Mälardalen University

Master Thesis

A Case Study of Parallel Bilateral Filtering on the GPU

Author:
Jonas Larsson

Supervisor:
Thomas Larsson

Examiner:
Björn Lisper

November 18, 2015

Abstract

Smoothing and noise reduction of images is often an important first step in image processing applications. Simple image smoothing algorithms like the Gaussian filter have the unfortunate side effect of blurring the image which could obfuscate important information and have a negative impact on the following applications. The bilateral filter is a well-used non-linear smoothing algorithm that seeks to preserve edges and contours while removing noise.

The bilateral filter comes at a heavy cost in computational speed, especially when used on larger images, since the algorithm does a greater amount of work for each pixel in the image than some simpler smoothing algorithms. In applications where timing is important, this may be enough to encourage certain developers to choose a simpler filter, at the cost of quality. However, the time cost of the bilateral filter can be greatly reduced through parallelization, as the work for each pixel can theoretically be done simultaneously.

This work uses Nvidia's Compute Unified Device Architecture (CUDA) to implement and evaluate some of the most common and effective methods for parallelizing the bilateral filter on a Graphics processing unit (GPU). This includes use of the constant and shared memories, and a technique called 1 x N tiling. These techniques are evaluated on newer hardware and the results are compared to a sequential version, and a naive parallel version not using advanced techniques. This report also intends to give a detailed and comprehensible explanation to these techniques in the hopes that the reader may be able to use the information put forth to implement them on their own.

The greatest speedup is achieved in the initial parallelizing step, where the algorithm is simply converted to run in parallel on a GPU. Storing some data in the constant memory provides a slight but reliable speedup for a small amount of work. Additional time can be gained by using shared memory. However, memory transactions did not account for as much of the execution time as was expected, and therefore the memory optimizations only yielded small improvements. Test results showed 1 x N tiling to be mostly non-beneficial for the hardware that was used in this work, but there might have been problems with the implementation.

Acknowledgements

I would like to thank my girlfriend Caroline, my family, and my supervisor Thomas for their patience and support. I would also like to thank Mälardalens Datorförening (MDF) for providing me with a place to work and a computer to work on.

List of Figures

Contents

1

Introduction

In signal processing, the word **noise** refers to unwanted variations in a signal. An example of noise in an audio signal would be scratching sounds in an old recording. Noise is also common problem in images, and the removal of different kinds of noise is an important field of study. Common examples of image noise are film grain and speckle noise in medical ultrasound images. Since the early beginnings of digital image processing, a common problem has been how to remove noise from images in order to propagate the features of interest in them. This is one of the most common uses for a smoothing filter. A smoothing filter is a digital filter that smooths or blurs an image. This will often remove noisy details in an image but can also have the effect of blurring the actual features of the image, which may be detrimental for many image processing or computer vision tasks. Several more advanced techniques have been developed to deal with this issue, and one of them is the bilateral filter [25]. The bilateral filter extends a typical smoothing filter with edge detecting properties in order to avoid blurring edges of images. This results in smooth surfaces without obscuration of major details.

The bilateral filter is computationally expensive and has a non-linear component which complicates optimization. When run sequentially on a normal Central processing unit (CPU), it takes a lot of time, especially on larger images. This problem has been addressed many times by different people, and there exists a wide range of fast approximations of the bilateral filter(see later section). Instead of making changes to the algorithm itself, this work speeds up the bilateral filter in its regular form by running it in massive parallel on a GPU.

GPUs consist of several Streaming Multiprocessors (SMs), which are capable of launching a high amount of threads each, usually in the thousands. This allows for high levels of parallelism, especially Single Instruction, Multiple Data (SIMD), i.e lots of threads perform the same task. This is very suitable for image filtering tasks such as the bilateral filter, as each pixel of the image is to be treated by the same algorithm. There are several different platforms and programming models for GPU programming, with Open Computing Language

(OpenCL) and Nvidia's Compute Unified Device Architecture (CUDA) being the most commonly used. This work uses CUDA to run the bilateral filter on an Nvidia GPU.

When using CUDA, the largest and slowest memory of the GPU, the main memory, is called global memory. Just converting the bilateral filter to run in parallel on the GPU while storing everything on the global memory will usually result in a hefty speedup on its own. Many parallel optimization techniques revolve around the use of other memories such as the shared memory which is an explicitly handled cache. Other techniques involve altering the pattern that threads are launched in and the workload for each thread.

The bilateral filter has been parallelized on GPUs many times before. This work aims to make a comprehensive case study on the common techniques used for running the parallel filter on a GPU, coupled with extensive experiments and results from running on the specific hardware available during this thesis. With improvements to global memory management, bandwidth and caching behavior of newer GPUs and versions of CUDA, it is of interest to determine which common techniques still provide a meaningful improvement that is worth the amount of effort put in by the programmer. When GPU technology evolves, older optimization techniques become less effective and new means for speedup may be required.

1.1 Research Questions

1. How can a fast bilateral filter be realized using modern massively parallel hardware?
2. What common methods of filter acceleration, such as use of shared memory and 1×N tiling, are effective on newer GPUs?

1.2 Method

This work will collect various methods for parallelization through extensive literature research. The techniques will then be applied and tested with a variety of different parameters and launch configurations. For this task, a program will be written in C++, using Visual Studio and CUDA.

The program that will be used for testing these various techniques will need to fulfill these requirements:

- Allow for consecutive execution of different functions that use different techniques.
- Allow easy display of all necessary timing data from the different functions.
- Allow for easy configuration of parameters and images.

The timing measurements of the various approaches are then presented as results.

1.3 Thesis Outline

The following chapter begins with a short background on digital image processing and image smoothing in particular. It then attempts to give a thorough and understandable explanation of the bilateral filter and the filters that led to its conception. Then the various uses of the bilateral filter are elaborated upon. Following this is a background and explanation of the parallel computing model and architecture used in this work. The related works section discusses both the works done in parallel image filtering and some fast sequential approximations of the bilateral filter.

The next chapter first goes over some initial design decisions related to the program in general before discussing the specific designs of the different parallel versions of the bilateral filter, and explaining the different techniques involved. The results chapter displays, compares and evaluates the various timing measurements of the different versions run with the chosen parameters and launch configurations on several images of different sizes.

The last chapter holds a discussion about the results achieved and conclusions to be made and answers to the two research questions. Lastly the future work section discusses what is left undone and how the work can be expanded.

2

Image smoothing

The field of digital image processing by computer dates back to as early as the 1950s [14], but really gained traction with the production of more powerful computers in the 1960s [16]. Early fields of study included encoding and approximation; filtering, restoration and enhancement, as well as pattern recognition and picture description [22]. All of these subjects persist to some degree, perhaps most of all the field of image filtering and enhancement, which includes image smoothing.

2.1 Simple image smoothing

There are several reasons as to why one would want to smooth an image. Apart from purely aesthetic purposes, smoothing can be an important initial step in image processing, as it can remove visual noise and distracting textures that would otherwise make it harder to extract relevant information from an image.

A digital image is made up of pixels that hold their own numerical values that determine the visual properties of the pixel. What these values represent is dependent on the color format used in the picture. Generally, a pixel in a grayscale image only holds one 8-bit value that determines its intensity. This value ranges between 0 and 255, where 0 is usually pure black and 255 is pure white.

When talking about image smoothing, one usually refers to replacing each pixel with an average, the mean value, of a surrounding area. Smoothing by averaging is a technique that dates back to the beginnings of digital image processing [14]. If every pixel in an image is replaced by a mean value of the 3 $\times$ 3 area around it, including the center pixel itself, then the operation can be explained mathematically as follows:

Let I be the input image, and O be the output image. Then I(x,y) is the intensity value of the pixel with the horizontal position x and the vertical position y, and O(x,y) is the corresponding pixel in the output image. What follows is that

$$O(x,y) = \frac{1}{9} \times (I(x-1,y-1) + I(x,y-1) + I(x+1,y-1) \\ +I(x-1,y) + I(x,y) + I(x+1,y) \\ +I(x-1,y+1) + I(x,y+1) + I(x+1,y+1)) \tag{2.1}$$

This can be written in the form of a double series:

$$O(x,y) = \frac{1}{9} \sum_{i=-1}^{1} \sum_{j=-1}^{1} I(x+i,y+j) \tag{2.2}$$

or for the general case(with varying sizes of areas):

$$O(x,y) = \frac{1}{n^2} \sum_{i=-n/2}^{n/2} \sum_{j=-n/2}^{n/2} I(x+i,y+j) \tag{2.3}$$

This is simply the mean value of a square matrix with sides of length n where n^2 is the total amount of entries in the matrix. The size of this area used to calculate the mean value has a distinct impact on the results. Figure 2.1 demonstrates this by applying this averaging on a test image with increasing neighborhood sizes. This is called a box filter, and is a very simple way of performing image smoothing. The area that is used for the calculation is referred to as a kernel, a convolution matrix or a mask. The image is convolving with a matrix of size n^2. In the case of the box filter every entry in the convolution matrix is 1(or more correctly 1 divided by the total number of entries):

$$\left(\begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix} /25\right) \textit{ or } \begin{pmatrix} 1/25 & 1/25 & 1/25 & 1/25 & 1/25 \\ 1/25 & 1/25 & 1/25 & 1/25 & 1/25 \\ 1/25 & 1/25 & 1/25 & 1/25 & 1/25 \\ 1/25 & 1/25 & 1/25 & 1/25 & 1/25 \\ 1/25 & 1/25 & 1/25 & 1/25 & 1/25 \end{pmatrix}$$

This means that during the calculation of the mean value, every pixel has the same relative impact; they are all weighted equally. The above matrix can be separated into two 1D matrices, or arrays:

$$\left(\begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix} /25\right) = \left(\begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} /5\right) * \left(\begin{pmatrix} 1 & 1 & 1 & 1 & 1 \end{pmatrix} /5\right)$$

This also means that the operation itself can be separated into two 1D operations. First convolving the image with one matrix(or array) in one direction, and then the other array in the other direction. This can save a lot of processing time.

As can be seen from Figure 2.1, increasing the size of the kernel means more blurring of the image. At lower sizes, this smooths the more noisy textures of the image, but it also blurs the sharp edges of the details in the image. Another way

Figure 2.1: A figure showing the effects of a box filter with increasing neighborhood size. (a)Top left is the original image, (b)top right shows the image after applying a boxfilter with a kernel size of 3×3. (c)Bottom left is the result of a 5×5 box kernel and (d) at the bottom right the image has been convolved with a 7×7 box kernel.

of using the box filter is to iteratively run it several times on the image, using the output of the previous run as the input to the next. Figure 2.2 demonstrates this procedure with two different kernel sizes and with several iterations.

The 5×5 filter seems to blur the image excessively in a few iterations, while the 3×3 filter has a more visually satisfying result, at least for this particular situation. Running a box filter iteratively like this results in a close approximation of another image smoothing filter called the Gaussian filter.

