Efficient Detection of Sunspots with GPU Acceleration Through CUDA

Kenny May
Department of Computer Science
Hood College

Abstract - Tracking sunspots is not an easy task given that multiple sources of data are acquired using a variety of different instruments. With the sources of data and contributors to this repositories quickly growing, it is increasingly important to have an efficient solution to analyze the photographs to record trends and possibly make predictions. CUDA (Compute Unified Device Architecture) provides an excellent means of parallelizing the image processing stage and speeding up the analysis process. With more images being analyzed we will be able to make clearer, more accurate judgments regarding sunspots, solar flares, and space weather.

1 INTRODUCTION

Analyzing sunspots might not seem to be of importance, but it could help prevent or mitigate what could be a global catastrophe. Sunspots are caused by extremely intense magnetic fields that move, appear, and disappear on the Sun's surface. These areas of intense magnetic activity inhibit convection on the Sun's surface resulting in a cooler region that is about 3,000 to 4,500 Kelvin [1]. Since solar flares are also caused by the same magnetic fields, they can be directly related to sunspots. If we can detect a sunspot early enough we can attempt to predict when and how intense the next solar flare will be. The larger and denser a sunspot is, the larger and more dangerous the solar flare will be. A small solar flare can have annoying consequences such as interfering with GPS signals, but a large solar flare can have a more noticeable effect. It would be so noticeable that the global economy could be impacted, according to a study done by the *National Academy of Sciences* in 2009 [2].

In 1859, a large number of sunspots and solar flares were detected that created a solar storm headed for Earth which reached its target 18 hours later. This storm shocked telegraph workers and started several fires [3]. If the storm had occurred in today's electricity-dependant society the consequences would have been devastating. Everything that is dependent on electricity could be disrupted for months, including food, water, transportation, heating, cooling, and just about everything else in today's society. A month long disruption in power while infrastructure is repaired and replaced could have dire economic consequences. This is just one of the many reasons that the analysis of sunspots should be even more efficient and accurate.

After searching for a solution to this problem, it seemed natural to use a graphics card for image processing. There were many other current studies being conducted regarding image processing on a graphics device, but none that focused on the analysis of sunspots.

2 CUDA

The highly specialized architecture of a GPU (Graphics Processing Unit) has led to a recent development GPGPU concept or, General Purpose Graphics Processing Unit). GPGPUs are well suited for speeding up highly parallel computations such as: image processing, media compression, matrix multiplication, and other highly parallel problems. To aid developers in easily writing code for these devices NVIDIA developed a parallel computing architecture called CUDA. Current graphics cards from NVIDIA with the support of CUDA technology can support thousands of threads simultaneously to provide an enormous amount of information throughput. This is opposed to the 10's of threads that a powerful multi-core CPU can handle at once. This power allows the programmer to transfer the computationally intense portion of the problem to the GPGPU while the smaller operations can be left on the CPU. This is easily adapted to image processing because this kind of computation is easily parallelized. An image can be copied to the GPGPU, and every pixel in the image can then be processed at the same time instead of analyzing thousands of pixels in serial with the CPU. This type of processing is a well suited fit for the sunspot problem. An example of how processing is completed on the GPGPU can be seen in the following figure.

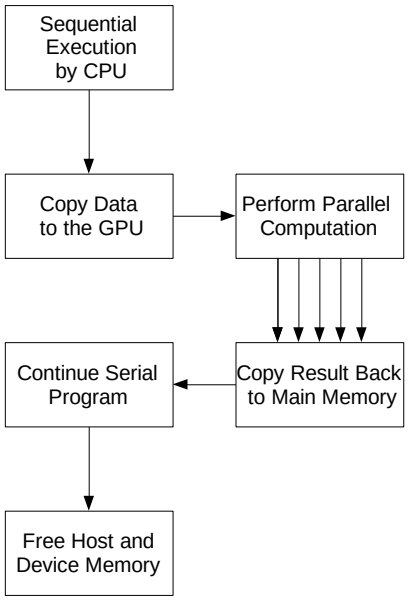

Figure 1. How computation is completed using a GPU device.

GPU computation works best when the computation to memory access ratio is very high. Global memory accesses on the device are expensive and could even possibly slow down a program's execution more than if it had just been done by the CPU. Shared memory can also be used on the device to cut down on access to its main memory. The shared memory on the device is extremely fast, but cannot be accessed by every thread. Shared memory can only be shared between threads in a thread block. A thread block is a way of logically organizing a group of threads (horizontal green row of blocks pictured below).

