specular or points away from the light source, the connection would have no contribution, so there is no need to evaluate the connection. Similarly, as already discussed, due to Russian roulette not all samples are extended. Therefore, the two output streams may contain inactive elements. These inactive elements are removed from the stream through stream compaction. After compaction, new samples are regenerated (4) at the end of the extension stream. Finally, all connections and extensions are evaluated, and for each successful connection some energy is deposited on the image plane. Each extension results in a new path vertex. Because a sample only stores the last path vertex, these new vertices form the input sample stream for the next iteration. The sample stream buffers are reused through double buffering.

4.3 Stream Compaction

Using stream compaction to remove all inactive elements from an output stream shows similarities to the deferred shading approach

by Horn [Horn et al. 2007]. Horn described a method for parallel stream compaction on the GPU using parallel scan primitives in CUDA as presented by Sengupta [Sengupta et al. 2007]. In this stream compaction method all active elements in the stream are labeled as active. A parallel scan is applied to these labels, computing the number of active elements preceding each element in the stream. This number indicates the destination of each active element in the compacted stream. The active elements are then scattered to these locations to construct the compacted stream. This method requires a separate parallel scan and scatter pass. Furthermore, the execution of a full parallel scan also requires multiple passes.

We use a different method that immediately generates a compacted output stream without requiring the separate compaction passes. To achieve this, all active elements are packed before they are written to the output stream using atomic instructions. A counter keeps track of the number of elements written to an output stream. The counter is initialized to zero. Each time a batch of active elements is written to the stream, the counter is atomically increased by the size of this batch, effectively allocating stream space for the batch.

Colliding atomic instructions are serialized by the hardware, so it is important to keep the number of atomic operations to a minimum. It is therefore impractical for each active sample to independently allocate an element in the output stream. Instead, groups of samples work together. In CUDA, threads running within a block can efficiently synchronize and communicate through shared memory. The parallel scan implementation by Sengupta uses this property to reduce the number of passes required to do a full scan. A block can perform a partial parallel scan on its segment of the stream in one pass using shared memory. We use a similar method to generate one batch of active elements per block. First, the location of each active element within the local batch is determined using a parallel scan operation over activation labels in the block through shared memory. Because this scan is performed through shared memory, no separate pass is required. Then, a single atomic instruction per block is used to allocate enough space in the final output stream to hold this batch. Finally, using the globally allocated space and the local index in the batch, each active thread in the block writes its element to the output stream. The resulting output stream is a continuous stream of active elements. Because the compaction is performed just before actual output, no separate parallel scan or scatter passes need to be performed. Furthermore, as only a single atomic instruction is issued per block and the work per block is significant, atomic collisions do not significantly degrade performance.

4.4 Results

Table 2 compares the performance of our SPT implementation with and without sample regeneration. As a reference, the performance is also compared with a regular path tracing implementation with in-place sample regeneration (PT+R) as described by Novák [Novák et al. 2010]. The table shows that path tracing with in-place regeneration has similar performance as path tracing with stream compaction. The increase in primary ray coherence and SIMD efficiency for SPT offset the degradation in GPU utilization due to decreasing stream lengths and the slight overhead of stream compaction. Adding sample regeneration to SPT shows a significant increase in performance of roughly 22%, indicating better utilization of GPU resources due to longer sample streams.

Our method benefits from primary ray coherence and preserves ray coherence for reflections at specular and glossy materials. Despite the overhead of stream compaction, the achieved ray coherence gives a significant performance improvement. However, we did not attempt to extract further coherence from secondary rays. After a few bounces, rays will usually be completely diverged and all initial coherence will be lost. As proposed by Garanzha, it might be worthwhile to extract further coherence from secondary rays using spatial reordering [Garanzha and Loop 2010]. Note however that the overhead of spatial reordering is relatively high. Furthermore, as shown in Table 1, a significant percentage of all samples is regenerated after each iteration and thus already benefits from primary ray coherence. Therefore, we expect the gains of secondary ray reordering to be small.

Performance in 10^6 samples/s				
	PT+R	**SPT**	**SPT+R**	**Speedup**
CRYSIS SPONZA	8.7	9.9	10.8	24.1%
CORNELL BOX	5.8	6.2	7.1	22.4%
GLASS EGG	16.4	15.4	19.8	20.7%
KITCHEN	11.0	11.3	13.5	22.7%
BUDDHA	8.3	11.0	12.3	48.2%
INVISIBLE DATE	15.6	15.1	18.1	16.0%

Table 2: *PT performance comparison between in-place sample regeneration (PT+R), stream compaction (SPT), and stream compaction with sample regeneration (SPT+R).*

5 Bi-Directional Path Tracing

When implementing BDPT on the GPU it seems natural to construct many BDPT samples in parallel, similar to a GPU path tracer. The construction of these BDPT samples consists of two phases: a random walk and a connect phase. During the random walk phase, an eye and light path is constructed for each sample using two random walks. When both the eye and light path of a sample have terminated, all connections between these paths are evaluated during the connect phase. As explained in Section 1, the stochastic termination of eye and light paths results in an uneven workload between samples. Consequently, constructing a single BDPT sample per GPU thread results in relatively low SIMD efficiency. Table 3 shows the average number of active threads in a GPU warp during the random walk and connect phases for such a naive implementation.

