GPU Assignment: Image processing

Stefano Sandonà

Vrije Universiteit Amsterdam, Holland

1 GPUs: NVIDIA GTX480

The aim of this assignment was to learn how to use many-core accelerators, GPUs in this particular case, to parallelize data-intensive code. All the implementations were written for the **NVIDIA GTX480**, using CUDA, a parallel computing platform and programming model invented by NVIDIA. Programming with CUDA, there is a straightforward mapping onto hardware, for this reason it is necessary to study the available HW before start developing an application. The architecture of the given accelerator is is shown in Figure 1, its main characteristics and limits are shown in Table 1.

Figure 1: NVIDIA GTX480 Architecture

2 CImg

The image processing library used in this project was CImg, a small, modern and open-source toolkit developed for C++. CImg implements the RGB color model, an additive color model in which red, green, and blue light are added together in various ways to reproduce a broad array of colors. Each colored image of size *N*M* is composed by three parts (R,G,B) of the same size, so that *N*M*3* values are necessary to define an image. The Figure 2 shows an example of image composition.

Microarchitecture	Fermi
Compute capability (version)	2.0
Maximum dimensionality of grid of thread blocks	3
Maximum x-dimension of a grid of thread blocks	65535
Maximum y-, or z-dimension of a grid of thread blocks	65535
Maximum dimensionality of thread block	3
Maximum x- or y-dimension of a block	1024
Maximum number of threads per block	1024
Cores per SM (warp size)	32
SM	15
Cores	480 (32 * 15)
Maximum number of resident blocks per multiprocessor	8
Maximum number of resident warps per multiprocessor	48
Maximum number of resident threads per multiprocessor	1536 (48 * 32)
Number of 32-bit registers per multiprocessor	32K
Maximum amount of shared memory per multiprocessor	48K

Table 1: NVIDIA GTX480 Specifications

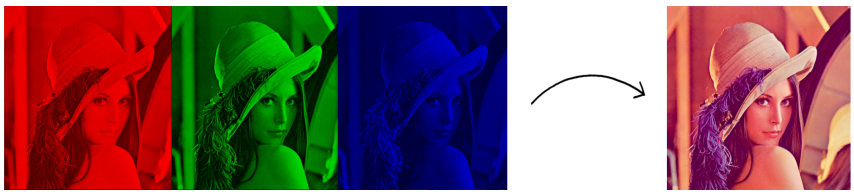


Figure 2: RGB model

3 The processing flow

Using CUDA there are two parts of the code: the device code, or GPU code, or the Kernel, that is a sequential program, write for one thread and execute for all and the HOST code, or CPU code, that is used to instantiate the grid, run the kernel, manage the memory. Figure 3 shows the processing flow of a CUDA application. In the particular case of image processig, everything starts from the CPU, that store the image from a file into a local buffer, allocates IN and OUT buffers on the GPU (*cudaMalloc*) and copy the image into the GPU's IN buffer (*cudaMemCpy*). After that, the CPU launches the GPU kernel with a defined grid configuration (*kernel_function «gridDim, blockDim»(params)*), that is executed by the GPU following the SIMT (Single Instruction, Multiple Threads) NVIDIA model. The threads are executed in parallel in each core, and they read the assigned part of IN data and generates the assigned part of OUT data. At the end, the results are copied out back to the CPU (*cudaMemCpy*) and the image is written to a file by the CPU.

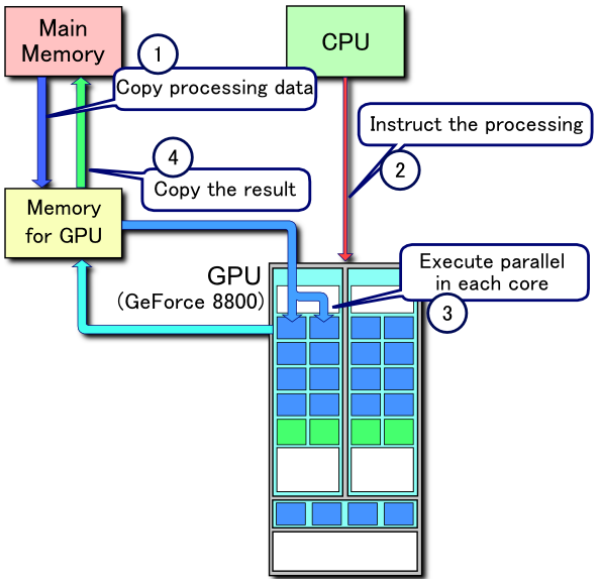
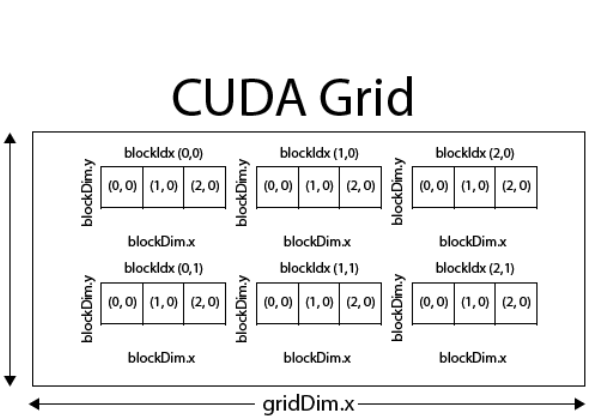


