

# Optimizing Spatiotemporal Variance-Guided Filtering for Modern GPU Architectures

Rostyslav Pikulsky 

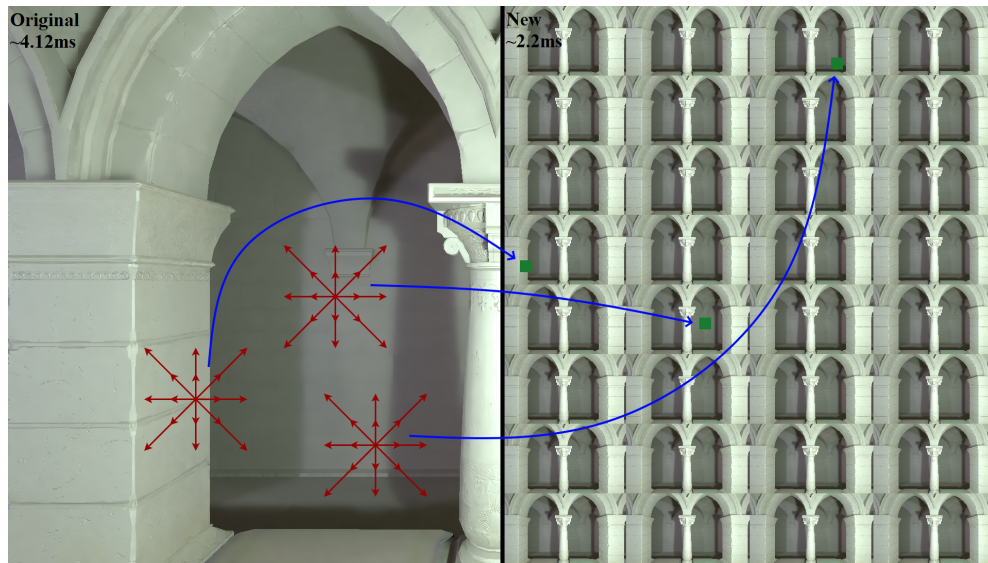


**Figure 1.** Primary idea of the optimizations proposed in this work. By rearranging data in memory improved execution efficiency and memory access performance may be achieved.

## Abstract

Path tracing is generally considered one of the most promising real-time rendering solutions. One of the major problems of its real-time application is large variance of the outputs, which is a result of the low number of Monte Carlo samples being processed due to execution time constraints. In response to that, real-time path tracing reconstruction (denoising) has become an important research direction. Spatiotemporal variance-guided filtering (SVGF) is one of the state-of-the-art real-time denoising techniques that serves as a basis for a lot of production solutions. We present a set of optimization techniques for the à-trous wavelet transform of SVGF. They achieve execution time reduction without any filtering quality degradation by optimizing memory access efficiency based on modern GPU architecture specifics. Compared to the base SVGF implementation, a  $1.3\text{--}2.5\times$  performance gain is achieved in different testing scenarios. Proposed techniques can be implemented for different hardware targets and usage scenarios and do not pose limitations to filter configuration.

## 1. Introduction and Related Work

Path tracing [Kajiya 1986] is currently one of the most preferable algorithms for photorealistic rendering. Being a spectral radiance Monte Carlo estimator, the algorithm can produce an unbiased radiance estimation with practically unrecognizable variance when enough samples (light paths) are processed. This quality has caused path tracing to gain significant popularity in animated movie production, film effects visualization, and other offline rendering scenarios. However, its applicability in interactive graphics is limited due to the high computational complexity of ray tracing operations. Although the introduction of hardware ray tracing support has increased ray tracing efficiency significantly on consumer graphics hardware, a typical interactive application’s ray tracing budget is still limited to several rays per pixel per frame, or one path per pixel per frame, which results in severe variance (noise) in the resulting image in typical rendering configurations.

This motivated the development of the path tracing reconstruction (denoising) filters—algorithms intended to interactively produce the estimation of the low-variance result for the frame given the varying path tracing output. Spatiotemporal variance-guided filtering (SVGF) [Schied et al. 2017] became a significant advancement in this direction, being one of the first reconstruction filters that can interactively perform decent-quality denoising on consumer graphics hardware. It is considered one of the state-of-the-art path tracing reconstruction filters, thus being used in various developments and production solutions [Schied et al. 2018; Oberberger et al. 2022; Hirvonen et al. 2019; AMD 2025; NVIDIA 2025]. Recently, the potential of neural-based ray reconstruction approaches [Burnes 2023; Kim et al. 2024] has been demonstrated; however, SVGF remains relevant as a simple and flexible alternative producing predictable results.

Despite SVGF being suitable for low-latency applications, using it in a real-time graphics engine is still hard due to large execution time requirements. The authors of the original work claimed a 10-millisecond execution time for FullHD resolution ( $1920 \times 1080$  pixels) on NVIDIA TITAN X. Although this number is likely lower for later hardware, the filter still consumes a significant portion of the 16-millisecond rendering time budget that 60 Hz interactive rendering implies. We are unaware of any specific SVGF optimizations; developers typically optimize the filter by reducing the number of operations performed. Classic examples of such modifications are quantizing the storage data to decrease video memory bandwidth, simplifying the arithmetic operations (for instance, edge-stopping function formulations), reducing the number of spatial iterations performed, etc. While helping to improve filter performance, typically these optimizations also decrease the output image quality and thus need to be carefully tuned to avoid extensive graphical defects.

This work presents novel optimization techniques to increase SVGF execution efficiency on modern graphics hardware. They are focused on reducing the execution

time of the spatial filtering, which is observed to be the most time-consuming stage of the filter [Schied et al. 2017]. Rather than changing the filter specifics and introducing quality-performance trade-offs, the proposed techniques improve the efficiency of the SVGF spatial filtering by leveraging the strengths of modern GPU architectures. Hence, they completely preserve filter output and can be used to improve application performance with no resulting quality sacrifice.

Our specific contributions include the following:

1. A definition of the special pixel processing layout that improves memory operations efficiency of the convolution kernels with non-one dilations;
2. Application of the abovementioned principle to SVGF spatial filtering, achieving optimal performance on modern graphics hardware;
3. A definition of the pixel processing method that decreases the memory bandwidth required to perform high-dilation convolution kernels and its application to corresponding à-trous spatial iterations.

Section 2 contains an overview of the SVGF implementation in the Falcor network we use as a base for our modifications; it also provides the performance testing results of the base implementation. Section 3 covers the actual optimization techniques and their application to the SVGF base implementation; Section 4 provides optimized implementation performance testing results.

## 2. SVGF Overview

### 2.1. Implementation Details

We use the Falcor framework [Kallweit et al. 2022] and its SVGF implementation for filter performance analysis and testing; we also use it to implement the optimizations we present and evaluate achieved performance gains. In this section, a brief overview of the Falcor SVGF implementation is provided. For detailed information about the algorithm, see the original paper [Schied et al. 2017].

The filter has the following configuration parameters:

1. *The number of spatial iterations to be performed:* The largest reasonable value is  $\lceil \log_2(\max(w, h)) \rceil$ , where  $w$  and  $h$  are the screen width and height in pixels, respectively; any further iterations are redundant since convolution would exceed the screen dimensions. This parameter acts as a trade-off between performance and spatial filter quality; increasing it enlarges the spatial filter footprint but slows down the filter execution. In the original paper, the authors considered five iterations to be optimal for up to  $1920 \times 1080$  resolution, which gives a footprint of  $65 \times 65$  pixels; however, this might be too low for higher resolutions or complex inputs. We test the filter for all possible iteration numbers to

provide a full picture of filter performance and present optimizations for different iteration number cases.

