# Bilateral Filtering with CUDA

Lasse Kløjgaard Staal, 20072300*

**Figure 1. An example of the bilateral filter, the input image to the left is noisy, whereas the output is smoothed but edges is preserved**

***Abstract*—This paper implements the Bilateral filter, using CUDA enhanced parallel computations. The Bilateral filter allows smoothing images, while preserving edges, in contrast to e.g. the Gaussian filter, which smoothes across edges. While delivering visually stunning results, Bilateral filtering is a costly operation.**

**Using NVidia's CUDA technology the filter can be parallelized to run on the GPU, which allows for fast execution, even for high definition images.**

***Index Terms*— Bilateral Filtering, CUDA, Parallel, Gaussian blur**

## I. Introduction

Filtering images often is a highly parallelizable processes where each pixel in the image is affected by a given filter. Filtering each pixel is an operation which is independent of the application of the filter to other neighbor pixels. Since an image consists of, often, millions pixels this makes a good candidate for parallelization.

The Gaussian Filter, is a smoothing filter. It can be applied to noisy images to smooth out impurities, but at the cost of less distinct edges. The Bilateral Filter [2] smoothes surfaces, just as the Gaussian Filter, while maintaining sharp edges in the image. Although Bilateral Filtering delivers impressive results, see **figure 1**, it does so at the cost of speed.

This paper implements the Bilateral Filter using NVidia's CUDA technology [1], obtaining a much lower runtime than for the corresponding standard implementation. Several improvements are made to the initial parallel implementation, resulting in an even larger speedup of applying the filter to an image.

*University of Aarhus, e-mail: staal@cs.au.dk

This paper has been developed as the mandatory project for the course "Data-parallel computing" 2011/2012.

## II. Contributions and outline of paper

### A. Contributions and related work

While there exist other implementations that are faster [3], they are often close approximations of the Bilateral Filter. For areas which requires the perfect filter, this paper describes a method for doing this using a massively parallel implementation.

A comparison between a naïve sequential implementation of the Bilateral Filter and different parallel implementations is also given, suggesting various methods for optimizing spatial filtering using CUDA.

### B. Outline

Section *III* describes the theory behind filtering, necessary for understanding the Bilateral Filter. First the Gaussian blur filter is outlined, and the edge smoothing problems with it described. Based on this, the Bilateral Filter is outlined which improves on the Gaussian blur filter. Section *IV* describes the various methods used for implementing the Bilateral Filter using CUDA, and the results are shown in section *V*, and discussed in section *VI*. Section *VII* highlights potential future work and section *VIII* concludes.

## III. Filtering

Filtering an image by a linear filter, is also referred to as convolution. Each output pixel in the process is the product between a local neighborhood of pixels and corresponding weights. Gaussian Blurring is a filter that does exactly that, for a pixel **p** in an image *I* the resulting value is:

$$GF(I)_{\mathrm{p}} = \sum_{q \in N(\mathrm{p})} G_{\sigma_s}(p,q) I_q \tag{1}$$

where the one and two dimensional Gaussian functions is defined as [4]:

$$G_\sigma(x,y) = \frac{1}{2\pi\sigma^2}\exp\left(-\frac{x^2+y^2}{2\sigma^2}\right) \tag{2}$$

$$G_\sigma(x) = \frac{1}{\sqrt{2\pi\sigma^2}}\exp\left(-\frac{x^2}{2\sigma^2}\right) \tag{3}$$

The Gaussian function will yield the highest value at 0 and fall off in a manner described by the standard deviation σ. A high σ means a smoother fall off. Although the Gaussian function never reaches a value of 0, it quickly converges towards it. Because the Gaussian tends towards 0 quickly, the Gaussian Blur filter can utilize this by only considering a region (neighborhood) around the current position.

The Gaussian function also is separable, which means that the value of $G_\sigma(x,y)$ is equal to $G_\sigma(x)G_\sigma(y)$.

The Gaussian filter does not take into account that pixels which are in close proximity, spatial wise, can be far away from each other, color wise. This means that the filter even will blur across edges, where we have a large intensity difference. The Bilateral Filter seeks to remedy this weakness of the Gaussian Filter by taking into account the intensities of the pixels. The filter is defined in [2] as:

$$\begin{aligned} BF(I)_{\mathrm{p}} &= \frac{1}{W_{\mathrm{p}}} \sum_{\mathrm{q} \in N(\mathrm{p})} G_{\sigma_s}(\mathrm{p},\mathrm{q}) G_{\sigma_r}\left(\left|I_{\mathrm{p}} - I_{\mathrm{q}}\right|\right) I_{\mathrm{q}} \\ W_{\mathrm{p}} &= \sum_{\mathrm{q} \in N(\mathrm{p})} G_{\sigma_s}(\mathrm{p},\mathrm{q}) G_{\sigma_r}\left(\left|I_{\mathrm{p}} - I_{\mathrm{q}}\right|\right) \end{aligned} \tag{4}$$

The term in front of the sum is a normalization term, which makes sure that the overall intensity of the image remains the same. The added Gaussian of the intensity difference between the neighbor **q** and the pixel at the position we're currently calculating **p**, helps preserving the edges. Using a Gaussian with a low standard deviation we can ensure pixels with a high intensity difference compared to the center pixel, is not taken into account by the smoothing. The larger $\sigma_r$ becomes, the more the filter go towards a regular Gaussian Blur.

