**332:451 Final Project**

**Image Processing**

Eric Gilbert

Gregory Paton

30 December 2012

**Description**

For our project we have created programs to perform various image processing tasks. We’ve implemented resize, blur, and graphical drawing for bitmap images. All these tasks require processing a fair amount of data. Even a relatively small 512x512 bitmap image has over 250,000 pixels with 3 bytes per pixel (one for each of red, green, and blue). Processing this data sequentially is very time intensive, especially when the resolution increases. The HD standard 1080p consists of over 2 million pixels and manufacturers continue to up the standard. UHD 4k resolution is four times the data, over 8 million pixels. Furthermore, when dealing with video, even more processing time is required. Even a relatively low 24 frames per second would take an infeasible amount of time to process sequentially, especially for use in streaming applications that have become popular with the rise of cloud computing. One method to solve this problem is compression, which trades image quality for a smaller storage size. However, a raw image is still required for processing.

This is where parallelization comes in. By distributing the image processing work over multiple threads, the overall computation time can be reduced dramatically. By reading from the original image and storing the results in a new image, all race conditions can be avoided. This makes the process of parallelization fairly straightforward. However, the process can be further optimized by understanding how memory is physically allocated and taking advantage of temporal and spatial locality. To stay competitive in the image processing industry you must have fast programs. In this project we explored exactly how that’s done.

**Goals**

The main goal of this project is to understand how image processing algorithms can be sped up using parallelization. We chose to use CUDA as it seemed like a natural choice for out application. GPUs are typically used for 3D graphics rendering, so using one to process a single image will be no problem.

Also, memory alignment is a huge aspect of image processing. By understanding how the image data is laid out in memory is crucial to optimizing our code. Another goal of ours is to understand how this works and write our program to take full advantage of it to minimize the processing time.

**Resize**

It is often necessary to change the resolution of an image. For this process, a resize filter can be used, also commonly referred to as upsampling (when increasing the image resolution) and downsampling (when decreasing the image resolution). When making the image smaller (downsampling), pixels are dropped from the original image to create a lower resolution output. For enlarging the image (upsampling), some pixel values are duplicated to increase the resolution. In both cases, the algorithm must decide which pixels to drop or keep. A bad decision can result in blockiness, which is undesirable and very noticeable, even to the average viewer. This is where the filter aspect comes in. There are various interpolation types that can be applied to result in a smoother output, such as nearest neighbor, bilinear, and bicubic, each of which varies in complexity (and thus time to code and compute). For our purposes we chose nearest neighbor. It is relatively simple and results in a blockier result, but we were more interested in how parallelization could speed up the process as opposed to the intricacies of sampling algorithms.

Once the algorithm was implemented in C++, the process of optimization parallelization began. First, we looked at how the image was being iterated over. At first, the algorithm was viewing the image by columns. However, when we changed this to have it travel by rows, we noticed a significant speedup. This is due to how the image is laid out in memory. A bitmap image can be thought of as one large array of the image rows placed next to each other. Also, in our case the red, green, and blue components of each pixel are given one byte and placed in memory in the order blue green red. Therefore, to optimize performance, we read and write from images by iterating through rows and accessing first blue, then green, then red.

Next, the decision of how to allocate work among threads and blocks was made. We decided to give each thread a block of the image to process. This was done by dividing the columns up and distributing them among the threads and dividing the rows and distributing them among thread blocks. Among all the implementations we tried, this seemed to work the best and we noticed significant speedup when increasing the number of threads and blocks.

**Testing**

For testing, we used the classic Lena image in a 512x512 bitmap. The image is begin upsampled to 1500x1000. We chose this size as it’s not an even power of two and will definitely require use of the nearest neighbor aspect of the algorithm. Running the program sequentially takes about 563ms. This was calculated running the CUDA program with a block size of 1x1 and a grid size of 1x1. Running it as a standard C++ program would be slightly faster due to the overhead of CUDA memory allocation, but we felt it was a good base comparison for our tests.

First, the program was run while varying a single dimension of the block size from 1 to 8 and holding the grid size constant. Varying the x or y dimension resulted in similar results, shown in the graph below.

*Varying a single block dimension from 1 to 8 (time in milliseconds)*

Next we did the same thing, but held the block size constant and altered the grid size. Varying the x or y dimension resulted in similar results, shown in the graph below.

*Varying a single block dimension from 1 to 8 (time in milliseconds)*

When changing the grid dimension we saw a much bigger speedup than when changing the block dimension. This is most likely due to the fact that each block has it’s own memory. When increasing threads within a single block, the memory is filling up and global memory must be accessed. This takes more time and results in a decent slow down. By increasing the number of blocks on the grid, we are increasing the overall memory, and global memory accesses are decreased. This goes to show how memory accesses can really hurt performance.

Note: We also tested dimensions that resulted in the same number of blocks/threads and got similar results.

**Blur**

**Testing**

For testing, again we used the Lena 512x512 bitmap. A window size of 9x9 is used as this resulted in a noticeable blur without taking too much time to run. Running the program sequentially takes about 8996ms. Again, this was calculated running the CUDA program with a block size of 1x1 and a grid size of 1x1 as opposed to a standard C++ program.