# Exploration of Optimization Options for Increasing Performance of a GPU Implementation of a Three-Dimensional Bilateral Filter

E. Wes Bethel

Computer and Data Sciences Department,
Computational Research Division
Lawrence Berkeley National Laboratory,
Berkeley, California, USA, 94720.

January 2012

## Acknowledgment

This work was supported by the Director, Office of Science, Office and Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231 through the Scientific Discovery through Advanced Computing (SciDAC) program's Visualization and Analytics Center for Enabling Technologies (VACET).

## Legal Disclaimer

# Contents

**Abstract**

This report explores using GPUs as a platform for performing high performance medical image data processing, specifically smoothing using a 3D bilateral filter, which performs anisotropic, edge-preserving smoothing. The algorithm consists of a running a specialized 3D convolution kernel over a source volume to produce an output volume. Overall, our objective is to understand what algorithmic design choices and configuration options lead to optimal performance of this algorithm on the GPU. We explore the performance impact of using different memory access patterns, of using different types of device/on-chip memories, of using strictly aligned and unaligned memory, and of varying the size/shape of thread blocks. Our results reveal optimal configuration parameters for our algorithm when executed sample 3D medical data set, and show performance gains ranging from 30x to over 200x as compared to a single-threaded CPU implementation.

# 1 Related Work

## 1.1 Bilateral Filtering

Image smoothing, or denoising, is a fundamental operation in computer vision and image processing. One of the simplest approaches to smoothing is to perform averaging of nearby points to compute an estimate of the denoised signal. A "box filter" computes an estimate using equal weights for all the nearby sample points. A better estimate of the average would be to afford greater weights to nearby points and smaller weights to more distant points. The Gaussian low-pass filter performs such an averaging using a set of weights defined over a normal distribution such that points nearby the target sample point have a greater contribution to the average than points far away from the sample point. This type of smoothing is isotropic in the sense that the filter application is performed independent of the underlying signal. The result is that it smooths equally in all directions, which has the unfortunate side effect of blurring edges.

In contrast, anisotropic smoothing methods would, ideally, remove noise while preserving important features like edges. Perona [8] developed an anisotropic smoothing technique based upon diffusion. Diffusion-based smoothing methods, which are based upon the solution of partial differential equations, aim to detect region boundaries using a computationally expensive iterative method. The idea is to perform smoothing within, but not across, regions.

Bilateral filtering, as defined by Tomasi [11], aims to perform anisotropic image smoothing using a low-cost, non-iterative formulation. The idea is to smooth images by computing the influence of nearby points in a way that removes noise "within regions," and that does not have the undesirable property of smoothing edge features. This formulation uses a straightforward, tunable estimate for region boundaries: a Gaussian-weighted difference in signal, or photometric space. The idea is that where a sharp edge exists, there will be a large difference in signal. That estimate is combined with a traditional Gaussian-weighted distance function to lessen the contribution from pixels distant in both geometric and signal space.

In bilateral filtering, the output at each image pixel $d(i)$ is the weighted average of the influence of nearby image pixels $\bar{i}$ from the source image $s$ at location $i$. The "influence" is computed as the product of a geometric spatial component $g(i,\bar{i})$ and signal difference $c(i,\bar{i})$.

$$d(i) = \frac{1}{k(i)} \sum g(i,\bar{i}) c(i,\bar{i}) \tag{1}$$

where $k(i)$ is a normalization factor that is the sum of all weights $g(i,\bar{i})$ and $c(i,\bar{i})$, computed as:

$$k(i) = \frac{1}{\sum g(i,\bar{i}) c(i,\bar{i})} \tag{2}$$

While it is possible to precompute the portions of $k(i)$ contributed by $g(i,\bar{i})$, which depend only on the 3D Gaussian PDF, the set of contributions from $c(i,\bar{i})$ are not known *a priori* as

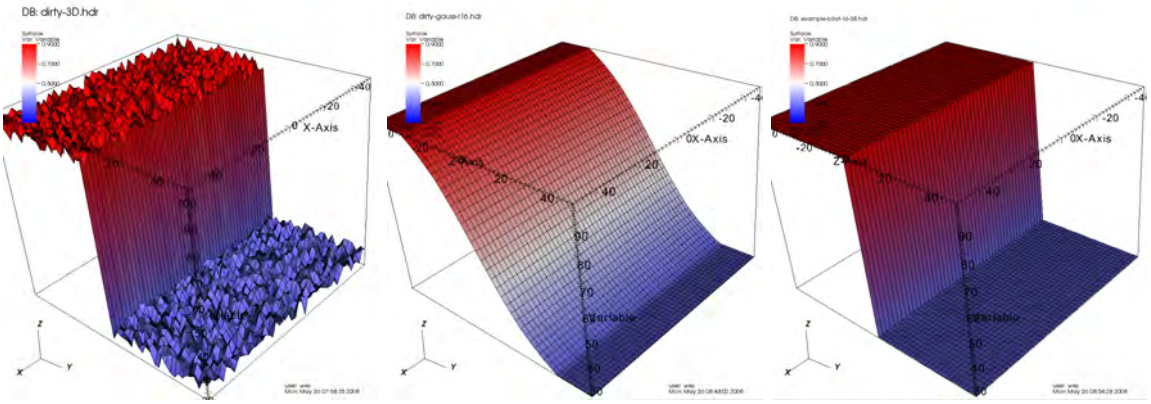
they depend upon the actual set of photometric differences observed across the neighborhood of $c(i, \bar{i})$ and will vary depending upon the source image contents and target location $i$.

Tomasi defines $g$ and $c$ to be Gaussian functions that attenuate the influence of nearby points such that those nearby in geometric or signal space have greater influence, while those further away in geometric or signal space have less influence according to a Gaussian distribution. So,

$$g(i, \bar{i}) = e^{-\frac{1}{2}\left(\frac{d(i,\bar{i})}{\sigma_d}\right)^2} \tag{3}$$

Here, $d(i, \bar{i})$ is the distance between pixels $i$ and $\bar{i}$. The photometric similarity influence weight $c(i, \bar{i})$ uses a similar formulation, but $d(i, \bar{i})$ is the absolute difference $\|s(i) - s(\bar{i})\|$ between the source pixel $s(i)$ and the nearby pixel $s(\bar{i})$.