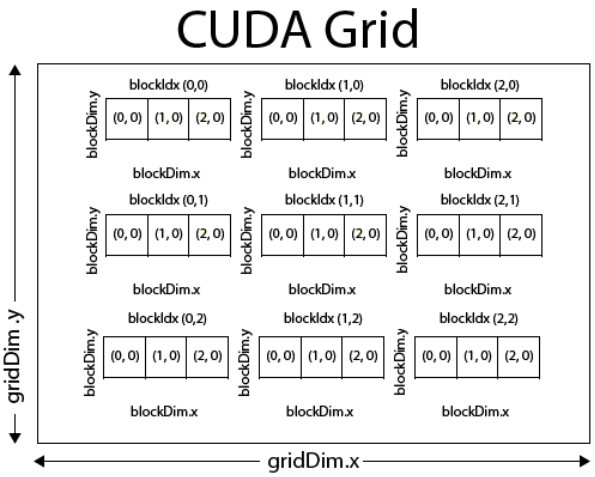
Figure 3: CUDA processing flow

4 CUDA grid configuration

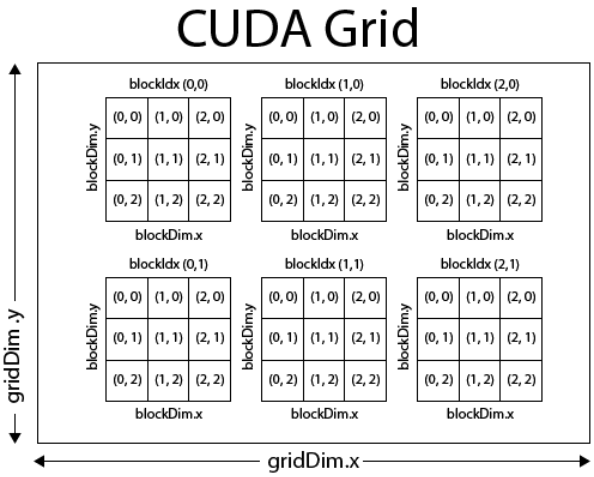
In CUDA, as mentioned before, there is a strinct mapping with the hardware, so that an hardware virtualization model is fixed with the concepts of thread, block and grid. Each **thread** executes the kernel code, running on one CUDA core. The threads are logically grouped into **thread blocks**, so that the threads of the same block will run on the same multiprocessor. The thread blocks are logically organized in a **Grid**, that represent the entire dataset. The blocks and the grid can be of 1D, 2D or 3D. An image is a 2D structure and for this reason the most convenient choice is to set up also a 2D grid. However, setting up a 2D grid, there is not only one way to follow. For this particular project, 4 possible combinations were considered. The first was a non square grid of 1D blocks (Figure 9a), the second a square grid with 1D blocks (Figure 9b), the third a non square grid of 2D blocks (Figure 4c) and the third a square grid of 2D blocks (Figure 4d). Different configurations were tested in order to find the best to suit the particular case.