There are various ways on how to calculate the intensity difference. For grayscale images it's simply the difference in intensities between the pixel at positions **p** and **q**, but it's not as straightforward for RGB-colored images. In [3] they suggest using a 3 dimensional Gaussian on the difference of all the color channels, but since the Gaussian is separable, this corresponds to taking the product between the Gaussian applied to each color channel individually. Using a 3D Gaussian on the colors in such a manner can produce minor artifacts, since the intensity difference of colors of one channel may be low, while another channel has a high intensity difference. To solve this, [5] suggests using the CIE-Lab color space, in which an Euclidian distance between two RGB elements makes sense.

## IV. Parallel Implementation

As a reference implementation, a naïve sequential version of the Bilateral Filter has been implemented, which runs on the CPU. It simply runs the Bilateral filter on each pixel on the image, using a kernel radius to define the size of the local neighborhood. The difference between pixel-intensities is calculated by using 3, one dimensional Gaussians on intensity difference on each of the 3 RGB color bands.

Since the naïve implementation runs through each pixel in the image, a simple initial transfer to using the GPU, is done by taking the code for the individual pixel and use that as a basis for a kernel. This kernel is then launched with parameters which generates a thread for each pixel in the image, the resulting filter is GPU_v1, see the appendix.

The values of the two dimensional Gaussian for the spatial difference can be pre computed, as it only depends on distances, and not the actual values of the pixels. This is done in GPU_v2, where the kernel gets an extra parameter, which is a map over the Gaussian function.

A lot of the time running the kernel, is spent calculating the Gaussian of color intensities. CUDA allows for fast execution of hardware based implementations for a set of functions, at the price of a slight imprecision. GPU_v3 utilizes this by using the hardware functions instead, the resulting image is indistinguishable from the reference implementation. GPU_v3 builds on the optimizations gained from earlier implementations, and as such also uses a pre computed Gaussian, this also applies to the rest of the implementations.

To increase the memory throughput, GPU_v4, uses a 1D texture to look up the pixels in the input image instead of using the global memory. Texture caching was chosen over shared memory, since the author was unsuccessful in finding a scheme for using the shared memory without causing too much thread divergence.

NVidia's visual profiler showed, that all the previous implementations was register bound. GPU_v5 is doing register optimizations, which leads to a higher occupancy in the streaming multiprocessors (SM's). This in turn, can help hide the memory latency, thus increasing the throughput. The implementation go from using 51 registers pr thread, to using 34. The thread configuration was configured to maximize the occupancy according to NVidia's occupancy calculator [6].

In GPU_v5 each thread does 3 texture lookups pr thread. Instead of looking up in the same texture, GPU_v6 tries to split up the 3 color channels into a texture each. GPU_v7 further extends this idea, by running a thread, not only on a per pixel basis, but on a per color basis, thus running 3 kernels to compute the filter. This also results in a much lower register saturation, allowing for a larger occupancy of each SM.

In GPU_v8, which is based on GPU_v5, the block and grid configuration is explored. For all the previous kernels, the blocks and grids has been 1 dimensional, GPU_v8 is modified to run on a two dimensional grid, in an attempt at hitting the texture cache more often.

Each of the above mentioned techniques can be looked up in the appendix, where the kernel code for each is supplied.

## V. Results

The machine used for the following tests has the following specifications:

*AMD Phenom II X4 940 3.00GHz*
*8 GB DDR2 ram, 1066 MHz*
*GeForce GTX 460, 1GB DDR5*
*NVidia driver version 285.38*

All GPU-test runs is an average of 10 launches. The timing excludes reading the image into memory, and storing it back onto the hard drive, but includes memory transfer to and from the device, and all kernel launches.

### A. Input size of the image

The first test was a comparison of the size of the input image, and speed. **Figure 2** shows a plot of the various implementations. GPU_v5 and GPU_v8 is almost equally fast, although GPU_v8 is a bit faster in the long run. The CPU and GPU_v1 and GPU_v2 has been left out for a more clear graph, but are supplied in the appendix.

### B. Filter radius

The second test focuses on the neighborhood size. The resulting plot is given in **figure 3**. CPU running times has again been left out, they can be found in the appendix. **Table 1** and **table 2** shows the improvement gain over the CPU implementation for all GPU versions. The speed up factor versus the CPU implementation is constant for varying input size, but increases for a larger neighborhood radius.

| | Time (s) | Factor |
|---|---|---|
| CPU | 50,226 | 1 |
| GPU1 | 0,499 | 101 |
| GPU2 | 0,434 | 116 |
| GPU3 | 0,259 | 194 |
| GPU4 | 0,188 | 268 |
| GPU5 | 0,050 | 1011 |
| GPU6 | 0,157 | 321 |
| GPU7 | 0,133 | 377 |
| GPU8 | 0,050 | 1002 |

**Table 1**. 2 Mpixel, radius: 4

| | Time (s) | Factor |
|---|---|---|
| CPU | 268,381 | 1 |
| GPU1 | 2,545 | 105 |
| GPU2 | 2,202 | 122 |
| GPU3 | 1,193 | 225 |
| GPU4 | 0,908 | 295 |
| GPU5 | 0,162 | 1653 |
| GPU6 | 0,254 | 1057 |
| GPU7 | 0,320 | 838 |
| GPU8 | 0,161 | 1665 |

**Table 2**. 2 Mpixel, radius: 10

### C. Kernel comparisons

**Figure 4** is from NVidia's "Compute visual profiler", used to run the test implementation. Note how GPU_v7 is called 3 times as described earlier. This sums to 4972,9 us, and is slower than implementation 5,6 and 7. The test run was with a 512x512 image, and a kernel size of 4.

