ECE 8780

High Performance Computing with GPUs

Homework No: 1

Image Greyscaling

Dakota Fulp

February 4, 2021

Contents

List of Figures

1 Abstract

Recently, many significant advancements in technology have made more problems tractable. Parallel computing is one area that has seen plenty of advancements with CUDA's release in 2007, making GPU programming more accessible. However, while CUDA is simple to use, optimizing a CUDA kernel is challenging due to several factors. These factors include optimizing memory access patterns, kernel launch configurations, and thread block size.

In this assignment, I aim to develop a CUDA kernel to convert RGBA colored images to greyscale images. During this process, I highlight the many different challenges encountered during the development of the kernel and demonstrate the impacts each has on the kernels' overall performance. Upon developing the kernel and launch configuration, I find 32x32 and 16x16 thread block sizes lead to the best performance across all tested GPU models. I also find that the kernel has the highest performance when run on p100 or v100 GPU models and has a lower performance when run on a k40 GPU model. Finally, upon analyzing the percent of time spent moving data between the host and the device, I find the kernel can be further optimized to reduce this overhead.

2 Introduction

In recent years, significant advancements in technology have made previously intractable problems possible. The area of parallel computing has seen many improvements. In 1994, the first draft of MPI was released, leading to significant changes in parallel programming. Another major shift came in 2007 with the release of CUDA. CUDA is a GPU interface platform developed by Nvidia and has made working with GPU's accessible to many more individuals.

CUDA programming has uses in various fields and is simple to integrate into any existing codebase. Specifically, CUDA and GPU programming enable the completion of repetitive processes in a fraction of the serial time. However, while CUDA is easy to implement, it is challenging to master due to the many possible nuanced optimizations one can make.

In this assignment, I develop a CUDA kernel that converts colored RGBA images to greyscale. During the process of creating the kernel, I also aim to understand which approaches lead to more optimized kernels and which lead to less optimized kernels. Using these findings, I will be better equipped to create even more optimized kernels in the future.

3 Methodology

When developing the kernel, there are two main focus areas: the kernel function and the launch configuration. By testing different approaches to each, I demonstrate best practices for developing other kernels in the future.

The first focus area is building and optimizing the kernel function itself. In this assignment, the kernel must take an RGBA image of any size and convert it to greyscale. The algorithm I use to accomplish this is seen in the Appendix's Kernel Function Code. The first step of this algorithm determines the pixel's location for the current thread that is working on the pixel. The algorithm uses the current thread's block, the block's dimensions, and the block's location in the grid to accomplish this. The algorithm then uses this location to determine the offset value, which indicates the pixel's location in the one-dimensional RGBA input array and greyscale output array. Next, the algorithm loads the corresponding RGBA pixel's value and uses it to determine the corresponding grey value using Equation 1. This new value is then loaded back to the correct location in the greyscale array. Due to the way this algorithm indexes the data, it is more optimized than other forms of this algorithm. With the kernel function developed, the next step is to determine the optimal launch configuration.

$$grey[offset] = 0.299 * R + 0.587 * G + 0.114 * B \tag{1}$$

The launch configuration one uses to call a GPU kernel can greatly impact a kernel's performance. To demonstrate this, I investigate four possible launch configurations to determine which leads to the best performance on a p100 GPU. During this investigation, I test four images of different sizes: Lena (512x512), Airplane (512x512), Tulips (768x512), and Watch (1024x768). The code for each of the following configurations can be seen in the Appendix's Kernel Launch Configurations 1 to 4. The before and after for each of the four test images can be seen in the Appendix's Image section.