2.2 The Gaussian Filter

The Gaussian filter is similar to a box filter, but the entries in the convolution matrix are weighted according to a Gaussian function, also known as the normal distribution:

$$G(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{x^2}{2\sigma^2}} \tag{2.4}$$

Figure 2.2: The left column of images shows Figure 2.1a after applying a boxfilter with a 3×3 kernel once(top left), twice(middle left) and thrice(bottom left). The right column shows Figure 2.1a after applying a boxfilter with a 5×5 kernel once(top right), twice(middle right) and thrice(bottom right).

The function takes the form of a bell curve. Figure 2.3 is a Gaussian function where σ, the standard deviation, has been set to 3.

In the case of the Gaussian filter, the point of origin in the above graph represents the current middle pixel in the mask. The value of the weights decreases as the distance to the middle pixel increases. σ, the standard deviation, is responsible for how quickly the weights decrease with distance, i.e. the width of the bell curve. Since the impact of pixels at large distances gets attenuated,

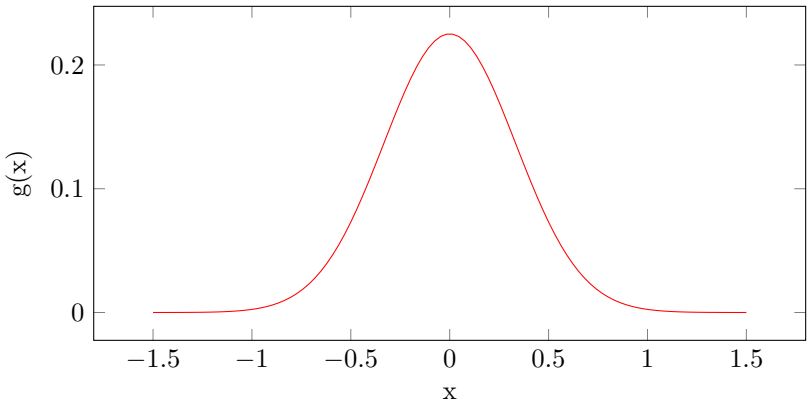

Figure 2.3: A Gaussian function with $\sigma = 3$.

the Gaussian filter is essentially a low-pass filter.

In a 2D image, the weights for the mask are calculated with the following Gaussian function, which is the product of two Gaussian functions:

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{2.5}$$

The 5×5 box filter kernel used in the previous section was comprised only of $1/25$. Using a Gaussian function to generate weights for a 5×5 Gaussian kernel with $\sigma = 1$, the resulting kernel looks like this:

$$\begin{pmatrix} 0.0029150 & 0.0130642 & 0.0215393 & 0.0130642 & 0.0029150 \\ 0.0130642 & 0.0585498 & 0.0965324 & 0.0585498 & 0.0130642 \\ 0.0215393 & 0.0965324 & 0.1591549 & 0.0965324 & 0.0215393 \\ 0.0130642 & 0.0585498 & 0.0965324 & 0.0585498 & 0.0130642 \\ 0.0029150 & 0.0130642 & 0.0215393 & 0.0130642 & 0.0029150 \end{pmatrix}$$

This could also be seen as a kernel with radius 2, which is a terminology that will be used a lot in this report. It is clear that the weights drop off fairly quickly when the distance from the middle point is increased. Like the box filter, the Gaussian filter is also separable into two 1D kernel operations. This is often used to increase the speed of the Gaussian filter. Running the Gaussian filter on the image in Figure 2.1a produces the image in Figure 2.4b.

The double series in equation 2.3 gave the general case for a box filter. The general case for a Gaussian filter can be written in the same notation, with O as the output and I as the input, as

$$O(x, y) = \frac{1}{k} \times I(x, y) \sum_{i=-n/2}^{n/2} \sum_{j=-n/2}^{n/2} G(i, j) \times I(x + i, y + j) \tag{2.6}$$

where G is the 2D Gaussian function in equation 2.5. Note that the normalization factor n^2 from equation 2.3 has been replaced by k, which is the normalization acquired by summation of the Gaussian kernel:

Figure 2.4: Figure 2.1a after applying a 5×5 Gaussian filter with $\sigma = 1$(top right), a 5×5 kernel with $\sigma = 2$ (bottom left) and a 7×7 kernel with $\sigma = 2$ (bottom right). Top left is the original image as reference.

$$k = \sum_{i=-n/2}^{n/2} \sum_{j=-n/2}^{n/2} G(i,j) \tag{2.7}$$

Note that in both the Gaussian filter and the box filter, normalization is achieved by dividing the result with the sum of the weights.

Increasing the value of σ or increasing the size of the kernel will cause more blurring, as can be seen in the examples in the bottom row of Figure 2.4. Increasing the kernel size to 7×7 while keeping σ at 1 did not have much impact as the added weights on the outer parts of the kernel were too small. The values for σ do not necessarily have to be integers, but have been chosen as such for simplicity in these examples.

While the noisy background textures are certainly suppressed, the results of the Gaussian filter are also blurry. The Gaussian filter is often called Gaussian blur and blurring may be the intended effect when using it. But for the purpose of reducing noise as to promote other details in the image, the Gaussian filter may be found lacking.

2.3 The Bilateral Filter

One problem with the Gaussian filter is that it smooths across edges, making the image blurry. This can result in information being obscured as well as the image being less visually appealing. The Bilateral filter [25] improves upon this aspect of the Gaussian filter, and other similar smoothing filters. In the Gaussian filter, pixels at a spatially larger distance weigh less than pixels at a closer distance. In Bilateral filtering, this same principle is also applied to distance in range, i.e. the numerical values in pixels. A pixel that is less similar to the center pixel will weigh less than a pixel that is more similar. The calculation of these range weights can also be done through a 1D Gaussian function. This requires the introduction of a second σ variable, to determine how much the impacts of pixels decrease with distance in range. From here on, σ_s will refer to the previous σ concerned with spatial difference, while σ_r will be the new standard deviation for the Gaussian concerned with range filtering.

Continuing with the notation used to write the series in equations 2.3 and 2.6, bilateral filtering can be written as

$$O(x,y) = \frac{1}{k(x,y)} \sum_{i=-n/2}^{n/2} \sum_{j=-n/2}^{n/2} G_s(i,j) \times G_r\big(I(x,y) - I(x+i,y+j)\big) \times I(x+i,y+j) \tag{2.8}$$

Here G_s is the regular spatial Gaussian, while G_r is a 1D Gaussian function with the color difference between the two pixels as input. In a grayscale image, this is simply the difference between the intensity values. Differences between colors may be more complicated to calculate, and this will be discussed in a later section.

Notice how the result of G_r will differ depending on which pixels are being compared. Thus there cannot be a pre-calculated kernel of weights for the range-Gaussian if the resolution of the pixel values is infinite, for example if the pixels are made up of floating point numbers between 0 and 1. If the resolution is made finite, it is possible to pre-calculate weights for every allowed color difference. This would be impractical for many applications and may require the color resolution to be lowered to avoid large overhead. Another consequence of the range-Gaussian being dynamic across the image is that the normalization factor k is not a constant, and will have to be calculated separately for each pixel according to:

$$k(x,y) = \sum_{i=-n/2}^{n/2} \sum_{j=-n/2}^{n/2} G_s(i,j) \times G_r(I(x,y) - I(x+i,y+j)) \tag{2.9}$$

This also means that the bilateral filter is not separable, which removes an option for optimization. There are separable approximations of the bilateral filter that provide results that may be sufficient in most cases, and those implementations will be discussed in the related works section. The non-separability

of the bilateral filter makes it more interesting for this work, as it makes parallelization for efficiency more necessary.

Figure 2.5 shows how the bilateral filter is able to smooth out surfaces while keeping important edges intact, using different parameters. Some extreme parameters will cause images to seem cartoon-like, because the surfaces of the image are heavily smoothened while the edges are kept intact. This is demonstrated in Figure 2.6.

Figure 2.5: Figure 2.1a after applying a 5×5 bilateral filter with $\sigma_s = 3$ and $\sigma_r = 20$(top right), after applying a 7×7 filter with $\sigma_s = 3$ and $\sigma_r = 40$(bottom left), after applying an 11×11 filter with $\sigma_s = 5$ and $\sigma_r = 60$(bottom right) with the original image as reference (top left).

2.3.1 Measuring color differences and choice of color space

To implement bilateral filtering, there needs to be an appropriate way to measure and quantify color difference. It is possible to calculate a 3D euclidean distance between two colors in the RGB color space, but this approach has some issues. The human eye perceives differences in certain color bands more clearly than in others, and so the range measurements would not be consistent with the differences recognized by a human observer. The CIE-Lab color space has been developed to be in tune with human color perception, and was recommended by the creators of the initial bilateral filter [25]. Instead of the R, G and B channels, the CIE-Lab color space consists of the L*(light), a*(which ranges from green to red) and b*(which ranges from blue to yellow) channels. The 3D euclidean

distance between Lab-values has good correlation to color differences observed by the human eye and is thus an efficient approach to the range measurement in bilateral filtering. It is also possible to simply perform the filtering on each color channel separately, removing the need to convert to a different color format.

2.3.2 Handling of the image border

When performing image convolution, a problem occurs when a pixel is close enough to the border of the image that parts of the kernel fall outside of the image. There are several solutions to this problem. One is to only include the pixels within the image border when performing the operation, effectively reducing the kernel size along the edges. Another solution is to take a step in the opposite direction when reaching the border, effectively mirroring the edges. Both of these approaches may be sufficient in a sequential implementation, but when working in parallel, it is optimal if all threads perform the same operation, decreasing the amount of thread divergence.

A better solution in cases where it is important that all pixels are filtered is to extend the image with an artificial border, performing the filtering within the initial region of the image, then removing the border again. OpenCV [5] has a function named copyMakeBorder that can perform this operation, with different options for what the border should consist of. In other cases, it may be simplest to just ignore the outer pixels and leave an unprocessed border in the output image.

2.3.3 Common usage of the bilateral filter

The main application for the bilateral filter is denoising, but it has various other uses including texture and illumination separation, tone mapping, retinex, and tone management; data fusion; and 3D fairing [18]. New uses for the filter are still being discovered. For denoising, both Adobe Photoshop [1] and GIMP [4] use their own variations of the bilateral filter, called Surface Blur and Selective Gaussian Blur, respectively. The bilateral filter also has various denoising uses in the field of medicine.

Figure 2.6: An image bilaterally filtered with incrementing parameters making the image look increasingly cartoon-like. Top left shows the unaltered image. Top right shows the image after being convolved with a 5×5 filter with $\sigma_s = 3$ and $\sigma_r = 30$. Bottom left uses a 21×21 filter with $\sigma_s = 6$ and $\sigma_r = 30$. Bottom right uses a 31×31 filter with $\sigma_s = 9$ and $\sigma_r = 50$.

