

# Cheap and simple in-place block based path regeneration\*

Frolov V.A.<sup>1,2</sup>, Galaktionov V.A.<sup>1</sup>

vfrolov@graphics.cs.msu.ru | vlgal@gin.keldysh.ru

<sup>1</sup>Keldysh Institute of Applied Mathematics RAS, Moscow, Russia;

<sup>2</sup> Moscow State University, Moscow, Russia;

Monte Carlo Path Tracing is a core light transport technique which is used for modern methods (like BDPT, MLT, ERPT, VCM and others). One of the main challenge of efficient GPU Path Tracing implementation is inefficient workload caused by paths of different lengths; few threads process the long paths, while other threads are idle. A work distribution technique called “Path Regeneration” is commonly used to solve this problem. We introduce a novel GPU implementation of path regeneration technique called “in place block based path regeneration”. In comparison to previous approaches our algorithm possesses two main advantages: it has lower self-cost and it does not move any per-ray data along threads in memory, thus, our algorithm can be easily integrated to any advanced path tracing technique (like BDPT, MLT and other) or photon mapping. We tested our solution with path tracing using both CUDA and OpenCL.

**Keywords:** OpenCL Path Tracing, Path Regeneration

Path tracing (PT) generates Monte Carlo samples by simulating light transport via random path walk in the scene. A path starts with a primary ray at camera, traces in to the scene, randomly reflects several times and finishes at light, some Lambertian surface or an environment. The other light transport methods start a path at the light (Light Tracing, Photon Mapping) or both at the eye and light (BDPT). Due to the stochastic nature of all modern light transport algorithms, an effective GPU implementation with high trace depth becomes a challenge: deep reflection levels have only several active paths while GPU has to execute mostly all of them. Due to SIMD nature even inactive threads (i.e. terminated paths) have to be executed, as long as there is at least one active thread in their warp. Moreover, a completely dead warp could still waste multiprocessor resources reading “inactive” flag from intermediate data in DRAM (see “multiple-kernel” further) or due to inefficient work distribution implementation in driver when the whole block is still occupies multiprocessor resources until all of its warps become inactive (one possible solution refers to “persistent threads” [1]).

## Introduction

**Single and multiple kernel** There are two main software architectures for GPU path tracing implementation. They are “single kernel” and “multiple kernel”. The naive “single kernel” implementation generally holds the whole algorithm in a single large kernel, essentially the same as a standard CPU path tracer implementation. Such implementation is inefficient due to the limited number of registers on GPUs and register spilling via local memory [2]. The “Uber-kernel” is modification of naive “single kernel” implementation that saves GPU registers converting tradi-

tional CPU-based code to a state-machine where huge and complex code splits to some simpler parts – each part per one state. NVIDIA OptiX [3] uses this way. The multiple-kernel moves these parts to different kernels, which leads to more efficient register usage and significant performance gain for critical code such as BVH traversal [2]. However, “multiple-kernel” implementation has to store explicitly intermediate per-ray data (position, direction, hit normal, material reference, some flags etc.) to DRAM. Despite very simple scenes, it does not affect the performance but can significantly limit flexibility of light transport algorithm implementation. For example, simulating recursion via stack is easy in “single kernel” implementation but is very difficult in “multiple kernel”. Since in “multiple kernel” implementation per-ray data addresses are usually strictly bound to thread index, the other good examples where “multiple kernel” complicate things are rays compaction, rays sorting, path regeneration. During our research, we additionally found out that OpenCL kernel compiler on some devices (HD 5770, Intel and AMD CPUs) failed to compile huge single-kernel path tracing or even multiple-kernel path tracing with complex materials. Thus, “multiple-kernel” path tracing is the natural way of splitting code complexity to prevent “fresh” OpenCL compilers from unexpected faults.

## Regeneration overhead

Thus, both single and multiple-kernel PT have their advantages and disadvantages. Wanting to have a stable multi-platform implementation, we prefer “multiple-kernel” approach rather than a “single kernel”. Then we found that with “multiple-kernel” PT implementation existing path regeneration approaches were not stable in the sense of performance gain: while several heavy and complex scenes benefit from path regeneration, the majority of them are not. The reason was that the PT regeneration wins less performance for tracing rays than its self-cost. In some cases (fig. 3,4) total performance went down. Thus, the pri-

The work was supported by RFBR, grant # 13-01-00454; president scholarship (RF) SP-4053.2013.5.

mary motivation for our research was to propose lightweight path regeneration algorithm we can use without fear for degrading performance. The second motivation was flexibility and simplicity. Having big plans for advanced Monte Carlo techniques we would like to have simple and flexible PT core. The existing path regeneration approaches lack of these properties.