SIMD efficiency		
	Random walk phase	**Connect phase**
CRYSIS SPONZA	29.2%	17.4%
CORNELL BOX	25.0%	14.0%
GLASS EGG	35.5%	19.4%
KITCHEN	33.2%	18.4%
BUDDHA	30.7%	17.3%
INVISIBLE DATE	29.7%	17.0%

Table 3: *Average fraction of active threads in a GPU warp for naive BDPT implementation.*

We propose to use stream compaction and sample regeneration to keep SIMD efficiency high, similar to SPT. Because terminated paths are immediately removed from the path stream during the random walk phase, the algorithm works on a continuous steam of active paths at all times, resulting in maximum SIMD efficiency during the random walk phase. Note however that this approach only increases SIMD efficiency during the random walk phase. During the connect phase, some samples still have to evaluate many more connections than others. Therefore, instead of running a single GPU thread per sample which sequentially evaluates all connections, we further propose to evaluate all connections for all samples in parallel by executing a single GPU thread for each bidirectional connection. Evaluating a single connection per thread results in an

even workload between threads and consequently maximum SIMD efficiency.

This combination of stream compaction, sample regeneration, and parallel evaluation of connections results in an efficient, GPU-only BDPT implementation with high SIMD efficiency in spite of stochastic random walk termination. Furthermore, because we did not alter the BDPT sampling method itself, it turned out to be fairly straightforward to implement MLT on the GPU using our BDPT implementation.

5.1 Overview

Figure 5 gives an overview of the flow of execution in our BDPT implementation. The random walk phase uses our SPT implementation. At the start of the algorithm two fresh streams of new eye and light paths are generated; one eye and light path for each sample. These paths are then repeatedly extended with one path vertex. Because all path vertices of a sample are required during the connection phase, they are temporarily stored with the sample. The temporary storage of vertices is further discussed in Section 5.3. After each extension, the explicit connections of all light vertices to the lens are evaluated. The evaluation of all remaining connections is postponed to the connect phase. Similar to SPT, just before applying the next extension, all terminated eye and light paths are removed from the streams using stream compaction to keep SIMD efficiency high.

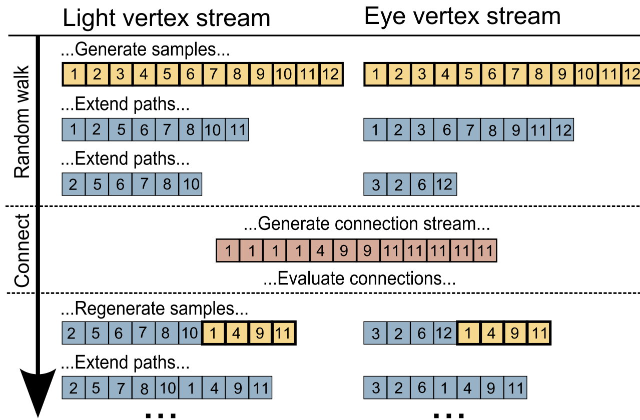

Figure 5: *Flow of execution in BDPT.*

Whenever both the eye and light path of some sample have terminated, the sample is complete and ready for evaluation during the next connect phase. Because terminated paths are removed from the path streams, the streams becomes shorter after each extension. Eventually, all paths will terminate and all samples will be ready for evaluation. However, before this point is reached, the path streams may become very short, thus under-utilizing the GPU. Similar to the PT implementation, we use sample regeneration to solve this problem. Instead of waiting for all paths to terminate, the paths are repeatedly extended until the streams become too short to fully utilize the GPU. Then, the random walk phase is interrupted and the next connect phase is started. At this point, many of the samples will be complete while a few samples with very long eye or light paths will yet be incomplete. In Figure 5 the eye and light paths for samples 1,4,9 and 11 have terminated after two extensions and are ready for evaluation. All complete samples are evaluated during the connection phase, skipping the incomplete samples. The parallel evaluation of all connections is discussed in the next section.

