Image Convolution Methods on the GPU

CSE262

June 6, 2011

Julian Bui

Introduction

The advent of the general purpose graphical processing unit (GPGPU) has led to the notion of greatly enhancing a program's performance. It does not necessarily, however, lead to fantastical improvements that make it the end-all of performance hardware. GPGPUs typically work for applications with high data parallelism, low inter-worker communication, and plenty of computation instructions. One such application that may work well on a GPU may be image convolution – a "common image processing technique that changes the intensities of a pixel to reflect the intensities of neighboring pixels." Common uses for it include image sharpening, blurring, and edge detection. Due to the amount of data parallelism of imagery and number of floating point operations, promising performance gains may be expected. The main goals of this research are to investigate alternative methods of image convolution on the GPU as well as to determine what kind of performance gains image convolution may achieve through the use of the GPU.

Many image processing and computer vision techniques rely on kernel convolution. Some examples include edge detection, line detection, blurring, and sharpening. Computer vision and autonomous vehicles, vehicles that drive themselves without the aid of a human, may benefit immensely from faster processing. These applications use lane detection algorithms and image recognition to help the make sense of what their cameras see. To make split-second decisions which protect passengers and other drivers from danger, these algorithms must execute in quickly (in real time) and correctly. Graphical processing units have the ability to alleviate these problems.

Problems and Goals

Memory latency poses one of the major problems of typical CPU implementations of these algorithms. Hi-resolution imagery of large, floating-point data types may require hundreds of megabytes worth of memory space. A 10Kx10K pixel image at four bytes per pixel and three color bands would require over 1GB of memory space. CPU hardware does not have the immense amount of threads required to hide much of this latency and doesn't have the parallel memory interface to support it either. NVIDIA cards with compute capability 2.0 can maintain 1536 threads per SM and cards with compute capability 1.3 like the C1060 can maintain 1024 threads per SM (Programming guide 3.0). The high amount of threads allow computation to overlap the data fetching so much of this latency can be hidden, provided enough work exists for threads to do while waiting for data.

Memory access patterns for parallel computation also hinder fast execution. With multi-threaded applications, work partitioning is not quite as simple since workers would require the loading of ghost cells – data from neighboring regions that the worker is not responsible for. For example, if the algorithm distributes regions of the image to each worker and each worker needs to bring in **kernel_radius** number of pixels bordering its region in order to calculate the output pixels around the edges of its region. Since another worker will also require those same pixels, these pixels will be loaded in multiple times by multiple threads. These extra loads contribute to memory latency and overall execution time [5]. In order to optimize the memory access patterns, an algorithm must minimize the number of multiple pixel loads from memory. One way to do this involves increasing the partition size so fewer partitions exist and therefore the application loads fewer overall ghost cells. Another way to do this may involve different block geometries.

Computation hardware limitations also prevent fast execution on CPUs. Kernel convolution may involve floating point data, so in terms of processing units, GPUs like the Fermi have far more units to do computation. An AMD Phenom II x6 1090T CPU may have 3 floating point units per core and 6 cores, whereasa Fermi GPU will have 512 cores that each do floating point computation [1]. With fairly large kernel sizes like larger than 10x10, the number of floating point computations becomes prohibitively high for real time processing on the CPU. In this example, 100 floating point adds and 100 floating point multiplies need to occur for each output pixel. This quickly adds up. For example, a 10K x10K image with 3 color bands and a 15x15 kernel requires roughly 67 billion floating point operations per image. The CPU simply does not have enough floating point units to deal with this in real time.

General Kernel Convolution Implementation and Basics

The general kernel convolution implementation scheme involves the stencil method which takes the sum of products each pixel in the input array and the corresponding pixel in the kernel array when the kernel is centered directly over top. Figure 1 provides a clearer description.