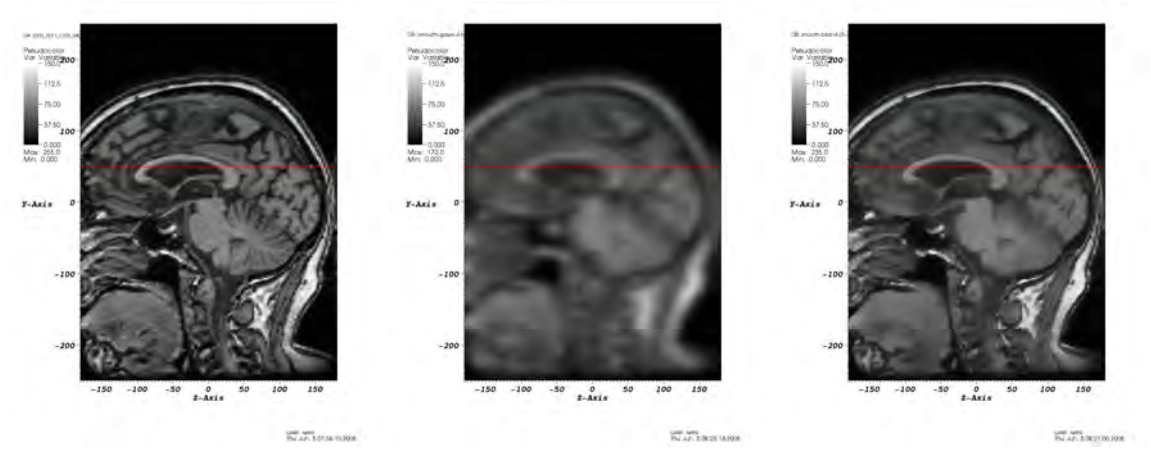
The bilateral filtering approach – combining spatial and signal weights to provide a robust, anisotropic estimate of a smoothed signal – has proven flexible and adaptable to a broad set of applications. Jones adapts this formulation for use in mesh smoothing [6]. There, they replace the photometric difference component with one that measures the difference in facet normals in noisy meshes. More recently, the formulation has been extended for use in smoothing diffusion tensor magnetic resonance imaging data [4]. In that work, the authors replace the notion of photometric similarity with a metric suitable for measuring the dissimilarity of diffusion tensors. Their non-iterative technique combines weighted averages of diffusion tensors with a diffusion tensor dissimilarity metric. The authors also show the results of segmentation operations applied to unfiltered and filtered DTMRI data. In principle, their technique is applicable to 3D DTMRI data, though their results are for 2D data only.




(a) Synthetic, noisy dataset. (b) Gaussian smoothing ($r = 16$) (c) Bilateral smoothing ($r = 16$, $\sigma = 15\%$)

Figure 1: These images show the results of applying 3D Gaussian (middle image) and bilateral smoothing (right image) to a synthetic dataset containing added noise (left image). While Gaussian smoothing blurs the sharp edge in this dataset, bilateral filtering removes noise while preserving the sharp edge.

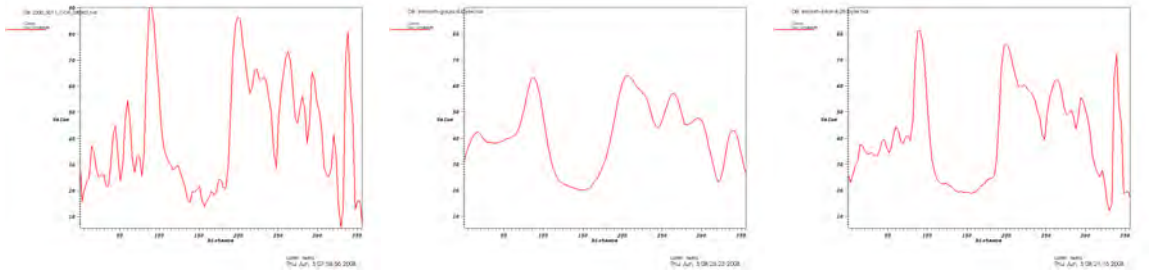
In earlier work, we extended the original Tomasi bilateral filtering formulation for use on 3D volumetric data and compared its scalability and performance on shared- and distributed-memory platforms using several different parallel programming models and execution frameworks [1]. Figure 1 shows how bilateral filtering does a much better job of anisotropic smoothing as compared to traditional Gaussian filtering on a 3D, synthetic dataset with noise. Figure 2 compares bilateral and Gaussian filtering on a 3D medical dataset. In this work, we extend our 3D bilateral filtering formulation for use on the GPU and explore the performance impact of several different algorithmic design choices and tunable algorithmic parameters.



(a) Slice of raw data.

(b) Slice of 3D Gaussian filtered data.

(c) Slice of 3D bilateral filtered data.



(d) Plot of one row of raw data.

(e) Plot of one row of 3D Gaussian filtered data.

(f) Plot of one row of 3D bilateral filtered data.

Figure 2: These images compare the results of 3D Gaussian and 3D bilateral smoothing. The top row contains slices of raw, Gaussian- and bilateral-smoothed data (left to right); the bottom row contains an xy plot of a row of pixels from each slice. We see the 3D Gaussian filter indiscriminately smooths data, whereas the 3D bilateral filter performs smoothing while preserving major features of the original dataset. For both 3D Gaussian and bilateral filtering examples above, the filter radius is equal to four. For the bilateral filtering's photometric difference is set to $\sigma = 15\%$.