2. *Feedback data source*: This parameter defines the source of history data for the next frame's temporal filtering. It is possible to use the luminance resulting from the variance filtering stage or the output of one of the spatial iterations. This data provides a trade-off between the image noisiness and the filter's bias; the latter the stage whose result is used, the less noisy and more biased the feedback data is. The original paper's authors used the first spatial iteration's output as feedback. In initial implementation, changing this parameter does not affect the performance; however, it slightly affects optimized implementation performance.
3. *Color  $\sigma_l$  and normal  $\sigma_n$  edge-stopping parameters*: These are scalars used for  $\Delta$ -trous convolution weight calculations. The original paper defines these as  $\sigma_l = 128$  and  $\sigma_n = 4$ . There is also a depth edge-stopping parameter in the original paper  $\sigma_z = 1$ ; in Falcor this value is used implicitly. These parameters are used as arithmetic operations' arguments, so their values do not affect filter performance.
4. *Color and variance moments' history reuse factors*: These values determine how much history data is blended with the current frame's data in temporal filtering. They provide a trade-off between temporal stability and temporal lag of the filter result. In the original paper, these parameters were set to 0.2. Similarly to edge-stopping parameters, they do not affect filter performance because they are used for arithmetic calculations only.

We use a simplified renderer configuration for benchmarking compared to the original paper. Specifically, we do not perform separate filtering of direct and indirect illumination terms; instead, we filter the sum of these terms. The demodulated illumination is calculated as the difference of path tracer global illumination color and G-buffer emission divided by G-buffer albedo. This simplifies the SVGF integration and performance measurements as the filter is only executed once per frame. The described pipeline modifications only affect the SVGF inputs rather than changing the filter's performance.

The filter is performed as a sequence of full-screen passes representing the following stages:

1. *Depth/normal packing stage*: This consists of one full-screen pass packing the G-buffer depth and normal buffer data into one render target. It simplifies depth/normal data sampling and improves its efficiency by reducing the memory size and render target numbers to sample. The pass might be omitted if

similar packed data is already available (for example, similar packing may be performed during G-buffer generation).

2. *Reprojection stage*: This stage consists of one full-screen pass that performs illumination albedo demodulation and temporal filtering of illumination and variance moments data. The result is stored in three render targets: one for illumination and variance data, and two others for filtered moments and history length respectively. This stage needs some of the previous frame's data as input; this includes demodulated illumination (feedback data), moments, history length, and packed depth/normal data. The ping-pong principle is used to manage this data efficiently: for every data type, two ping-pong render targets are allocated and a special flag is declared indexing render targets of this pair. For every frame, the render target currently indexed by the flag is used as a read-only previous frame data storage; the other is used as current frame data storage, which gets populated by values calculated in one of the stages and might be used in the following stages. At the end of the filter execution, the flag is flipped so that in the next frame current data storage becomes previous, and previous may be processed as current storage. This is intended to avoid data copying between render targets, which would decrease filter performance.
3. *Variance filtering stage*: This stage has one pass that performs variance filtering for pixels that have small enough history length ( $< 4$  frames as defined in the original paper [Schied et al. 2017]). Variance filtering is performed through a  $7 \times 7$  bilateral filter of illumination and variance moments that uses illumination and packed depth/normal data for weight calculations. The output of the stage is a filtered illumination/variance render target.
4. *À-trous wavelet transform stage*: This stage contains one full-screen pass per every spatial iteration, so the number of passes equals the spatial iteration count. Each pass performs the following operations:
  - (a) First, there is a special check whether the pixel should be spatially filtered. It is common for denoising algorithms to allow skipping spatial filtering for some pixels (typically those that do not contain geometry hits); the Falcor SVGF implementation marks these pixels by writing negative depth values to the G-buffer. If the pixel depth check is not passed, the corresponding shader thread's input luminance and variance values are passed to the output unmodified, and spatial filtering is canceled. The same check exists in the variance filtering stage (in addition to the history length check) as the particular pixel's filtered variance is only required for spatial filtering.

- (b) If the check is passed, variance convolution is performed—the local variance estimate is calculated through a  $3 \times 3$  Gaussian kernel. This value is used in the luminance edge-stopping function for the à-trous wavelet transform.
- (c) Then, the à-trous wavelet transform is performed—this is a  $5 \times 5$  illumination and variance convolution that uses packed depth/normal and illumination data for weight calculations. The dilation (the horizontal and vertical distance between the kernel’s neighboring pixels) of the à-trous wavelet transform gets doubled every iteration—the dilation for the  $i$ -th iteration (where  $i$  is a 1-based index) equals  $2^{i-1}$ , so 1 for the first iteration, 2 for the second, 4 for the third, and so on.

Each iteration outputs the filtered illumination/variance values using the resulting illumination/variance data from the previous pass and the packed depth/normal data as input (the first iteration uses the variance filtering stage output illumination and variance). Similarly to the reprojection stage, ping-pong render targets are used for illumination/variance data; in this case, the render target indexing flag gets flipped after every spatial iteration pass. Also, feedback data copying is performed at this stage; depending on the configuration, the variance filtering stage output or one of the spatial iteration’s outputs gets copied to a special render target.

- 5. *Final modulation stage*: This stage is just one pass that performs albedo modulation of the spatial filtering stage output, which is the inverse of what the reprojection stage does with path tracer color output. The output of this stage is the output of the SVGF.

We made several enhancements to the original SVGF implementation. First, compute shader dispatches were used for screen passes instead of screen quad rasterization. This is essential for further optimizations, as it allows the use of the shared memory. Second, feedback data copying was eliminated; instead, we ensure that the feedback data render target is directly used as an output in the corresponding pass instead of one of the ping-pong ones. This is a simple improvement we rely on in optimized implementation. These changes were mirrored in the initial implementation to make the performance comparison more objective.

## 2.2. Performance Testing

We use the Falcor renderer in the default SVGF demonstration mode to test the filter’s performance. As mentioned above, it differs from the original proposal for filter application; however, it is sufficient for proper performance testing.

Performance testing is done in different configurations. Two graphics cards are used—AMD Radeon RX 7800 XT and NVIDIA RTX 2080 Super. For every GPU, we test static and dynamic scenarios: static scenario implies a static environment with a static observer camera; dynamic scenario implies a highly dynamic camera that is supposed to generate a high amount of disocclusions. We have not found significant performance differences when using different scenes, so we only provide measurements on the Crytek Sponza scene. Then, for every case, we test the filter with one to ten spatial iterations; other filter parameters remain constant with screen resolution being  $1920 \times 1080$  and the first spatial iteration’s output used as a feedback data source. Together this results in 40 testing configurations; for each, we measure the average GPU execution time for different SVGF passes and the filter overall.

Tables 1, 2, and 3 show the testing results. As can be observed, the biggest performance bottleneck of the filter is usually spatial filtering. The reason for this is a spatial convolution kernel that requires a large amount of memory loads to be performed. An iteration’s execution time varies depending on the iteration index. Starting from the first iteration, it gradually grows due to increasing dilation of the kernel, which causes cache misses. Conversely, large dilation decreases the total number of the memory operations required for the convolution since fetches become invalid due to sampling over the screen border; this results in significant efficiency improvement over the later spatial iterations. Another notable performance bottleneck is the variance moments’ filtering stage in a dynamic scenario; in this case, a large amount of pixel disocclusions causes a loss of the temporal history and requires expensive variance moments’ convolution to be performed.

| Hardware | 1    | 2    | 3    | 4    | 5    | 6    | 7    | 8    | 9    | 10   |
|----------|------|------|------|------|------|------|------|------|------|------|
| AMD      | 0.73 | 0.60 | 0.53 | 0.49 | 0.48 | 0.54 | 0.59 | 0.54 | 0.44 | 0.33 |
| NVIDIA   | 0.74 | 0.74 | 0.76 | 0.77 | 0.98 | 0.99 | 0.96 | 0.89 | 0.77 | 0.48 |

**Table 1.** SVGF spatial iteration execution times in milliseconds. A certain iteration’s execution time is mostly independent of the scene and filter configuration, so one timing per iteration index per hardware target is given.