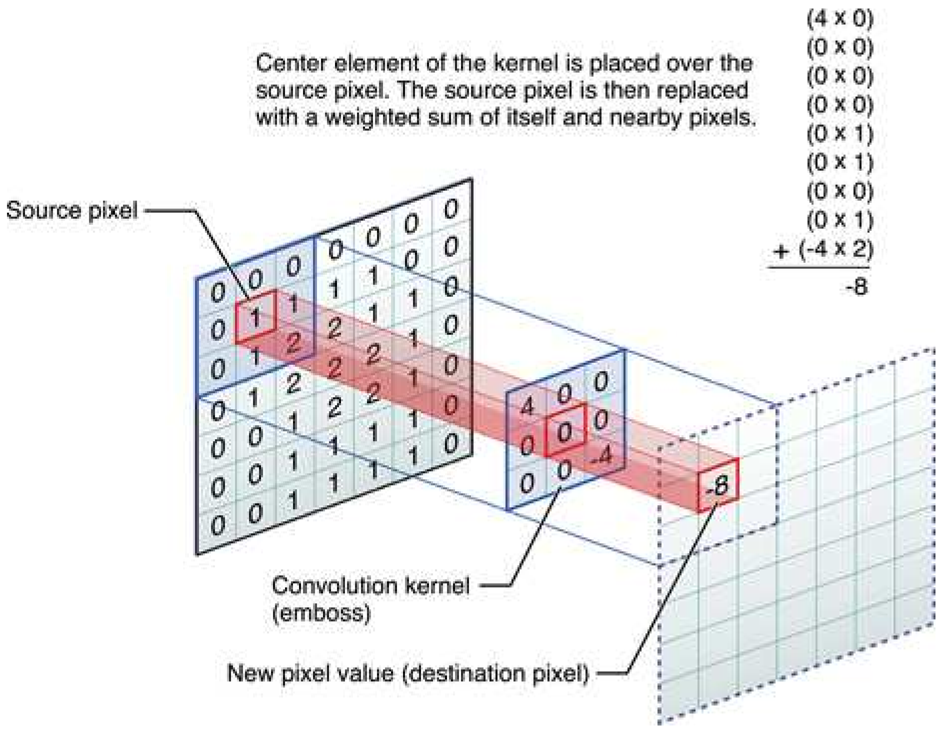
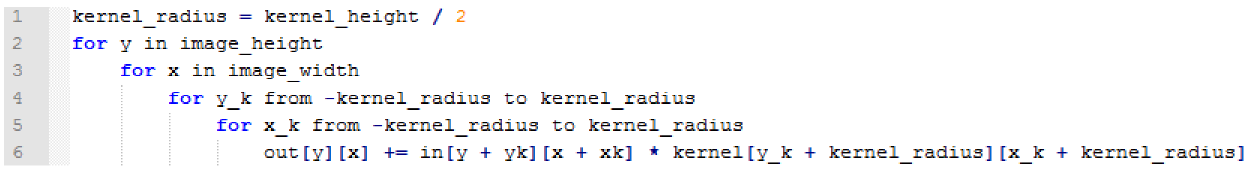

Center element of the kernel is placed over the source pixel. The source pixel is then replaced with a weighted sum of itself and nearby pixels.
(4 x 0)
(0 x 0)
(0 x 0)
(0 x 0)
(0 x 1)
(0 x 1)
(0 x 0)
(0 x 1)
+ (-4 x 2)
-8
Source pixel
Convolution kernel (emboss)
New pixel value (destination pixel)

Figure 1, Kernel Convolution, credits: [4]

One can picture kernel convolution as sliding the kernel over every pixel of the input image to generate every pixel of the output image. Note that during this process, each element of the kernel is multiplied by each corresponding pixel on the input image to produce a single output. The single-threaded CPU implementation is provided in figure 2.



```
kernel_radius = kernel_height / 2
for y in image_height
    for x in image_width
        for y_k from -kernel_radius to kernel_radius
            for x_k from -kernel_radius to kernel_radius
                out[y][x] += in[y + yk][x + xk] * kernel[y_k + kernel_radius][x_k + kernel_radius]
```

Figure 2, CPU single-threaded pseudocode

The kernel may be padded to avoid any boundary conditions that arise when calculating output pixels near the edges of the image that have no neighboring pixels. For example, the upper-left most pixel in the image won't have pixels to the left and above it, so one cannot apply the kernel weights to pixels that do not exist. In this research, zero padding is done to avoid checks for these conditions. One can also use extrapolation to fill in these non-existent pixels or a mirroring method, but for this research the padding method is insignificant since it only serves to change the appearance of the output image but will not significantly affect the performance of the algorithm.

Scope of Investigation and Setup

Due to the number of variables possibly involved in this research, the kernels will be limited to over 7x7 in size and will not be separable kernels – kernels that can be represented by only a single column and row vector that allows kernel convolution to require fewer computations.