## 1.2 GPU Computing

A GPU implementation of bilateral filtering appeared in [3]. That work introduces the idea of a "bilateral grid," which is a data structure for representing an image in a higher-dimensional space. For a two-dimensional image, the three-dimensional bilateral grid contains two spatial dimensions that correspond to pixel location, and the third dimension corresponds to pixel range information, typically pixel intensity. The idea is that the photometric difference between pixels or voxels need not be computed, rather, that difference is implicit due to position in the bilateral grid. To avoid the potential explosion of data that would otherwise result from having a different grid location for every ordinal data value, they use a grid that is much coarser than the original data in terms of both spatial and photometric resolution. The authors implement bilateral filtering, along with several other edge-aware algorithms, on the GPU using a combination of vertex and fragment shaders. Because the bilateral grid enables aggressive downsampling of the source image, they are able to achieve high performance using relatively small memory footprints on the GPU.

In contrast, our work is a direct implementation in CUDA of bilateral filtering in 3D, rather than an approximation. While our approach does not offer any of the advantages resulting from aggressive downsampling, it does produce exactly correct results. Such results are crucial in many applications like clinical medical imaging.

Jiang studies autotuning a matrix multiplication kernel on GPUs [5]. Their system takes a matrix multiplication kernel written in the BrookGPU language [2], a portable high-level programming language for GPUs, and uses techniques to improve data reuse so as to leverage SIMD GPU instructions. They present an algorithm to search the tuning space that relies on a combination of algorithm- and platform-centric heuristics with adaptive search to find the combination of tuning parameters that results in optimal performance. Our work is a manual, rather than automatic, exploration of the performance impact of tunable parameters and algorithmic design choices. Also, our work includes studying the impact of these choices across filters of varying size.

More recently, Ryoo presents a set of performance metrics to estimate the performance of a given optimization configuration for CUDA-based code running on a GPU [9]. They compute two metrics – efficiency and utilization – by examining developer-readable assembler and GPU resource utilization maps produced by the NVIDIA CUDA compiler. The basic idea is to estimate values for each of these metrics by examining resource utilization maps. Then, to avoid an exhaustive search of the optimization space, they estimate the relative performance change by altering parameters that contribute to both metrics. They prune the size of the search space by examining only those configurations that have only high levels in both efficiency and utilization metrics.

Sumanweera presents results describing performance optimization of an FFT kernel on the GPU [10]. The implementation is based upon OpenGL fragment and vertex program extensions rather than CUDA. They solve a load balance problem, where the aim is to keep both stages of a two-stage algorithm filled with work, by automatically searching for the right balance between workload at each of these two processing stages and choosing the one with the best performance. Our work, in part, investigates optimization that might occur well before auto-tuning and automatic selection of tunable algorithm parameters. Namely, we evaluate the performance impact of fundamental algorithmic design choices. Based upon those results, which in effect prune the size of the autotuning search tree, we then conduct a manual exploration of tunable algorithm parameters, thread block size and shape.

# 2 Implementation and Results

## 2.1 Implementation

Our implementation consists of a C `main()`, which is responsible for parsing command line arguments, computing filter weights, loading source medical data, invoking the bilateral filtering kernel (whether a CPU or GPU implementation), and writing the resulting smoothed volume data to disk for subsequent inspection.

The GPU version of the filtering kernel is written in CUDA, and consists of a "main-like" routine that is responsible for allocating memory on the device, copying data from the host to the device, invoking the requested CUDA kernel, and copying the results back to host memory. There are a total of twelve different CUDA kernels in this implementation: they explore different algorithmic implementations, primarily different ways of accessing memory.

With one exception, all memory accessing strategies and block size/shape parameters are specified via the command line to the main program. The one exception is the choice of using global vs. constant device memory for holding the filter weights (Section 2.3.2). In this case, the choice is made by a `#define` inside the CUDA code, thus requiring a recompilation.

## 2.2 Test Machine and Sample Data

All performance experiments in this project were run on a workstation consisting of dual-socket, dual-core 2.0Ghz AMD 270 Opteron CPUs, 8GB of RAM, and an NVIDIA GTX 280. The software environment consists of OpenSUSE 11.0 and the 180.29 NVIDIA OpenGL drivers and the CUDA 2.1 toolkit.

Sample 3D medical data was provided by Prof. Owen Carmichael, UC Davis Departments of Neurology and Computer Science, and the UC Davis Alzheimer's Disease Center. It consists of 3D data acquired by an MRI device, and is of resolution 256x256x120 voxels, each having an approximate volume of $1mm^3$. Data is stored in the Analyze 7.5 format[1]

## 2.3 Different CUDA/GPU Implementations

This section describes several different CUDA implementations of the 3D bilateral filter. Generally speaking, these implementations vary in how they map the problem onto thread blocks and in how each thread accesses memory.

To begin, we investigate the relationship between how a thread accesses device memory and its impact on performance (Section 2.3.1). We measure the performance impact of using high-speed, on-chip memory and high-latency device memory for holding the filter weight tables (Section 2.3.2). After finding the optimal memory access pattern in these sections, we fix those variables and then evaluate the performance impact of varying the number of thread blocks as well as their size and shape (Section 2.3.3). With this information, we can evaluate the relative performance speedup of naive GPU and tuned GPU implementations with the original CPU implementation (Section 2.4).

### 2.3.1 Depth-, Width-, and Height-Row Kernels

In our first CUDA implementation, our objective is to determine the per-thread memory access pattern that yields the best performance. To that end, we implemented three different memory access strategies. The first, called *depth-row* processing, has each thread process all voxels along all depths for a given `(i,j)` location. This approach appears as the blue line painted on the 3D grid shown in Figure 3. A *width-row* order has each thread process all voxels along width at a given `(j,k)` location, and appears as a red line in Figure 3. Similarly, a *height-row* traversal has each thread process all voxels along height at a given `(i,k)` location, and is shown as a green line in Figure 3.

In all of these implementations, the number of thread blocks is set to be one of the width, height or depth (a one-dimensional thread block grid) and the number of threads in each block is set to be one of the remaining dimensions. For example, in a depth-row traversal, the number of thread blocks is set to be the width of the volume, and the number of threads in each block is set to be the height of the volume.

To measure performance, we ran a battery of tests in which we iterate over different-sized filters, and then measure the elapsed time spent in executing the kernel on the GPU. Our application measures the elapsed time to perform problem setup (malloc'ing on-device memory), to copy data to the GPU, to execute the kernel, and to move the results back to the host. For the purposes of these tests, we report only the time spent executing the CUDA kernel. I/O performance is discussed in more detail later in Section 2.5.

