Progressive Light Transport Simulation on the GPU: Survey and Improvements (rev 1.)

TOMÁŠ DAVIDOVIČ
Saarland University, Intel VCI
and
JAROSLAV KŘIVÁNEK
Charles University in Prague, Faculty of Mathematics and Physics
and
MILOŠ HAŠAN
Autodesk, Inc.
and
PHILIPP SLUSALLEK
Saarland University, DFKI

Graphics Processing Units (GPUs) recently became general enough to enable implementation of a variety of light transport algorithms. However, the efficiency of these GPU implementations has received relatively little attention in the research literature and no systematic study on the topic exists to date. The goal of our work is to fill this gap. Our main contribution is a comprehensive and in-depth investigation of the efficiency of the GPU implementation of a number of classic as well as more recent progressive light transport simulation algorithms. We present several improvements over the state-of-the-art. In particular, our Light Vertex Cache, a new approach to mapping connections of sub-path vertices in Bidirectional Path Tracing on the GPU, outperforms the existing implementations by 30-60%. We also describe a first GPU implementation of the recently introduced Vertex Connection and Merging algorithm [Georgiev et al. 2012], showing that even relatively complex light transport algorithms can be efficiently mapped on the GPU. With the implementation of many of the state-of-the-art algorithms within a single system at our disposal, we present a unique direct comparison and analysis of their relative performance.

Categories and Subject Descriptors: I.3.7 [**Computer Graphics**]: Three-Dimensional Graphics and Realism—*Raytracing*

Additional Key Words and Phrases: Global Illumination, GPU, High Performance, Bidirectional Path Tracing, Vertex Connection and Merging

davidovic@cs.uni-saarland.de; jaroslav.krivanek@mff.cuni.cz
milos.hasan@gmail.com; slusallek@cs.uni-saarland.de

DOI 10.1145/XXXXXXX.YYYYYYY
http://doi.acm.org/10.1145/XXXXXXX.YYYYYYY

1. INTRODUCTION

Global illumination research has recently focused on progressive global illumination algorithms: ones that converge to the correct solution to the rendering equation (under some assumptions) given enough time, and display partial results during convergence. Well-known approaches such as Path Tracing and Bidirectional Path Tracing belong to this category. Furthermore, recent advances brought more such algorithms: Progressive Photon Mapping (including a bidirectional version), and Vertex Connection and Merging.

On the other hand, graphics processing units (GPUs) became more flexible over recent years, and a proliferation of global illumination algorithms were ported to them. These ports have mostly focused on proof-of-concept GPU implementations, showing a performance improvement over CPU rendering.

We take it as established that progressive global illumination algorithms are useful, and that they can be ported to the GPU, achieving significant speedups. But are these implementations optimal? And which progressive algorithms perform the best on GPUs? These questions are far from answered; there is currently a complete lack of such a rigorous and systematic study of the implementation and performance of different progressive global illumination algorithms on GPUs.

Our work fills this wide gap with a first comprehensive and in-depth investigation of the problem. We survey and reimplement the best published techniques for GPU-based Path Tracing, Bidirectional Path Tracing, and Progressive Photon Mapping, so we can analyze the impact of both high-level and low-level optimizations on their performance. We take advantage of the lessons learned from this investigation to develop the first GPU implementation of the recent Vertex Connection and Merging algorithm [Georgiev et al. 2012] (the algorithm has been independently developed by Hachisuka et al. [2012] as Unified Path Space, we use the former name), showing that even relatively complex light transport algorithms can be efficiently mapped on the GPU. In addition, we present new techniques that outperform the existing ones in most cases. For example, our Light Vertex Cache, a new approach to mapping connections of sub-path vertices in Bidirectional Path Tracing on the GPU, outperforms the state-of-the-art implementa-

tions by 30-60%. With the implementation of the aforementioned algorithms in a single system, we provide a detailed comparison of their performance on scenes with various characteristics, and provide a comprehensive in-depth analysis of the findings.

Our work can have broad benefits in both research and industry, ranging from low-level implementation insights for the practitioner, to high-level implications for driving researchers' focus towards finding algorithms and implementation techniques that improve upon the plots shown in our results.

2. RELATED WORK

Our work focuses on progressive Monte Carlo methods that converge to the solution of the Rendering Equation [Kajiya 1986]. An overview of these methods along with the challenges associated with their GPU implementation is given in Section 3. In the rest of this section, we discuss related work on other GPU-based global illumination approaches, as well as on ray shooting.

Realtime Global Illumination. A large volume of work has focused on realtime global illumination on the GPU. To achieve this ambitious goal, the methods generally offer only an approximate light transport solution. Because our focus is on algorithms that converge to the exact solution, we refer to [Ritschel et al. 2012] for an overview of the realtime methods.

Many-light methods are an important class of real-time global illumination algorithms based on Instant Radiosity [Keller 1997]. These methods first distribute a number of virtual lights into the scene, approximating the global illumination, and then each pixel is illuminated by a selected subset of the lights. Please refer to [Křivánek et al. 2012; Dachsbacher et al. 2013] for a review.

Ray Shooting. At the heart of virtually every light transport algorithm is a ray shooting query, both in the form of finding the closest intersection along a given ray, and testing visibility between two points. The research in this area focuses on three main issues: selection of a suitable acceleration structure, algorithms for their effective construction, and algorithms for their effective traversal.

The initial research on GPU ray shooting focused on overcoming the hardware limitations of then current GPUs by using specially adapted data structures and traversal algorithms [Carr et al. 2002; Foley and Sugerman 2005; Popov et al. 2007]. More recently, Aila and Laine [2009; 2012] showed how to approach the theoretical peak ray casting performance using the Spatial Bounding Volume Hierarchies (SBVH) [Stich et al. 2009], which became the structure of choice for GPU. A large body of work focuses on fast construction of the acceleration structures directly on the GPU, to allow interactive rendering of dynamic geometry. Most of this research focuses on BVHs [Lauterbach et al. 2009; Pantaleoni and Luebke 2010; Hou et al. 2011; Karras and Aila 2013], uniform grids [Kalojanov and Slusallek 2009] and kd-trees [Zhou et al. 2008]. We refer the reader to [Havran 2000; Aila et al. 2013] for an overview of CPU-based acceleration structure construction.

Well-established high-performance ray shooting solutions are publicly available. These libraries include Intel's Embree [Woop et al. 2013] (aimed at more traditional SIMD architectures), NVIDIA's OptiX [Parker et al. 2010], and the software framework of Karras et al. [2012], which is the basis of our implementation.

3. OVERVIEW

Tracing full light transport paths is the most important building block for all the investigated light transport algorithms. In Section 4, dedicated to **Path Tracing** [Kajiya 1986], we focus on finding the best implementation of this building block. While Purcell et al. [2002] deals with mapping Path Tracing to the limited programmability of then current GPUs, focus of the more recent work is on the full utilization of the GPU. The main challenge of efficient Path Tracing implementation is reduction of thread divergence within warps, caused by paths of different lengths; a few threads are processing the long paths, while other threads are idle. Novák et al. [2010] propose to use path regeneration, where persistent threads whose path has already terminated are assigned a new path from a larger pool of paths. Van Antwerpen [2011a] improves on this approach by compacting the paths so all regenerated paths are processed in a contiguous block, further increasing thread coherence. We examine their approaches, determine their best kernel configurations for the current GPU architectures, and test them against simple, yet previously unpublished, single-kernel implementations.

Section 5 focuses on the main challenge in implementing **Bidirectional Path Tracing** [Lafortune and Willems 1993; Veach and Guibas 1994; Veach and Guibas 1995], that is, efficient evaluation of camera and light sub-path vertex connections. The key GPU challenge is storing and connecting sub-paths of varying lengths. Novák et al. [2010] address this issue by modifying the basic algorithm to limit the light sub-paths to a maximum length of 5. Van Antwerpen [2011b] also modifies the algorithm and avoids storing sub-paths by retracing a full light sub-path for each segment of the camera sub-path, so only one segment has to be stored at a given time. His other approach [van Antwerpen 2011a] uses a user-defined maximum path length to conservatively pre-allocate memory for the sub-paths. In this section we evaluate these modifications, as well as propose a new algorithm, based on the Light Vertex Cache, which allows for a simple implementation and outperforms the current state-of-the-art by 30-60%.

In Section 6, we investigate methods based on **Photon Mapping**. Spatial data structures used to accelerate photon map queries are the main challenge and the focus of our investigation. To avoid the prohibitive cost of transfering all the photons to the CPU, it is necessary to build the acceleration structure directly on the GPU. Zhou et al. [2008] show that it is possible to build kd-trees at interactive rates. However, this approach exhibits random memory access patterns, which are suboptimal for the GPU. Alternatively, grids can be used both for k-NN and range queries [Purcell et al. 2003; Hachisuka et al. 2008]. Hachisuka and Jensen [Hachisuka and Jensen 2010] also propose the Stochastic Hash Grid, which avoids an explicit construction phase at the cost of storing only a subset of the generated photons. We examine and evaluate these acceleration structures, and provide some optimizations to further accelerate the queries.

Using the experience gathered above allows us to present in Section 7 the first GPU implementation of the recently introduced **Vertex Connection and Merging** algorithm [Georgiev et al. 2012], which combines both Bidirectional Path Tracing and Progressive Photon Mapping in a common framework and allows for efficient handling of a wide range of lighting effects.

Having implemented many of the state-of-the-art light transport simulation algorithms, we have a unique opportunity to compare them with each other. In Section 8 we compare the convergence graphs of the best implementation of each of the introduced algorithms on a set of six diverse scenes, explain the behavior of each algorithm with respect to the scene characteristics and draw conclusions as to which algorithm is best suited for which type of scene.

Summary of Challenges Associated with GPUs. There are three main challenges in efficient implementation of light transport algorithms on the GPU. First is code divergence within a warp which can greatly reduce the GPU utilization due to the SIMD na-

ture of execution units. Second, the memory management possibilities from within the GPU code are limited at best and GPU algorithms have to have their memory requirements known prior to kernel execution. Efficient GPU utilization requires a large number (several thousand) threads executed in parallel, which prevents generous allocation of per-thread memory as a solution to the memory management challenge. Third, to utilize the full memory bandwidth of the GPU, the accesses from a single SIMD execution unit should be coalesced, that is, access neighboring addresses. This has wide implications across all algorithms, e.g., Path Tracing performance benefits from work compaction, which results in the coherent primary rays being executed together in a compact block of warps.

3.1 Terminology

CUDA. As the paper discusses algorithm efficiencies that depend on hardware specifics, let us briefly introduce some concepts we will be referring to. We will introduce these concepts for NVIDIA's CUDA platform, however many are universal for any architecture using a wide SIMD model (e.g., Xeon Phi or AMD Radeon).

The basic execution unit of CUDA is a *thread*, which executes scalar code called a *kernel*. Each thread is allocated a certain number of *registers* and a certain amount of *local memory*. Local memory is used when the executed code requires more registers than are available, and is allocated only for the currently executed threads. Threads are grouped into *warps* of 32 threads. All threads in each warp execute the same instruction. When threads in a warp need to execute different branches of code, all threads have to execute all the code; masks are used to make sure the results are used only for threads that actually should have executed the code. This is called code divergence, and has negative impact on efficiency proportional to the size of code in the different branches. *Global memory* is memory that can be seen by both the CPU and GPU. It contains all inputs and outputs of a kernel call, including all intermediate data between consecutive kernel calls. The GPU has an *atomic counter* primitive, which we use for dealing with variable-sized inputs and outputs (e.g. queues) and for compaction. We have not observed any contention problems due to many threads simultaneously incrementing the same counter.

Images and frames. All of the presented algorithms are progressive in nature. For clarity, we distinguish between a *frame*, the result of a single progressive iteration, and an *image*, the final result obtained by averaging multiple frames. Raw low-level performance is compared on a single frame as a basic workload unit, while higher level comparisons between different Monte Carlo estimators or completely different algorithms measure error between images and a reference solution.

3.2 Testing Setup

We have implemented our algorithms on top of a publicly available implementation of GPU ray casting [Karras et al. 2012]. We use their acceleration structure as well as ray casting core and claim no contributions in these areas. We support environment illumination, point lights, directional lights, and area lights; our BSDFs include reflection and refraction, diffuse textures, glossy lobes from Kelemen, Ward, Ashikhmin-Shirley and Phong models, and Fresnel-weighted combinations of diffuse and glossy components. We use Tiny Encryption Algorithm to generate random numbers [Zafar et al. 2010]. While not used in this survey, our design also supports using low-discepancy sequences.

For our tests, we chose six scenes (Figure 1), representing various configurations found in practical applications. All are rendered in 720p resolution (1280x720), i.e., roughly 1 megapixel.