The GPU applications will be implemented in CUDA on NVIDIA GPUs – specifically the C1060 and the GTX 570.

Video Card	Compute Capability	Number of SMs	Number of total cores	Core clock (MHz)	Peak GFLOPs/s Single/Double	Peak Memory Bandwidth GB/s	Total Memory
Tesla C1060	1.3	30	240	602	933/78	102 GB/s	4096 MB
GTX 570	2.0	15	480	732	1405/702	152 GB/s	1280 MB

Table 1 – GPU information

Note in table 1 that the Peak GFLOPs/s rate sources are unknown and it is not even clear how these units are measured.

Baseline Implementation on CPU

Single Threaded Vs. OpenMP Generated Implementations

Baseline single- and multi-threaded CPU implementations were created to give a reasonable comparison for the implementations on CUDA. Since there was limited time to do this project and most of the time allotted would go towards the CUDA implementation, only reasonably tuned single- and multi-threaded GPU implementations would be created. Both implementations were compiled with the highest level optimization flags and the multi-threaded versions were created using OpenMP pragmas that long horizontal strips of the image amongst all available cores of the processor using dynamic scheduling. All these applications were compiled using the intel c++ compiler for linux version 12.0.2. These applications were run on the AMD Phenom II X6 1090T and the Xeon E5504. The 1090T is a six core processor running at 3.2 GHz. It has a L2 cache size of 6x512 KB and an L3 cache size of 6MB. The E5504 has 4 cores running at 2GHz each. It has a L2 cache size of 4x256KB and an L3 cache size of 4MB. Figure 3 shows the executions of square images with a square 15x15 kernel.