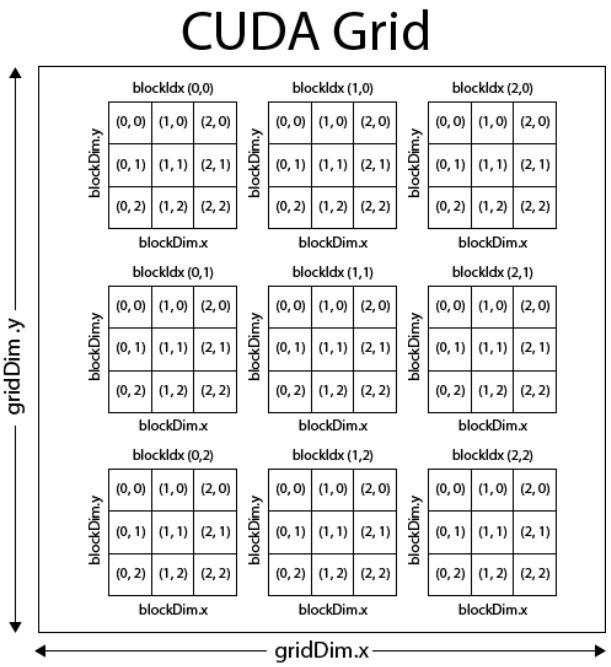
(a) No square grid with 1D blocks



(b) Square grid with 1D blocks



(c) No square grid with 2D blocks



(d) Square grid with 2D blocks

Figure 4: Possible kernel configurations

5 Coalesced memory access

One of the main bottlenecks with the GPUs is the global memory access that is expensive. CUDA uses a SIMT approach, in which all threads of a warp execute the same instruction. If the kernel is correctly designed, when the threads of a warp ask a value stored in the global memory, instead of having one access per thread, these accesses can be grouped if consecutive threads access consecutive memory addresses. This practise, is very useful to reduce the memory overhead, so that it was adopted in each algorithm.

6 Algorithm 1: Grayscale Conversion and Darkening

From an RGB image, the output of this algorithm is a darker grayscale image. The gray value of a pixel is generated by weighting the three values (`0.3*R, 0.59*G, 0.11*B`) and then summing them together. To darken the obtained grayscale image, the final pixel value is multiplied by a constant (0.6). The Figure 5 shows an example of the result. The sequential algorithm, simply go through the entire image and computes for each pixel the corresponding value (Figure 1).

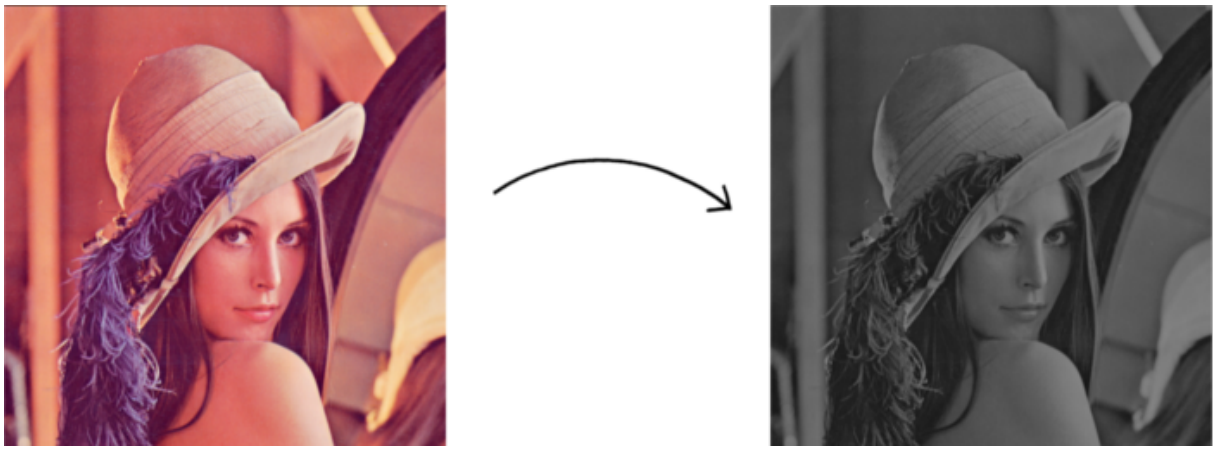


Figure 5: Grayscale Conversion and Darkening

```
for ( int y = 0; y < height; y++ ) {
                for ( int x = 0; x < width; x++ ) {
                        ...
                        float r = static_cast< float >(inputImage[(y * width) + x]);
                        float g = static_cast< float >(inputImage[(width * height) + (y
                        float b = static_cast< float >(inputImage[(2 * width * height) +
                        ...
                        darkGrayImage[(y * width) + x] = static_cast< unsigned char >(gr
```

Listing 1: Sequential code

6.1 Parallelization

6.2 First method

After copying the input image into the GPU global memory and allocating some memory to content the output image, the kernel is ready to be launched. The GPU code is the same as the code content inside the loop of the sequential version (Figure 1), but instead of using the indexes of the loop to access the image pixels, it uses the index associated to the thread. To exploit the coalesced memory access, two consecutive threads computes/accesses the values of two consecutive pixels. Different grid configurations were tested launching this program.

