# Lab 0 Report – Simple CUDA Processing
# ECSE 420 – Parallel Computing – Fall 2023

Group 14
Samantha Handal - 260983914
Rebecca Mizrahi - 260975001

# Architecture

Google Colab offers a free T4 GPU for computational tasks, which is part of NVIDIA's Turing family. Characteristics of the T4 that are relevant to lab 0:

- **CUDA Cores:** T4 has 16 Streaming Multiprocessors (SMs) and a total of 2560 CUDA cores. This means it's well-suited to handle thousands of threads in parallel, displaying benefits from increasing the number of threads in grid configuration.

- **Memory Bandwidth:** With a memory bandwidth of up to 320 GB/s, data transfer between the global memory and the CUDA cores is rapid, minimizing potential bottlenecks that might arise when fetching pixel data.

# Image Rectification

## Part 1

**Objective**

The objective of part 1 of the lab is to perform a rectification process wherein each pixel of the image is set to 0 if the pixel's value is less than 0. The pixel's value represents the RGBA integer values which have the range [-127, 127). Thus, all pixel values are non-negative, giving the results shown in figure 2.

**Discussion**

The part 1 code leverages CUDA to perform rectification in a parallel scheme, taking advantage of the GPU's power to accelerate this process. To do this, memory is allocated on the GPU and the image data is copied from the host to the GPU. Then, the kernel is ready to be launched. It is launched with a specified number of threads and blocks, where the number of blocks and threads per block is calculated to ensure that all pixels are processed. The logic for the number of blocks calculated from the specified number of threads is that if there are 256 pixels and 256 threads, only 1 block is needed, whereas if there is 1 thread, 256 blocks are needed. The equation to calculate the number of blocks also accounts for the case if the number of bytes to be processed is not a multiple of the number of threads:

```
int num_blocks = (width * height * 4 + num_threads - 1) / num_threads;
```

Once the device is synchronized and all threads have completed, the data is transferred back to the host. Figures 1 and 2 show an example of the rectification process.


Figures 1 and 2. Images before and after the rectification process.

This rectification process was tested with different thread numbers - 1 through 256, doubling the number of threads each time - to measure time taken. This was tested three times on differently-sized images, as seen in figures 3 through 5, but they all reflect the same pattern. With an increased number of threads, there is effectively an increased level of parallelism. Each thread can perform its computations independently of the others, which means multiple operations can occur simultaneously. By utilizing GPU cores efficiently through a higher number of threads, the workload of the program is spread out, and a substantial speedup is achieved. This relationship isn't perfectly linear, though, since there exists an ideal number of threads to use. This is why speedup is relatively insignificant between 128 and 256 threads, and even halts for the smaller image; we can assume that the ideal number for the small image is between 128 and 256, and for the larger images the ideal number is near 256.