3

CUDA and Parallel Computing on the GPU

General-purpose computation on graphics processing units (GPGPU) concerns the programming of GPUs for general-purpose tasks that are traditionally handled by a Central processing unit(CPU). While GPUs are normally in charge of rendering graphics for computer screens, the architecture of GPUs also make them suitable for other heavily parallel tasks. A single GPU can run thousands of threads in parallel, and as long as the vast majority of them can perform largely the same instructions, a great speedup can be achieved when moving a suitable task from the CPU to the GPU. Systems that use a combination of CPU and GPU are called heterogeneous systems, and are becoming more common as the field of GPGPU is expanded.

CUDA is a parallel programming platform developed by Nvidia for use on their GPUs [7]. It allows C, C++ and Fortran programmers to utilize GPUs with simple language extensions. Since the same algorithm is applied independently to every pixel when performing bilateral filtering, it qualifies as an 'embarrassingly parallel' problem, and is thus greatly accelerated when thousands of threads handle one pixel each, or a small group of pixels, simultaneously.

Several versions of CUDA have been released, with the newest as of the time of writing being CUDA 7.5. CUDA-supported devices are classified according to their compute capability, which is a number of the form x.y which explains which features they have access to. The current highest compute capability is 5.3. The program associated with this work is written in C++ using CUDA 6.5 and run on a device with compute capability 3.0.

A GPU consists of several multi-core processors called SMs(Streaming Multiprocessors). SMs can launch threads in groups called blocks, and also have their own local memories which are shared between the threads in the blocks. There may be several blocks run on one SM but never one block occupying more than one SM. Even though several blocks on the same SM are utilizing the same physical memory, there is no sharing of specific data from those mem-

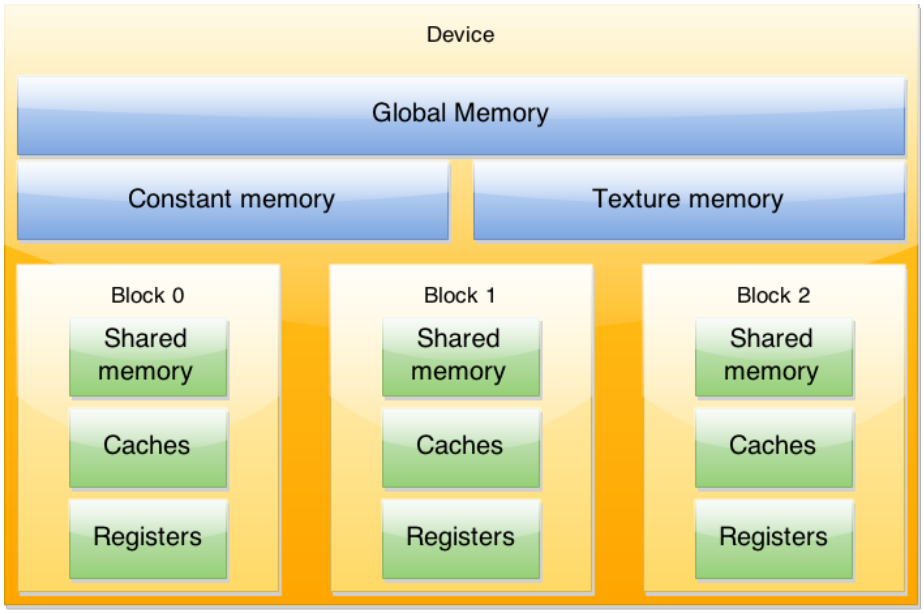

Figure 3.1: A simplified programmer's perspective on the memory hierarchy of CUDA

ories between blocks. Figure 3.1 shows a very simplified version of the memory hierarchy used by CUDA, from a programmer's perspective.

The global, texture/surface and constant memories are available to all active SMs and have higher latency than the SM-specific memories. Global memory is the largest memory on the GPU and is the memory accessed by the main memory-copy functions. Constant and texture memories are similar to global memory with some key differences that make them more fit for certain tasks, which will be elaborated upon in later subsections. Each SM has separate caches for each of these memories. Apart from the caches, the SMs also have shared memory, which is memory that is shared between threads in a block. They also have an amount of registers used to store local variables to each thread.

Thread blocks can have up to 3 dimensions, and the dimensionality is up to the programmer's discretion. The blocks are launched in a grid, which can also have up to three dimensions on newer devices, or up to two dimensions on older devices. For image processing tasks, a 2D grid of 2D blocks is usually the most intuitive, and sufficient. In this case, it would be beneficial for the entire grid of threads to have the same dimensions as the image to be processed, when possible, and every thread in the grid to be responsible for one pixel on the corresponding position of the image. Figure 3.2 shows this layout of a 2D grid of quadratic blocks superimposed over an image.

Threads are launched to run GPU-functions usually called kernels. Since the word kernel has been used in another sense for much of this report, they will

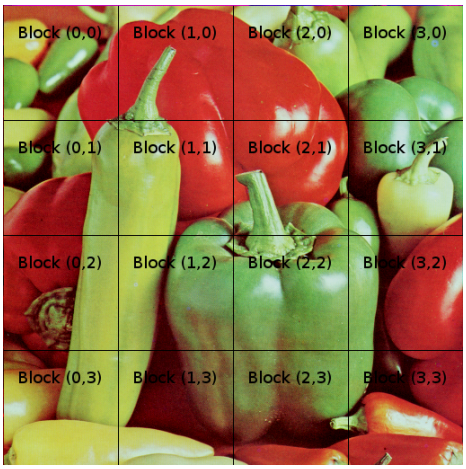

Figure 3.2: An image with a visual representation of a 2D grid of 2D thread blocks superimposed over it. Generally, one thread will filter one pixel in the image, but in practice the blocks would have to be smaller than they are in this simplified example.

be referred to as CUDA-kernels for differentiation. CUDA-kernels are launched with specific grid and block dimensions, and the actual distribution of blocks onto SMs is handled automatically by CUDA. The threads are then executed one instruction at a time in groups called warps. A warp is 32 threads, and the threads in a warp are executed in parallel, when they are all performing the same operation in a SIMD fashion. When the threads diverge, for example due to an if-else statement, the threads that execute the if-code will run separate from the threads executing the else-code. For this reason, it is beneficial to reduce thread divergence to an absolute minimum if one wants to run fast and efficient code. When diverging paths converge again, the threads in the warp resume to run in parallel. Adjusting the block size can have severe effects on the execution time of the kernel, and is therefore something to keep in mind when optimizing code.

Threads have access to their own thread indices for all dimensions of the block(threadIdx.x, threadIdx.y, threadIdx.z), as well as the indices of their block, for all dimensions of the grid(blockIdx.x, blockIdx.y, blockIdx.z). They also have access to the dimensions of the grid and blocks through functions blockDim and gridDim (with their respective .x, .y and .z endings). This is important as it allows the programmer to write general code for all threads but still have them work on different data, and branch according to that data or their indices.

After a CUDA-kernel call, there is an implicit synchronization of the device before following calls. Apart from this, all threads in a block can be synchronized through the simple barrier command __syncthreads(). This means that threads in the block will wait at the barrier call until all threads have reached this point

before moving on. If there is need for synchronization between blocks, it might be necessary to rewrite the CUDA-kernel into two separate kernels and utilize the implicit barrier between the calls.

An important aspect of GPU computing is SM occupancy, which is the amount of active warps per SM compared to the maximum supported amount. As a general rule of thumb, because the instructions are executed in a pipeline, a base amount of occupancy is necessary to hide the latency of the CUDA pipeline by keeping it occupied. Increased occupancy can cause further speedups but it depends on the application. Occupancy is governed by how large thread blocks are, how much shared memory is used by each block (as the amount of shared memory per SM is limited), and register usage.

The Kepler architecture used in this work has stronger but fewer SMs than older architectures like Fermi. This increases the requirement for parallelism per SM, which may add new restrictions to the ways parallel programs can be optimized.

3.0.4 Global Memory

Global memory is the main device memory in CUDA and it can be accessed by all active threads. It is cached both in L1 and L2 on some devices. On the device used for this work, it is only cached in L2. Global memory is manipulated from the host using the cudaMalloc(for memory allocation) and cudaMemcpy(for memory copying and transferral) functions, along with a few other variants. cudaMemcpy can also copy data back to the host, which is the common way to return results from a CUDA-kernel.

Global memory has much higher latency than the on-chip memories. Reads from global memory are done in segments of a certain size, depending on the amount of data requested by every thread of the current warp. It can read data in segments of 32, 64 or 128 bytes, meaning threads access data of 1, 2 or 4 bytes. If several threads attempt to read data that are within the same segment, only one request is necessary. Therefore, it is beneficial for global memory reads to be coalesced, to reduce the amount of reads and thereby lessen the effect of the lower bandwidth. For example, if all threads in a warp access one byte each, and all bytes are adjacent to each other in memory, it will only require one global read if the data is aligned so that it begins at the start of a 32-byte segment of global memory. If every byte of required data is instead n bytes away from the next, n memory request are required.

3.0.5 Constant Memory

Constant memory is a 64 KB device memory that can hold read-only data. It is written to from the host using the cudaMemcpyToSymbol function. Constant memory has the advantage that if all threads in a warp try to access the same address, only one request is generated, and the data is then broadcast to the other threads. After a constant memory access, the data is cached, and so succeeding warps will have even faster access to the data. Constant memory has

the downside of being slower when threads attempt to access different addresses, as this results in several device memory fetches.

3.0.6 Shared Memory

Shared memory is a user-managed on-chip cache, that has far lower latency than the larger device memories. For data to be stored in shared memory, it needs to be loaded from some other memory by the threads during runtime. This causes the need for different loading algorithms depending on the size and layout of the memory to be loaded.

Shared memory is stored in memory banks. On newer devices, like the device used in this work, there are 32 memory banks. When data is written to shared memory, it is divided consecutively into the banks in 4-byte words. The size of the banks can be increased to 8 bytes through a simple command. The writing is cyclical, so when banks 0-31 have been written to, it starts over at a different address of bank 0 and so on. If several threads in a warp try to access different data in the same bank, the accesses are serialized, resulting in what is called a shared memory bank conflict. These can cause severe slowdown and it is therefore best to try to avoid this situation when possible, for example by changing the access pattern of the threads or the pattern in which shared memory is written to.

During image filtering, with 2D blocks, the size of the shared memory array is given by:

$$sharedmemorysize = (blockwidth + radius * 2) * (blockheight + radius * 2) \quad (3.1)$$

Where radius refers to the size of the bilateral filtering kernel. If the block width is 32, there are no bank conflicts as threads in warps are all on the same line of pixels, and the data accessed in shared memory by a warp is stored in sequence. If a pixel is 4 bytes in size, and the radius of the kernel is 3 for example, then the size of each line in shared memory is $32 + 3 * 2 = 38$, writing to 38 words in shared memory. Figure 3.3 shows the access pattern of warps in this case, when each thread is accessing the first pixel in the kernel to be calculated.

