ECSE 420: Parallel Computing Lab 1

October 14th, 2019

David DAlleva - 260689096
Jingxu Hu - 260606017

Lab Group: 39
Professor: Zeljko Zilic

1 Rectification

The rectification algorithm was simple to build. On the CPU side, it was a linear process on a 1D flattened pixel array. Every 4 elements in the array was 1 pixel since it follows the order of RGBA. Thus, looping through the channels of the pixels, we were able to rectify the image based on if the channel was above or below 127. Since this algorithm only took 1 thread to run in the CPU and CPU qantas were divided among many kernel processes, it took some time to execute. When this same algorithm was altered and executed in the GPU, it ran significantly faster as one could see from the speedup graph of the parallel process below. Parallel execution in the GPU required the algorithm to be divided among blocks and threads per block. In order to meet these specifications, the starting offset in our 1D input image array would be modified. In figure 1 below, the first line within the function, the offset was set equal to the thread I.D number multiplied by the actual starting offset based on how many threads was inputed. For example, if we were running with 8 threads, then each thread would have [(4*imgWidth*imgHeight)/8] element locations to process in the image array. Once we've figured out where each thread would be starting in the array, then the rest of the algorithm entailed linear processing of the pixels within that thread's defined number of elements to be served. We've also noticed that the threads within a block in the GPU do not execute in order from I.D = 0 to I.D = N. It jumped back and forth. Thus, during the actual processing of the image array, the algorithm actually jumped around in the image processing a row of defined number of pixels at a time then offsetting to another starting point in the image as defined by our rgbaOffset. For the speedup of the GPU, as the graph indicates, when the number of instantiated threads approaches 128, GPU's computation time plateaus. This is because at a certain number of threads higher than a defined constant, there is a lot more overhead in thread management thus causing the curve to increase as indicated by our speedup curve. Lastly, the rectification process fails on 1 thread when executed within the GPU. This is because the processing time is longer than the given quanta for the thread to execute. Thus, as the thread rectifies the image, as it times out it calls "lodepng" and the function returns a bad access error indicating the image array was corrupted at some index value.

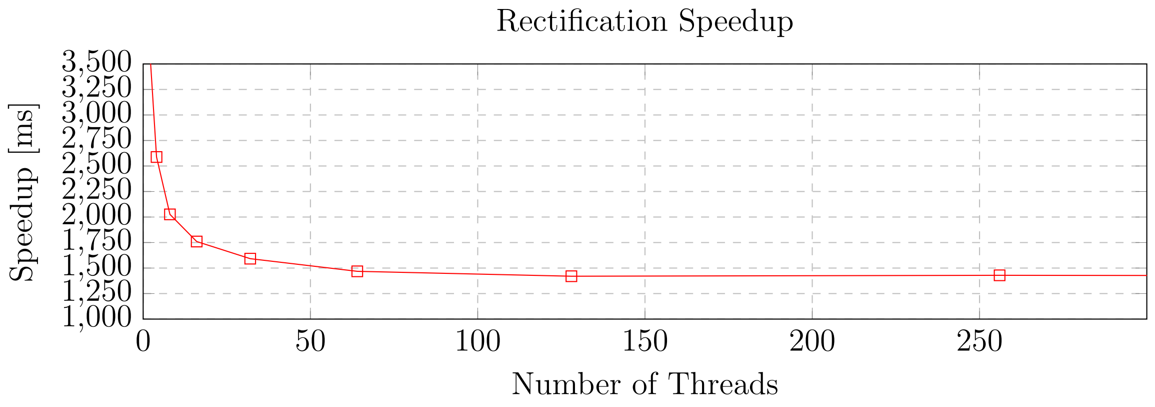
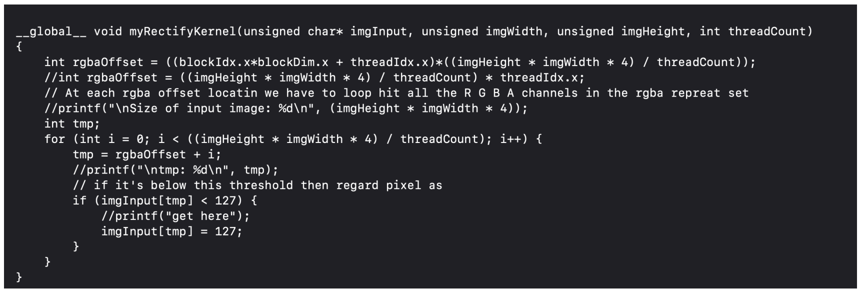