Time partition in %		
	Random walk	**Connect**
CRYSIS SPONZA	55.4%	42.8%
CORNELL BOX	55.3%	42.8%
GLASS EGG	55.3%	42.7%
KITCHEN	54.2%	44.1%
BUDDHA	39.4%	58.8%
INVISIBLE DATE	49.3%	48.0%

Table 4: *BDPT execution time partition between the random walk and connect phases.*

After the connection phase, all completed samples are regenerated. The old eye and light path of each regenerated sample are discarded and the construction of a fresh eye and light path is started. Similar to path regeneration in SPT, the new eye and light path are regenerated at the end of the eye and light path streams. The regenerated paths will be constructed during the next random walk phase, together with the remaining paths from incomplete samples. This regeneration guarantees enough parallelism during the random walk phase to keep the GPU utilized. Finally, after regeneration, the random walk phase is resumed. Table 5 shows the performance speedup due to sample regeneration. By starting the connection phase after 60% of all samples are complete, the performance is increased by roughly 15%. The impact of sample regeneration is less than for PT because BDPT spends almost half of its execution time in the connection phase, which does not benefit from sample regeneration. Table 4 shows a partition of the algorithm's time spent in the random walk phase and connection phase.

Performance in 10^6 samples/s per connect percentage			
	100%	**60%**	**Speedup**
CRYSIS SPONZA	2.98	3.64	22.1%
CORNELL BOX	2.53	2.93	15.8%
GLASS EGG	2.67	3.07	15.0%
KITCHEN	2.62	3.03	15.6%
BUDDHA	2.73	3.10	13.6%
INVISIBLE DATE	4.22	5.23	23.9%

Table 5: *Performance increase due to intermediate connection after 60% of all samples are complete.*

5.2 Parallel Connect

During the connection phase, the bidirectional connections for all complete samples must be evaluated. However, in order to guarantee an even workload between GPU threads, instead of processing each sample in parallel and evaluating a sample's connections sequentially, all connections are evaluated in parallel. This is done by generating a stream of all connections to be evaluated. Each complete sample contributes a number of connections to this stream. For some sample i, with N_i^L and N_i^E respectively its light and eye path lengths, a total of $N_i^L \times N_i^E$ connections must be evaluated (see Figure 6). Note that explicit connections to the eye are already evaluated during the random walk phase. Figure 6 also shows how a sample's connections map onto the connection stream.

What is left is the construction of the connection stream. Algorithm 1 shows how to construct this stream in parallel. First, all complete samples write out their number $n_i = N_i^L \times N_i^E$ of connections. All incomplete samples just write out a zero, effectively skipping them. Next, a parallel scan is applied to this list, resulting in a sequence of prefix sums $s_0, \cdots, s_{N-1}$. The prefix sum s_i equals the number of connections to be evaluated for samples 0 to i. Hence, $M = s_{N-1}$ equals the total number of connections to be evaluated. The next

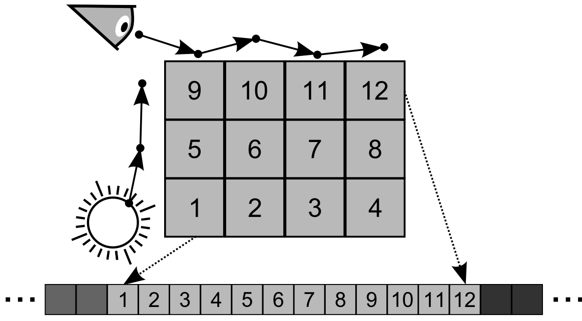

Figure 6: *Mapping of bidirectional connections to connection stream.*

step is to evaluate all M connections in parallel. A separate thread is run for each connection. The problem that remains is for thread j to know which eye and light vertex it is supposed to connect. First, we need to figure out which sample i some connection j belongs to. Remember that s_i indicates the total number of connections to be evaluated for samples 0 to i. So, connection j belongs to the first sample i with $s_i > j$. This sample is found through a binary search in the prefix sum list. Because nearby threads in the connection stream will search for similar indices in the list, their binary searches are highly coherent, resulting in high GPU performance. The computation time of this binary search is negligible compared to the actual evaluation of the connections. The next step is to compute the connection index $c_j = j - (s_i - N_i^L N_i^E)$ within sample i. The corresponding eye and light vertex are now easily obtained through the inverse mapping from Figure 6. At this point, each thread can load the vertices and evaluate the connection. Because each thread evaluates a single connection, all threads have a similar workload resulting in high SIMD efficiency.