The first configuration I test is the simplest to implement and makes a number of blocks equal to the number of pixels in the image. While this is simple to implement, this approach is not optimized at all and leads to massive overhead, as I show later. The second configuration I test limits the number of blocks used. In this configuration, the user determines the number of blocks to use in the grid, and this is used to determine the number of threads to use per block. If this configuration finds the number of threads is over 1024, which is the maximum number of threads per block possible in CUDA, it increases the number of blocks to account for this. The third configuration I test creates blocks equal to the number of rows in the image. Each of these blocks has threads equal to the number of columns in the image. While this is also simple to code and is more effective than the first configuration, it is limited to images with a width of less than 1024. The final configuration I test limits the number of threads in each block given user input. Using this input and the dimensions of the image, the number of needed blocks is also determined.

Upon testing each of these configurations using the four test images and a p100 GPU,

I find the results found in Figure 1. From these results, it is clear that all tested configurations do better than serial. However, as previously stated, configuration one introduces too much overhead and is not optimized. As such, this configuration should never be used. When looking at configurations two, three, and four, I find each takes significantly less time than configuration one. While each look to be equally efficient, configuration four is the best option as it works for images of any size. Specifically, configuration two introduces more logistic overhead to determine how many threads to use per block. Also, even when the user desires only so many blocks, this is not guaranteed due to the 1024 thread limit per block found in CUDA. As such, this configuration is not ideal. In configuration three, a similar challenge exists since images with a width greater than 1024 will fail, and therefore, this configuration is also not ideal. Accordingly, I decide to use configuration four for all future trials as this is the most flexible kernel launch configuration.

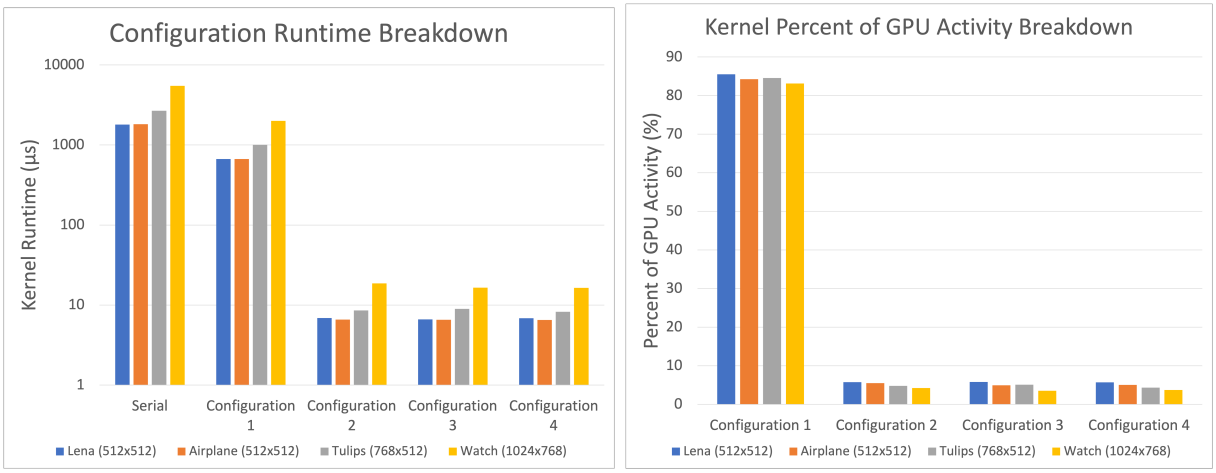

Figure 1: Configuration Comparisons

While I have determined the optimal kernel function and launch configuration, more optimizations are possible. To further optimize and test my code's abilities, I must test different thread block sizes and how this kernel works on different GPUs.

4 Results

To further optimize and test the ability of the previously mentioned kernel function and configuration four, I test different thread block sizes and different GPU models. Through these tests, I determine which block size fits each image best and how my approach changes with the GPU model.

In my tests, I test four different thread block sizes and three different GPU models. The block sizes I test are 4x4, 8x8, 16x16, and 32x32 threads. The GPU models I test are k40, p100, and v100 GPU models. During each test, I use nvprof to analyze the total kernel's

runtime, the percentage of the overall runtime used by the kernel, and the percentage of the overall runtime used moving data to and from the host.

