Boosting Histogram-Based Denoising Methods with GPU Optimizations

Sebastian Szeracki[*], Thorsten Roth[*†], André Hinkenjann[*], Yongmin Li[†]

[*] Hochschule Bonn-Rhein-Sieg
Grantham-Allee 20, 53757 Sankt Augustin
Germany
sszera2s@smail.inf.h-brs.de
{troth2m | ahinke2m}@h-brs.de

[†] Brunel University London
Uxbridge, Middlesex, UB8 3PH
United Kingdom
yongmin.li@brunel.ac.uk

Abstract: We propose a high-performance GPU implementation of Ray Histogram Fusion (RHF), a denoising method for stochastic global illumination rendering. Based on the CPU implementation of the original algorithm, we present a naive GPU implementation and the necessary optimization steps. Eventually, we show that our optimizations increase the performance of RHF by two orders of magnitude when compared to the original CPU implementation and one order of magnitude compared to the naive GPU implementation. We show how the quality for identical rendering times relates to unfiltered path tracing and how much time is needed to achieve identical quality when compared to an unfiltered path traced result. Finally, we summarize our work and describe possible future applications and research based on this.

Keywords: Computer Graphics, Global Illumination, CUDA, Image Processing

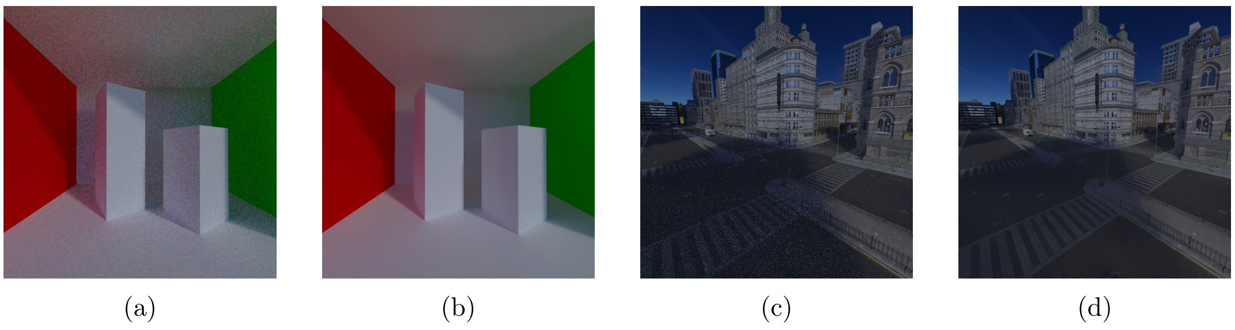


Figure 1: Ray Histogram Fusion (RHF) applied to two images, (a) Cornell Box scene rendered at 64 samples per pixel, no filtering applied, (b) Same image with RHF, (c) Urban Sprawl scene without filtering at 512 samples per pixel; noise from indirect illumination is clearly visible. (d) Same image with RHF applied; the noise has completely disappeared, only slight artifacts have been introduced.

1 Introduction

Global Illumination methods have been around for several decades. Ray-based approaches partially considering global illumination first came to practical use with Turner Whitted's work on recursive ray tracing [Whi79]. Later, Kajiya introduced the Rendering Equation and a statistical approach for solving a large part of it based on Monte Carlo simulation [Kaj86]. This approach is generally called *Path Tracing (PT)* and can deliver photorealistic images, assuming that physically-based materials and highly detailed geometric information about a scene is available. However, Path Tracing itself is a computationally expensive process, which means that computing high-quality images can take from seconds to days even on an up-to-date rendering workstation, strongly depending on the scene's geometry, illumination and material properties. Much work has been done in order to improve the image quality and convergence rate of Monte-Carlo-based methods: While *Bidirectional Path Tracing (BDPT)* [Vea97, Laf93] extends PT with paths originating at the light sources, more recent approaches also rely on hybrid methods like the unification of BDPT and Stochastical Progressive Photon Mapping [GKDS12].

However, even the most recent methods still rely on Monte Carlo simulation and thus also suffer from noise resulting from this complex, high-dimensional simulation process. As a consequence, much work has been done in the field of filtering this kind of noisy images. Ray Histogram Fusion (RHF) [DMB$^+$14] is one of the developed approaches. Its basic idea is to filter an image based on histogram similarities between neighboring pixels.

