Parallel GPU Image Processing Algorithms

Henry Crute, Huy Doan, Xin Xu

University of Texas at Austin

EE382C-Multicore Project Report

November 17, 2016

1 Abstract

We implemented and analyzed two different filters for image processing on the CPU and compare it to implementations on the GPU. The Gaussian Blur filter single threaded complexity is typically $O(n \cdot r^2)$. That algorithm can be improved to a linear approximation of the Gaussian filter that is much faster with a complexity of $O(n)$. The Bilateral filter is like Gaussian blur, but it preserves the edges between images and blurs portions of the images where there aren't edges. This algorithm requires more computation compared to Gaussian Blur as the single threaded complexity is $O(n^2)$. We don't get in to the details of Bilateral filter approximations to speed up the algorithm. The Gaussian and Bilateral filter algorithms different complexity, and parallelizability are analyzed. We attempted to exploit maximal parallelism in these algorithms translating them to multi-threaded algorithms for the GPU using CUDA.

2 Introduction

In recent years, the computation speed and user usage of the graphics processing unit (GPU) has increased rapidly. Applying image processing algorithms to this task is a natural fit for parallel data processing. An image often consists of thousands of pixels, and filtering each pixel is an operation which is independent of the other neighbor pixels. The algorithms can usually map each pixel directly to threads. The Gaussian filter (also known as Gaussian smoothing) is the result of blurring an image by a Gaussian function. It is a widely used effect in graphics software, typically to reduce image noise, and detail. The Bilateral filter is a nonlinear filter that smooths an image while preserving strong edges. These filters has demonstrated great effectiveness for a variety of problems in computer vision and computer graphics.

This paper implements the Gaussian filter and Bilateral filter algorithms with a single thread on the CPU and many threads with Nvidia CUDA technology. A much lower run-time can be achieved by using CUDA, compared to the CPU. A sequential implementation of the Gaussian and Bilateral Filter are given and analyzed to find which parts of the algorithms can be parallelized and which parts have data dependencies. Different parallel implementations are also explored, suggesting various methods for optimizing spatial filtering using CUDA.

3 Description

We will first discuss the typical literature implementation of the Gaussian and Bilateral filters, and then go into detail about our parallel implementations. We compare the non-parallel implementations to the parallel implementations. This analysis includes our ideas on what the runtime of the algorithms are comparatively.

3.1 Gaussian Filter

Filtering an image with 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 a corresponding weight. A resulting image of our implementation of the Gaussian Filter is in appendix 1.

The one and two dimensional Gaussian functions are defined in equation 1 and 2.

$$G(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{x^2}{2\sigma^2}} \tag{1}$$

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{2}$$

The Gaussian smoothing operator is a 2-D convolution that is used to 'blur' images and remove detail and noise.The discrete convolution of function f and the weight g to produce the blur is shown in equation 3.

$$b[i,j] = \sum_{y=i-r}^{i+r} \sum_{x=j-r}^{j+r} f[y,x] * w[y,x] \tag{3}$$

An algorithm implementing this convolution will have the complexity $O(n \cdot r^2)$ where n is the size of the image and r is the Gaussian blur radius. An improvement of this is to do 2D box blur by first doing a horizontal motion blur and then doing a vertical motion blur. The horizontal motion blur and the vertical motion blur are outlined in equations 4 and 5 respectively.

$$b_h[i,j] = \sum_{x=j-r}^{j+r} f[i,x]/(2 \cdot r) \tag{4}$$

$$b_t[i,j] = \sum_{y=i-r}^{i+r} f_h[y,j]/(2 \cdot r) b_t[i,j] = \sum_{y=i-r}^{i+r} \sum_{x=j-r}^{j+r} f[y,x]/(1 \cdot r)^2 \tag{5}$$

The result of equation 5 uses equation 4. This improvement reduces the complexity of the process to $O(n \cdot r)$. A further improvement is that we could calculate motion blur faster than that. The observation is that $b_h[i, j]$ and $b_h[i, j+1]$ are very similar and only differ by the most left and most right values. Therefore we can calculate $b_h[i, j+1]$ in constant time algorithmic time in equation 6.

$$b_h[i, j+1] = b_h[i, j] + f[i, j+1+r] - f[i, j-r] \tag{6}$$

This improvement leads to an algorithm with independence of the radius r and the complexity $O(n)$. To summarize the improvements for Gaussian blur, we do box blur 3 times and in each box blur, we do 2 1D motion blurs. The final complexity of the algorithm is $O(6 \cdot n)$. Note that this filter is an approximation of the original, and that is how complexity is mostly reduced [1].