The results of the memory access experiment, shown in Figure 4, indicate the depth-row traversal offers the best performance by a significant amount.

The best performance stems from the depth-row algorithms ability to achieve so-called "coalesced memory" accesses. In the case of the depth-row algorithm, each of the simultaneously executing threads will be accessing source and destination data that differs by one address location, e.g., contiguous blocks of memory. The other approaches, in contrast, will be accessing source and destination memory in non-contiguous chunks. The performance penalty for non-contiguous memory accesses is apparent in Figure 4.

[1]See `http://mayoresearch.mayo.edu/mayo/research/robb_lab/analyze.cfm` for more information about the Analyze software, and `http://www.grahamwideman.com/gw/brain/analyze/formatdoc.htm` for information about the Analyze 7.5 data format.

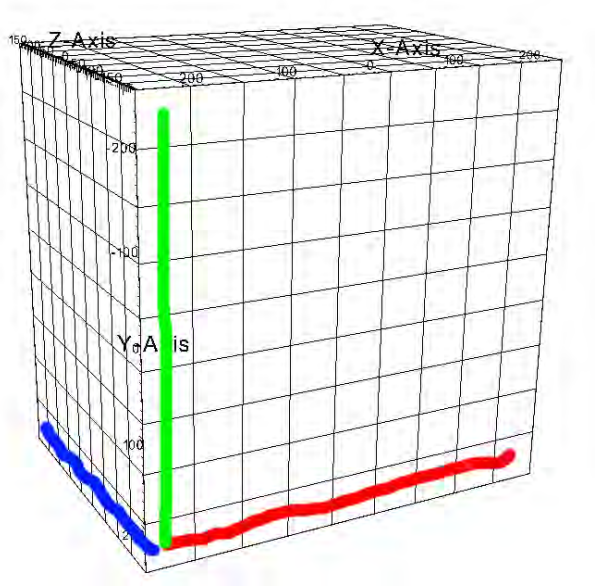



Figure 3: In a depth-row traversal (blue line), a thread processes all voxels along volume depth. The number of thread blocks is set to be the volume width and the number of threads per block is set to be the volume height.

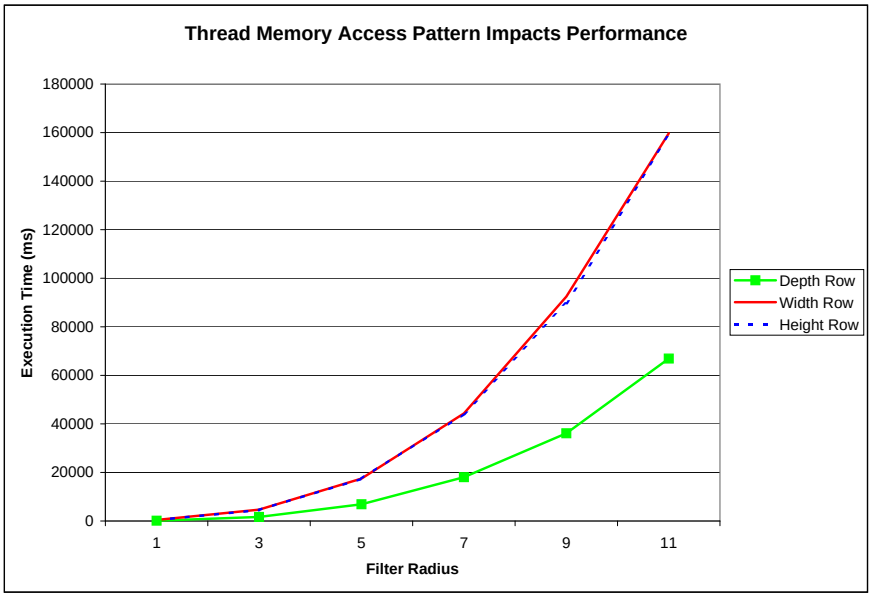



Figure 4: Performance comparison of three different processing kernels. The depth-row algorithm is the dramatic winner due its memory access pattern, which is more efficient than the depth- or height-row algorithms.

### 2.3.2 Constant vs. Global Memory for Filter Weights

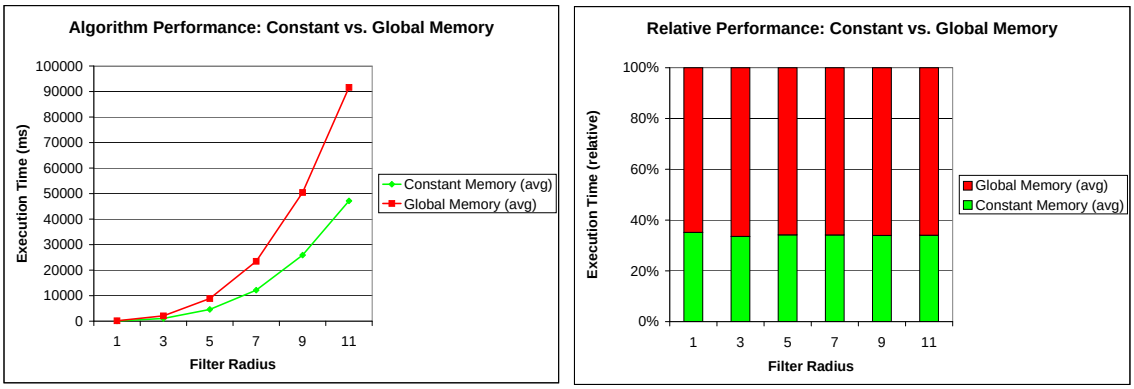
The NVIDIA GPU presents multiple types of memory, some slower and uncached, others faster and cached. According to the NVIDIA CUDA Programming Guide (see [7]), the amount of global (slower, uncached) and texture (slower, cached) memory varies as a function of specific customer options on the graphics card. Under CUDA v1.0 and higher, there is 64KB of *constant* memory and 16KB of *shared* memory that is resident on-chip and visible to all threads. Generally speaking, device memory (global and texture memory) has higher latency and lower bandwidth than on-chip memory (constant and shared memories).