| Hardware       | DNP  | R    | VFS  | VFD  | S1   | S2   | S3   | S4   |
|----------------|------|------|------|------|------|------|------|------|
| RX 7800 XT     | 0.12 | 0.54 | 0.12 | 1.78 | 0.73 | 1.33 | 1.86 | 2.35 |
| RTX 2080 Super | 0.17 | 0.65 | 0.17 | 1.47 | 0.74 | 1.48 | 2.24 | 3.01 |
| Hardware       | S5   | S6   | S7   | S8   | S9   | S10  | FM   |      |
| RX 7800 XT     | 2.83 | 3.37 | 3.96 | 4.50 | 4.94 | 5.27 | 0.16 |      |
| RTX 2080 Super | 3.99 | 4.98 | 5.94 | 6.83 | 7.60 | 8.08 | 0.21 |      |

**Table 2.** SVGF stages’ execution times on different hardware and conditions in milliseconds. DNP corresponds to the depth/normal packing stage; R is reprojection; VFS and VFD are variance moments’ filtering in static and dynamic scenarios, respectively; S1–S10 are spatial filtering configured to one to ten iterations, respectively; and FM is the final modulation stage.

| Iterations | AMD, Static | AMD, Dynamic | NVIDIA, Static | NVIDIA, Dynamic |
|------------|-------------|--------------|----------------|-----------------|
| 1          | 1.67        | 3.33         | 1.94           | 3.24            |
| 2          | 2.27        | 3.93         | 2.68           | 3.98            |
| 3          | 2.80        | 4.46         | 3.44           | 4.74            |
| 4          | 3.29        | 4.95         | 4.21           | 5.51            |
| 5          | 3.77        | 5.43         | 5.19           | 6.49            |
| 6          | 4.31        | 5.97         | 6.18           | 7.48            |
| 7          | 4.90        | 6.56         | 7.14           | 8.44            |
| 8          | 5.44        | 7.10         | 8.03           | 9.33            |
| 9          | 5.88        | 7.54         | 8.80           | 10.10           |
| 10         | 6.21        | 7.87         | 9.28           | 10.58           |

**Table 3.** SVGF total execution times on all test configurations in milliseconds.

### 3. Optimizations

#### 3.1. Remainder-Based Data Reordering

As described above, the SVGF spatial filtering takes excessive execution time. One reason for this is the increasing à-trous kernel dilation, which leads to poor sampling locality and dramatically increases the cost of data fetching. Also, large dilation limits the à-trous convolution data sharing between pixels processed in the same workgroup, which one could benefit from by reading all the data only once per workgroup to the temporary storage allocated in the shared memory [Podlozhnyuk 2007]. Let  $S$  be a dilation,  $W$  be a dimension of the 2D workgroup (assuming horizontal and vertical dimensions are equal), and  $R$  be a radius of the à-trous convolution kernel; if  $S < W$  then the iteration would require  $(2SR + W)^2$  samples per workgroup or  $((2SR + W)/W)^2$  samples per workgroup thread. Assuming the workgroup dimensions are  $16 \times 16$  threads, which is a typical value for similar cases, the first spatial iteration would require 400 samples per workgroup or 1.5625 samples per thread, the second would require 576 (2.25) samples, the third 1024 (4) samples, and the fourth 2304 (9) samples; starting from the fifth iteration, step size exceeds workgroup dimension and no further data sharing is possible. As we see, the amount of data that needs to be stored in group-shared memory and the number of samples per thread is growing fast, making group-shared memory usage only practical for a couple of early iterations. Changing workgroup dimensions does not help much; increasing the dimensions increases the required shared memory amount, while decreasing them increases the number of fetches each thread needs to do.

We address these issues by applying remainder-based reordering to SVGF spatial filtering. We define remainder-based reordering with divisor  $d$  of a given array as a stable sorting of its elements by the remainder of their initial array indices division by  $d$  (Figure 2). Given the array of the size  $N$  reordered with divisor  $d$ , the group of

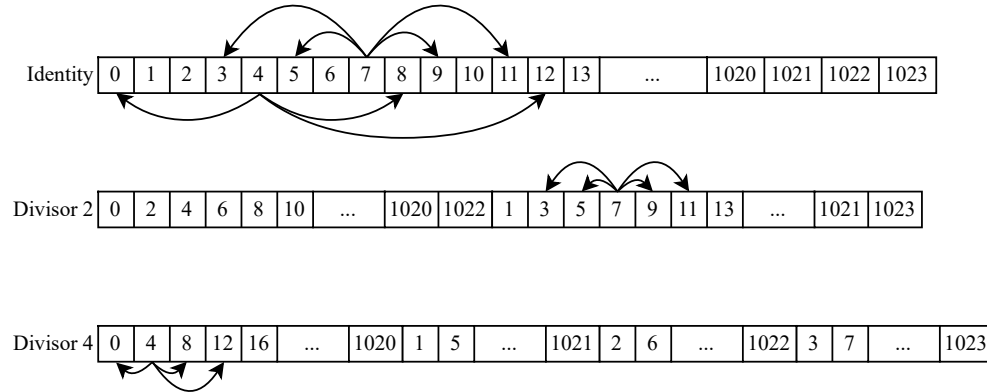


**Figure 2.** Remainder-based reordering with divisors 2 and 4, applied to an array with a size of 1024. The number inside the cell indicates the index of the cell in the non-reordered array. Arrows demonstrate how convolutions with different dilations apply to reordered arrays.

elements with a remainder of 0 after division by  $d$  is stored first ( $0, d, 2d, \dots$ ), the group of elements with remainder equal to 1 is stored next ( $1, 1 + d, 1 + 2d, \dots$ ), and so on up to the group of elements with remainder equal to  $d - 1$  ( $d - 1, 2d - 1, 3d - 1, \dots$ ). We call these reordering groups. Assuming  $d < N$ , the number of reordering groups equals the divisor  $d$ , and the size can vary depending on array size; the size of the first  $(N \bmod d)$  reordering groups equals  $\lceil N/d \rceil$ , while the size of the others equals  $\lfloor N \bmod d \rfloor$ . Note, that reordering with the divisor 1 (identity reordering) is identical to the original order of elements. Also, reordering with  $d \geq N$  is identical to the identity reordering.

As may be observed from the figure, remainder-based reordering can be extremely beneficial for convolutions with the dilation equal to the reordering divisor. When applying reordering to pixels on the screen both horizontally and vertically, the convolution is practically transformed into the convolution with dilation 1, which is performed inside every reordering group. This improves the sampling locality and allows efficient data sharing.

This optimization was inspired by [Bavoil 2014]. In that paper, the screen-space depth buffer is subdivided into four buffers each containing pixels that have the same remainders as its coordinates division by 2 ( $(0, 0)$ ,  $(0, 1)$ ,  $(1, 0)$ , or  $(1, 1)$ ), and these buffers are processed separately to generate a screen-space ambient occlusion effect [McGuire et al. 2012]. Such a separation improves sampling locality and performance, as the sampling radius is halved for half-dimension buffers. We extend this idea by using one buffer to store all the data instead of several smaller ones and by introducing an arbitrary divisor (we only use power-of-two divisors in practice, though).