3.2 Bilateral Filter

Bilateral Filter replaces each pixel by a weighted average of its neighbors. The weight assigned to each neighbor decreases with both the distance in the image plane (the spatial domain S) and the distance on the intensity axis (the range domain R). Using a Gaussian G_σ as a decreasing function, the result I^{bf} of the bilateral filter is defined by equations 7 and 8.

$$W_p = \sum_{q \in N(p)} G_{\sigma_s}(p, q) G_{\sigma_r}(|I_p - I_q|) \tag{7}$$

$$BF(I)_P = \frac{1}{W_p} \sum_{q \in N(p)} G_{\sigma_s}(p, q) G_{\sigma_r}(|I_p - I_q|) I_q \tag{8}$$

The term in front of the sum in equation 8 is a normalization term, which makes sure that the overall intensity of the image remains the same. The added Gaussian function 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 σ_r becomes, the more the filter converges towards a regular Gaussian Blur.

3.3 Parallel Implementation

We analyzed the different sums and loop sections of the algorithms and tried to exploit the most parallelism we could manage to get out of the threads on the GPU.

3.3.1 Gaussian Blur Filter

Due to the nature of the Gaussian Filter Approximation in equations 4 and 5, the horizontal and vertical portions of the algorithm can be separated and be computed in parallel. The first advantage is when the algorithm does box blur, it needs to copy the source to the destination and each thread of the GPU will copy one pixel. The second advantage is in motion blur, each row and column can be process independently. We assigned each row/column to individual threads. We experimented by comparing the implementations of using as many threads in a block as possible against using one thread/block but many blocks. Although it may take up more blocks than needed, the former approach resulted in much lower processing time. We could have parallelized the computation of the boxes; however, due to its very small size, we left the simple calculations to the CPU instead.

Each color channel was processed separately and serially. The new algorithm should run in $O(n)$ time. The performance will even be much greater if we can figure out how to parallelize the calculation of the three color channels. Due to the thread dispatch complexity and implementation time constraint, this can be future work.

3.3.2 Bilateral Filter

The improvement we made to the Bilateral Filter was to have individual threads mapped to each pixel in the image, using a radius to define the size of the local pixel neighborhood. This kernel is then launched with parameters which maps a thread for each pixel in the image. If there are enough threads for each pixel in the image, the algorithm should almost turn in to $O((2 \cdot r)^2)$.

Future work could use threads for the internal sums done for each pixel. We would have to dispatch threads for a parallel prefix sum of the values instead of a regular serial sum per GPU thread that we're doing now. The hardest part of this task would be to map threads to the task dynamically as the sum ranges change.

4 Results

We tested our code and measured our results using the Stampede server from TACC. The log-in node was used for the single core implementation of our algorithms, and the GPU compute nodes were used for the parallel implementations. We can compare the parallelism in each algorithm, and how they attributed to our results in appendix B using the radius and total number of pixels.

4.1 Input size of the image

4.1.1 Gaussian Filter

We expected the new algorithm for the Gaussian blur to be improved much more than the original naive implementation. The improved Gaussian Approximation algorithm performed an order of magnitude (about 10 times) better with acceptable margin of error. See appendix B for the resulting data in table 1. The GPU implementation of the Gaussian Approximation performed about 5 times better for smaller images, and about 18 times better for larger images compared to the CPU version. The function seems to reduce run-time even more for larger images. This is due to the fact that the algorithm can be run separately on each row and column of the image. The larger the size, the more the GPU implementation can compute in parallel. The same behavior is observed in figure 3.

4.1.2 Bilateral Filter

For the CPU implementation of the Bilateral filter, the run-time scales according to how many pixels are in the image. Since we know that the number of pixels in an image grow in a nonlinear squared fashion, this affirms our predictions of the algorithm being $O(n^2)$. It is not obvious from the graph, but the GPU implementation's slope is almost linear, but not quite. This doesn't exactly match the prediction of the being constant when compared to the image size (it was a function of the radius). There may be more overhead with dispatching a larger number of blocks and threads. The GPU implementation performs about 200 times faster than the CPU implementation over all image sizes. This means run-time is growing dependently of image size, which wasn't expected for the prediction of $O((2 \cdot r)^2)$ complexity.

4.2 Filter Radius

We expected that the Gaussian had nearly constant time with different radii since the algorithm is independent of the radius. The result in table 2 matches well with the expectation. It is worth noticing that the GPU implementation had more stable performance than the CPU one. As expected, when the radius increases for the Bilateral filter, it increases as a function of the radius squared added over all work.