In the case of our implementation, the source $S$ and destination $D$ volumes are too large to fit into either of the on-chip memories (constant or shared). However, the filter weights, both 3D and 1D, will fit into on-chip memory under most conditions, which are as follows:

- **Constant memory.** Given the size of constant memory is 64KB, the maximum filter size we can accommodate is $\sqrt[3]{\frac{64KB}{4}}$, or a filter that is about 25x25x25 floating point values.
- **Shared memory.** Given the size of shared memory is 16KB, the maximum filter size we can accommodate is $\sqrt[3]{\frac{16KB}{4}}$, or a filter that is about 15x15x15 floating point values. Further investigation reveals this 16KB memory is split into 16 banks across the multiprocessors. This organization suggests a maximum 1KB usable shared memory size in the context of caching filter weights. This amount of memory is not enough to meet our needs for this problem, so further investigation into use of shared memory as a cache for filter weights was abandoned.

The performance question we wish to understand in this context is "how is performance impacted by having all of the filter weights resident in on-chip rather than in device memory?"

To begin, we ran a battery of tests comparing the performance of having all filter weights being resident first in device memory then in constant memory. The results, shown below in Figure 5, indicate the implementation where filter weights are in device memory performs twice as slowly as the implementation where filter weights are in constant memory. This result is not surprising given the different latency and bandwidth characteristics of these two different memory subsystems.




(a) Absolute performance (smaller is better).

(b) Relative performance (smaller is better).

Figure 5: These results compare the absolute and relative performance difference of having filter weights resident in device versus constant memory. In all ranges of filter sizes, we see the constant memory implementation performs about twice as fast as the device memory implementation.

### 2.3.3 Performance Impact of Varying Thread Block Size

Next, we wish to understand the performance impact of varying the size and shape of thread blocks. While this type of activity is typically the domain of autotuning, our investigation here is a "manual exploration" of this algorithmic tunable parameter. Our objective is to determine whether or not thread block/size has an impact on performance, and if so, to determine the thread block size and shape that produces optimal performance.

To begin, we used the "best practices" from the previous sections: we use a depth-row traversal for our thread kernel; our thread kernel contains no conditionals (and the row size for our volume data just happens to fall exactly on 128-byte boundaries in memory); we store the filter weights (both 3D and 1D) in constant memory to leverage fast, on-chip memory.

Next, we modified the code to allow the user to specify the number of thread blocks in each dimension[2] Then, we wrote a script that invoked the application over a range of different thread block sizes and shapes. In all cases, each thread block processes all voxels along depth. We performed this battery of tests over a range of filter sizes to fully explore the potential performance impact of block size/shape. As expected, the size and shape of thread block has an impact on performance. What is somewhat surprising is the amount of impact the size and shape has on performance.

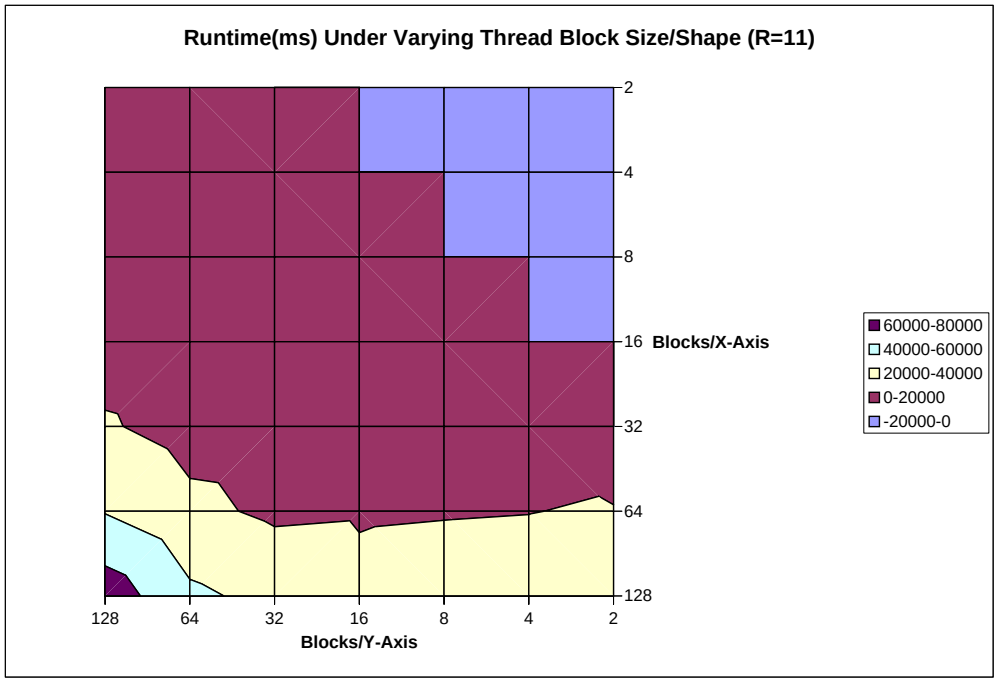



Figure 6: The size and shape of thread blocks can have a profound impact on performance. Here, we see the worst performance results when having a large number of thread blocks with a small number of threads (lower left corner of chart). The best performance results when having fewer thread blocks having more threads responsible for contiguous X locations in the volume. The maroon region reflects performance levels that are about three times better than the worst configuration.

Figure 6 shows the results of this test battery run on the largest-size filter (radius=11, filter box size=(23x23x23)). The axes of that figure are the number of thread blocks per axis. A large value means there are relatively few threads per block – the number of threads per block is computed as $\frac{VolumeSize_i}{BlocksPerAxis_i}$. In the lower left corner of the chart, there are 128x128 total thread blocks, each of which has four threads. In the upper right, there are 2x2 total thread blocks, each of which has 128x128 threads. This case is not valid since CUDA permits a maximum of 512 threads per thread block. These "invalid" configurations appear in Figure 6 as light blue blocks. The legend in Figure 6 reports the absolute runtime of the tested configuration in milliseconds. The absolute maximum runtime for the $r = 11$ filter size was about 9.4s for the (8, 128) configuration.