The remainder of this paper is organized as follows: We give a general overview of related filtering approaches in section 2, with the focus being put on RHF in section 3, as this is the method we developed an efficient GPU implementation for. In the corresponding subsections we describe the naive GPU implementation and its (performance) issues and present suitable optimizations for increased performance.

In section 4, we compare our optimized GPU implementation to the naive GPU implementation and the original CPU implementation regarding performance and potential visual differences. We also compare the different stages of our optimizations, so it becomes clear which optimizations yielded the biggest performance improvements. Section 5 gives a summary of the contents of this paper and discusses potential extensions and implications.

2 Related Work

Several filtering methods for noisy images have been published in the more general photographic field as well as in the more specialized global illumination field. While most of these methods work in image space, some also rely on additional information like samples from a scene's light field.

He et al. [HST10] present a filtering algorithm based on a so-called *Guiding Image*, which is applied to path tracing specifically by Bauszat et al. in [BEM11]. Sen and Darabi present *Random Parameter Filtering* [SD12], a method for estimating the functional relationship between the chosen random numbers and the resulting sample values. This information is used for parameterizing a cross-bilateral filter in image space. Lehtinen et al. [LALD12] show how the indirect light field can be reconstructed from a relatively low number of input samples without using any geometry information, but also accounting for occlusion using their own visibility heuristic.

Li et al. [LWC12] apply Stein's Unbiased Risk Estimator (SURE) in order to compute more effective, non-symmetric filter kernels as well as adaptively sample the image plane.

Mehta et al. present their axis-aligned filtering methods in [MWRD13] and [MYRD14], with the former concentrating on the extension of previously developed sheared and axis-aligned filters and the latter discerning between primary and secondary distribution effects for filter parameterization and adaptive sampling.

Moon et al. [MCY14] show their results on the application of local regression theory to image-plane adaptive sampling and filtering. Bauszat et al. [BEJM15] focus on denoising images containing depth-of-field effects at low sampling rates while aiming for interactive frame rates.

3 Ray Histogram Fusion

Recently introduced by Delbracio et al. [DMB$^+$14], Ray Histogram Fusion is a filtering method specifically designed for Monte-Carlo-based rendering processes. It is inspired by the non-local means filter first discussed in [BCM05]. The basic idea is to search for similar pixels in the neighborhood and to combine them if they meet some defined criteria. This idea can be further improved by not only comparing individual pixels, but groups of nearby pixels, forming so-called patches, in order to maintain texture and geometry information within the image domain.

The problem with filters like non-local means (NL-means) is their denoising of an image being based purely on its color information. This provides a solution which can lead to false assumptions and is therefore not the ideal solution for Monte-Carlo-based renderers. Because we have the option to use and generate more information during the rendering process than the sole pixel colors, we can also use more reliable information for filtering. Histograms, categorizing the sample color information for each pixel, lead to a far more reliable source of information. By limiting the histogram to a pre-defined number of bins, the computation time can be bounded, making it independent of the number of total samples used in each pixel.

For the filtering algorithm itself, each histogram of a patch is compared to the histograms in other patches from a defined neighborhood within a certain search radius. For the comparison we are using the χ^2 distance to evaluate the similarity between two patches. If the result falls below a certain threshold, the pixel colors of the patches are assumed to be suited for combination. This process is continued until all patches within the search window have been evaluated.

Because low-frequency noise cannot be captured by this algorithm, downsampled versions of the original noisy image need to be evaluated additionally and merged with the finer scale output to filter out this noise.

3.1 (Naive) GPU Implementation

We started with a direct CUDA port of the published code from Delbracio's CPU implementation [Del14] written in C++, which contains no special optimizations, to guarantee a correct version which produces the same result as the CPU counterpart. Below, we showcase the transition of the original CPU code to an efficient GPU implementation and describe the actual differences and limitations of modern GPUs. We provide information on how to overcome these limits and exploit the strengths of this kind of architecture.

3.2 Suited Optimizations

Many problems that used to be solved on the GPU are memory bound and our implementation is no exception to this. The main problem is the histogram data which is repeatedly read from the GPU's global memory for each comparison because of the overlapping search windows of each patch and even within the patches themselves. Therefore the memory bandwidth quickly reaches its limit and the code execution has to wait for the data to arrive.

To achieve better memory coalescing we have to consider the total number of bins per histogram. We chose to use 8 bins per color, which leads to a total bin size of 24. It would be possible to fit 8 extra values to have a perfect alignment of 32 values and therefore 128 byte cache line size, but this would also lead to a much higher memory consumption and a greater number of requests in total. This would have a high negative impact on the memory bandwidth and would thus result in a negative performance impact.