Algorithm 1 : ParallelConnect

for $i = 0$ **to** $N-1$ **in parallel do**
  $n_i \leftarrow 0$
  if Sample i is complete **then**
    $n_i \leftarrow N_i^L \times N_i^E$
  end if
end for

$\{s_0, \cdots, s_{N-1}\} \leftarrow$PARALLEL_SCAN$(n_0, \cdots, n_{N-1})$

$M \leftarrow s_{N-1}$
for $j = 0$ **to** M **in parallel do**
  $i \leftarrow$BINARY_SEARCH$(j, \{s_0, \cdots, s_{N-1}\})$
  $c_j \leftarrow j - (s_i - N_i^L N_i^E)$
  Evaluate connection c_j of sample i
end for

5.3 Implementation Details

In contrast to the PT algorithm, BDPT requires the data of all path vertices of a sample during the connection phase. Therefore, vertices must be temporarily stored. During the random walk phase, path stream buffers are reused through double buffering. The path streams themselves only hold enough information from the last path vertex to extend each path with another vertex. The actual generated path vertices are stored elsewhere with the sample they belong to. Enough memory is allocated for each sample beforehand to store some maximum length eye and light path. Note that limiting the path length introduces bias so the maximum path length should be chosen significantly large.

5.3.1 Vertex Scattering

After each extend phase, all new path vertices are written to their corresponding memory slots. The path vertices are stored as an array of structures. Due to stream compaction and sample regeneration, consecutive paths in a path stream may correspond to non-consecutive samples and may be working on paths of different lengths. Therefore, if each thread in the stream just wrote out its new path vertex to sample memory, the resulting memory access patterns would be highly incoherent. Instead, we let threads in a GPU warp work together to store their path vertices.

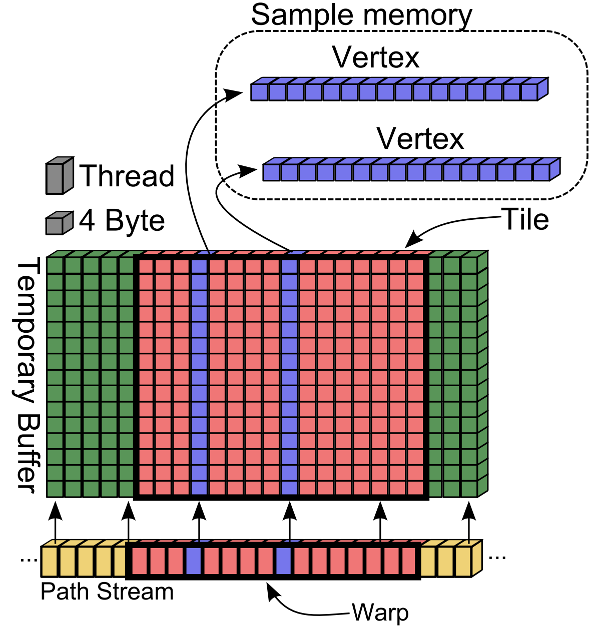

Figure 7: *Coalesced scattered vertex access.*

Figure 7 shows how this is done: First, all threads write out their subsequent path vertex as a structure of arrays in a temporary 2D buffer. Each column in this buffer corresponds to a path vertex. Because the vertices are written as a structure of arrays, memory access is coalesced. Now, each column in the temporary buffer needs to be transposed and stored at the correct vertex location in sample memory. Note that this problem is similar to transposing a matrix, except that the destination rows are now scattered through sample memory. Ruetsc showed how to implement an efficient matrix transpose in CUDA using shared memory[1][Ruetsc and Micikevicius 2009]. We use a similar solution for vertex scattering. In our implementation a compacted path vertex is at most 128 bytes in size. Each warp of 32 threads reads one tile of 32 vertices (columns) into shared memory. The tile is then transposed in shared memory and each row (vertex) is coherently written to memory by all 32 threads together, each thread writing one 4-byte word of each vertex. This solution allows for coherent memory access on CUDA GPUs, resulting in high effective memory bandwidth.

During the connect phase, each connection needs to read two path vertices, again resulting in incoherent memory access patterns. A similar approach could be used to coherently read the path vertices through shared memory. However, note that nearby connections in

[1]An example is shipped with the CUDA SDK.

the connection stream often belong to the same sample and therefore access the same path vertices. It therefore turned out to be more efficient to read path vertices through cached texture memory.

```
typedef struct
{
        Vector3 position;
        Vector3 normal;
        Vector3 in_direction;
        Vector3 in_radiance;
        float MIS[2];
        int materialIdx;
        char aux_bsdf[68];
} Vertex;
```

Listing 1: *Path vertex structure*

Listing 1 shows what the structure of a 128 byte path vertex might look like. Each vertex must store its position, normal and the incoming direction and radiance/importance along the path. Furthermore, each vertex must keep track of two temporary values used to compute the optimal MIS weights during the connection phase (See Section 5.3.2). The local shading context is identified using a material index. All remaining space may be used to cache auxiliary data such as BSDF colors and shading normals. In our implementation, we used a similar structure with compressed normals. Because we only used the relatively simple Blinn-Phong shading model [Blinn 1977], all vertex data fitted within the 128 bytes. However, for more complex shading models some information might not fit within the auxiliary cache and must be regenerated during connection. The auxiliary data must provide enough information to regenerate the missing data, for example by storing a triangle index and barycentric coordinates on the triangle.