To implement remainder-based reordering management, three shader functions were introduced to manipulate the elements' indices. The first function converts the given element index to the element index in the reordered array; the second per-

forms inverse converting. These two functions receive the given index, array size, and reordering divisor as inputs. The third function calculates the boundaries of the reordering group containing the pixel, given the pixel's reordered index, array size, and reordering divisor; it performs pixel coordinate out-of-bounds checks during spatial convolution. The implementation of these three shader functions in HLSL-based pseudocode is given in Listing 1.

```
uint PosReordToNorm(uint pos, uint d, uint size)
{
    uint size_overflow = size % d;
    uint group_size = size / d;

    if (pos < size_overflow * (group_size + 1))
        return pos / (group_size + 1) +
            d * (pos % (group_size + 1));
    else
        return (pos - size_overflow) / group_size +
            d * ((pos - size_overflow) % group_size);
}

uint PosNormToReord(uint pos, uint d, uint size)
{
    uint size_overflow = size % d;
    uint group_size = size / d;
    uint group_index = pos % d;
    return pos / d + (group_size + 1) * group_index -
        max(group_index - size_overflow, 0);
}

uint2 PosReordBounds(uint pos, uint d, uint size)
{
    uint size_overflow = size % d;
    uint group_size = size / d;
    if (pos < size_overflow * (group_size + 1))
    {
        uint lower_bound = pos - pos % (group_size + 1);
        return uint2(lower_bound, lower_bound + group_size + 1);
    }
    else
    {
        uint lower_bound = pos - (pos - size_overflow) % group_size;
        return uint2(lower_bound, lower_bound + group_size);
    }
}
```

**Listing 1.** Shader functions for manipulation of reordering indices.

### 3.2. Applying Reordering to À-Trous Wavelet Transform

We distinguish between two types of reordering applications: the execution and the storage reordering. Execution reordering defines the mapping between GPU dispatch threads and pixels being processed such that the thread index is the reordered index of the pixel screen coordinate. Defining the way pixels are distributed across workgroups, it has a crucial effect on data sharing efficiency, maximizing it for convolutions with dilation equal to the execution reordering divisor. Storage reordering defines the mapping between pixels and memory storage elements such that the index of the pixel data in memory is a reordered index of this pixel's screen coordinate. By changing it for a particular resource, one can tune the locality of corresponding memory operations; for convolution kernels, the optimal storage reordering divisor would be equal to the kernel dilation.

The variety of reordering definitions throughout the algorithm execution is limited. To change execution reordering, the current execution scope (i.e., variables) necessary for performing the following calculations should be transferred between threads of the dispatch. Therefore, it can be changed efficiently at the beginning of the dispatch when there is no relevant local data. Changing it in the middle of the dispatch is problematic, though, because it requires transferring the data through global memory (using shared memory is not sufficient since cross-workgroup transfer is required). Due to this, we define a distinct execution reordering for each separate dispatch. To change storage reordering, a full resource overwrite is required to transfer data between memory elements of the resource, which might be a notable workload. Besides, using ping-pong render targets is necessary for every resource that has variable storage reordering to prevent a data race during the resource overwrite. Changing storage reordering in the middle of the dispatch requires full resource memory synchronization, which would have a significant performance cost. Therefore, we define every resource's storage reordering per dispatch as well. By definition, the resource should be available as input in the specified reordering in the current ping-pong render target. If the next dispatch requires the resource in a different reordering, the current pass outputs the data to another ping-pong render target, and after dispatch the render target index gets flipped. Data output is performed by every dispatch thread writing the value of the output pixel to the corresponding memory location.

We decided to use execution reordering with the divisor equal to the à-trous convolution dilation for every spatial iteration. So the first iteration uses a divisor of 1 for reordering in both dimensions, the second uses 2, the third uses 4, and so on. As was mentioned above, such a scheme provides the best à-trous convolution data sharing and allows efficient shared memory usage. This decreases the data loading memory throughput by a huge margin and provides big performance benefits. The downside of this approach is that non-identity reordering practically changes spatial passes' variance convolution dilation, which originally always equals one. This de-

creases the variance convolution's efficiency for every spatial iteration except the first one; however, this flaw is outweighed by the performance gain achieved for à-trous convolution.

Spatial filtering uses two storage resources that we found to have different optimal reordering schemes. Illumination/variance data changes every spatial iteration, implying a full overwrite of the corresponding resource at the end of every spatial filtering dispatch. Therefore, changing its storage reordering is cheaper, as instead of adding a new memory write operation, it only requires changing the output coordinate of existing memory outputs. We configure the illumination/variance resource to use the storage reordering with divisor equal to the à-trous convolution dilation for every spatial pass, similarly to the execution reordering of these dispatches. This provides the best possible sampling locality when fetching illumination/variance data for spatial passes and minimizes the performance penalty for reordering changes as doubling the storage reordering divisor tends to be relatively fast. Note that, since the output of the first spatial iteration is used as feedback data for temporal filtering, this reordering scheme results in feedback data having non-identity storage reordering; we modify the reprojection stage accordingly to process feedback data in the corresponding state.

Depth/normal data is constant during spatial filtering; hence, an additional memory write operation is required to change the storage reordering of the corresponding resource. It is therefore necessary to balance between maintaining a data layout plausible for fetching and minimizing the number of reordering changes thus reducing their performance penalty. We perform depth/normal resource storage reordering once per two spatial iterations starting from the first one. For every such iteration, if it is the last one then reordering is not performed (as depth/normal data is not further required); otherwise, if the next iteration is last then the reordering divisor is doubled; otherwise, the reordering divisor is quadrupled. Hence, at least half of the iterations get depth/normal data in optimal reordering since it is identical to the execution reordering of the corresponding dispatch; others get this data reordered with the divisor twice as large as the corresponding execution reordering divisor, which produces a relatively plausible sampling pattern. According to our testing, this scheme produces good results across different hardware targets and filter configurations. Note that two additional storage images are required to be used as ping-pong render targets for depth/normal data.

While providing fine spatial filtering performance, the described scheme can cause severe inefficiency for the last iteration's resulting luminance store operation, since the final modulation stage requires the luminance result to have identity storage reordering, implying a large mismatch with the final iteration's execution reordering. To restrain this inefficiency, we modify the final modulation stage to support an arbi-

trary storage reordering for the filtered luminance input. The optimal divisor is chosen during filter initialization based on the following heuristic:

$$D = \begin{cases} d & d^2 \leq S, \\ 2d & d^2 > S, \end{cases}$$

where  $D$  is the luminance horizontal/vertical storage reordering divisor,  $S$  the corresponding screen dimension (width or height),  $d = 2^{N-1}$  the last iteration's execution reordering divisor, and  $N$  the number of spatial iterations. Note that, based on this method, we can get different horizontal and vertical reordering divisors. Also, we implemented an option to perform the final modulation stage in place at the end of the last spatial iteration's dispatch, which saves memory throughput by avoiding writing filtered luminance to global memory. We only use it when the identity storage reordering turns out to be optimal for filtered luminance in both dimensions (so either  $D = 0$  or  $D > S$ ) to avoid the overhead of the dispersed sampling of the diffuse albedo and emission data required for final modulation.

Figure 3 shows the reordering scheme example built for the filter configured to run in FullHD resolution and perform five spatial iterations. It is aimed to help with picturing and understanding the presented reordering management methods.

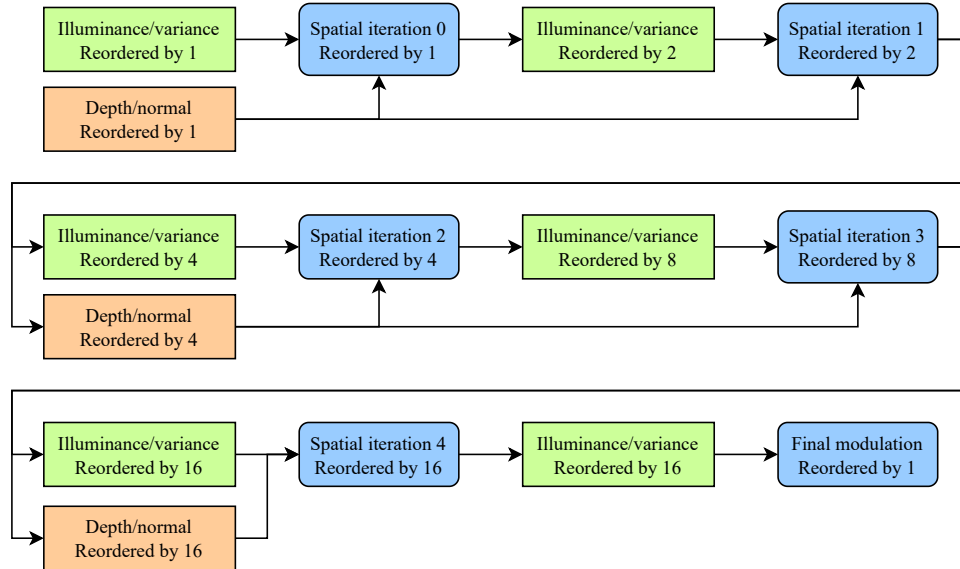


**Figure 3.** Reordering scheme for spatial filtering with five iterations in FullHD resolution. Rectangles represent resources, rounded rectangles represent execution passes, and arrows represent input/output relations between them.

