Introduction

In this lab we rectified (brightened) an image and compressed an image using multithreaded programming on the GPU using CUDA.

Image Rectification

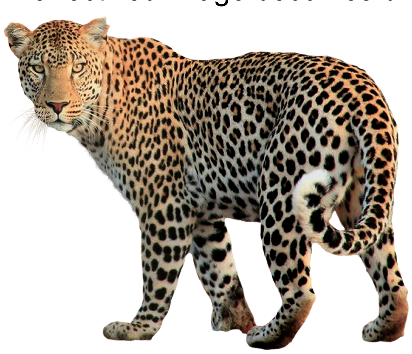
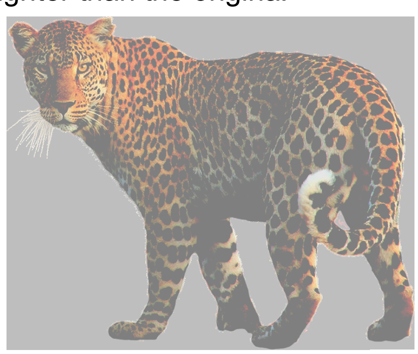
To set up rectification. we created a copy of the image on the GPU and modified it in place. If the corresponding RGBA had a value less than 127, set it to 127.

Every thread runs in a certain block that is defined by the total number of chars in the image array divided by the number of threads. For the offset of each thread, we used threadIdx.x* block (mentioned above).

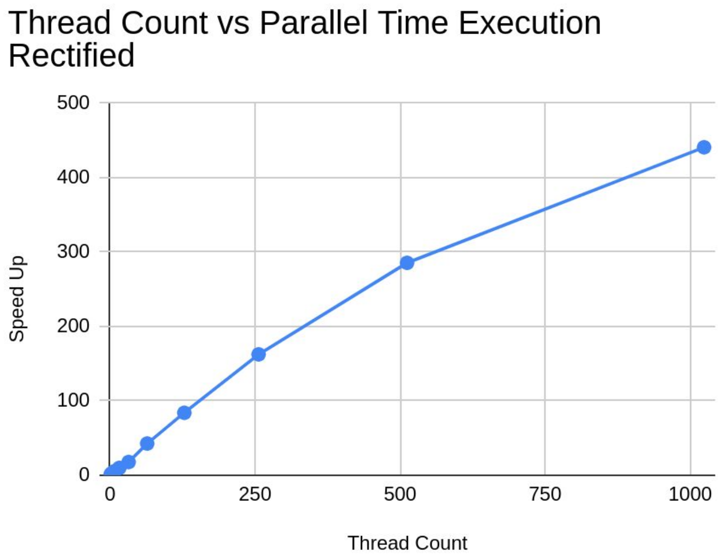
```
// rectify <<< block_count, thread_count >>>
__global__ void rectify(unsigned char* image, unsigned height, unsigned width, int thread_count)
{
    // process image
    int block = (height * width * 4) / thread_count;
    int offset = threadIdx.x * block;
    for (int i = 0; i < block; i++)
    {
        int j = offset + i;
        if (image[j] < 127) image[j] = 127;

    }

}
```

<u>Results</u>

The rectified image becomes brighter than the original

Before Image *After Image*



This graph displays a logarithmic growth in processing speed as the number of threads increase.

Speed up was calculated the following way:
Speed Up = Time for 1-Thread / Time for K-Threads

In both cases (rectification and max pooling) the execution time of the parralizable part of the code was measured with different thread number, the speed up was then computed and plotted.

Image Pooling

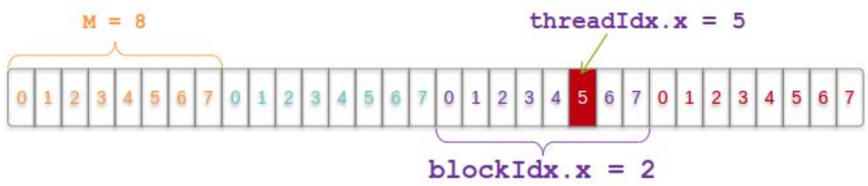
For image pooling, we had 2 cases: 1 thread pooling and k-threads pooling.

1) 1 Thread Pooling:
   The 1 thread pooling is done on the CPU since the computer in the lab did not permit it to run due to how slow it was.

```
void pool_1_thread(unsigned char* image, unsigned char* new_image, unsigned height, unsigned width, int thread_count)
{
    // process image
    int n = 0;

    for (int i = 0; i < width * height* 4;  i +=8)
    {
        if (i%width * height * 4 == 0 ) { i += width * 4; }
            for (int k = 0; k < 4;k++)
            {
                // k = color
                // i = offset in image array
                // the rest is for correct pixel in 2X2 block
                int a = image[k + i];
                int b = image[k + i + 4];
                int c = image[k + i + width*4];
                int d = image[k + i + width * 4  + 4];

                int max = a > b ? a : b;
                max = c > max ? c : max;
                max = d > max ? d : max;
                new_image[n++] = max;
            }
    }
}
```