Figure 2 demonstrates how the total kernel runtime changes with different block sizes and different GPU models. From these figures, it is clear that the thread block size directly impacts the total runtime. Using smaller thread block sizes, we see results similar to our previous tests with configuration 1, which used 1x1 block sizes. From this trend, it is clear that if the block size is too small, the performance will suffer due to the overhead of creating the extra blocks. This trend is consistent across GPU models as well, and when comparing the runtime over different GPU models, I find the more powerful GPU's take less time to complete the work. This is expected as the more powerful GPU's have more resources to complete the necessary work.

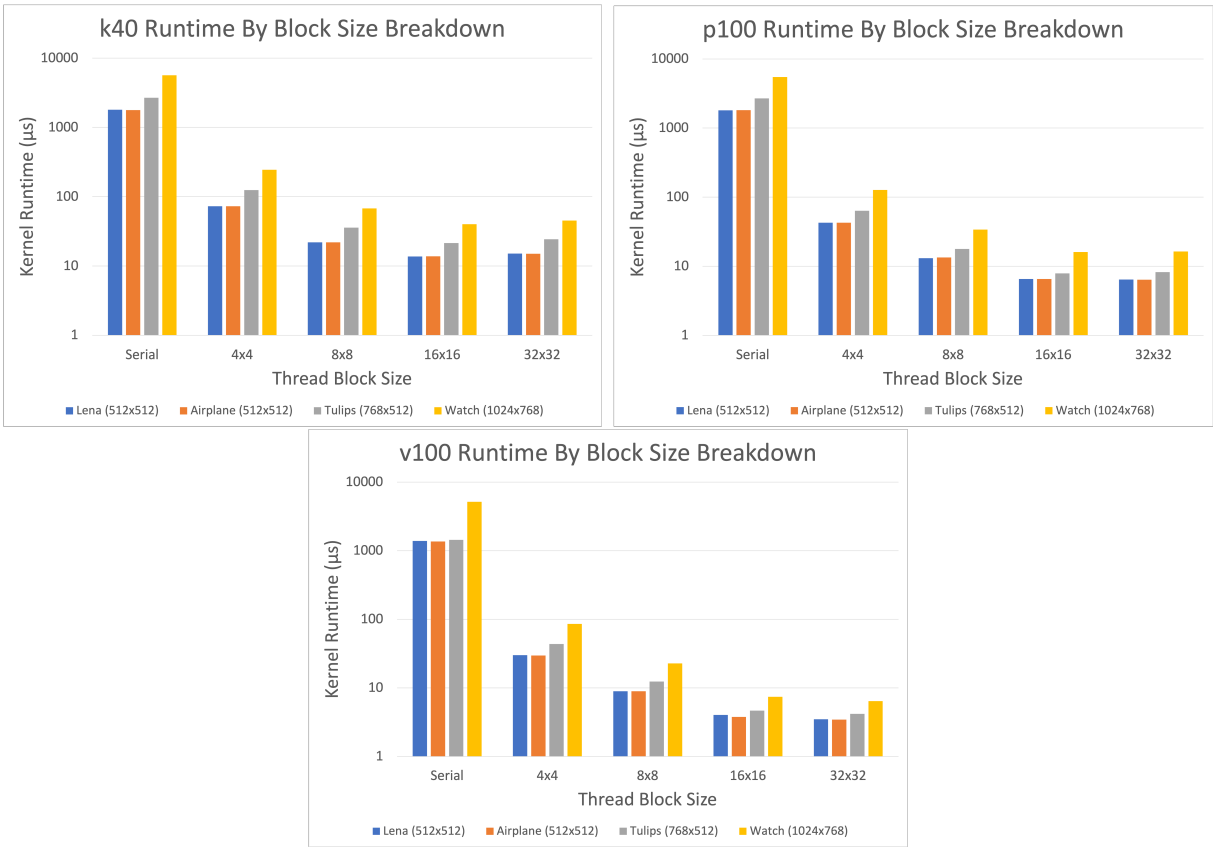

Figure 2: Block Size Effects on Runtime when Run on Different GPU Models

The next metrics I analyze are the percentages of the overall runtime used by the kernel function, transferring data from the host to the device, and transferring data from the device to the host.