Figure 8: Speedup for Rectification



```
__global__ void myRectifyKernel(unsigned char* imgInput, unsigned imgWidth, unsigned imgHeight, int threadCount)
{
    int rgbaOffset = ((blockIdx.x*blockDim.x + threadIdx.x)*((imgHeight * imgWidth * 4) / threadCount));
    //int rgbaOffset = ((imgHeight * imgWidth * 4) / threadCount) * threadIdx.x;
    // At each rgba offset locatin we have to loop hit all the R G B A channels in the rgba repreat set
    //printf("\nSize of input image: %d\n", (imgHeight * imgWidth * 4));
    int tmp;
    for (int i = 0; i < ((imgHeight * imgWidth * 4) / threadCount); i++) {
        tmp = rgbaOffset + i;
        //printf("\ntmp: %d\n", tmp);
        // if it's below this threshold then regard pixel as
        if (imgInput[tmp] < 127) {
            //printf("get here");
            imgInput[tmp] = 127;
        }
    }
}
```

Figure 1: The Rectification Algorithm

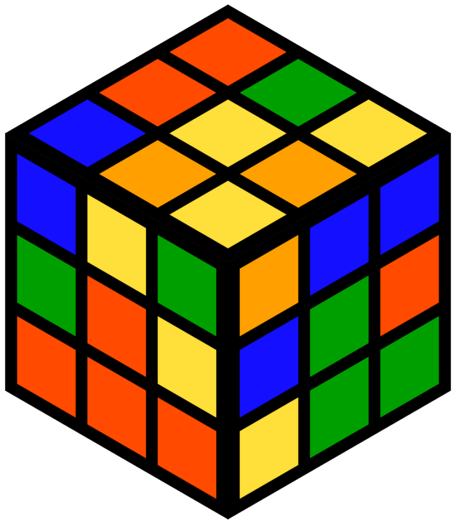


Figure 2: Input Image

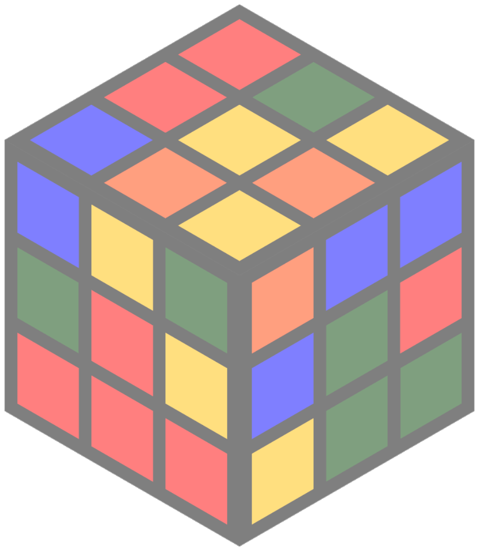


Figure 3: Output Image

2 Pooling

Just like the rectification process, running the pooling process in parallel with more than one thread significantly sped up the sequence. The image was divided in groups of 2 by 2 pixels as described in the specifications. The challenge was determining the new offset in order to traverse the image's pixels in this 2 by 2 pixel fashion. The initial offset was determined as follows: int rgbaOffset = (blockIdx.x * blockDim.x + threadIdx.x) * 4. We then iterate over all the groups of 2 by 2 pixels with the following loop: for (int i = rgbaOffset; i < (imgWidth * imgHeight); i = i + (threadCount * 4)). Then, as the pooling process describes, we needed to find the the max value for the RGBA elements between every pixel in the groups of 4. Simple iterations with a variable holding the current maximum value allowed us to achieve this. Once we had the maximum value for the four RGBA elements we were able to build our new array of 4 pixels. As mentioned in the previous section, the threads within a block in a GPU do not get generated in order. So, just as in the rectification process, the program will not process all these 2 by 2 blocks of pixels in a linear sequence. If we were able to see the image getting pooled as it happened, iteration per iteration, the image would get processed in a random order of blocks across it. We only see the finished product of this sequence. Additionally, just like the rectification algorithm, the GPU's computation time plateaus around 128 threads. Finally, the pooling algorithm fails on one thread when executed within the GPU because the processing time is longer than the given quanta for the thread to execute.