When the block width is smaller, for example 16, a different situation occurs. The first 16 threads of the warp are accessing the first pixels to be used for the first row, but the second 16 threads are accessing the first pixels to be used for the second row. The shared memory array width is now(according to equation 3.1) 16 + 3*2 = 22 pixels, which takes up 22 words of shared memory. Figure 3.4 shows the access pattern for this case. It is clear that every warp suffers from bank conflicts, when different threads attempt to access different words from the same bank.

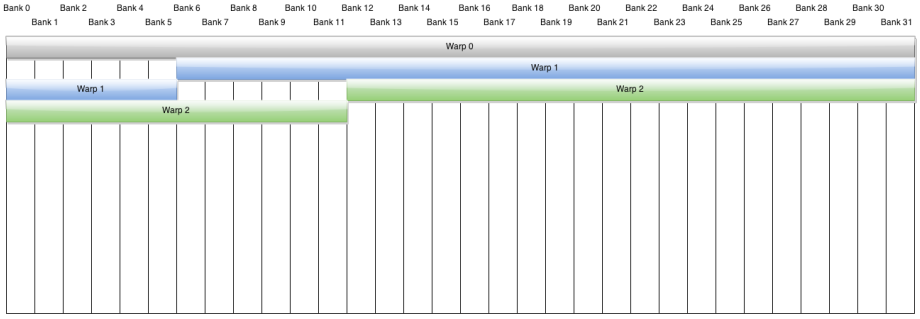

Figure 3.3: The access pattern of warps when the block width is 32 and the filter radius is 3. At no points do threads in the same warp try to access different words in the same memory bank.

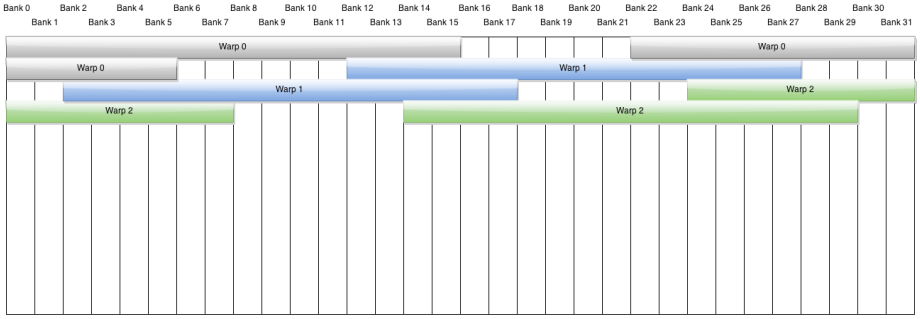

Figure 3.4: The access pattern of warps when the block width is 16 and the filter radius is 3. Memory bank conflicts occur for each warp as threads in the warp attempt to access different words from the same memory bank.

3.0.7 Texture Memory

Texture memory is a cached device memory that is generally used to hold textures and structures for traditional GPU-tasks. Texture memory works fastest when threads access data that are adjacent to each other. Textures can be 1, 2 or 3 dimensional, and the caching benefits when the accessed elements are adjacent in any of the directions. Texture usage in CUDA used to be handled through texture references, which are created at compile time and need to be declared globally. With CUDA 5.0 came the new texture objects for compute capability 3.0 and higher, which enable dynamic runtime creation of objects that can be passed as arguments like a regular pointer. This removes a lot of costly overhead of using textures and thus makes it a more appealing alternative.

4

Related Work

While Tomasi and Manduchi [25] where the first to give the bilateral filter its name in 1998, the idea behind it had been used as early as 1995 by Aurich and Weuler [10] in their non-linear Gaussian filter. They conceived it as a combination of the principles behind anisotropic diffusion [19] and robust filters like the Gaussian filter.

The bilateral filter has in later days seen much use in the medical fields. Dallai et al. used a real-time bilateral filter to remove speckle noise from medical ultrasound images [13]. Anantrasirichai et al. used an adaptive-weighted version of bilateral filtering as a step in their image enhancement method for retinal optical coherence tomography(OCT) images, which also includes the removal of speckle noise [9]. Yang et al. also used bilateral filtering as a noise removal step in segmentation of dense objects in computerized tomography(CT) scans [30]. Xu and Mueller showed that Bilateral filtering could be an alternative to other popular algorithms for regularization in iterative CT reconstruction [29][28].

Ryoo et al. extensively investigated the possible optimizations for GPU applications and showed how the optimization space for their example applications could be pruned by 98% by evaluating a set of metrics [23]. They also discuss $1 \times N$ tiling and loop unrolling, two techniques that are utilized in this work. Yang et al. created an optimizing compiler for GPGPU systems, which also includes, among other things, loop unrolling and $1 \times N$ tiling.

4.1 Fast Approximations of the Bilateral Filter

There are several close approximations to the bilateral filter that produce results closely resembling those of an actual bilateral filter but at a severely reduced processing time.

Barash [11] demonstrated the relationship between bilateral filtering and anisotropic diffusion [19]. This mathematical relationship was utilized in several speed-up attempts of the bilateral filter [15][17]

Durand and Dorsey, as part of an attempt to reduce the contrast of high-

dynamic range pictures while preserving their textures, proposed a "piecewise-linear bilateral filtering" [15]. By making linear approximations of the bilateral filter, segment-wise, they were able to convert the operation to the frequency domain using the fast fourier transform, and converting it back using its inverse. The convolution becomes simpler in the frequency domain, and coupled with downsampling, a speedup is achieved.

Later, Durand and Paris employed a signal processing approach to the bilateral filter, expressing the operation as a convolution, followed by two non-linearities, in the product space of the range domain and the spatial domain [17]. This combined with downsampling of the signal produces a speed-up.

In 2005, Pham and van Vliet presented a separable approximation of the bilateral filter [20]. On a 2D image, the filter basically runs in one dimension first and then in the other dimension on the results from the first pass. This produces an approximation that differs from a 2D kernel bilateral filter, and the results may differ depending on the order that the dimensions are passed. While the results of the separable approximation are shown to slightly differ from those of the regular bilateral filter, it nonetheless provides noise smoothing while preserving edges, and at increased speed.

Weiss [27] developed a median filter implementation that ran in logarithmic time, using partial histograms to reduce the number of operations. This approach could be used for the bilateral filter as well, with a box-kernel as the spatial filter and an arbitrary range-kernel function, which resulted in a bilateral filter that converges to logarithmic time.

With a few changes to Weiss' distributed histogram approach, Porikli [21] made the filter time-constant. He also proposed two additional ways to achieve constant time bilateral filtering; one using polynomial range filters which allows for arbitrary spatial kernels, and another using an approximation of a Gaussian range kernel through taylor series of polynomials and an arbitrary spatial kernel.

Chaudhury et. al. [12] sought to improve upon the precision of Porikli's approximation by using trigonometric range kernels instead of polynomial. The algorithm was also shown to be parallelizable.

Yang et. al [31] took a different route to obtaining constant time bilateral filtering, instead choosing to build upon the work of Durand and Dorsey [15].

While these methods may be sufficient for many situations, this paper is interested in parallelizing a non-separable non-approximative version of the bilateral filter.

4.2 Parallel Bilateral Filters

The work closest related to this one is probably the work by Van Werkhoven et al. published in 2014 [26]. While their work concerns convolution operations on a GPU in a more general sense and not the bilateral filter in particular, most of the techniques that are evaluated in this work are used in theirs as well. Those techniques are $1\times$ N tiling, shared memory padding and loop unrolling. Through rigorous testing, they created a huge lookup table to find the best combination

of parameters for specific situations, and introduced 'adaptive tiling' as a new optimization technique. The GPUs used in their work are the GTX680, GTX480 and Tesla K20, which are all older cards than what are used in this thesis. However, the Tesla K20 is made specifically for GPGPU.

The work of Kløjgaard Staal also has a lot in common with this work, as they both have the same general goal of parallelizing the bilateral filter using CUDA [24]. Using a naive version running on a CPU as reference, a speedup of 100 was achieved simply by changing the naive version to run on several threads on a GPU. By improving on the program through different optimization techniques, the final speedup was greater than a 1000 and increasing with higher filter radii. He was unable to efficiently utilize the shared memory, and instead stored the image on the GPU using texture memory while sending the precalculated Gaussian kernel as an extra parameter.

Xu and Mueller implemented a parallelized version of the bilateral filter on a GPU [29], choosing to pre-calculate the possible results for both the spatial and the photometric kernel and store them in texture memory. This reduces the resolution of the range calculation, and also adds a computationally expensive overhead. Later, Xu and Mueller, together with Zheng, elaborated on their previous work by using shared memory to hold the image data required for each block, and only pre-computing the spatial kernel, opting to calculate the range function continuously [33], as the results of doing so had not been evaluated earlier.

Agarwal et al. made changes to the bilateral filtering algorithm to make the kernel pair-symmetric, cutting the number of pair-neighbor calculations in half, at the expense of increased memory consumption [8]. The program used the shared memory and a tiling approach, and was heavily dependent on the automatic synchronization within warps, due to the possible race conditions that occur from the pair-symmetric algorithm. This approach is unsuitable for memory-bound applications, but may produce a speedup in others.

5

Algorithms and design choices

5.1 Initial decisions

Some trivial design choices of the bilateral filter need to be addressed. Decisions on library usage and representation of images as well as limitations in scope also need to be made:

5.1.1 Image format

This work uses OpenCV(Open Source Computer Vision), a well used open source framework for computer vision tasks [5]. OpenCV is used for loading images from file, writing images to files and displaying images on the screen during runtime, as well as performing other operations such as converting image formats and adding borders. During later stages of the work, when many of the advanced functions of OpenCV were no longer necessary, the work instead uses a simpler, customized library for loading and storing images.

When an image is read from the disk, it is stored in an array where each entry consists of either one 8-bit intensity(in the case of grayscale images), or three 8-bit intensities for red, green and blue(in color images).

5.1.2 Test images

The images used for testing were chosen for their sizes. As this work does not intend to improve on the functionality of the bilateral filter, the visual results are not of great interest.

The smallest image is a standard test image downloaded from the website of the University of Southern California's Signal and Image Processing Institute[6]. This image will be referred to as image 1 and has a resolution of 512×512.

Figure 5.1: The smallest test image, Image 1, with a resolution of 512 × 512.

The medium size image used is available on various websites that provide free high definition wallpaper. It will be referred to as image 2 and has a resolution of 1920 × 1080.

The biggest image was taken from ForestWander Nature Photography[2]. It will be referred to as image 3 and has a resolution of 5522 × 3651.

5.1.3 Algorithm-related choices and launch parameters

OpenCV has a function for converting the color space of images, called cvtColor. The function is capable of converting from on color space to another, for example from RGB to CIE-Lab, allowing for the filter to calculate a 3D euclidean distance between pixels. The final results of the filter when using this approach looked less visually appealing than simply performing the averaging on the R,G and

Figure 5.2: The medium size test image, Image 2, with an original resolution of 1920×1080, but presented in a smaller format here.