Green - ALUs | Light Orange - Control | Small Orange - Cache

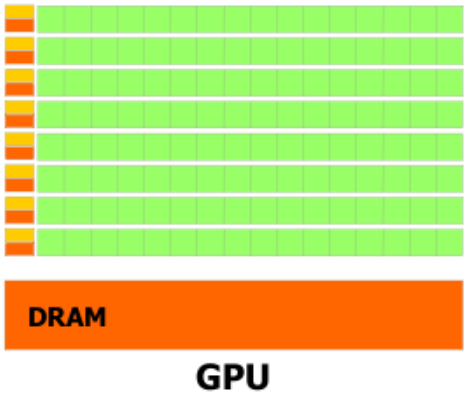

Figure 2. Diagram of Memory/Processor layout of GPU [4]

3 IMAGE PROCESSING

One image processing technique that lends itself well to parallel computing is thresholding. Thresholding involves taking a set range of colors and flagging them in a picture. Any pixels found inside of a specific color range are set to one color and any pixels found outside that range are set to another color. That way you are only left with areas of interest in your image. The process begins by reading a bitmap or jpeg into raw data; this data is in the form of a XY coordinate and a numeric number representing the color. This was realized with use of OpenCV. OpenCV (Open Source Computer Vision) is a library of programming functions for real time computer applications. Originally designed as a webcam interface, OpenvCV quickly grew into a comprehensive library of optimized graphics functions. OpenCV provided a fast and easy way to read an image file into a useable format that could be operated on at the pixel level. The GPGPU can fire up thousands of threads to analyze the extremely large dataset of coordinates and colors. This process would involve categorizing points in the data set by the use of thresholding.With thresholding we can then begin to pinpoint where the sunspots appear on the image.

1	7	6	3	2	4	7	7	0	0
2	5	2	1	3	3	6	6	3	7
5	2	8	8	1	2	8	8	7	6
3	2	8	9	2	3	8	8	1	4
4	1	2	3	7	4	3	1	2	2
4	1	8	4	7	4	5	9	2	6
2	5	3	8	9	8	8	3	1	2
3	4	5	6	7	0	5	3	3	3

Figure 3. Image data before thresholding

0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	1	1	0	0	1	1	0	0
0	0	1	1	0	0	1	1	0	0
0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	1	0	0
0	0	0	1	1	1	1	0	0	0
0	0	0	0	0	0	0	0	0	0

Figure 4. Image data after thresholding

The above tables represent an image in its raw data form. Since the images dealt with in this study were loaded as grayscale the only value that matters is intensity. In the images represented above there is

a smiley face drawn with intensities of "8" and above. Given that information we can set a threshold of "8" as the least intense color that we want to include in our final solution. If we flag anything over the threshold of "8" as a "1" and anything under the threshold of "8" as a "0" we generate a picture similar to figure 4. Using CUDA it was possible to assign one thread per pixel in the image. Given the size of most of the images that were used, a modern NVIDIA card could easily process every single pixel in the image in a few iterations. Images are divided up and worked in logical pieces. A thread block, or a group of logical threads are assigned one area of the image. These logical sections are worked on in parallel on the image up to 65535 [4] threads at once. After that limit the remaining threads must wait until enough resources are freed for them to continue. This allows for an image of any size to be operated on even though it may not finish in one iteration. The following is a small sample of some of the important parts of the thresholding code in C utilizing CUDA.

```
dim3 blocksDim(width / BLOCKSIZE,
               height / BLOCKSIZE, 1)
dim3 threadDim(BLOCKSIZE, BLOCKSIZE, 1);

ThresholdKernel<<< blocksDim, threadDim
>>>(imageDataDevice, width, BLOCKSIZE);

// Kernel that executes on the device
__global__ void ThresholdKernel(char *
imageData)
{
   // Let index be the thread
   // number to operate on
   int index = (blockIdx.y * BLOCKSIZE
             + threadIdx.y) * width +
               (blockIdx.x * BLOCKSIZE
             + threadIdx.x);
   // If the pixel (x) is within the
   // range of 10-60 then mark it as a
   // finding.
   if ((imageData[index] >= 10
     && imageData[index] < 60)){
       imageData[index] = 255;
   }else{
       imageData[index] = 0;
   }
}
```

Code 2. Thresholding completed on the device.