### 3.3. Shared Memory Usage Implementation

As mentioned above, shared memory usage is an important part of SVGF optimization. Instead of performing a global memory load operation for every data bit required for a particular pixel, all data is loaded into a fast-access shared memory block and then used during processing. The population of the storage is distributed among the workgroup's threads such that no duplicate global memory loads are performed. Due to that, global memory access is reduced in the case that some data is shared between threads.

When applying remainder-based reordering, high data sharing is achieved for à-trous convolution in each spatial filtering iteration. Using a  $16 \times 16$  workgroup size, we only need a  $20 \times 20$  shared memory block to store data for a  $5 \times 5$  à-trous convolution, which reduces the global memory access amount from at most 6400 to 400 data fetches (where 6400 is 25 data fetches for each of the 256 workgroup threads). We store illumination, variance, depth, and normal values for every pixel in shared memory, which is sufficient for à-trous convolution. This data is also used for variance convolution in the first spatial iteration. It is also possible to use shared memory in the variance filtering stage. We can store all pixel data required for a  $16 \times 16$  workgroup in a  $22 \times 22$  shared memory block; global memory access is then reduced from at most 12,544 to 484 data fetches. In this case, shared memory stores illumination, depth, normal, and variance moments. The described global memory throughput reduction provides a major part of the achieved performance gain.

One issue of shared memory usage is that it is not suited well for frequent early exit cases. With that, every thread needs to perform corresponding data global memory loads in the beginning of the shader for the shared memory storage to be valid even if this thread does not need to perform the calculations for which the storage was allocated. Conversely, initial implementation performs an early-exit test immediately, which requires only the pixel depth to be loaded, and avoids any further operations in the case of an the early exit. Hence, the new implementation can fetch more data than the initial one when the amount of active threads in a workgroup is low.

However, this does not always result in a performance drop in practice. First, the number of active threads in a workgroup should be very small to achieve such a situation; it can be roughly estimated by the proportion between shared memory block size and convolution kernel size, which results in less than  $\lceil 400/25 \rceil = 16$  active threads for spatial iteration passes and less than  $\lceil 484/49 \rceil = 10$  active threads for the variance filtering pass. Second, the storage population workload is relatively small due to memory operations being non-duplicating and distributed across the workgroup threads; based on earlier discussion, each thread in fact has to load data for either one or two pixels. Overall, during our testing, we found that typically this problem is only crucial when all threads in the workgroup are inactive. Practically, this results in a

significant performance penalty in the variance moments' filtering pass in the static testing scenario, when no moments' filtering is in fact required.

To address this issue, a check for a fully inactive workgroup is performed before the shared memory population in the variance filtering stage. We define a boolean flag in shared memory and initialize it to "false." Then, each thread performs a partial early-exit check: history length value is fetched from global memory to check if it is low enough to perform variance moments' filtering. A full check would also require depth data loading to check whether the pixel should be filtered at all. If no early exit is required, the thread writes the value "true" to the boolean flag; this allows to determine whether there is any active thread in the workgroup without using relatively expensive atomic operations. Afterward, each thread checks the value of the flag and finishes execution if it is false, thus avoiding any additional overhead for fully inactive workgroups. It is still necessary to perform an early-exit check for every thread again if the workgroup is not fully inactive and to perform distinct early exits if needed.

To decrease the overhead of the workgroup inactivity check, we distribute the shared memory storage population such that every thread immediately fetches the data of the pixel it processes; then, data of the "frame" is fetched (see Figure 4). This allows reuse of data that might be loaded during the workgroup inactivity check for storage population, avoiding global memory load duplications.

The described techniques could be improved to cover more potential performance issues. In the case that the input image often contains large areas that do not need spatial filtering, a corresponding depth-based check should be considered as a part

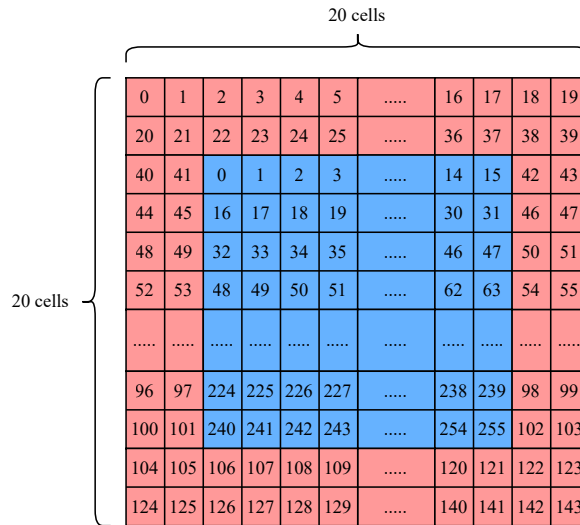


**Figure 4.** Shared memory block fetching mechanism for spatial iterations. The number in a cell indicates the local index of the thread in the workgroup, which is supposed to sample the cell's pixel data. Blue cells are sampled first; red are sampled afterward.

of fully inactive workgroup checks. To implement it efficiently for the variance moments' filtering stage, we would suggest setting the history length to the maximum value for the target pixels in the reprojection stage. In this case, variance filtering would be skipped automatically based on the existing history length check, and no additional memory loads would be needed (including the reprojection stage, since it loads the pixel's depth anyway). For spatial filtering passes, a fully inactive workgroup check similar to the one described above should be implemented; by using the abovementioned shared memory loading scheme, depth values fetched for the check could be used later to populate the shared memory block. Such modifications would add performance overhead for spatial filtering, but would provide a significant performance boost for workgroups that don't contain pixels that should be spatially filtered. We have not implemented such modifications since we didn't have non-filtered pixels in our testing scenarios, thus we haven't encountered corresponding performance issues.

It is also possible to extend the described technique to optimize the execution of workgroups with a low number of active threads. For this, we can calculate the number of active threads in a workgroup using atomic additions on a shared integer counter. Then, if this number is lower than some threshold for a given workgroup, we can avoid shared storage population and fall back to using global memory loads during the following calculations; as a threshold, the abovementioned thread number estimations or some profiling-optimized values can be used. Although this technique provides finer data fetching optimization, it is more expensive to perform; also, fallback support would complicate shader code, which may result in a performance penalty as well. We found it to be impractical in our testing environment; however, it might be beneficial for some rendering scenarios.

### 3.4. Reordering Group-Bound Execution

Convolution kernels essentially blend a given pixel's data with the data of pixels that have a specific offset from the given one. When such an offset leads to the offscreen area, blending is not done as no valid data is present. This is a common case for kernels with high dilation and thus for latter SVGF spatial iterations. The greater the dilation, the more distant the pixels from the edges of the screen sample's offscreen area; due to this, in the initial implementation the latter spatial iterations are faster than the earlier ones.