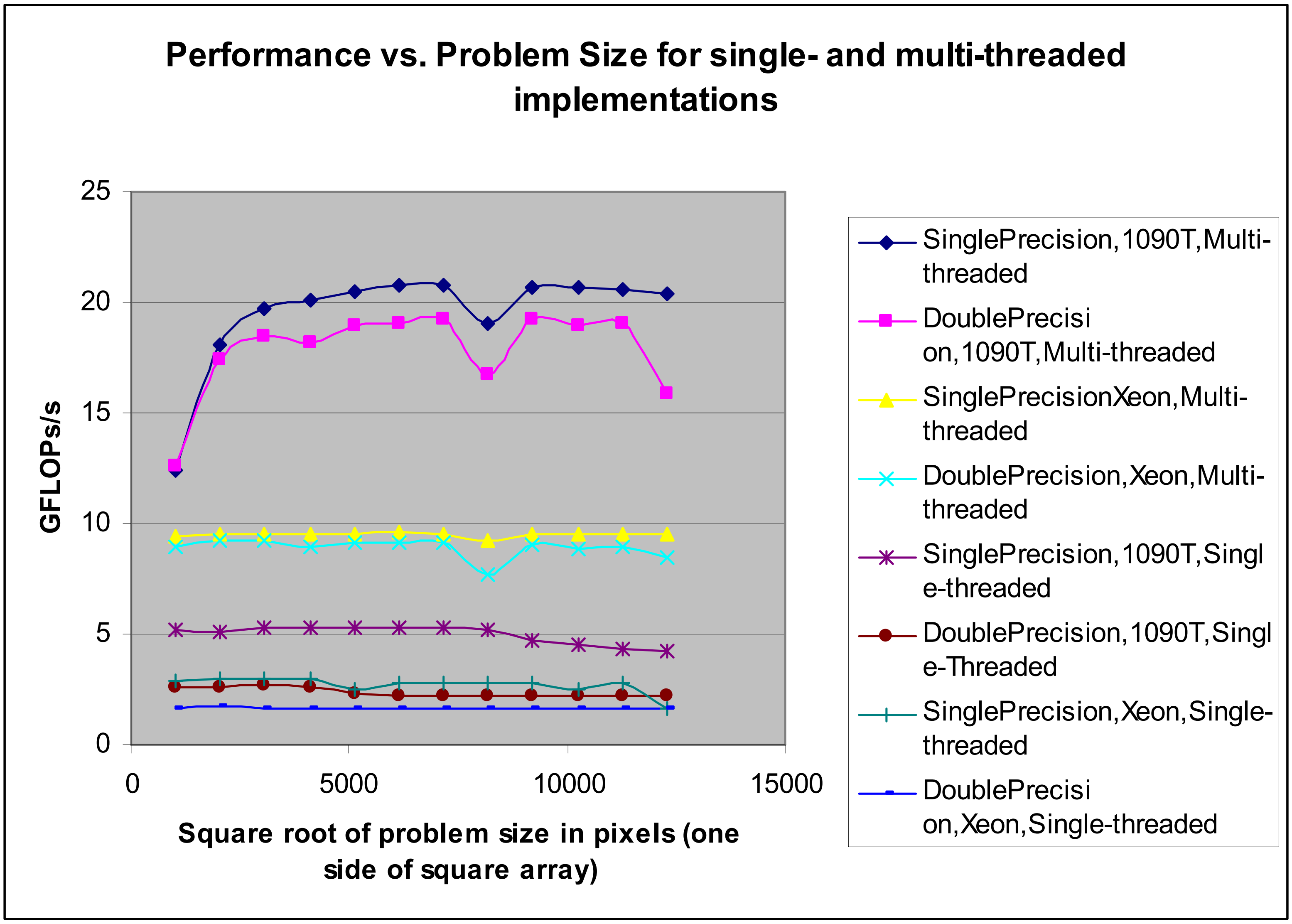


Figure 3 – Performance vs. Problem Size for single- and multi-threaded implementations using a 15x15 kernel

The average speedup for the 1090T executions between multi- and single- threaded applications was exactly 6.0, which is how many cores that 1090T has. This excludes the anomaly problem size of 1024x1024 which appears to have not ramped up to its full performance. The average speedup for the Xeon executions between multi- and single- threaded applications was 3.8 which approximately matches the four cores the Xeon has.

With the single core implementations running between ¼ of a second and 43 seconds for the 1024x1024 and the 12288x12288 sizes on Lilliput with double precision, the cost to do kernel convolution is too expensive both for real time applications in the smaller case and even for non-real time applications in the larger case.

The graph shows two dips in the 1090T's performance around 4196, 8192, and 12288KB for the multi-threaded implementations. These data points mark the multiples 4KB TLB cache size so most likely the processor is having trouble caching all the pages.

The multi-threaded versions of the code seem to exhibit super-linear speedup. The current hypothesis is that the addition of threads allowed the processor to do more prefetching while the threads were busy processing and that the extra threads allowed for a lot of memory latency hiding.

Parallel Implementation Process and Analysis

Since the ideal implementation for a parallel was not known in advance, the application was implemented incrementally with design decisions made along the way. This section describes the thought process that took place between implementation increments and describes the effects seen.

Naïve Implementation

The first incremental stage was to just set up the skeleton of the code and get a working CUDA implementation. The easiest thing to do would be to assign partitions of the image to thread blocks and have the block load in all its corresponding pixels as well as the ghost cells that make up the region needed for the kernel computation. This implementation used no shared memory, no aligned memory, and no loop unrolling, though the O3 optimization level was used.

Problem Size	Kernel Size	GPU Time (s)	GFLOPs/s	Precision	GPU
5120x5120	15x15	.90	5.5	Double	C1060
5120x5120	15x15	1.4	3.5	Single	C1060
5120x5120	15x15	1.3	37.4	Double	GTX570
5120x5120	15x15	1.5	34.2	Single	GTX570

Table 2 – Naïve implementation results

As seen in table 2, the GTX570 naïve gpu implementation already has a lower execution time and GFLOP rate than both the single and multi-threaded CPU implementations.

Interestingly, on both these machines the single precision versions of the code did worse than the double precision ones. It was suggested that this could be that it's because the way the warps load memory, a warp loading in all doubles could take advantage of 128 bytes load whereas a half-warp loading in all floats could not since 16*4 = 64. In a later increment, the implementation exploits this fact.

Reduction of Idle Threads and Using Shared Memory

Not utilizing the shared memory prohibited the naïve implementation from performing well. The shared memory exists to give close to register speed data loading by caching data from global memory. The main problem with the naïve implementation was the redundant loading from global memory. Each thread loads **kernel_size** elements from global memory and neighboring threads will also load these exact same elements from global memory. Having each thread block store data in shared memory reduces the amount of global memory accesses since threads can reuse data that neighboring threads have already loaded and stored in shared memory.