The problem with the above code is that it only functions correctly when the image size is evenly divisible by 32. So an image size of 500 x 500 would not work correctly and pixels would not be operated on. In an attempt to mitigate this problem, logic was introduced to attempt to dynamically try and find the best block size for the current picture. This logic will start at 32 and try all numbers from 32 to 1 in attempt to find an evenly divisible number. The code for this logic is as follows.

```
int BLOCKSIZE = 32;
int not_divisble = 1;

// Attempt the largest blocksize that is
// possible for the image that is given
while (not_divisble ){
   if ((width % BLOCKSIZE == 0)
   &&  (height % BLOCKSIZE == 0)){
      not_divisble = 0;
      break;
   }else{
      BLOCKSIZE = BLOCKSIZE - 1;
   }
}
```

Code 2. Finding the largest possible block size

The following images are an example product of thresholding.

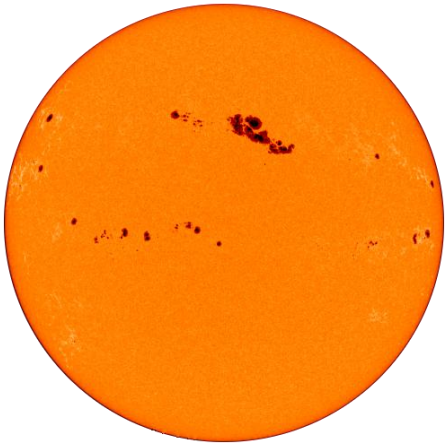

Figure 5 Sun image before thresholding[5].

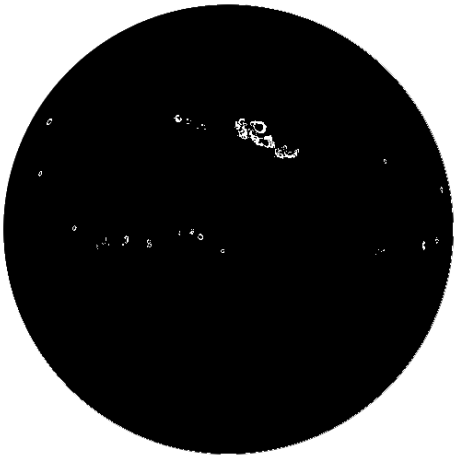

Figure 6 Sun image after thresholding.

After the thresholding is completed the image needs cleaned up a little bit. The outer edge of the Sun is found as the same color as the sunspots that

appear on the image. To solve this problem another method was used to search vertically on the image to detect where the outer ring of the Sun occurs. After the ring is detected it can be erased. This process must be done in a sequential manner and would not fit the GPGPU architecture. After the outer ring is erased the only pixels left on the image are the spots themselves. OpenCV can then be used to find each contour. The contours can then be labeled on the image and recorded according to size and position. This operation was done with OpenCV because of the difficulty of threads to communicate between thread blocks. If a spot pans to thread blocks the spot would end up being counting twice. The OpenCV operations for this function are highly optimized and extremely fast, so there was no demand to create a solution for this operation in CUDA. The relative size of each sunspot can be found by searching for the diameter of the Sun in pixels. This can be done by starting at the top and bottom of the image and searching inward until a more intense pixel is discovered. Subtracting the amount of pixels searched through to find the more intense pixels from the total height of the image will give the diameter of the Sun. Then take the diameter of the Sun itself which is about 1,392,00km [6] and divide it by the number of pixels that the Sun in the image is and get an estimate of how large each pixel in the image represents in reality. This can then be applied to each sunspot to get a real life size. This size, however, is only accurate down to the size of the pixel in the image. Therefore the higher resolution of the image the more accurate the size estimates will be. This operation was not completed on the device because of the sequential nature of the problem. To find the diameter pixels need to be searched in order starting from the top and bottom. Since only about 20 pixels at most will need to be searched it was not conducive to use CUDA to analyze this small dataset.

3 RESULTS AND DISCUSSIONS

Testing was performed on two machines: the first features a Phenom 9950 Quad-Core 2.5 GHz processor with 6GB of ram and a GeForce GTX 470 running Ubuntu 10.10 32-bit with CUDA Toolkit version 4.0 RC2. The second system features an Intel Core i7 950 Quad-Core 3.07GHz with 12GB of ram, a Tesla C2050, and a GeForce GT 430 for display purposes running on Fedora 14 64-bit. The timing differences found between these two devices were negligible so the first machine was chosen for the results in this study.