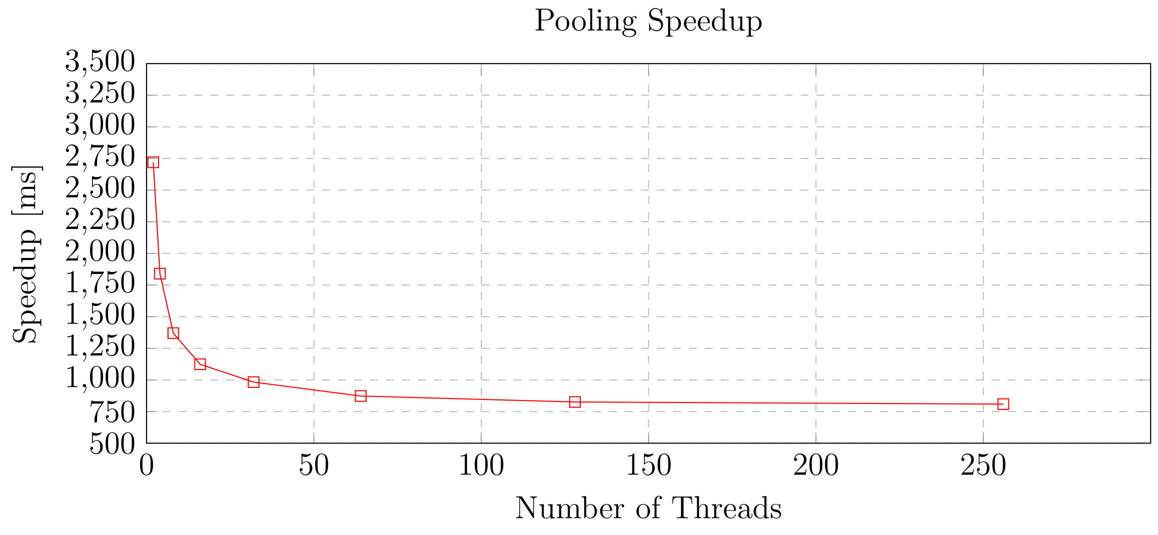
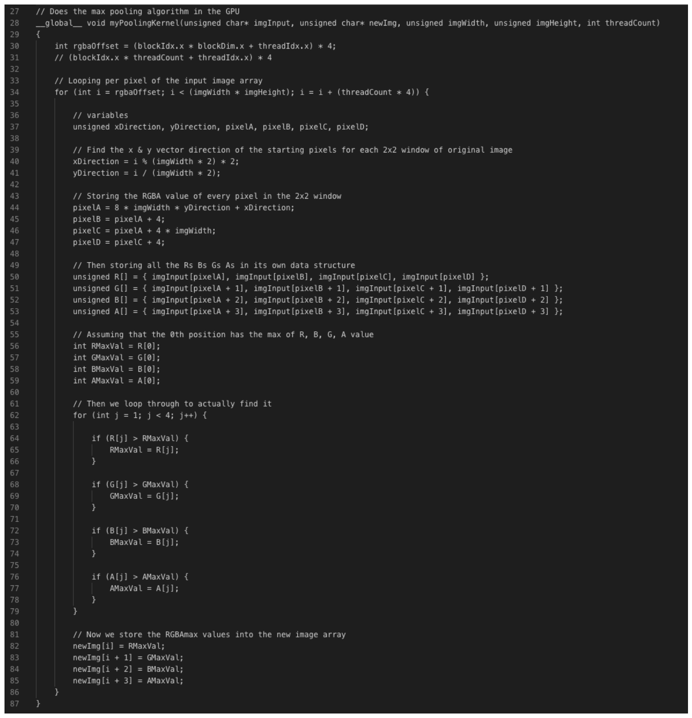