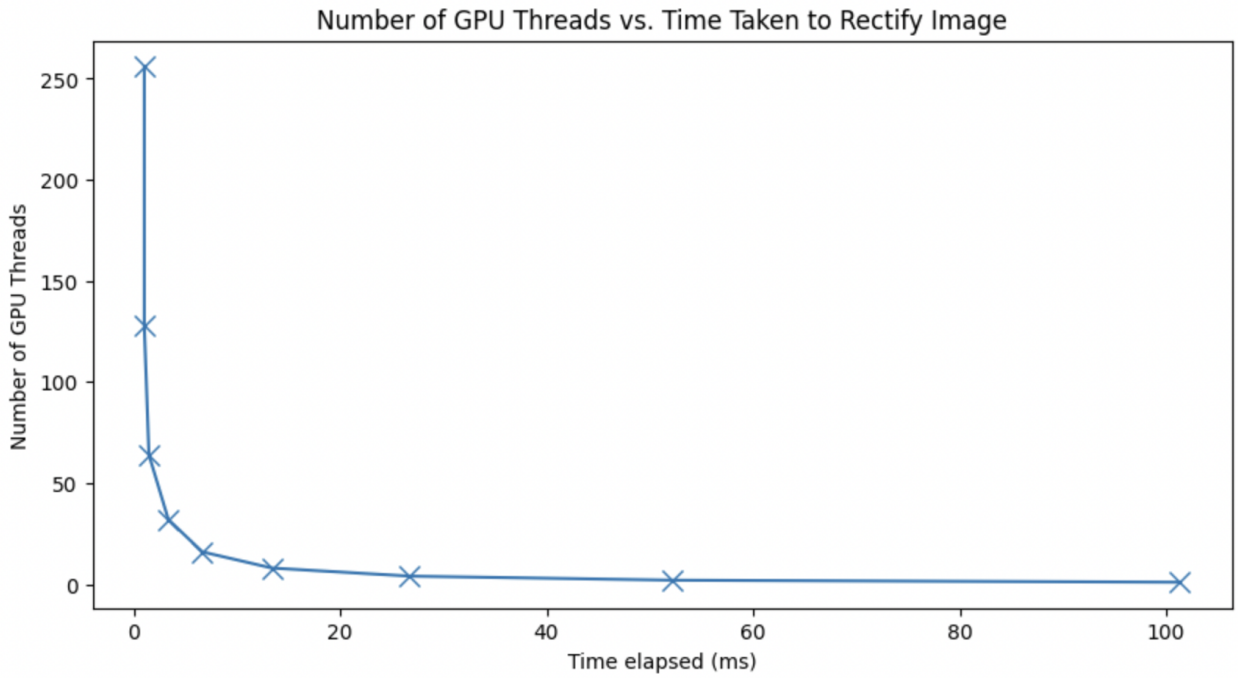


Figure 3. Time vs. thread count for the rectification process of a large (3840×2400) image.


Figure 4. Time vs. thread count for the rectification process of a medium (2217×1663) image.

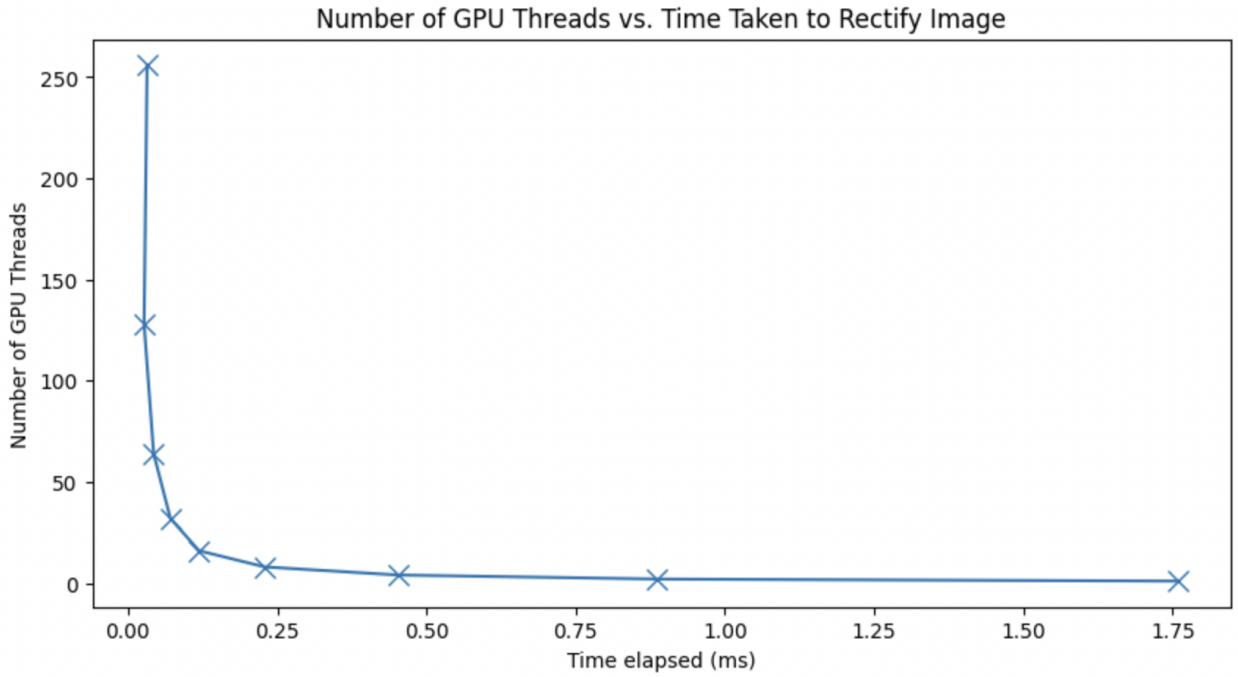



Figure 5. Time vs. thread count for the rectification process of a small (400 x 400) image.