## Related work

### Path regeneration

Novak et al. [4] introduce path regeneration technique restarting the terminated paths and tracing additional ones. The newly generated paths for same thread come always from the same pixel when anti-aliasing or DOF enabled. Major disadvantage of this approach is branch divergence growth: regenerated rays (which may be coherent for several bounces) mixed with incoherent rays in the same warp. This leads to significant performance penalty (2.x-3.x) for coherent rays and eliminates performance gain of regeneration.

The Streaming PT regeneration approach [5] uses compaction to move all active threads in the beginning and fills inactive threads with new rays/paths. The coherent primary rays of the new paths in this approach assigned to threads that executed together. However, this approach has other disadvantages. They are self-cost and complexity:

1. While the self-cost of regeneration approach from [4] is near zero, compaction from [5] could simply eat 10-20% of ray tracing performance. Thus, the typical regeneration gain of 20-30% is suppressed by algorithm self-cost. This is mostly because of the forced necessity to move per-ray local data in memory or to use indices. When “multiple-kernel” PT implementation considered, each per-ray data (like ray position, path color, path throughput and etc.) location is bound to thread index. For example, we can read path color in analogue to (a). The compaction changes actual thread index for active paths, so data must be moved or indices should be used like (b). Beside additional indirection level indices in this way break memory coalescing.

(a) `color = in_color[threadId];`

(b) `color = in_color[original_threadId[threadId]];`

2. An attempt to use indices for only several attributes (many of per ray data live only one bounce) forces us to remember which data are affected and which are not. Each time we read attribute we must recall what to use: (a) or (b).

3. More complex light transport algorithms (like BDPT/VCM or MLT) make compaction even trickier and slower. The necessity of storing per-path vertex data replaces simple indices with lists (because single path may change its thread index several times) and makes “moving alternative” more expensive due to larger per-ray data.

To address issue of inefficient global loads and stores when path regeneration used with “multiple kernel” Davidovic et al. use “single-kernel” path tracing [6]. We discussed “single kernel” disadvantages earlier. Moreover, implementation from [6] has approximately the same performance as [5].

Wald [7] concluded that terminated threads in a warp incur no major performance penalties due to the remaining threads executed faster. According to [7] “certain limitations of today’s hardware lead to sources of overhead that significantly affect the final outcome, eventually leading to disappointingly small speed-ups of only 12–16% for even the best performing of our kernels”. This slightly differs from [4–6] results, but it is consistent with our experiments – existing approaches do have too much overhead. We should notice that there could be at least two reasons for low path regeneration efficiency in [7]; they are simple materials and low path length (maximum 8).

Laine et al. [8] further noticed the first reason and introduced “wavefront path tracing” aiming efficient evaluating extremely complex BSDFs with “multiple kernel” implementation. However, only “small performance benefit” from path regeneration is still marked in their work.

### Proposed method relations

Before considering proposed algorithm we should mention two techniques that are not directly related to path regeneration, but, in general, related to efficient path tracing implementation on GPU. They are tile-based work distribution for path tracing [2] and “per-warp Russian Roulette” [9].

The tile-based work distribution [2] splits an entire screen to 16x16 tiles and use them as an atomic unit of work distribution. Tiles are important because they reduce work-distribution algorithm self-cost by dividing actual operations number by 256 (16x16).

Per-warp Russian roulette introduced in [9]. The original Russian roulette randomly terminates a path to restrict its depth – trace fewer paths on high depth but takes them into account with greater weight. For GPU this additionally increases amount of “sparse warps” [4] with dead threads. Per-warp Russian roulette decides to terminate (or not to terminate) the whole warp by slightly changing weights computation scheme.

By combining tiles idea from [2] and “per-warp Russian roulette” from [9] we introduce a new path regeneration method that is as simple and flexible as approach from [4], as efficient as [5] in the sense of ray tracing performance gain, and cheap in the sense of self-cost.