B channels separately, see Figure 5.4. Thus, separate filtering on the three different color-channels is the approach used in this work. This may be slightly more computationally expensive than using the other method.

The problem of handling the borders of the image were deemed uninteresting to the overall theme of this report. Therefore, this work ignores this problem by simply running the filter only on the pixels that have all their kernel neighbors inside of the image. This leaves a border of unprocessed pixels, the size of the radius of the kernel.

The σ-values were set to $\sigma_s = 3$ for the spatial Gaussian and $\sigma_r = 30$ for the range Gaussian. These values do not effect the runtime and were simply chosen because they produced visually pleasing results. The tests are done with odd filter radii ranging from 1 to 15, meaning kernels with sides of 3, 7, 11 and so on up to 31.

5.1.4 Timing and launch configuration

This work is limited to quadratic block sizes with sides of powers of two: 4, 8, 16 and 32. This is to narrow the scope of the report. Each CUDA-kernel with each combination of parameters and launch configurations is executed 5 times, and the measured time is the average of those 5 executions. The timing function used to time the CUDA-kernels uses CUDA events, while the function used to time the sequential version is the standard C++ clock function. The sequential version is only executed once for each parameter configuration, as it is very slow on larger images with larger radii.

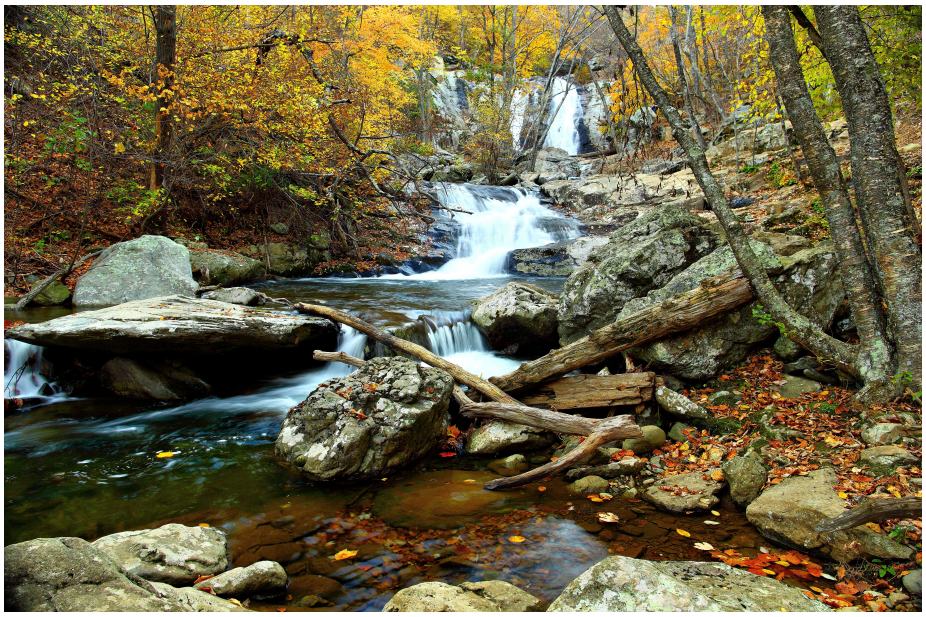

Figure 5.3: The largest test image, Image 3, with an original resolution of 5522 $\times$ 3651, but presented in a smaller format here.

5.2 Tested optimization steps

This work evaluates 8 different optimization steps for a parallelized bilateral filter using CUDA. These steps are tested through individual CUDA-kernels. First, a simple sequential version of the bilateral filter was written in C++ as a base example for comparison. The function iterates through each pixel of the code, only skipping the borders where parts of the kernel fall outside the image, and executes the same algorithm on each pixel. Since the spatial Gaussian kernel is only dependent on σ_s, the weights of this kernel are precalculated and stored in an array, while the weighting function for the range filtering is done dynamically for each pair of pixels. The output images of all the parallel versions are compared to the sequential output to make sure that they all produce the correct result. However, there is a slight difference in how the math is handled by the math libraries of the GPU and CPU, and therefore rounding differences may occur. Because of this, an intensity difference of 1 between the sequential and the GPU functions is considered acceptable. However, there should be no differences between GPU solutions.

5.2.1 Simple parallel versions

The sequential function was then used to implement the simple naive CUDA-kernel that served as another base example for comparison, and is considered

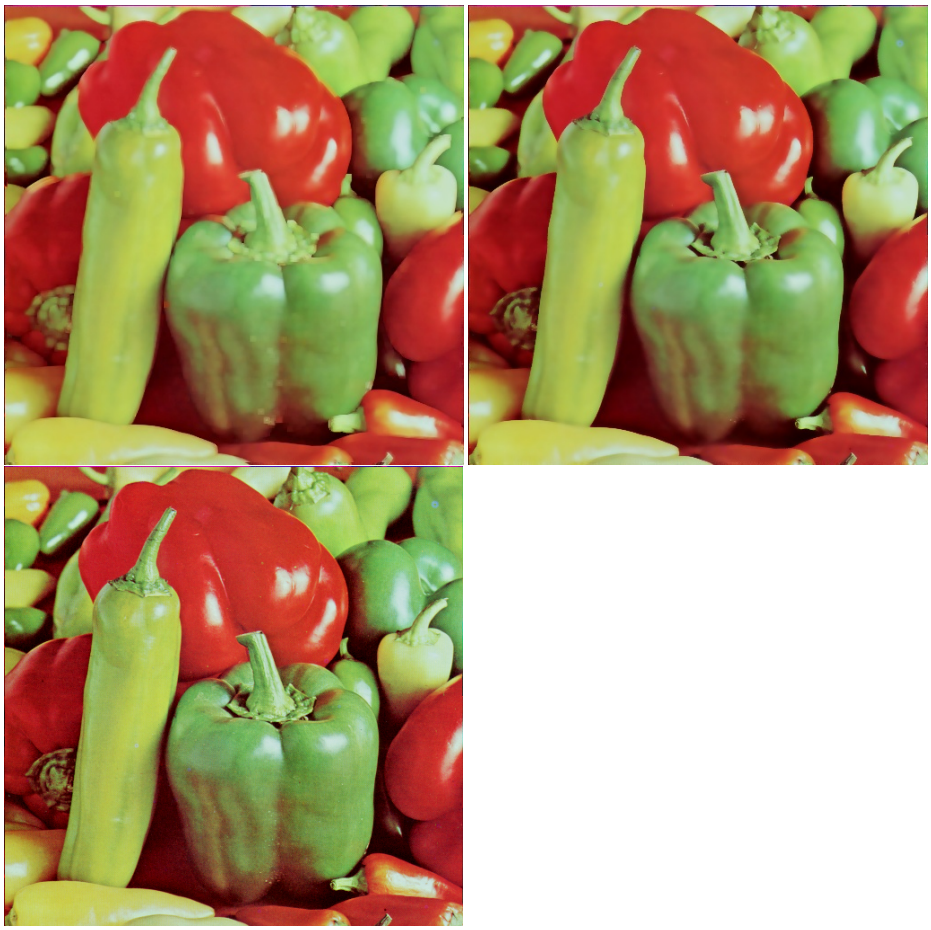

Figure 5.4: The original image(bottom) is bilaterally filtered using the euclidean difference between all color channels using OpenCV's function for converting to and from CIE-Lab(Top left), and channel-wise on an RGB image(top right). The latter result was deemed more appealing.

the first optimization step. In the naive parallel version, one thread is assigned to each pixel, and the input and output images are stored in global memory, as well as the spatial kernel. The range kernel function is converted to a device function that is called from the main CUDA-kernel.

The second optimization step involves the use of constant memory to store the spatial kernel. As all threads in a warp are accessing the same weight-value in the spatial kernel, constant memory is a good fit thanks to its broadcast functionality when several threads attempt to access the same data. The spatial kernel is kept in constant memory for all the following CUDA-kernels.

5.2.2 Use of shared memory

The third optimization step is storing the input image in shared memory. The image is first moved to global memory as in the previous versions, and the threads then cooperate in reading the pixels into a shared memory array. The pixels that are read into shared memory are only those that need to be accessed by the block. The size of the shared memory array was given by equation 3.1 (shown again below) and is simply the pixels that are to be filtered together with the neighboring border.

$$sharedmemorysize = (blockwidth + radius * 2) * (blockheight + radius * 2) \quad (5.1)$$

The algorithm used to load into shared memory is non-trivial, and is explained through pseudocode in Algorithm 1. This algorithm was inspired by the algorithm used by Van Werkhoven et al. [26]

Algorithm 1 The algorithm to load data into shared memory.

$tileidx \leftarrow threadIdx.x$, $tileidy \leftarrow threadIdx.y$
$tiledim \leftarrow blockDim.x + radius \times 2$
$xstart \leftarrow max(0, blockIdx.x \times blockDim.x)$
$ystart \leftarrow max(0, blockIdx.y \times blockDim.y)$
$xend \leftarrow min(columns, xstart + blockDim.x + 2 \times radius)$
$yend \leftarrow min(rows, ystart + blockDim.y + 2 \times radius)$

for $y \leftarrow ystart + threadIdx.y$ to $yend - 1$ with step $blockDim.y$ **do**
    for $x \leftarrow xstart + threadIdx.x$; to $xend - 1$ with step $blockDim.x$ **do**
        $shared[tileidx + tileidy \times tiledim] \leftarrow inputImage[x + y \times columns]$
        $tileidx \leftarrow tileidx + blockDim.x$
    end for
    $tileidy \leftarrow tileidy + blockDim.y$
    $tileidx \leftarrow threadIdx.x$
end for
synchronize()

Here, the word "tile" refers to the shared memory array. After the initial read into shared memory, this test version functions much the same as the previous one, except the pixel data is now accessed from shared memory rather than global memory.