The resulting times of this project seemed to be an approximate 3-10 times speedup. Initial results were around a 1000 times slowdown, but after tweaking and testing, a problem was discovered. There is a significant "warm-up" time that is involved with allocating the first memory space on the device. This normally takes around 1.5 seconds. If the initial memory allocation is done during the first image analysis this will cause the overall timing of the device function to be very slow. However, if this "warm-up" is performed before any computation, by allocating and freeing some temporary memory, the computational steps that deal with the image become much faster. Every memory allocation after the initial one is very fast and thousands of images can be processed in a row with no repeat of the initial slowdown. This is a crippling factor for GPU processing while processing small data sets. However, this time can be made up in the long run with larger data sets. The timing done in this experiment only takes into consideration the time it takes to complete the thresholding operation on the device and host. The times to load the image, warm-up the GPU, and find contour lines are not considered because both of these functions could not be completed on both GPU and CPU.

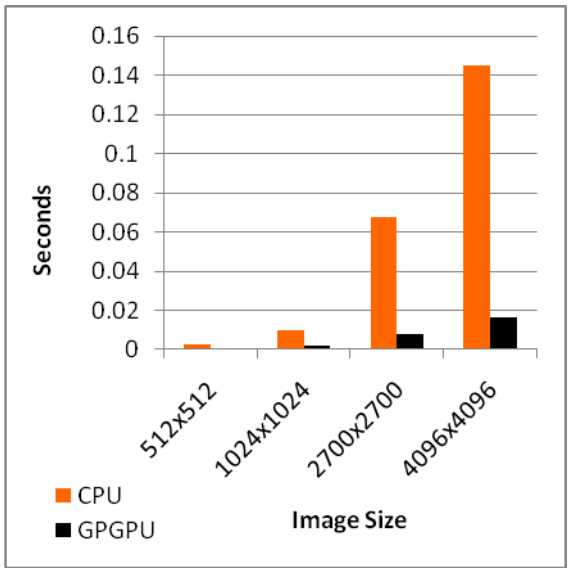

Figure 7. Thresholding performance measurements for a GTX 470 and AMD Phenom 9950 Quad-Core 2.5 (Times do not include GPU warm-up time or reading the file in from disk. Values are an average of 100 runs each.)

One main bottleneck with processing these images with CUDA is waiting for the I/O of loading the image from a hard disk. A file must be opened and read sequentially into memory which will take much more time than actually processing the image with CUDA once it is in memory. Also, the time it takes to "warm-up" the GPU is significant, so significant that the CPU can actually finish the job 100 times before

the GPU is ready to process its first image. The fact that more of the image processing cannot be placed on the GPU is also a problem, so the speedup is left to a minimal amount. If it were easier for thread blocks to communicate information so that more of the contour detection could have been placed on the GPU, it would have provided much more impressive times.

Another difficulty in analyzing sunspots is not the discovery of the sunspot, but tracking of the sunspot. A sunspot can be easily found by searching for a more intense area of darkness surrounded by much lighter pixels and recording size and shape of the sunspot. In order to track the sunspot, however, after detecting the area it must then be compared to previously saved data to see if it is actually the same sunspot that has moved, grown or shrunk. [2] This analysis will give scientists a better idea if there is an abundance of new sunspot activity or long-lasting sunspots that change gradually. This becomes even more difficult because of the Sun's rotation; in other words, just because a sunspot appears to have moved on the Sun's surface that doesn't actually mean it has changed location, but in a still image photograph it could have that illusion. So in order to track the movement of the sunspots, past runs will need to be kept in memory as a reference. One drawback to the GPGPU architecture is that it makes difficult to divide up the data set in the contour detection step. If each section in the image was assigned a thread block there would be conflicts where a critical region, or sunspot, spanned between two logical areas that were scanned by separate thread blocks. Each thread block would then try to perform the region growing on the same area and try to assign two different values to the same region. This area will take more time and research to develop a possible solution. Other efficient algorithms will need to be established in the future to allow the tracking of sunspots from one picture to the next. This will be the next step for research in this project.