5.3.2 Multiple Importance Sampling

As explained in Section 2, a BDPT sample usually contains multiple light transport paths. The contributions of these paths must be combined into a single Monte Carlo estimate using MIS. Veach proposed to weigh the contribution of each bidirectional connection using the balance heuristic [Veach and Guibas 1995]. He further showed that no other combination strategy can be a significant improvement over the balance heuristic. We therefore use this strategy. In order to do so, the balance heuristic weight must be computed during the evaluation of each bidirectional connection. Traditionally, the computation scheme for constructing a balance heuristic weight requires an iteration over all vertices on the corresponding light transport path. Using this scheme would lead to an uneven workload and therefore reduce the SIMD efficiency during the connection phase. Instead, we use a recursive computation scheme for the balance heuristic [Antwerpen 2011]. By recursively computing an extra quantity in each path vertex during eye and light path construction, the balance heuristic weights can be evaluated during the connection phase using only data from the eye and light vertex directly involved in the bidirectional connection. Furthermore, the evaluation requires only a fixed amount of operations, independent of the path lengths.

During the random walk phase, the probabilities $p\left(x_i \rightarrow x_{i+1}\right)$ and $p\left(x_{i-1} \leftarrow x_i\right)$ of sampling respectively path vertices x_{i+1} and x_{i-1} from vertex x_i are computed for each path vertex x_i. Using these quantities (still during the random walk phase) the following recursive quantity is computed and stored for each vertex on the path:

$$d^{x_i} = \frac{1 + p\left(x_{i-1} \leftarrow x_i\right) d^{x_{i-1}}}{p\left(x_{i-1} \rightarrow x_i\right)}$$

Now, when connecting some eye vertex y_i and light vertex z_j during the connection phase to form light transport path $X = y_0 \cdots y_i z_j \cdots z_0$, the corresponding balance heuristic weight $w_{i,j}\left(X\right)$ can be computed from these quantities as follows:

$$\frac{1}{w_{i,j}\left(X\right)} = d^{y_i} p\left(y_i \leftarrow z_j\right) + 1 + d^{z_j} p\left(y_i \rightarrow z_j\right)$$

Using the precomputed quantities d^{y_i} and d^{z_j}, evaluating this weight only requires data from the two connected vertices y_i and z_j. Furthermore the total amount of operations required does not depend on either i or j. Therefore, this scheme results in an even workload and high SIMD efficiency.

Note that the actual implementation is slightly more involved because the sample probability $p\left(x_i \rightarrow x_{i+1}\right)$ usually also depends on x_{i-1}. These issues are easily solved by carefully splitting up the computation of all quantities involved, but this requires an extra temporary quantity per path vertex (see Listing 1).

6 Metropolis Light Transport

We implemented MLT on top of our BDPT implementation. The implementation runs many independent MLT samplers in parallel. Each sampler repeatedly generates a mutated proposal sample using our BDPT implementation. At the end of the connection phase, each completed proposal sample is either accepted or rejected. Samples are mutated according to Kelemen's mutation strategy by mutating pseudo-random numbers [Kelemen et al. 2002]. A proposal sample is constructed by lazily mutating all random numbers used in the construction of the current sample. This results in a convenient, symmetrical mutation strategy. We use a small variation on this strategy: Instead of lazily mutating all numbers, we only mutate those random numbers that are required for both the construction of the current sample and that of the proposed sample. If the proposal sample requires more random numbers, these extra numbers are freshly generated. The resulting mutation strategy remains symmetrical, without ever having to postpone mutations. This improves SIMD efficiency and reduces memory bandwidth because we now can mutate random numbers during the random walk phase, without having to keep track of the amount of postponed mutations for each vertex.