Because of the limited memory bandwidth we also have to reduce the redundant global memory reads as much as possible because of their high computational cost. This can be achieved by sharing as much data between threads as possible. One possible solution to this is using the much faster shared memory to store the data needed by a block of threads in advance. The main challenge here is working with the limited size of the shared memory (48 KiB per block on modern devices). Due to the size of the histograms the shared memory cannot hold all values for each search window within a thread block simultaneously. To get most out of the shared memory we have to adjust the block size to reduce the number of required histograms that are needed at once and fit best in the limited space.

Because the shared memory only leads to faster loading time within a block, the redundancy between blocks cannot be optimized by this method. Since the Kepler architecture the L1 Cache is deactivated for global memory access and only activated for local memory by default. So it has to be explicitly activated on the device during the compiling step. The global cache also needs to be ideally exploited to achieve better overall loading times. Because of the parallel nature of the GPU one can assume that a high utilization leads to better performance.

However, this is only true if there are no other limiting factors like memory bandwidth. The cache efficiency drops noticeable with a higher utilization because the potential cache that each thread can use decreases. By reducing the number of active blocks (and in turn also reducing the utilization) the cache can be used more efficiently by these blocks, leading to a performance improvement.

Another approach is to use half float precision for our histograms, because this data does not rely on high precision. It should be possible to achieve a nearly equal quality with reducing the memory space, and therefore the required bandwidth, by half.

3.3 Implementation Details

Because our window search does not need all histogram data at once and because of the linear search we exactly know when specific parts of the data are needed. With this in mind we adjust our block layout so that only a small window of histograms has to be available at once (see Figure 2). When a row in the search window is processed, we replace the unneeded row with a new one on demand and continue the search. This leads to a circular reuse of the shared memory space. Our first attempt at optimizing the code itself was to use shared memory to replace the global memory accesses within a block, as with a patch size of 3×3 up to eight of the nine histograms would be reused in the next step on a thread within a window search. Therefore using the shared memory, which has a much smaller latency than global memory, should lead to a huge performance boost. As mentioned earlier only 48 KiB shared memory is available per block.

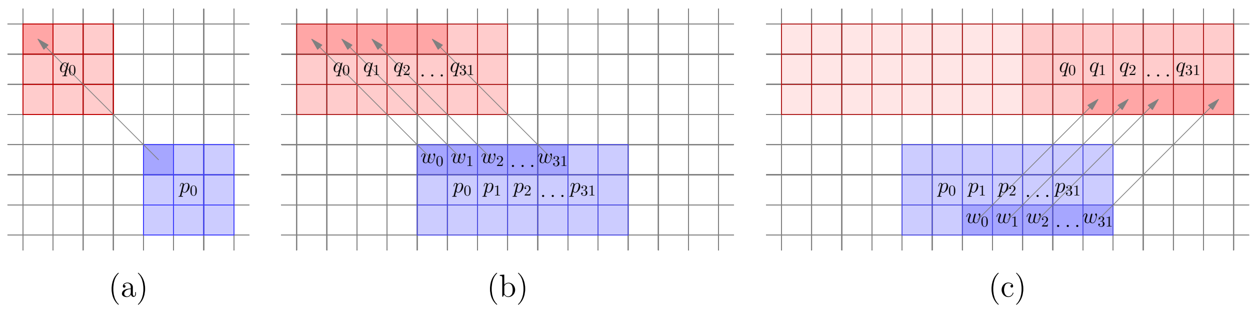

q_0
p_0
(a)
q_0 q_1 q_2 ... q_{31}
w_0 w_1 w_2 ... w_{31}
p_0 p_1 p_2 ... p_{31}
(b)
q_0 q_1 q_2 ... q_{31}
p_0 p_1 p_2 ... p_{31}
w_0 w_1 w_2 ... w_{31}
(c)

Figure 2: Access Pattern of a warp, showing the accesses a thread makes during the search of the first row of histograms. (a) The relation between the source patch p_0 and the target patch q_0 is shown. (b) The combined relation within a warp. (c) illustrates all histograms that are accessed if the access pattern is continued until the last patch of the row. These need to be stored in shared memory before this row is processed.