Figure 3 demonstrates how the kernel activity percentage of the overall runtime changes with different block sizes and different GPU models. Analyzing this metric alongside the

runtime is critical as it is capable of highlighting trends that basic runtime analysis would not. From this set of Figures, we see a similar trend to the runtime figures. Specifically, we find that the thread block size directly impacts the runtime percentage occupied by kernel activity. We also find similar trends across each of the different tested GPU models.

Figure 3: Block Size Effects on Kernel Percent of GPU Activity when Run on Different GPU Models

Figure 4 and Figure 5 demonstrates how the transfer from host to device and vice versa percentages changes with different block sizes and different GPU models. From the first sets of figures, it is clear that larger block sizes lead to a larger percentage of the overall time being used to transfer data from the host to the device. While this may look like it is taking more time, this is actually due to the overall time being smaller on larger block size runs since the kernel finished more quickly. When looking at this trend across different GPU models, I find the more powerful GPUs have higher percentages of time taken to transfer data from the host to the device. Again, this is due to the newer, more powerful GPUs taking less time to accomplish the work. When looking at the second set of figures, an opposite view is seen. Specifically, when using newer GPU models, the percentage of time needed to transfer the data from the device back to the host looks to decrease on newer

models. This is an interesting trend and is most likely due to the improved performance on the higher end GPUs.