Also, if partitions of images are given to thread blocks for them to load and store into shared memory, only a fraction of those threads would actually end up calculating output pixels because many of the threads would only be used to read ghost pixels and wouldn't actually do any computation since a thread block would need to bring in extra data outside the pixel region they are responsible for creating output for. The threads that were used for loading ghost pixels remain idle while other threads in the block do computation.

To solve the idle thread problem, threads must pull in extra data for which they are not directly responsible. In this scheme, threads not only load in the pixel that they are responsible for, but also some of the threads will load in ghost pixels. This allows the thread block to bring in all the pixels plus ghost pixels while every single thread has work to do during the calculation phase of the CUDA method.

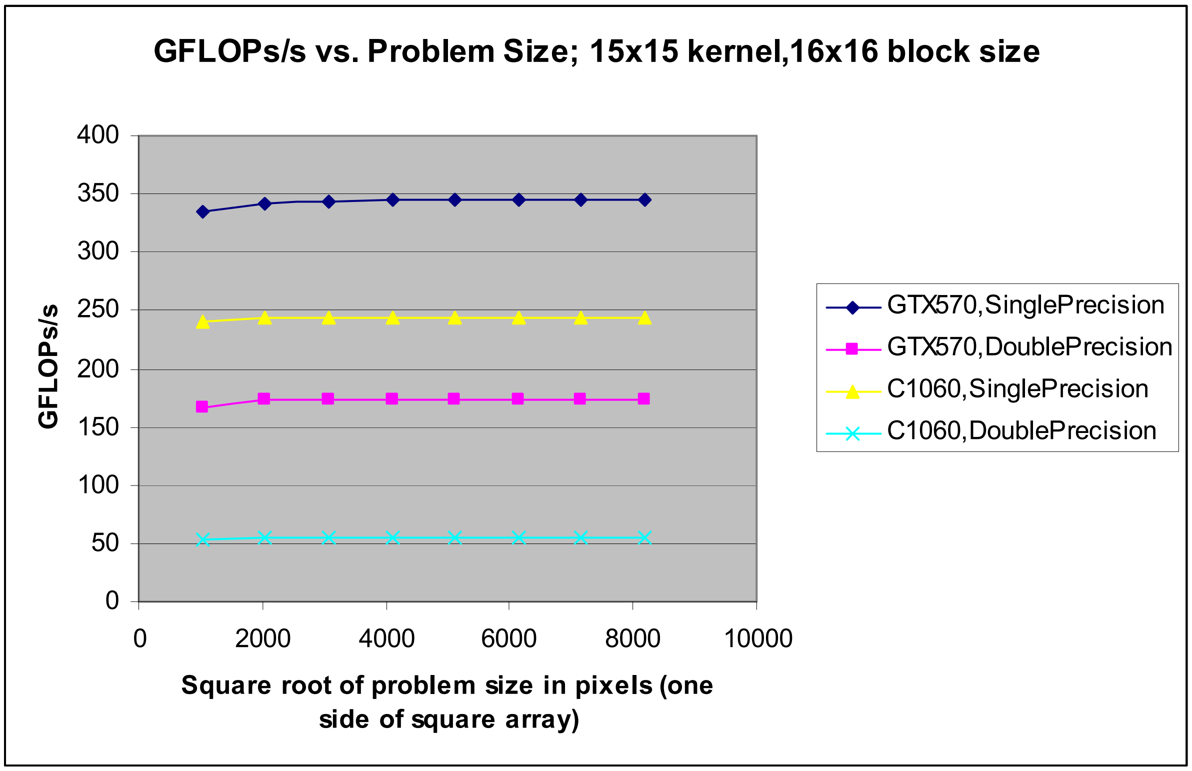


Figure 4 – Performance of GPU implementation using shared memory and no idle threads; 15x15 kernel and 16x16 thread blocks

Figure 4 shows that shared memory and reduction of idle threads leads to a drastic improvement over the naïve GPU implementation because more threads are utilized and data is explicitly cached. The Fermi single-precision code and Tesla double-precision code improved by roughly a factor of 10 over the naïve GPU code. The Fermi double-precision code improved roughly 5x over the naïve gpu code which is expected since it’s half the speedup of the single-precision code. The Tesla single-precision code, however, got the biggest performance improvement with roughly a 70x speedup over the naïve gpu code.

This implementation also uses __constant__ declared memory for the kernel array which allows the global memory accesses to be cached. The main changes between this iteration and the previous one are the changes in memory accesses. The dramatic difference in performance indicates that the naïve GPU implementation was still heavily bounded by memory.