## VI.

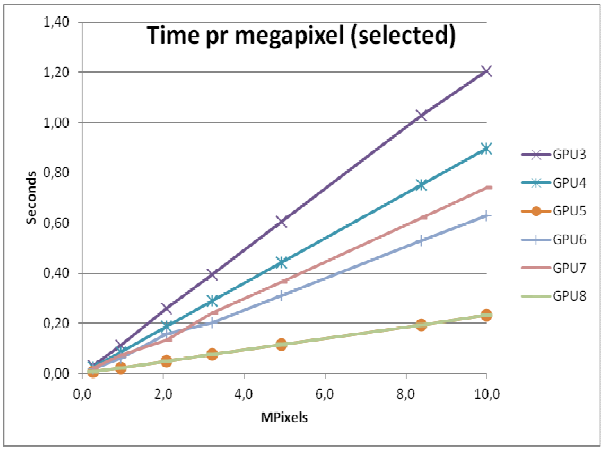


**Figure 2.** Runtime of the different implementations as a function of the input size of the image.

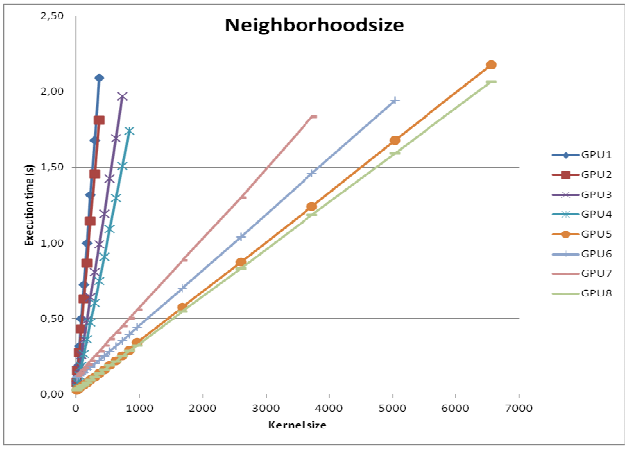


**Figure 3.** Runtime of the different implementations as a function of neighborhood size

Profiler Output | Summary Table

| | GPU Timestamp (us) | Method | GPU Time (us) | CPU Time (us) | grid size | **thread block size** | registers per thread | Occupancy |
|---|---|---|---|---|---|---|---|---|
| 1 | 1186,05 | bilateralFilterGPU_v1 | 59198,4 | 59206,9 | [512 1 1] | [512 1 1] | 45 | 0,333 |
| 2 | 90238,2 | bilateralFilterGPU_v2 | 50983,1 | 50991,6 | [512 1 1] | [512 1 1] | 51 | 0,333 |
| 3 | 171170 | bilateralFilterGPU_v3 | 27755,4 | 27763,9 | [512 1 1] | [512 1 1] | 51 | 0,333 |
| 4 | 228361 | bilateralFilterGPU_v4 | 21076,6 | 21091,6 | [512 1 1] | [512 1 1] | 51 | 0,333 |
| 5 | 355504 | bilateralFilterGPU_v6 | 3338,18 | 3356,57 | [1024 1 1] | [256 1 1] | 34 | 0,5 |
| 6 | 279364 | bilateralFilterGPU_v5 | 3291,68 | 3302,92 | [1365 1 1] | [192 1 1] | 34 | 0,625 |
| 7 | 312418 | bilateralFilterGPU_v8 | 3234,59 | 3245,49 | [32 42 1] | [16 12 1] | 34 | 0,625 |
| 8 | 397500 | bilateralFilterGPU_v7 | 1658,91 | 1676,62 | [1365 1 1] | [192 1 1] | 23 | 0,875 |
| 9 | 401986 | bilateralFilterGPU_v7 | 1657,63 | 1673,64 | [1365 1 1] | [192 1 1] | 23 | 0,875 |
| 10 | 399746 | bilateralFilterGPU_v7 | 1656,35 | 1673,04 | [1365 1 1] | [192 1 1] | 23 | 0,875 |

**Figure 4.** Runtime of the different implementations as a function of kernel (neighborhood size)

## DISCUSSION

From the results, it's clear to see that just a simple transfer from CPU to GPU, with no optimizations can yield a good result, with a speed increase of more than 100 times the CPU. Granted, the CPU implementation is unoptimized, but compared to [3] which suggests an optimized CPU implementation may compute at roughly 1 megapixel per second, it is still very good for a minimal investment.

Using a precalculated map over the Gaussian gave only a slight improvement, from 101 to a 116 times faster, according to table 1. This is most likely due to bad memory access to the global memory, in which the map resides. Each thread points to the same point at each iteration, leaving little room for coalescence. This could be improved by copying the map to shared memory, but since the map can be of varying size, potentially larger than a block, this turns out to be cumbersome.

Using the hardware functions (GPU_v3) yields a larger gain, from 116 to 192 times as fast, but with no noticeable degradation in image quality. Just as large is the improvement of using a 1D texture to access the data (GPU_v4), which can be attributed to multiple threads hitting the texture cache. The test machine was unable to run a profiling of cache hits, so this is left to speculations.