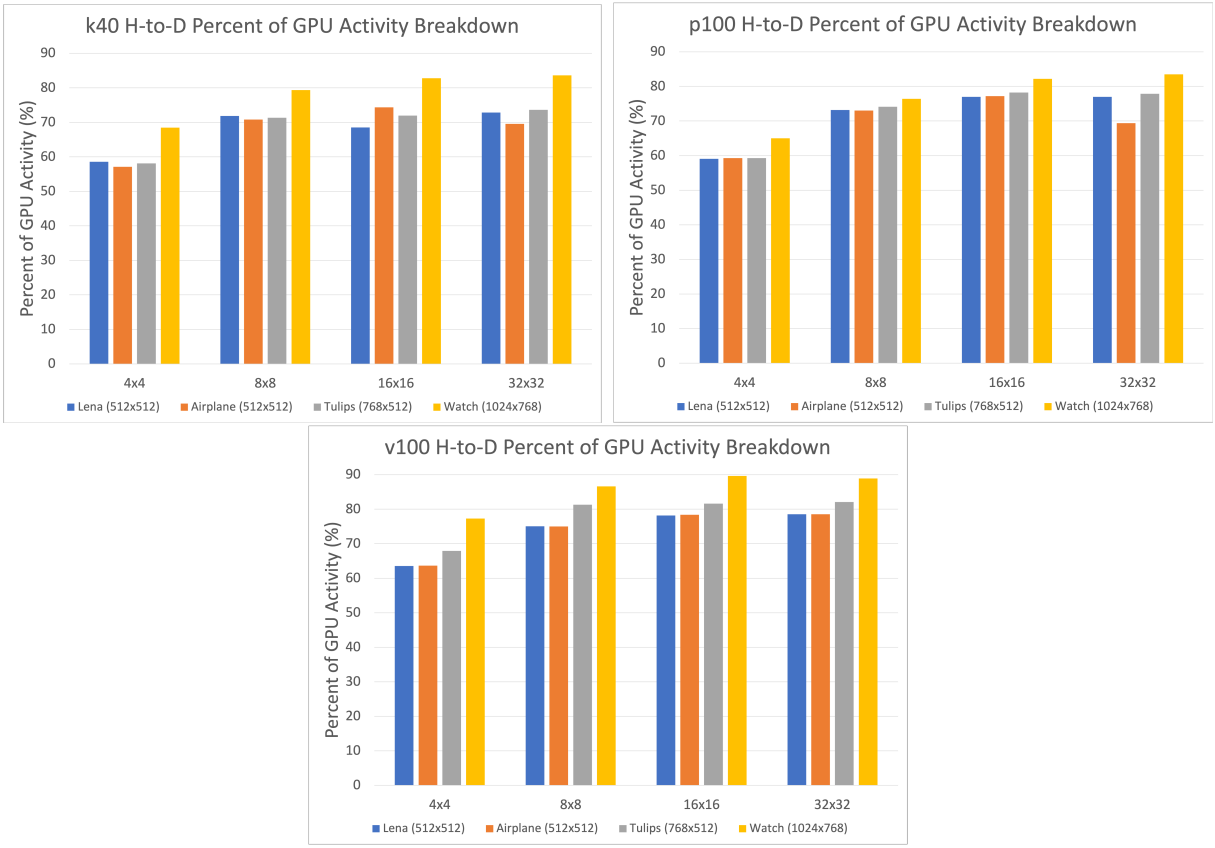

Figure 4: Block Size Effects on Host to Device Data Transfer Percent of GPU Activity when Run on Different GPU Models