All our tests have been performed on a computer with Intel i7-3770K @ 3.50GHz and 16GB RAM. We test on two different NVIDIA GPUs of two architecture generations: Gainward Phantom GeForce GTX 580 3GB (Fermi architecture [NVIDIA 2011]), and Gainward GTX 680 4GB (Kepler architecture [NVIDIA 2012b]). We note that while the GTX 680 is a newer card and has higher theoretical FLOPS, the architecture is significantly different from GTX 580 and some of these differences are adversarial to our algorithms. For example, the global memory access is no longer cached by L1 cache but only in L2, and the clock rate has been decreased (from 1566 MHz to 1072 MHz for our cards).

4. PATH TRACING

We first focus on Path Tracing [Kajiya 1986], one of the simplest and most well-understood light transport simulation algorithms. All implementations discussed represent different mappings of the same algorithm onto the GPU. As tracing paths is an essential building block for all algorithms introduced in the later sections, good understanding and optimization of Path Tracing has significant impact on their performance.

4.1 Algorithm Overview

Path Tracing generates path samples by simulating a random walk through a scene. A path starts with a primary ray at the camera. It is traced into the scene and on each surface hit the path is extended into a random direction. To increase efficiency and prevent infinitely long paths, Russian roulette at each path vertex randomly determines whether the path will be extended or terminated. The survival probability is commonly based on surface reflectance (in the range 0-1).

In the basic algorithm described by Kajiya [1986], a path contributes to the frame only when it randomly hits a light. However, this is inefficient and the algorithm is almost always used with next event estimation (direct illumination). At each hitpoint, in addition to path extension, a random light is sampled and, if visible from the hitpoint, its contribution is accumulated. With this strategy, lights of finite extent can be sampled in two ways: direct connection or random hit; these two strategies are combined using Multiple Importance Sampling (MIS) [Veach and Guibas 1995].

4.2 Survey of Existing GPU Implementations

All implementations introduced in this section use multiple kernels. For clarity, we make this information part of the implementation name, even though the original names did not include it.

Naive Path Tracing (multiple kernels). To motivate the discussion on previous work, let us first consider the GPU implementation of *Naive Path Tracing (multiple kernels)* (Algorithm NaivePTmk). All paths used to obtain a frame (usually one path per pixel) are processed in parallel. The implementation uses one thread for each path, each path keeps its current state in global memory. In kernel (a), all paths are initialized and their primary rays are generated. While there is any active path, all active paths are extended (kernel (b)) and their next event estimation is evaluated (kernel (c)).

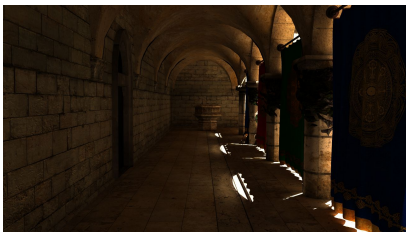
It is important to note that because of SIMD, even inactive threads (i.e. terminated paths) have to be executed as long as there is at least one active thread in their warp. If Russian roulette terminates 50% of active paths after each path extension, the utilization of GPU will be 100% during the first extension, then drop to 50% for the second extension, 25% for the third and so on, which is clearly detrimental for the overall performance.

(a) CoronaRoom (680k triangles): a living room. The illumination comes mostly from sun and sky, the lamp on the left provides little illumination besides the spots on the wall. The windows do not contain glass. Scene courtesy of Ludvík Koutný (`http://raw.bluefile.cz/`).

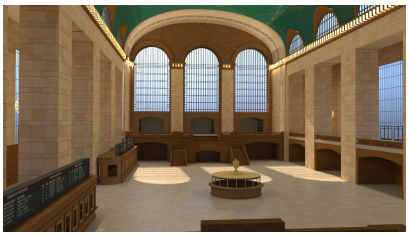
(b) CoronaWatch (918k triangles): the watch has a bezel made of highly glossy metal, glass with specular reflections and refractions, and a black dial with diffuse numbers. Illumination is provided by several large area lights. Scene courtesy of Jerome White.

(c) LivingRoom (783k triangles): a room seen in a mirror. The objects on the table are illuminated by two small lamps next to the mirror, more lights are at the other side of the room. The major feature of the scene is the caustics on the table, reflected in the mirror. Scene courtesy of Iliyan Georgiev.

(d) BiolitFull (166k triangles): an office scene illuminated solely by area lights enclosed in diffuse tube-like fixtures. Only the spots directly beneath and above the fixtures are directly illuminated, and they act as secondary light sources. Scene courtesy of Jiří "Biolit" Friml (`http://biolit.wordpress.com/`).



(e) CrytekSponza (262k triangles): a modified version of the classic Sponza. The camera is in one of the arcades on the ground floor, the only illumination is coming around the drapes from a strongly illuminated atrium. Scene courtesy of CryTEK (`http://www.crytek.com/cryengine/cryengine3/downloads`).



(f) GrandCentral (1527k triangles): a large open hall illuminated by an environment map and over 900 point lights. Each of the 200 alcoves near the ceiling contains one point light. The remaining point lights are on the chandeliers in the side halls. Scene courtesy of Cornell University Program of Computer Graphics (`http://www.graphics.cornell.edu/`).

Fig. 1: Our test scenes.

Path Tracing with Regeneration (multiple kernels). To address this issue, Novák et al. [2010] propose *Path Tracing with Regeneration (multiple kernels)* (Algorithm RegenerationPTmk). This implementation decouples threads from paths. It uses a fixed pool of threads. Each thread processes one path at a time. When a thread has no assigned path or its path has terminated, we say it is idle, otherwise it is active. All threads are idle at the start. While there are any active threads and the path queue is not empty, all idle threads are assigned a new path from the path queue (kernel (a)), all paths are extended (kernel (b)), and their shadow ray is cast (kernel (c)). This way, all threads on the GPU are active until the queue becomes empty. Another advantage is that the path state is kept only for currently processed paths, so the required memory depends only on the number of threads, not the number of paths.

Stream Path Tracing with Regeneration (multiple kernels). Van Antwerpen [2011a] introduces *Stream Path Tracing with Regeneration (multiple kernels)* (Algorithm StreamingPTmk) to improve upon the previous approach. The inefficiency comes from code divergence in kernel (a) of Algorithm RegenerationPTmk. When at least one thread in a warp needs to regenerate its path, all threads in the warp have to execute the kernel, even though the other threads do not need regeneration.

Van Antwerpen proposes to use stream compaction [Sengupta et al. 2007] to separate the threads into active and idle threads (Figure 2). This way, at most one warp can have both active and idle threads, essentially removing the code divergence. Another advantage is that the coherent primary rays of the new paths will be assigned to threads that will be executed together, and it has been shown that ray coherence has positive effect on ray casting performance [Wald et al. 2001]. Similarly, the shadow rays for next event estimation are compacted to reduce code divergence in kernel (c).

The compaction is a part of kernel (b), avoiding separate compaction kernel calls. It uses two sets of threads, one as input and the other as output, and a global atomic counter, initially set to zero. For each active thread in the input set, the counter is increased by one and its old value is used as the thread's target position in the output set.

Wavefront Path Tracing. Laine et al. [2013] analyze Path Tracing in the cases when BSDFs are expensive to evaluate (e.g., surface characteristics described by complex noise functions). Such situations can lead to extreme code divergence. Their solution separates BSDF evaluation (for both next event estimation and continuation sampling) into a separate kernel call, sorts paths based on their BSDF and executes the BSDF kernels in a coherent fashion. However, this technique is only effective for these expensive BSDFs. For simpler BSDFs, such as those used in our test scenes, the overhead of extra kernel calls and sorting greatly outweights any

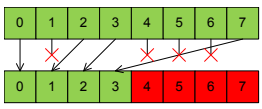


Fig. 2: *Compaction*: Active paths are compacted (green), while terminated paths are discarded (red).

```
(a) kernel path in all paths:                          // Generate
        path.ray = setup primary ray for path

    while Any path active:
(b)     kernel path in all paths:                      // Extend path
            if path.termianted: return
            trace path.ray
            if no hit:
                accumulate background color
                path.terminated = true
            else:
                accumulate surface emission
                compute contribution of a random light
                path.directIllum = (shadowRay, contrib)
                if terminated with Russian roulette:
                    path.teminated = true
                else: path.ray = sample BSDF
(c)     kernel path in all paths:                      // Shadow test
            if path.directIllum.contrib ≠ 0:
                if path.directIllum.shadowRay not blocked:
                    accumulate path.directIllum.contrib
```

Algorithm NaivePTmk: *Naive Path Tracing (multiple kernels)*: Naive GPU implementation of Path Tracing. All paths are processed in parallel, each path is assigned to one thread. Kernel (a) generates a primary ray for each path, and kernels (b) and (c) perform path extension and shadow test, respectively, until all paths have terminated.

```
    // All threads marked as idle
    // All paths in pathQueue
    while pathQueue not empty and any thread not idle:
(a)     kernel thread in all threads:                  // Regenerate
            if thread.state == idle:
                thread.path = next path in queue
                thread.ray = setup primary ray for thread.path
                thread.state = active
(b)     kernel thread in all threads:                  // Extend path
            if thread.state == idle: return
            trace thread.ray
            if no hit:
                accumulate background color
                thread.state = idle
            else:
                accumulate surface emission
                compute contribution of a random light
                thread.directIllum = (shadowRay, contrib)
                if terminated with Russian roulette:
                    thread.state = idle
                else: thread.ray = sample BSDF
(c)     kernel thread in all threads:                  // Shadow test
            if thread.directIllum.contrib ≠ 0:
                if thread.directIllum.shadowRay not blocked:
                    accumulate thread.directIllum.contrib
```

Algorithm RegenerationPTmk: *Path Tracing with Regeneration (multiple kernels)*: This algorithm is almost identical to Algorithm NaivePTmk, but it decouples threads from paths. Kernel (a) now resides within the main while loop, and initializes new path from the path queue for any thread that is idle. This reduces the number of idle threads in each loop and increases GPU utilization. Kernels (b) and (c) are almost identical, with the difference that intermediate data are now stored with the thread rather than with the path.

gains from increased execution coherence and they recommend executing all such BSDFs in a single kernel call.

4.3 Proposed Alternative Implementations

All of the presented implementations launch multiple kernels, at least one per path extension. This approach has several potential bottlenecks: kernel launch overhead, path state stores and loads, and the fact that the number of active paths has to be communicated to the CPU. We investigate confining the whole algorithm into a single kernel launch, which naturally removes all three potential bottlenecks.

Naive Path Tracing (single kernel). We propose *Naive Path Tracing (single kernel)* (Algorithm NaivePTsk) as a simpler alternative to NaivePTmk. All three kernels and the while loop are combined into a single large kernel, giving us code which is essentially the same as a standard CPU path tracer. While, in theory, all paths are still processed in parallel, the execution specifics of CUDA impose some degree of serialization. Let us assume that the number of paths to be traced is significantly larger than the number of threads that can be processed by the GPU at once. In such a case, the GPU schedules threads up to its capacity, these threads process their paths, and when all threads within a scheduling unit (*block* in CUDA) have terminated, new threads are scheduled, resulting in path regeneration on a coarser level than individual threads.

This approach has several advantages. Path state does not have to be explicitly stored and loaded, as it is kept in thread local memory all the time (which benefits from L1 cache even on the Kepler architecture of the GTX 680 GPU [NVIDIA 2012a]). This also means that there is no need to allocate any per-path memory on the GPU, and the memory footprint is governed solely by the number of concurrently executed threads. Only one kernel is launched, effectively removing any impact that kernel execution overhead has on the overall performance, and when this kernel terminates we know that all paths have terminated.

Path Tracing with Regeneration (single kernel). To explore per-thread regeneration in the single-kernel setting, we introduce *Path Tracing with Regeneration (single kernel)* (Algorithm RegenerationPTsk). We use persistent threads, where the number of threads is set to the GPU capacity for concurrent threads. Path (re)generation is again moved into the while loop, and threads that do not have a path are assigned one from a path queue.

While code divergence in the regeneration step is still an issue, compaction is no longer an option. The use of two sets of threads would require a global barrier to swap the sets. However, such barriers are currently not supported, so we did not explore this option any further.