4.3 Gaussian Filter vs. Bilateral Filter

Comparing the parallelism of both algorithms, the GPU implementation improved the CPU implementation more for the Bilateral Filter than the Gaussian Filter. This is because the single threaded Gaussian Approximation was already optimized before we translated it using CUDA. Extra analysis could be done by implementing a GPU version of the regular Gaussian filter, and comparing it to the GPU version of the Gaussian Approximation.

Both GPU filters performed similar over image sizes (figure 6). One particularly interesting thing to note is that for smaller images up to 1500x1500 resolution, the Bilateral filter runs faster than the Gaussian Approximation, but for images larger than that, the Bilateral filter seems to run better. The run-time function for the Bilateral filter grows faster than the function for the Gaussian approximation filter.

5 Conclusion

For the Gaussian blur, the improved algorithm performs better than the naive algorithm and is independent on the size of the radius. The GPU implementation even has a much better performance gain than the CPU one. The point of the improved algorithms is to maximize the parallel portion in order to have better gain with GPU. Future work may include research as to if downsampling, filtering, and then upsampling of the images yields any performance gain with the image retaining the quality of the regular filtered image [3]. More tests and comparisons can be done to further analyze the behavior of these algorithms in GPU systems compared to CPU systems.

References

[1] Ivan Kucskir. Fastest gaussian blur. ”http://blog.ivank.net/fastest-gaussian-blur.html”, 2016. Accessed: 2016-11-16.

[2] NVIDIA. Cuda toolkit documentation v8.0. ”http://docs.nvidia.com/cuda/index.html#axzz4QEIkVu7D”, 2016. Accessed: 2016-11-16.

[3] Sylvain Paris and Frédo Durand. A fast approximation of the bilateral filter using a signal processing approach. *International Journal of Computer Vision*, 81(1):24–52, 2009.

A

Our Filtered Images

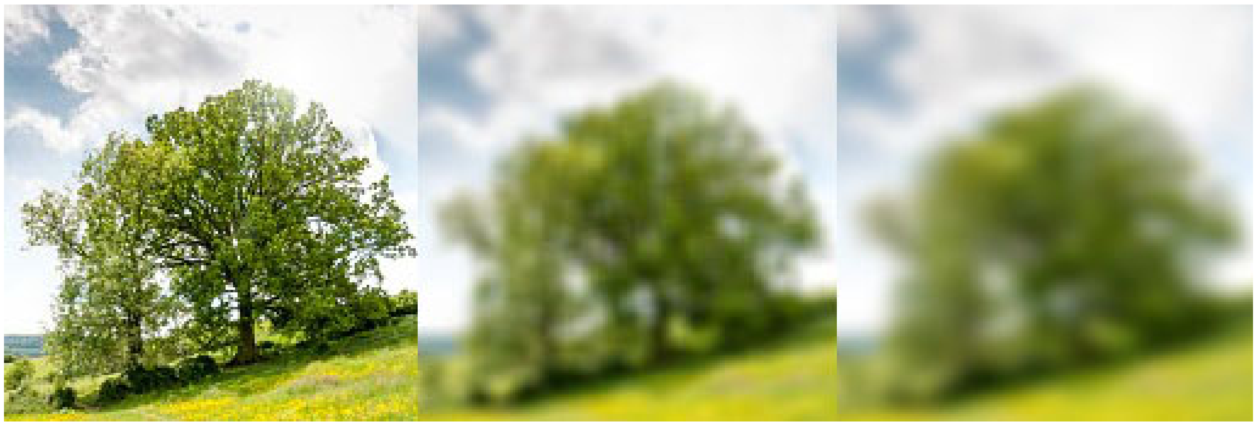

Figure 1: Original image (left), Gaussian filtered image (center), and Gaussian approximation image (right).