# Pooling

## Parallelization Scheme

A 1-dimensional grid and block structure was used for max-pooling. Instead of having each thread process a single pixel, the design implemented has each thread process a 2x2 block. The number of total threads needed is ***width/2 * height/2*** (halving the dimensions of the image through pooling).

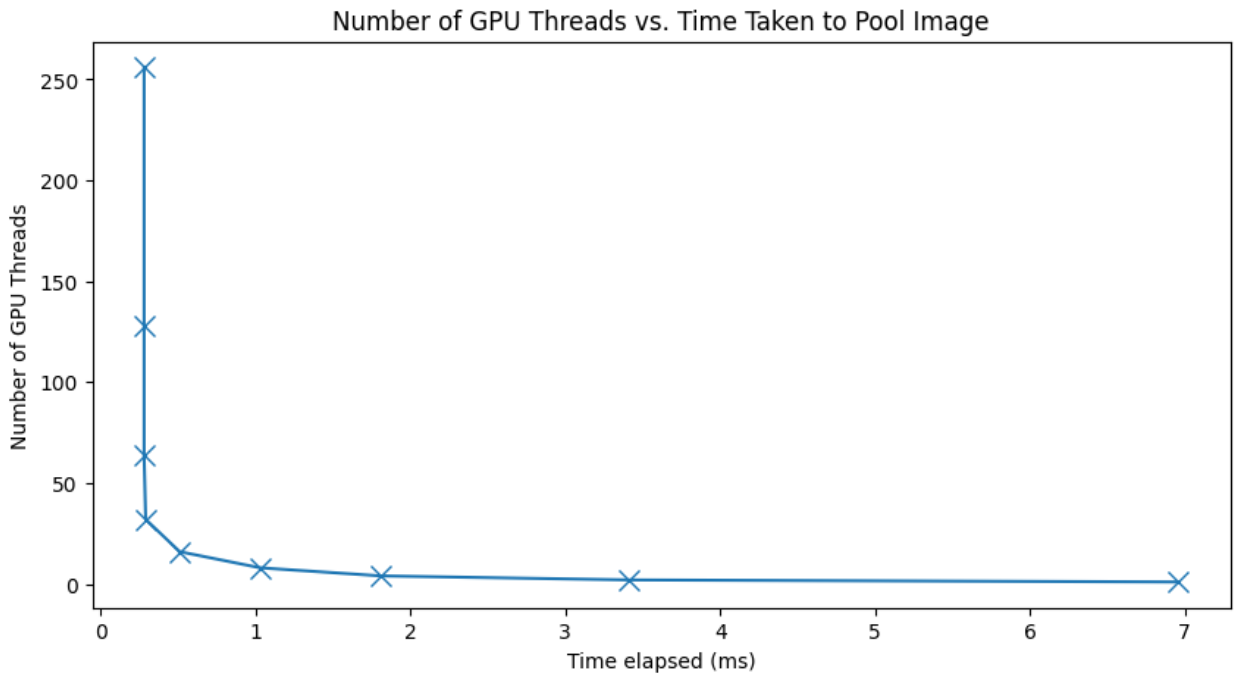
Input images are considered to have 4 channels - Red, Green, Blue, and Alpha (for transparency). When referring to the value of a pixel, it's not just one number; it's four numbers. The loop ***for(int channel = 0; channel < 4; channel++)*** in the kernel ensures that the max-pooling operation is performed for each of the four channels separately. This ensures that the final pooled image retains its colors and transparency correctly.

The x and y coordinates of the current pixel is determined based on the block and thread ID, the global unique ID for any thread in the grid can be calculated as: ***global ID = blockIdx.x * blockDim.x + threadIdx.x***. The 2D position (x, y) in the input image is derived from global ID:

- ***global ID % (width/2)***: The modulo operation gives the x-coordinate within the output image (which is half the size of the input).
- ***global ID / (width/2)***: The integer division gives the y-coordinate within the output image.
- ***Multiplying by 2***: Since each thread processes a 2x2 patch of the input, the x and y coordinates for the input image are multiplied by 2.