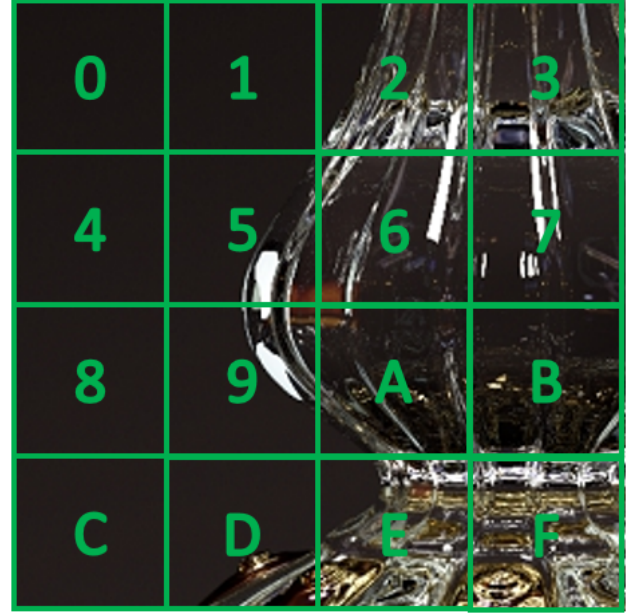
## Suggested approach

The proposed algorithm is designed under the assumption that to achieve high actual occupancy we don't need completely dense thread pool. There is no significant performance penalty if some continues gaps (25-50% of total tiles number in thread pool) with dead threads exist in it. Our algorithm combines several ideas:

1. Similar to [2] we subdivide screen to tiles of 16x16 pixels. Thus, pixels are separated from threads. Each screen tile is mapped to continuous sequence of 256 rays in thread pool.
2. Similar to [4] we perform "in place" path regeneration, assigning new paths to dead threads. In contrast to [4] we never regenerate single thread but always regenerate the whole 16x16 tile simultaneously, i.e. all 256 continuous threads. A thread block always processes other screen tile when it has been regenerated.
3. To terminate the whole block we extend "per-warp" Russian roulette to a "per-block" Russian roulette. This efficiently terminates a block of threads allowing a new tile to come in the same place (replacing dead tile) when low actual occupancy met for an old tile.
4. During path tracing we store for each tile its maximum trace depth and sort all tiles according to that value before starting next pass. Thus, tiles with high trace depth always come to thread pool first and hang there as long as they are needed. In contrast to that, tiles with low depth come much later and usually process several bounces only. Such strategy reduces total number of kernel launches in range of 30 - 60% depending on scene and maximum trace depth.
5. We don't regenerate each bounce. For each second bounce we check a number of tiles we want to regenerate. If this number is greater than a threshold (half of a current active tiles number), we invoke regenerate kernel. We call this "threshold based regeneration".
6. Particularly, for ray-tracing kernels we use local thread compaction via shared memory inside the 256-thread block. Because these kernels are very light-weight in the sense of reading and writing per-ray data, local thread compaction has no valuable overhead. However, due to compaction reduces sparse warps number it speeds up ray tracing performance in average by 8-15%. Such local thread permutations inside block do not lead to per-ray data movements because it changes thread indices only inside trace kernel.

Thus, when too many tiles are dead, according to point (5) we regenerate (fig. 1). Threshold based regeneration (5) decreases regeneration code cost and increases actual occupancy due to when regenera-

tion kernel launched, we know that significant number of continues sequences of threads does regenerate, thus, we prevent empty kernel calls when a kernel is launched but no threads are doing actual work.



**Fig. 1.** An example of image subdivided to 16 tiles. Each tile is enumerated with hexadecimal number from 0 to F. We sort all tiles basing on their maximum trace depth. Tiles E and F have very large maximum trace depth. They come to the thread buffer first and finish last. Opposite to that tiles 0,1,4,8,C have low maximum trace depth equals to one. They come last and live just 1 bounce in the buffer. Thus, we reduce total kernel calls number.