5 CONCLUSIONS

The GPU provides a suitable solution for the overwhelming amounts of data that are provided by professional and amateur astronomers alike. CUDA provides an easy solution when a large amount of data throughput is required, however it may not be suited for smaller processing jobs. The amount of time that it takes to warm up the GPU and transfer memory to the device is far greater than CPU processing time on smaller projects. OpenCV has recently added CUDA support to its own libraries and claims a 5 to 10 time's speedup. This falls in line with the results that have been shown in this study. They have also had similar problems with the warm-up time of the device and claim a speedup only after the first picture has been processed. In the future if a workaround for this "warmup" time could be found it may tip the favor for the GPU for all size processing jobs. This style of processing architecture is becoming more common and appears to be in the market to stay. This study demonstrates the potential power of the GPU as this technology matures. Similar styles of programming will dominate the future with both CPU and GPU cores increasing yearly. However, given the growth trends of the GPU architecture, it seems to be much more scalable than CPU architecture.

6 FUTURE WORK

There are several ways to expand research on this software, the first of which would be to expand the ability to process video. During this study a function that allows video to be captured from a camera device and processed with CUDA in order to detect sunspots has been created. It currently can process around 3.7 fps. With some refining and tweaking it could be possible to double or triple that number. Most of the bottleneck for this operation comes from capturing the image from the camera device and copying it to the desired data set. The program will need to be modified to accept a prerecorded video that can be loaded from the hard disk to improve input speed. This can be further improved if multiple images could be processed in parallel on both CPU and GPGPU. The actual analysis of the image is extremely fast. It would also be important to add tracking to this video feature, so that spots can be recorded as they move and fade in and out. This part of the project was postponed due to time constraints.

7 LITERATURE REVIEW

There are many studies involving image processing that are very similar to what was accomplished in this paper. OpenCV has even implemented their own GPU version of a thresholding function in their libraries. There is also other software that is being developed for analyzing and tracking sunspots. A few examples are discussed in *Automatic Sunspot Classification for Real-Time Forecasting of Solar* by T.Colak and Qahwji [7] and *Technique for Automated Recognition of Sunspots on Full Disk Solar Images* by S.Zharkov, V.Zharkova, S.Ipson and A. Benkhali [8]. As both of these studies implement a similar thresholding technique as the one described previously, it would be an easy transition to move the computation to a GPGPU device. With the right equipment solar forecasts

could become 5 to 10 times more efficient with minimal new code.

8 ACKNOWLEDGEMENTS

Many thanks to Dr. George Dimitoglou for presenting the topic of sunspot analysis and providing background information.

REFERENCES

[1] Wikipedia, (2011, February). *Sunspots.* [Online]. Available: http://en.wikipedia.org/wiki/Sunspot

[2] S.Zharkov, V.Zharkova, S.Ipson and A. Benkhalil, *Automated Recognition of Sunspots on the SOHO/MDI White Light Solar Images.* Department of Cybernetics, University of Bradford, UK. 2002.

[3] T. Phillips, *(2009, January) Server Space Weather -- Social and Economic Impacts* . [Online]. Available:http://science.nasa.gov/science-news/science-at-nasa/2009/21jan_severespaceweather/

[4] NVIDIA Corporation, "NVIDIA CUDA C Programming Guide Version 4.0", NVIDIA Corporation, 2011. pp. 3

[5] NASA, *NASA Chats Ask an Expert Your Questions*, [Online]. Available: http://www.nasa.gov/connect/chat/solar_chat2.html

[6] Wikipedia, (2011, April). *Sun.* [Online]. Available: http://en.wikipedia.org/wiki/Solar_diameter

[7] T. Colak, R.Qahwaji, *Automatic Sunspot Classification for Real-Time Forecasting of Solar Activities*, [Online]. Available: http://spaceweather.inf.brad.ac.uk/journals/rast2007_tc_rq.pdf

[8] S.Zharkov, V.Zharkova, S.Ipson, A. Benkhalil, *Technique for Automated Recognition of Sunspots on Full Disk Solar Image*, [Online]. Available: http://solar.inf.brad.ac.uk/publications/eurasip_jsp_ssdetection.pdf

[9] D. Phillips, "Image Processing in C (Second Edition)" in *Plastics*, 2nd ed. vol. 3, J. Peters, Ed. R & D Publications, 1994, pp. 135–145.

[10] NVIDIA Corporation, (2011), *What is Cuda.* [Online]. Available: http://www.nvidia.com/object/what_is_cuda_new.html]

[11] Wikipedia, (2011, February). *Geomagnetic storm.* [Online]. Available: http://en.wikipedia.org/wiki/Geomagnetic_storm

[12] Wikipedia, (2011, February). *Segmentation (image processing).* [Online]. Available: http://en.wikipedia.org/wiki/Segmentation_(image_processing)#Histogram-based_methods