The biggest gain was from reducing the number of registers (GPU_v5). It turns out that the initial implementation of the Gaussian function used a lot of registers. By restructuring and by doing a few minor operations multiple times, a reduction from 51 to 34 registers was obtained. According to figure 4, this results in a occupancy increase from 33,3% to 62,5%, meaning the streaming multiprocessors has access to almost twice the amount of warps at the time. Increasing the number of warps in a SM is one way to increase the memory throughput, since while waiting for one memory request, another warp can be put to work. This yields almost a 4 times improvement over GPU_v4 and 1011 times compared to the reference implementation. Increasing the occupancy isn't always enough though. In GPU_v7, 3 times as many threads is launched, but with a lower register saturation. This yields a occupancy of each SM of 87,5% but the end result is slower. This could be due to each thread also doing 1/3 less computations. The warps thus do a bad job at hiding the memory latency, resulting in a lot of time where all warps in a single SM is waiting for memory requests.

Organizing the threads into blocks of 16 by 12 in GPU_v8 instead of a single linear block of size 192, only gives rise to a slight improvement in running times. Figure 3 suggests this difference increases for a larger neighborhood, due to a bit more hits in the texture cache.

Figure 3 shows that the implementations execution time depends linearly by the input size. Figure 4 shows a linearity between neighborhood size and execute times, but the neighborhood size depends on the quadratic of the input neighborhood radius.

The relative speed gain for various input image sizes is constant, for example, the speed gain by using GPU_v8 is 999,89 times the sequential implementation on average. For a larger input radius and hereby neighborhood size, the gain increases for larger input. A comparison between table 1 and 2 outlines this, for GPU_v8 there is an increase from 1002 times the reference, to 1665 times the speed due to the quadratic dependency on the input radius.

GPU_v8 takes 0.050s to do the computation, on a full HD image (2M pixels), which is a frequency of 20 images per second, which is close to real time updating. This is only for a small neighborhood, of size 81 (with a radius of 4), but even for large radii the implementation only takes a few seconds.

## VII. FUTURE WORK

Texture lookups at times can produce erroneous results in the implementations, shown as artifacts on the top and sometimes the left border. This is particularly bad after running GPU_v7, suggesting a bug in this particular implementation. Locating this bug would be something worthwhile, it could potentially increase the execution speed of GPU_v7.

The tests includes the time it takes for GPU_v6 and GPU_v7 to reorganize the input memory into 3 separate arrays

before sending it to the CUDA device, which is unfair to those implementations. The CUDA kernel for GPU_v7 run 3 times is still slower than implementations 5,6 and 8. However, an error discovered in the late minutes suggests that GPU_v6 may fare better with a different block configuration, yielding a higher occupancy. CUDA do not support fetching a float3 from a single texture, it does however support float4 texture maps. An interesting future test would be to see if wasting a byte per pixel, and using a float4 texture map results in a higher percentage of texture cache hits.

Another future improvement could be, to look closer at utilizing shared memory and constant memory. So far the author has been unable to create a suitable strategy that does not suffer too much from a high thread divergence.

## VIII. Conclusion

The Bilateral Filter has been implemented using CUDA, with a large speed gain. For a naïve transfer from sequential to parallel, this yields an improvement of just above 100 times in running time. Optimizing the code to run more efficiently on the GPU pays off, with the final result running at more than a thousand times faster than the sequential implementation. This allows 20 images pr second at full HD for small image manipulations.

## References


[1] http://www.nvidia.com/object/cuda_home_new.html.

[2] S. Paris, P. Kornprobst, J. Tumblin and F. Durand: "Bilateral Filtering: Theory and Applications" in Computer Graphics and Vision Vol 4, No. 1 (2008) 1 - 73.

[3] J. Chen, S. Paris and F. Durand: "Real-time Edge-Aware Image Processing with the Bilateral Grid". Available at: http://groups.csail.mit.edu/graphics/bilagrid/bilagrid_web.pdf

[4] Gaussian Blur formulas, available at Wikipedia: http://en.wikipedia.org/wiki/Gaussian_blur

[5] C. Tomasi and R. Manduchi: "Bilateral Filtering for gray and color images". Proceeding ICCV '98 Proceedings of the Sixth International Conference on Computer Vision.

[6] http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls

## APPENDIX

### A. *Running the Bilateral Filter implementation*

The implementation and this report can be found at:
http://www.cs.au.dk/~staal/dpc/

To run it, call Bilateral_filter.exe from the "Release" directory, with the following arguments:

```
<input file> <width> <height> <channels>
<neighborhood radius> <spatial sigma> <range
sigma> <iterations>
```

where the input file is in .raw format. The width, height, and channels describes the input image. Spatial sigma and range sigma are floats, with good values of for example 2.0 and 0.1. Iterations is how many times each of the 8 test programs must be run, during the test. The code was compiled using Visual Studio 2010, CUDA 4.0 and specifically for "compute_20 ,sm_20" compatible hardware. The CPU timing is commented out, it takes a long time to compute.

Alternatively the Release/runtest.bat can be run.