With a size of 8×3 bins each histogram consumes 96 Bytes of memory space. To calculate a patch distance for a patch size of $w = 3$ we need to load 3×3 histograms additional to the patch we want to denoise. Based on the window size b we can extend the calculation of the data size T needed in Bytes for a block size of $b_x \times b_y$ which leads to the folloing equation:

$$T = ((b + w) - 2 + b_x) \cdot (w + b_y - 1) \cdot 96.$$

To optimize the utilization of the shared memory, the block size needs to be adjusted in order to allow for a better data reusage and minimize waste of memory in total. In our tests a block size of 64×4 pixels, a patch size of $w = 3$ and a window size of $b = 13$ yielded the best performance and a memory usage of 44928 Bytes, which is approximately the amount of space we can allocate for a block. If we store each histogram in a row we would get a bank conflict on each third histogram in a warp. To reduce such conflicts to a minimum we add one additional value after each histogram which results in a total amount of 100 Bytes per histogramm and a total usage of 46800 Bytes, which still is within the bounds of 48 KiB per block.

Guided by CUDA's Visual Profiler, we are able to find an additional problem within our implementation. Because we are using a tracker to track all patch distances within a search window per thread, register spilling occurs. This leads to additional latencies outside of the normal histogram comparison. Because the original algorithm stores k results of the nearest neighbor search performed on the highest scale we do not need to store all results, but only k values. Additionally we combine the results of the pixels on-the-fly every time the distance lies below the threshold. This optimization completely resolves the register spilling.

Bank 00	Bank 01	Bank 02	Bank 03	Bank 04	Bank 05	Bank 06	Bank 07	Bank 08	Bank 09	Bank 10	Bank 11	...	Bank 31
$h_0(0)$	$h_0(1)$	$h_0(2)$	$h_0(3)$	$h_0(4)$	$h_0(5)$	$h_0(6)$	$h_0(7)$	$h_0(8)$	$h_0(9)$	$h_0(10)$	$h_0(11)$	...	$h_0(31)$
$h_0(32)$	$h_0(33)$	$h_0(34)$	$h_0(35)$	$h_0(36)$	$h_0(37)$	$h_0(38)$	$h_0(39)$	$h_0(0)$	$h_1(1)$	$h_1(2)$	$h_1(3)$	...	$h_1(23)$
$h_1(24)$	$h_1(25)$	$h_1(26)$	$h_1(27)$	$h_1(28)$	$h_1(29)$	$h_1(30)$	$h_1(31)$	$h_1(32)$	$h_1(33)$	$h_1(34)$	$h_1(35)$	...	$h_2(15)$
$h_2(16)$	$h_2(17)$	$h_2(18)$	$h_2(19)$	$h_2(20)$	$h_2(21)$	$h_2(22)$	$h_2(23)$	$h_2(24)$	$h_2(25)$	$h_2(26)$	$h_2(27)$	...	$h_3(7)$
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Figure 3: Shared Memory Layout. Each histogram value $h_i(x)$ where i denotes the bin and x denotes the position is stored in a way so that each thread in a warp accesses a different bank in order to avoid bank conflicts.

Since the Kepler Architecture the L1 Cache for global memory is deactivated by default and reserved for local memory. It has to be activated with a compiler flag (`-Xptxas -dlcm=ca`) or with a PTX instruction to allow for a single transaction to use the cache [NVI]. With the usage of shared memory we gain a much better bandwidth within a thread block but lack the ability of reusing the data between neighboring blocks. The global L1 Cache allows for better bandwidth usage due to holding the data used in another block before. However, due to the large amount of data the cache cannot hold much of it at once and with high utilization the cache lines are invalidated before all threads can use the data, which leads to a lower bandwidth. With a lower utilization we have a lower latency due to high hit rates in the cache and an overall better performance.

Though, even with all these optimizations the memory latency is still too high, so we looked a bit deeper into the memory access patterns. The shared memory layout still leads to bank conflicts and therefore the layout must follow the access pattern of the warps themselves. Because each thread accesses the same bin of the histogram at the same time we can exploit this property by aligning the shared memory bin-wise and not histogram-wise. The resulting histogram layout is shown in figure 3. This layout leads to no bank conflicts during reads within a warp, but it leads to bank conflicts during the storing process. To solve this problem we simply changed the histogram layout in the global memory in the same way. This does not only lead to a better performance during our storing and reading process but also an overall performance increase, because the algorithm does not have to load all values of each histogram in advance, but only the ones needed for the current comparison. This leads to decreased memory latencies.