Using a single-kernel implementation has one potential drawback. Suppose that the separate kernels in a multi-kernel implementation have significantly different register requirements. The number of registers influences the number of threads a GPU can pro-

```
// Two thread pools threadsIn, threadsOut
// All threadsOut marked as idle
// All paths in pathQueue
// atomics: pathCount = 0, directCount = 0
while pathQueue not empty and any path active:
(a)  kernel thread in all threadsOut:                    // Regenerate
       if thread index ≥ pathCount:
         thread.path = next path in queue
         thread.ray = setup primary ray for thread.path
         thread.state = active
     // Swap threadsIn ↔ threadsOut
     // pathCount = 0, directCount = 0
(b)  kernel thread in all threadsIn:                     // Extend path
       if thread.state == idle: return
       trace thread.ray
       if no hit:
         accumulate background color
         thread.state = idle
       else:
         accumulate surface emission
         compute contribution of a random light
         if contrib ≠ 0:
           index = directCount++
           threadsOut[index].directIllum =
               (thread.pixel, shadowRay, contrib)
         if not terminated with Russian roulette:
           thread.ray = sample BSDF
           index = pathCount++
           threadsOut[index] = thread
(c)  kernel thread in all threadsOut:                    // Shadow test
       if thread index < directCount:
         if thread.directIllum.shadowRay not blocked:
           accumulate thread.directIllum.contrib to
               thread.directIllum.pixel
```

Algorithm StreamingPTmk: *Streaming Path Tracing with Regeneration*: Similar to Algorithm RegenerationPTmk, but threads do not "own" their path for its entire lifetime. Instead, paths that are still active are compacted to threads with low index. Atomic counter pathCount contains the current number of active paths. Two sets of threads, threadsIn and threadsOut are used for compaction. In kernel (a), first "pathCount" threads in the threadsOut set contain active paths, and paths are regenerated for all the remaining threads. The two sets are swapped, the "pathCount" counter is reset, and kernel (b) processes all threads from threadsIn. Paths that are not terminated are compacted to the threadsOut set. Direct illumination with non-zero contribution is handled in the same way. Note that a thread can now handle path extension and shadow test for different pixels.

```
kernel path in all paths:
(a)  path.ray = setup primary ray for path               // Generate
     while path.terminated == false:
       trace path.ray
       if no hit:
         accumulate background color
         path.terminated = true
       else:
         accumulate surface emission
         compute contribution of a random light
(c)      if contribution ≠ 0:                            // Shadow test
           if shadowRay not blocked:
             accumulate contribution
(b)      if terminated with Russian roulette:
         // Ext. path
           path.teminated = true
         else: path.ray = sample BSDF
```

Algorithm NaivePTsk: *Naive Path Tracing (single kernel)*: This is a single-kernel version of Algorithm NaivePTmk. All path states are kept in local memory and only for threads currently executed on the GPU, reducing the required memory footprint. The code is greatly simplified and essentially identical to a standard CPU implementation.

```
// All threads marked as idle
// All paths in pathQueue
kernel thread in all threads:
  while pathQueue not empty:
(a)  if thread is idle:                                  // Regenerate
       thread.path = next path in queue
       thread.ray = setup primary ray for thread.path
       thread.state = active
     trace path.ray
     if no hit:
       accumulate background color
       thread.state = idle
     else:
       accumulate surface emission
       compute contribution of a random light
(c)    if contribution ≠ 0:                              // Shadow test
         if shadowRay not blocked:
           accumulate contribution
(b)    if terminated with Russian roulette:
       // Ext. path
         thread.state = idle
       else: thread.ray = sample BSDF
```

Algorithm RegenerationPTsk: *Path Tracing with Regeneration (single kernel)*: This is a single-kernel version of Algorithm RegenerationPTmk, utilizing persistent threads. When a thread has no path assigned, it is given a new path from the queue, and processes the path until its termination.

cess concurrently, which in turn influences the overall performance. The first kernel would have three times as many active threads as the second kernel. When such kernels are combined into a single kernel, the GPU cannot use more threads for the more lightweight steps of the code and is underutilized in those steps.

We acknowledge that both are straightforward implementations of Path Tracing, and none, in itself, is a major contribution. However, comparing these simpler implementations with the the ones

Algorithm	RegenerationPTmk						StreamingPTmk					
GPU	GTX 580			GTX 680			GTX 580			GTX 680		
Kernels	1	2	3	1	2	3	1	2	3	1	2	3
CoronaRoom	23.3	36.4	**42.7**	20.8	**31.3**	29.5	50.0	**55.9**	52.4	48.9	**51.4**	40.6
CoronaWatch	52.3	54.8	**55.9**	**35.8**	34.5	34.0	71.9	**78.1**	68.3	**51.6**	49.5	42.6
LivingRoom	30.4	39.1	**43.9**	27.6	**34.1**	33.5	55.0	**59.6**	55.7	56.0	**57.1**	49.0
BiolitFull	27.9	41.0	**49.5**	24.5	**35.0**	33.5	60.5	**68.9**	66.0	61.0	**65.0**	53.0
CrytekSponza	36.8	58.8	**73.4**	31.9	**48.3**	44.6	95.1	**95.4**	88.2	74.5	**74.8**	52.8
GrandCentral	20.1	33.0	**37.9**	19.2	**29.3**	28.1	43.4	**49.1**	46.4	44.0	**47.3**	38.6
Units	millions of rays per second (more is better)											

Table I. : Performance, in millions of rays per second, of RegenerationPTmk, i.e. Path Tracing with Regeneration (multiple kernels), and StreamingPTmk, i.e. Streaming Path Tracing with Regeneration (multiple kernels), in different kernel configurations.

introduced in previous work is beneficial and will lend us useful insights.

4.4 Results and Discussion

To configure the implementations, i.e., to set the required number of registers and the size of the thread pool used by the implementations with regeneration, we have measured all possible configurations and used the one that resulted in the highest performance. The optimal number of registers greatly varies between both the individual implementations and the GPU architectures with the difference between the best and the worst in the the tested range (32 – 63 registers) being up to 2×. As a result, we cannot give a summary advice and only recommend always conducting performance tests for each reimplementation.

Memory requirements. All implementations require only a few megabytes of local memory for the active thread variables that do not fit into registers. Multi-kernel implementations require additional global memory to store the input of individual kernels. This translates to less than 100 MB per frame in all methods.

Kernel configurations. We tried several configurations with different separation of tasks into kernels. First, we tried to separate the ray casting part of kernel (b) into a separate kernel, to better utilize dedicated ray casting kernels from [Aila et al. 2012]. This however dropped the performance to less than 40% on both GTX 580 and GTX 680, compared to the three-kernel variant. The bottleneck was in the increased loading and storing of path data between kernel runs, suggesting that further increases in the number of kernels would not bring any benefits.

Going the opposite way, we reduced the number of kernels to two, by making the Regeneration kernel (a) part of the Extend path kernel (b). This saves one set of loads and stores, at the cost of lower GPU utilization in the regeneration step. Importantly, the effectivity of compaction is preserved.

We also tried reducing to just a single kernel, by folding in the Shadow test kernel (c). Here we save another load and store, this time at the cost of losing the benefits of compaction. Note that this is still different from RegenerationPTsk, which runs only one kernel for each frame, while RegenerationPTmk with one kernel runs this kernel for each path vertex.

Table I shows the performance, in millions of rays per second, of RegenerationPTmk and StreamingPTmk with one, two, and three kernels, on both GTX 580 and GTX 680. Starting with StreamingPTmk, it is quite clear that the two-kernel configuration is the best, having superior performance in almost all scenes. The three-kernel configuration on GTX 680 performs significantly worse than the one- and two-kernel configurations, mainly due to changes in memory system and caching of global memory accesses.

The situation with RegenerationPTmk is less clear. On GTX 580, the three-kernel variant clearly outperforms the other two configurations. However, this changes for GTX 680, where the lower number of loads and stores into global memory results in a slight advantage of the two-kernel configuration. One of the reasons for the difference between two and three kernels in StreamingPTmk and RegenerationPTmk is that StreamingPTmk performs non-coalesced accesses in the compaction phase, making it more sensitive to the missing L1 cache. For the following performance analysis, we use the optimal RegenerationPTmk and StreamingPTmk kernel configuration for their respective GPU (that is, we use three-kernel RegenerationPTmk for GTX 580, two-kernel RegenerationPTmk for GTX 680, and two-kernel StreamingPTmk for both).

Performance tests. For each of the introduced implementations we measured performance (in rays per second) for different numbers of paths per frame. We tested from 330 thousand to 100 million paths per frame, which covers a wide range of desired applications, from 1 path per pixel at resolution of 640×480 to 100 paths per pixel at resolution of 1280×720. The individual per-scene measurements, to be found in the supplemental material, have been aggregated in Figure 3.

Let us first look at Figure 3a (GTX 580). Here, StreamingPTmk, is the clear winner across all path counts. Its base performance at 10^6 paths per frame is increased by another 15% for 10^7 paths, making almost it 50% faster than RegenerationPTsk and

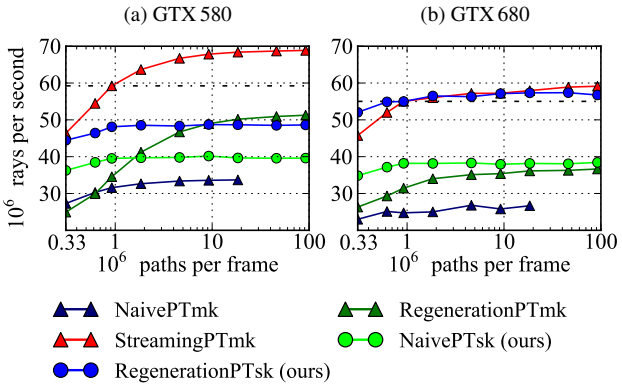


Fig. 3: *Path tracing performance.* Performance in rays per second with increasing number of paths per frame, averaged across all six test scenes.

RegenerationPTmk, which compete for being the second fastest. The comparison of RegenerationPTsk and RegenerationPTmk, respectively, shows that our RegenerationPTsk is less sensitive to the size of workload, keeping stable performance from approximately 10^6 paths per frame. RegenerationPTmk can still outperform the single-kernel implementation, but requires 10^7 paths or more, and even then the difference is rather marginal. Both naive implementations, NaivePTmk and NaivePTsk, exhibit low performance. As NaivePTmk requires allocated memory for each path of the frame, it could not be tested in the full range. We note that for $2 \cdot 10^6$ paths per frame NaivePTsk is faster than RegenerationPTmk, witnessing of the overhead of global memory stores and loads.

For GTX 680 (Figure 3b) the story changes. Both StreamingPTmk and our RegenerationPTsk perform roughly the same, with RegenerationPTsk having more stable and StreamingPTmk slightly higher peak performance. This is caused by the difference in the memory subsystem on the Kepler architecture of the GTX 680 GPU. While StreamingPTmk benefits from compaction and slightly better GPU utilization, our RegenerationPTsk has the benefit of storing intermediate data in the L1-cached local memory instead of global memory. For similar reasons NaivePTsk is faster than RegenerationPTmk in all cases.

4.5 Conclusion

Lower numbers of larger kernels benefit both Fermi (GTX 580) and Kepler (GTX 680) architectures. The disadvantage of more loads and stores outweighs gains from the optimal number of concurrently executed threads for a given step. The relative performance of the measured kernel configurations of RegenerationPTmk differs between the architectures, with three-kernel configuration being the fastest on GTX 580, and two-kernel (joined Regeneration and Extend path kernels) on GTX 680. For StreamingPTmk, the two-kernel configuration is the fastest on both architectures.

With regard to the performance on GTX 680, our Path Tracing with Regeneration (single kernel) (RegenerationPTsk) and Streaming Path Tracing with Regeneration (multiple kernels) (StreamingPTmk) have similar performance, but our implementation is faster for low number of paths per frame, as well as simpler to implement. This changes on the older GTX 580, where StreamingPTmk is the optimal implementation for all but the lowest number of paths per frame, with its peak performance being over 50% faster than our RegenerationPTsk, which competes with its multi-kernel variant for the second place.

5. BIDIRECTIONAL PATH TRACING

While Path Tracing is sufficient for simpler open scenes, scenes with more complex indirect illumination (e.g., BiolitFull) greatly benefit from more advanced Bidirectional Path Tracing (BPT) [Lafortune and Willems 1993; Veach and Guibas 1994]. The algorithm itself is more complicated than Path Tracing and as such opens opportunity for a different type of optimization. In Path Tracing we focused only on the very low-level mapping of a single algorithm onto GPU. Here, on the other hand, we examine options of modifying the underlying Monte Carlo estimator (and thus the algorithm itself) so it lends itself better to a GPU implementation.

