GCA Session 3 - Lab Report

Maxime Rombaut
Master of Electronics and ICT Engineering Technology

Abstract: In this report, we present our work on developing a CUDA-based application that takes an inputimage and converts its colors on a few different ways. We then make a comparison between the different ways.

Keywords: CUDA, GPU, CPU, C++

1 No striding

The first implementation takes the input image and inverts the colors. This happens via the GPU. The inversion happens via the following formula: $image\_array[idx] = 255 - image\_array[idx];$. With $idx = blockIdx.x * blockDim.x + threadIdx.x;$. We ran this kernel with different block sizes and different amount of threads. With the amount of threads a power of 2 and between 1 and 1024. The amount of blocks is equal to $TOTAL/threadsPerBlock$, with TOTAL being the total amount of pixels in the image. The problem with this implementation is that the amount of warps is very big. With the amount of warps being: $ceilf((float)threads/32) * blocksPerGrid$. To reduce this we have implemented striding.

2 Striding

In this implementation we use only 1 block and let the threads in this block do multiple calculations. To calculate the stride we do: $ints = (int)ceil(((float)(TOTAL)/threads))$. This value is used to make every thread do the correct amount of calculations on the correct pixel. We get the correct pixel via: $image\_array[idx + threads * i]$ with i in the range of 0 to s. This implementation uses a lot less warps than the previous one. A comparison can be found in the table underneath.

# Threads	No striding	Striding
1	1987392	7763
2	1987392	7763
4	1987392	7763
8	1987392	7763
16	1987392	7763
32	1987392	7763
64	3974784	15526
128	7949568	31052
256	15899136	62104
512	31798272	124208
1024	63596544	248416

Table 1: *comparison between no striding and striding*

3 Divergence

In this implementation we create a divergence. Not all threads in the same warp will do the same sort of calculation. Every three array elements, we get the value for the red color of a pixel. We give this the value calculated by the following formula: $image\_array[idx + threads * i] = (image\_array[idx + threads * i]\%25) * 10;$. The other elements, green and blue pixel values, will get the same value as in previous methods. Notice the difference in complexity: the red value is calculated via a % and $*$ and the others only use subtraction. The subtraction operation is less complex than the others so it goes faster. The threads that execute the subtractions will thus have to wait on those that to the other operations. In the next implementation we try to avoid this.

4 Reformatted input

This implementation tries to avoid the thread divergence by reformatting the input. The original input array has the following structure: RGBRGB...RGB. In the reformatted input the RGB values are sorted per color so it looks like RRRR..GGGG...BBBB. We then execute the kernel that works the same as the previous one, but check that when the calculated index is in the first 1/3 of the elements, the formula for the red values is applied.

5 Comparison

All the implementations have been tested multiple times and the average of those times is displayed in the graph underneath. Before executing and timing the kernels, they were executed a single time because it sometimes takes a bit more time to call the kernel for the first time. The tests were ran with a different amount of threads and always with 1 block.

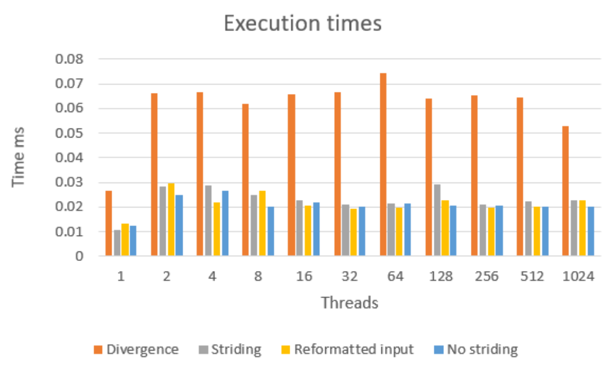

Figure 1: *execution times of the different implementations*

In the graph we can see that the obvious outlier is the one where divergence is present. Those times are always double those without it. The other implementations lie rather close together. Another thing to notice is that the times over the different amount of threads are relatively close to each other, except when there is only one thread. To put these times into perspective we have also made an implementation that only uses the CPU. We have ran this implementation 5 times and came to the average of $22.64498ms$. We can see that even the slowest GPU implementation is much faster than the CPU implementation.

6 Github link

`https://github.com/MaximeRombaut1412/GCA_labs.git`