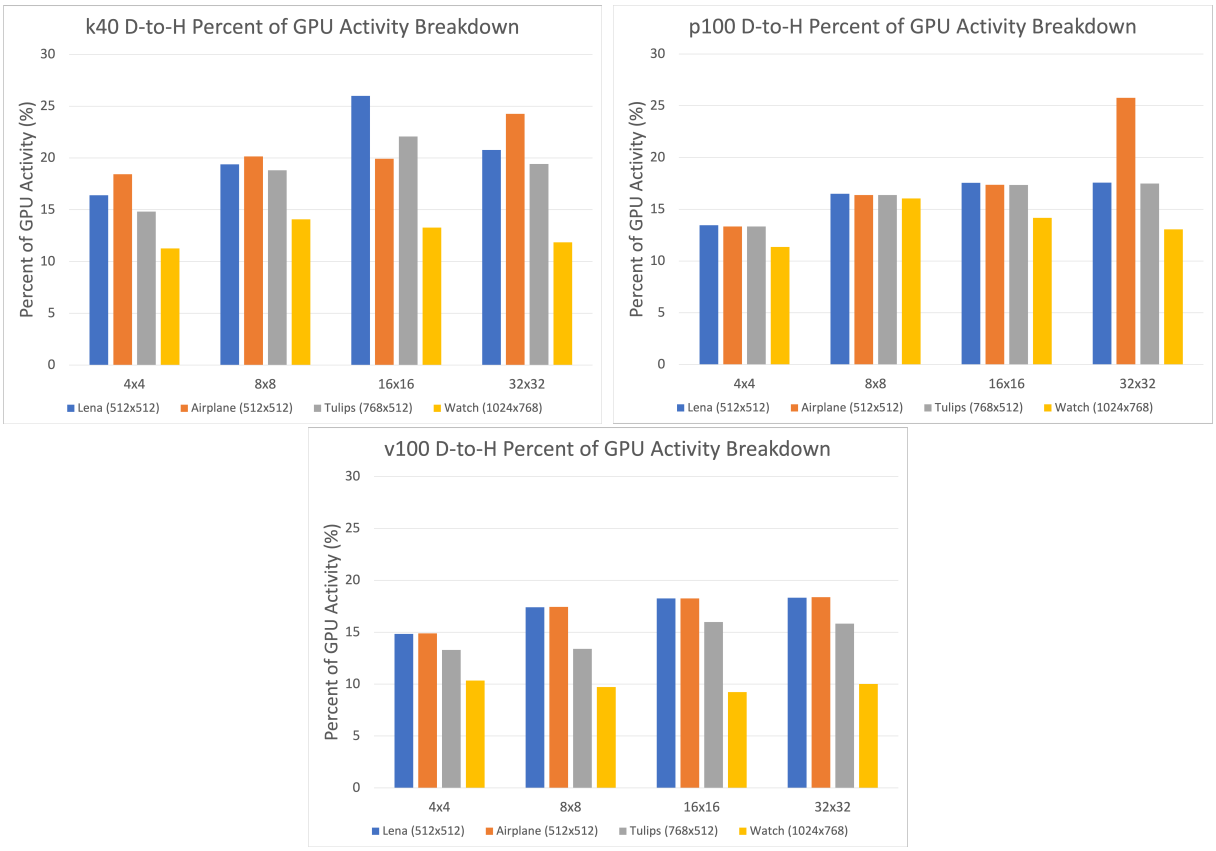

Figure 5: Block Size Effects on Device to Host Data Transfer Percent of GPU Activity when Run on Different GPU Models

By looking at all three sets of Figures together, a full breakdown of each runs time is seen. From this view, I find that as the block size increases, the time needed to complete the work decreases, which in turn increases the host to devices transfer percentage. However, I do not see too much of a difference in the device to host transfer percentage. I expect this is due to most of the time being used to transfer from the host to the device. Overall, it is clear that choosing the correct block size leads to less time being used by the kernel.

From these results it is clear that the most optimal block size for each images is either 32x32 or 16x16 and the most optimal GPU model for each is either the v100 or p100.

5 Discussions and Conclusion

The results from this assignment have made some aspects of coding CUDA kernels clear and have highlighted the many challenges to optimizing them. First, when testing different kernel launch configurations, I find that not all configurations are created equal. The kernel launch configuration one uses to launch a kernel function directly impacts the performance

of the kernel function. This is due to the introduced overhead that comes from creating and working with extra blocks in the grid. If the block size is too small, then more blocks are needed to complete the work, and this causes more overhead as each block has its own shared memory and set of threads. Therefore, some of the main factors that affect the kernel function's performance are the number of blocks in the grid, the organization of the blocks in the grid, and the number of threads used within each block of the grid. However, these are not the only factors that affect the kernel function's performance. Another performance factor is the model of the GPU being used. From my findings, it is clear that some GPU models handle data more efficiently while other older models are not as efficient in handling the data. For example, the older k40 GPU model takes longer passing data from the host to the device, while the v100 more effectively does this. In conclusion, the many factors that come into play when attempting to optimize a kernel function's performance add extra challenges. It is also clear that the kernel setup I currently have could still be further optimized.

A Appendix

Below are the images used in the completion of this assignment.