In this case, (8,128) means each thread block had a total of $256/8 + 256/128$, or 64 threads. Each thread block was responsible for 32 contiguous I locations and 2 contiguous J locations, and each thread performed its processing in depth. Due to MS Excel oddities, the X and Y axes in Figure 6 are reversed, so proper interpretation requires looking for the coordinate labeled (128,8) in the upper left portion of the chart. Also due to MS Excel oddities, this image is about the best it can produce. Attempts with several other visualization tools didn't produce any more satisfying results.

[2]It was disappointing to learn that NVIDIA does not support a 3D grid of thread blocks at this time.

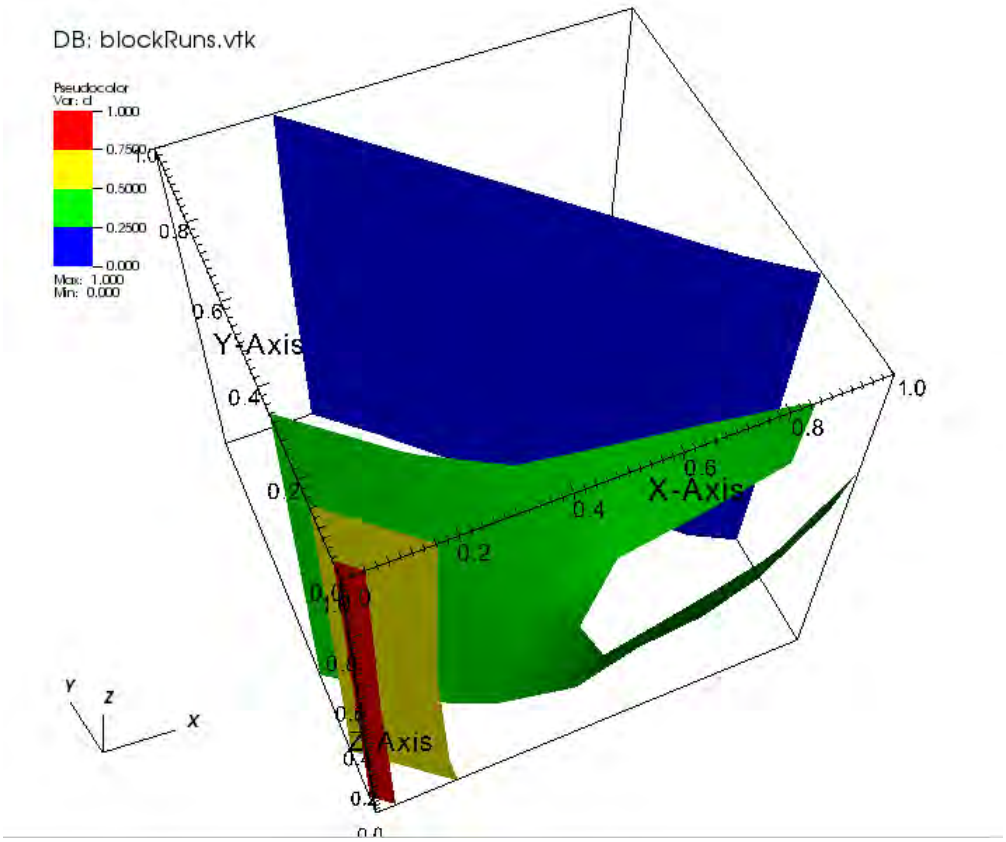



Figure 7: Here we see the normalized runtime performance of the 3D bilateral filter over varying block sizes and shapes and for three different filter levels. The "Y axis" is the number of thread blocks along the X axis of the volume; the "X axis" is the number of thread blocks along the Y axis of the volume. The "Z axis" corresponds to different filter levels: $r = 1$, $r = 5$, and $r = 11$. Locations closer to the origin of the chart correspond to configurations having more thread blocks with fewer threads per block. Towards larger coordinates, we have fewer thread blocks having larger numbers of threads: it is in these regions, particularly fewer blocks along the "Y axis" of the image, or X axis of the volume, where we see the best algorithmic performance.

Next, we wished to determine if the maxima and minima present in the $r = 11$ case extended across a range of filter sizes. We ran two additional test batteries at $r = 5$ and $r = 1$. A 3D view of these results appears in Figure 7. In that image, we see the "invalid configurations," namely those that result in more than 512 threads per thread block, being present in the upper right hand corner "behind the blue curtain." Valid configurations lie in front of the blue curtain. In this figure, all run times have been normalized to the range $0 \ldots 1$ where a value of 1.0 is the maximum runtime for a given filter size.

On interesting feature is that all block size/shape configurations that are close to the (X,Y) origin result in poor performance. These correspond to configurations where there are relatively few threads per thread block. Another interesting feature is that the relative performance as a function of block size and shape appears more or less consistent across all filter sizes tested. Perhaps the most interesting feature is where the green surface intersects what is labeled the "Y-axis." Here, the "Y-axis" corresponds to the number of thread blocks along the X axis of the volume. The conclusion we draw is that block configurations that perform well at any given filter size will likely perform similarly well for other filter sizes.

Figure 8 shows a slightly different view of the same data. Here, the raw performance data has been normalized by the minimum value at each of the three different filter sized. Thus, the minimum execution time for each filter normalizes to 1.0, and all other values are larger. This