F E A B 6 7 9 5 D 3 2 1 0 4 8 C

```

X X X X X X X X | F E A B 6 7 ...
F E A B 6 7 9 5 | D 3 2 1 8 C
F E A B 6 7 9 5 | D 3 2 1 8 C
F E A B 6 7 9 5 | D 3 2 1 8 C (9,5)->(D,3)
F E A B 6 7 D 3 | 2 1 0 4 .. (A,6,7)->(2,1,0)
F E 2 B 1 0 D 3 | 4 8 C
F E 2 B 1 0 D 2 | 4 8 C (1,0)->(4,8)
F E 2 B 4 8 D 2 | C (4,8)->(C,X)
F E 2 B C X D 2 | (C)->(X)
F E 2 B X X D 2 | (D,2)->(X,X)
F E 2 B X X X X |
...
X X X X X X X X |

```

**Listing 1.** Simplified algorithm walkthrough (without "threshold-based regeneration"). A thread buffer holds 8 tiles (8x256 rays). The total number of tiles is 16. The left part represents active tiles in thread buffer. The right part after the slash is a queue of tiles that waiting for process. Symbol "X" represents dead tiles. The notation "(9,5)->(D,3)" means that tiles "9" and "5" were died and

new paths were regenerated replacing “9” with “D” and “5” with “3”.

### Implementation details

We used the same kernel set in CUDA and OpenCL. Our path-tracing pipeline has 7 kernels in total. They are: regenerate, trace, surfEval, lightSample, shadowTrace, shade, nextBounce. Such fine-grained “multiple kernel” was used due to heavy feature list. Merging kernels in this case leads to poor performance and unexpected faults for OpenCL on some drivers. Almost all of our kernels are heavy in the sense of code complexity: surfEval kernel has Parallax Occlusion Mapping for surfaces with relief; lightSample implements huge sets of light including complex lights with HDR environment and IES distribution; Shade and nextBounce use material tree for flexible material architecture with different type of BRDS in leafs and different types of blending in non-leaf nodes. This is like a “Mila material” in Mental Ray [10]. Thus, in our case not only tracing rays was a kernel that benefits from path regeneration.

## Results and discussion

### Test setup

Our test setup focuses on different cases when performance benefit should come from different reasons (Fig.A-D). Test # 1 has simple geometry but infinite possible reflections. It is an ideal way to compare different regeneration (and path tracing in general) implementations efficiencies and overheads with no relation to ray tracing performance. Test # 2 is simple in all senses and should not benefit of regeneration at all. It helps us to measure pure regeneration overhead. Tests # 3 (specular/glossy walls) and # 4 are typical real-word examples when path regeneration is applied due to high maximum trace depth.

### Comparison to previous approaches

We compare our approach to Streaming PT from [5] (call it “sm” on Fig. 2-5) and path tracing with no regeneration at all (“no regeneration”). For both Streaming PT and “no regeneration” PT we used per block Russian roulette. For Streaming PT we used indices instead of moving per-ray data and each 4-th/8-th bounce for regeneration to have approximately the same regeneration kernel call number on each test scene that we got in our implementation. Otherwise, (each bounce regeneration) Streaming PT costs much and gives us no benefit at all. We used “thrust::copy\_if” implementation of Streaming compaction in CUDA and self-implemented compaction based on NVSDK prefix sum sample in OpenCL.

Compared to [5] our algorithm has same performance gain (Test 3, 4) from regeneration but it has lower overhead (Test 1, 3, 4). Main reasons for this are points (1-2), (4), (5). Blocks/tiles (1-2) reduce operation cost by factor of 256. It allows in-place regeneration, though, no per-ray data movement or indices is required. Blocks sorting (4) decreases total kernel calls number. Threshold base regeneration (5) additionally decreases regeneration code self-cost. Besides performance benefit, our algorithm is simpler for implementation because it doesn’t change actual thread indices for rays. Once a tile comes into the thread pool, all its rays gain fixed thread index which will not be changed during any number of bounces. So, per-ray data could be stored at fixed memory locations.

## Conclusion

Our experiments showed that path regeneration gives essential benefit only for large trace depth values (greater than 20). For low trace depth values our results in general are consistent with Wald experiments [7].

Thus we believe that reduction of thread divergence within warps (solved via regeneration in [4], [5]) could be successfully solved with per-warp/per-block Russian Roulette only. Path regeneration is needed only when active warps number became lower than the value of warps that GPU process simultaneously with high actual occupancy.