The fourth optimization step is addressing the problem of bank conflicts. In the example of shared memory conflicts given in the background, and in Figure 3.4, pixels were said to occupy one 4-byte word of memory each, to make the explanation simpler. While they have so far been occupying only 3 bytes, the problem of shared memory conflicts remains. The first step to dealing with this was storing the pixels in shared memory as 4-byte words, to help implement the following measures. This means that 1 byte of every word is unused. The

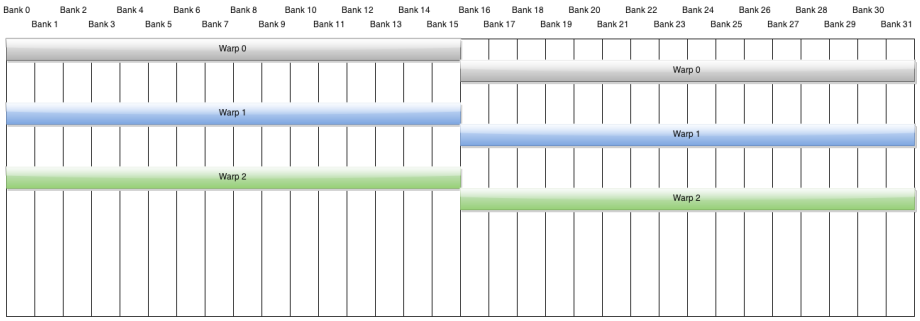

Figure 5.5: The access pattern of warps when the block width is 16 and the filter radius is 3, using padded memory. Memory bank conflicts are avoided, as all threads in a warp are accessing different memory banks.

second step is memory padding according to the approach presented by Van Werkhoven et al. [26], which extends the width of the shared memory array so that every line of threads within the same warp will always read from the next unused memory bank. Previously, the size of the shared memory array was given by 3.1, with the width of the array being:

$$sharedmemorywidth = blocksize.x + radius \times 2 \tag{5.2}$$

The padding approach instead sets the width to

$$sharedmemorywidth = \lceil \frac{radius \times 2}{32} \rceil \times 32 + blockwidth \tag{5.3}$$

In the example with a block width of 16 and a kernel radius of 3, this would result in a width of 48 for the shared memory array. The extra entries of the array would be unused, and as such this approach may use a lot more shared memory than the previous version. Figure 5.5 shows the new access patterns of threads in a warp, now without memory bank conflicts. Note that the figure would look the same for other filter radii as long as they are less than 17.

5.2.3 1 × N Tiling

The next 4 test versions use a technique called 1 x N tiling [26][23] or thread-block merging [32], which is the fifth optimization step. This technique extends the amount of work per thread by having blocks process extended image regions while keeping the block size the same. 1×2 tiling means that a block processes an area twice the size of the block. Figure 5.6 shows how the workload and the shared memory array is extended. The area that served as the rightmost border in the top figure, showing how the previous versions operate, is now reused with tiling when processing the second tile of pixels. The tiling process increases the workload for each thread and also heavily increases the amount of shared memory used by each block. This can result in fewer blocks per SM, as the

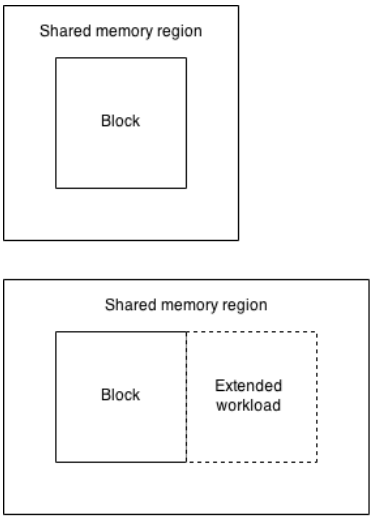

Figure 5.6: 1×2 tiling demonstrated. In the top image, the threads in the block process one thread each. In the bottom, the workload for each thread is doubled, and the shared memory array is extended.

amount of shared memory per SM is limited, which will in turn lead to reduced occupancy.

The fifth CUDA-kernel simply implements this without taking memory bank conflicts into consideration, while the sixth version also expands on the padding approach by setting the shared memory array widths to:

$$sharedmemorywidth = \lceil \frac{radius * 2 + blockwidth \times (tilingfactor - 1)}{32} \rceil \times 32 + blockwidth \tag{5.4}$$

where the tiling factor is N for $1 \times$ N tiling. This will result in a lot of shared memory usage for higher tiling factors and radii.

During execution, the blocks first execute their initial square area before collectively moving block width steps and doing the same operations again. The tiling factors tested were 1×2 through 1×6.

The seventh test uses a different kind of tiling. Instead of expanding the size of the shared memory area, this version periodically loads new data into shared memory after the current data has been used. This way, the same region of the image is calculated as with the other two tiling versions, but the shared memory region used is smaller. This allows for higher occupancy but may be costly due to more synchronization calls and possibly inefficient memory loads. Reusable data is reorganized in shared memory and is not reloaded from global memory. This version is based on version 4, which means that it uses memory padding to avoid bank conflicts. This version uses the same tiling factor as the other tiling functions.

Lastly, the eighth optimization method applies loop unrolling to version 6. Loop unrolling means that the compiler prints loops in their entirety, increasing the size of the code but reducing the amount of comparisons. Loop unrolling is known to have positive effects on parallel programs. To perform loop unrolling, the compiler needs to have full information on the variables used to determine how many loop iterations will occur, and therefore this approach makes the CUDA-kernel a template function, where the tiling factor is a template variable. This in turn means that the compiler creates a new CUDA-kernel for each tiling factor, further increasing the size of the code and the compile time.

6

Results

An extensive framework was written for simple evaluation of the optimization steps proposed in section 5.2. Originally, the user had a lot of control during runtime, by being able to select which image was to be filtered, decide the parameters including the two σ values, the kernel radius, the block size, and the number of executions for each CUDA-kernel. The user could also specify which particular optimizations were to be added to the current run. This was useful during earlier stages of experimentation, but later the program was written to be more automatic and comprehensive, having details be controlled by constant variables that could be set before compilation and execution. The program runs the sequential version and the 8 CUDA-kernels that involve the different evaluated optimizations, writing useful data and running times to text files and also writing basic timing data of interest to the program window during runtime. The timing data that was collected from these experiments will be presented in this chapter. First, some information about the hardware that was used is presented.

6.1 Hardware

The sequential function, CUDA-kernels and surrounding testing framework were executed on a PC with the following specifications:

- Intel(R) Core(TM) i5-4670 CPU @ 3.40GHz
- 8.00 GB of RAM
- Windows 8.1 Pro 64-bit
- NVIDIA GeForce GTX 760

The GeForce GTX 760 is a GPU of the Kepler architecture, with CUDA compute compatibility 3.0. Table 6.1 lists a small selection of relevant specifications of the functionality of compute compatibility 3.0, while table 6.2 lists some

specifications of the GPU itself. The Kepler architecture has 192 CUDA cores per SM, meaning the GTX 760 has a total of 6 SMs. Complete specifications of the GTX 760 can be found on the GeForce homepage [3].

Table 6.1: Selected technical specifications of CUDA compute capability 3.0

Maximum number of threads per block	1024
Maximum number of resident blocks per multiprocessor	16
Maximum number of resident warps per multiprocessor	64
Maximum number of resident threads per multiprocessor	2048
Number of 32-bit registers per multiprocessor	64 K
Maximum number of 32-bit registers per thread	63
Maximum amount of shared memory per multiprocessor	48 KB
Number of shared memory banks	32
Amount of local memory per thread	512 KB
Constant memory size	64 KB
Cache working set per multiprocessor for constant memory	8 KB

Table 6.2: Selected specifications on GeForce GTX 760

CUDA Cores	1152
Base Clock(MHz)	980
Memory Speed	6.0 Gbps
Standard Memory Config	2048 MB
Memory Bandwidth	192.2 GB/sec

6.2 Test Results

In the tables provided in this section, the tested CUDA-kernels are referred to by their version numbering, and an abbreviation that represents which optimization attempts were used in them. The versions are listed in table 6.3. More detailed information about the optimization steps is available in section 5.2.

Tables 6.4, 6.5 and 6.6 show which version provided the best results for every tested kernel radius on the 3 test images. Speedups are provided in comparison with the sequential version. Version 4, using padded shared memory is usually the fastest, with version 8, using loop unrolling, being slightly faster in a few instances. The speedups are very slight compared to the naive version, however. The biggest optimization step is clearly the initial step of simply running the filter in parallel on a GPU. Some of the attempted optimizations actually provided worse results than the naive parallel version. This can be seen more clearly in tables 6.7, 6.8 and 6.9, showing the results from all the individual CUDA-kernels on a selection of radii on the two smaller images. It can be seen that while version 2, using constant memory, is not the fastest in any of these

Table 6.3: Test versions

Version number	Explanation	Abbreviation
0	Sequential version	SEQ
1	Naive CUDA implementation	CUDA
2	Constant memory	CM
3	Shared memory	SM
4	Padded shared memory	PD
5	Tiling without padded memory	TL - PD
6	Tiling with padded memory	TL + PD
7	Alternative tiling with padded memory	TL2 + PD
8	Tiling with padded memory and loop unrolling	UL

tests, it is always a reliable improvement over the naive version, and the choice to keep the Gaussian kernel in constant memory for the later CUDA-kernels as well seems to have been correct.

Table 6.4: Best running times for a range of radii on image 1 with size 512 × 512

Radius	Sequential	Fastest	Time(seconds)	Speedup	Tile Size	Tiling factor
1	0.062	4 - PD	0.00180024	34.4398	16	N/A
3	0.344	4 - PD	0.00892978	38.5228	16	N/A
5	0.828	4 - PD	0.0214979	38.5153	16	N/A
7	1.516	4 - PD	0.0394262	38.4516	16	N/A
9	2.391	4 - PD	0.0601966	39.7198	16	N/A
11	3.438	4 - PD	0.0871645	39.4427	16	N/A
13	4.625	4 - PD	0.118996	38.8669	16	N/A
15	6	4 - PD	0.156193	38.414	16	N/A

Table 6.5: Best running times for a range of radii on image 2 with size 1920 × 1080

Radius	Sequential	Fastest	Time(seconds)	Speedup	Tile Size	Tiling factor
1	0.532	4 - PD	0.0140122	37.967	16	N/A
3	2.75	4 - PD	0.0707598	38.8638	16	N/A
5	6.75	4 - PD	0.170991	39.4758	16	N/A
7	12.392	4 - PD	0.314089	39.4538	16	N/A
9	19.704	4 - PD	0.496465	39.6886	16	N/A
11	28.643	4 - PD	0.724649	39.5267	16	N/A
13	39.19	4 - PD	0.984764	39.7964	16	N/A
15	51.285	4 - PD	1.29131	39.7155	16	N/A

Table 6.6: Best running times for a range of radii on image 3 with size 5522 × 3651

Radius	Sequential	Fastest	Time(seconds)	Speedup	Tile Size	Tiling factor
1	5.172	4 - PD	0.13292	38.9105	16	N/A
3	26.909	8 - UL	0.665448	40.4374	16	4
5	66.348	8 - UL	1.60866	41.2443	16	4
7	122.353	8 - UL	2.9652	41.2629	16	4
9	195.406	4 - PD	4.74923	41.1448	16	N/A
11	285.037	4 - PD	6.92436	41.1644	16	N/A
13	391.468	4 - PD	9.51281	41.1517	16	N/A
15	514.588	4 - PD	12.5168	41.1117	16	N/A

Of the block sizes examined, 16 provided the best results, and 4 is clearly the worst. This can be seen in figure 6.1. While only presented for the smallest image and radius, this is representative of the results on larger images and radii as well. When using tile size 16, bank conflicts occur when using shared memory, and therefore the padded memory approaches have a slight edge over the other shared memory attempts. On higher radii, when also using tiling, the extra shared memory required by using padded memory costs more than what is gained.