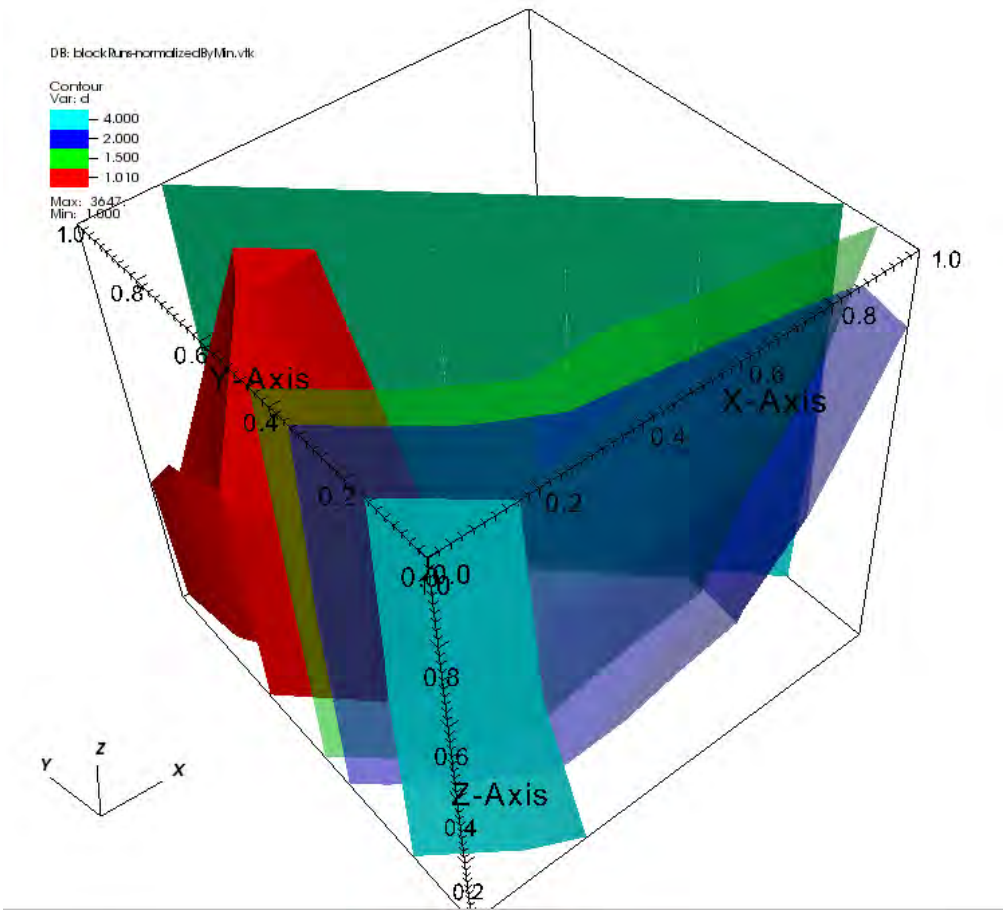



Figure 8: The axes and orientation are the same in this figure as in Figure 7. In this image, the execution time for the different block size/shape configurations is scaled by the minimum run time within each filter size. This type of scaling makes it easier to quickly identify the minimum values, whereas Figure 7 shows the maximum values. The performance "sweet spot" is enclosed within the red volume, which corresponds to normalized runtime values close to 1.0.

image shows the "sweet spot" across all configurations as the region enclosed by the red surface, which contains values close to or equal to 1.0. The x/y-plane closest to the viewer corresponds to the smallest filter size ($r = 1$), while the one furthest from the viewer corresponds to the smallest filter size ($r = 11$). We observe that the "sweet spot" appears to be much smaller for the smallest filter size (close to the viewer) than for the largest filter size.

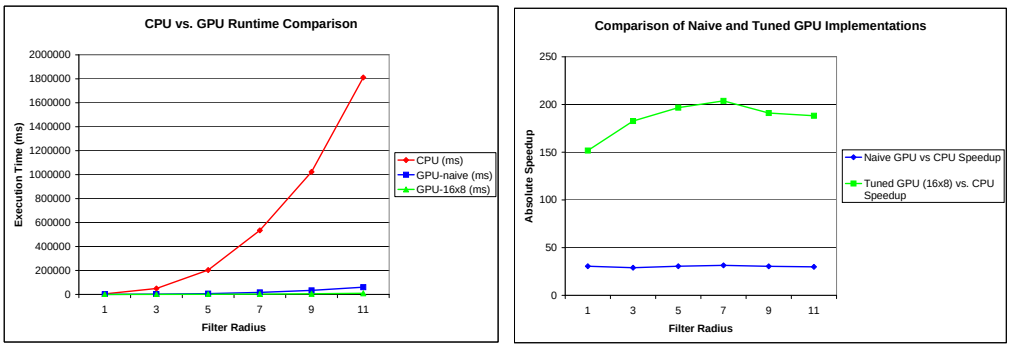
Interestingly, this result is confirmed by a related study (research in progress, no data available at this time) that uses an autotuning tool to automatically explore configuration settings for this algorithm on both GPU and multicore CPU architectures.

## 2.4 GPU vs. CPU Comparison

As with many GPUGPU endeavors, it is always worthwhile to compare the runtime performance of the GPU and CPU implementations of the algorithm. Figure 9 compares the performance of a CPU implementation, a "naive" GPU implementation (our untuned depth-row traversal), and a GPU implementation where we chose a thread block size/shape that corresponds to an optimal performance configuration discovered in previous section.

Figure 9(a) shows both GPU implementations vastly outperform the CPU implementation, particularly at larger filter sizes. Figure 9(b) shows the degree to which the GPU algorithms

outperform the CPU. The naive GPU implementation shows a steady 30x performance advantage over the CPU implementation. The tuned GPU implementation[3] outperforms the CPU implementation by amounts ranging from between 150x to over 200x. In general, the tuning exercise in the previous section resulted in selection of a set of configuration parameters that result in about a 6x speedup.




(a) Both GPU implementations dramatically outperform the CPU implementation, particularly at larger filter sizes.

(b) Selecting the right set of configuration parameters for the GPU implementation can have a profound impact on GPU algorithm performance.

Figure 9: Comparison of CPU and GPU runtimes. The naive GPU implementation outperforms the CPU implementation by about 30x; the tuned GPU implementation outperforms the CPU implementation by between 150x and over 200x.

## 2.5 "Unexpected" Results

One unexpected result that arose while testing was the observation that I/O rates between the host and GPU are drastically different depending upon whether or not the X server is running. Figure 10 shows the amount of elapsed time required to copy our problem's dataset, a volume consisting of 256x256x120 4-byte voxels, between the host and the GPU.