## Speedup Analysis

The speedup is determined by comparing the runtime of the parallelized version with a given number of threads to the baseline runtime with 1 thread. Below is a graph plotting the speedup as the number of threads doubles (1,2,4,8,16,32,64,128,256) for the given test image (Test_1.png).



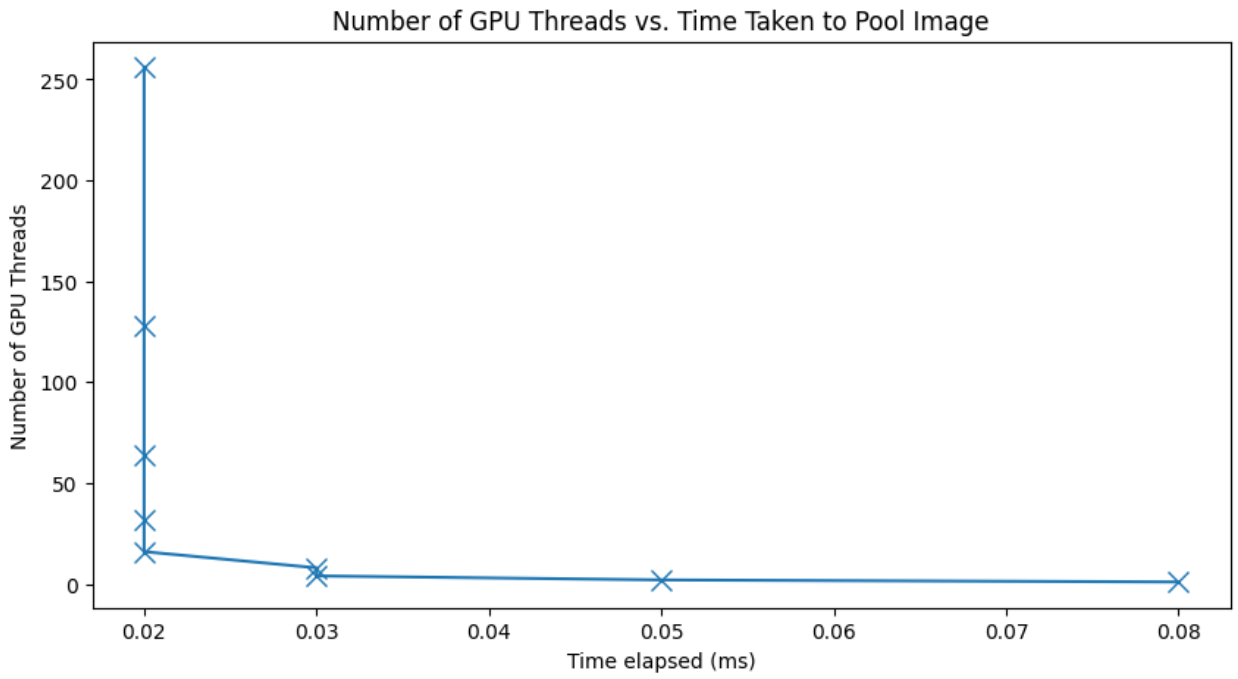

Observations:

- **Initial Gains:** The runtime decreases significantly as the number of threads increases initially, showing the benefits of parallel processing and the processing capability of the T4 GPU. Especially in the lower range (1-32 threads), the speedup is more pronounced.
- **Diminishing Returns:** Beyond 32 threads, the reduction in runtime starts to plateau. Even though the T4 has thousands of CUDA cores, the actual number of threads that can run in parallel on each SM is determined by several factors including the number of registers used by the kernel, shared memory, etc. After a certain point, simply increasing the number of threads doesn't equate to linear speedup.
- **Consistent Output:** Regardless of the thread count, the processed images remain consistent with the expected output, showcasing the correctness of the implementation.