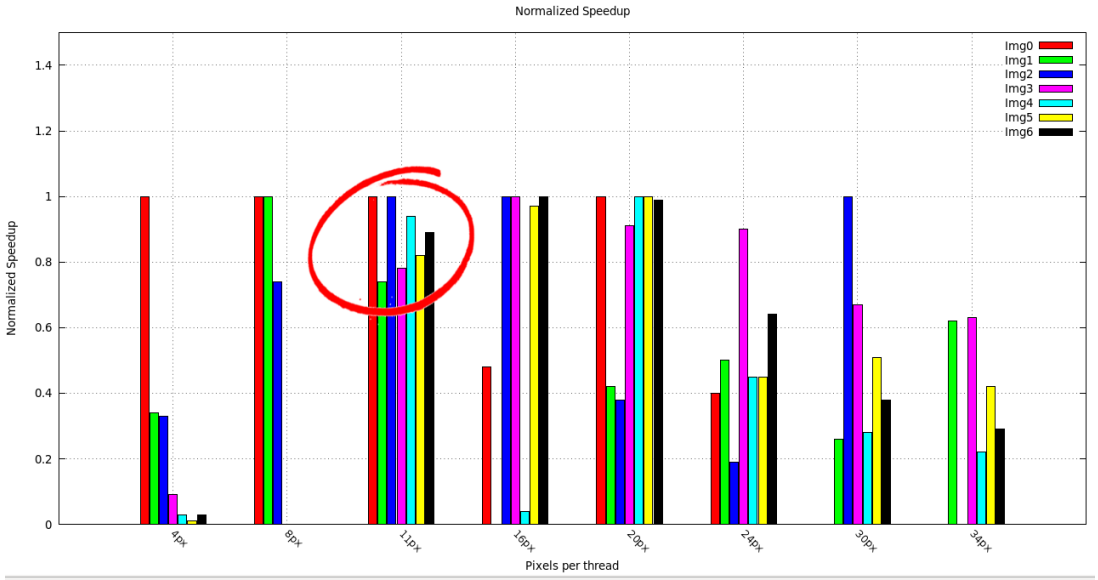


Figure 6: Normalized Speedup

7 Algorithm 2: Histogram Computation

From an RGB image, the output of this algorithm is a grayscale image with the relative histogram of 256 possible values of gray. The histogram measures how often a value of gray is used in an image. The sequential algorithm, simply go through the entire image, computing for each pixel the corresponding gray value and incrementing the corresponding counter. The Figure 7 shows an example of the result.

```
for ( int y = 0; y < height; y++ ) {
                for ( int x = 0; x < width; x++ ) {
                        float grayPix = 0.0f;
                        float r = static_cast< float >(inputImage[(y * width) + x]);
                        float g = static_cast< float >(inputImage[(width * height) + (y
                        float b = static_cast< float >(inputImage[(2 * width * height) +

                        grayPix = ((0.3f * r) + (0.59f * g) + (0.11f * b)) + 0.5f;

                        grayImage[(y * width) + x] = static_cast< unsigned char >(grayPi
                        histogram[static_cast< unsigned int >(grayPix)] += 1;
                }
}
```

Listing 2: Sequential code

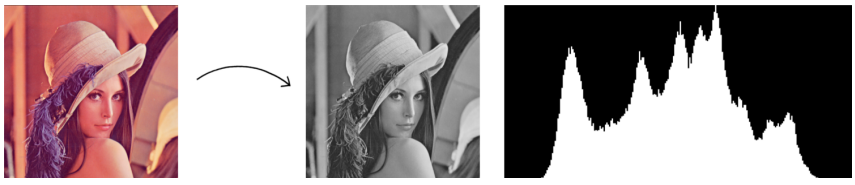


Figure 7: Histogram Computation

7.1 Parallelization

7.2 First method

After copying the input image and the initial empty histogram into the GPU global memory and allocating some memory to content the output image, the kernel is ready to be launched. As for the previous

algorithm, the GPU code is the same as the code content inside the loop of the sequential version (Figure 2), but instead of using the indexes of the loop to access the image pixels, it uses the index associated to the thread. Also in this case, to exploit the coalesced memory access, two consecutive threads computes/accesses the values of two consecutive pixels. The interesting aspect of the histograms, is that it is possible that two threads read at the same time read the same value of gray for a pixel, so that at the same time they try to increment the same corresponding histogram bin. For a correct execution, to avoid wrong updates, the increment has to be an atomic operation, so that only one thread at the time will modify the bin value.