### B. *Figures*

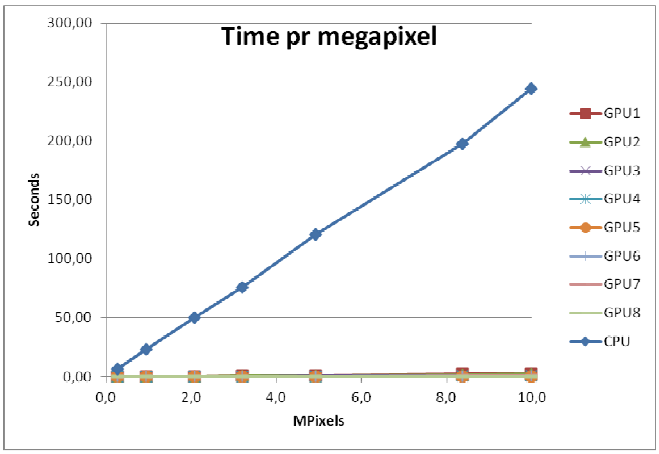


**Figure 2** including CPU time

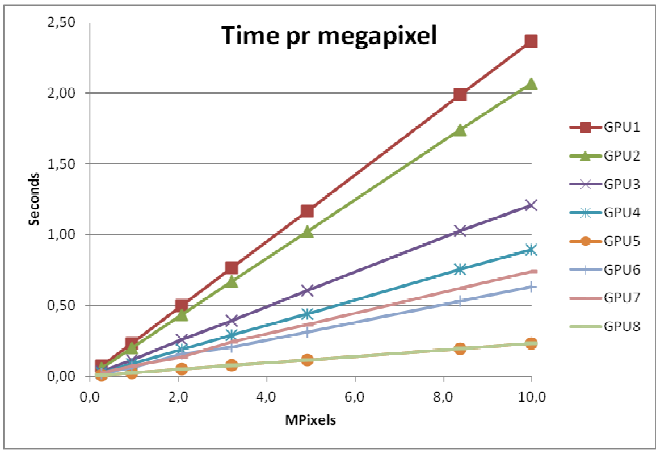


**Figure 2** excluding CPU time, including GPU1 and GPU2.

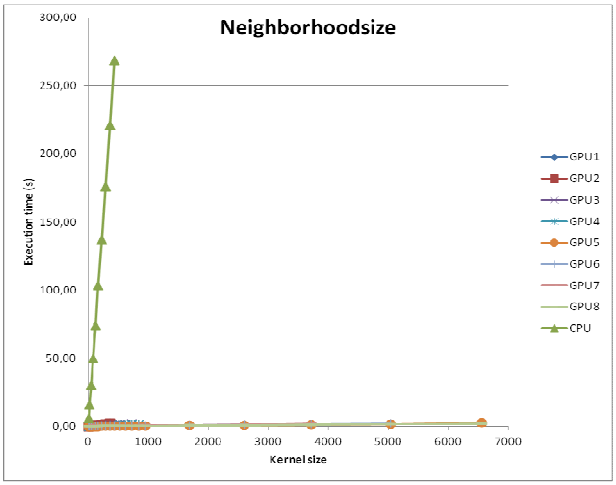


**Figure 3.** including CPU time, full scale.

### C. *Code*

No host function launching the corresponding kernel is supplied. All of them are trivial, copying the various resources to and from the GPU and launching the kernel.

#### 1) *Function to calculate the gaussian map/kernel*

This (host) function precalculates the a 2dimensional Gaussian:

```
float* generateGaussianKernel(int radius, float sigma)
{
   int area = (2*radius+1)*(2*radius+1);
   float* res = new float[area];

   for(int x = -radius; x <= radius; x++)
      for(int y = -radius; y <= radius; y++)
      {
         //Co_to_idx inspired
         int position = (x+radius)*(radius*2+1) + y+radius;
         res[position] = gaussian2d(x,y,sigma);
      }
   return res;
}
```

#### 2) *Gaussian device functions*

```
__host__ __device__
float gaussian1d(float x, float sigma)
{
   float variance = pow(sigma,2);
   float exponent = -pow(x,2)/(2*variance);
   return expf(exponent) / sqrt(2 * PI * variance);
}

inline __device__
float gaussian1d_gpu(float x, float sigma)
{
   float variance = __powf(sigma,2);
//this doesnt work for some reason: __powf(x,2.0f)
   float power = pow(x,2);
   float exponent = -power/(2*variance);
   return __expf(exponent) / sqrt(2 * PI * variance);
}

__host__ __device__
float gaussian2d(float x, float y, float sigma)
{
   float variance = pow(sigma,2);
   float exponent = -(pow(x,2) + pow(y,2))/(2*variance);
   return expf(exponent) / (2 * PI * variance);
}
```

The final implementation of the Gaussian, this one saves a bunch of registers:

```
inline __device__
float gaussian1d_gpu_reg(float x, float variance, float
sqrt_pi_variance)
{
   float gaussian1d = -(x*x)/(2*variance);
   gaussian1d = __expf(gaussian1d);
   gaussian1d /= sqrt_pi_variance;
   return gaussian1d;
}
```