With the use of half float precision for histogram values we are able to achieve even better results. To make this possible we use CUDA intrinsics to cast a float value to a half float value (`__float2half_rn()`) and vice versa (`__half2float()`). Because it is not possible to directly use arithmetic operations with these values, they need to be transformed back to float values to compute the distance which leads to some overhead. Still, in comparison to the performance gain, this impact is negligible. In addition to a smaller size per histogram we had to once again compare different block sizes for the optimal performance. This leads to a block size of 64×2 pixels, resulting in 11232 KiB per Block. Measurements have shown differences between single precision floats and half floats to be negligible.

4 Results

The runtime is only marginally affected by the chosen scene and scales linearly with the image resolution so there is no difference between the results regarding these settings. However, we provide results for two scenes, namely a standard Cornell Box with environmental lighting as well as the Urban Sprawl 2 scene by Stonemason. We used a GeForce GTX 970 GPU for our performance benchmarks at an image resolution of 1280×720 pixels and compiled the program with the Linux version of CUDA 7. Futhermore the runtimes used are those for the biggest scale only, because the other scales also behave linear in respect to their runtime.

The comparison of the runtimes of each optimization step discussed in section 3.3 is shown in figure 4. The runtime of the CPU implementation for the biggest scale was about 39.8s on an Intel i7 960 (4 Cores with 3.2 GHz). Figures 5 and 6 show comparisons in rendering quality between the RHF-filtered and unfiltered results. These tests were performed at a resolution of 640×640 pixels. All error values are based on the comparison to a reference image (32768 samples for Cornell Box, 524288 samples for Urban Sprawl). The error measure we used is Root Mean Square (RMS), which we adapted to be evaluated in a block-wise fashion for a regular image subdivision so the error values can be associated with image areas and a single, large error in the image does not influence the error measure for the whole image. The RMS error is based on radiance values taken directly from the path tracer, without post-processing of any kind. In order to perform the comparison, we chose an RHF-filtered result which showed no visible artifacts without directly comparing it to the reference image. Then we compared the block-wise RMS error to unfiltered renderings until the mean over all blocks

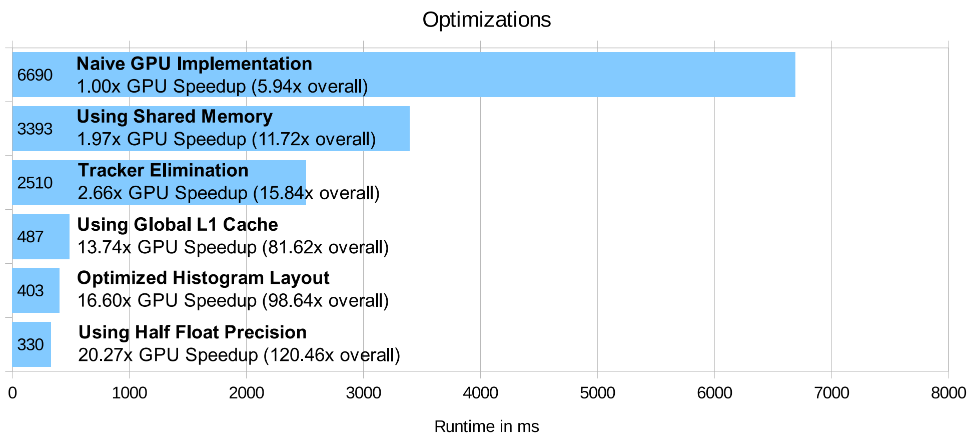


Figure 4: Comparison of the GPU optimization steps on an NVIDIA GTX 970 GPU starting with the naive GPU implementation as a starting point. Each implementation step includes all previous optimizations.

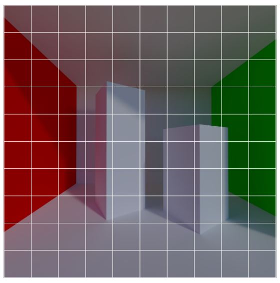
was similar. For this number of samples, we also show the block-wise RMS error compared to a reference image for the RHF-filtered result, as the rendering time is roughly equal to the original rendering because of the high performance of our implementation (execution time of RHF, which is approx. 200ms for 640×640, is negligible compared to pure rendering time).