## References

- [1] K. Gupta, J. A. Stuart, J. D. Owens. A Study of Persistent Threads Style GPU Programming for GPGPU Workloads // In proceedings of Innovative Parallel Computing. May 2012. San Jose, CA.
- [2] Frolov V., Kharlamov A., Ignatenko A. Biased Global Illumination via Irradiance Caching and Adaptive Path Tracing on GPUs. // Proceedings of GraphiCon’2010 international conference on computer graphics and vision. St. Petersburg, 2010, pp. 49–56.
- [3] S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison, and M. Stich. OptiX: a general purpose ray tracing engine // In ACM SIGGRAPH 2010 papers: ACM, NY, Article 66, 13 pp.
- [4] Nock, J., Havran, V., and Daschbacher Path regeneration for interactive path tracing // In EUROGRAPHICS 2010, short papers, pp.61–64.
- [5] Van Antwerpen. Unbiased physically based rendering on the GPU // M.S. thesis, Delft University of Technology, the Netherlands, 2011.
- [6] Tomas Davidovic, Jaroslav Krivanek, Milos Hasan, and Philipp Slusallek. Progressive Light Transport Simulation on the GPU: Survey and Improvements // ACM Trans. Graph. 33, 3, Article 29 (June 2014), 19 pages.
- [7] Wald, I. Active thread compaction for GPU path tracing // High Performance Graphics, 2011, pp. 51–58.
- [8] Laine, S., Karras, T., and Aila, T. Megakernels considered harmful: Wavefront path tracing on GPUs // Proc of High-Performance Graphics 2013, pp. 137–143.
- [9] Novak J. Global Illumination Methods on GPU with CUDA // MS Thesis: Ph. D. thesis.: Czech Technical University, Prague, 2009.
- [10] NVIDIA Mental Ray // 2014. <http://www.nvidia.com/object/nvidia-mental-ray.html>

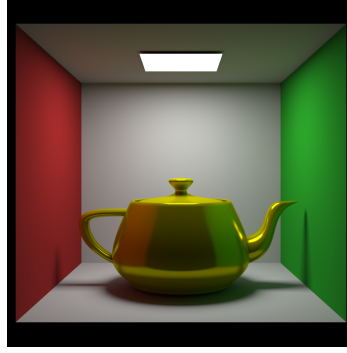
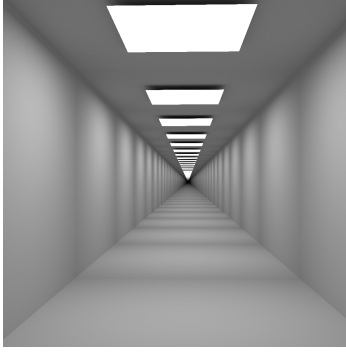


Figure A. Test 1, mirror corridor Figure B. Test 2, cornel box. Figure C. Test 3, arch. Figure D. Test 4, glass.

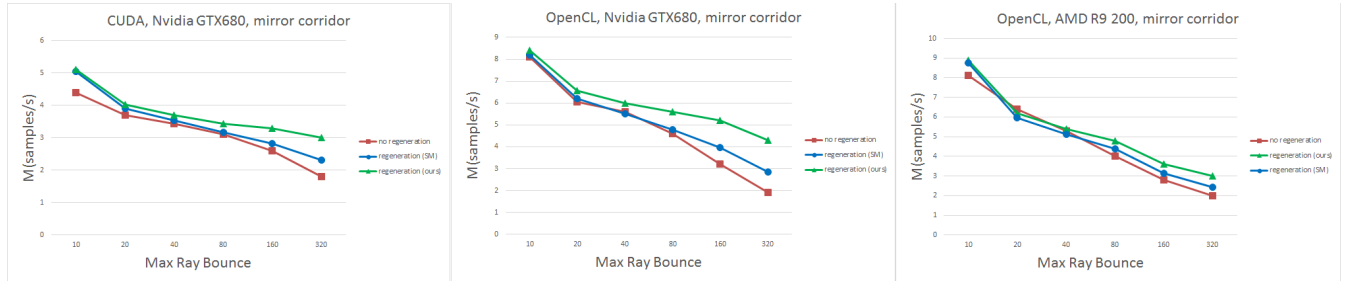


Figure 2. Test 1, mirror corridor paths per second (referred to fig. A).