6.1 Implementation Details

The Kelemen mutation strategy uses a small step and large step mutation. As proposed by Kelemen, we combined the results of both mutations using MIS. Furthermore, we used the results of the large step mutations to estimate the normalization constant. We choose a large step selection probability p_L of $\frac{1}{2}$. Although selecting p_L much smaller can be beneficial in some scenes, choosing p_L relatively large guarantees that MLT never performs much worse than BDPT, making it a more robust alternative. Furthermore, choosing p_L large reduces startup bias, which is important because running many independent MLT samplers in parallel significantly increases startup bias

Besides the path vertices, in MLT we also need to store the random numbers from which the current and proposed samples are constructed. During a random walk, the extension of a path with another path vertex requires 4 random numbers: One for Russian roulette and BSDF component selection, two for the directional sampling of the BSDF, and one for selecting the scattering distance in the presence of participating media. Each sample has two slots of 4 random numbers for each path vertex: One slot corresponds to the current sample and one to the proposed mutation. All samples have a flag indicating which slot contains the numbers for the current

sample. During the random walk, a vertex is mutated by reading its current numbers, mutating these and writing the mutated numbers back in the mutation slot. When a mutation is accepted, the sample flag is inverted, effectively setting the current sample to the newly mutated sample.

Explicit connections from a light vertex to the lens can contribute to any pixel. In the BDPT implementation, we use atomics to deposit sample contributions on the image plane. However, in MLT the contribution of these connections must be postponed until the proposal sample is actually accepted. Consequently, these connections significantly complicate the MLT implementation, increasing memory consumption and reducing SIMD efficiency. Therefore, our MLT implementation does not support explicit connections from light vertices to the lens. In practice, the use of MLT makes up for increased variance in the underlying BDPT sampler, at the cost of some extra startup bias.

7 Results and Conclusion

All algorithms were implemented within the Brigade path tracing framework [Bikker 2010]. We compared the performance of our implementations with similar CPU algorithms, parallelized to take advantage of all virtual cores in the system. For both the CPU and GPU implementations, we used a BVH as acceleration structure for intersection tests.

7.1 Results

All experiments were executed on an Intel® Core™i7 CPU 920 with an NVIDIA® GeForce® GTX 480 GPU. Table 7 shows the performance of our implementations for the test scenes in samples per second. The test results show that the GPU outperforms the CPU by an order of magnitude. It also shows that the advanced light transport rendering algorithms enjoy a similar speedup from the use of GPUs as the more basic PT algorithm.

Memory consumption per sample in bytes			
	Overhead	**Per Vertex**	**Total**
SPT	152	0	152
BDPT	332	96	3404
MLT	388	128	4434

Table 6: *Memory footprint per sample for SPT, BDPT, MLT for path lengths of up to 16 vertices.*

Table 6 shows the memory consumption of the three algorithms for a single sample with up to 16 vertices per path (32 vertices in total for BDPT). As the table shows, the memory consumption of BDPT and MLT is significantly higher than for SPT, mainly because the memory consumption of BDPT and MLT depends on the maximum path length. To allow for effective load balancing and enough parallelism to utilize all GPU resources, stream sizes should be at least several times the theoretical maximum number of threads that can execute concurrently on a GPU. Therefore, these algorithms are best suited for a modern GPU with a large amount of video memory, being able to run several tens of thousands of samples in parallel while leaving enough free memory to store a reasonably sized scene.

Figure 10 compares the noise between PT, BDPT, and MLT after 30 seconds of rendering. As expected, these images show that BDPT and MLT often perform significantly better for scenes with difficult lighting conditions. BDPT performs especially better than PT for scenes with caustics, such as the Cornell Box, Glass Egg and Kitchen scenes. In these last two scenes, lamps were modeled with lenses, resulting in large caustics in the scene. BDPT however still performs badly for reflected caustics (Buddha) and very inaccessible light sources (Invisible Date). In case of the Buddha scene, BDPT performs even worse than PT because PT traces many more eye paths per second, which are required to capture the reflected caustics. MLT performs much better than BDPT for these hard cases, producing much less noise. However, for some scenes MLT suffers heavily from startup bias. As shown in Figure 8, the startup bias for intermediate images can be very significant. Luckily, the startup bias in MLT is largest when the underlying sampler (BDPT) performs the worst, leaving MLT as a robust alternative. Furthermore, the startup bias problem could be eliminated by resampling the initial samples as proposed by Veach [Veach and Guibas 1997].

7.2 Conclusion and Future Work

In this paper we have shown how to improve the SIMD efficiency for random walk construction using stream compaction and sample regeneration. Furthermore, we showed how to evaluate all bidirectional connections in parallel, further improving the SIMD efficiency during BDPT sample evaluation. We presented GPU-only implementations for the rendering algorithms PT, BDPT, and MLT. We showed that, as expected, the BDPT and MLT implementations often perform much better than the basic PT implementation. Furthermore, we showed that our GPU implementations outperform similar CPU implementations by an order of magnitude, proving the worth of GPUs for implementing advanced Metropolis Light Transport algorithms. In this paper, we have only touched upon the topic of participating media. However, extending the algorithms to support participating media as described by Lafortune was fairly straightforward [Lafortune and Willems 1996]. For simplicity, we only implemented homogeneous participating media, but extending this to inhomogeneous participating media is easily done by executing a ray marching kernel right after each ray intersection kernel. Figure 9 shows the Glass Egg scene with participating media, rendered with our MLT implementation.