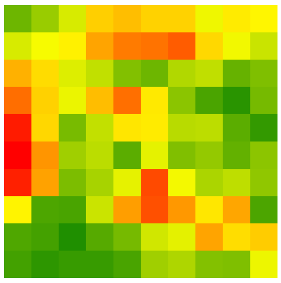
The false-color images in Figures 5 and 6 depict areas with a relatively large error in red, medium errors in yellow and small error values in green. The colors have been computed based on the RMS error values shown in Figures 5e and 6e, respectively, by normalizing them to $[0, 1]$ with the error range taking into account all images, i.e., all images are normalized based on the same interval for better comparison. This representation makes it easier to determine image regions which are difficult to compute and which benefit most from applying RHF.

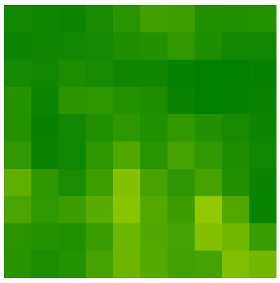
Figure 5e shows that the RHF-filtered result at 1024 samples exhibits a much lower RMS error for all blocks compared to the unfiltered result. However, figure 6e shows that the RHF-filtered result at 32768 samples is better regarding its RMS error for the lower half of the image, but even tends to be a little worse than the original rendering in the upper half. This is most probably because the upper half of the image is converging relatively quickly and thus shows not much noise early in the process, so that filtering mostly blurs out fine illumination details. A variance-based blending of unfiltered and filtered results could help reduce this behaviour.

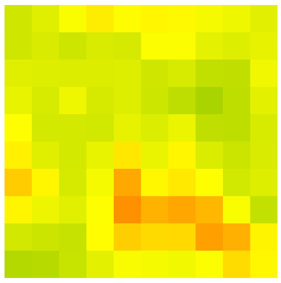
5 Conclusion and Future Work

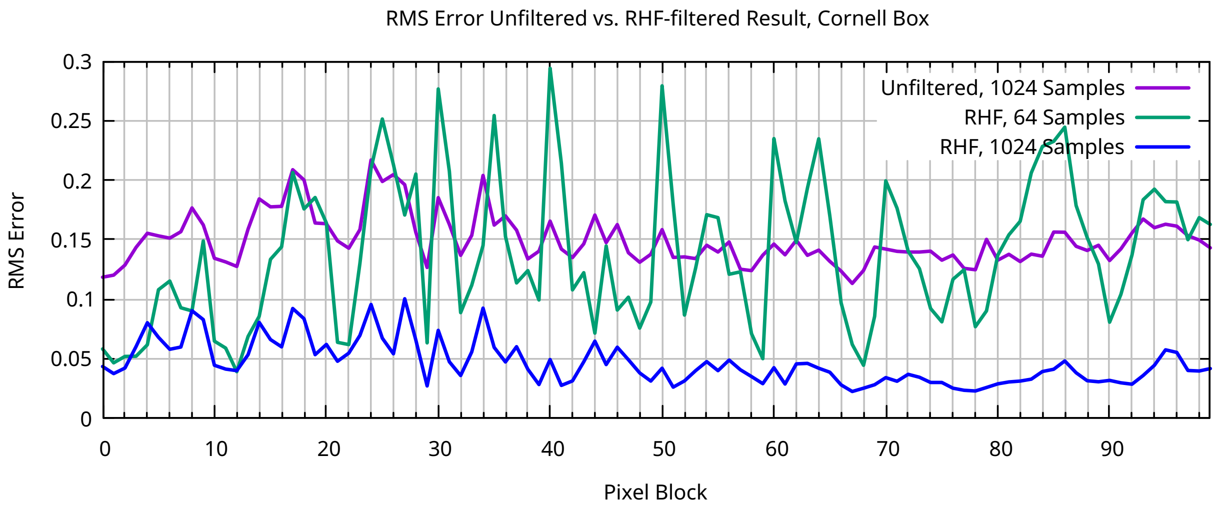
In this paper, we have presented a high-performance GPU implementation of a histogram-based denoising method for stochastic global illumination rendering. Based on the original CPU implementation, we have created a naive port to the GPU and performed a set of optimization steps leading to vast performance improvements. Eventually, depending on the resolution and the available hardware, our implementation reached a state in which it can even be used in an interactive (though not real-time) system. The filtering times of our optimized implementation are mostly negligible when compared to the pure rendering times. Also, judging from the block-wise RMS error comparison shown in section 4, a quality similar to a much higher number of samples can be achieved in far less time, even when rendering and filtering times are combined. As for future applications, we plan to employ filtering methods like RHF in rendering setups based on large, high-resolution displays in order to be able to quickly deliver high-quality rendering results. As RHF depends on a relatively large number of samples, combinations with other filtering approaches might be necessary. Thus, methods have to be developed to determine how well some specific filtering algorithms are suited to process the data at hand.