Figure 7: Speedup for Pooling



```
// Does the max pooling algorithm in the GPU
__global__ void myPoolingKernel(unsigned char* imgInput, unsigned char* newImg, unsigned imgWidth, unsigned imgHeight, int threadCount)
{
    int rgbaOffset = (blockIdx.x * blockDim.x + threadIdx.x) * 4;
    // (blockIdx.x * threadCount + threadIdx.x) * 4

    // Looping per pixel of the input image array
    for (int i = rgbaOffset; i < (imgWidth * imgHeight); i = i + (threadCount * 4)) {

        // variables
        unsigned xDirection, yDirection, pixelA, pixelB, pixelC, pixelD;

        // Find the x & y vector direction of the starting pixels for each 2x2 window of original image
        xDirection = i % (imgWidth * 2) * 2;
        yDirection = i / (imgWidth * 2);

        // Storing the RGBA value of every pixel in the 2x2 window
        pixelA = 8 * imgWidth * yDirection + xDirection;
        pixelB = pixelA + 4;
        pixelC = pixelA + 4 * imgWidth;
        pixelD = pixelC + 4;

        // Then storing all the Rs Bs Gs As in its own data structure
        unsigned R[] = { imgInput[pixelA], imgInput[pixelB], imgInput[pixelC], imgInput[pixelD] };
        unsigned G[] = { imgInput[pixelA + 1], imgInput[pixelB + 1], imgInput[pixelC + 1], imgInput[pixelD + 1] };
        unsigned B[] = { imgInput[pixelA + 2], imgInput[pixelB + 2], imgInput[pixelC + 2], imgInput[pixelD + 2] };
        unsigned A[] = { imgInput[pixelA + 3], imgInput[pixelB + 3], imgInput[pixelC + 3], imgInput[pixelD + 3] };

        // Assuming that the 0th position has the max of R, B, G, A value
        int RMaxVal = R[0];
        int GMaxVal = G[0];
        int BMaxVal = B[0];
        int AMaxVal = A[0];

        // Then we loop through to actually find it
        for (int j = 1; j < 4; j++) {

            if (R[j] > RMaxVal) {
                RMaxVal = R[j];
            }

            if (G[j] > GMaxVal) {
                GMaxVal = G[j];
            }

            if (B[j] > BMaxVal) {
                BMaxVal = B[j];
            }

            if (A[j] > AMaxVal) {
                AMaxVal = A[j];
            }
        }

        // Now we store the RGBAmax values into the new image array
        newImg[i] = RMaxVal;
        newImg[i + 1] = GMaxVal;
        newImg[i + 2] = BMaxVal;
        newImg[i + 3] = AMaxVal;
    }
}
```

Figure 4: The Pooling Algorithm

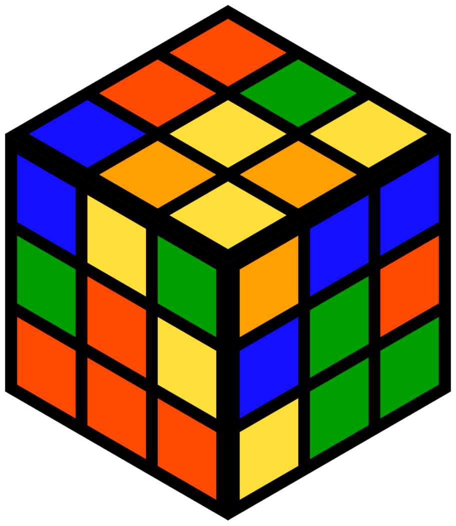


Figure 5: Input Image

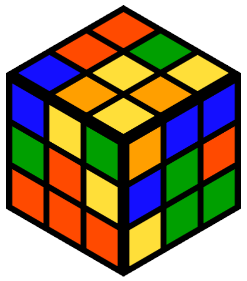


Figure 6: Output Image