Although our algorithms can handle complex shading models, we only implemented the relatively simple Blinn-Phong shading model. A naive implementation of more complex layered or procedural shading models is likely to reduce the SIMD efficiency. Implementing such shading models without significantly reducing SIMD efficiency requires further research.

Our GPU implementations currently require the entire scene to fit into video memory, limiting the size of the scenes. Extending these rendering algorithms for out-of-core rendering on the GPU is left as future work.

We would like to thank Erik Jansen, Jacco Bikker and the anonymous HPG reviewers for their valuable suggestions and comments. The Glass Egg scene was provided by Reinier van Antwerpen.

References

AILA, T., AND LAINE, S. 2009. Understanding the Efficiency of Ray Traversal on GPUs. In *Proc. HPG '09*, ACM, 145–149.

ANTWERPEN, D. G. 2011. Recursive MIS Computation for Streaming BDPT on the GPU.

BIKKER, J., 2010. Brigade Real-Time Path Tracing. `http://igad.nhtv.nl/~bikker/`, June.

BLINN, J. F. 1977. Models of Light Reflection for Computer Synthesized Pictures. *SIGGRAPH Comput. Graph. 11* (July), 192–198.

Figure 8: *Startup bias in Buddha scene is high w.r.t sample variance. Buddha scene after 10 seconds, 1 minute and 10 minutes.*

Figure 9: *Glass Egg scene with participating media.*

GARANZHA, K., AND LOOP, C. 2010. Fast Ray Sorting and Breadth-First Packet Traversal for GPU Ray Tracing. *Computer Graphics Forum 29*, 2 (May).

HORN, D. R., SUGERMAN, J., HOUSTON, M., AND HANRAHAN, P. 2007. Interactive k-D Tree GPU Raytracing. In *Proc. I3D '07*, ACM, 167–174.

KELEMEN, C., SZIRMAY-KALOS, L., ANTAL, G., AND CSONKA, F. 2002. A Simple and Robust Mutation Strategy for the Metropolis Light Transport Algorithm. In *Proc. of Computer Graphics Forum*, 531–540.

LAFORTUNE, E. P., AND WILLEMS, Y. D. 1993. Bi-Directional Path Tracing. In *Proc. COMPUGRAPHICS '93*, 145–153.

LAFORTUNE, E. P., AND WILLEMS, Y. D. 1996. Rendering Participating Media with Bidirectional Path Tracing. In *Proc. EGRW'96*, 91–100.

MARQUES, R., AND SANTOS, L. P. 2010. Instant Global Illumination on the GPU using OptiX. In *Proc. INForum'10*, 329–340.

NOVÁK, J., HAVRAN, V., AND DACHSBACHER, C. 2010. Path Regeneration for Interactive Path Tracing. In *Proc. EUROGRAPHICS '07, Short Papers*, 61–64.

NVIDIA CORPORATION, 2009. NVIDIA CUDA Compute Unified Device Architecture Programming Guide. `http://developer.nvidia.com/cuda`, Jan.

PAJOT, A., BARTHE, L., PAULIN, M., AND POULIN, P. 2011. Combinatorial Bidirectional Path-Tracing for Efficient Hybrid CPU/GPU Rendering. In *Proc. EUROGRAPHICS '11*, ACM, 315–324.

POPOV, S., GÜNTHER, J., SEIDEL, H.-P., AND SLUSALLEK, P. 2007. Stackless kD-Tree Traversal for High Performance GPU Ray Tracing. *In Computer Graphics Forum 26*, 3 (Sept.), 415–424.

PURCELL, T. J., BUCK, I., MARK, W. R., AND HANRAHAN, P. 2002. Ray Tracing on Programmable Graphics Hardware. In *Proc. SIGGRAPH '02*, ACM, 703–712.

RUETSC, G., AND MICIKEVICIUS, P. 2009. Optimizing Matrix Transpose in CUDA. Tech. rep., NVIDIA Corporation.

SENGUPTA, S., HARRIS, M., ZHANG, Y., AND OWENS, J. D. 2007. Scan Primitives for GPU Computing. In *Proc. SIGGRAPH '07*, 97–106.

SZIRMAY-KALOS, L., DORNBACH, P., AND PURGATHOFER, W. 1999. On the Start-Up Bias Problem of Metropolis Sampling. In *Proc. WSCG'99*, 273–280.

VEACH, E., AND GUIBAS, L. J. 1995. Optimally Combining Sampling Techniques for Monte Carlo Rendering. In *Proc. SIGGRAPH '95*, ACM, 419–428.