(a) Reference rendering (32768 samples per pixel, 1365s)


(b) Error image, RHF, 64 samples

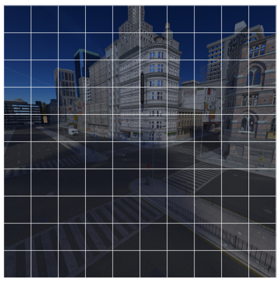

(c) Error image, RHF, 1024 samples

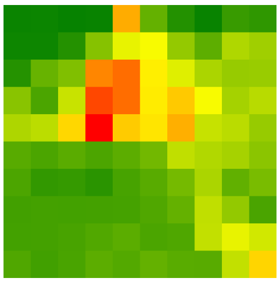

(d) Error image, unfiltered, 1024 samples

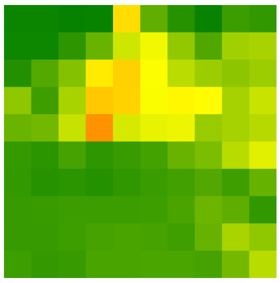


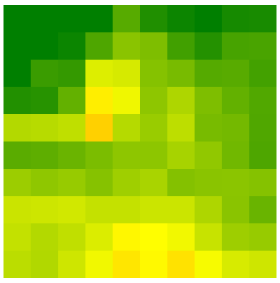
(e) Block-wise RMS error for the Cornell Box scene; on average, the unfiltered result at 1024 samples is roughly equal to the RHF-filtered result at 64 samples, but needs approx. 16 times the rendering time when using our GPU implementation of RHF.

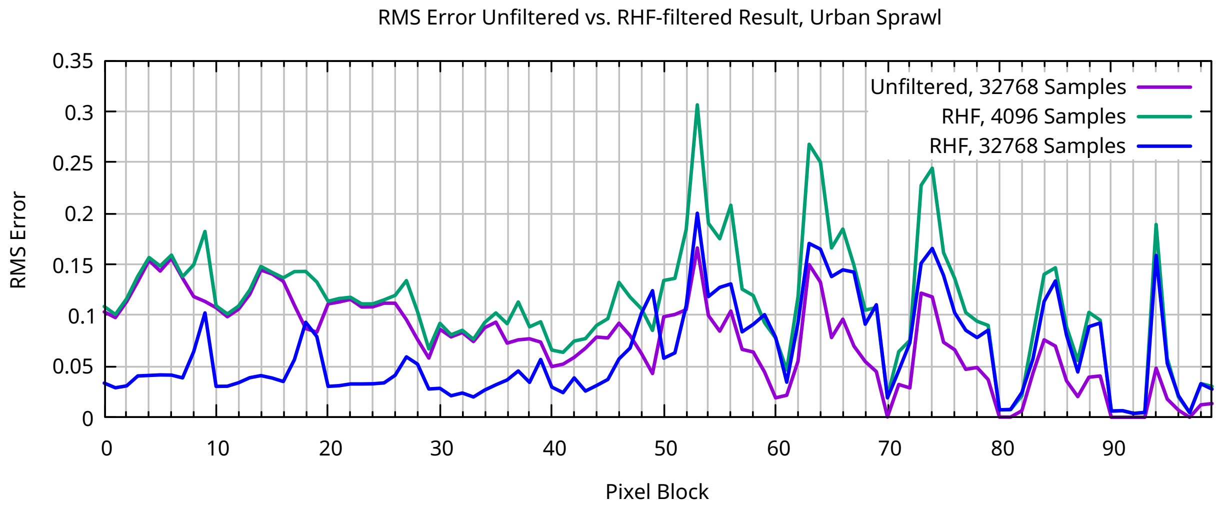
Figure 5: RMS Error comparison for the Cornell Box test scene; the overlay in (a) shows the grid subdivision used for block-wise RMS comparison, starting with block index zero at the bottom left and nine at the bottom right. The false-color images (b) to (d) allow for the easy identification of problematic image areas which benefit most from filtering.