The graph also indicates that the ramp up to steady-state performance is fairly quick with respect to problem size. After around 2048x2048, the problem size becomes large enough that performance doesn’t change.

Maximization of Single-Precision Float Bandwidth

Because the application requires so many loads from global memory, one must maximize memory bandwidth. To maximize the bandwidth during coalesced reads, the reads should be aligned on memory boundaries and the half-line should utilize as much as the cache line as possible – in the case of floats and doubles, the half-warp should attempt to load in a full 128 bytes of memory. Since the previous implementation read in one element at a time into shared memory, only half the bandwidth was satisfied because each thread in a 16 thread half-warp loads in a single-precision float, it only requests 16*4 = 64 bytes. Each scheduled half-warp should load in a full 128 bytes in order to maximize memory bandwidth. CUDA has defined a structure, float2, that is simply two floats. When a thread reads in a float2 object in one line of code, 8 bytes are requested from global memory, and a half-warp can reach its full potential.

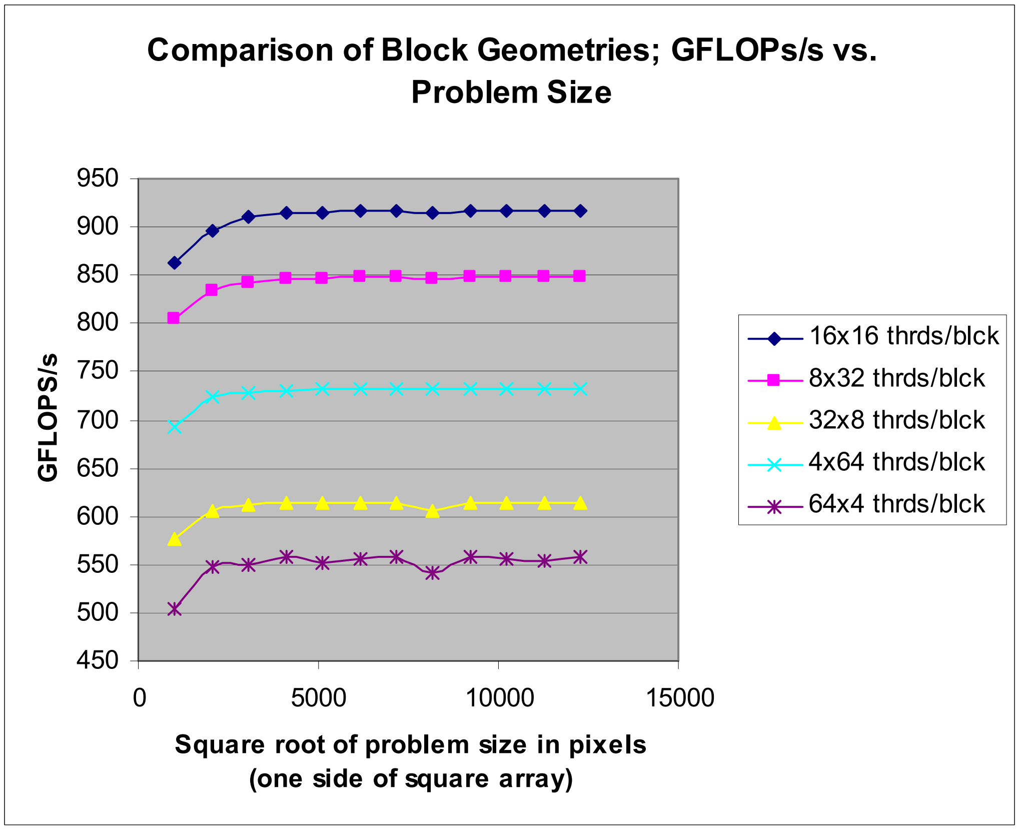


Figure 5 – Comparison of block geometries using a 15x15 kernel

Figure 5 demonstrates the single-precision computational power of the GTX 570. When the application accesses memory in an efficient manner, it can achieve almost a TFLOP per second. The maximum GFLOP rate of this implementation should double the rate of the previous implementation which only took advantage of half the bandwidth, however, the observed speedup was 2.6x. This extra 0.6 speedup may come from the change in indexing schemes in the newer implementation. This implementation's method of iterating over the image used fewer integer additions and multiplies because of modifications made to simplify the index calculation.