5.1 Algorithm Overview

Bidirectional Path Tracing, as originally described by Lafortune and Willems [1993] and Veach and Guibas [1994], generates, for each image sample, two separate sub-paths: one starting at the camera, and one at a light (Fig. 4). The first vertex of each sub-path is located directly on the camera or on a light, respectively. The sub-paths are extended, by adding one vertex at a time, in the same way as in Path Tracing. After the two sub-paths have been generated, each vertex of the camera sub-path is connected to each vertex of the light sub-path, forming full paths (connecting camera to light). We can view this as a generalization of Path Tracing with next event estimation, in which the light sub-path had always just a single vertex directly on the light source. As there are multiple ways to construct each full path from light to camera, the paths are weighted using Multiple Importance Sampling (MIS) [Veach and Guibas 1995].

The original formulation of MIS by Veach and Guibas [1995] assumes that when two vertices are connected, all vertices on both sub-paths preceding the connected vertices have to be accessed to gather required data to compute the appropriate MIS weight. Recursive Multiple Importance Sampling (MIS), introduced in [van Antwerpen 2011b; Georgiev 2012], removes this requirement and allows computing the MIS weight from information stored only in the vertices being connected. This is especially important for GPU implementation, where random memory accesses should be limited, and all presented implementations use this method.

Connecting each camera sub-path vertex to all the vertices on the light sub-path introduces two new GPU implementation issues that have to be addressed. First, where PT has a fixed memory footprint per path, the memory requirements in BPT depend on the length of the light sub-path, as the whole light sub-path has to be traced and stored before the camera sub-path can be started. While the average length of a path is not high, this storage has to be multiplied by the number of parallel threads. Second, unlike PT, the work required per camera sub-path vertex depends on light sub-path length and can be vastly different for different camera sub-paths, which makes efficient mapping to GPU more complicated.

5.2 Survey of Existing GPU Implementations

Bidirectional Path Tracing with Regeneration (RegenerationBPT). The first fully GPU-based implementation was introduced in [Novák et al. 2010]. It uses two separate passes. In the first pass, all light sub-paths are generated and stored in the GPU memory. In the second pass, camera sub-paths are created and traced as in Path Tracing with Regeneration, except that each vertex is also connected to all vertices of a randomly chosen light sub-path. To address the memory issue, the authors limit light sub-path length to five vertices. Limiting maximum light sub-path length also requires more complex logic for computing MIS weights and the authors therefore did not use MIS, which has a negative impact on the final image quality.

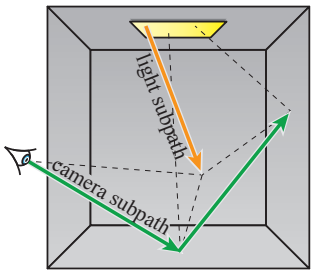


Fig. 4: The standard Bidirectional Path Tracing sample consists of a camera sub-path (green) and a light sub-path (orange), where each vertex on the camera sub-path is connected to each vertex on the light sub-path (dashed).

```
while pathQueue not empty and any path active:
    foreach thread in all threads:
        if thread is idle:
            thread.camera = setup camera path
            thread.light = setup light path
    CompletedPaths = 0
    while CompletedPaths < 60%:
        foreach thread in all threads:
            if thread.light not terminated:
                Extend thread.light
                thread.lightVertices += light vertex
            if thread.camera not terminated:
                Extend thread.camera
                thread.cameraVertices += camera vertex
            if thread.light and thread.camera terminated:
                CompletedPaths++
    foreach thread in all terminated threads:
        Generate all lightVertices and cameraVertices pairs
        foreach vertex pair in thread:
            shadowRay = pair.light to pair.camera
            if shadowRay not occluded:
                accumulate contribution
```

Algorithm StreamingBPT: *Streaming Bidirectional Path Tracing with Regeneration*: All threads are initialized with a camera and light sub-path. Then a two stage algorithm is executed, where all sub-paths are extended in a similar way to Algorithm StreamingPTmk (details left out for brevity). When more than 60% threads have both sub-paths terminated, all pairs of vertices for each thread are generated (implicitly), and all such pairs have their visibility evaluated and contributions accumulated. All terminated threads are then regenerated, until there are no paths left in the queue.

Multi-path Bidirectional Path Tracing (MultiBPT). Both the memory consumption and the workload issues are solved by the algorithm introduced by van Antwerpen [2011b], originally under the name *Streaming Bidirectional Path Tracing*, which we changed to avoid confusion with a later introduced algorithm of the same name by the same author.

Instead of storing the whole light sub-path, the algorithm traces one complete camera sub-path for each light sub-path vertex, which requires storing only one light and one camera sub-path vertex. This naturally solves both the storage and the uneven load problem, at the cost of more camera paths; the algorithm essentially spends more time on "camera-side" effects (e.g., anti-aliasing) than on "light-side" effects (e.g., caustics). As tracing of both sub-paths is interleaved, an efficient implementation requires reusing the same code for both camera and light sub-paths, making the implementation quite involved.

Combinatorial Bidirectional Path-Tracing. Pajot et al. [2011] present a hybrid two-stage implementation of BPT. All sub-paths are generated on the CPU, and the GPU performs only connections between all camera and all light sub-path vertices. Unfortunately, this way some paths (e.g., caustic paths) can only be handled by the CPU. As this is not a pure GPU implementation, we are including it only for completeness and it does not appear in our comparison.

```
vertexCount = 0                                  // Preparation phase
foreach path in 10k light paths:
    while path not terminated:
        trace path.ray if no hit: return
        vertexCount += 1
        path = extend path
averageLength = vertexCount/10 k
LVCache = reserve |light paths| · averageLength · 1.1
connections = max(1, ⌈averageLength⌉)
vertexCount = 0                                  // Light trace
foreach path in light paths:
    while path not terminated:
        trace path.ray
        if no hit: return
        LVCache[vertexCount++] = path.vertex
        path = extend path
foreach path in camera paths:                    // Camera trace
    while path not terminated:
        trace path.ray
        if no hit: return
        repeat connections times:
            path connects to LVCache[random]
        path = extend path
```

Algorithm LVC-BPT: *Light Vertex Cache BPT* (proposed algorithm): In LVC-BPT we first, once for each scene, estimate average light path length (Preparation phase), reserve room in light vertex cache for the estimated total number of light sub-path vertices, plus a 10% safety margin, and estimate the number of connections for each camera sub-path vertex. We then execute two main stages of the algorithm. First, we trace all light sub-paths, storing the light sub-path vertices in the cache. Second we trace all camera sub-paths, connecting to the required number of random vertices in the cache.

Streaming Bidirectional Path Tracing with Regeneration (StreamingBPT). In contrast to his [2011b] algorithm, van Antwerpen [2011a] presents a more traditional BPT (Algorithm StreamingBPT). The approach is a two stage algorithm, using a pool of threads, with one thread processing one camera and light sub-path pair. Initially, all threads have their sub-paths generated. Both sub-paths are then extended, with their vertices stored with the thread. When more than 60% of the threads have both their sub-paths terminated, the algorithm enters the second stage, in which all pairs of light and camera sub-path vertices within each thread are tested for visibility and their contributions are accumulated to the image. The pairs are formed implicitly, and tested with one thread for each pair, which solves the issue with uneven work per vertex. After that, all terminated threads have their sub-paths regenerated and the whole algorithm is repeated until there are no more paths to be traced. The required storage size is determined by the size of the thread pool and the user-defined maximum path length.

5.3 Proposed Alternative: Light Vertex Cache BPT

To remove the requirement for user-defined maximum path length while keeping the implementation as simple as possible, we introduce Light Vertex Cache Bidirectional Path Tracing (Algorithm LVC-BPT). The key idea is that instead of connecting each camera sub-path vertex to all vertices from a given light sub-path,

Algorithm	Vertex Storage	Path extension	Shadow test	Other	On our configuration
StreamingBPT	$2 \cdot S \cdot L \cdot$ B	$4 \cdot S \cdot 64$ B	$S \cdot 44$ B	$S \cdot 100$ B	1.1 GB
MultiBPT	$S \cdot 100$ B	$2 \cdot S \cdot 64$ B	$S \cdot 44$ B	—	108MB
NaiveBPT	$L \cdot E \cdot 100$ B	—	—	—	8–10 MB
LVC-BPTsk (ours)	$P \cdot AL \cdot 100$ B	—	—	—	16–164 MB
LVC-BPTmk (ours)	$P \cdot AL \cdot 100$ B	$2 \cdot S \cdot 64$ B	$S \cdot (V+1) \cdot 44$ B	—	148–316 MB

S – thread pool size; E – number of concurrently executed threads
L – maximum path length; P – paths per frame; AL – average light sub-path length

Table II. : *Summary of BPT memory requirements*: We present the memory requirement of each component of each algorithm as a function of several parameters. We also give the total amount of memory the algorithm used on our configuration.

the vertex is connected to a given number of uniformly randomly chosen vertices across all light sub-paths. It can also be seen as first choosing a random light sub-path (similar to RegenerationBPT) with probability proportional to its number of vertices, and then choosing a uniformly random vertex on the path, which arrives at the same uniform probability for all light sub-path vertices.

This, along with the recursive MIS weight computation, enables us to store all vertices in a single global *Light Vertex Cache (LVC)*, without storing any information regarding the light sub-path they originate from. As all vertices are stored in a common cache, we do not need to know the maximum path length. Instead, we only need the average path length, to allocate a large enough cache. We estimate this by tracing a small number (ten thousand) of light sub-paths, only counting the number of vertices they would store. This kernel takes less than 1 ms on both tested GPUs, and has to be performed only once for each scene. Using the average path length, we compute the expected number of light sub-path vertices (adding a 10% safety margin) and reserve the required memory for the cache. In theory, it is possible that the algorithm will generate more light vertices than the LVC capacity, in which case we would discard the extra vertices (causing bias). However, this has not happened in any of our experiments.

The implementation of LVC-BPT is fairly straightforward and can be based on any of the algorithms introduced in Section 4. We present results based on Path Tracing with Regeneration (single kernel) (as LVC-BPTsk) and on Streaming Path Tracing with Regeneration (multiple kernels) (as LVC-BPTmk).

As the second pass of LVC-BPT accesses the cache in a random pattern, we load the vertices through texture units in Array of Structures (AoS) layout for optimal performance.

5.4 Results and Discussion

We test our implementations of the following five algorithms:

—StreamingBPT [van Antwerpen 2011a] represents the current state-of-the-art algorithm. To confirm that our performance is on par with the paper, we measured the number of samples (i.e., camera and light sub-path pairs) on the same scene and GPU as in the original paper and our implementation (8.66 million samples per second) was roughly twice as fast as reported in the original paper (3.64 million samples per second).

—MultiBPT [van Antwerpen 2011b]. We use a straightforward extension of the approach and complement it with a dual algorithm, tracing one light sub-path for each camera sub-path vertex. During the progressive rendering of the image, we alternate between the two algorithms, balancing the number of camera and light sub-paths.

—NaiveBPT is a straightforward port of CPU code to GPU, to compare the relative gain of the more advanced implementations. The implementation consists of two while loops of NaivePTsk (Alg. NaivePTsk) within a single kernel. Persistent threads are used for better control of memory requirements (see below).

—LVC-BPT is our new algorithm. Its two versions use either Path Tracing with Regeneration (for LVC-BPTsk) or Streaming Path Tracing with Regeneration (for LVC-BPTmk) as the basic algorithm for tracing camera and light sub-paths.

Memory requirements. Table II gives a summary of memory used by each of the algorithms as a function of several parameters. The state-of-the-art StreamingBPT uses the most memory. It uses two sets of threads and requires large storage for all light sub-path vertices, using up 1.1 GB. MultiBPT stores only one light sub-path vertex per thread, does not use two sets of threads for each sub-path, lowering the total memory requirements to 108 MB. NaiveBPT performs all its computation within a single kernel launch, so it does not require any extra memory for path extension and shadow test kernels, lowering memory requirements to only 8–10 MB. LVC-BPTsk also stores only light vertices, but the memory is given by the average light sub-path length and the total number of light sub-paths per frame. LVC-BPTmk again adds requirement for the path extension and the shadow test kernel stores.

Performance tests. In Bidirectional Path Tracing, the number of rays per second does not provide a good comparison between the algorithms. Instead, we measure performance as the time required to achieve a given image quality in terms of Root Mean Square Error (RMSE) with the respect to the reference solution (computed by NaiveBPT in 10 hours).

We performed measurements on both GTX 580 and GTX 680 with 10^6 samples per frame, and chose our target quality as RMSE achieved by the state-of-the-art StreamingBPT in 10 minutes on GTX 580. Table III shows the relative speedup against StreamingBPT on GTX 580. The average result is a simple average of the speedups for each given algorithm. Note that we do not use LivingRoom in this comparison, as the RMSE is dominated by the missing reflected caustics that none of the BPT methods can reasonably capture (Figure 5).