(a) Reference rendering (524288 samples per pixel, approx. 9h)


(b) Error image, RHF, 4096 samples


(c) Error image, RHF, 32768 samples


(d) Error image, unfiltered, 32768 samples



(e) Block-wise RMS error for the Urban Sprawl scene; on average, the unfiltered result at 32768 samples is roughly equal to the RHF-filtered result at 4096 samples, but needs approx. 8 times the rendering time when using our GPU implementation of RHF.

Figure 6: RMS Error comparison for the Urban Sprawl test scene; the overlay in (a) shows the grid subdivision used for block-wise RMS comparison, starting with block index zero at the bottom left and nine at the bottom right. The false-color images (b) to (d) allow for the easy identification of problematic image areas which benefit most from filtering.

References

[BCM05] A. Buades, B. Coll, and J.-M. Morel. A non-local algorithm for image denoising. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2005. CVPR 2005*, volume 2, pages 60–65 vol. 2, June 2005.

[BEJM15] P. Bauszat, M. Eisemann, S. John, and M. Magnor. Sample-Based Manifold Filtering for Interactive Global Illumination and Depth of Field. *Computer Graphics Forum*, 34(1):265–276, February 2015.

[BEM11] Pablo Bauszat, Martin Eisemann, and Marcus Magnor. Guided Image Filtering for Interactive High-quality Global Illumination. In *Computer Graphics Forum*, volume 30, pages 1361–1368, 2011.

[Del14] M. Delbracio. Ray histogram fusion. `https://github.com/mdelbra/rhf`, 2014.

[DMB+14] Mauricio Delbracio, Pablo Musé, Antoni Buades, Julien Chauvier, Nicholas Phelps, and Jean-Michel Morel. Boosting Monte Carlo Rendering by Ray Histogram Fusion. *ACM Trans. Graph.*, 33(1):8:1–8:15, February 2014.

[GKDS12] Iliyan Georgiev, Jaroslav Křivánek, Tomáš Davidovič, and Philipp Slusallek. Light transport simulation with vertex connection and merging. *ACM Transactions on Graphics (TOG)*, 31(6):192, 2012.

[HST10] Kaiming He, Jian Sun, and Xiaoou Tang. Guided image filtering. In *Computer Vision–ECCV 2010*, pages 1–14. Springer, 2010.

[Kaj86] James T. Kajiya. The Rendering Equation. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '86, pages 143–150, New York, NY, USA, 1986. ACM.

[Laf93] Eric P. Lafortune. Bi-Directional Path Tracing. In *Proceedings of Compugraphics '93*, pages 145–153, 1993.

[LALD12] Jaakko Lehtinen, Timo Aila, Samuli Laine, and Frédo Durand. Reconstructing the indirect light field for global illumination. *ACM Transactions on Graphics (TOG)*, 31(4):51, 2012.

[LWC12] Tzu-Mao Li, Yu-Ting Wu, and Yung-Yu Chuang. SURE-based Optimization for Adaptive Sampling and Reconstruction. *ACM Trans. Graph.*, 31(6):194:1–194:9, November 2012.

[MCY14] Bochang Moon, Nathan Carr, and Sung-Eui Yoon. Adaptive Rendering Based on Weighted Local Regression. *ACM Trans. Graph.*, 33(5):170:1–170:14, September 2014.

[MWRD13] Soham Uday Mehta, Brandon Wang, Ravi Ramamoorthi, and Fredo Durand. Axis-aligned filtering for interactive physically-based diffuse indirect lighting. *ACM Transactions on Graphics (TOG)*, 32(4):96, 2013.

[MYRD14] Soham Uday Mehta, JiaXian Yao, Ravi Ramamoorthi, and Fredo Durand. Factored Axis-aligned Filtering for Rendering Multiple Distribution Effects. *ACM Trans. Graph.*, 33(4):57:1–57:12, July 2014.

[NVI] NVIDIA. Tuning CUDA Applications for Kepler.

[SD12] Pradeep Sen and Soheil Darabi. On Filtering the Noise from the Random Parameters in Monte Carlo Rendering. *ACM Trans. Graph.*, 31(3):18:1–18:15, June 2012.

[Vea97] Eric Veach. *Robust Monte Carlo methods for light transport simulation.* PhD thesis, Stanford University, 1997.

[Whi79] Turner Whitted. An Improved Illumination Model for Shaded Display. In *Proceedings of the 6th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '79, pages 14–, New York, NY, USA, 1979. ACM.