Concurrent Programming
Assignment #5

Jarrod Boone 201714680
Yash Patel 201842812

July 30th, 2022

Introduction

We decided to improve our Assignment #4 implementation using CUDA to filter each pixel. This would allow us to use numerous amounts of cores from the GPU in order to improve the performance of the image modification process.

Code Design

Our design using MPI relied on 18 processes where 16 were used to filter the 16 sub-images, the 17th process was used to determine the state of each sub-image process, and an 18th process was used for image reconstruction, accepting the filtered pixels from the 16 sub-images.

For our CUDA design, our implementation was broken down into much simpler terms. We would basically distribute the filtering work of each thread into the number of pixels each thread had to process.

The program accepts two arguments, the number of blocks, as well as the number of threads in each block. It then runs a filter function, which uses the threads to filter the necessary amount of pixels, dividing the work up between threads and pixels evenly.

To start, we will look at the code structure for the CUDA implementation:

```
69  int main(int argc, char *argv[])
70  {
71      const size_t N = 256 * 256; // 1M elements
72      int *image;
73      int *new_image;
74
75      if (argc != 3 || !isNumber(argv[1]) || !isNumber(argv[2]) || 65536 % (atoi(argv[1]) * atoi(argv[2])) != 0)
76      {
77          std::cout << "Usage:\n\n./main.exe <#Blocks> <#Threads>\n\nRequired: 65536 % (#Blocks * #Threads) == 0\n" << std::endl;
78          exit(0);
79      }
80
81      int number_of_blocks = atoi(argv[1]);
82      int number_of_threads = atoi(argv[2]);
83
84      cudaMallocManaged(&image, N * sizeof(int));
85      cudaMallocManaged(&new_image, N * sizeof(int));
86
87      read_pgm_file("pepper.ascii.pgm", image, new_image);
88
89      // Performance varies depending on total # threads.
90      // Try running nvprof ./main.exe using different combinations
91      // of blocks and threads to see the performance difference.
92      filter<<<number_of_blocks, number_of_threads>>>(image, new_image);
93
94      cudaDeviceSynchronize();
95      write_pgm_file("output.pgm", new_image);
96
97      // Free memory
98      cudaFree(image);
99      cudaFree(new_image);
100
101     return 0;
102 }
```

We start by accepting two command line arguments and use those as the number of threads and blocks. We then allocate 65536 integers for both the image and new_image matrix in order to store the image and the filtered image.

We then call the filter function, which applies the Gaussian kernel to each pixel, where the work is divided up amongst threads. After, we just synchronize the threads once they all complete their work on the image, write to an output file, and then free the memory.

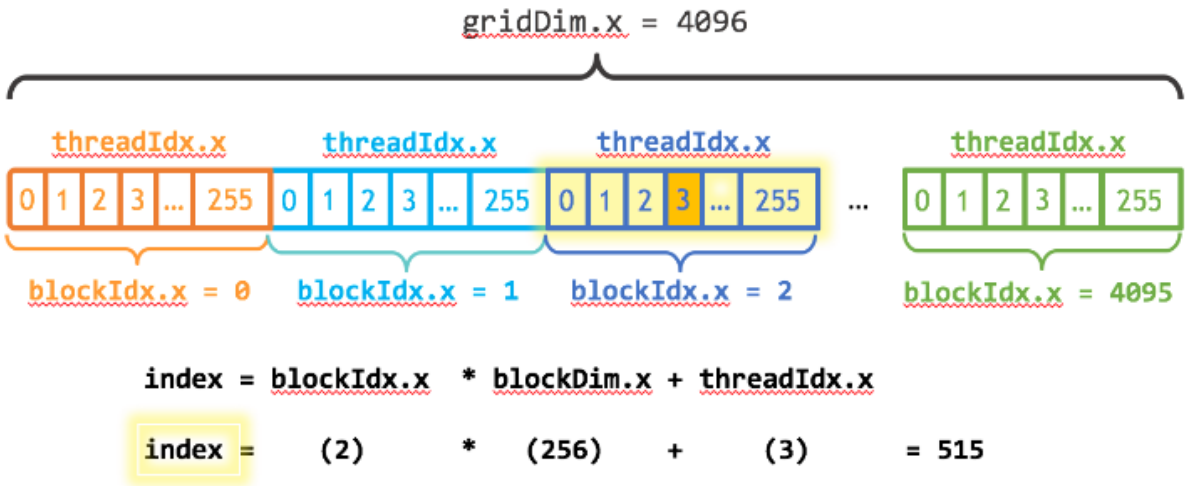
```
__global__
void filter(int* image, int* new_image)
{
    int gaussian_kernel[9][9] = {
        {0, 0, 3,   2,   2,   2, 3, 0, 0},
        {0, 2, 3,   5,   5,   5, 3, 2, 0},
        {3, 3, 5,   3,   0,   3, 5, 3, 3},
        {2, 5, 3, -12, -23, -12, 3, 5, 2},
        {2, 5, 0, -23, -40, -23, 0, 5, 2},
        {2, 5, 3, -12, -23, -12, 3, 5, 2},
        {3, 3, 5,   3,   0,   3, 5, 3, 3},
        {0, 2, 3,   5,   5,   5, 3, 2, 0},
        {0, 0, 3,   2,   2,   2, 3, 0, 0},
    };

    // Idiomatic way to get to get the rank of each thread in order when blocks are involved.
    size_t max_pixels = 65536 / (blockDim.x * gridDim.x);
    size_t index = (blockIdx.x * blockDim.x + threadIdx.x) * max_pixels;

    // Each thread will be responsible for a specific amount of pixels.
    for (int i = index; i < index + max_pixels && index < 65536; i++)
    {
        // Convert 1D array of pixel to get 2D coordinate.
        size_t pixel_x = i % 256;
        size_t pixel_y = i / 256;
        int new_pixel_val = 0;

        for (size_t row = 0; row < 9; row++)
        {
            for (size_t col = 0; col < 9; col++)
            {
                // Subtract 4 to center the gaussian filter on the pixel.
                int mapped_gauss_x = pixel_x + col - 4;
                int mapped_gauss_y = pixel_y + row - 4;

                size_t gauss_idx = mapped_gauss_x + 256 * mapped_gauss_y;

                if (mapped_gauss_x >= 0 && mapped_gauss_x <= 255 && mapped_gauss_y >= 0 && mapped_gauss_y <= 255)
                {
                    new_pixel_val += gaussian_kernel[col][row] * image[gauss_idx];
                }
            }
        }

        if (new_pixel_val > 255)
        {
            new_pixel_val = 255;
        }

        if (new_pixel_val < 0)
        {
            new_pixel_val = 0;
        }

        new_image[i] = new_pixel_val;
    }
}
```