The graph also shows a preference for a square block geometry but this may be due to the square kernel and square image size. It has been suggested to try longer strips in order to reduce the number of partition columns and also the total number of ghost cells loaded from global memory. This tactic probably will not work on a machine with a fixed shared memory size because if the thread blocks are made longer, more row partitions will be required and the extra ghost cell loading will be shuffled but not eliminated. Still, depending on the kernel size and image size, this tactic may work in some situations.

With 256 threads per block and 12 registers used per thread and 1024 or 2048 bytes per block, the warp occupancy per SM was still maxed out. The shared memory usage could grow 8 times as large and remain within the 48KB limit of the shared memory on the Fermi machines [3].

Summary of Improvements

Technique	**Speedup vs. without the technique (on the GPU)**	**Notes**
Pitched memory to ensure that data begins at an aligned position in memory	~1.5x	Sequential access by threads before and after the technique was implemented
Shared Memory w/ elimination of idle threads	~16x	On a 15x15 kernel
Unrolling	~4x	Reduces control flow dependencies and exposes instruction level parallelism
__constant__ memory usage	~2x	Stores memory in global but cached read only memory; was previously using a const array for the kernel
Float2 usage	~2x	Doubles the memory bandwidth for single precision floats

Table 3 – Summary of Improvements

<u>Fast Fourier Transform Implementation</u>

The original plan for this project had not included a fast fourier transform (FFT) convolution implementation. When it was discovered that it could potentially convolve faster than the stencil based convolution, it became relevant. However, little time remained to implement FFT convolution on the GPU. Therefore, existing implementations were needed. ConvolutionFFT2D, a program that comes from the NVIDIA CUDA SDK and uses the CUFFT (CUDA FFT) library, does image convolution and has parameters within the application to specify the kernel and image size. This required modifying the source and compiling the program.