When we look at the results on GTX 580, we notice a surprisingly high performance of NaiveBPT. On GrandCentral it outper-

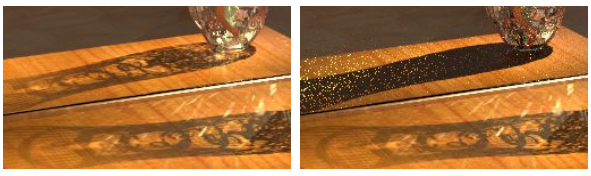


Fig. 5: Reflected caustic dominates the RMSE of LivingRoom scene. **Left**: Inlay from reference image. **Right**: Inlay from LVC-BPTsk after 15 min.

	GeForce GTX 580				
	StreamingBPT	NaiveBPT	MultiBPT	**LVC-BPTsk**	**LVC-BPTmk**
CoronaRoom	1.00×	0.91×	0.81×	1.15×	**1.21**×
CoronaWatch	1.00×	**1.41**×	0.93×	**1.41**×	1.15×
BiolitFull	1.00×	0.52×	0.57×	1.53×	**1.81**×
CrytekSponza	1.00×	1.07×	0.93×	**1.35**×	1.29×
GrandCentral	1.00×	0.70×	0.70×	1.29×	**1.34**×
Average	1.00×	0.85×	0.77×	**1.33**×	**1.33**×

	GeForce GTX 680				
	StreamingBPT	NaiveBPT	MultiBPT	**LVC-BPTsk**	**LVC-BPTmk**
CoronaRoom	0.86×	0.72×	0.69×	**1.24**×	1.19×
CoronaWatch	0.74×	1.17×	0.80×	**1.32**×	1.07×
BiolitFull	0.89×	0.42×	0.58×	1.73×	**1.92**×
CrytekSponza	0.84×	0.92×	0.92×	**1.55**×	1.12×
GrandCentral	0.97×	0.55×	0.68×	1.39×	**1.49**×
Average	0.85×	0.68×	0.71×	**1.38**×	1.27×

Table III. : *Relative BPT speed up*: Speedup of different BPT algorithms, in terms of time to a given quality, relative to StreamingBPT on GTX 580. The target quality is chosen as RMSE achieved by StreamingBPT in 10 minutes on GTX 580.

forms the StreamingBPT, and on CoronaWatch it is even tied for the fastest algorithm with our LVC-BPTsk. In these scenes, the work for each sample is highly uniform, which mitigates the inefficiencies of the naive approach. However, on average, the naive approach is about 15% slower than StreamingBPT. MultiBPT is the slowest of the algorithms, mainly due to its more complex implementation and imperfect interleaving of camera and light sub-path tracing. Both LVC implementations are faster than StreamingBPT on all scenes, with average speedup of 33%. A major factor is that, unlike StreamingBPT, LVC-BPT stores only light vertices, which comprise less than 40% of all vertices stored by StreamingBPT. The algorithm also benefits from a more straightforward control flow.

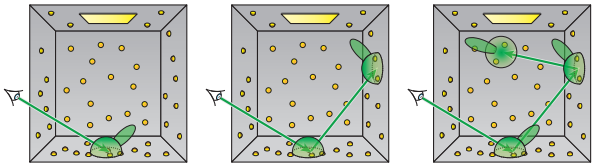
The results on GTX 680 are consistent with our findings from Section 4. We again see a drop in the absolute performance of multi-kernel implementations, significantly influenced by the lack of L1 cache for global memory accesses. Our LVC-BPTsk is the only algorithm that actually shows increase in performance on GTX 680; all other algorithms, including our LVC-BPTmk, have decreased performance.

With 10^7 samples per frame, the findings are again consistent with the findings from Section 4, and multi-kernel implementations (StreamingBPT, MultiBPT, and LVC-BPTmk) benefit from the increased number of samples more than single-kernel variants (NaiveBPT and LVC-BPTsk). The full results can be found in the supplemental material.

5.5 Conclusions

In this section we surveyed several Bidirectional Path Tracing algorithms. While NaiveBPT and StreamingBPT implement the standard BPT Monte Carlo estimator, only limiting the maximum path length, other approaches modify the estimator to achieve better GPU mapping. Our proposed LVC-BPT significantly simplifies the implementation by decoupling light and camera sub-paths.

We compared the state-of-the-art StreamingBPT with NaiveBPT, MultiBPT, and our LVC-BPT. As LVC-BPT can utilize almost any Path Tracing implementation, we measured two versions: LVC-BPTsk, based on Path Tracing with Regeneration (single kernel), and LVC-BPTmk, based on Stream Path Tracing with Regeneration (multiple kernels). We conducted a performance test, measuring time required to achieve a given image quality. The difference between LVC-BPTsk and LVC-BPTmk closely follows differences between their respective Path Tracing algorithms. The single-kernel implementation is simpler and more suited for GTX 680 and low numbers of samples per frame, while the multi-kernel implementation is slightly more involved and is more suited for GTX 580 and larger numbers of samples per frame. The simplicity of the LVC-BPT implementation allows it to outperform the other algorithms by 30-60% on all tested configurations.



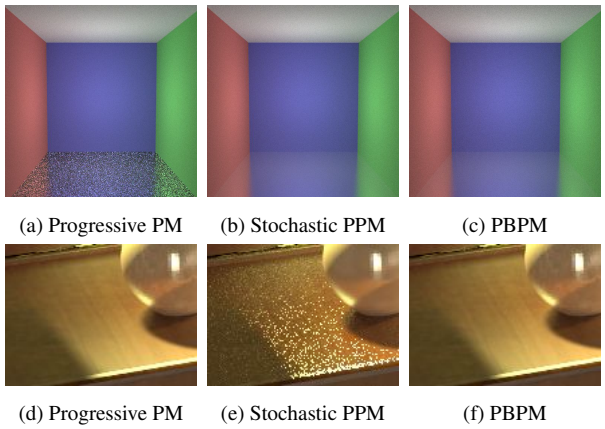
(a) Progressive PM (b) Stochastic PPM (c) PBPM

Fig. 6: The original Progressive Photon Mapping (left) performs density estimation on the first camera sub-path vertex. Stochastic Progressive Photon Mapping (center) extends camera sub-path using glossy BSDF components and performs density estimation on both vertices, using only diffuse BSDF components on the first one. Progressive Bidirectional Photon mapping (right) traces full camera sub-path, performs density estimation on all its vertices, and weights them using Multiple Importance Sampling.

6. PHOTON MAPPING-BASED APPROACHES

While Path Tracing and Bidirectional Path Tracing are an excellent choice for a wide range of scenes, some effects, e.g., reflected caustics in Figure 5, remain notoriously hard to capture. In this section we focus on a family of methods based on Photon Mapping (PM) [Jensen 2001] which can handle such effects.

Photon Mapping based approaches are similar to BPT in that they require both light and camera sub-paths. However, unlike



(a) Progressive PM (b) Stochastic PPM (c) PBPM

(d) Progressive PM (e) Stochastic PPM (f) PBPM

Fig. 7: *PPM, SPPM and PBPM on glossy surfaces.* Given a Cornell Box with a highly glossy floor (top row), Progressive Photon Mapping (a) produces a noisy image, because only very few photons have a significant contribution. Stochastic Progressive Photon Mapping (b) instead extends the camera sub-path in the direction of the glossy lobe and performs density estimation on the diffuse wall, giving a much smoother result. However, for only slightly glossy surfaces (bottom row) it is not beneficial to follow the glossy lobe and PPM (d) produces less noisy result than SPPM (e). Progressive Bidirectional Photon Mapping (c, f) uses multiple importance sampling to weight both techniques to produce a noise-free image in both cases.

BPT, camera sub-path vertices connect to all light sub-path vertices within a certain radius, an operation that requires a suitable acceleration structure. These acceleration structures have to be rebuilt for every frame and are the focus of this section.

Unlike in previous sections, where the differences between the compared variants consisted in mapping to the GPU or a slight algorithmic modification, in this section we examine acceleration structures with very different asymptotic complexities of both construction and queries. To compare the structures, we implemented three different algorithms based on Progressive Photon Mapping and we compare the results achieved with different data structures in all three of them.

All Photon Mapping based approaches share the following two-pass algorithm. In the first pass, light sub-paths are traced into the scene, on each interaction with the scene a photon is stored, and the sub-paths are extended in the same way as in BPT. The photons store only their position, incoming direction, and energy; they do not require the BSDF. In the second pass, camera sub-paths are traced and density estimation is performed at their vertices. Each photon within some radius r of the hitpoint is treated as if it arrived exactly at the hitpoint, that is, incoming direction and energy of the photon is used to evaluate the BSDF at the hitpoint. Contributions from all such photons are accumulated and divided by πr^2. The choice of the radius depends on the specific algorithm, with the common choices being a fixed radius (range query), and radius such that k nearest photons contribute (k-nearest neighbor, k-NN, query). Which vertices of the camera sub-paths perform the density estimation also depends on the specific algorithm.

Progressive Photon Mapping (PPM). Progressive Photon Mapping by Hachisuka and Jensen [2008] uses range queries to perform density estimation on the first non-specular vertex of each camera sub-path (see Figure 6a) and, using per-vertex statistics such as number of accumulated photons, reduces the query radius in a way that the whole algorithm is consistent.

Stochastic Progressive Photon Mapping (SPPM). In their follow up paper Hachisuka et al. [2009] show that the per-vertex statistics can be reused for all vertices originating from the same pixel. This can be used to improve performance on glossy surfaces, as standard density estimation on glossy surfaces produces noisy results (Figure 7a). SPPM instead uses only the diffuse component of the BSDF on the first camera sub-path vertex, extends the sub-path using glossy components and performs another density estimation on the second vertex (see Figure 6b). This often leads to less noisy results (Figure 7b).

Progressive Bidirectional Photon Mapping (PBPM). While highly glossy surfaces greatly benefit from SPPM (Figure 7, top row), always extending the camera sub-path can be adversarial when the glossy lobe is wide (Figure 7, bottom row). In that case, PPM is actually better. To address this issue, Vorba [2011] introduces Progressive Bidirectional Photon Mapping (PBPM), where the camera sub-path is extended in the same way as in Path Tracing, and density estimation is performed on each of its vertices. Multiple Importance Sampling is then used to properly weight the individual contributions, leading to a smooth result on both high and low gloss surfaces (Figure 7c and 7f).

Knaus and Zwicker [2011] show that the per-vertex (or per-pixel) statistics are not required and the radius can be reduced using a global scaling factor. In all our implementations we use this approach rather than the original reduction scheme.

All three algorithms share common elements, many of which we have already addressed. The sole new challenge is an efficient implementation of density estimation using a range query, accelerated through the use of a spatial data structure. Since both photon generation and queries are done on the GPU, it is essential that the data structure construction is also handled by the GPU, to limit CPU-GPU transfers. In the next part we focus on this aspect.

6.1 Survey of Existing GPU Implementations of Photon Map Search Structures

kD-tree. Zhou et al. [2008] describe an algorithm for GPU construction of kD-trees, the acceleration structure used in the original Photon Mapping. The algorithm first sorts photons by their coordinates and then builds the kD-tree incrementally, by levels. For each level of the kD-tree three prefix sums and three scatter/gather operations are executed. This build process is significantly more involved than the build process of Hash Grids, introduced later in this section, and our experiments show that even the range queries are slower (Table V).

Full Hash Grid. While kD-trees excel at queries with an unknown or highly varying radius, their build as well as traversal is quite costly. The original Progressive Photon Mapping [Hachisuka et al. 2008] implementation instead uses Hash Grids. We use the name Full Hash Grid to distinguish it from Stochastic Hash Grid introduced later. Here the whole scene is partitioned into a grid with cell sizes roughly equal to the diameter of expected queries and the photons are stored in these cells. As representing each cell in memory is unnecessary, a 1D array of cells, typically equal to the number of light sub-paths, is used instead. A photon's position in this array is given by a hash of its coordinates in the full grid. A good hash function should be used (we use Jenkins' hash [Jenkins 1997]). The construction of the structure is simple and easy to parallelize (see Algorithm 1). When querying the grid for photons within radius r, we iterate through all cells that are within this radius, collect all the photons, and discard those that are farther than r. When the radius is smaller than half of the cell edge length, only 8 cells have to be searched.

```
// Each cell has 1 atomic counter
// storage - array of photon indices
foreach cell:
  cell.counter = 0
foreach photon:
  cell[hash(photon)].counter += 1
Exclusive prefix sum over cell.counter
foreach photon:
  position = cell[hash(photon)].counter++
  storage[position] = photon index
```

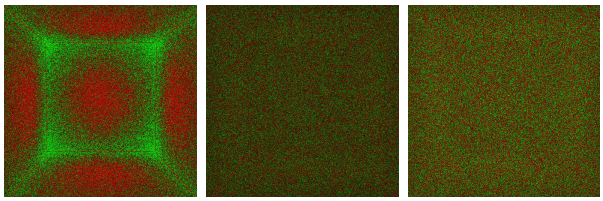
Algorithm 1: *Building hash grid*: Each cell has a single atomic counter, that is initially set to 0. Each photon increments this atomic counter, to determine how many photons belong to each cell. Exclusive prefix sum is performed over these counters, giving a start index on which photon indices belonging to each cell should be stored. In the final pass each photon increments the counter and fetches its old value. The photon's index is stored at the position given by this value in the storage array. The range of photons in the storage array that belong to a cell with index *cidx* is given by cell[*cidx*-1].counter (inclusive) to cell[*cidx*].counter (exclusive), with cell[-1].counter = 0.