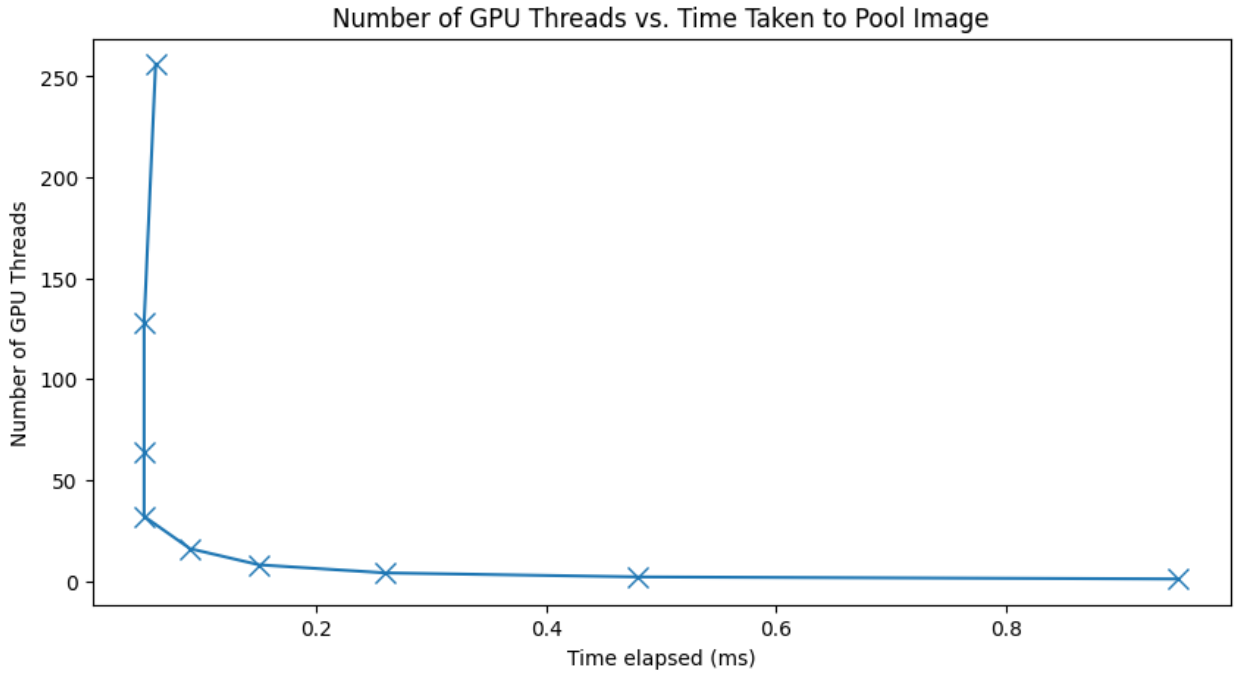
## Image Discussion

Pooling was tested on three identical images with varying sizes (Appendix B). For each size, computational time for thread count 1,2,4,8,16,32,64,128,256 was recorded and plotted in the graphs below