   The process was simply iterate over the whole image jumping 2 pixels (8 slots) at a time. Then for each RGBA, find the highest and store it in a new smaller image at the next available index (n).
   To find the appropriate pixels forming a 2X2 block, relative to the top left pixel, we find remaining 3 pixels:
   - Top Right index = Top Left + 4
   - Bottom Left index = Top Left + Image Row Size (width)
   - Bottom Right index = Top Left + Image Row Size (width) + 4

2) K-Thread Pooling:
   The process is very similar but we have a thread offset that is calculated as follows:
   We took care of of splitting things between blocks by setting the indexes as seen in the tutorial .




```
int index =  threadIdx.x  + blockIdx.x * M;
```

Our offset = (blockIdx.x * blockDim.x + threadIdx.x)*4 since we iterate over whole pixels each having 4 elements inside.

```
int offset = (blockIdx.x * blockDim.x + threadIdx.x)*4;
```

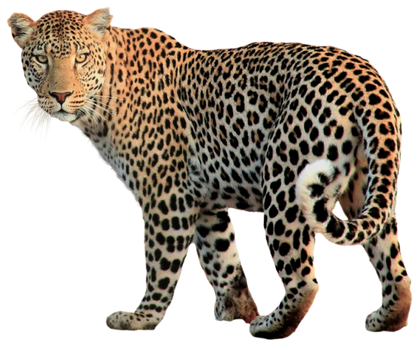
At every pixel, we calculate the horizontal and vertical direction where the other 3 pixels that for a 2X2 block are located.

After getting every necessary pixel, a list of R.G.B.A are created we find the highest value and save it in the new image.

```
for (int i = offset; i < (width*height); i+=(thread_count*4) )
{
    unsigned p1 = 8 * width *  i / (width * 2) + i % (width * 2) * 2;
    unsigned p2 = 8 * width *  i / (width * 2) + i % (width * 2) * 2 + 4;
    unsigned p3 = 8 * width *  i / (width * 2) + i % (width * 2) * 2 + 4 *  width;
    unsigned p4 = 8 * width *  i / (width * 2) + i % (width * 2) * 2 + 4 * width + 4;

    //setup initial values
    unsigned r[] = { image[p1],   image[p2],   image[p3],   image[p4] };
    unsigned g[] = { image[p1+1], image[p2+1], image[p3+1], image[p4+1] };
    unsigned b[] = { image[p1+2], image[p2+2], image[p3+2], image[p4+2] };
    unsigned a[] = { image[p1+3], image[p2+3], image[p3+3], image[p4+3] };

    unsigned rMax = r[0];
    unsigned gMax = g[0];
    unsigned bMax = b[0];
    unsigned aMax = a[0];

    for (int j = 1; j < 4; j++ )
    {
        if (r[j] > rMax) rMax = r[j];
        if (g[j] > gMax) gMax = g[j];
        if (b[j] > bMax) bMax = b[j];
        if (a[j] > aMax) aMax = a[j];
    }
    new_image[i] = rMax;
    new_image[i+1] = gMax;
    new_image[i+2] = bMax;
    new_image[i+3] = aMax;


}
```

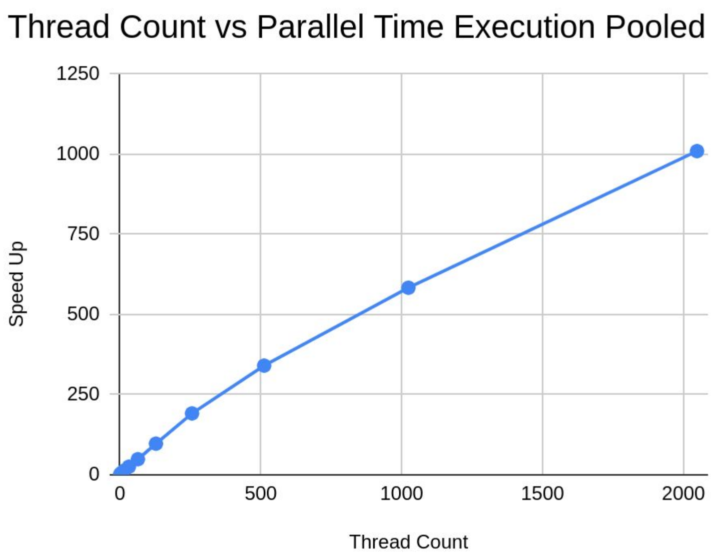
Results

The pooled image becomes 4 times smaller than the original because it is compressed following max pooling over 2x2 blocks.



Before Image

After Image



This graph displays a logarithmic growth in processing speed.

Conclusion

By using more and more threads we saw that the speed up grows following a logarithmic curve which is expected since while it could be theoretically be linear the more threads you have the more overhead there is which limit the speed up.