The logic behind an atomic operations is that each thread "locks" the variable, modify it, and "unlocks" it, so that the other threads that try to modify the same variable has to wait that this will be "unlocked". The worst case, in the histogram scenario, is a monocromo image, in which all the threads try to modify the same bin, which implies a long waiting queue of threads. Different grid configurations were tested launching this program.

8 Algorithm 3: Smoothing

Smoothing is the process of removing noise from an image by the means of statistical analysis. To remove the noise, each point is replaced by a weighted average of its neighbours. In this way small-scale structures are removed from the image. In this case a two-dimensional 5-point triangular smooth filter was used. The Figure 8 shows an example of the result.

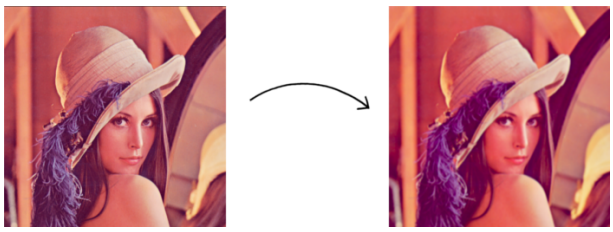
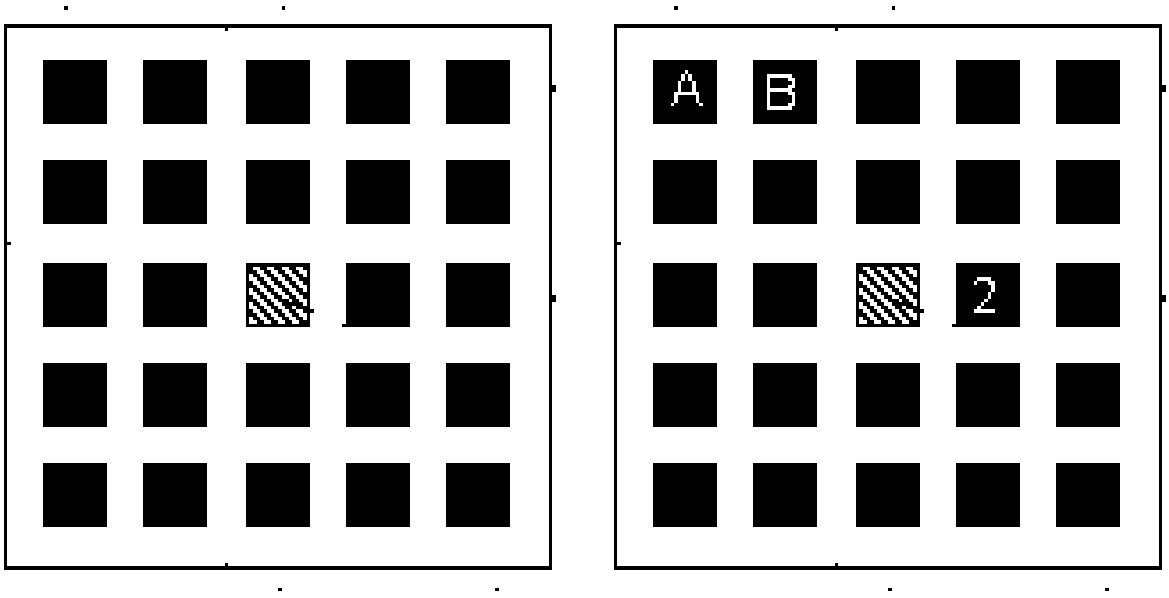


Figure 8: Smoothing

8.1 Parallelization

This particular algorithm deals with square areas of the image (filter), so that using 2D blocks, the threads can efficiently share memory and prevent a lot of global memory accesses.



(a) No square grid with 1D blocks

(b) Square grid with 1D blocks

Figure 9: Possible kernel configurations