*GPU_v1*

```
__global__
void bilateralFilterGPU_v1(float3* input, float3* output, uint2
dims, int radius, float sigma_spatial, float sigma_range)
{
   const unsigned int idx = blockIdx.x*blockDim.x + threadIdx.x;

   uint2 pos = idx_to_co(idx,dims);
   int img_x = pos.x;
   int img_y = pos.y;

   if(img_x >= dims.x || img_y >= dims.y) return;

   float3 currentColor = input[idx];

   float3 res = make_float3(0.0f,0.0f,0.0f);
   float3 normalization = make_float3(0.0f,0.0f,0.0f);;


   for(int i = -radius; i <= radius; i++) {
      for(int j = -radius; j <= radius; j++) {
         int x_sample = img_x+i;
         int y_sample = img_y+j;

         //mirror edges
         if( x_sample < 0) x_sample = -x_sample;
         if( y_sample < 0) y_sample = -y_sample;
         if( x_sample > dims.x - 1) x_sample = dims.x - 1 - i;
         if( y_sample > dims.y - 1) y_sample = dims.y - 1 - j;


         float3 tmpColor =
input[co_to_idx(make_uint2(x_sample,y_sample),dims)];

         float gauss_spatial = gaussian2d(i,j,sigma_spatial);

         float3 gauss_range;
         gauss_range.x = gaussian1d(currentColor.x - tmpColor.x,
sigma_range);
         gauss_range.y = gaussian1d(currentColor.y - tmpColor.y,
sigma_range);
         gauss_range.z = gaussian1d(currentColor.z - tmpColor.z,
sigma_range);

         float3 weight = gauss_spatial * gauss_range;
         normalization = normalization + weight;
         res = res + (tmpColor * weight);

      }
   }
   res.x /= normalization.x;
   res.y /= normalization.y;
   res.z /= normalization.z;
   output[idx] = res;
}
```

*GPU_v2*

The argument named "kernel" is the precalculated map. In image filtering, the map is often referred to as a kernel.

```
__global__
void bilateralFilterGPU_v2(float3* input, float3* output, uint2
dims, int radius, float* kernel, float sigma_range)
{
   const unsigned int idx = blockIdx.x*blockDim.x + threadIdx.x;

   uint2 pos = idx_to_co(idx,dims);
   int img_x = pos.x;
   int img_y = pos.y;

   if(img_x >= dims.x || img_y >= dims.y) return;

   float3 currentColor = input[idx];

   float3 res = make_float3(0.0f,0.0f,0.0f);
   float3 normalization = make_float3(0.0f,0.0f,0.0f);;

   for(int i = -radius; i <= radius; i++) {
      for(int j = -radius; j <= radius; j++) {

         int x_sample = img_x+i;
         int y_sample = img_y+j;

         //mirror edges
         if( x_sample < 0) x_sample = -x_sample;
         if( y_sample < 0) y_sample = -y_sample;
         if( x_sample > dims.x - 1) x_sample = dims.x - 1 - i;
         if( y_sample > dims.y - 1) y_sample = dims.y - 1 - j;


         float3 tmpColor =
input[co_to_idx(make_uint2(x_sample,y_sample),dims)];

         float gauss_spatial =
kernel[co_to_idx(make_uint2(i+radius,j+radius),make_uint2(radius*
2+1,radius*2+1))];

         float3 gauss_range;
         gauss_range.x = gaussian1d(currentColor.x - tmpColor.x,
sigma_range);
         gauss_range.y = gaussian1d(currentColor.y - tmpColor.y,
sigma_range);
         gauss_range.z = gaussian1d(currentColor.z - tmpColor.z,
sigma_range);

         float3 weight = gauss_spatial * gauss_range;
         normalization = normalization + weight;
         res = res + (tmpColor * weight);

      }
   }

   res.x /= normalization.x;
   res.y /= normalization.y;
   res.z /= normalization.z;
   output[idx] = res;
}
```

*GPU_v3*

The 3rd version no longer smoothes the result, due to an unknown bug. This bug was introduced somewhere down the line of optimizations and is yet to be located.

```
__global__
void bilateralFilterGPU_v3(float3* input, float3* output, uint2
dims, int radius, float* kernel, float sigma_range)
{
   const unsigned int idx = blockIdx.x*blockDim.x + threadIdx.x;

   uint2 pos = idx_to_co(idx,dims);
   int img_x = pos.x;
   int img_y = pos.y;

   if(img_x >= dims.x || img_y >= dims.y) return;

   float3 currentColor = input[idx];

   float3 res = make_float3(0.0f,0.0f,0.0f);
   float3 normalization = make_float3(0.0f,0.0f,0.0f);;

   for(int i = -radius; i <= radius; i++) {
      for(int j = -radius; j <= radius; j++) {

         int x_sample = img_x+i;
         int y_sample = img_y+j;

         //mirror edges
         if( x_sample < 0) x_sample = -x_sample;
         if( y_sample < 0) y_sample = -y_sample;
         if( x_sample > dims.x - 1) x_sample = dims.x - 1 - i;
         if( y_sample > dims.y - 1) y_sample = dims.y - 1 - j;

         float3 tmpColor =
input[co_to_idx(make_uint2(x_sample,y_sample),dims)];

         float gauss_spatial =
kernel[co_to_idx(make_uint2(i+radius,j+radius),make_uint2(radius*
2+1,radius*2+1))];

         float3 gauss_range;
         gauss_range.x = gaussian1d_gpu(currentColor.x -
tmpColor.x, sigma_range);
         gauss_range.y = gaussian1d_gpu(currentColor.y -
tmpColor.y, sigma_range);
         gauss_range.z = gaussian1d_gpu(currentColor.z -
tmpColor.z, sigma_range);

         float3 weight = gauss_spatial * gauss_range;
         normalization = normalization + weight;
         res = res + (tmpColor * weight);

      }
   }

   res.x /= normalization.x;
   res.y /= normalization.y;
   res.z /= normalization.z;
   output[idx] = res;
}
```