VEACH, E., AND GUIBAS, L. J. 1997. Metropolis Light Transport. In *Proc. SIGGRAPH '97*, Addison Wesley, 65–76.

WALD, I., SLUSALLEK, P., BENTHIN, C., AND WAGNER, M. 2001. Interactive Rendering with Coherent Ray Tracing. In *Proc. of Computer Graphics Forum*, 153–164.

WHITTED, T. 1980. An Improved Illumination Model for Shaded Display. *Commun. ACM 23* (June), 343–349.

Performance in 10^6 samples/s									
	PT			**BDPT**			**MLT**		
	CPU	GPU	Speed up	CPU	GPU	Speed up	CPU	GPU	Speed up
CRYSIS SPONZA	0.61	10.8	17.8x	0.24	3.64	15.2x	0.19	2.84	15.0x
CORNELL BOX	0.62	7.1	11.5x	0.28	2.93	10.4x	0.28	2.97	10.5x
GLASS EGG	1.74	19.8	11.4x	0.38	3.07	8.1x	0.3	2.71	9.0x
KITCHEN	1.12	13.5	12.1x	0.27	3.03	11.3x	0.24	2.89	12.1x
BUDDHA	1.18	12.3	10.4x	0.32	3.10	9.7x	0.33	3.06	9.2x
INVISIBLE DATE	1.49	18.1	12.2x	0.38	5.23	13.9x	0.31	3.85	12.5x

Table 7: *Performance comparison between CPU and GPU in samples per second for PT, BDPT and MLT on an Intel®Core™i7 CPU 920 and NVIDIA®GeForce®GTX 480.*

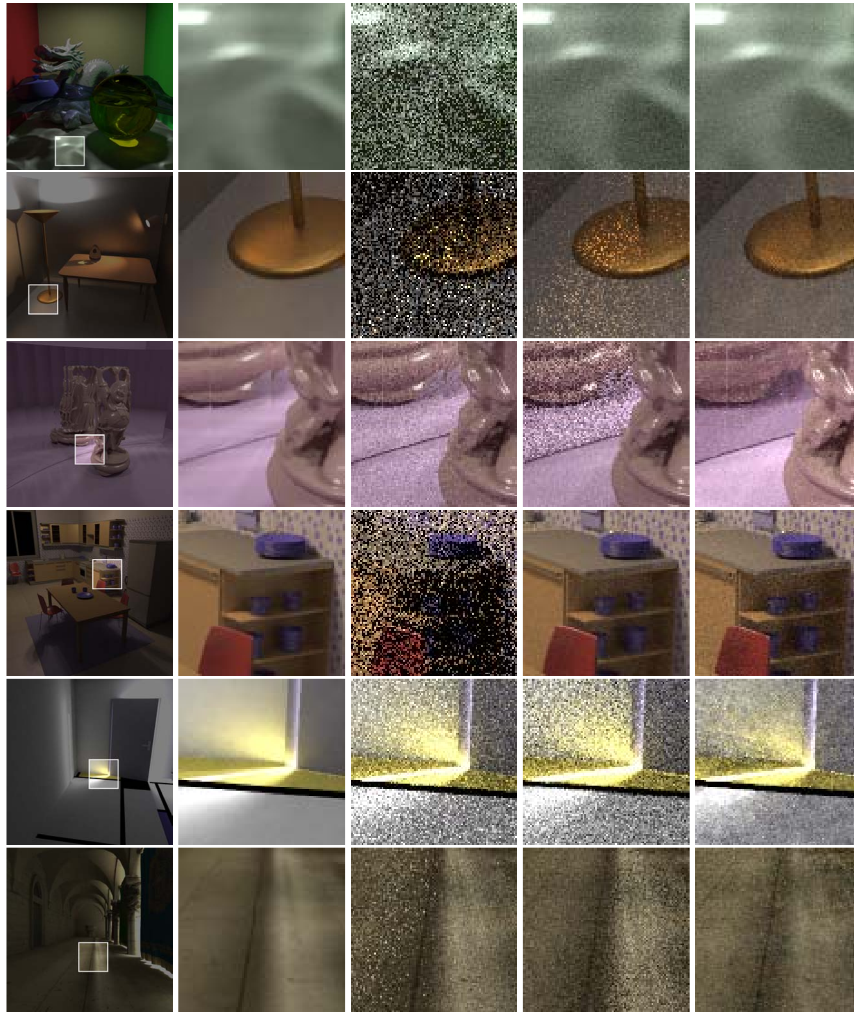

Figure 10: *Noise comparison for PT, BDPT and MLT after 30 seconds. From left to right: Reference, PT, BDPT and MLT.*