This approach has two drawbacks. When the radius is significantly smaller than the size of a cell, the cell can contain many photons that will be outside the query radius and discarded. The second problem stems from hash collisions, when multiple full cells are mapped into a single hash cell. As a result the cell can, once again, contain many photons that will be outside the query range.

Stochastic Hash Grid. Hachisuka and Jensen [2010] identify two GPU-specific problems with the Full Hash Grid approach and propose the Stochastic Hash Grid to address them. All photons that belong to the same cell have to be serialized (e.g., using an atomic counter) when added to the cell. The second problem stems from the uneven number of photons in each cell – for example, surfaces close to lights can have a significantly higher density of photons. This means the number of photons processed in each query can be significantly different between camera vertices in the warp, lowering the GPU efficiency. Instead of storing all photons, they propose to store only one photon for each cell, uniformly randomly chosen from all photons that belong to the cell, with the energy increased accordingly. In their implementation each cell has an atomic counter and whenever a photon should be stored in a cell, it is simply written there and the counter is increased. For rendering, the energy of the photon is multiplied by the value of the counter.

6.2 Rectified Stochastic Hash Grid

The Stochastic Hash Grid is based on the assumption that independent threads tracing the photons lead to equal probability for each photon to be the last written to a cell. Unfortunately, this is not the case, as photons with longer paths have a higher probability of being the last. This is demonstrated in Figure 8a, depicting the positive (green) and negative (red) luminance difference between SPPM and reference rendering of Cornell Box (walls with albedo 0.99). When compared to the results of Full Hash Grid (Figure 8b), it is obvious that the original method is biased towards longer paths. [1]



(a) Stochastic Hash Grid (b) Full Hash Grid (c) Rectified Stochastic Hash Grid

Fig. 8: The positive (green)/Negative (red) difference of SPPM using the original Stochastic Hash Grid (left) and Path Traced reference, using Cornell Box with walls with albedo 0.99. The corners of the Cornell Box are visibly lighter than they should be. Our Rectified Stochastic Hash Grid (center) matches the results given by the Full Hash Grid (right).

	GeForce GTX 580			GeForce GTX 680		
	PPM	SPPM	PBPM	PPM	SPPM	PBPM
CoronaRoom	0.99	0.99	0.99	0.98	1.03	1.03
CoronaWatch	1.01	1.01	1.01	1.02	0.99	1.01
LivingRoom	1.03	1.03	1.00	1.01	1.03	1.01
BiolitFull	1.33	1.31	1.03	1.43	1.34	1.04
CrytekSponza	1.09	1.08	1.05	1.11	1.13	1.08
GrandCentral	1.03	1.03	1.01	0.98	0.98	1.02
Average	1.09	1.09	1.01	1.10	1.10	1.03

Table IV. : *WhileQuery speedup*: Speedup of *WhileQuery* over *NaiveQuery* tested on Progressive Photon Mapping (PPM), Stochastic Progressive Photon Mapping (SPPM), and Progressive Bidirectional Photon Mapping (PBPM).

Our Rectified Stochastic Hash Grid (Figure 8c) selects photons using reservoir sampling (see Algorithm R in [Vitter 1985]), where the n^{th} photon replaces the stored photon with a probability of $p = \frac{1}{n}$. This gives each photon an equal probability to be selected for the cell, irrespective of the order they arrive in.

Our second modification solves a possible race condition when writing the photon into a cell. As the write of a larger structure is not atomic, it is possible to have a result that is combined from photons of multiple threads. To prevent this, we store each photon into its globally unique memory location and write only the index of the photon.

6.3 Implementation Detail: Improved Hash Grid Query

Algorithm 2 shows two different approaches to performing a range query in a Hash Grid. The standard *NaiveQuery* processes all cells that are within range in a serial manner. On a GPU, all threads wait until each thread has processed all photons in its current cell before processing the next cell. This means that even if the total number of photons is the same across the threads, some threads might be idle, while others are still processing their photons from a given cell. Our *WhileQuery* removes this issue in a manner similar to the "while-while" loop used in [Aila and Laine 2009]. First all threads find their next photon to process, from all the cells in range, and then are the photons processed. This way the query execution is

[1] The actual implementation in [Hachisuka and Jensen 2010] is correct, as their code actually first traces all photons into a separate buffer and only then builds the Stochastic Hash Grid on top of all the photons, removing the dependency on path length, so the results given are correct.

		GeForce GTX 580							
			Full		Rectified Stochastic			kD-tree	
Algorithm	#photons	Light	Construct.	Camera	Light	Camera	Light	Construct.	Camera
CoronaRoom	147k	16.81 ms	1.56 ms	10.82 ms	16.72 ms	10.59 ms	18.50 ms	34.45 ms	15.37 ms
CoronaWatch	175k	15.18 ms	1.68 ms	14.77 ms	15.21 ms	12.97 ms	14.69 ms	41.76 ms	20.75 ms
LivingRoom	1788k	49.93 ms	9.49 ms	27.29 ms	51.52 ms	21.16 ms	47.85 ms	506.27 ms	31.56 ms
BiolitFull	1493k	36.73 ms	6.69 ms	17.78 ms	37.95 ms	12.32 ms	35.55 ms	504.84 ms	25.05 ms
CrytekSponza	571k	42.73 ms	3.50 ms	14.26 ms	43.65 ms	12.14 ms	36.77 ms	191.52 ms	13.51 ms
GrandCentral	746k	44.30 ms	4.54 ms	18.76 ms	44.77 ms	16.02 ms	42.00 ms	254.72 ms	20.18 ms
		GeForce GTX 680							
CoronaRoom	147k	16.09 ms	2.05 ms	14.65 ms	16.55 ms	14.53 ms	16.20 ms	32.79 ms	16.46 ms
CoronaWatch	175k	15.48 ms	2.27 ms	17.20 ms	15.50 ms	16.55 ms	14.97 ms	44.36 ms	21.69 ms
LivingRoom	1788k	38.59 ms	17.36 ms	27.39 ms	41.61 ms	20.24 ms	36.67 ms	603.88 ms	30.10 ms
BiolitFull	1493k	27.70 ms	12.49 ms	19.39 ms	30.96 ms	14.73 ms	25.42 ms	610.04 ms	24.20 ms
CrytekSponza	571k	27.45 ms	5.83 ms	17.16 ms	28.35 ms	15.84 ms	24.16 ms	232.81 ms	15.95 ms
GrandCentral	746k	31.12 ms	7.64 ms	23.49 ms	34.56 ms	19.79 ms	35.97 ms	311.52 ms	18.92 ms

Table V. : Time per frame for Full Hash Grid (Full), Rectified Stochastic Hash Grid (Rectified Stochastic), and kD-tree, broken down to the separate passes of PPM: photon tracing (Light), acceleration structure construction (Construct.), and camera sub-path tracing and density estimation (Camera). The times have been average over 15 minutes. Note that the Stochastic Hash Grid does not have a separate construction phase, as photons are inserted during the photon tracing. The column #photons. shows the average number of photons per frame.

driven only by the number of photons for each thread and not by their distribution within the grid cells.

We measured the performance of Progressive Photon Mapping (PPM), Stochastic Progressive Photon Mapping (SPPM), and Progressive Bidirectional Photon Mapping (PBPM) using both query algorithms on Full Hash Grid. Table IV represents the results as a speedup of the whole algorithm when using *WhileQuery*. We see that in many scenes the difference for both PPM and SPPM is negligible. However, in BiolitFull and CrytekSponza the speedup is 10-43%, as both scenes have greatly varying photon density. The effect on PBPM is significantly smaller, possibly due to the overall complexity of the algorithm, meaning the density estimation itself represents smaller fraction of the total time. The effect is more pronounced on GTX 680 than on GTX 580.

6.4 Results and Dicussion

In this section we discuss memory requirements and performance of these structures. All our implementations are single-kernel (the performance reasons are identical to LVC-BPT) and we therefore omit the *sk* suffix from the acronyms.

Memory requirements. Both Full Hash Grid and Rectified Stochastic Hash Grid require only 4 B per cell. In our setup the Hash Grids occupy only 3.5 MB. kD-tree memory requirements depend on the specific flavor used, but in our tests the size of the tree was always below 20 MB. In all cases, the required storage is dominated by the photons, not the data structure.

Performance. Table VI shows the relative speedup of the three acceleration structures as tested on PPM: Full Hash Grid (Full), Stochastic Hash Grid (Stoch.), and kD-tree (kD). To compare them we use the same time to the same quality (RMSE) method introduced in Section 5.4. Our baseline is RMSE achieved by Full Hash Grid on GTX 580 in 10 minutes.

We can see that the overall performance of kD-trees is, at best, $2\times$ slower than the Full Hash Grid: not only is the build time of the kD-tree larger than for Full Hash Grid (up to $75\times$), but the queries themselves also take slightly more time, as the traversal of the tree is more costly than simply gathering photons from 8 cells.

We note that Hash Grid, unlike kD-tree, is susceptible to hash collisions, where the 8 examined cells will include photons from different parts of the scene. This effect is responsible for the longer query times in the LivingRoom, where the highly concentrated caustic photons have to be evaluated in multiple cells across the scene. Even so, the overall performance of the Full Hash Grid on this scene is better than the kD-tree, due to the shorter build time.

While the Rectified Stochastic Hash Grid is about 25% faster, per frame, the lower number of photons used in density estimation results in an overall lower performance than the Full Hash Grid. We conclude that the cost of building a Full Hash Grid is negligible compared to the benefits and use it in all our tests.

6.5 Conclusions

In this section we investigated three progressive algorithms based on Photon Mapping, namely Progressive Photon Mapping (PPM), Stochastic Progressive Photon Mapping (SPPM), and Progressive Bidirectional Photon Mapping (PBPM). The common element of all the algorithms is density estimation, based on gathering photons in a certain radius from a query point. We examined three data structures designed to accellerate this process: kD-tree, Full Hash Grid, and Rectified Stochastic Hash Grid. Note that we did not use the original Stochastic Hash Grid [Hachisuka and Jensen 2010], as its incorrect convergence prevents using time to a given quality metrics.

As an implementation improvement of Hash Grid queries, we proposed the *WhileQuery* that processes photons from all cells as a single group, reducing thread divergence during evaluation. This speeds up range queries on Full Hash Grid by up to 43% and is used in all our tests.

While the Rectified Stochastic Hash Grid is faster, per frame, than the Full Hash Grid, owing to the fact it has no construction phase, this does not make up for the lower number of stored pho-

```
def NaiveQuery:
    activeCell = cellsInRange.nextCell
    while activeCell ≠ None:
        activePhoton = activeCell.nextPhoton
        while activePhoton ≠ None:
            if activePhoton is in range:
                Process activePhoton
            activePhoton = activeCell.nextPhoton
        activeCell = cellsInRange.nextCell
def WhileQuery:
    activeCell = cellsInRange.nextCell
    while True:
        repeat
            activePhoton = activeCell.nextPhoton
            if activePhoton = None:
                activeCell = cellsInRange.nextCell
                activePhoton = activeCell.nextPhoton
                if activeCell = None:
                    return
            if activePhoton not in range:
                activePhoton = None
        until activePhoton ≠ None
        Process activePhoton
```

Algorithm 2: *Hash Grid Query*: The *NaiveQuery* processes each cell in range serially, introducing possible inefficiencies when cells examined by threads in a warp contain different numbers of photons. The *WhileQuery* essentially concatenates all photons from all cells in range and processes this list, limiting the inefficiency only to cases when the total number of photons in range differs between threads.