*GPU_v4*

The 4th version no longer smoothes the result, due to an unknown bug, just as the 3rd version.

```
__global__
void bilateralFilterGPU_v4(float3* output, uint2 dims, int
radius, float* kernel, float sigma_range)
{
   const unsigned int idx = blockIdx.x*blockDim.x + threadIdx.x;

   uint2 pos = idx_to_co(idx,dims);
   int img_x = pos.x;
   int img_y = pos.y;

   if(img_x >= dims.x || img_y >= dims.y) return;

   float3 currentColor = make_float3(tex1Dfetch(tex,
3*idx),tex1Dfetch(tex, 3*idx+1),tex1Dfetch(tex, 3*idx+2));

   float3 res = make_float3(0.0f,0.0f,0.0f);
   float3 normalization = make_float3(0.0f,0.0f,0.0f);

   for(int i = -radius; i <= radius; i++) {
      for(int j = -radius; j <= radius; j++) {

         int x_sample = img_x+i;
         int y_sample = img_y+j;

         //mirror edges
         if( x_sample < 0) x_sample = -x_sample;
         if( y_sample < 0) y_sample = -y_sample;
         if( x_sample > dims.x - 1) x_sample = dims.x - 1 - i;
         if( y_sample > dims.y - 1) y_sample = dims.y - 1 - j;

         int tempPos =
co_to_idx(make_uint2(x_sample,y_sample),dims);

         float3 tmpColor = make_float3(tex1Dfetch(tex,
3*tempPos),tex1Dfetch(tex, 3*tempPos+1),tex1Dfetch(tex,
3*tempPos+2));//input[tempPos];

         float gauss_spatial =
kernel[co_to_idx(make_uint2(i+radius,j+radius),make_uint2(radius*
2+1,radius*2+1))];

         float3 gauss_range;
         gauss_range.x = gaussian1d_gpu(currentColor.x -
tmpColor.x, sigma_range);
         gauss_range.y = gaussian1d_gpu(currentColor.y -
tmpColor.y, sigma_range);
         gauss_range.z = gaussian1d_gpu(currentColor.z -
tmpColor.z, sigma_range);

         float3 weight = gauss_spatial * gauss_range;

         normalization = normalization + weight;
         res = res + (tmpColor * weight);

      }
   }

   res.x /= normalization.x;
   res.y /= normalization.y;
   res.z /= normalization.z;
   output[idx] = res;
}
```

*GPU_v5*

```
__global__
void bilateralFilterGPU_v5(float3* output, uint2 dims, int
radius, float* kernel, float variance, float sqrt_pi_variance)
{
   const unsigned int idx = blockIdx.x*blockDim.x + threadIdx.x;

   uint2 pos = idx_to_co(idx,dims);

   if(pos.x >= dims.x || pos.y >= dims.y) return;

   float3 currentColor = make_float3(tex1Dfetch(tex,
3*idx),tex1Dfetch(tex, 3*idx+1),tex1Dfetch(tex, 3*idx+2));

   float3 res = make_float3(0.0f,0.0f,0.0f);
   float3 normalization = make_float3(0.0f,0.0f,0.0f);
   float3 weight;
   for(int i = -radius; i <= radius; i++) {
      for(int j = -radius; j <= radius; j++) {

         int x_sample = pos.x+i;
         int y_sample = pos.y+j;

         //mirror edges
         if( x_sample < 0) x_sample = -x_sample;
         if( y_sample < 0) y_sample = -y_sample;
         if( x_sample > dims.x - 1) x_sample = dims.x - 1 - i;
         if( y_sample > dims.y - 1) y_sample = dims.y - 1 - j;

         int tempPos =
co_to_idx(make_uint2(x_sample,y_sample),dims);

         float3 tmpColor = make_float3(tex1Dfetch(tex,
3*tempPos),tex1Dfetch(tex, 3*tempPos+1),tex1Dfetch(tex,
3*tempPos+2));//input[tempPos];

         float gauss_spatial =
kernel[co_to_idx(make_uint2(i+radius,j+radius),make_uint2(radius*
2+1,radius*2+1))];

         weight.x = gauss_spatial *
gaussian1d_gpu_reg((currentColor.x -
tmpColor.x),variance,sqrt_pi_variance);
         weight.y = gauss_spatial *
gaussian1d_gpu_reg((currentColor.y -
tmpColor.y),variance,sqrt_pi_variance);
         weight.z = gauss_spatial *
gaussian1d_gpu_reg((currentColor.z -
tmpColor.z),variance,sqrt_pi_variance);

         normalization = normalization + weight;
         res = res + (tmpColor * weight);

      }
   }

   res.x /= normalization.x;
   res.y /= normalization.y;
   res.z /= normalization.z;
   output[idx] = res;
}
```