The only 1 × N tiling version that provides any speedup on any radii is the one that is also using loop unrolling. The differences in results between tiling factors for the 4 algorithms that use 1 × N tiling can be seen in Figure 6.2. The tiling factor that produced the best results in the tests listed here was 4. A tiling factor of 4 means that each block processes an area of 4 times the size. This was only true for the CUDA-kernel that uses loop unrolling though. It has execution time dips at tiling factors 4 and 6. The other versions seem to get worse with increased tiling factors. Especially version 7, which becomes significantly slower with larger tiling factors. Version 7 uses the other form of tiling that keeps the shared memory array a smaller size and makes several reads, which seems to be an ineffective approach when the amount of reads is increased.

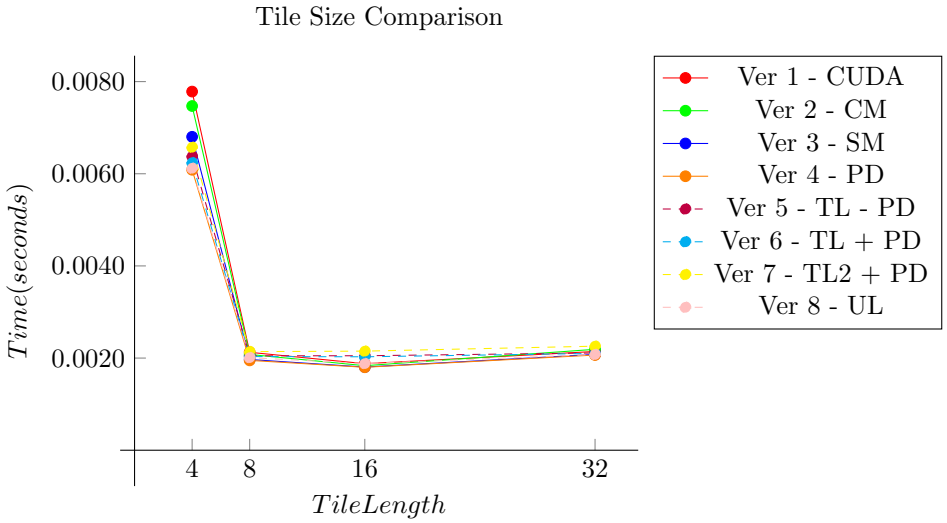

Figure 6.1: The individual algorithms' performance with different tile lengths on image 1. The radius is set to 1 and the last 4 versions use the tiling factor that produced the best result.

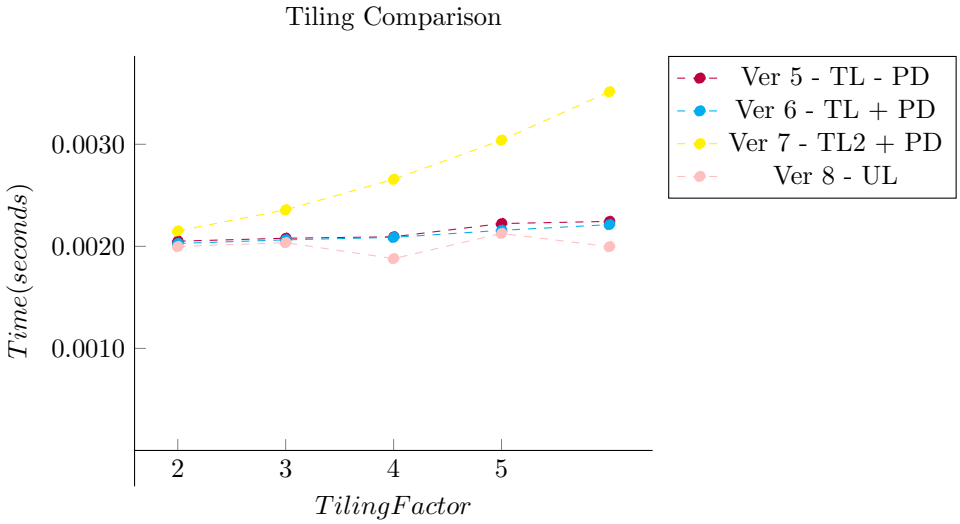

Figure 6.2: The effects on increasing the tiling factor of the three algorithms running 1 x N tiling on image 1. The example uses a radius of 1 and a tile length of 16.

Table 6.7: Best running times for each CUDA-kernel on image 1 with size 512 × 512, using radius 1, 7 and 15.

Version	Running time(seconds)	Speedup(sequential)	Speedup(Naive)
1 - CUDA	0.00187809	33.0122	1
2 - CM	0.00183448	33.7971	1.02378
3 - SM	0.0018051	34.3471	1.04044
4 - PD	0.00180024	34.4398	1.04324
5 - TL-PD	0.00204914	30.2566	0.916528
6 - TL+PD	0.00202799	30.5721	0.926084
7 - TL2+PD	0.00213972	28.9758	0.877729
8 - UL	0.00187982	32.9819	0.999081
Function	Running time(seconds)	Speedup(sequential)	Speedup(Naive)
1 - CUDA	0.0419822	36.1105	1
2 - CM	0.0410349	36.9442	1.02309
3 - SM	0.0397063	38.1803	1.05732
4 - PD	0.0394262	38.4516	1.06483
5 - TL-PD	0.0433843	34.9435	0.967681
6 - TL+PD	0.0438378	34.582	0.957671
7 - TL2+PD	0.0441124	34.3668	0.95171
8 - UL	0.0407951	37.1613	1.0291
Function	Running time(seconds)	Speedup(sequential)	Speedup(Naive)
1 - CUDA	0.165683	36.2138	1
2 - CM	0.161773	37.0891	1.02417
3 - SM	0.156998	38.2171	1.05532
4 - PD	0.156193	38.414	1.06076
5 - TL-PD	0.171034	35.0807	0.968711
6 - TL+PD	0.180457	33.2489	0.918128
7 - TL2+PD	0.17083	35.1227	0.969871
8 - UL	0.179943	33.3439	0.92075

Table 6.8: Best running times for each CUDA-kernel on image 2 with size size 1920×1080, using radius 1, 7 and 15.

Version	Running time(seconds)	Speedup(sequential)	Speedup(Naive)
1 - CUDA	0.0146601	36.2891	1
2 - CM	0.014283	37.2471	1.0264
3 - SM	0.014054	37.8539	1.04312
4 - PD	0.0140122	37.967	1.04624
5 - TL-PD	0.0157043	33.8761	0.933507
6 - TL+PD	0.0155314	34.2532	0.9439
7 - TL2+PD	0.0163254	32.5873	0.897992
8 - UL	0.0144374	36.8486	1.01542
Function	Running time(seconds)	Speedup(sequential)	Speedup(Naive)
1 - CUDA	0.334711	37.023	1
2 - CM	0.327136	37.8803	1.02316
3 - SM	0.31658	39.1433	1.05727
4 - PD	0.314089	39.4538	1.06566
5 - TL-PD	0.345841	35.8314	0.967817
6 - TL+PD	0.343644	36.0606	0.974006
7 - TL2+PD	0.345981	35.817	0.967427
8 - UL	0.314637	39.3851	1.0638
Function	Running time(seconds)	Speedup(sequential)	Speedup(Naive)
1 - CUDA	1.37305	37.3512	1
2 - CM	1.34032	38.2633	1.02442
3 - SM	1.30017	39.4449	1.05605
4 - PD	1.29131	39.7155	1.0633
5 - TL-PD	1.40947	36.386	0.974159
6 - TL+PD	1.42569	35.972	0.963076
7 - TL2+PD	1.40918	36.3934	0.974359
8 - UL	1.41922	36.136	0.967466

Table 6.9: Best running times for each CUDA-kernel on image 2 with size size 5522 × 3651, using radius 1, 7 and 15.

Version	Running time(seconds)	Speedup(sequential)	Speedup(Naive)
1 - CUDA	0.138826	37.2554	1
2 - CM	0.135381	38.2034	1.02545
3 - SM	0.133291	38.8022	1.04152
4 - PD	0.13292	38.9105	1.04443
5 - TL-PD	0.147222	35.1307	0.94297
6 - TL+PD	0.145597	35.5227	0.953491
7 - TL2+PD	0.153328	33.7316	0.905416
8 - UL	0.135977	38.0359	1.02095
Function	Running time(seconds)	Speedup(sequential)	Speedup(Naive)
1 - CUDA	3.16396	38.6708	1
2 - CM	3.08927	39.6058	1.02418
3 - SM	3.00274	40.7471	1.05369
4 - PD	2.98024	41.0548	1.06165
5 - TL-PD	3.22675	37.9183	0.980541
6 - TL+PD	3.21392	38.0697	0.984457
7 - TL2+PD	3.2293	37.8884	0.979769
8 - UL	2.9652	41.2629	1.06703
Function	Running time(seconds)	Speedup(sequential)	Speedup(Naive)
1 - CUDA	13.2725	38.7711	1
2 - CM	12.9592	39.7082	1.02417
3 - SM	12.6202	40.775	1.05168
4 - PD	12.5168	41.1117	1.06037
5 - TL-PD	13.5052	38.1028	0.982763
6 - TL+PD	13.6019	37.8322	0.975782
7 - TL2+PD	13.4706	38.2008	0.985289
8 - UL	13.5452	37.9905	0.979866

7

Discussion, Conclusions and Future Work

7.1 Discussion

All the optimizations yielded very little differences in running time on the whole, with times usually being within 10% of the naive parallel version. This seems to be partly because the operation is more compute bound than had been anticipated, meaning that computations are a significantly bigger time spender than memory fetches. Many of the optimization steps in this work focus on minimizing costly memory operations, since the bandwidth of the GPU is usually the biggest bottleneck in GPGPU applications, and this was assumed to be the case for the bilateral filter as well. Shared memory has much lower latency than global memory, yet usage of shared memory did not provide the substantial speedups that were expected. Using shared memory adds extra operations because the threads have to load the data from global memory into shared memory. This traditionally pays off when the threads later access the data several times from the faster memory, and definitely should do so in the case of the bilateral filter as well. The lack of significant speedup therefore suggests that memory transactions did not account for much of the total execution time. Shared memory usage can also have a negative impact on the occupancy, if each block requires a large enough amount of shared memory, this could also impact performance. The amount of blocks allowed per SM for a specific CUDA-kernel and set of parameters can be examined with the help of a simple CUDA function, and this only seems to have been a factor for the padded memory approach on higher radii when also using tiling. The caching behavior of the global memory is improved with newer GPUs, and therefore it is also possible that the use of shared memory will give lesser results when the global memory access pattern enables reasonable caching. The fact that the operation is compute bound was easily proven through a simple experiment:

Since the intensities are represented by single bytes per channel in this work,

the results of the range Gaussian kernel operation can be precalculated for all possible range differences and stored in an array. If this array is put in constant memory and the CUDA-kernels fetch their results from it instead of calculating them each time, the running time is decreased by about 250%. This is obviously a much greater improvement than any of the memory optimizations yielded. Since this is not a general solution, as it is reliant on the image format, it was not counted as an optimization to be tested fully in this work, but it shows how computational improvements yield much greater rewards. With this knowledge in mind, several other optimization techniques seem very interesting to try in the future, and will be discussed in the Future Works section.