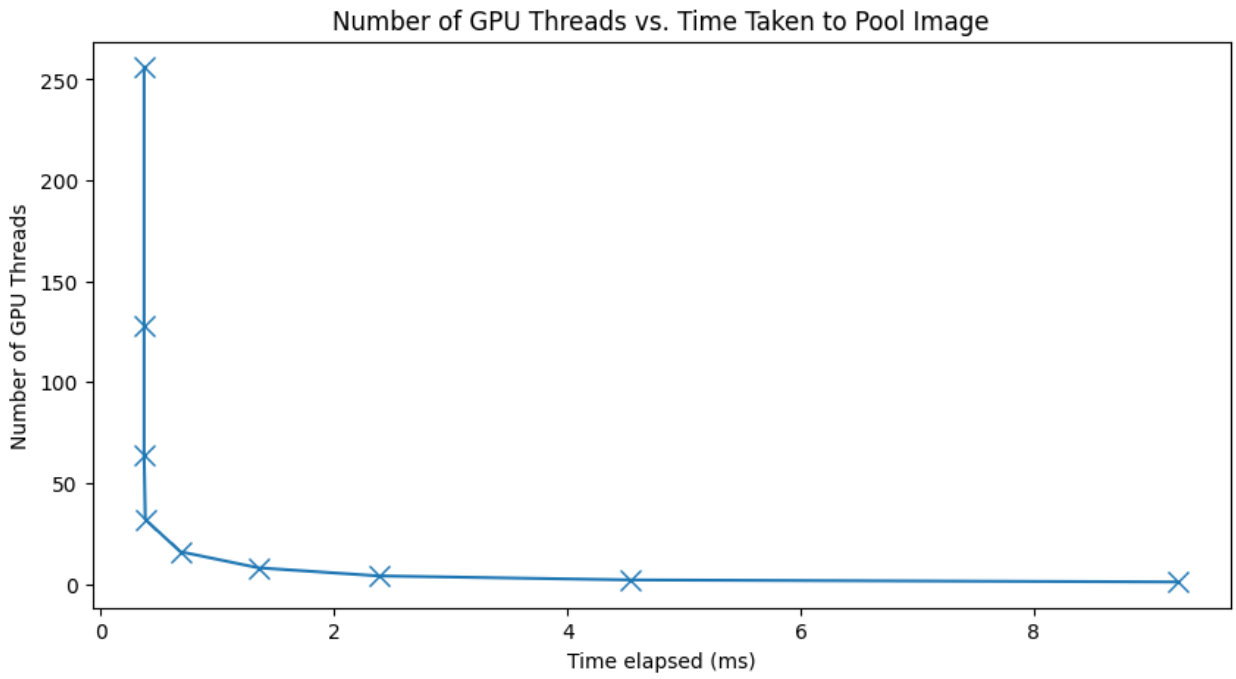
1) Small (320×240)




2) Medium (1280×960)




3) Large (4032×3024)




Observations:

- **Performance Scaling with Image Size:**

  As the image size grows, the computational time also increases. However, the speedup trends due to parallelism remain evident. For instance, the runtime reduction when doubling the thread count (e.g., from 1 to 2 or 4 to 8) is pronounced for all sizes.

- **Thread Efficiency vs. Image Size:**

  *Small image (320 x 240):* The performance gain is consistent until 64 threads, after which the improvements plateau and marginally increase when moving from 64 to 128 threads. However, at 256 threads, the runtime slightly decreases again. This suggests that for smaller images, the overhead of handling more threads starts to become noticeable around 64-128 threads.

  *Medium image (1280 x 960):* The trend of diminishing returns starts earlier, around 32 threads. Moving from 32 to 64 threads shows a minimal increase in runtime, and this pattern continues to 256 threads. This might suggest that for this size, the GPU's processing units are efficiently saturated around 32 threads.

  *Large image (4032 x 3024):* The decrease in runtime is notable up to 16 threads. Beyond that, the benefits start to diminish, and there's a plateau between 32 to 256 threads.

From these observations we can conclude that the optimal thread count is highly dependent on the image size. As data size increases, the GPU can utilize more threads effectively, but there's a point of diminishing returns, beyond which increasing thread count provides minimal to no benefits.

## Conclusion

The two sections of this lab provided several valuable insights into parallel computing and image processing with CUDA. Firstly, this lab demonstrates the power of parallel processing by utilizing the GPU. Leveraging the GPU's CUDA cores significantly speeds up image processing tasks compared to sequential CPU-based approaches. Secondly, it showed the importance of optimizing threads and blocks in a kernel in order to achieve the best performance. This optimal number varies with size of the image and other factors, but can be found by testing multiple numbers, as we did in this lab. To conclude, this lab showcased important aspects about the power of the GPU and efficiency of parallel processing that can be used in our future coding endeavors.

## Appendix A. Image Rectification

Figure 1A. Medium-sized picture used for testing the rectification process, before and after.

Figure 1B. Small-sized picture used for testing the rectification process, before and after.

## Appendix B. Pooling

Figure 1.B: Original small test image (320 x 240)

Figure 2.B: Pooled small test image (320 x 240)

Figure 3.B: Original medium test image (1280×960)

Figure 4.B: Pooled medium test image (1280×960)

Figure 5.B: Original large test image (4032×3024)

Figure 6.B: Pooled large test image (4032×3024)