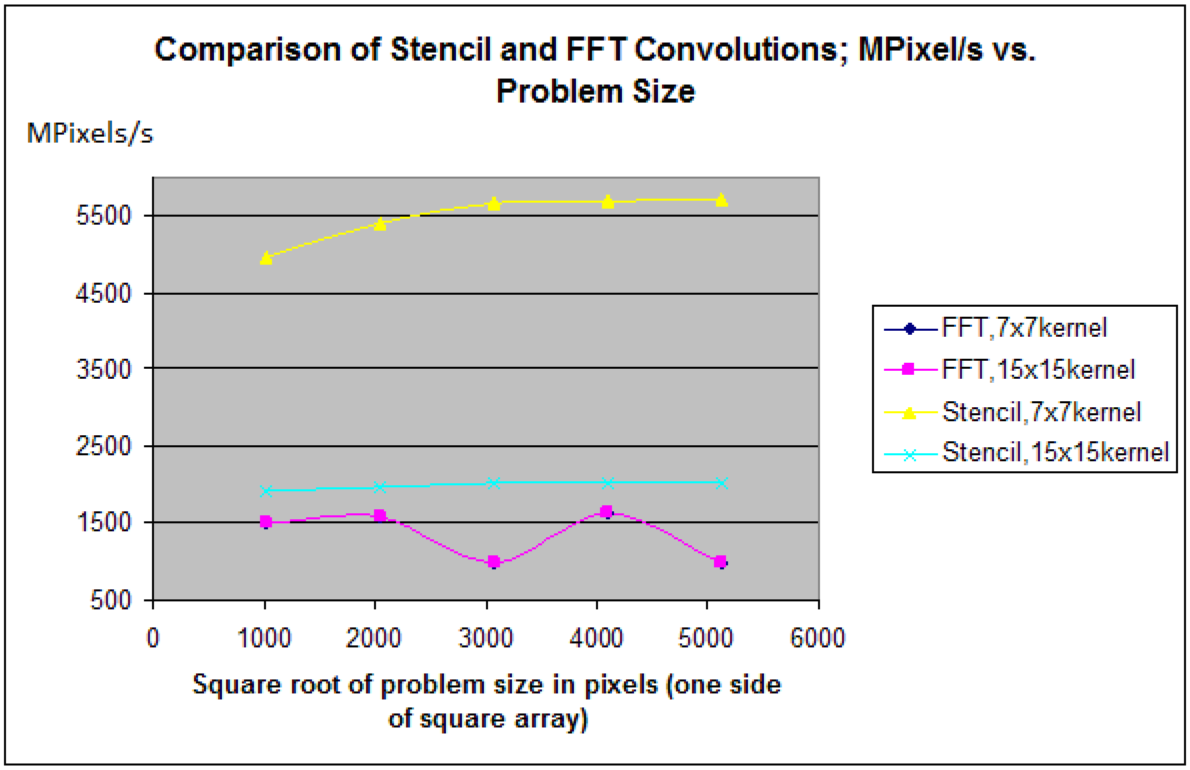


Figure 6 – CUFFT Implementation vs. Stencil Method on CUDA using a 16x16 thread block and single-precision data on the GTX570

Because the FFT algorithm has a O(nlogn) asymptotic runtime the FFT method performs roughly the same for similar kernel sizes as evident in figure 6 [2]. Notice that the two best fit lines for the FFT convolutions almost perfectly overlap. The graph suggests that when using GPUs, the stencil method will outperform the FFT convolution for 15x15 kernels and smaller. At some size bigger than a 15x15 kernel, one would prefer to use the FFT convolution. Though this doesn't show exactly what point one would prefer to use one method over the other, it is clear that for most convolution applications, since a 15x15 kernel is rather large, the stencil method works well enough.

The graph lacks data for larger kernel sizes because the ConvolutionFFT2D application did not work for certain inputs for unknown reasons.

<u>Criticisms and Future Work</u>

As stated earlier, the shared memory use per block was still rather low. Given more time, the implementation could be modified to user more shared data per block and still have the maximum warp occupancy. This would allow larger partitions and therefore fewer data points that have to be loaded multiple times due to ghost cell regions between workers [3].

The fact that ConvolutionFFT2D program did not work for certain inputs makes the usage of the program questionable. Understanding why it did not work for these inputs would be critical to understanding the relationship between CUFFT and the stencil method. ConvolutionFFT2D does, however, do its own checking against the stencil method, so we can still verify the results that were gathered. Also the lack of understanding of the CUFFT implementation makes the curved best-fit line of CUFFT graph difficult to analyze.

With more time the implementations should reduce the number of arithmetic – especially the multiplies – operations that the program uses to calculate indices. This reduces the floating-point instruction issue rate that goes through the processor.

Because the single and multi-threaded CPU codes were rather simple, much of the optimization work was left to the compiler since there wasn't much code to hand-optimize. It is possible that these compiler-optimized codes might not be as optimized as they could be.

<u>Conclusions</u>

The GPU implementations created in this study verified that the GPU could indeed produce excellent speedups over multi-threaded CPU implementations which shows that one should not depend solely on CPUs to perform image kernel convolution. The hardware comparison is not fair but the GTX570 GPU best single-precision performance does 44 times better than the 1090T 6core CPU single-precision performance. Also, for such problem with so much data, the memory access methods are incredibly important and play a big role in performance. The GPU implementations were able to do typical geospatial operations in very acceptable times and they were also able to do smaller problem sizes in real time for autonomous vehicle applications possibly. Finally, for these large kernel sizes in this study, the GPU stencil method outperforms the CUFFT convolution implementations. Any larger kernel sizes may need to start relying on CUFFT.

<u>Bibliography</u>

1. "A Comparison of the Opteron CPU vs G5 CPU and Their Dual-CPU Systems Architecture."*UNIXgods*. Web. 06 June 2011. <http://www.unixgods.org/~tilo/Opteron_vs_G5.html>.

2. "Explained: The Discrete Fourier Transform." *MIT*. Web. 06 June 2011. <http://web.mit.edu/newsoffice/2009/explained-fourier.html>.

3. NVIDIA. *NVIDIA CUDA Programming Guide 3.0*. *NVIDIA CUDA*. NVIDIA. Web. 6 June 2011. http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_CUDA_ProgrammingGuide.pdf.

4. "Performing Convolution Operations" *Apple Developer*. Web. 06 June 2011. <http://developer.apple.com/library/mac/#documentation/Performance/Conceptual/vImage/ConvolutionOperations/ConvolutionOperations.html>.

5. Podlozhnyu, Victor. "Image Convolution with CUDA." *Image Convolution with CUDA*. NVIDIA. Web. 6 June 2011. <http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_64_website/projects/convolutionSeparable/doc/convolutionSeparable.pdf>.