A few things to note about our filter function. We use the following calculation:

blockIdx.x * blockDim.x + threadIdx.x

As an idiomatic way of getting the rank of each thread. So say we have 4 blocks with 32 threads each, the last thread in the first block will have the index 31, and the first thread from the second block will have the index 32.

The following is a good example of how this line works:



Afterward, there are simply two loops that filter over the Gaussian kernel and the pixel coordinates. This works the same as our Assignment #4 implementation.

The index and max_pixel values work out so that the correct number of pixels will have their filtering done by the exact amount of threads needed. And since our program allows the user to input the number of threads, using the **nvprof** command line tool, it is very easy to measure performance.

Performance

Using Python's time module, we clocked our Assignment #4 implementation as having an execution time of around 7 seconds. We can then use nvprof tool to measure our CUDA implementation.

To start, we will use a single thread:

```
PS C:\Users\Jarrod Boone\Desktop\Concurrent Programming\A5> nvprof --profile-api-trace none --unified-memory-profiling off .\main.exe 1 1
==13016== NVPROF is profiling process 13016, command: .\main.exe 1 1
==13016== Profiling application: .\main.exe 1 1
==13016== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:  100.00%  104.34ms         1  104.34ms  104.34ms  104.34ms  filter(int*, int*)
No API activities were profiled.
```

We start at 104 ms. Already quite a bit faster than the MPI implementation.

We can try 64 threads on a single block:

```
PS C:\Users\Jarrod Boone\Desktop\Concurrent Programming\A5> nvprof --profile-api-trace none --unified-memory-profiling off .\main.exe 1 64
==16700== NVPROF is profiling process 16700, command: .\main.exe 1 64
==16700== Profiling application: .\main.exe 1 64
==16700== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:  100.00%  2.7425ms         1  2.7425ms  2.7425ms  2.7425ms  filter(int*, int*)
No API activities were profiled.
```

So just with 64 threads, our implementation goes from 104ms to around 3ms. Around 34 times faster!

We can try instead 32 blocks with 32 threads, instead of just a single block. This would amount to 1024 threads in total, so each thread would handle 64 pixels to filter:

```
PS C:\Users\Jarrod Boone\Desktop\Concurrent Programming\A5> nvprof --profile-api-trace none --unified-memory-profiling off .\main.exe 32 32
==11332== NVPROF is profiling process 11332, command: .\main.exe 32 32
==11332== Profiling application: .\main.exe 32 32
==11332== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:  100.00%  215.11us         1  215.11us  215.11us  215.11us  filter(int*, int*)
No API activities were profiled.
```

215 microseconds! Or 0.215 ms. Around 14 times after the single block using 64 threads.

The max performance we can probably achieve would be to have a single thread per pixel. This would mean that the #blocks * #threads must be equal to 65536 as that is the number of pixels in the thread.

We can therefore try 64 blocks with 1024 threads each:

```
PS C:\Users\Jarrod Boone\Desktop\Concurrent Programming\A5> nvprof --profile-api-trace none --unified-memory-profiling off .\main.exe 64 1024
==11504== NVPROF is profiling process 11504, command: .\main.exe 64 1024
==11504== Profiling application: .\main.exe 64 1024
==11504== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:  100.00%  26.240us         1  26.240us  26.240us  26.240us  filter(int*, int*)
No API activities were profiled.
```

Where we get 26 microseconds. So in total, we went from around 7 seconds of the MPI implementation to 26 microseconds with our CUDA implementation.

This means that our CUDA implementation is around **269230** times faster than our MPI implementation!