This phenomenon is expressed differently in the reordered SVGF implementation. Since optimal execution reordering is used for every spatial iteration, the modified execution dilation always equals one; however, pixels are considered invalid not only when they are outside of the screen, but also when they are outside of the reordering group to which the kernel's target pixel belongs. Since this does not affect shared memory storage population in any way, reordered implementation does not benefit

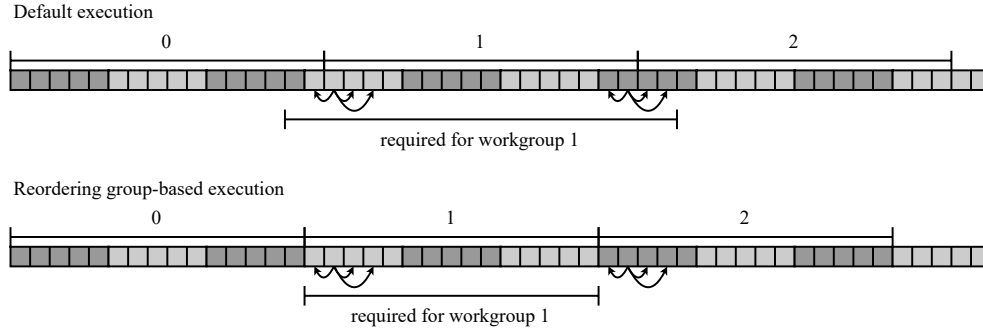


**Figure 5.** Two workgroup allocation methods for reordered pixel array processing. The reordering groups are marked by shades of gray. The ranges above the arrays indicate the workgroup size, the ranges under show the region of pixels required for spatial convolutions, and the arrows demonstrate the convolution footprint. Since convolution does not sample over the reordering group border, reordering group-based execution does not require a two-pixel “frame” to be loaded.

from high convolution dilations in the way initial implementation does. This is not a performance issue by itself; however, it means that the effective reuse of the sampled pixel data (i.e., the average number of convolutions the sampled pixel data is used by) decreases from the first to the last spatial iteration.

To alleviate this drawback, we modify the mapping between dispatch threads and screen pixels in spatial filtering passes with small reordering groups such that each reordering group is always processed in the same workgroup (Figure 5); we call it reordering group-based execution. In this scenario, a workgroup does not need a “frame” of pixel data to perform convolutions, since it belongs to different reordering groups and thus is considered offscreen. So, we only load data of the pixels processed by a workgroup, reducing the shared memory storage size and VRAM throughput required for the spatial filtering. Iterations with reordering group-bound execution enabled require 1 pixel data sample per working thread instead of 1.5625 samples as in the usual scenario, which means all pixel data is loaded only once per iteration.

The method has its drawbacks. First, it is only applicable when the workgroup can fit at least one full reordering group; hence, the possible reordering group size is determined by the workgroup dimension hardware limits (which is typically 1024 or  $32 \times 32$  threads on current hardware). This limits the reordering group-based execution application to iterations with very large dilation. Given the  $1920 \times 1080$  screen resolution, the reordering group maximum size for the sixth iteration equals  $\lceil (1920 \times 1080)/2^5 \rceil = (60 \times 34)$  and for the seventh is  $\lceil (1920 \times 1080)/2^6 \rceil = (30 \times 17)$ ; hence, it is only possible to use the method for the seventh iteration and onward. Second, the reordering group size can vary by one element (see reordering

definition above), and it might be impossible then to pack reordering groups in workgroups tightly. Considering the seventh spatial iteration on FullHD resolution, while there are 64 reordering groups with a size of 30 horizontally, vertically the screen has 56 groups with a size of 17 and 8 groups with a size of 16. Hence, in this case, it is required to set the workgroup dimensions to  $30 \times 17$  to ensure that every reordering group can be processed; only a  $30 \times 16$  thread block will be used, though, when processing reordering groups of reduced size. Unused dispatch threads decrease execution efficiency and should be minimized to achieve optimal performance.

In order to determine optimal workgroup dimensions, we perform a brute-force search for every screen dimension separately among the possible options (we found (12, 32) to be the optimal dimension range). For each option, we calculate the corresponding dispatch dimension in threads required to process all the reordering groups and choose the option providing the minimal dimension. This does not pose any performance problems since their number is limited and the search is only done at the initialization of the filter. The implementations of optimal workgroup dimension calculations and workgroup thread-to-pixel mapping are given in Listings 2 and 3.

Unfortunately, our reordering group-based execution implementation does not affect performance similarly on different hardware configurations. In particular, we found that it increases NVIDIA configuration performance, but decreases AMD configuration performance. The reason for such a difference is unclear. Due to that, we only enable this technique on the NVIDIA target; we assume profiling can be performed on the target configuration to determine whether using it is beneficial.

#### 4. Testing Results and Discussion

Table 4 shows the execution time and performance gain measurements of the optimized SVGF implementation. Tables 5 and 6 provide detailed measurements of the different filter stages. As can be observed, the optimized SVGF implementation provides a severe speed-up for the spatial filtering and variance moments' filtering stages. The temporal filtering stage suffers a slight performance degradation due to the feedback data having a non-identity reordering, which negatively affects the feedback data sampling pattern. Overall,  $1.6\text{--}2.8\times$  spatial filtering speed-up and  $1.3\text{--}2.5\times$  SVGF total execution speed-up are achieved on the tested hardware targets and filter configurations.

As was mentioned above, optimized implementation requires more VRAM as a result of additional storage images being allocated to perform depth/normal data reordering. It should be noted, though, that these are temporary resources only needed during SVGF execution; hence, the occupied memory can be reused by other render techniques of the engine, which might reduce or even eliminate the practical memory overhead of the optimized implementation. Table 7 shows the memory consumption

```
uint2 screen_dimensions;
uint iteration_index; // 0-based

// Reordering group count
uint rg_count = 1 << iteration_index;
// Number of bigger reordering groups
uint2 big_rgs = screen_dimensions % rg_count;
// Size of the smaller reordering groups
uint2 rg_size = screen_dimensions / rg_count;
uint2 big_rg_size = rg_size + (big_rgs[dim] ? 1 : 0);

uint2 min_thread_count = uint2(uint::max());
// Optimal workgroup size
uint2 optimal_wg_size;
uint2 optimal_bwgs;

for (uint dim = 0; dim < 2; ++dim)
{
    uint min_wg_size = max(big_rg_size[dim], 12);
    for (uint wg_size = min_wg_size; wg_size <= 32; ++wg_size)
    {
        // Number of smaller reordering groups that fit
        // in the workgroup of given size
        uint rgs_per_wg = wg_size / rg_size[dim];
        // Number of bigger reordering groups that fit
        // in the workgroup of given size
        uint brgs_per_wg = wg_size / big_rg_size[dim];
        // How many workgroups will process
        // at least one big reordering group
        uint bwgs = (big_rgs[dim] + brgs_per_wg - 1) / brgs_per_wg;
        // How many small reordering groups are processed in
        uint small_rgs = rg_count - bwgs * brgs_per_wg;
        // How many workgroups will only process
        // smaller reordering groups
        uint swgs = (small_rgs + rgs_per_wg - 1) / rgs_per_wg;
        // Total number of threads needed
        uint threads = (bwgs + swgs) * wg_size;

        if (threads < min_thread_count[dim])
        {
            min_thread_count[dim] = threads;
            optimal_wg_size[dim] = wg_size;
            optimal_bwgs[dim] = bwgs;
        }
    }
}
```

**Listing 2.** Reordering group-based execution optimal workgroup determination.

```
// DIVISOR is the execution reordering divisor
uint2 gScreenSize;
// Block ID and local thread ID
uint3 bid, lid;
// Defined in the preprocessor, based on values from Listing 2
uint4 G_PER_WG = uint4(
    optimal_wg_size / big_rg_size, // xy
    optimal_wg_size / rg_size      // zw
);
uint2 BWGS = optimal_bwgs;

uint2 size_overflow = gScreenSize % DIVISOR;
uint2 group_size = gScreenSize / DIVISOR;
uint2 groups_per_wg = ( bid.xy < BWGS ) ? G_PER_WG.xy : G_PER_WG.zw;
uint2 min_bid_offset = min( bid.xy, BWGS );
uint2 group_index_offset = bid.xy * G_PER_WG.zw +
    min_bid_offset * ( G_PER_WG.xy - G_PER_WG.zw );
uint2 pos_start = group_index_offset * group_size +
    min( group_index_offset, size_overflow );
uint2 pos_end = ( group_index_offset + groups_per_wg ) * group_size +
    min( group_index_offset + groups_per_wg, size_overflow );
uint2 reordered_pos = pos_start + lid.xy;
if ( any( reordered_pos >= min( pos_end, gScreenSize ) ) )
    return; // Out of bounds: threads are wasted
```

