ECSE 420
Lab 1: Rectify & Pooling Image Compression

Alex Hale
260672475
Thomas Hillyer
260680811

October 15th, 2019

Rectify

Our rectification solution resulted in a properly rectified image. The image is included alongside this report as test_rectify_result.png. Passing a different image through the solution would also yield a properly rectified image. Unfortunately, we were unable to test such an image, because we didn't have access to the lab computers after the demo over the weekend.

why exactly? Also it was due on Tuesday!

The solution works as follows:

After defining filenames and bringing in the test image, the user may specify the number of threads per block with which the test will run. Running the test with more than 1024 threads does not work, because the GPUs have a maximum of 1024 threads executing per block. Any number of threads per block less than or equal to 1024 will work. The hardware limits us to a maximum of 1024 threads per block, but the number of blocks that the hardware possesses is greater than the number of blocks that we need.

Next, pointer variables are defined and the input data is sent from the CPU to the GPU. The number of blocks required is then calculated, and the number of blocks and number of threads are passed to the *rectification()* function along with the required parameters:

```
// figure out how many blocks we need for this task
unsigned int num_blocks = (size_image + thread_number - 1) / thread_number;

// call method on GPU
rectification <<< num_blocks, thread_number >>> (cuda_image, cuda_new_image, size_image);
```

Inside the rectification function, the thread's index in the PNG array is determined by adding *(thread index + (block index)(threads per block))*. The thread then examines the value at that pixel - if it's lower than 127, it sets it to 127. Otherwise, the pixel value is left alone.

One feature that should be added is that if the calculated index % 3 == 0 (i.e. the index represents a value in the Alpha pixel layer), the value should not be rectified. Doing so causes errors in images that have transparent areas within them.

After the rectification function finishes executing, the new_image PNG value is copied back from the GPU to the CPU, and then the resulting PNG is saved to a file. The memory is not freed, because it will next be used in the rectification function.

The final element that was intended to be performed was to calculate the execution time of rectification using {1,2,4,8,16,32,64,128,256} threads. Since we did not have access to the lab computers, this analysis was unable to be performed. It is expected that the speedup would be dramatic for the first few numbers of threads, but would eventually start diminishing at higher numbers of threads as the upper limit of parallelization is reached.

15/25

→ No image
→ No graph/speedup.

Pooling

Pooling compresses an image by performing some operation on regularly spaced sections of the image. In our case, we're performing pooling by choosing the maximum pixel value in each 2x2 area.

We determined that we could only make use of a maximum of one thread per 16 values in an image when performing 2x2 pooling. This is because the same thread must be used for each 2x2 pixel area (because the maximum value in that area must be determined within the context of a single thread), and each pixel in the area has 4 values (R,G,B,A), coming out to 16 values per thread.

```
// maximum number of threads we can use is 1 per 16 pixel values
if (thread_number > ceil(size_image / 16)) {
    thread_number = ceil(size_image / 16);
}
```

In hindsight, it may have been possible to use one thread per 4 pixel values - this is because the maximum was determined within the context of a single colour plane (R, G, B, or A), and so one thread could have been used to determine each maximum, then slot it into the correct index in the new_image pixel array.

Sticking with the original determination that each thread would be responsible for 16 values within the pixel array, the number of blocks required was determined, then passed to the pooling function as follows:

```
// figure out how many blocks we need for this task
num_blocks = ceil((size_image / thread_number) / 16) + 1;
unsigned int blocks_per_row = ceil(width / thread_number);

// call method on GPU
compression <<< num_blocks, thread_number >>> (cuda_image_pool, cuda_new_image_pool, width, size_image, blocks_per_row);
```

Inside the pooling function, the thread's starting index in both the old and new image are determined:

```
unsigned int index = threadIdx.x + (blockIdx.x % blocks_per_row) * blockDim.x;
unsigned int new_index = (threadIdx.x + blockIdx.x * blockDim.x) + 4;
```

Starting from this index, the maximum value within the same colour plane is determined, then slotted into the appropriate index in the new image:

```
for (int i = 0; i < 4; i++) {                          // iterate through R, G, B, A
    unsigned int max = image[index];
    if (image[index + 4 + i] > max) {                  // pixel to the right
        max = image[index + 4 + i];
    }
    if (image[index + (4 * width) + i] > max) {        // pixel below
        max = image[index + (4 * width) + i];
    }
    if (image[index + (4 * width) + 4 + i] > max) { // pixel below & to the right
        max = image[index + (4 * width) + 4 + i];
    }
    new_image[new_index + i] = max;
}
```

Unfortunately, there must be a bug somewhere within the logic of this function, because the output was simply a grey patterned image. The bug was unable to be found.

20/25

→ No image

Appendix? → I asked for all your code in the appendix!