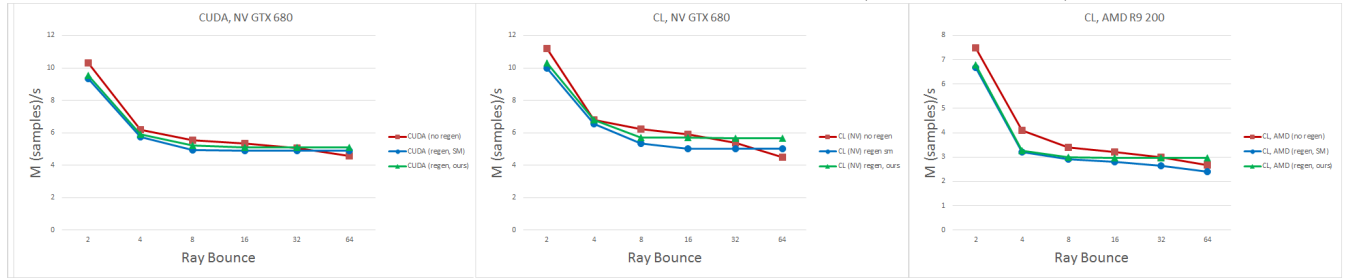


Figure 3. Test 2, cornel box paths per second (referred to fig. B).

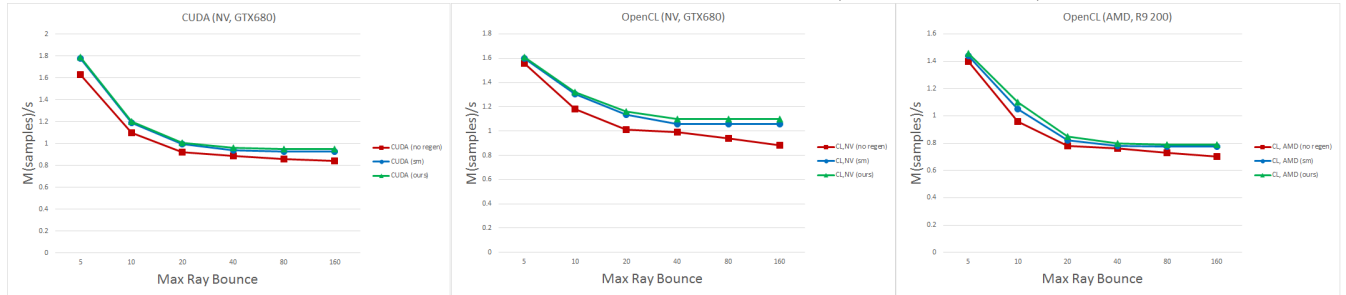


Figure 4. Test 3, arch\_1; paths per second (referred to fig. C).

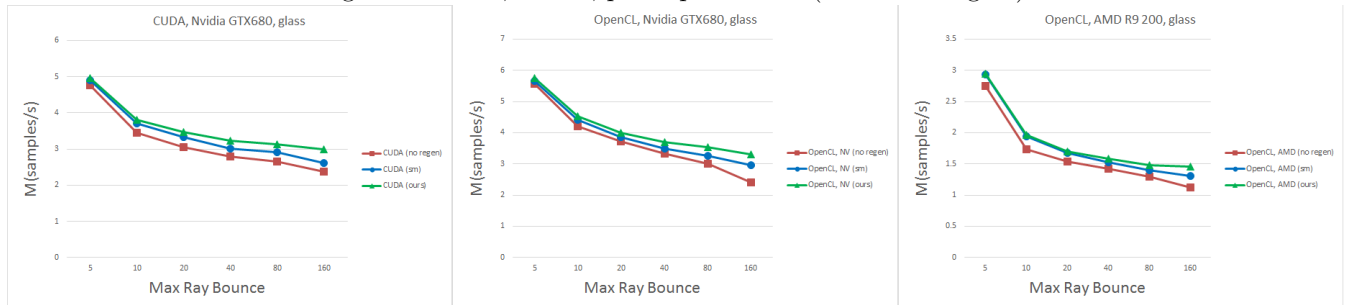


Figure 5. Test 4, glass; paths per second (referred to fig. D).