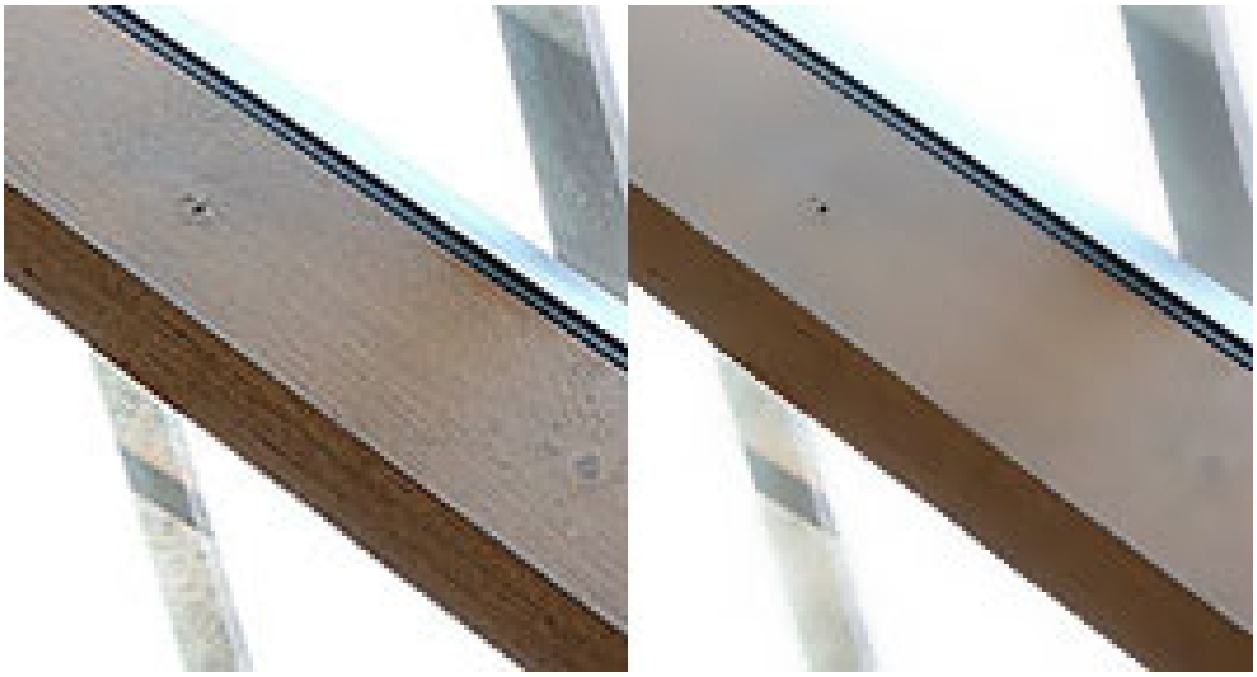

Figure 2: Original image (left) and the bilateral filtered image (right).

B

Runtime Graphs and Tables

Measurements are done over an average for measurement of the typical time for the tasks. The radii are set to a constant 4 for all of the figures.

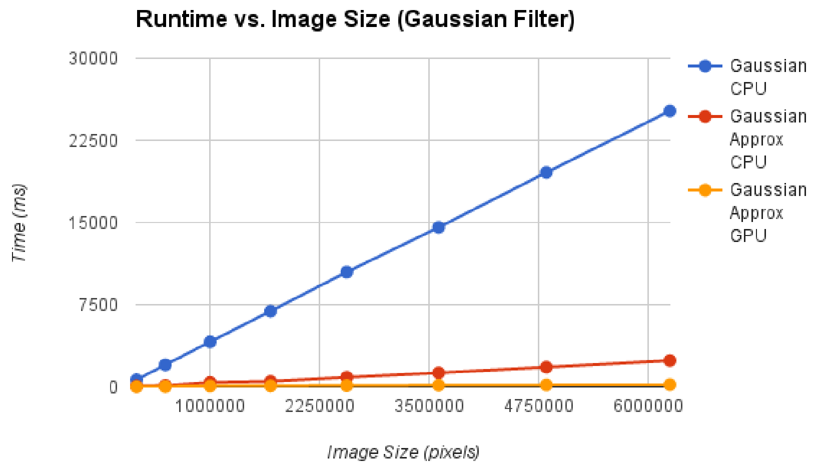

Figure 3: Original image (left) and the bilateral filtered image (right).

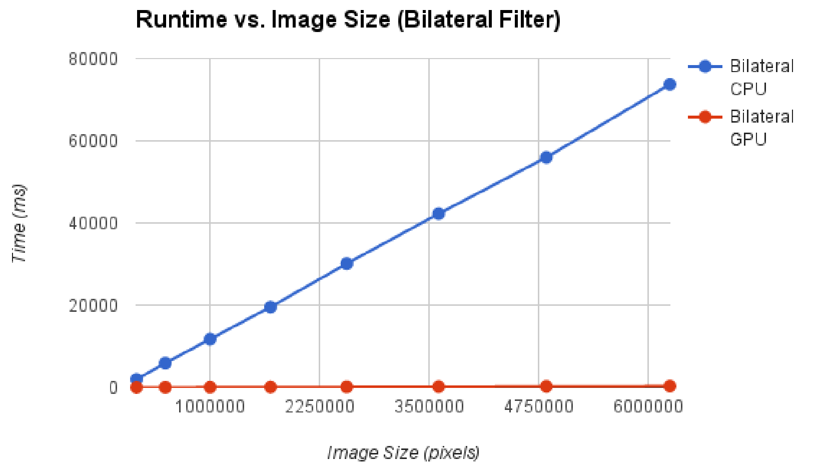

Figure 4: Original image (left) and the bilateral filtered image (right).