Figure 6: Lena 512 x 512

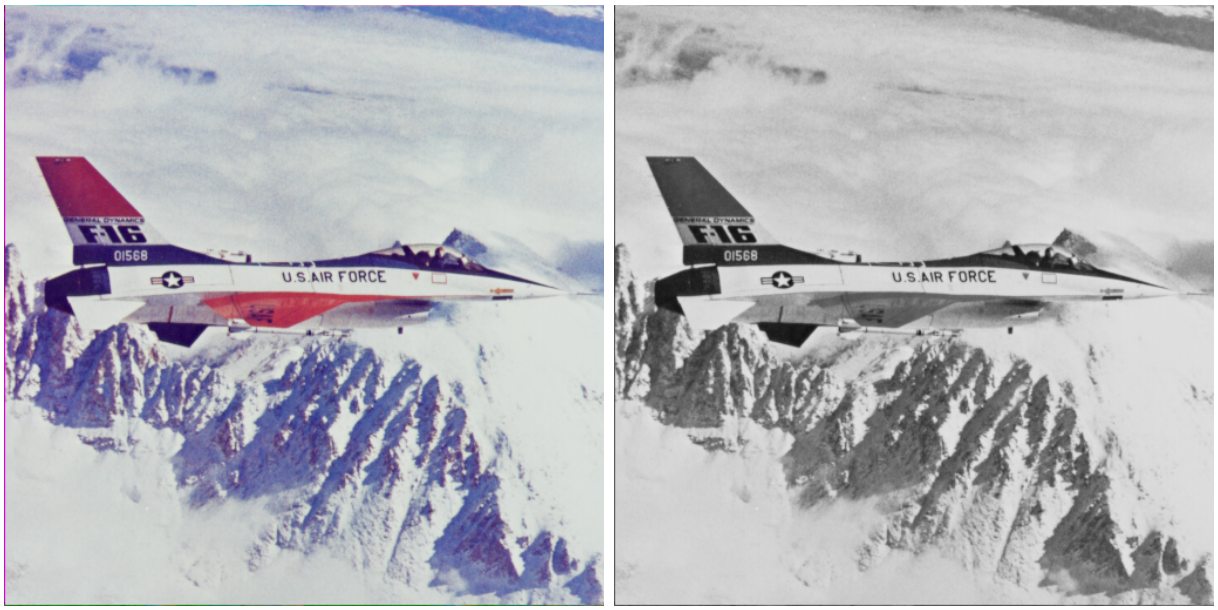

Figure 7: Airplane 512 x 512

Figure 8: Tulips 768 x 512

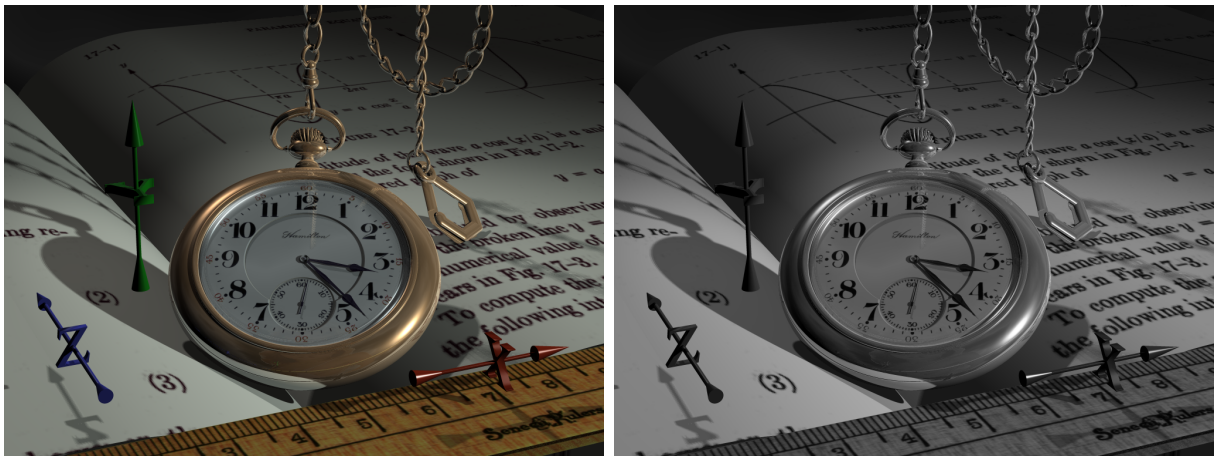

Figure 9: Watch 1024 x 768

End

Below is the kernel code used in the completion of this assignment.