We observe that the time needed top copy the resulting volume from the GPU back to the host is about 15% faster when the X server is running. More noteworthy is the observation that the time required to copy the volume from the host to the device is about 10x slower when the X server is not running.

This discovery came about when testing larger filter box sizes, all of which run for a period of time ranging between a few and tens of seconds. When our CUDA program had an execution duration of more than a few seconds, the X server on the test machine would "become wedged" to a degree requiring a hard reboot. In order to avoid this annoying and time-consuming side-effect, we ran the majority of test runs without an X server. The test program is instrumented to measure and report the amount of time required inside the CUDA portion needed to move data from the host to the GPU, the time elapsed while running the filtering kernels, and the time required to move the resulting filtered volume back to the host. The unusual I/O performance characteristic was observed while comparing test results taken from runs with and without an X server.

[3]The "optimal" configuration consists of 16x32 thread blocks, each being responsible for 16x8 threads. This optimal configuration was discovered by manual inspection of the data produced by experiments described in Section 2.3.3.

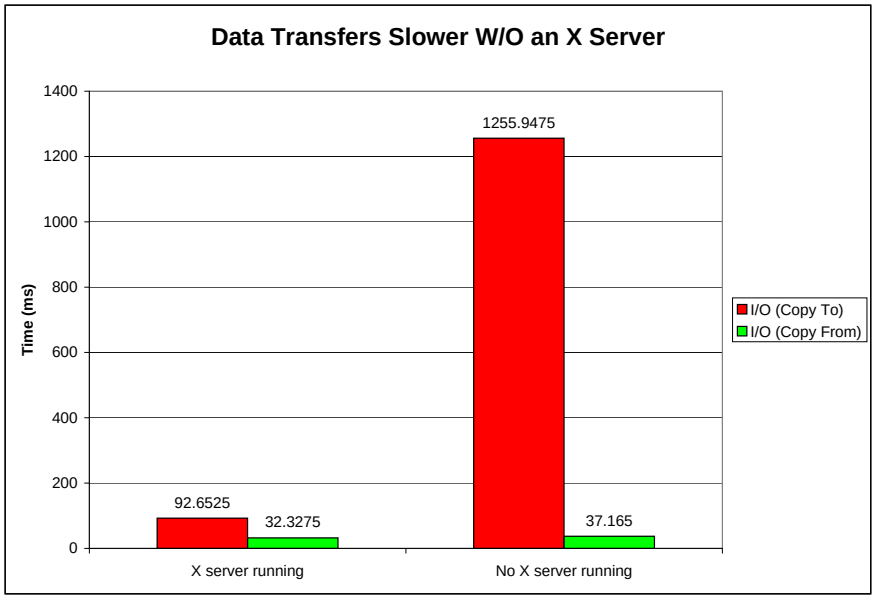



Figure 10: Our testing reveals that data transfers from the host to the GPU are substantially faster – approximately 10x faster – when the CUDA program is run while an X server is running.

# 3 Conclusion and Future Work

This project investigates several different factors that can have an impact on performance for a GPU implementation of a 3D bilateral filter. These factors include: algorithmic design, to access memory in an effective way; use of high-speed, on-chip memory for storing data that is frequently read by all processing threads; determining the performance impact of thread block size and shape. We found that a "naive" GPU implementation runs about 30 times faster than a single-threaded CPU implementation, and that an "optimized, tuned" GPU version runs between 150 and 200 times faster than a single-threaded CPU implementation.

Future work would include investigating additional algorithmic design options to further enhance performance. One idea would be to perform memory blocking, where subsets of the source volume are copied (via windowing) into high-speed, on-chip shared memory for use by processing threads. This strategy could significantly improve performance by eliminating redundant reads from slower global memory.

Another idea is to explore the use of autotuning tools that perform more extensive optimizations, like loop unrolling. The work in this paper is a manual exploration of a few parameters. A fully optimized GPU implementation will likely involve considering a larger number of design and tunable parameters that are outside the scope of this particular study.

# References


[1] E. Wes Bethel. High Performance, Three-Dimensional Bilateral Filtering. Technical Report LBNL-1601E, Lawrence Berkeley National Laboratory, 2009.

[2] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. In *Proceedings of ACM Siggraph*, August 2004.

[3] Jiawen Chen, Sylvain Paris, and Frédo Durand. Real-time Edge-aware Image Processing with the Bilateral Grid. In *SIGGRAPH '07: ACM SIGGRAPH 2007 papers*, page 103, New York, NY, USA, 2007. ACM.

[4] G. Hamarneh and J. Hradsky. Bilateral Filtering of Diffusion Tensor Magnetic Resonance Images. *IEEE Transactions on Image Processing*, 16(10):2463–2475, Oct. 2007.

[5] Changhao Jiang and Marc Snir. Automatic Tuning Matrix Multiplication Performance on Graphics Hardware. In *Proceedings of the Fourteenth International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 185–196, September 2005.

[6] Thouis R. Jones, Frédo Durand, and Mathieu Desbrun. Non-iterative, Feature-preserving Mesh Smoothing. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, pages 943–949, New York, NY, USA, 2003. ACM.

[7] NVIDIA Corporation. *NVIDIA CUDA$^{TM}$ Version 2.1 Programming Guide*, 2008.

[8] P. Perona and J. Malik. Scale-Space and Edge Detection Using Anisotropic Diffusion. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(7):629–639, 1990.

[9] Shane Ryoo, Christopher I. Rodrigues, Sam S. Stone, Sara S. Baghsorkhi, Sain-Zee Ueng, John A. Stratton, and Wen mei W. Hwu. Program optimization space pruning for a multithreaded GPU. In *CGO '08: Proceedings of the Sixth Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 195–204, 2008.

[10] Thilaka Sumanaweera and Donald Liu. Medical Image Reconstruction with the FFT. In Matt Pharr, editor, *GPU Gems 2*, chapter 48, pages 765–784. Addison Wesley, March 2005.

[11] C. Tomasi and R. Manduchi. Bilateral Filtering for Gray and Color Images. In *ICCV '98: Proceedings of the Sixth International Conference on Computer Vision*, page 839, Washington, DC, USA, 1998. IEEE Computer Society.