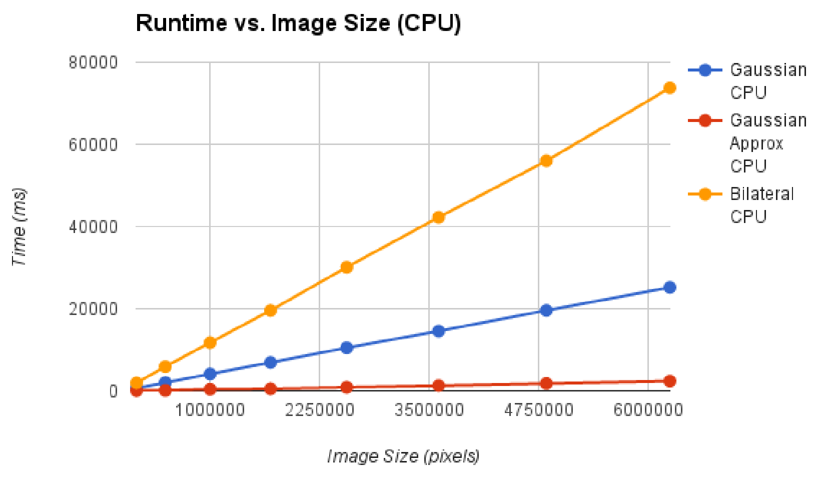

Figure 5: Original image (left) and the bilateral filtered image (right).

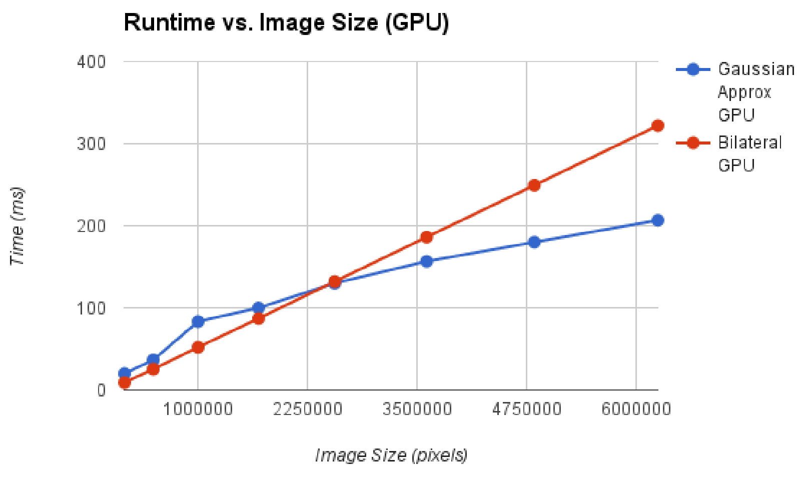

Figure 6: Original image (left) and the bilateral filtered image (right).

Iamge Size	Naive Gaussian(CPU)	Gaussian(CPU)	Gaussian(GPU)	Bilateral(CPU)	Bilateral(GPU)
500x300	640	37.9	16.6	1880	7.99
1920x1080	8815	685	145.7	26120	102.4
2560x1600	17395	1470	196.5	50650	208.51
3836x2877	47020	4750	302.6	138450	556.57
4390x2948	55140	5715	344.5	163025	645.51
5696x4016	97606	9974	458.2	287665	1149.85

Table 1: Image size impacts on Running Time (ms)

Radius	Naive Gaussian(CPU)	Gaussian(CPU)	Gaussian(GPU)	Bilateral(CPU)	Bilateral(GPU)
4	640	37.9	18.1	1880	7.99
5	950	35.8	16.7	2810	11.77
6	1320	34.6	17.8	3910	16.37
7	1760	33.7	17.3	5200	21.70
8	2250	32.2	16.4	6670	27.78
9	2810	32.2	16.9	8350	34.58
10	3430	32.5	16.5	10180	42.12

Table 2: Filter Radius(pixels) impacts on Running Time (ms) IMAGE SIZE: 500x300