Kernel Function Code

```
__global__
void im2Gray(uchar4 *d_in, unsigned char *d_grey, int numRows, int numCols){
  // Column Indicator
  int x = blockIdx.x * blockDim.x + threadIdx.x;
  // Row Indicator
  int y = blockIdx.y * blockDim.y + threadIdx.y;

  // Only do valid pixel locations
  if(x < numCols && y < numRows) {
    // Get one dimension array offset
    int grey_Offset = y * numCols + x;
    // Get corresponding rgba pixel at this location
    uchar4 rgba_pixel = d_in[grey_Offset];
    // Calculate new grey value
    d_grey[grey_Offset] = (unsigned char)((float)rgba_pixel.x*0.299f +
    (float)rgba_pixel.y*0.587f + (float)rgba_pixel.z*0.114f);
  }
}
```

End

Kernel Launch Configuration 1

```
void launch_im2gray(uchar4 *d_in, unsigned char* d_grey,
```

```
size_t numRows, size_t numCols, int BLOCK){
    // Ensure there are not over BLOCK number of blocks
    // Given the number of total blocks, determine the number of threads
    // needed per block

    // Configuration 1
    dim3 grid(numCols,numRows,1);
    dim3 block(1,1,1);

    // Call Kernel
    im2Gray<<<grid,block>>>(d_in, d_grey, numRows, numCols);
    cudaDeviceSynchronize();
    checkCudaErrors(cudaGetLastError());
}
```

End

Kernel Launch Configuration 2

```
void launch_im2gray(uchar4 *d_in, unsigned char* d_grey,
size_t numRows, size_t numCols, int BLOCK){
    // Ensure there are not over BLOCK number of blocks
    // Given the number of total blocks, determine the number of threads
    // needed per block

    // Configuration 2
    size_t grid_x = std::ceil((float)BLOCK/2);
    size_t grid_y = std::ceil((float)BLOCK/2);
    size_t block_x = std::ceil((float)numCols/grid_x);
    size_t block_y = std::ceil((float)numRows/grid_y);
    size_t block_size = block_x * block_y;
    size_t new_block = BLOCK;

    while (block_size > 1024){
      new_block = new_block * 2;
      grid_x = std::ceil((float)new_block/2);
      grid_y = std::ceil((float)new_block/2);
      block_x = std::ceil((float)numCols/grid_x);
      block_y = std::ceil((float)numRows/grid_y);
      block_size = block_x * block_y;
    }

    std::cout << "x: " << grid_x << " y: " << grid_y << std::endl;
    dim3 grid(grid_x,grid_y,1);
    std::cout << "x2: " << block_x << " y2: " << block_y << std::endl;
    dim3 block(block_x,block_y,1);

    // Call Kernel
    im2Gray<<<grid,block>>>(d_in, d_grey, numRows, numCols);
    cudaDeviceSynchronize();
```

```
    checkCudaErrors(cudaGetLastError());
}
```

End

Kernel Launch Configuration 3

```
void launch_im2gray(uchar4 *d_in, unsigned char* d_grey,
size_t numRows, size_t numCols, int BLOCK){
    // Ensure there are not over BLOCK number of blocks
    // Given the number of total blocks, determine the number of threads
    // needed per block

    // Configuration 3
    dim3 grid(1,numRows,1);
    dim3 block(numCols,1,1);

    // Call Kernel
    im2Gray<<<grid,block>>>(d_in, d_grey, numRows, numCols);
    cudaDeviceSynchronize();
    checkCudaErrors(cudaGetLastError());
}
```

End

Kernel Launch Configuration 4

```
void launch_im2gray(uchar4 *d_in, unsigned char* d_grey,
size_t numRows, size_t numCols, int BLOCK){
    // Ensure there are not over BLOCK number of blocks
    // Given the number of total blocks, determine the number of threads
    // needed per block

    // Configuration 4
    // Ensure BLOCK is valid
    size_t block_size = BLOCK;
    if (block_size > 32){
      block_size = 32;
    } else if (block_size <= 0){
      block_size = 1;
    }
    // Set grid and block dimensions
    dims3 grid(std::ceil((float)numCols/(float)BLOCK),
    std::ceil((float)numRows/(float)BLOCK),1);
    dim3 block(BLOCK, BLOCK, 1);

    // Call Kernel
    im2Gray<<<grid,block>>>(d_in, d_grey, numRows, numCols);
    cudaDeviceSynchronize();
    checkCudaErrors(cudaGetLastError());
}
```