*GPU_v6*

```
__global__
void bilateralFilterGPU_v6(float3* output, uint2 dims, int
radius, float* kernel, float variance, float sqrt_pi_variance)
{
   const unsigned int idx = blockIdx.x*blockDim.x + threadIdx.x;

   uint2 pos = idx_to_co(idx,dims);

   if(pos.x >= dims.x || pos.y >= dims.y) return;

   float3 currentColor = make_float3(tex1Dfetch(tex_red,
idx),tex1Dfetch(tex_green, idx),tex1Dfetch(tex_blue, idx));

   float3 res = make_float3(0.0f,0.0f,0.0f);
   float3 normalization = make_float3(0.0f,0.0f,0.0f);

   for(int i = -radius; i <= radius; i++) {
      for(int j = -radius; j <= radius; j++) {

         int x_sample = pos.x+i;
         int y_sample = pos.y+j;

         //mirror edges
         if( x_sample < 0) x_sample = -x_sample;
         if( y_sample < 0) y_sample = -y_sample;
         if( x_sample > dims.x - 1) x_sample = dims.x - 1 - i;
         if( y_sample > dims.y - 1) y_sample = dims.y - 1 - j;

         int tempPos =
co_to_idx(make_uint2(x_sample,y_sample),dims);

         float3 tmpColor = make_float3(tex1Dfetch(tex_red,
tempPos),tex1Dfetch(tex_green, tempPos),tex1Dfetch(tex_blue,
tempPos));

         float gauss_spatial =
kernel[co_to_idx(make_uint2(i+radius,j+radius),make_uint2(radius*
2+1,radius*2+1))];

         float3 gauss_range;
         gauss_range.x = gaussian1d_gpu_reg(currentColor.x -
tmpColor.x, variance, sqrt_pi_variance);
         gauss_range.y = gaussian1d_gpu_reg(currentColor.y -
tmpColor.y, variance, sqrt_pi_variance);
         gauss_range.z = gaussian1d_gpu_reg(currentColor.z -
tmpColor.z, variance, sqrt_pi_variance);

         float3 weight = gauss_spatial * gauss_range;

         normalization = normalization + weight;
         res = res + (tmpColor * weight);

      }
   }

   res.x /= normalization.x;
   res.y /= normalization.y;
   res.z /= normalization.z;
   output[idx] = res;
}
```

*GPU_v7*

```
__global__
void bilateralFilterGPU_v7(float* output, uint2 dims, int radius,
float* kernel, float variance, float sqrt_sigma)
{
   const unsigned int idx = blockIdx.x*blockDim.x + threadIdx.x;

   uint2 pos = idx_to_co(idx,dims);

   if(pos.x >= dims.x || pos.y >= dims.y) return;

   float currentColor = tex1Dfetch(tex, idx);

   float res = 0.0f;
   float normalization = 0.0f;

   for(int i = -radius; i <= radius; i++) {
      for(int j = -radius; j <= radius; j++) {

         int x_sample = pos.x+i;
         int y_sample = pos.y+j;

         //mirror edges
         if( x_sample < 0) x_sample = -x_sample;
         if( y_sample < 0) y_sample = -y_sample;
         if( x_sample > dims.x - 1) x_sample = dims.x - 1 - i;
         if( y_sample > dims.y - 1) y_sample = dims.y - 1 - j;

         float tmpColor = tex1Dfetch(tex, y_sample*dims.x +
x_sample);

         float gaussian1d = -((currentColor -
tmpColor)*(currentColor - tmpColor))/(2*variance);
         gaussian1d = __expf(gaussian1d);
         gaussian1d /= sqrt_sigma;

         float gauss_spatial =
kernel[co_to_idx(make_uint2(i+radius,j+radius),make_uint2(radius*
2+1,radius*2+1))];

         float weight = gauss_spatial * gaussian1d;

         normalization = normalization + weight;
         res = res + (tmpColor * weight);

      }
   }

   output[idx] = res / normalization;
}
```

*GPU_v8*

```
__global__
void bilateralFilterGPU_v8(float3* output, uint2 dims, int
radius, float* kernel, float variance, float sqrt_pi_variance)
{
   const unsigned int idx_x = blockIdx.x*blockDim.x + threadIdx.x;
   const unsigned int idx_y = blockIdx.y*blockDim.y + threadIdx.y;

   if(idx_x >= dims.x || idx_y >= dims.y) return;

   int idx = idx_y*dims.x + idx_x;
//co_to_idx(make_uint2(idx_x,idx_y),dims);
   float3 currentColor = make_float3(tex1Dfetch(tex,
3*idx),tex1Dfetch(tex, 3*idx+1),tex1Dfetch(tex, 3*idx+2));

   float3 res = make_float3(0.0f,0.0f,0.0f);
   float3 normalization = make_float3(0.0f,0.0f,0.0f);
   float3 weight;
   for(int i = -radius; i <= radius; i++) {
      for(int j = -radius; j <= radius; j++) {

         int x_sample = idx_x+i;
         int y_sample = idx_y+j;

         //mirror edges
         if( x_sample < 0) x_sample = -x_sample;
         if( y_sample < 0) y_sample = -y_sample;
         if( x_sample > dims.x - 1) x_sample = dims.x - 1 - i;
         if( y_sample > dims.y - 1) y_sample = dims.y - 1 - j;

         int tempPos =
co_to_idx(make_uint2(x_sample,y_sample),dims);

         float3 tmpColor = make_float3(tex1Dfetch(tex,
3*tempPos),tex1Dfetch(tex, 3*tempPos+1),tex1Dfetch(tex,
3*tempPos+2));//input[tempPos];

         float gauss_spatial =
kernel[co_to_idx(make_uint2(i+radius,j+radius),make_uint2(radius*
2+1,radius*2+1))];

         weight.x = gauss_spatial *
gaussian1d_gpu_reg((currentColor.x -
tmpColor.x),variance,sqrt_pi_variance);
         weight.y = gauss_spatial *
gaussian1d_gpu_reg((currentColor.y -
tmpColor.y),variance,sqrt_pi_variance);
         weight.z = gauss_spatial *
gaussian1d_gpu_reg((currentColor.z -
tmpColor.z),variance,sqrt_pi_variance);

         normalization = normalization + weight;
         res = res + (tmpColor * weight);

      }
   }
   res.x /= normalization.x;
   res.y /= normalization.y;
   res.z /= normalization.z;
   output[idx] = res;
}
```