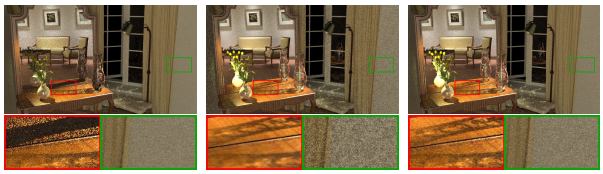
	GeForce GTX 580			GeForce GTX 680		
	Full	Stoch.	kD	Full	Stoch.	kD
CoronaRoom	**1.00**	0.87	0.43	**0.91**	0.78	0.46
CoronaWatch	**1.00**	0.54	0.44	**0.91**	0.48	0.43
LivingRoom	**1.00**	0.51	0.15	**1.06**	0.59	0.13
BiolitFull	**1.00**	0.45	0.11	**1.04**	0.50	0.10
CrytekSponza	**1.00**	0.64	0.25	**1.21**	0.80	0.22
GrandCentral	**1.00**	0.53	0.21	**1.10**	0.61	0.18
Average	**1.00**	0.55	0.22	**1.06**	0.61	0.20

Table VI. : *Acceleration Structure comparison*: Speedup, in terms of time to a given quality, of Full Hash Grid (Full), Rectified Stochastic Hash Grid (Stoch.), and kD-tree (kD), relative to Full Hash Grid on GTX 580.

tons, resulting in 13 to 55% slower performance in all scenes except the CrytekSponza. Both Hash Grids significantly outperform kD-tree, mainly due to its substantial construction cost. As result, we recommend using the Full Hash Grid with our *WhileQuery* for a GPU implementation of PM methods.

7. VERTEX CONNECTION AND MERGING

In this section we combine the experience gathered from the previous sections to introduce the first GPU implementation of the recent Vertex Connection and Merging (VCM) algorithm [Georgiev et al. 2012]. While Bidirectional Path Tracing fails to capture reflected caustics and methods based on Photon Mapping have difficulties producing noise-free diffuse surfaces under illumination from distant light sources, Vertex Connection and Merging combines the best of both algorithms by using Multiple Importance Sampling to give a high weight to the best strategy for each situation (see Figure 9). Hachisuka et al. [2012] developed the same idea concurrently, using different derivations, under the name Unified Path Sampling. For the purpose of this paper we choose the former name, but note that both approaches result in the same algorithm.



(a) Bidirectional Path Tracing (b) Progressive Bidirectional PM (c) Vertex Connection and Merging

Fig. 9: The reflected caustics (red) are extremely difficult for Bidirectional Path Tracing (left). Progressive Bidirectional Photon Mapping (middle), on the other hand, results in noticeable noise on the diffuse walls (green). In the same rendering time (10s), Vertex Connection and Merging (right) handles both effects well.

7.1 Algorithm Overview

In the original paper the authors trace an equal number of light and camera sub-paths, forming predetermined path pairs. First, all light sub-paths are traced and their vertices are stored. Then camera sub-paths are traced, each camera sub-path vertex is connected to all vertices on the corresponding light sub-path, as well as merged with vertices (i.e., 'photons'), from all light sub-paths, that are within a given range. Vertex merging is a name used for an operation virtually identical to the range query in Progressive Bidirectional Photon Mapping, the only difference being the different calculation of the Multiple Importance Sampling weights.

7.2 Proposed GPU Implementation

Given our Light Vertex Cache Bidirectional Path Tracing (LVC-BPT), introduced in Section 5.3, and our GPU implementation of Bidirectional Photon Mapping (Section 6), GPU implementation of Vertex Connection Merging is easy. Our implementation first traces all light sub-paths. Then a Full Hash Grid is built over these vertices to accelerate range queries (we use our *WhileQuery*), as in PBPM. Finally, a camera pass is performed, where each camera sub-path vertex is connected to a predetermined number of light sub-path vertices (identical to LVC-BPT) and also merged with light sub-path vertices using a range query (identical to PBPM).

7.3 Results and Discussion

Memory requirements. Memory requirements are almost identical to the requirements of LVC-BPT. Using the formulas introduced in Table II we arrive at 17 to 171 MB for VCMsk, with VCMmk adding another 137 to 158 MB. We also need memory for the Full Hash Grid used to accelerate vertex merging range queries, leading to a memory footprint of 3.5 MB.

Kernel configurations. Similar to Light Vertex Cache BPT, we tested both single-kernel (VCMsk) and multi-kernel (VCMmk) im-

	GeForce GTX 580		GeForce GTX 680	
	VCMsk	VCMmk	VCMsk	VCMmk
CoronaRoom	**1.00**×	0.98×	**0.93**×	0.85×
CoronaWatch	**1.00**×	0.79×	**0.87**×	0.71×
LivingRoom	1.00×	**1.04**×	**0.92**×	0.88×
BiolitFull	1.00×	**1.03**×	**0.92**×	0.86×
CrytekSponza	**1.00**×	0.91×	**1.00**×	0.85×
GrandCentral	**1.00**×	0.97×	**0.92**×	0.87×
Average	**1.00**×	0.95×	**0.93**×	0.83×

Table VII. : Relative performance of VCM implemented with a single kernel (VCMsk) and using two kernels (VCMmk).

plementation of VCM. Table VII shows performance relative to VCMsk on GTX 580. We can see that in almost all cases, VCMmk is inferior to VCMsk. On GTX 580, VCMmk outperforms VCMsk in two scenes by 3 and 4% respectively, but in general is approximately 5% slower. This difference is more pronounced on GTX 680. Because multi-kernel VCM uses a larger light sub-path vertex structure as well as the whole merging stage, it has greater pressure on the memory system, leading to a decrease in performance with respect to the single-kernel implementation. We conclude that the single-kernel VCM is the better choice for both GPUs.

To conclude, our Vertex Connection and Merging implementation draws heavily on the experiences from both Bidirectional Path Tracing and Progressive Bidirectional Photon Mapping implementations. The main approach is almost identical to our LVC-BPT, using a single-kernel implementation. Compared to the CPU implementation used in [Georgiev et al. 2012], our GPU implementation achieves a 6 to 10× speedup on the scenes used in the paper.

8. ALGORITHM COMPARISON

Up until now we have focused on optimizing the individual algorithms. Now, with state-of-the-art GPU implementations of a number of light transport simulations algorithms at our disposal, within a single framework, we have a unique opportunity to compare the algorithms against each other. Our comparison is "unbiased" in the sense that we did not introduce any of the algorithms in this paper, and so have no motivation to selectively prefer any of them. Our results are not strictly GPU-specific; we are not aware of a similar unbiased comparison for CPU implementations either.

Figure 10 shows results for GTX 680. PPM, SPPM, and PBPM use single-kernel implementations and all algorithms that perform density estimation use the Full Hash Grid and our *While-Query*. The results for GTX 580, using multi-kernel versions of RegenerationPTmk and LVC-BPTmk, closely follow the results of GTX 680. The graphs are given in the supplemental material. Our references are computed by NaiveBPT in 10 hours, except for the LivingRoom scene, where the reference is computed by VCMsk, as BPT cannot resolve the reflected caustic even after 10 hours.

8.1 Path Tracing

Path Tracing excels in scenes with a simple illumination, as expected. From our test scenes, it achieves the best results on *CoronaRoom*, a mostly diffuse scene where majority of the illumination comes from an environment lighting behind the glass-less window.

Good results are also achieved on *CoronaWatch*, which is dominated by direct illumination. However, on Figure 10b we can see that the convergence of PT starts to level off after approximately 100 s, due to inappropriately sampled gloss-to-gloss transport.

The somewhat surprising poor performance on *GrandCentral* is caused by the many individual point lights in the niches beneath the ceiling. While the overall illumination of the scene is smooth, these niches are each illuminated by essentially a single point light, which poses a great challenge for next event estimation and causes majority of the variance we see in the graph.

The other three scenes are strongly illuminated by indirect light sources, which renders next event estimation essentially useless in these cases and the overall convergence of PT suffers.

8.2 Bidirectional Path Tracing

Bidirectional Path Tracing performs well on all the scenes except *LivingRoom*, a scene tailored to showcase Vertex Connection and Merging, where BPT does not have any technique suitable for efficiently capturing reflected caustics.

On the two scenes dominated by direct illumination, i.e., *CoronaRoom* and *CoronaWatch*, the algorithm is slower than Path Tracing, as the extended set of techniques offered by BPT is not really useful. In *GrandCentral*, the niches are illuminated by paths traced from the point lights, greatly reducing noise when compared to PT, and the algorithm naturally handles well both scenes that are dominated by indirect illumination (*BiolitFull* and *CrytekSponza*).

8.3 Photon Mapping-based Methods

The Photon Mapping-based methods are most beneficial on *LivingRoom*, where none of the path-based algorithms can efficiently capture the reflected caustics. Somewhat surprising is the good behavior of Progressive Bidirectional Photon Mapping on both *CoronaRoom* and *CoronaWatch*, when compared to Progressive Photon Mapping and Stochastic Progressive Photon Mapping. The key insight here is that PBPM has Path Tracing without next event estimation amongst its techniques and both scenes, with their large area lights, represent a very good case for this technique, up to the point that the convergence on *CoronaWatch* is actually dominated by it and matches the convergence rate of path based techniques.

In the case of *BiolitFull*, PPM and SPPM give very good results in a short time. This is due to mostly diffuse nature of the scene, where each photon contributes to several pixels, giving a good, albeit blurry, initial estimate (Figure 11).

However, in the *CrytekSponza*, which is purely diffuse and also indirectly illuminated, the results are quite different. Unlike *BiolitFull*, where all the lights are within a single room and roughly half the scene and third of the lightsources are visible to the camera, in *CrytekSponza* we see only a fraction of the scene and all the lights are completely outside the view. As result, a significantly lower fraction of all photons contributes to the frames (0.11% vs 6.3%), giving much more noisy results.

8.4 Vertex Connection and Merging

VCM excels in *LivingRoom*, which has been tailored to showcase the algorithm. It resolves the reflected caustics using techniques from PBPM, while resolving the diffuse light transport with BPT techniques. In the other scenes however, VCM simply mirrors the performance of BPT, in general being slightly slower, as the vertex merging techniques have a rather negligible effect at the cost of non-negligible overhead.

9. CONCLUSIONS

In this paper we presented an extensive study of GPU-based implementations of several progressive light transport simulation al-

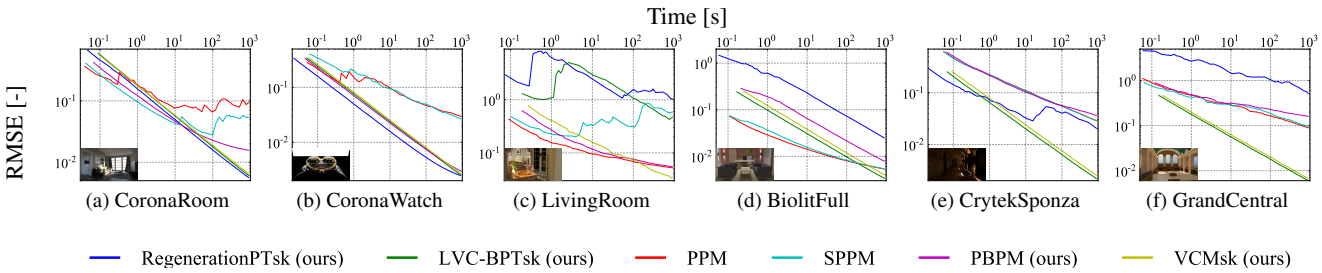
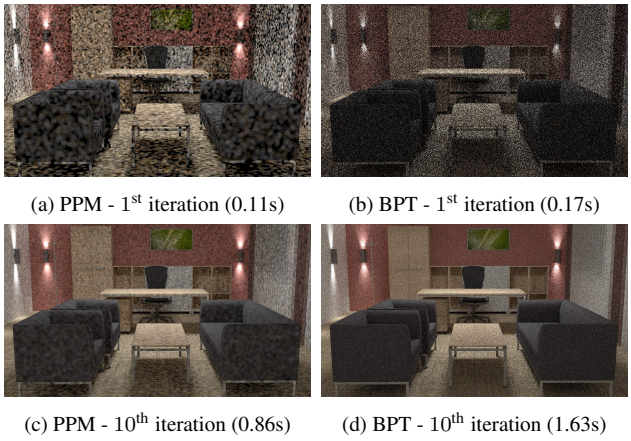


Fig. 10: The log-log plot of RMSE-vs-time convergence of the six tested methods on each of the test scenes.



(a) PPM - 1st iteration (0.11s) (b) BPT - 1st iteration (0.17s)

(c) PPM - 10th iteration (0.86s) (d) BPT - 10th iteration (1.63s)

Fig. 11: After the first iteration (top) the PPM (a), with a large radius, gives a better initial intuition about the lighting than BPT (b). After the tenth iteration (bottom) the asymptotically faster convergence of BPT has already removed this advantage.

gorithms. For each algorithm, we evaluated existing and new approaches on GPUs of two different NVIDIA architectures, the older Fermi (GTX 580) and the newer Kepler (GTX 680) architecture.