1 × N tiling is not a memory optimization, and was expected to produce better results. One reason for this could be that the lower amount of blocks per SM is more detrimental on a newer GPU compared to older ones. For certain configurations of block size, radius and tiling factor, the tiling CUDA-kernels only allowed one block per SM. Kepler GPUs have fewer but stronger SMs than older architectures like Fermi, which means Kepler GPUs demand more parallelism per SM to become effective. Many of the works that have gained improvements from tiling have been using older GPUs [23][32]. Van Werkhoven et al. also remarked that tiling factors need to be kept low for high radius operations, because of the detriment to occupancy[26]. At lower radii, better results were still expected. Another problem with using tiling with bilateral filtering compared to using it with a spatial filter like the Gaussian filter is that the bilateral filter does so much work for each pair of pixels. For a tiling spatial filter that also uses loop unrolling, the extra code for each tiling factor can amount to as little as one extra line, and the work can be done very efficiently with only one reading from the weight kernel. This is not the case for the bilateral filter because of its non-linear nature. As such tiling might not be the best choice for bilateral filtering, but as tiling may not have been implemented optimally in this work, it is too early to tell.

Unrolling was only added to the version running 1 × N tiling and padded shared memory. This was believed to be the algorithm that would benefit the most from it, but it is likely that unrolling would benefit many of the other algorithms as well. The fact that the unrolling caused an improvement at bigger tiling factors seems to suggest that unrolling is of special interest to tiling algorithms, however. Originally, both the tiling factor and the radius were made into templates for this CUDA-kernel, but this was later deemed to costly in compilation times, as 5 tiling factors and 7 radii meant a total of 35 kernels being created by the compiler. This number was initially even higher as all radii between 1 and 15 were tested. Having them both as templates did enable more loop unrolling which did yield greater results however, but they were still rarely above a 10% improvement.

Texture memory was evaluated early on in the work, but eventually disregarded, as it did not seem to provide any benefit, and newer related works seemed to suggest that shared memory was generally superior. It is possible that texture memory could be utilized to greater effect in systems that represent images as two-dimensional arrays, because of its caching behaviour.

When comparing this work to the work of Van Werkhoven et al. [26] that uses many of the techniques used here, it is important to remember that their work only concerned convolution in the spatial domain, which is much less computationally expensive. Most related works also used older hardware, which may react differently to the techniques evaluated in this report.

7.2 Conclusions

To answer the first research question, a fast parallel bilateral filter can easily be realized by porting the operation to a GPU with the help of CUDA. In many situations, this may be enough of an improvement, and requires little work on the part of the programmer, apart from becoming familiar with CUDA. The speedup from moving the operation to a GPU ranged between 30 times to 110 times depending on which computer the program was run on. To further decrease the running time, depending on the hardware used, some of the techniques evaluated in this work may yield additional improvements.

To answer the second research question, using constant memory to hold the spatial Gaussian kernel will usually yield a small but reliable time decrease. Shared memory will often yield an improvement, especially for programs that are less computationally expensive than the bilateral filter. When shared memory is used in bilateral filters, using padded memory is beneficial for avoiding bank conflicts, if the block size causes bank conflicts. This is not always the case when using $1 \times N$ tiling, as the amount of extra shared memory used when padding can be detrimental to the occupancy.

The author of this report was unable to implement $1 \times N$ tiling in a sufficient way, and the results are therefore disappointing and hard to draw any real conclusions from. In the case of the last tested algorithm that also uses loop unrolling, a tiling factor of 4 was the most efficient. It is likely that other systems that substantially benefit from tiling would benefit from bigger tiling factors. It is interesting to note that the only tiling version that doesn't seem to do worse with larger radii is the 7th version, using the alternative form of tiling that does not increase the size of the shared memory array. This is likely due to being able to fit more blocks per SM. Loop unrolling should be useful in many parallel applications if the increase in code size is not a problem.

In general, this work indicates that while traditional memory optimizations still have merit on newer GPUs, they may need to be combined with computational optimizations that do not have big negative effects on the amount of parallelism per SM, especially when working with computationally expensive algorithms like the bilateral filter. Memory accesses are not the bottleneck they once were, and this needs to be taken into account when optimizing parallel programs on modern GPUs.

7.3 Future Work

Taking the lessons learned throughout this work into account, it seems the way forward is to attempt computational optimizations of the bilateral filter. This could mean making a functional implementation of tiling for example. For the specific case where the intensities are represented as bytes, the precalculations of the range kernel might be a realistic alternative. Many of the ideas proposed in section 4.1 would be interesting to try and further improve through parallelization. The pair-symmetric approach presented by Agarwal et al. [8], and discussed further in section 4.2, would also be interesting to try, seeing as memory usage does not seem to be a huge concern.

Perhaps changing the color space to CIE-Lab or some other appropriate color space, and using the 3D euclidean distance between pixels could yield a reduction in the number of computations, even though the visual results were deemed inferior in this work.

Bibliography

[1] Adobe Photoshop homepage. https://www.adobe.com/Photoshop. Accessed: 2015-10-28.

[2] Forest Wander Nature Photography. http://www.forestwander.com. Accessed: 2015-05-06.

[3] GeForce GTX 760 Specifications. http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-760/specifications/. Accessed: 2015-10-25.

[4] GIMP homepage. https://http://www.gimp.org/. Accessed: 2015-10-28.

[5] OpenCV. http://opencv.org/. Accessed: 2015-05-06.

[6] The USC-SIPI Image Database. http://sipi.usc.edu/database/. Accessed: 2015-05-06.

[7] What is CUDA? http://www.nvidia.com/object/cuda_home_new.html. Accessed: 2015-05-06.

[8] D. Agarwal, S. Wilf, A. Dhungel, and S.K. Prasad. Acceleration of Bilateral Filtering Algorithm for Manycore and Multicore Architectures. In *Parallel Processing (ICPP), 2012 41st International Conference on*, pages 78–87, Sept 2012.

[9] N. Anantrasirichai, L. Nicholson, J.E. Morgan, I. Erchova, and A. Achim. Adaptive-weighted Bilateral Filtering for Optical Coherence Tomography. In *Image Processing (ICIP), 2013 20th IEEE International Conference on*, pages 1110–1114, Sept 2013.

[10] Volker Aurich and Jörg Weule. Non-Linear Gaussian Filters Performing Edge Preserving Diffusion. In *Mustererkennung 1995, 17. DAGM-Symposium*, pages 538–545, London, UK, UK, 1995. Springer-Verlag.

[11] D. Barash. Fundamental Relationship between Bilateral Filtering, Adaptive Smoothing, and the Nonlinear Diffusion Equation. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 24(6):844–847, Jun 2002.

[12] K.N. Chaudhury, D. Sage, and M. Unser. Fast O(1) Bilateral Filtering Using Trigonometric Range Kernels. *Image Processing, IEEE Transactions on*, 20(12):3376–3382, Dec 2011.

[13] Alessandro Dallai and Stefano Ricci. Real-time bilateral filtering of ultrasound images through highly optimized dsp implementation. In *Education and Research Conference (EDERC), 2014 6th European Embedded Design in*, pages 278–281, Sept 2014.

[14] G. P. Dinneen. Programming Pattern Recognition. In *Proceedings of the March 1-3, 1955, Western Joint Computer Conference*, AFIPS '55 (Western), pages 94–100, New York, NY, USA, 1955. ACM.

[15] Frédo Durand and Julie Dorsey. Fast Bilateral Filtering for the Display of High-dynamic-range Images. In *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '02, pages 257–266, New York, NY, USA, 2002. ACM.

[16] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing (2nd Edition)*. Pearson, 2002.

[17] Sylvain Paris and Frédo Durand. A Fast Approximation of the Bilateral Filter Using a Signal Processing Approach. *Int. J. Comput. Vision*, 81(1):24–52, January 2009.

[18] Sylvain Paris, Pierre Kornprobst, and Jack Tumblin. *Bilateral Filtering: Theory and Applications*. Now Publishers Inc., Hanover, MA, USA, 2009.

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

[20] T.Q. Pham and L.J. van Vliet. Separable Bilateral Filtering for Fast Video Preprocessing. In *Multimedia and Expo, 2005. ICME 2005. IEEE International Conference on*, pages 4 pp.–, July 2005.

[21] F. Porikli. Constant Time O(1) Bilateral Filtering. In *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*, pages 1–8, June 2008.

[22] Azriel Rosenfeld. Picture Processing by Computer. *ACM Comput. Surv.*, 1(3):147–176, September 1969.

[23] 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 *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '08, pages 195–204, New York, NY, USA, 2008. ACM.

[24] Lasse Klojgaard Staal. Bilateral Filtering with CUDA. 2012.

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

[26] Ben Van Werkhoven, Jason Maassen, Henri E. Bal, and Frank J. Seinstra. Optimizing Convolution Operations on GPUs Using Adaptive Tiling. *Future Gener. Comput. Syst.*, 30:14–26, January 2014.

[27] Ben Weiss. Fast Median and Bilateral Filtering. In *ACM SIGGRAPH 2006 Papers*, SIGGRAPH '06, pages 519–526, New York, NY, USA, 2006. ACM.

[28] Wei Xu and K. Mueller. Evaluating Popular Non-linear Image Processing Filters for their Use in Regularized Iterative CT. In *Nuclear Science Symposium Conference Record (NSS/MIC), 2010 IEEE*, pages 2864–2865, Oct 2010.

[29] Wei Xu and Klaus Mueller. A Performance-driven Study of Regularization Methods for GPU-accelerated Iterative CT. *Workshop on High Performance Image Reconstruction (HPIR)*, 2009.

[30] Qiao Yang, A. Maier, N. Maass, and J. Hornegger. Edge-preserving Bilateral Filtering for Images Containing Dense Objects in CT. In *Nuclear Science Symposium and Medical Imaging Conference (NSS/MIC), 2013 IEEE*, pages 1–5, Oct 2013.

[31] Qingxiong Yang, Kar-Han Tan, and N. Ahuja. Real-time O(1) Bilateral Filtering. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 557–564, June 2009.

[32] Yi Yang, Ping Xiang, Jingfei Kong, and Huiyang Zhou. A GPGPU Compiler for Memory Optimization and Parallelism Management. *SIGPLAN Not.*, 45(6):86–97, June 2010.

[33] Ziyi Zheng, Wei Xu, and Klaus Mueller. Performance Tuning for CUDA-accelerated Neighborhood Denoising Filters. *Proceedings of the 3rd Workshop on High Performance Image Reconstruction (held with Fully 3D 2011), Potsdam, Germany (July, 2011)*, pages 52–55, jan 2011.