**Listing 3.** Reordering group-based execution thread-to-pixel screen mapping.

| Iterations | AMD, Static |       | AMD, Dynamic |       | NVIDIA, Static |       | NVIDIA, Dynamic |       |
|------------|-------------|-------|--------------|-------|----------------|-------|-----------------|-------|
|            | Time        | Gain  | Time         | Gain  | Time           | Gain  | Time            | Gain  |
| 1          | 1.05        | 1.59× | 1.35         | 2.47× | 1.39           | 1.4×  | 2.15            | 1.51× |
| 2          | 1.48        | 1.53× | 1.78         | 2.21× | 1.98           | 1.35× | 2.74            | 1.45× |
| 3          | 1.78        | 1.57× | 2.08         | 2.14× | 2.58           | 1.33× | 3.34            | 1.42× |
| 4          | 2.05        | 1.6×  | 2.35         | 2.11× | 3.14           | 1.34× | 3.90            | 1.41× |
| 5          | 2.36        | 1.6×  | 2.66         | 2.04× | 3.83           | 1.36× | 4.59            | 1.41× |
| 6          | 2.61        | 1.65× | 2.91         | 2.05× | 4.60           | 1.34× | 5.36            | 1.4×  |
| 7          | 2.89        | 1.69× | 3.19         | 2.06× | 4.96           | 1.44× | 5.72            | 1.48× |
| 8          | 3.14        | 1.73× | 3.44         | 2.06× | 5.23           | 1.54× | 5.99            | 1.56× |
| 9          | 3.4         | 1.73× | 3.70         | 2.04× | 5.45           | 1.61× | 6.21            | 1.63× |
| 10         | 3.59        | 1.73× | 3.89         | 2.02× | 5.58           | 1.66× | 6.34            | 1.67× |

**Table 4.** Optimized SVGF implementation execution times in milliseconds and performance gain measurements. Performance gain is calculated as the proportion between the original implementation's and the optimized implementation's corresponding execution time.

| GPU, Iterations | 1    | 2    | 3    | 4    | 5    | 6    | 7    | 8    | 9    | 10   |
|-----------------|------|------|------|------|------|------|------|------|------|------|
| AMD, 1          | 0.26 |      |      |      |      |      |      |      |      |      |
| AMD, 2          | 0.24 | 0.27 |      |      |      |      |      |      |      |      |
| AMD, 3          | 0.25 | 0.30 | 0.25 |      |      |      |      |      |      |      |
| AMD, 4          | 0.25 | 0.30 | 0.26 | 0.26 |      |      |      |      |      |      |
| AMD, 5          | 0.25 | 0.30 | 0.28 | 0.29 | 0.25 |      |      |      |      |      |
| AMD, 6          | 0.25 | 0.30 | 0.28 | 0.29 | 0.25 | 0.25 |      |      |      |      |
| AMD, 7          | 0.25 | 0.30 | 0.28 | 0.29 | 0.26 | 0.28 | 0.24 |      |      |      |
| AMD, 8          | 0.25 | 0.30 | 0.28 | 0.29 | 0.26 | 0.28 | 0.25 | 0.24 |      |      |
| AMD, 9          | 0.25 | 0.30 | 0.28 | 0.29 | 0.26 | 0.28 | 0.26 | 0.27 | 0.23 |      |
| AMD, 10         | 0.25 | 0.30 | 0.28 | 0.29 | 0.26 | 0.28 | 0.26 | 0.27 | 0.23 | 0.36 |
| NVIDIA, 1       | 0.40 |      |      |      |      |      |      |      |      |      |
| NVIDIA, 2       | 0.42 | 0.37 |      |      |      |      |      |      |      |      |
| NVIDIA, 3       | 0.43 | 0.41 | 0.52 |      |      |      |      |      |      |      |
| NVIDIA, 4       | 0.43 | 0.41 | 0.61 | 0.46 |      |      |      |      |      |      |
| NVIDIA, 5       | 0.43 | 0.40 | 0.72 | 0.52 | 0.44 |      |      |      |      |      |
| NVIDIA, 6       | 0.43 | 0.41 | 0.73 | 0.51 | 0.60 | 0.37 |      |      |      |      |
| NVIDIA, 7       | 0.43 | 0.40 | 0.73 | 0.52 | 0.63 | 0.41 | 0.37 |      |      |      |
| NVIDIA, 8       | 0.43 | 0.41 | 0.73 | 0.52 | 0.63 | 0.41 | 0.46 | 0.33 |      |      |
| NVIDIA, 9       | 0.43 | 0.41 | 0.72 | 0.52 | 0.63 | 0.42 | 0.49 | 0.33 | 0.28 |      |
| NVIDIA, 10      | 0.43 | 0.40 | 0.73 | 0.51 | 0.63 | 0.41 | 0.50 | 0.32 | 0.32 | 0.33 |

**Table 5.** Optimized SVGF implementation’s spatial iteration execution times in FullHD resolution. The timings are scene-agnostic but depend on the spatial iteration count.

comparison between implementations.

Since the presented optimizations avoid filter algorithm modifications or data precision degradation, the original and optimized implementations’ outputs are equivalent provided the same inputs and history data. Hence, these techniques can be used as a filter performance enhancement with no negative impact aside from increased implementation complexity. At the same time, they can be combined with the previously mentioned common optimizations that do not preserve the filtered result, including data storage quantization, edge-stopping functions modification, etc. Reducing the precision of the pixel data required for the variance moments’ filtering and spatial filtering stages would have severe performance benefits in particular, since that would reduce both the required video memory throughput and the required shared memory size, thus improving shader occupancy and execution performance.

Although SVGF was used to demonstrate optimizations described in this work, their application is not limited to it. One can adopt them to improve the implementation of any denoising filters or other techniques that involve performing a series of image convolutions with dilation being doubled after every iteration. Hence, the proposed optimizations can be applied to other filters that use the à-trous wavelet transform [Schied et al. 2018; Hirvonen et al. 2019]. They might also be useful for

| GPU, Iterations | DNP  | R    | VFS  | VFD  | SF   | FM   | $\Delta$ SF |
|-----------------|------|------|------|------|------|------|-------------|
| AMD, 1          | 0.12 | 0.55 | 0.12 | 0.42 | 0.26 |      | 2.81×       |
| AMD, 2          | 0.12 | 0.56 | 0.12 | 0.42 | 0.51 | 0.17 | 2.61×       |
| AMD, 3          | 0.12 | 0.57 | 0.12 | 0.42 | 0.80 | 0.17 | 2.33×       |
| AMD, 4          | 0.12 | 0.57 | 0.12 | 0.42 | 1.07 | 0.17 | 2.20×       |
| AMD, 5          | 0.12 | 0.57 | 0.12 | 0.42 | 1.37 | 0.18 | 2.07×       |
| AMD, 6          | 0.12 | 0.57 | 0.12 | 0.42 | 1.62 | 0.18 | 2.08×       |
| AMD, 7          | 0.12 | 0.57 | 0.12 | 0.42 | 1.90 | 0.18 | 2.08×       |
| AMD, 8          | 0.12 | 0.57 | 0.12 | 0.42 | 2.15 | 0.18 | 2.09×       |
| AMD, 9          | 0.12 | 0.57 | 0.12 | 0.42 | 2.42 | 0.18 | 2.04×       |
| AMD, 10         | 0.12 | 0.57 | 0.12 | 0.42 | 2.78 |      | 1.90×       |
| NVIDIA, 1       | 0.17 | 0.65 | 0.17 | 0.93 | 0.40 |      | 1.85×       |
| NVIDIA, 2       | 0.17 | 0.64 | 0.17 | 0.93 | 0.79 | 0.21 | 1.87×       |
| NVIDIA, 3       | 0.17 | 0.67 | 0.17 | 0.93 | 1.36 | 0.21 | 1.65×       |
| NVIDIA, 4       | 0.17 | 0.67 | 0.17 | 0.93 | 1.91 | 0.22 | 1.58×       |
| NVIDIA, 5       | 0.17 | 0.67 | 0.17 | 0.93 | 2.51 | 0.31 | 1.63×       |
| NVIDIA, 6       | 0.17 | 0.66 | 0.17 | 0.93 | 3.05 | 0.55 | 1.70×       |
| NVIDIA, 7       | 0.17 | 0.67 | 0.17 | 0.93 | 3.49 | 0.46 | 1.74×       |
| NVIDIA, 8       | 0.17 | 0.66 | 0.17 | 0.93 | 3.92 | 0.31 | 1.80×       |
| NVIDIA, 9       | 0.17 | 0.66 | 0.17 | 0.93 | 4.23 | 0.22 | 1.80×       |
| NVIDIA, 10      | 0.17 | 0.66 | 0.17 | 0.93 | 4.58 |      | 1.76×       |