In the Path Tracing section we examined the *low-level details* of mapping the basic building block of the more complex algorithms – path sampling – onto the GPU. We conclude that for optimal performance it is beneficial to use a low number of separate kernels, as the lower number of loads and stores outweighs the gains from improved GPU occupancy. Notably, on Kepler, the speed gained by using Path Tracing with Regeneration with only a single kernel actually matches the speed gained by stream compaction used in state-of-the-art Streaming Path Tracing with Regeneration, using 2 kernels.

In the Bidirectional Path Tracing section, we show that maximal simplification of the algorithm structure leads to the best performance. We proposed our Light Vertex Cache BPT, storing only light path vertices without the notion of sub-paths. Doing so increases the performance by 30-60% when compared to the state-of-the-art, while at the same time removing the necessity of a maximum path length.

In the Photon Mapping section we show that a simpler but asymptotically slower algorithm, in our case the Hash Grid, can outperform a more complex asymptotically faster algorithm, in our case the kD-tree. Another important low level optimization is our *WhileQuery*, used to gather photons from a Hash Grid. By removing thread synchronization after gathering photons from a single cell, we reduce the thread divergence of the gather process, which can increase the performance by up to 40%. All of the findings are combined in Vertex Connection and Merging, showing the first GPU implementation of the algorithm.

Our algorithm comparison shows the raw performance of Path Tracing makes it ideal for scenes with a low lighting complexity, but the more sophisticated sampling strategies of Bidirectional Path Tracing are useful in scenes with more complex lighting. In most scenes, the performance of Vertex Connection and Merging follows that of BPT, but due to the overhead of merging (which has only a marginal impact on the final image) it is about 15% slower to achieve the same image quality. Of course, in scenes with a strong reflected caustic component, VCM outperforms BPT since the merging is essential to capturing these light paths. The Photon Mapping based algorithms do not present a significant advantage over any of the other algorithms.

In the future, it would be interesting to examine Metropolis Light Transport (MLT) on the GPU, such as GPU implementation of [Hachisuka and Jensen 2011], but for some of the presented algorithms (Light Vertex Cache, VCM), there is no established MLT apporoach at all. Another challenging future work venue concerns out-of-core and over-the-network texture accesses, e.g., when threads executing the same BSDF kernel need to access different out-of-core texture tiles from several separate texture images.

ACKNOWLEDGMENTS

We would like to thank the following people for providing the respective scenes: Ludvík Koutný (`http://raw.bluefile.cz/`) for *CoronaRoom*, Jerome White for *CoronaWatch*, Iliyan Georgiev (`http://www.iliyan.com`) for *LivingRoom*, Jiří "Biolit" Friml (`http://biolit.wordpress.com/`) for *BiolitFull*, Marko Dabrovic (`http://hdri.cgtechniques.com/~sponza/files/`) and Frank Meinl from CryTEK (`http://www.crytek.com/cryengine/cryengine3/downloads`) for *CrytekSponza*, and the Cornell University Program of Computer Graphics (`http://www.graphics.cornell.edu/`) for *GrandCentral*. We would also like to thank Jan Novák and Iliyan Georgiev for their consultations on many finer implementation details. Thanks to Petr Kadleček and Martin Kahoun for proofreading. This work was supported by the Czech Science Foundation grant P202-13-26189S.

REFERENCES

AILA, T., KARRAS, T., AND LAINE, S. 2013. On quality metrics of bounding volume hierarchies. In *Proc. High-Performance Graphics 2013*. 101–107.

AILA, T. AND LAINE, S. 2009. Understanding the efficiency of ray traversal on GPUs. In *Proc. High-Performance Graphics 2009*. 145–149.

AILA, T., LAINE, S., AND KARRAS, T. 2012. Understanding the efficiency of ray traversal on GPUs – Kepler and Fermi addendum. NVIDIA Technical Report NVR-2012-02, NVIDIA Corporation.

CARR, N. A., HALL, J. D., AND HART, J. C. 2002. The ray engine. In *Proc. Graphics Hardware 2002*. 37–46.

DACHSBACHER, C., KŘIVÁNEK, J., HAŠAN, M., ARBREE, A., WALTER, B., AND NOVÁK, J. 2013. Scalable realistic rendering with many-light methods. In *Compute Graphics Forum (STAR)*.

FOLEY, T. AND SUGERMAN, J. 2005. Kd-tree acceleration structures for a GPU raytracer. In *Proc. of Graphics Hardware 2005*. 15–22.

GEORGIEV, I. 2012. Implementing vertex connection and merging. Tech. Rep. Nov. 12, Saarland University.

GEORGIEV, I., KŘIVÁNEK, J., DAVIDOVIČ, T., AND SLUSALLEK, P. 2012. Light transport simulation with vertex connection and merging. *ACM Trans. Graph. 31,* 6 (Nov.), 192:1–192:10.

HACHISUKA, T. AND JENSEN, H. W. 2009. Stochastic progressive photon mapping. *ACM Trans. Graph. 28,* 5 (Dec.), 141:1–141:8.

HACHISUKA, T. AND JENSEN, H. W. 2010. Parallel progressive photon mapping on GPUs. In *ACM SIGGRAPH ASIA 2010 Sketches*. 54:1–54:1.

HACHISUKA, T. AND JENSEN, H. W. 2011. Robust adaptive photon tracing using photon path visibility. *ACM Trans. Graph. 30,* 5 (Oct.), 114:1–114:11.

HACHISUKA, T., OGAKI, S., AND JENSEN, H. W. 2008. Progressive photon mapping. *ACM Trans. Graph. 27,* 5 (Dec.), 130:1–130:8.

HACHISUKA, T., PANTALEONI, J., AND JENSEN, H. W. 2012. A path space extension for robust light transport simulation. *ACM Trans. Graph. 31,* 6 (Nov.), 191:1–191:10.

HAVRAN, V. 2000. Heuristic ray shooting algorithms. Ph.D. thesis, Faculty of Electrical Engineering, Czech Technical University in Prague.

HOU, Q., SUN, X., ZHOU, K., LAUTERBACH, C., AND MANOCHA, D. 2011. Memory-scalable GPU spatial hierarchy construction. *Visualization and Computer Graphics, IEEE Transactions on 17,* 4, 466–474.

JENKINS, B. 1997. Hash functions. *Dr Dobbs Journal 22,* 9.

JENSEN, H. W. 2001. *Realistic Image Synthesis using Photon Mapping*. A.K. Peters.

KAJIYA, J. T. 1986. The Rendering Equation. In *Computer Graphics (Proc. of SIGGRAPH)*. 143–150.

KALOJANOV, J. AND SLUSALLEK, P. 2009. A parallel algorithm for construction of uniform grids. In *Proc. of High-Performance Graphics 2009*. 23–28.

KARRAS, T. AND AILA, T. 2013. Fast parallel construction of high-quality bounding volume hierarchies. *Proc. of High-Performance Graphics 2013*, 89–99.

KARRAS, T., AILA, T., AND LAINE, S. 2012. Understanding the efficiency of ray traversal on GPUs framework. `http://code.google.com/p/understanding-the-efficiency-of-ray-traversal-on-gpus/`.

KELLER, A. 1997. Instant Radiosity. In *Computer Graphics (Proc. of SIGGRAPH)*. 49–56.

KNAUS, C. AND ZWICKER, M. 2011. Progressive photon mapping: A probabilistic approach. *ACM Trans. Graph. 30,* 3 (May), 25:1–25:13.

KŘIVÁNEK, J., HAŠAN, M., ARBREE, A., DACHSBACHER, C., KELLER, A., AND WALTER, B. 2012. Optimizing realistic rendering with many-light methods. In *ACM SIGGRAPH 2012 Courses*. 7:1–7:217.

LAFORTUNE, E. AND WILLEMS, Y. D. 1993. Bi-directional path tracing. In *Proc. of CompuGraphics '93*.

LAINE, S., KARRAS, T., AND AILA, T. 2013. Megakernels considered harmful: Wavefront path tracing on GPUs. *Proc of High-Performance Graphics 2013*, 137–143.

LAUTERBACH, C., GARLAND, M., SENGUPTA, S., LUEBKE, D., AND MANOCHA, D. 2009. Fast BVH construction on GPUs. In *Computer Graphics Forum*. Vol. 28. 375–384.

NOVÁK, J., HAVRAN, V., AND DASCHBACHER, C. 2010. Path regeneration for interactive path tracing. In *EUROGRAPHICS 2010, short papers*. 61–64.

NVIDIA. 2011. *Fermi Compute Architecture Whitepaper*.

NVIDIA. 2012a. *CUDA C Programming Guide 5.0*. NVIDIA.

NVIDIA. 2012b. *NVIDIA GeForce GTX 680 Whitepaper*.

PAJOT, A., BARTHE, L., PAULIN, M., AND POULIN, P. 2011. Combinatorial bidirectional path-tracing for efficient hybrid CPU/GPU rendering. In *Computer Graphics Forum*. Vol. 30. 315–324.

PANTALEONI, J. AND LUEBKE, D. 2010. HLBVH: hierarchical LBVH construction for real-time ray tracing of dynamic geometry. In *Proc. of High-Performance Graphics 2010*. 87–95.

PARKER, S. G., BIGLER, J., DIETRICH, A., FRIEDRICH, H., HOBEROCK, J., LUEBKE, D., MCALLISTER, D., MCGUIRE, M., MORLEY, K., ROBISON, A., ET AL. 2010. OptiX: A general purpose ray tracing engine. *ACM Trans. Graph. 29,* 4 (July), 66:1–66:13.

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*. Vol. 26. 415–424.

PURCELL, T. J., BUCK, I., MARK, W. R., AND HANRAHAN, P. 2002. Ray tracing on programmable graphics hardware. In *ACM Trans. Graph.* Vol. 21. 703–712.

PURCELL, T. J., DONNER, C., CAMMARANO, M., JENSEN, H. W., AND HANRAHAN, P. 2003. Photon mapping on programmable graphics hardware. In *Proc. Graphics Hardware 2003*. 41–50.

RITSCHEL, T., DACHSBACHER, C., GROSCH, T., AND KAUTZ, J. 2012. The state of the art in interactive global illumination. In *Computer Graphics Forum (STAR)*. Vol. 31. 160–188.

SENGUPTA, S., HARRIS, M., ZHANG, Y., AND OWENS, J. D. 2007. Scan primitives for GPU computing. In *Proc. Graphics Hardware 2007*. 97–106.

STICH, M., FRIEDRICH, H., AND DIETRICH, A. 2009. Spatial splits in bounding volume hierarchies. In *Proc. of High-Performance Graphics 2009*. 7–13.

VAN ANTWERPEN, D. 2011a. Improving SIMD efficiency for parallel Monte Carlo light transport on the GPU. In *Proc. of High-Performance Graphics 2011*. 41–50.

VAN ANTWERPEN, D. 2011b. Unbiased physically based rendering on the GPU. M.S. thesis, Delft University of Technology, the Netherlands.

VEACH, E. AND GUIBAS, L. 1994. Bidirectional Estimators for Light Transport. In *Proc. of Eurographics Rendering Workshop*. 147–162.

VEACH, E. AND GUIBAS, L. J. 1995. Optimally combining sampling techniques for monte carlo rendering. In *Computer Graphics (Proc. of SIGGRAPH)*. 419–428.

VITTER, J. S. 1985. Random sampling with a reservoir. *ACM Trans. Math. Softw. 11,* 1 (Mar.), 37–57.

VORBA, J. 2011. Optimal strategy for connecting light paths in bidirectional methods for global illumination computation. M.S. thesis, Charles University in Prague.

WALD, I., SLUSALLEK, P., BENTHIN, C., AND WAGNER, M. 2001. Interactive rendering with coherent ray tracing. In *Computer Graphics Forum*. Vol. 20. 153–165.

WOOP, S., FENG, L., WALD, I., AND BENTHIN, C. 2013. Embree: Ray tracing kernels for CPUs and the Xeon Phi architecture. In *ACM SIGGRAPH 2013 Talks*. 44:1–44:1.

ZAFAR, F., OLANO, M., AND CURTIS, A. 2010. GPU random numbers via the tiny encryption algorithm. In *Proc. of High-Performance Graphics 2010*. 133–141.

ZHOU, K., HOU, Q., WANG, R., AND GUO, B. 2008. Real-time kd-tree construction on graphics hardware. *ACM Trans. Graph. 27,* 5 (Dec.), 126:1–126:11.