**Table 6.** Optimized SVGF implementation execution time measurements in milliseconds. DNP indicates depth/normal packing stage; R is the reprojection stage; VFS and VFD are variance moments' filtering in static and dynamic scenarios, respectively; SF is spatial filtering; FM is the final modulation stage; and  $\Delta$ SF is the execution time gain for spatial filtering. Final modulation is absent for one- and ten-iteration spatial filtering.

| Implementation                     | Original | Optimized |
|------------------------------------|----------|-----------|
| Persistent memory per pixel, bytes | 42       | 42        |
| Temporary memory per pixel, bytes  | 58       | 90        |
| Total memory per pixel, bytes      | 100      | 132       |
| Persistent memory in FullHD, MB    | 83.06    | 83.06     |
| Temporary memory in FullHD, MB     | 114.7    | 178       |
| Total memory in FullHD, MB         | 197.75   | 261.04    |

**Table 7.** SVGF implementation memory consumption comparison.

neural-based techniques using similar data access patterns.

## 5. Conclusions

In this paper, we have presented optimization techniques for spatial wavelet-based filtering and applied them to spatiotemporal variance-guided filtering. Providing notable performance gains with no quality degradation, these optimizations widen the possibilities of path tracing usage in real-time computer graphics. Being suited for dif-

ferent series of convolutions with doubling dilation, the work is not limited to SVGF and has various applications.

## References

- AMD. AMD FidelityFX™ Denoiser: Shadows and reflections. *AMD GPUOpen*, <https://gpuopen.com/fidelityfx-denoiser/>, 2025. Accessed July 27, 2025. 21
- BAVOIL, L. Deinterleaved texturing for cache-efficient interleaved sampling. NVIDIA Corporation White Paper, 2014. URL: <https://d29g4g2dyqv443.cloudfront.net/sites/default/files/akamai/gameworks%2Fsamples%2FDeinterleavedTexturing.pdf>. 28
- BURNES, A. NVIDIA DLSS 3.5: Enhancing ray tracing with AI; Coming this Fall to Alan Wake 2, Cyberpunk 2077: Phantom Liberty, Portal with RTX & more. *NVIDIA GeForce*, <https://www.nvidia.com/en-us/geforce/news/nvidia-dlss-3-5-ray-reconstruction/>, August 22, 2023. 21
- HIRVONEN, A., SEPPÄLÄ, A., AIZENSHTEIN, M., AND SMAL, N. Accurate real-time specular reflections with radiance caching: High-quality and real-time rendering with DXR and other APIs. In HAINES, E. AND AKENINE-MÖLLER, T., editors, *Ray Tracing Gems*, pages 571–607. Apress, 2019. URL: [https://doi.org/10.1007/978-1-4842-4427-2\\_32](https://doi.org/10.1007/978-1-4842-4427-2_32). 21, 40
- KAJIYA, J. T. The rendering equation. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '86, pages 143–150. Association for Computing Machinery, 1986. URL: <https://doi.acm.org/10.1145/15922.15902>. 21
- KALLWEIT, S., CLARBERG, P., KOLB, C., DAVIDOVIČ, T., YAO, K.-H., FOLEY, T., HE, Y., WU, L., CHEN, L., AKENINE-MÖLLER, T., WYMAN, C., CRASSIN, C., AND BENTY, N. The Falcor rendering framework. *GitHub*, <https://github.com/NVIDIAGameWorks/Falcor>, August 2022. 22
- KIM, S., KAŹMIERCZYK, P., USS, W., KALIŃSKI, W., GALAJ, T., MACIEJEWSKI, M., TYAGI, K., SZABO, K., AND HARIHARA, R. Neural supersampling and denoising for real-time path tracing. *AMD GPUOpen*, [https://gpuopen.com/learn/neural\\_supersampling\\_and\\_denoising\\_for\\_real-time\\_path\\_tracing/](https://gpuopen.com/learn/neural_supersampling_and_denoising_for_real-time_path_tracing/), October 28, 2024. Last updated June 26, 2025. 21
- MCGUIRE, M., MARA, M., AND LUEBKE, D. Scalable ambient obscurance. In *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics*, EGGH-HPG'12, page 97–103. Eurographics Association, 2012. 28
- NVIDIA. NVIDIA RTX Kit. *NVIDIA Developer*, <https://developer.nvidia.com/rtx/ray-tracing/rt-denoisers>, 2025. Accessed July 27, 2025. 21
- OBERBERGER, M., CHAJDAS, M. G., AND WESTERMANN, R. Spatiotemporal variance-guided filtering for motion blur. *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, 5(3):22:1–22:13, July 2022. URL: <https://doi.org/10.1145/3543871>. 21

- PODLOZHNYUK, V. Image convolution with CUDA. NVIDIA Corporation White Paper 2097, July 2007. URL: [https://developer.download.nvidia.com/compute/DevZone/C/html\\_x64/3\\_Imaging/convolutionSeparable/doc/convolutionSeparable.pdf](https://developer.download.nvidia.com/compute/DevZone/C/html_x64/3_Imaging/convolutionSeparable/doc/convolutionSeparable.pdf). 27
- SCHIED, C., KAPLANYAN, A., WYMAN, C., PATNEY, A., CHAITANYA, C. R. A., BURGESS, J., LIU, S., DACHSBACHER, C., LEFOHN, A., AND SALVI, M. Spatiotemporal variance-guided filtering: Real-time reconstruction for path-traced global illumination. In *Proceedings of High Performance Graphics*, HPG '17, pages 2:1–2:12. Association for Computing Machinery, 2017. URL: <https://doi.org/10.1145/3105762.3105770>. 21, 22, 24
- SCHIED, C., PETERS, C., AND DACHSBACHER, C. Gradient estimation for real-time adaptive temporal filtering. *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, 1(2):24:1–24:16, 2018. URL: <https://doi.org/10.1145/3233301>. 21, 40

## Author Contact Information

Rostyslav Pikulsky  
[rostyslav.pikulsky@gmail.com](mailto:rostyslav.pikulsky@gmail.com)

---

Rostyslav Pikulsky, Optimizing Spatiotemporal Variance-Guided Filtering for Modern GPU Architectures, *Journal of Computer Graphics Techniques (JCGT)*, vol. 15, no. 1, 20–43, 2026  
<https://jcgt.org/published/0015/01/02/>

Received: 2025-04-09

Recommended: 2025-06-05

Published: 2026-01-12

Corresponding Editor: Angelo Pesce

Editor-in-Chief: Alexander Wilkie

© 2026 Rostyslav Pikulsky (the Authors).

The Authors provide this document (the Work) under the Creative Commons CC BY-ND 4.0 license available online at <http://creativecommons.org/licenses/by-nd/4.0/>. The Authors further grant permission for reuse of images and text from the first page of the Work, provided that the reuse is for the purpose of promoting and/or summarizing the Work in scholarly venues and that any reuse is accompanied by a scientific citation to the Work.

